Empirical research in economics, finance, and management using R: Essentials, real examples, and troubleshooting

Compiled from session04.tex @ 2023-12-13 16:27:05+01:00.

Day 4: Functions, a.k.a. the pith of R

Andreï V. Kostyrka
27th of September 2023

UNIVERSITÉ DU LUXEMBOURG

Quick recap

We learned:

- How to check conditions and run loops
- How to make changes to data sets
- How to handle data of different types

Today, we harness the full power of functional abstraction and their vectorised application.

Presentation structure

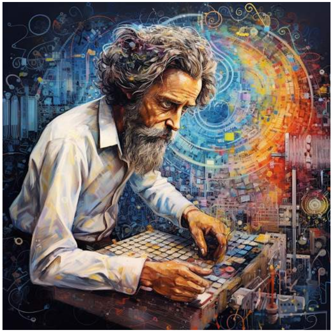
1. How to write functions
2. Methods, namespaces, calls & ellipses
3. Debugging
4. Vectorisation and parallel computing
5. Speeding up, benchmarking, profiling

How to write functions

Functions

The real power of R! Direct productivity gains if you have a piece of code that you use at least twice.

- Very easy to create flexible functions; zero costs
- Can be generalised, wrapped, nested to create super-convenient user-friendly wrappers
- Declare once, use everywhere; publish, upload to a repository, or package them



Creating functions

- Formalise a set of instructions applied to objects given as the input
- Mandatory and optional (with default values) arguments
- May have multi-object return via a list
- Many tutorials on good practices

Pure functions

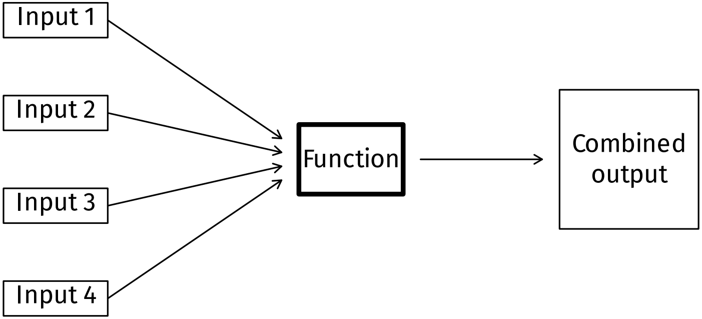
Pure function: a function that will always return the same output for the same input, does not depend on the environment, and does not modify the environment.

In R, functions may rely on global objects, but this is bad practice:

- Non-pure functions are more prone to errors
- Non-pure functions are harder to debug

The crucial function of your project should rely only on inputs and make no assumptions about the existence of objects in other environments.

How to imagine functions



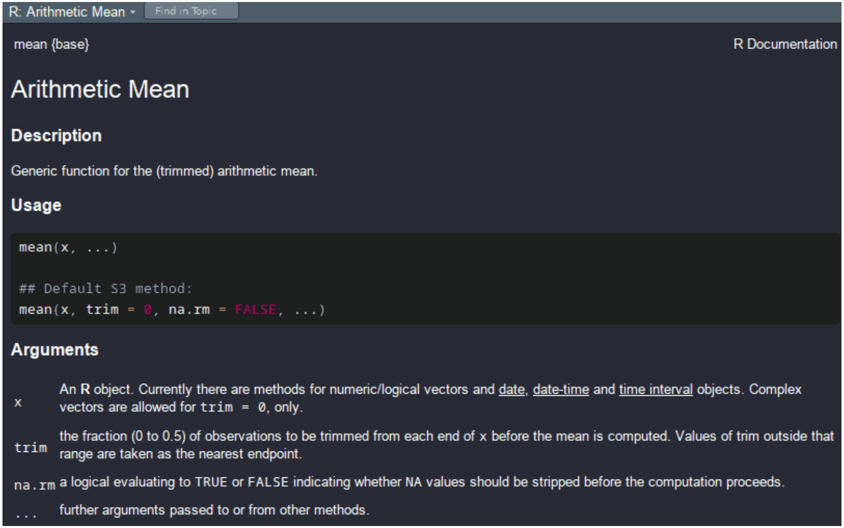
Function arguments

Functions can have mandatory and optional arguments.

- **Mandatory:** must be provided, otherwise the function won't run
- **Optional:** may be omitted because some default values are already pre-programmed

Check `?funcName` and see the help: quite often, there are agreed-upon defaults that may be unsatisfactory for certain applications.

Always read function help



mean {base} R Documentation

Arithmetic Mean

Description

Generic function for the (trimmed) arithmetic mean.

Usage

```
mean(x, ...)

## Default S3 method:
mean(x, trim = 0, na.rm = FALSE, ...)
```

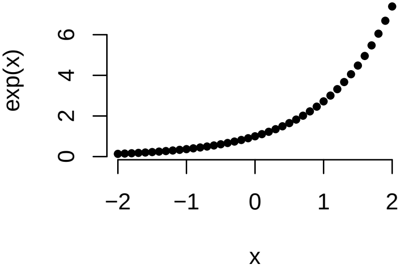
Arguments

`x`	An **R** object. Currently there are methods for numeric/logical vectors and date, date-time and time interval objects. Complex vectors are allowed for `trim = 0`, only.
`trim`	the fraction (0 to 0.5) of observations to be trimmed from each end of x before the mean is computed. Values of trim outside that range are taken as the nearest endpoint.
`na.rm`	a logical evaluating to `TRUE` or `FALSE` indicating whether `NA` values should be stripped before the computation proceeds.
`...`	further arguments passed to or from other methods.

Functions of a single argument

`exp()` is a function that takes any numeric argument (x) and returns e^x:

```
exp(1)  # 2.71828...
exp(0)  # 1
x <- seq(-2, 2, 0.1)
plot(x, exp(x))
```



```
exp()
#> Error in exp() :
#>   0 arguments passed to 'exp' which requires 1
```

Function of multiple arguments

`mean()` is a function that takes any argument (x) of length `n = length(x)` and returns $\frac{1}{n}\sum_{i=1}^{n} x_i$ – but it has two extra arguments:

- Logical `na.rm` determines if `NA` are omitted or propagated
- Numeric `trim` determines the fraction of observations to omit from both ends (i. e. `trim = 0.5` yields the median; `trim = 0.25` the inter-quartile mean)

Argument names

Every function argument has a name.

```
mean(x = c(1:9, 100))    # 14.5
mean(c(1:9, 100))        # Same
```

Named arguments can go in any order:

```
mean(x = c(1:9, 100), trim = 0.1)  # 5.5
mean(trim = 0.1, c(1:9, 100))      # Same
```

Check the help page and the definition:

```
mean(x, trim = 0, na.rm = FALSE)
```

Hence, `mean(c(1:9, 100, NA), 0.1, TRUE)` works but `mean(c(1:9, 100, NA), TRUE, 0.1)` fails.

Who needs user functions

Many functions used in economic analysis take other functions as inputs:

- Optimisers / solvers: objective function to minimise + some information about the search range + solver parameters
- Wrappers to pre-transform data / do sanity checks
- Random number generators for Bayesian / Monte-Carlo / simulation / stochastic methods

Plots from complex returns are best produced with functions.

Creating a function

Create a function by assigning the following structure to the name that you want:

```r
myFunName <- function(arg1, arg2) {
  # Necessary operations
  # carried out on the input arguments
  # (and maybe global objects -- but
  # this is bad practice: non-pure function)
  return(output)
}
```

Default return value

If the return value is not stated, the last evaluated expression is returned:

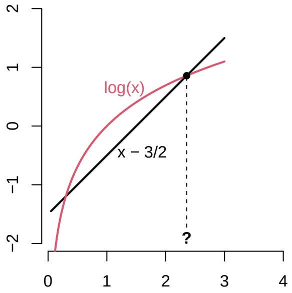
```
multByTwo <- function(x) {
  out <- x*2
  return(out)
}
```

is the same as

```
multByTwo <- function(x) x*2
```

Example: function as an input

Solve the equation $\log x = x - 3/2$ by finding the roots of $f(x) = \log(x) - (x - 3/2)$ (i. e. find x^*: $f(x^*) = 0$). Suppose that we are interested in the greater root ($1 < x^* < 3$).



```
f <- function(x) log(x) - x + 1.5
uniroot(f, interval = c(1, 3), tol = 1e-8)
#> $root
#> 2.357677
#> ...
log(2.357677) + 1.5  # 2.357677
```

Shorthand and anonymous functions

`\(x)` is the same as `function(x)`, which can save space and – in *some* cases – improve readability.

```r
f <- \(x) log(x) - x + 1.5
```

However, if `uniroot()` require a function as an input, we can supply it on the fly without ever saving it as an environment object – thus creating an **anonymous function**:

```r
uniroot(\(x) log(x) - x + 1.5,
        interval = c(1, 3), tol = 1e-8)
```

Returning multiple objects

Return is always a single object. Multi-object returns are not supported. In case more than one result has to be saved, combine everything into a list / vector / matrix etc.:

```
f1 <- function(x) return(x^2, x^3)
f1(2)
#> Error: multi-argument returns are not permitted

f2 <- function(x) return(list(square = x^2,
                               cube = x^3))
f2(2)
#> $square
#>       4
#> $cube
#>       8
```

Avoiding wrong returns

If an operation is carried out on a subset, only a subset will be returned – but not the full object! Suppose that we want to replace the 3rd element of a vector with `NA`.

```r
makeThirdNA <- function(x) x[3] <- NA
a <- makeThirdNA(1:5)
a  # 5, but not 1 2 NA 4 5
```

Last evaluated expression: `x[3]` after `` `<-` `` – i. e. simply `NA`.

```r
makeThirdNAGood <- function(x) {
  x[3] <- NA
  x  # This should be returned!
}
makeThirdNA(1:5)  # 1 2 NA 4 5
```

Best practice: explicitly return (or call) the full object!

Stopping evaluation upon return

Once a `return()` is hit during evaluation, it stops and yields the value.

```
f <- function(x) {
  return("to sender")
  return("of the Jedi")
  print("You should not see this text")
  return(0)
}

f()
#> "to sender"
```

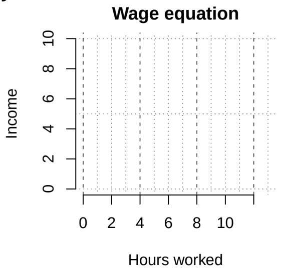
Returning early on condition

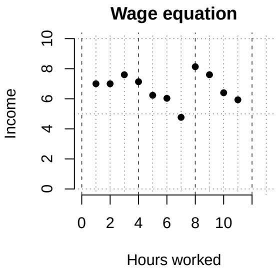
Function may have conditional returns: if some condition is satisfied, return A; otherwise, continue and return B.

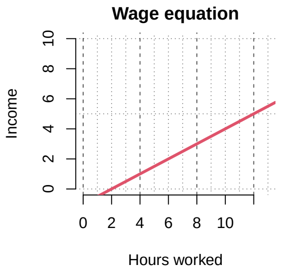
```r
myPrint <- function(x) {
  if (mean(x) < 0) {
    cat("Avg. x < 0; returning the mean\n")
    return(mean(x))
  }
  Sys.sleep(2) # Some long operation
  cat("Returning the square root of x[x>0]\n")
  return(sqrt(x[x > 0]))
}
myPrint(-9:2)  # Fast
myPrint(1:3)   # Slow
```

Wrap routines to lay the groundwork

```r
prepCanvas <- function() {
  plot(NULL, NULL, xlim = c(0, 13), ylim = c(0, 10), bty = "n",
    xlab = "Hours worked", ylab = "Income", main = "Wage equation")
  abline(v = (v4 <- seq(0, 12, 4)), col = "#000000AA", lty = 2)
  abline(v = setdiff(0:13, v4), col = "#00000088", lty = 3)
  abline(h = c(0, 5, 10), col = "#00000088", lty = 3)
}
```







```r
prepCanvas()
```

```r
prepCanvas()
points(1:11,
  mtcars$mpg[1:11]/3,
  pch = 16)
```

```r
prepCanvas()
abline(a = -1, b = 0.5,
  col = 2, lwd = 3)
```

Functions without arguments

Functions can have zero arguments:

```
f <- function(x) print("Hello")
f()
```

Functions with all default values of optional arguments can be called with no arguments:

```
g <- function(str = "dear user")
  print(paste0("Hello, ", str, "!"))
g("gorgeous")  # Hello, gorgeous!
g()            # Hello, dear user!
```

Non-pure functions

Functions may rely on global objects:

```r
d <- mtcars  # User object
f <- function(v) mean(d[, v])
f("mpg")     # Works
```

In some circumstances, it may cause issues, but in general, this is acceptable.

Not using arguments

It is completely safe to define arguments (even mandatory ones without default values!) and never use them:

```r
f <- function(x, y) print("Oh no, anyway...")
f()
f(1, "whatever")
```

Unforeseen named arguments

Unless special measures are taken (like the ellipsis), functions will not ignore named arguments absent in their definition but provided by the user.

```r
myPrint <- function(x) print(x[1])
myPrint(11:20)      # 11
myPrint(x = 11:20)  # 11

myPrint(y = 11:20)
#> Error:  unused argument (y = 11:20)
```

Errors and warnings

Instead of returning output, functions may stop with an error message, halting script execution.

```
plot(1:2, 1:3)
#> Error in xy.coords: 'x' and 'y' lengths differ
```

On warnings, execution continues, the output is returned:

```
1:2 + 1:3  # 2 4 4
#> Warning message: longer object length is not a
#>   multiple of shorter object length
```

Silent errors / side effects are dangerous: what if the user never knew that a non-trivial result was returned?

```
1:2 + 1:4  #  2 4 4 6 -- recycled 1:2, no warning!
```

Invocation: `stop("Message")` and `warning("Message")`.

Documenting functions

Document your functions: you **will** forget the details soon.

Describe the general logic, put URLs or references to formulæ or pages in papers.

```
# Pricing-error matrix: multiply M and Y element-wise and sum over
# the cash flows of a given PE investment. Then subtract the prices

# No need to do extra subsetting operations if there are no weights
```

If a function relies on generated formulæ / complex expressions / calibration constants, show the source:

```
# Computes the copula cross-derivative d/du d/dv amh(u, v)
# All expressions generated in Wolfram Mathematica 11
# amh = u*v/(1 - k*(1-u)*(1-v));
# FullSimplify[D[amh, u, v]] // InputForm
amh <- -(1+k*(-2+u+k*(u-1)*(v-1)+v+u*v)) / (-1+k*(u-1)*(v-1))^3
```

Input checking

Enforcing input types / properties is better than writing wishful comments. Quick checks prevent many errors.

```
if (length(x) != length(y))
  stop("Lengths of x and y differ.")
```

Be nice, produce helpful information (where to look):

```
id <- c(1, 1, 1, 2, 2, NA, 3, NA, 3)
p <- !is.finite(id)
s <- sum(p)
if (s > 0) {
  pos <- paste0(which(p), collapse = ", ")
  stop(paste0("Matching impossible: ",
    s, " IDs not finite; check rows ", pos))
}
```

```
#> Error: Matching impossible: 2 IDs not finite; check rows 6, 8
```

Warning or errors on type mismatch?

Throwing an error is safer (‘fail fast’ principle!), but an informative warning for an exception is good, too:

```
mean("Life")
#> Warning: argument is not numeric or logical
```

If a warning is too frequent or fussy (appears in cases where it can be safely ignored), the user may forget to check the warning (most users don’t!).

Check the last warnings with `warnings()`.

```
f <- function(x) sqrt(x) + 1
f(4:(-4))
warnings()
#> Warning message: In sqrt(x) : NaNs produced
```

Suppressing warnings

Recall Session 2: do not ignore console warnings; investigate them as if they were errors.

But what if a warning is silly, obnoxious, or ignorable in a given task, and we want to suppress specific warnings (not all, but some that we deem ‘unimportant’)?

Fussy function:

```r
f <- function(n = 10) for (i in 1:n) warning(i)
f()  # Prints 10 warnings by default
```

Suppress the warnings at your own risk:

```r
suppressWarnings(f(100))  # Quiet
```

Example: loading multiple packages at once

`library()` can only take 1 argument. Can we automate `library("this")`, `library("that")`, ...?

```r
loadPkgs <- function(pkgs, # Char vector of pkg names
                     install.missing = FALSE) {
 for (x in pkgs) {
   if (x %in% rownames(installed.packages())) {
     library(x, character.only = TRUE)
   } else {
     if (install.missing) install.packages(x) else
   warning(paste0("Install package ", x, " manually!"))
   }}
}
loadPkgs(c("boot", "BMS"))
loadPkgs(c("boot", "BMS"), install.missing = TRUE)
```

Default behaviour: warning (non-invasive).

Initialisation of arguments with NULL

NULL is useful where the absence of a default value requires a special action. **Example:** assign decreasing seat numbers to numbered attendees and use names if they are supplied (relying on length(NULL) being 0):

```
cards <- function(n, names = NULL) {
  x <- if (length(names) == n) names else paste("Guest", 1:n)
  y <- n:1
  return(paste0(x, " [seat ", y, "]"))
}
cards(4)  # "Guest 1 [seat 4]" "Guest 2 [seat 3]"
#           "Guest 3 [seat 2]" "Guest 4 [seat 1]"
cards(4, names = c("Alex", "Bella", "Camille", "Dana"))
# "Alex [seat 4]" "Bella [seat 3]" "Camille [seat 2]" "Dana [seat 1]"
```

Q: Why is `length(names) == n` safer than `is.null(names)`? **A:** to safeguard against bad length. **But!** `if (length(names) != n) warning(...)` is even better.

Any questions on functions and arguments?

Methods, namespaces, calls & ellipses

Checking function definition

Type the function name in the console to see what is going under its hood.

Example: linear regression with robust standard errors.

```
library(lmtest)   # For coeftest
library(sandwich) # For vcovHC
mod <- lm(mpg ~ cyl + hp, data = mtcars)
print(coeftest(mod, vcov. = vcovHC))
```

Run the next line (just one word – no brackets etc.):

```
vcovHC
#> function (x, ...) UseMethod("vcovHC")
```

But what does it mean?

Methods in R

Recall that everything is an object.

- Objects can be of multiple **classes** – well-defined types (matrix, linear model, hypothesis test arbitrary list)
  - Custom classes can be created
- The same function can be an umbrella term for multiple sub-functions that operate on various classes
- R packages aimed at user convenience provide new methods for their custom classes

Know thy methods

`plot()` is a method:

- By default, it plots scattered point pairs (*x*, *y*)
  `plot(1:20, 1:20, pch = 1:20)`
- For a time-series object (`class(x) == "ts"`), lines
  `plot(ts(mtcars$mpg, start=c(1996,1), freq=4))`
- For a kernel density estimator, line plot + bandwidth
  `plot(density(mtcars$mpg))`
- For a linear model, an *interactive* diagnostic plot (requiring the user to hit ⏎)
  `plot(lm(mpg ~ cyl + hp, data = mtcars))`

See `methods(plot)` for the full collection!

Disassembling the vcovHC() method

Type `vcovHC.default` (no brackets) in the console:

```
vcovHC.default
function (x, type = ...)
{
    type <- match.arg(type)
    rval <- meatHC(x, type = type, omega = omega)
    if (sandwich)
        rval <- sandwich(x, meat. = rval, ...)
    return(rval)
}
# <environment: namespace:sandwich>
```

Do the same for `meatHC`.

Hidden functions

When going into the rabbit hole of debugging and following the functions chains, one might encounter a function or method that is not callable:

```
m <- estfun.lm(obj) # Deep in meatHC
estfun.lm
#> Error: 'estfun.lm' is not an exported
#> object from 'namespace:sandwich'
```

When functions are not exported into a package namespace, they can still be invoked via the **triple colon**:

```
sandwich:::estfun.lm # Works
```

Using `:::` is extremely helpful for debugging errors cause by functions in others' packages.

Unveiling built-in methods

Whenever `plot()` is called, R checks the object class and reaches for the appropriate function.

Even the built-in methods can be hidden. Look them up in packages base, `utils`, `stats`, `graphics` etc. by using `:::` (luckily, RStudio auto-suggests the functions).

```
stats:::plot.density
stats:::plot.lm
graphics:::barplot
```

Methods are usually documented: check `?plot.lm` and `?plot.density` to see the difference. Note that these methods take completely different arguments, yet the function call – `plot()` – is the same.

Namespace precaution with methods

Packages provide functions, classes, and *methods* to apply to the custom *classes*. Namespace clashes and **masking** may occur with methods, too.

```
lag  # function (x, ...) UseMethod("lag")
#      <environment: namespace:stats>
lag(1:3)
#> 1 2 3
library(dplyr)
#> The following objects are masked from ‘package:stats’:
#>     filter, lag
lag(1:3)
#> 0 1 2
```

Solution: explicitly call functions that are masked by other packages: `stats::lag` or `dplyr::lag`.

Argument support in methods

Some packages support extra arguments for the methods of the same name. `summary.lm()` cannot compute robust SEs for a linear model, but `AER:::summary.ivreg()` supports custom robust estimators:

```
summary.lm
#> function (object, correlation = FALSE, ...)
AER:::summary.ivreg
#> function (object, vcov. = NULL,
              df = NULL, diagnostics = FALSE, ...)
```

```
modIV <- ivreg(mpg ~ hp + cyl |  # Very silly 2SLS
                 qsec + gear + cyl, data = mtcars)
class(modIV)    # "ivreg"
summary(modIV)
summary(modIV, vcov. = vcovHC)
```

Extracting coefficients

Different functions may return similar summaries as different classes. Cf. `stats::summary.lm()` and `lmtest::coeftest()`.

```
library(lmtest); library(sandwich)
mod <- lm(mpg ~ cyl + hp, data = mtcars)
s1 <- summary(mod)
s2 <- coeftest(mod, vcov. = vcovHC)
s1; s2
```

- `class(s2)="coeftest"`, but `is.matrix(s2)=TRUE` (see `str(s2)`); the robust SEs are `s2[, 2]`
- `class(s1)="list"` with elements `coefficients`, `r.squared`, `fstatistic` – they are non-robust and are thus useless at best (dangerous at worst)

Ellipsis

In C, C++, and R, `...` stands for ‘variable number of parameters to a function’.

`max()` returns the maximum scalar value from all of its inputs, which can be an arbitrary number of vectors:

```
max(1:100)                        # 100
max(1:10, mtcars$mpg, 100)  # 100
```

See `?max` – it is defined as `max(..., na.rm = FALSE)`.

Passing ellipses to nested functions

If a function extends existing functions, `...` is a way to catch all the inputs and pass them further:

```
myMean <- function(x, ...) {
  # In this example, ... will be passed to mean()
  print("Here are the original values")
  print(x)
  mean(x, ...)
}

myMean(c(1:9, 100))                # 14.5
myMean(c(1:9, 100), trim = 0.1)   #  5.5
```

Notice how `myMean()` accepts the `trim` argument despite it not appearing in the definition.

Ellipses capture bad inputs

Recall the example where `f <- function(x) x[10]` would fail upon `f(y = 11:20)`.

It is possible to sink all erroneous inputs into the ellipsis:

```
h <- function(x = NA, ...) x[10]
h(11:20)     # 20
h(x = 11:20) # 20
h(y = 11:20) # NA, but no error!
```

Function call

Instead of writing long lists of inputs for all functions, one can assemble them into a single list of named arguments and do a **function call**.

```
f <- function(x, y, z) max(x, y, z)
f(x = 1:3, y = mtcars$mpg, z = 50)
```

is the same as

```
a <- list(x = 1:3, y = mtcars$mpg, z = 50)
do.call(what = f, args = a)
```

Use case for function calls

Multiple plots with similar formatting can be produced with identical arguments – the naïve approach is cumbersome:

```r
plot(x = mtcars$cyl, y = mtcars$mpg, ylim = c(0, 40), bty = "n",
     pch = 3, col = 2,  xlab = "Cylinders",
     ylab = "Miles per gallon", main = "Fuel efficiency")
plot(x = mtcars$disp, y = mtcars$mpg, ylim = c(0, 40), bty = "n",
     pch = 3, col = 2, xlab = "Displacement",
     ylab = "Miles per gallon", main = "Fuel efficiency")
plot(x = mtcars$hp, y = mtcars$mpg, ylim = c(0, 40), bty = "n",
     pch = 3, col = 2, xlab = "Horsepower",
     ylab = "Miles per gallon", main = "Fuel efficiency")
```

What if a change is needed (e. g. `pch = 16`)? Changing all instances is tedious and error-prone.

Recall Session 2, the DRY principle!

Saving effort with argument list calls

In the example above, only x and `xlab` change. Consolidate all identical arguments into a named list and re-use them:

```r
a <- list(x = mtcars$hp, xlab = "Horsepower",
  y = mtcars$mpg, ylim = c(0, 40), bty = "n",
  pch = 3, col = 2, ylab = "Miles per gallon",
  main = "Fuel efficiency")
do.call(plot, a)
```

Change the *x* variable and axis label – keep the style:

```r
a[c("x", "xlab")] <- list(mtcars$cyl, "Cylinders")
do.call(plot, a)
```

Change the plotting character in the common definition:

```r
a$pch <- 1; do.call(plot, a)
```

Looping over called arguments

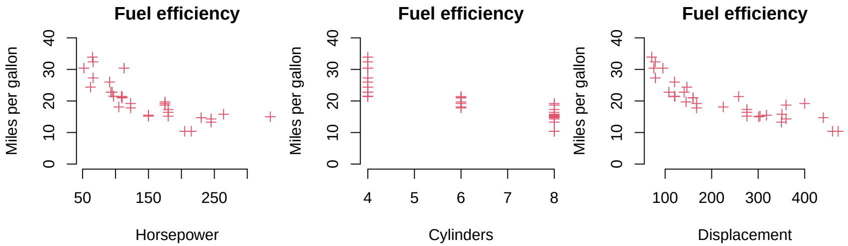
Create lists of constant and changing arguments:

```
a <- list(y = mtcars$mpg, ylim = c(0, 40),
  pch = 3, col = 2, ylab = "Miles per gallon",
  bty = "n", main = "Fuel efficiency")
b <- list(  # Our first list of lists!
  list(x = mtcars$hp,   xlab = "Horsepower"),
  list(x = mtcars$cyl,  xlab = "Cylinders"),
  list(x = mtcars$disp, xlab = "Displacement"))
```

Substitute dynamically and enjoy the show:

```
for (i in 1:3) {
        do.call(plot, c(a, b[[i]]))
        Sys.sleep(1)}
```

Looping over called arguments – result



Avoid repetition in places where changes can be expected in the future.

Capturing ellipses as a list

It is possible capture the ellipsis into a list and treat it as any other list:

```
f <- function (...) {
        a <- list(...)
        print(a)
}

f(1:3)  # Returns a list of length 1
#> [[1]]
#> 1 2 3

f(anything = "is", possible = 1:3, 100)
#>  $anything      $possible      [[3]]
#>       "is"          1 2 3        100
```

Being selective with ellipses

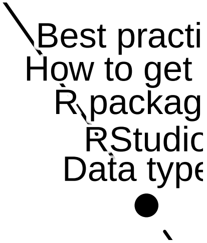
If an inner function *g* fails with *all* named arguments of the main function, call *g* with a sub-list of `list(...)`.

```
mean(c(1:9, 100, NA), na.rm = T, trim = 0.1)
sd(c(1:9, 100, NA),   na.rm = T) # Works
sd(c(1:9, 100, NA),   na.rm = T, trim = 0.1) # Error

meanSD <- function(x, ...) {
  a <- list(...)
  a$x <- x
  m <- do.call(mean, a)
  s <- do.call(sd, a[names(a) != "trim"])
  c(mean = m, SD = s)
}

meanSD(c(1:9, 100, NA))
meanSD(c(1:9, 100, NA), na.rm = T)
meanSD(c(1:9, 100, NA), na.rm = T, trim = 0.1)
```

Ellipses in these slides



Recall Session 1, *'Amount of technical detail per session'*. **NB:** the black text over black lines is somehow unobstructed.

This is achieved with `...` + calls of `text()`.

- Create 12 angles $\{\alpha\}_{i=1}^{12}$ from 0°to 330°(like a clock face)
- In a loop over $i \in \overline{1, 12}$, call `text()` at $x + 0.1 \cos \alpha_i$, $y + 0.1 \sin \alpha_i$ with white colour
- Call `text()` at (x, y) with black colour
  - If one requests smaller text / different font, both white and black `text()` calls are using the same custom parameters

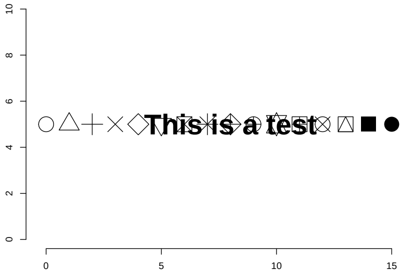
We shall learn how to create such plots in Session 5.

Text with white halo – simple version

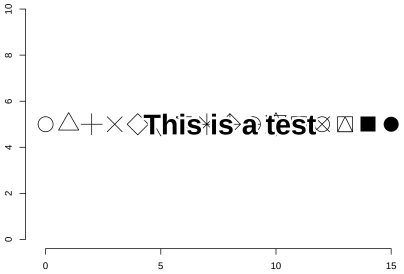
```r
textHalo <- function(x, y, labels, ...) {
  angl <- seq(0, 11/6*pi, length.out = 12)
  args <- list(...)
  args$labels <- labels
  args$col <- "white"
  for (i in 1:12) {
    args$x <- x + 0.1*cos(angl[i])
    args$y <- y + 0.1*sin(angl[i])
    do.call(text, args)
  }
  text(x = x, y = y, labels = labels, ...)
}

plot(0:15, rep(5, 16), pch = 1:16, cex = 3.5,
     ylim = c(0, 10), bty = "n", asp = 1)
textHalo(8, 5, "This is a test", font = 2, cex = 3)
```

Compare the legibility



Without halo



With a halo

Q: how can `textHalo()` be improved?

Any questions on argument lists and calls?

Debugging

Functions should be used responsibly

THIS IS YOUR MACHINE LEARNING SYSTEM?
YUP! YOU POUR THE DATA INTO THIS BIG PILE OF LINEAR ALGEBRA, THEN COLLECT THE ANSWERS ON THE OTHER SIDE.
WHAT IF THE ANSWERS ARE WRONG?
JUST STIR THE PILE UNTIL THEY START LOOKING RIGHT.
DATA
ANSWERS

Credit: xkcd. Technical explanation.

Reasons for debugging

With scripts containing no functions, it is easy to find the error: evaluation simply stops there.

However, if you wrote a 200-line function, its failure implies that the error must be located.

If you are using an external function / package, the error can be caused by hard-to-reach code.

How to reach it?

Debugging methods

One only knows that there is a need for debugging if the evaluation fails (programme error) or (if there is no error) if the resuls looks strange (eyeball / formal test).

- Change the function, insert `print()` in key places
  - Every available function can be modified
  - Print summary of intermediate objects or the ol’ reliable `print("Made it to here! 3")`
  - If the function relies on other functions ‘hidden’ in package namespaces, requires more effort
- Software debugger: step-by-step execution
  - Going inside functions
  - Step-by-step execution
  - Explore **local variables** in the memory
  - Trace which function calls which one

Debugging: real example

- A week ago, our colleague was trying to plot estimation results for a model
- Estimation succeeded, but plotting failed

The error is caused by some internal function deep down the call chain → we cannot copy-paste the source easily.

Non-functioning code

```r
# install.packages(c("BMS", "haven"))
library(BMS)    # Bayesian estimation
library(haven)
d <- read_dta("LEVEL_trial_A.dta")
d <- d[complete.cases(d), ]
set.seed(1)
mod <- bms(d, burn = 10000, iter = 100000,
           mprior = "uniform", mcmc = "bd",
           user.int = FALSE, logfile = "")

image(mod) # Error
plot(mod)  # Only 1 / 2 plots
```

Traceback

Function f_1 calls f_2, which calls f_3 and g_3, and g_3 calls h (which can be from package p)...

```r
traceback()
```

```
#> 5: crossprod(bmao$arguments$X.data[, 1] -
 ↪  mean(bmao$arguments$X.data[, 1]))
#> 4: as.vector(crossprod(bmao$arguments$X.data[, 1]
 ↪  - mean(bmao$arguments$X.data[, 1])))
#> 3: pmp.bma(x, oldstyle = TRUE)
#> 2: image.bma(mod)
#> 1: image(mod)
```

Traceback conclusions

- `image()` calls the `image.bma()` method, which calls the `pmp.bma()` function (posterior model probabilities)
- `pmp.bma()` fails in the line where a cross-product and a mean is computed
  - `crossprod()`: requires numeric / complex matrix / vector arguments
  - `mean.default(bmao$arguments$X.data[, 1])`: argument is not numeric or logical: returning `NA`

Enable / disable debugging

- `debugonce(functionName)`: enable step-by-step evaluation of once certain function for one time
- `debug(fnName)`, `undebug(fnName)`: enable or disable such evaluation at every run

Practice makes perfect

Live demonstration time!

Going inside functions with debugonce()

```r
debugonce(pmp.bma)
pmp.bma(mod)
```

Press F10 or type ‘n’ in the console to advance by 1 line.

Here is the culprit:

```r
mean(bmao$arguments$X.data[, 1])
#> NA
```

```
#> Warning message:
#> In mean.default(bmao$arguments$X.data[, 1]) :
#>  argument is not numeric or logical: returning NA
```

Problematic argument

How can the mean of a numeric vector without `NA`'s be `NA`?

```
head(bmao$arguments$X.data[, 1])
#> # A tibble: 6 × 1
#>        pcc
#>      <dbl>
#> 1 0.00933
#> 2 0.462
#> 3 0.0179
#> <...>
```

```
class(bmao$arguments$X.data[, 1])
#> "tbl_df"      "tbl"         "data.frame"
```

Shockingly, `X.data[, 1]` is not a numeric vector; the dimensions are not dropped → the `data.frame` attributes are not lost → `X.data[, 1]` remains a list (data frames are lists) → `mean()` is not defined on lists.

Fixing the problem

Turns out, the input data set belongs to a custom class (‘tibble’) that does not behave like a data frame would (here, it remained a list after 1 column was selected).

```r
d <- read_dta("LEVEL_trial_A.dta")
d <- d[complete.cases(d), ]
d <- as.data.frame(d)  # <=== ADD THIS LINE
```

Then, everything works.

Help page for `?bms`:

```r
bms(X.data, ...)
#> X.data -- a data frame or a matrix...
```

Tips for avoiding similar bugs

- Functions from one package do not work well with custom classes provided by other packages
  - Suffering from success: the ecosystem is proliferating, the community of contributors is very diverse
  - `sandwich`, `stargazer` etc. support certain model classes from certain packages, but not all
- Your collaborators can be working in Python, Matlab etc.
  - They have no idea what a 'tibble' is
  - The least common denominator is a numeric array
- Use any package the does the job for you, but convert the output to standard classes
  - Read the help page for the functions that you are using; make sure that the input belongs to a correct class: matrix → `as.matrix()`, DF → `as.data.frame()`
  - Check the properties of inputs used in help-page examples

Adding breakpoints with browser()

```r
f <- function(data) {
  n <- nrow(data)
  mod <- lm(mpg ~ cyl + hp, data = data)
  # browser()
  s <- sd(data$mpg)
  list(n.obs = n, SD = s, coef = coef(mod))
}
f(mtcars)  # Works

d <- mtcars; d$mpg[10] <- NA
f(d)  # One of the outputs is NA
```

Uncomment the `browser()` line and run `f()`; explore the memory at the breakpoint. `n` an `mod` are available. Check `length(mod$residuals)`.

Print debugging

Sometimes, simple methods are the best.

```r
f <- function(data) {
  n <- nrow(data)
  cat("Number of rows in 'data': ", n, "\n")
  mod <- lm(mpg ~ cyl + hp, data = data)
  cat("Number of obs in 'mod': ",
    length(mod$residuals), "\n")
  s <- sd(data$mpg)
  list(n.obs = n, SD = s, coef = coef(mod))
}
d <- mtcars; d$mpg[10] <- NA
f(d)
```

```
#> Number of rows in 'data':  32
#> Number of obs in 'mod':  31
```

Debugging with loops

The next section is about getting rid of loops (because they are ‘bad’ or can be replaced with vectorised functions). However, for debugging, loops can be kept for easier browsing via `browser()`.

```r
f <- function(x)  # Nice and vectorised
  ifelse(x == 5, stop("I fail here"), x^2)
f(1:10)
```

```r
fLoop <- function(x) {
  for (i in 1:length(x)) {
    print(i)
    f(x[i])}}
fLoop(1:10)
```

We see the point of failure explicitly.

Catching exceptions

Catching errors, especially in large-scale simulations or complex procedures is essential to allow *some progress* even in the presence of errors.

In economic research:

- Bootstrap replications might fail
- Solver or estimation routine might not converge
- Experiments / simulations might throw an error

Use `tryCatch()` to catch errors and allow the workflow to continue: in case of a warning or an error, call a different function that operates on the error / warning itself (or just returns `NULL`).

Using tryCatch() for exceptions

`f <- function (n) plot(1:2, 1:n)` works only for $n = 2$. If applied to other $n \in \overline{1, 10}$, it fails. Allow failures:

```
f2 <- \(n) tryCatch(f(n), error = \(e) return(e))
res <- lapply(1:10, f2)
res
```

```
[[1]] <simpleError in xy.coords(...): 'x' and 'y' lengths differ>
[[2]] NULL
[[3]] <simpleError in xy.coords(...: 'x' and 'y' lengths differ>
```

Check which returns contain ‘error’ in their class:

```
sapply(res, \(x) any(grepl("[Ee]rror", class(x))))
#> T F T T T T T T T T
```

Distinct conditions for tryCatch()

If the function should return something (and cannot possibly return `NULL`), return `NULL` on an error (debug later):

```r
tryCatch(5^list(), error = \(x) NULL)
```

Return different outputs for errors and warnings:

```r
a <- list(3, "Trap", -4:2)
lapply(a, log)  # Fails on a[[2]]
res <- lapply(a, \(x) tryCatch(log(x),
   error = \(e) NULL, warning = \(w) "Danger!"))
which(sapply(res, is.null))  # Indices of errors
which(sapply(res, \(x) is.character(x) &&
  length(x) == 1 && x[1] == "Danger!")) # Warnings
```

NB. Depending on the return type, the user might consider implementing simpler / safer checks.

Any questions on debugging methods?

Vectorisation and parallel computing

Vectorised cutting

To separate a numeric variable into a categorical one (e. g. 0–3 = ‘Fail’, 4–5 = ‘Low’, 6–7 = ‘Medium’, 8–10 = ‘High’), one may be tempted to use `ifelse()`:

```
ifelse(x < 0, NA,
  ifelse(x <= 3, "Fail",
    ifelse(x <= 5, "Low", ... )))
```

A much better approach is using `cut()` to cut at certain levels. k categories require $k + 1$ cuts (including the left and right extremes):

```
cut(x, breaks = c(-Inf, -1e-16, 3, 5, 7, 10, Inf),
       labels = c(NA,"Fail","Low","Med","Hi",NA))
```

Greatest sin: growing objects

The crucial thing to carry away form this course: **never grow objects in loops** by prepending / concatenating / binding. Always allocate space for the results and fill the result vector element by element.

Users often add columns to data frames one by one in a loop. This leads to catastrophic inefficiency, up to the point of halting, **especially with big data**.

Think like a computer: it is faster to process 1000 smaller objects than 1 large object 1000 times.

Unless you know that the loop is short and the object size is small, do not concatenate objects with themselves.

Possible vector-creating solutions

Task: create a vector of $\sqrt{i}$ for $i = 1, 2, \dots, n$.

Best: avoid loops at all if the function is vectorised (more on that later).

```
system.time({a <- sqrt(1:n)})
```

Next best: initialise result vectors (C/C++ flavour).

```
a <- numeric(n) # A vector of zeros
system.time({for (i in 1:n) a[i] <- sqrt(i)})
```

Bad: concatenate the old vector and new value.

```
a <- NULL
system.time({for (i in 1:n) a <- c(a, sqrt(i))})
```

Timing of object growing, milliseconds

	`sqrt(1:n)`	`a[i]<-`	`c(a, ...)`
n = 20	0.002	1.6	190
n = 2000	0.008	1.8	193
n = 200 000	0.570	13.1	33 160

- Research in 2013: on a large data set (e. g. 1-minute returns of 100 stocks for 15 years), compute hourly variances. If each value takes 1 ms, the total run time of `a <- c(a, ...)` is ≈300 years
- Research in 2023: computing 1-minute volatilities from 1-second returns via growing would take 300 m years

Average object grower

Credit: Edaphosaurus pogonias by Nobu Tamura.

Initialising vectors

Create vectors of given length and specified type:

```r
numeric(100)    # Filled with zeros
character(100)  # Filled with ""
logical(100)    # Filled with FALSE
rep(NA, 100)    # Not assuming any class
matrix(NA, nrow = 3, ncol = 4)
vector(mode = "list", length = 10)
```

Reserving memory and declaring the desired type is the best solution to avoid object growing.

Use of NULL for growing objects

Recall the slowest solution:

```r
a <- NULL   # Initialised to length 0
for (i in 1:100) a <- c(a, sqrt(i))
```

Such concatenation for growing objects is **strongly discouraged**. In most cases, there is a much faster and memory-efficient loop-free solution.

However, when the loop is guaranteed to be short and the grown objects do not take up much memory, such growing method (starting from `NULL`) *can* be viable.

Processing split data without loops

Benefits of lists: they can be processed by loop-less operations. *Learn to love them!*

- Even if the calculations are long (e. g. estimating a complex non-linear model for each year in a panel), they can be parallelised efficiently over CPU cores

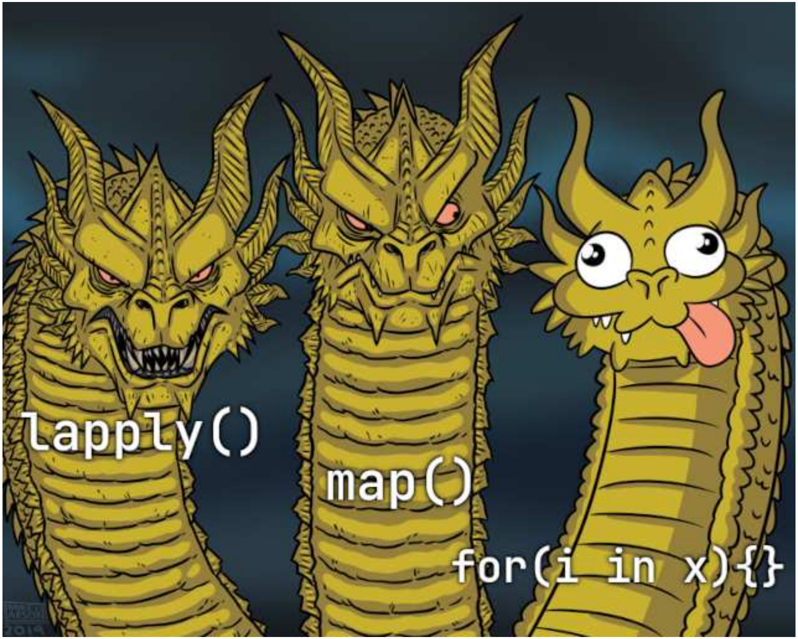
Can you read this in human English?

```r
d.list <- split(d, d$id)
res.list <- lapply(d.list, function(x) {
  if (mean(x$income[x$cond1])<1)
    x$incNA[x$cond2] <- NA
  return(x)
})
d <- do.call(rbind, res.list)
```

The apply family of functions

The `apply()`, `lapply()`, `sapply()` functions are incredibly useful.

- `apply()`: compute statistics across the selected dimensions of multisimensional arrays
  - Typical use: compute something by row or a column of a matrix (e. g. standard deviation by column, the percentage of missing observations per row)
- `lapply(x)`: apply a function to each element of the vector x and return a list
  - Lists are also vectors
- `sapply(x)`: apply a function to each element of the vector x and simplify the result if possible


lapply()
map()
for(i in x){}

apply: function across array dimension

Apply any function across any dimension of a multi-variate array. Recall the order: 1 = rows, 2 = columns, 3 = slices etc.

Compute the standard deviation by column.

```
apply(mtcars, 2, sd)
```

Compute the standard deviation by row (silly!).

```
apply(mtcars, 1, sd)
```

Which observation is the closest to the median?

```
f <- \(x) which.min(abs(x - median(x)))
apply(mtcars, 2, f)
```

NB. If there are multiple identical values, `which.min()` returns the **first** index.

apply with 3D arrays

We create a synthetic 3D array with 100 slices where every slice is equal to `mtcars` plus some random noise.

```r
nr <- nrow(mtcars); nc <- ncol(mtcars)
a <- array(NA, dim = c(nr, nc, 100))
set.seed(1)
for (i in 1:100)
  a[,,i] <- as.matrix(mtcars) + 0.1*rnorm(nr*nc)
dimnames(a) <- list(rownames(mtcars),
       colnames(mtcars), paste0("s", 1:100))
```

Compute the SD by column (all rows, all slices) and compare it to the original:

```r
apply(a, 2, sd)
apply(mtcars, 2, sd)  # Should be similar
```

apply across multiple dimensions

Compute the SD *across* all slices – eliminate the slice dimension (3), keep the row (1) and columns (2) to indicate what the SD was for row *i*, column *j*, all slices.

```r
apply(a, c(1, 2), sd)
```

The result should be around 0.1 because we added random white noise with SD 0.1 to every element of the original matrix multiple times.

Hint: the dimensions passed to `apply()` are kept – the remaining ones are collapsed.

apply with extra arguments

If the function `f` used in `apply()` accepts extra arguments, they can be supplied to `apply()` directly because...they will be carefully captured by the ellipsis (`rii`) and passed to `f()`.

Prepare data for copying and pasting to LaTeXin just 2 lines *without any extra packages*:

```
r <- apply(mtcars, 1, paste0, collapse = " & ")
cat(paste0(r, " \\\\"), sep = "\n")
```

Excel will accept tab-separated values:

```
r <- apply(mtcars, 1, paste0, collapse = "\t")
cat(r, sep = "\n")
```

lapply: function across lists

`lapply(x, f)` is identical to a loop:

```
result <- vector("list", length(x))
for (i in 1:length(x)) result[[i]] <- f(x[[i]])
```

- If `f()` returns a list (very common!), aggregating the results into a list of lists is often the only option
- `lapply()` saves the day where the function is not vectorised for some arguments
  - Example: trying various tweaking parameters that must be passed as a length-1 scalar

  `lapply()` is indispensable where the output has varying length depending on the input and the results cannot be represented by a rectangular array

lapply to study tweaking parameters

Recall our equation solver:

```
f <- function(x) log(x) - x + 1.5
uniroot(f, interval = c(1, 3), tol = 1e-8)
```

But what does the tolerance argument, `tol`, do? Intuition: if we are solving $f(x) = 0$, maybe the lenience of the solver (accepting slightly worse answers)?

```
uniroot(f, interval = c(1, 3),
        tol = c(1e-2, 1e-4, 1e-6))
```

Example of **unexpected behaviour**: we requested 3 tolerances and got only 1 result because conceptually `uniroot()` is programmed to do only one thing: find and return one root.

Getting vectors from lapply() returns

```r
tols <- 10^seq(-2, -12, -0.25) # 1e-2 ... 1e-12
r <- lapply(tols, function(y)
       uniroot(f, interval = c(1, 3), tol = y))
str(r)
```

For each value of tolerance, we got a full evaluation of the function an a list as each output value.

Extract the 'lenience' measure: $f(\text{root}) = \delta$, $\delta \approx 0$, but $\delta \neq 0$.

```r
err <- unlist(lapply(r, "[[", "f.root"))
plot(tols, abs(err), log = "xy")
```

sapply: combining homogeneous outputs

If the output of `f()` is a scalar but the function is not vectorised, calling it with `lapply()` will create a list of length-1 outputs. It could be nice to simplify it to a vector.

`sapply(x)` simply calls `lapply(x)` and checks if the lengths of the returnes elements are equal.

- If `f()` returns a scalar, `sapply(x, f)` returns a vector of length `length(x)`
- If `f()` returns a vector, `sapply(x, f)` returns a matrix by column-binding the outputs

Getting simplified returns with sapply()

For a vector of tolerances, compute a vector of errors at the solution found by the root finder:

```
tols <- 10^seq(-1, -8, -0.25)
err <- sapply(tols, \(y)
  uniroot(f, interval=c(1, 3), tol=y)$f.root)
plot(tols, abs(err), log = "xy")
```

For every variable of `mtcars`, compute the average and standard deviation:

```
meanSD <- \(x) c(mean = mean(x), SD = sd(x))
sapply(mtcars, meanSD)
```

`mtcars` is a data frame ⇒ a list. This is why DFs are handy: `lapply()` and `sapply()` work with them.

Vectorising loops

Think of a loop as of a function of the iterator. To vectorise a loop, take the entire expression in curly braces (the body of the loop) and declare it as a function of the iterator (excluding the assignments in parent environments).

Compare the two – only the return value changes:

```r
res <- vector("list", 100)
for (i in 1:100) {
 m <- cor(mtcars)
 Sys.sleep(0.1)
 if (i%%10 == 0) print(i)
 res[[i]] <- m
}
```

```r
f <- function(i) {
 m <- cor(mtcars)
 Sys.sleep(0.1)
 if (i%%10 == 0) print(i)
 return(m)
}
res <- lapply(1:100, f)
```

Benefits: `lapply()` is immediately parallelisable.

Direct vectorisation

- Define vectorised versions via `sapply()`
- Use `Vectorize()` as a convenience wrapper (to vectorise w. r. t. selected arguments)

Example: compute $f(n) \coloneqq \sum_{i=1}^{n} i$ for $n \in \{3, 4, 7\}$.

```r
f <- function(n) sum(1:n)
x <- c(3, 4, 7)
f(x)  # Throws a warnings, does not work
vf <- Vectorize(f)
vf(x)  # 6 10 28

vf2 <- function(x) sapply(x, f)
vf2(x)  # 6 10 28
```

Why parallel and vectorised processing

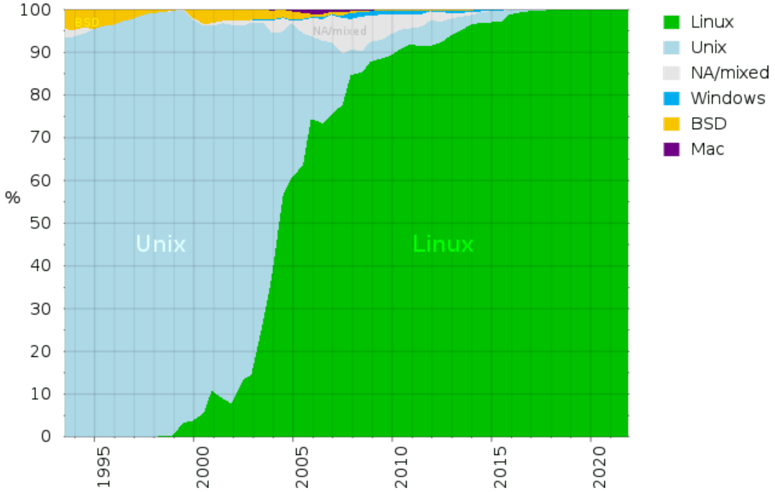
- Moore’s law is no longer applicable (clock speed = heat output)
- Modern processors support some parallel instructions (AVX512: vector exponentiation, fused mult-add…)
- But it is upon us whether those capabilities are used
- Interpreted programming languages tend to be slower

Most useful parallelisation applications

There is some sort of law working here, whereby statistical methodology always expands to strain the current limits of computation. *B. Efron (2000).*

- Computationally costly Monte Carlo simulations
- Splitting big data into chunks
- Trying many sets of tweaking parameters to see which one works
- Bootstrapping
  - Comparison of alternative confidence intervals
  - Selection of optimal block size in time-series models
  - Distribution of any complex statistic in general

Supercomputers nowadays (incl. UL HPC)



Parallelisation in R

Two facilities:

- `parLapply`
- `mclapply` (not on Windows)

`parLapply` workflow:

1. Create a cluster with cores
2. Export the objects
3. Run the function over a list in the cluster

Comparison on parallelisation types

On a machine with multiple cores, one can run parallel computations in two ways.

- **Forking:** multiple processes (`rsession`) are spawned with full environments
  - The fastest approach, but does not work on Windows
  - Can be memory-consuming – mind the bulky objects
- **Socket communication:** the master process exchanges memory and objects with child processes
  - Available on Win + Mac + Linux
  - Consumes less memory, but is slower due to the overhead
  - Requires many extra actions from the user

mclapply is easier in most cases

Developers hate extra actions – they usually develop support for forking first: it is easy. Compare the parallel bootstrap implementations (from the March 2023 course):

```r
library(parallel)  # This example will not run
parBoot <- function(...) boot::boot(data = d,
 ↪  statistic = bootCoefT, R = 999, ...)
if (.Platform$OS.type == "windows") {
  cl <- makeCluster(4)
  clusterExport(cl, c("bootCoefT", "myRegCoef"))
  res <- parBoot(parallel="snow", cl=cl, ncpus=4))
  stopCluster(cl)
} else {  # On non-Windows systems, just 1 line
  res <- parBoot(parallel="multicore", ncpus=4)
}
```

Maximising our utility with PSOCK

However, many participants are using Windows at the moment, and one of the goals of this course is to *make everyone's life easier*.

We adopt the next best solution that suits everyone: creating PSOCK clusters.

1. Create a cluster
2. Export all the objects used by the function into the cluster + load the packages
3. Use a parallel version of `lapply()` on the cluster
4. Stop the cluster manually

Speeding up slow functions

Serial evaluation of `f()` is slow (30 s):

```r
f <- function(x) {
  Sys.sleep(3); print(x)
  return(x^2)
}
system.time(ret1 <- lapply(1:10, f))
```

Parallel evaluation depends on the number of cores:

```r
library(parallel)
cl <- makeCluster(4)
system.time(ret2 <- parLapply(cl=cl, X=1:10, f))
stopCluster(cl)
```

Try using 2 or 3 cores instead of 4 and compare the timing.

Monitoring parallel execution

So far, the output has been silent because PSOCK clusters do not communicate by default. Enable returning the standard messages into the common console using the `outfile = ""` argument:

```
f <- function(x) {
  Sys.sleep(3); print(x)
  return(x^2)}
library(parallel)
cl <- makeCluster(4, outfile = "")
system.time(ret <- parLapply(cl=cl, X=1:10, f))
stopCluster(cl)
```

NB. The tasks are not done in the original order. Always a good idea to print something to track the progress.

Exporting global objects to the cluster

Non-pure functions are often applied to global objects.

```r
library(MASS) # For MASS::psi.huber
d <- mtcars
f <- \(v) psi.huber(d[1, v])
lapply(c("mpg", "cyl"), f) # Works
```

Since `MASS` and `d` belong to the global memory, this fails:

```r
cl <- makeCluster(4)
parLapply(cl = cl, X = c("mpg", "cyl"), f)
```

Export the objects (character vector of object names) and load the packages in the cluster:

```r
clusterExport(cl, "d")
clusterEvalQ(cl, library(MASS))
parLapply(cl = cl, X = c("mpg", "cyl"), f)
stopCluster(cl)
```

Overhead

More function calls result in more overhead. Compare:

- Generating 10 000 batches of 1 000 random numbers
- Generating 10 000 000 random numbers at once

```r
set.seed(1)
microbenchmark::microbenchmark(
  replicate(10000, runif(1000)),    # 0.54 s
  matrix(runif(1e7), ncol = 10000), # 0.28 s
  times = 20)
```

Twice as slow!

The penalty for loops is everywhere

Even in C++, where there are no theoretical penalties for double-looping over a matrix, some loops are better than others.

Example: compute a $G \times N$ matrix of kernel weights from observations $\{X_j\}_{j=1}^N$ on a grid $\{g_i\}_{i=1}^G$ with bandwidth:

$$\{w_{ij}\} \coloneqq k\left(\frac{g_i - X_j}{b}\right)$$

Each function call in R is penalised because it does the extra checks (classes etc.).

Call as few functions as possible.

Low-level speed-ups

C-like programming languages have the luxury of fine-tuning for performance gains.

- Enforcing special processor instructions (SIMD = single instruction, multiple data)
- Tweaking the memory layout (to reduce the latency of RAM-to-CPU data transfer, or CPU cache optimisation)
- Data dependencies
  - If $c_1 = a_1 + b_1$, $c_2 = a_2 + b_2$, $d = c_1 \cdot c_2$, then, the compiler may optimise the code to compute c_1 and c_2 simultaneously

However, none of this is available in R or other high-level languages; therefore, we should work smarter.

Minimising overhead in loops

Bad approach:

```r
for (i in 1:G) {
 for (j in 1:N)  w[i, j] = k((g[i]-x[j])/b)
}
```

Good approach:

1. Define `phi`, a vectorised version of *k* via an `sapply()`;
   - Maybe *k* can even handle vectors out of the box
2. Use only `for (i in 1:G)` with `phi((g[i]-x)/b)` (since `g[i]` is a scalar and `x` is a vector)

Even better: observe that the division can be done once.

- Generate $g_b := g/b$, $X_b := X/b$, use `phi(gb[j] - xb)`

High-level speed-ups

The smartest way of doing work is… not doing work at all.

- Never do the same work twice
  - Save and reuse intermediate operations
- Optimise core functions that are called multiple times
  - Constructing a matrix quickly is more important for speed than tweaking the model solver searching for the parameter using this matrix repeatedly
- Apply better algorithms, clever maths, and **domain knowledge**
  - Always check up-to-date practices for common tasks: sorting, merging, numerical optimisation
  - Make sure that the chosen algorithm is applicable to the problem: if you are wrong, it does not matter how quickly

R-specific speed-ups

- Do not grab more data than immediately necessary
  - Work with smaller objects to save memory or simpler data types (character → factor) or use integer types
- Avoid overhead by using vectorised functions
  - Everything that happens in a loop is addled with overhead
- Parallelise (preferably at the outermost level)
- Use lists to split huge panel data sets when non-trivial per-unit operations are necessary
  - Save memory by processing independent small chunks
  - The job can be parallelised over the chunks

Look for nestable & separable operations

For a panel data set `d` with many rows, never write

```
if (d$id == i & d$time = t) ...
```

The check for `i` does not depend on the check by `t`.

If something is computed for each unit *i* (internal optimisations, extra checks, transformations), split the data into a list and use `mclapply()` / `parLapply()`.

- Is overlooked in the tidyverse functions: they discourage time- and memory-saving parallel (l)application
- *Cura te ipsum:* split the data → (l)apply any functions from any package → unsplit

HE'S THINKING ABOUT OTHER WOMEN
THERE'S GOT TO BE A WAY TO VECTORISE THAT CODE

How to balance loads?

If you have the luxury of not having to use Windows, use `mclapply` for quick tasks (spawns several processes instantly) on Unix-like systems.

For larger / non-homogeneous tasks, use `makeCluster`; supports super-computer clusters.

Demonstration: comparing 10 000 quick operations vs 20 long ones.

Benchmark first to see if overhead is a problem.

Disabling explicit load balancing

R pre-assigns n tasks to k cores ($n/k \pm 1$ tasks per core). If different evaluations of `f()` take vastly different amounts of time (and they take a long time), then, some cores may terminate earlier and be idling.

- Common case: trying a grid of parameter values (e. g. random forest depth, penalising constants) – cores with the shallowest forests return the value more quickly.

Disable load balancing to keep all CPUs occupied with the next unprocessed vector item:

```
mclapply(1:100, f, mc.cores = 8,
  mc.preschedule = FALSE)  # Default is TRUE
parLapplyLB(cl, 1:100, f)  # Instead of parLapply
```

Any questions on vectorised and parallelised operations?

Speeding up, benchmarking, profiling

Is optimisation necessary?

*We should forget about small efficiencies, say about 97% of the time: **premature optimisation is the root of all evil**. Yet we should not pass up our opportunities in that critical 3%.*

Programmers waste enormous amounts of time thinking about, or worrying about, the speed of non-critical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered.

Donald Erwin Knuth.

Measuring the execution time

Use `system.time()` to record the execution time – the input must be an expression (e. g. a function).

```
f <- function(n) sum(rep(mtcars$mpg, n))
system.time(f(1e6))
system.time(f(1e7))
```

Record the time stamps with `Sys.time()`:

```
tic0 <- Sys.time()
f(1e6)
tic1 <- Sys.time()
f(1e7)
tic2 <- Sys.time()
print(difftime(tic1, tic0, units = "sec"))
print(difftime(tic2, tic1, units = "sec"))
```

Fine micro-benchmarking

The `microbenckmark` is immensely useful for timing of very quick operations at the nano-second level.

```r
system.time(replicate(10, runif(100))) # Crude

microbenchmark::microbenchmark(
  replicate(10, runif(100)),
  matrix(runif(1e3), ncol = 100), # 0.28 s
  times = 20)  # The default is 100
```

NB. Some functions are compiled just-in-time – repeated evaluations with the same arguments may be faster due to result **memoisation** *(sic)* – saving the results of computations and recalling them without recomputation for the same inputs.

Profiling the code

Recall `traceback()`: the function stack can be very deep.

Suppose that a regression with time series is estimated:

```
set.seed(1)
n <- 1e4 # Number of observations
k <- 10  # Number of regressors
x <- matrix(runif(n*k, 0, 10), nrow = n)
y <- 1 + x[,1] + rnorm(n)*rowSums(abs(x))
mod <- lm(y ~ x)
```

Autocorrelation-robust standard errors are slow:

```
library(sandwich)
system.time(vcovHAC(mod))
```

However, `system.time(NeweyWest(mod))` is fast! Why?

Using the built-in profiler

Enable and disable the profiler:

```r
Rprof("my.txt")
kernHAC(mod)
Rprof(NULL)
summaryRprof("my.txt")
```

Visualise the calls with the `profvis` package:

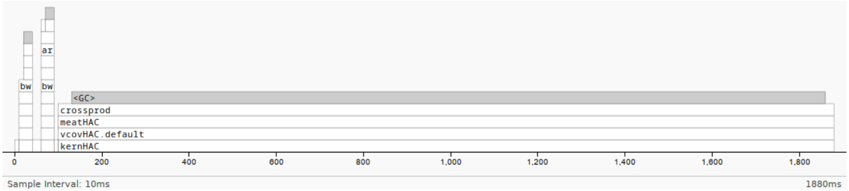
```r
profvis::profvis(kernHAC(mod))
```

Turns out, it is busy computing cross-products...

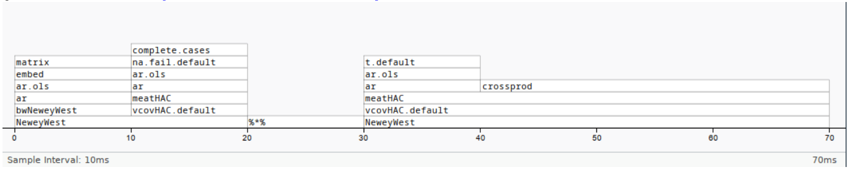
```r
profvis::profvis(NeweyWest(mod))
```

Flame graph

```
profvis::profvis(kernHAC(mod))
```



```
profvis::profvis(NeweyWest(mod))
```



This is the timeline of the functions called at all levels.

Conclusion: `kernHAC()` is too busy with cross-products.

Quenching function appetite

Without any technicalities, we cut down the amount of cross-products by using a different kernel (the default one had long ‘tails’, requiring more arithmetic operations):

```r
system.time(kernHAC(mod, kernel = "Truncated"))
```

In every application, find the slowest function and think why it is so slow.

- Maybe some operations are redundant
- If `[` is too slow, try to avoid subsetting
- Maybe there are legacy checks – speed them up

Exercise in benchmarking

Task:

1. For each individual, compute the average 2004–2007 income, $\bar{I}_i$
2. If $\bar{I}_i < 1$, replace their 2008–2019 income with **NA**

'Toy' panel data set (recall the panel from Session 2):

```
set.seed(1)
n <- 5000 # Individuals
p <- 23   # Periods
ids <- (1:n)*1000 + 1
ps  <- 2000 + 1:p
d <- data.frame(id = rep(ids, each = p),
    period = rep(ps, n), income = rchisq(n*p, 2))
d$incNA <- d$income  # To avoid overwriting
```

Conditional replacement – straightforward

```r
for (i in ids) {
  cond1  <- (d$id==i) & (d$period %in% 2004:2007)
  Ia <- mean(d$income[cond1]) # Income average
  cond2 <- (d$id==i) & (d$period %in% 2008:2019)
  if (Ia < 1) d$incNA[cond2] <- NA
}
```

This inefficient beast takes 43 seconds to run; this is why:

1. `%in% 2008:2019` is slow because it makes 12 checks (against every value: `==2008`, `==2009`, ..., `==2019`)
2. `cond2` is needed only if `Ia<1` (only ≈7% cases) – wasted $115k \times 13 \times 5000 = 7$ bn (!) checks
3. There are 115 000 rows in d; `cond1` requires $115k \times 5 \times 5000 = 3$ bn checks

Conditional replacement – speed-up 1

Replace `%in%` checks with simple `>=` and `<=`.

```r
for (i in ids) {
  cond1  <- (d$id == i) &
    (d$period >= 2004) & (d$period <= 2007)
  Ia <- mean(d$income[cond1])
  cond2 <- (d$id == i) &
    (d$period >= 2008) & (d$period <= 2019)
  if (Ia < 1) d$incNA[cond2] <- NA
}
```

Now this takes only 14 s – quick, easy, low-hanging fruit.

Conditional replacement – speed-up 2

Checking `d$id == i` twice is redundant – save and reuse the result of this evaluation.

```r
for (i in ids) {
  cond.id <- d$id == i  # Saved this
  cond1  <- cond.id &
    (d$period >= 2004) & (d$period <= 2007)
  Ia <- mean(d$income[cond1])
  cond2 <- cond.id &
    (d$period >= 2008) & (d$period <= 2019)
  if (Ia < 1) d$incNA[cond2] <- NA
}
```

Now this takes only 13 s – not much, but we can do better.

Conditional replacement – speed-up 3

The replacement based on `cond2` is not needed if `Ia<1`.

Therefore, `cond2` should be computed only if `Ia < 1`:

```
for (i in ids) {
  cond.id <- d$id == i
  cond1  <- cond.id &
    (d$period >= 2004) & (d$period <= 2007)
  Ia <- mean(d$income[cond1])
  if (Ia < 1) {
    cond2 <- cond.id & # <-- Moved this part
      (d$period >= 2008) & (d$period <= 2019)
    d$incNA[cond2] <- NA
  }
}
```

Now this takes only 8.5 s – a substantial improvement.

Refactoring ugly lines (same speed)

1. cond1 and cond2 are too long – pre-compute the period conditions and use (id & period) on the fly
2. Writing `d$period` every time < short name (p)

```r
for (i in ids) {
  p <- d$period
  cond.id <- d$id == i
  cond.p1 <- (p >= 2004) & (p <= 2007)
  Ia <- mean(d$income[cond.id & cond.p1])
  if (Ia < 1) {
    cond.p2 <- (p >= 2008) & (p <= 2019)
    d$incNA[cond.id & cond.p2] <- NA
  }
}
```

Conditional replacement – speed-up 4

Create a smaller DF (`di`) for each `i` to slash redundancy.

```r
for (i in ids) {
  cond.id <- d$id == i
  di <- d[cond.id, ] # A smaller object
  p <- di$period     # Now based on di
  cond.p1 <- (p >= 2004) & (p <= 2007)
  Ia <- mean(di$income[cond.p1]) # No more cond.id
  if (Ia < 1) {
    cond.p2 <- (p >= 2008) & (p <= 2019)
    di$incNA[cond.p2] <- NA # No more cond.id
    d[cond.id, ] <- di # Put changes back
  }
}
```

Time: 6.7 s. In practice, with large data sets, **this trick may yield big speed-ups and memory economy**.

Conditional replacement – speed-up 5?

Notice how `d$period >= 2004` does not depend on `i` – one initial creation of helper variables is enough!

```
d$p1 <- d$period >= 2004 & d$period <= 2007
d$p2 <- d$period >= 2008 & d$period <= 2019
for (i in ids) {
  cond.id <- d$id == i
  di <- d[cond.id, ]
  Ia <- mean(di$income[di$p1]) # No more cond.id
  if (Ia < 1) {
    di$incNA[di$p2] <- NA # No more cond.id
    d[cond.id, ] <- di
  }
}
```

Time: 8.5 s. What went wrong?

Analysis of speed-up 5 failure

In some applications, working on data subframes is indeed faster (otherwise, this example would not be here – the perfidy of toy data sets!).

`di <- d[cond.id, ]` adds a new costly operation that had not existed before – row subsetting for the full DF.

- Creating `p1` and `p2` increased `ncol(d)` to 6 – the DF simply became bulkier (but this is negligible)
- Eliminate this redundant copying – we are only using a subset of `d$income` and writing to a subset of `d$incNA`
- In applicatons with more complex operations for each subset `di` (e. g. nested computations in multi-level panels), the gains owing to subsets can be real

Conditional replacement – speed-up 6

Create helper indicator variables once, but do not copy `d[cond.id, ]`.

```r
d$p1 <- d$period >= 2004 & d$period <= 2007
d$p2 <- d$period >= 2008 & d$period <= 2019
for (i in ids) {
  cond.id <- d$id == i
  Ia <- mean(d$income[cond.id & d$p1])
  if (Ia < 1) d$incNA[cond.id & d$p2] <- NA
}
```

Time: 3.8 s. Huge savings (because each object in the evaluation is a vector, no matrices) + elegant code.

Splitting data frames into lists

Often, long logical vectors for condition-based operations are too slow. Splitting the data into a *list* of smaller data sets may offer substantial gains.

Example above: instead of checking `d$id == i` in a loop, split the data into smaller chunks and loop over them.

```r
dlist <- split(d, f = d$id)
for (i in 1:length(dlist)) {
  x <- dlist[[i]]
  Ia <- mean(x$income[x$p1])
  if (Ia < 1) dlist[[i]]$incNA[x$p2] <- NA
}
d <- do.call(rbind, dlist)
```

Time savings owing to splits

The example above is blazingly fast: 0.95 s.

- Overhead: 0.4 s to split the DF + 0.5 s to `rbind` the list
- …and the pure calculations take **0.035 s**

If multiple operations are carried out according to a condition, the only time losses are due to splitting and binding (and those are tiny compared to the redundant subsetting slow-down).

Core functionality speed-up: 43 s $\to$ 0.035 s = 1200 times!

Merging lists back into data frames

`rbind()` / `cbind()` accept multiple input vectors / matrices to ‘fuse’ them together (resp. vertically / horizontally).

```
byvar <- rep(1:4, each = 8)
a <- split(mtcars, f = byvar)
str(a, max.level = 1)
```

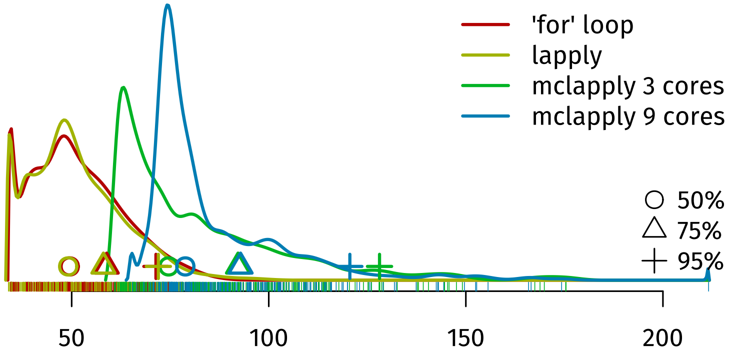
```
#> List of 4
#>  $ 1:'data.frame':       8 obs. of  11 variables:
#>  <...>
#>  $ 4:'data.frame':       8 obs. of  11 variables:
```

```
d1 <- do.call(rbind, unname(a))
d2 <- unsplit(a, f = byvar)
all.equal(d1, d2)  # TRUE
```

NB. `split()` gives names to the resulting list a – `unname(a)` is necessary to prevent the rownames from becoming ‘1.Mazda RX4’, ‘2.Merc 230’ etc.

Overhead with small operations

In this example, 5000 operations took 0.035 s → 7 μs each → parallel scheduling overhead can outweigh the gains.



Overhead: 60–90 ms. Compare to 1 function evaluation!

Any questions on the bottleneck-finding methodology?

Further reading

- How one programmer reverse-engineered GTA V and made it load 7 times faster

Thank you for your attention!