Working with 

A University of Queensland Advanced Workshop

Session 11: Introduction to S4

Bill Venables, CSIRO/Data61, Dutton Park

Rhetta Chappell, Griffith University

2–5 February, 2021

Contents

1 Limitations of S3

- S3 *classes* exist only in the mind of the programmer. This is a severe programming limitation.
- S3 *generic* functions dispatch methods based on the (possibly induced) class of the *principal* argument only.
- S3 *method functions* are not explicitly associated with any generic function, but rather loosely linked using a weak and fallible naming convention.
- Nevertheless, in practice, for most purposes S3 classes and methods are both easy to write, easy to use and work well!

2 Example: Two sample t–test, S3 version

```
TTest <- function(x, ...) {
  UseMethod("TTest")
}
TTest.numeric <- function(x, y, ...) {  ## the work horse
  stopifnot(is.numeric(y))
  nx <- length(x)
  ny <- length(y)
  m <- mean(x) - mean(y)
  s2 <- (sum((x - mean(x))^2) + sum((y - mean(y))^2))/(nx+ny-2)
  sd <- sqrt(s2*(1/nx + 1/ny))
  res <- within(list(tstat = m/sd, df = nx+ny-2), {
    p.value <- 2*pt(-abs(tstat), df)
  })
  class(res) <- "TTest"
  res
}
TTest.integr <- TTest.numeric
```

```r
TTest.factor <- function(x, y, ...) {
  stopifnot(is.numeric(y), length(levels(x)) == 2)
  x1 <- y[x == levels(x)[1]]
  x2 <- y[x == levels(x)[2]]
  TTest.numeric(x1, x2, ...)
}

TTest.character <- function(x, y, ...) {
  TTest.factor(factor(x), y, ...)
}

TTest.formula <- function(x, data, ...) {
  lhs <- eval(x[[2]], envir = as.environment(data))
  rhs <- eval(x[[3]], envir = as.environment(data))
  TTest.factor(as.factor(rhs), lhs, ...)
}
```

Testing:

```
TTest(Days ~ Eth, quine)

$tstat
[1] 3.485725

$df
[1] 144

$p.value
[1] 0.0006508376

attr(,"class")
[1] "TTest"
```

```
t.test(Days ~ Eth, quine, var.equal = TRUE)[cs(statistic, parameter)]

$statistic
       t
3.485725

$parameter
 df
144
```

A *print* method

```
find("print")
```

```
[1] "package:base"
```

```
print
```

```
function (x, ...)
UseMethod("print")
<bytecode: 0x55ad40ea6bc0>
<environment: namespace:base>
```

```
print.TTest <- function(x, ...) {
  with(x, cat("t = ", tstat,
              ", df = ", df,
              ", p = ", p.value, "\n", sep = ""))
  invisible(x)
}
```

```
TTest(Days ~ Eth, quine)
```

```
t = 3.485725, df = 144, p = 0.0006508376
```

3 Example: Two sample $t-$test, S4 version

First we should define a new class.

```
setClass("t_test", representation(x = "numeric",
                                  y = "numeric",
                                  t = "numeric",
                                  df = "integer",
                                  p = "numeric"))
```

Next, define a generic function to construct objects of the new class:

```
setGeneric("t_test", function(x, y) {
  standardGeneric("t_test")
})
```

```
[1] "t_test"
```

A new function has been created but as yet does nothing

The print method for such functions can be somewhat wordy:

```
t_test

nonstandardGenericFunction for "t_test" defined from package ".GlobalEnv"


function (x, y)
{
    standardGeneric("t_test")
}
<environment: 0x55ad47b0c400>
Methods may be defined for arguments: x, y
Use  showMethods("t_test")  for currently available ones.
```

Now to set a few methods so that we can use it:

```
setMethod("t_test", signature(x = "numeric", y = "numeric"),
          function(x, y) {
            tt <- stats::t.test(x, y, var.equal = TRUE) ## cheating!
            with(tt, new("t_test", x = x, y = y,
                         t = statistic,
                         df = as.integer(parameter),
                         p = p.value))
})
```

Other useful methods will essentially divert to this one, but in two ways

```
setMethod("t_test", signature(x = "factor", y = "numeric"),
          function(x, y) {
            hold <- y
            lev <- levels(x)
            y <- hold[x == lev[2]]
            x <- hold[x == lev[1]]
            callNextMethod(x, y)
          })

## If you put them in the wrong order...
setMethod("t_test", signature(x = "numeric", y = "factor"),
          function(x, y)
            callGeneric(y, x))

setMethod("t_test", signature(x = "formula", y = "data.frame"),
          function(x, y) {
            v <- eval(x[[2]], envir = y) ## the values
            f <- eval(x[[3]], envir = y) ## the factor
            callGeneric(f, v)
          })
```

Where are we at so far?

```
showClass("t_test")
```

```
Class "t_test" [in ".GlobalEnv"]

Slots:

Name:        x       y       t      df       p
Class: numeric numeric numeric integer numeric
```

```
showMethods("t_test")
```

```
Function: t_test (package .GlobalEnv)
x="factor", y="numeric"
x="formula", y="data.frame"
x="numeric", y="factor"
x="numeric", y="numeric"
```

At first test:

```
t_test(Days ~ Eth, quine)

An object of class "t_test"
Slot "x":
 [1]  2 11 14  5  5 13 20 22  6  6 15  7 14  6 32 53 57 14 16 16 17 40 43
[24] 46  8 23 23 28 34 36 38  3  5 11 24 45  5  6  6  9 13 23 25 32 53 54
[47]  5  5 11 17 19  8 13 14 20 47 48 60 81  2  0  2  3  5 10 14 21 36 40

Slot "y":
 [1]  6 17 67  0  0  2  7 11 12  0  0  5  5  5 11 17  3  4 22 30 36  8  0
....
       t
3.485725

Slot "df":
[1] 144

Slot "p":
[1] 0.0006508376
```

Ugh!

The standard S4 display function for objects is *show* (rather than *print*), which has just one argument, *object*:

```
show
```

```
standardGeneric for "show" defined from package "methods"


function (object)
standardGeneric("show")
<bytecode: 0x55ad429ec300>
<environment: 0x55ad4186a9d8>
Methods may be defined for arguments: object
Use  showMethods("show")  for currently available ones.
(This generic function excludes non-simple inheritance; see ?setIs)
```

```
setMethod("show", signature(object = "t_test"),
          function(object) {
            cat("t = ", signif(object@t, digits = 3),
                ", df = ", object@df,
                ", p = ", signif(object@p, digits = 2), "\n", sep="")
            invisible(object)
})
t_test(Days ~ Eth, quine)  ## that's better!
```

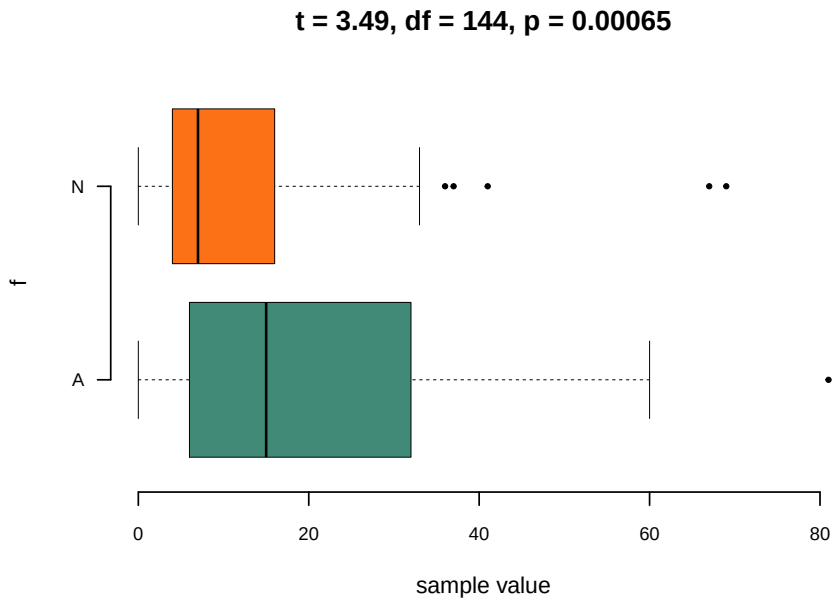
```
t = 3.49, df = 144, p = 0.00065
```

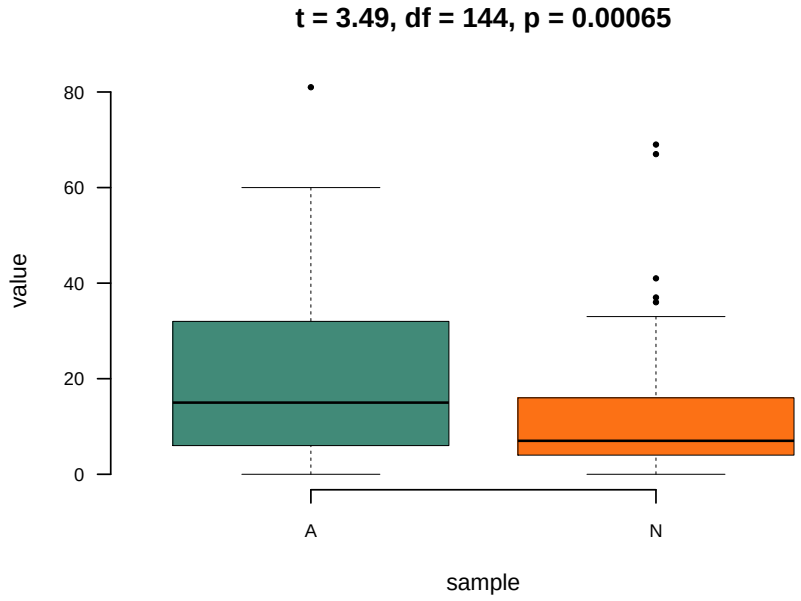
Since we have the data squirreled away, why not provide a graphical display as well?

```
setGeneric("plot", graphics::plot)  ## technically unneeded.  If omitted ..
```

```
[1] "plot"
```

```
setMethod("plot", signature(x = "t_test"),  ## .. this -> implicit generic
          function(x, ..., col = c("#418A78", "#FC7115"), main = top,
                   horizontal = TRUE, border = "black", lwd = 0.5,
                   xlab = if(horizontal) "sample value" else "sample") {
            dat <- rbind(data.frame(f = "x", value = x@x),
                         data.frame(f = "y", value = x@y))
            top <- capture.output(show(x))
            oldPar <- par(cex.axis = 0.8, cex = 0.8, bty = "n")
            on.exit(par(oldPar))
            graphics::boxplot(value ~ f, dat, col = col, lwd = lwd,
                              horizontal = horizontal, border = border,
                              main = main, xlab = xlab, ...)
            invisible(x)
          })
t0 <- t_test(Days ~ Eth, quine)
plot(t0, names = levels(quine$Eth))
plot(t0, names = levels(quine$Eth), horizontal = FALSE)
```


t = 3.49, df = 144, p = 0.00065
N
A
f
0
20
40
60
80
sample value


t = 3.49, df = 144, p = 0.00065
80
60
40
20
0
value
A
N
sample

4 Case study: A flexible *arrows* function

The WWRGraphics package has a version of *arrows* that extends *graphics::arrows* using S4 classes and methods.

A class union:

```
#' An S4 class to represent alternavive complex, matrix or list input forms.
#' @export
setClassUnion("xy", c("complex", "matrix", "list"))
```

A standard generic function:

```
#' Arrows
#'
#' Front end to the \code{graphics::arrows} function allowing
#' simplified specification of end points
#'
#' @param x0,y0 two numeric vectors of a single object of class \code{"xy"}
#' @param x1,y1 two numeric vectors of a single object of class \code{"xy"}
#' @param length length of the arrow head barb, in MILLIMETRES,
#'        see \code{\link{in2mm}}
#' @param angle as for \code{graphics::arrows}
```

```
#' @param gap numeric of length 1 or 2 giving the size of any gap to be left
#'        between the arrow and the points which it connects, in MILLIMETRES
#' @param circular logical: should the arrows link up with the initial point?
#'        (Single location argument only.)
#' @param ... additional arguments passed on to \code{graphics::segments}
#'
#' @return invisible null value
#' @export
#' @examples
#' z <- with(roundTrip, setNames(complex(real = Longitude,
#'                                       imaginary = Latitude), Locality))
#' plot(z, asp = 1, pch = 20, cex = 0.7, xlab = "Longitude", ylab = "Latitude")
#' arrows(z, cyc(z), col = "red", gap = c(0,1.5))
setGeneric("arrows", function(x0, y0, ...)
  standardGeneric("arrows"))
```

So far, so good. Now for a few methods.

```
#' @rdname arrows
#' @export
setMethod("arrows", signature(x0 = "numeric", y0 = "numeric"),
          function(x0, y0, x1, y1, length = 4, angle = 15, gap, ...) {
            length <- mm2in(length)
            if(!missing(gap)) {
              stopifnot(is.numeric(gap) && length(gap) > 0 &&
                          length(gap) < 3 && all(gap >= 0))
              gap <- rep_len(mm2in(gap), length.out = 2)
              if(any(gap > 0)) {
                z0 <- usr2in(x0, y0); z1 <- usr2in(x1, y1)
                gp0 <- gp1 <- z1 - z0
                Mod(gp0) <- pmin(gap[1], Mod(gp0)/3)
                Mod(gp1) <- pmin(gap[2], Mod(gp1)/3)
                xy0 <- in2usr(z0 + gp0); xy1 <- in2usr(z1 - gp1)
                x0 <- Re(xy0); y0 <- Im(xy0)
                x1 <- Re(xy1); y1 <- Im(xy1)
              }
            }
            graphics::arrows(x0 = x0, y0 = y0, x1 = x1, y1 = y1,
                             length = length, angle = angle, ... )
})
```

This method allows us to call *arrows* with two main arguments rather than four:

```
#' @rdname arrows
#' @export
setMethod("arrows", signature(x0 = "xy", y0 = "xy"),
          function(x0, y0, ...) {
            xy0 <- grDevices::xy.coords(x0)
            xy1 <- grDevices::xy.coords(y0)
            callGeneric(xy0$x, xy0$y, xy1$x, xy1$y, ...)
          })
```

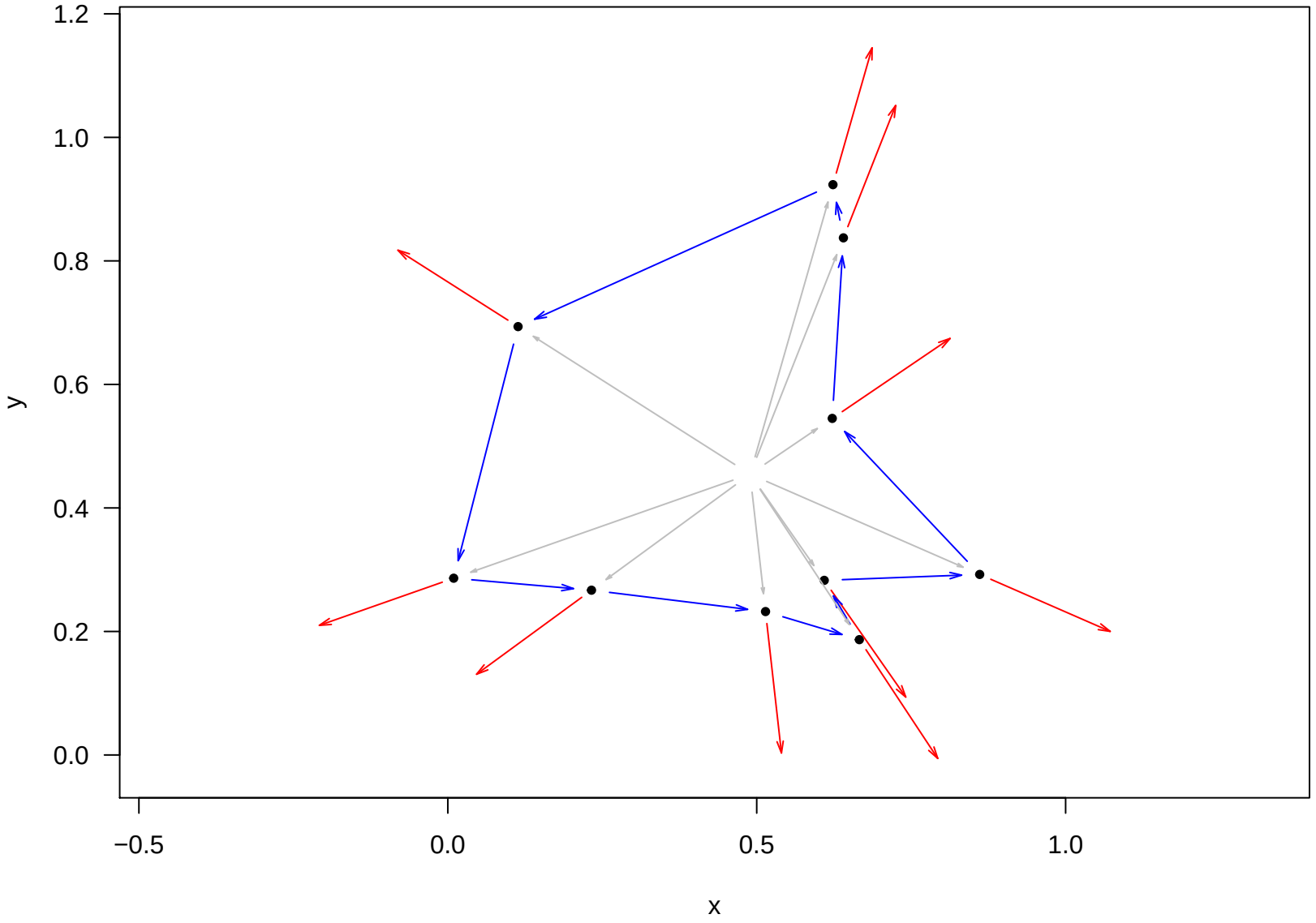
Finally, a method that allows us to call *arrows* with one argument only:

```r
#' @rdname arrows
#' @export
setMethod("arrows", signature(x0 = "xy", y0 = "missing"),
          function(x0, y0, ..., circular = FALSE) {
            z <- with(grDevices::xy.coords(x0), complex(real = x, imaginary = y))
            if(length(z) < 2) {
              return(invisible(z))
            }
            if(circular) {
              z0 <- z
              z1 <- c(z[-1], z[1])
            } else {
              z1 <- z[-1]
              z0 <- z[-length(z)]
            }
            callGeneric(Re(z0), Im(z0), Re(z1), Im(z1), ...)
          })
```

4.1 Examples

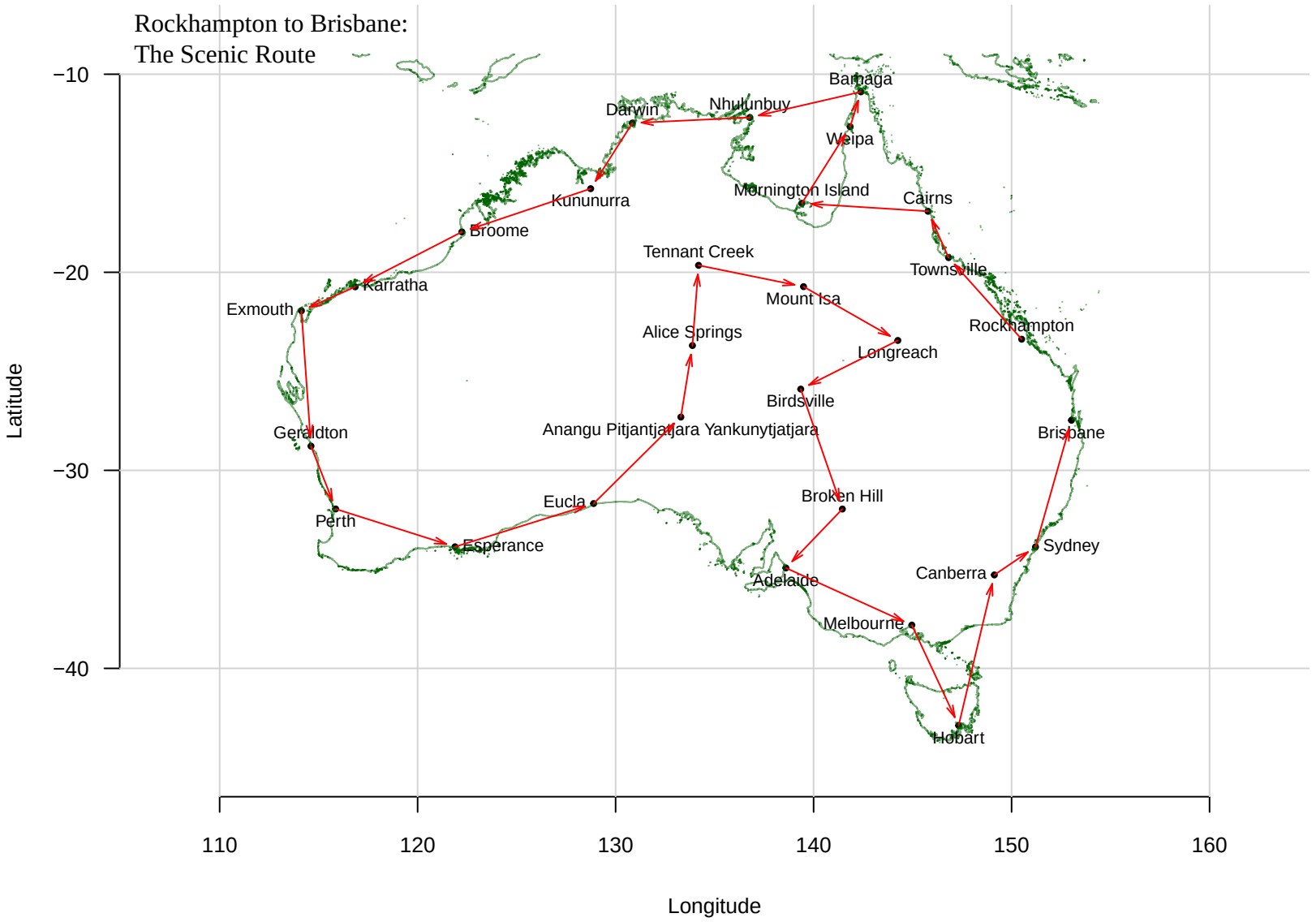
First some simple ones:

```
set.seed(1234)
n <- 10
z <- complex(real = runif(n), imaginary = runif(n))
z <- z[order(Arg(z - mean(z)))]
z1 <- complex(argument = Arg(z - mean(z)))/4
plot(c(z, z+z1), asp = 1, type = "n", xlab = "x", ylab = "y")
points(z)
arrows(z, z+z1, col = "red", gap = 2, length = 2)
arrows(z, col = "blue", circular = TRUE, gap = 3, length = 2)
arrows(mean(z), z, gap = 3, length = 1, col = "grey")
```


1.2
1.0
0.8
0.6
0.4
0.2
0.0
y
−0.5
0.0
0.5
1.0
x

A round trip.

```
z <- with(roundTrip, setNames(complex(real = Longitude,
                                      imaginary = Latitude),
                              Locality))
par(cex.axis = 0.8, cex.lab = 0.8, font.main = 1)
plot(z, asp = 1, pch = 20, cex = 0.7,
     xlim = c(110, 160), ylim = c(-45, -8),
     xlab = "Longitude", ylab = "Latitude", bty = "n")
grid(lty = "solid")
lines(Oz, col = alpha("darkgreen", 0.5))
text(z, names(z), pos = avoid(z), offset = 0.25, cex = 0.7)
arrows(rev(z), col = "red", gap = c(0,1.5), length = 2)
text("top left", c("Rockhampton to Brisbane:",
                   "The Scenic Route"), family = "serif")
```


Rockhampton to Brisbane:
The Scenic Route
Latitude
Longitude
−10
−20
−30
−40
110
120
130
140
150
160
Darwin
Nhulunbuy
Barnaga
Weipa
Mornington Island
Cairns
Kununurra
Broome
Tennant Creek
Townsville
Karratha
Exmouth
Mount Isa
Alice Springs
Rockhampton
Longreach
Birdsville
Geraldton
Anangu Pitjantjatjara Yankunytjatjara
Brisbane
Eucla
Broken Hill
Perth
Esperance
Sydney
Canberra
Adelaide
Melbourne
Hobart

Session information

Date: 2021-01-29

- R version 4.0.3 (2020-10-10), `x86_64-pc-linux-gnu`
- Running under: `Ubuntu 20.04.1 LTS`
- Matrix products: default
- BLAS: `/usr/lib/x86_64-linux-gnu/blas/libblas.so.3.9.0`
- LAPACK: `/usr/lib/x86_64-linux-gnu/lapack/liblapack.so.3.9.0`
- Base packages: base, datasets, graphics, grDevices, methods, stats, utils
- Other packages: dplyr 1.0.3, english 1.2-5, forcats 0.5.1, ggplot2 3.3.3, ggthemes 4.2.4, gridExtra 2.3, knitr 1.31, lattice 0.20-41, patchwork 1.1.1, purrr 0.3.4, readr 1.4.0, scales 1.1.1, stringr 1.4.0, tibble 3.0.5, tidyr 1.1.2, tidyverse 1.3.0, WWRCourse 0.2.3, WWRData 0.1.0, WWRGraphics 0.1.2, WWRUtilities 0.1.2, xtable 1.8-4
- Loaded via a namespace (and not attached): assertthat 0.2.1, backports 1.2.1, broom 0.7.3, cellranger 1.1.0, cli 2.2.0, colorspace 2.0-0, compiler 4.0.3, crayon 1.3.4, DBI 1.1.1, dbplyr 2.0.0, ellipsis 0.3.1, evaluate 0.14, fansi 0.4.2, farver 2.0.3, fractional 0.1.3, fs 1.5.0, generics 0.1.0, glue 1.4.2, grid 4.0.3, gtable 0.3.0, haven 2.3.1, highr 0.8, hms 1.0.0, httr 1.4.2, iterators 1.0.13, jsonlite 1.7.2, lazyData 1.1.0, lifecycle 0.2.0, lubridate 1.7.9.2, magrittr 2.0.1, MASS 7.3-53, modelr 0.1.8, munsell 0.5.0, parallel 4.0.3, PBSmapping 2.73.0, pillar 1.4.7, pkgconfig 2.0.3, R6 2.5.0, randomForest 4.6-14, Rcpp 1.0.6, readxl 1.3.1, reprex 1.0.0, rlang 0.4.10, rpart 4.1-15, rstudioapi 0.13, rvest 0.3.6, SOAR 0.99-11, splines 4.0.3, stringi 1.5.3, tidyselect 1.1.0, tools 4.0.3, vctrs 0.3.6, withr 2.4.1, xfun 0.20, xml2 1.3.2