Working with 

A University of Queensland Advanced Workshop

Session 15:
Recursive Programming: Solving Sudoku in R

Bill Venables, CSIRO/Data61, Dutton Park

Rhetta Chappell, Griffith University

2–5 February, 2021

Contents

1 Environments

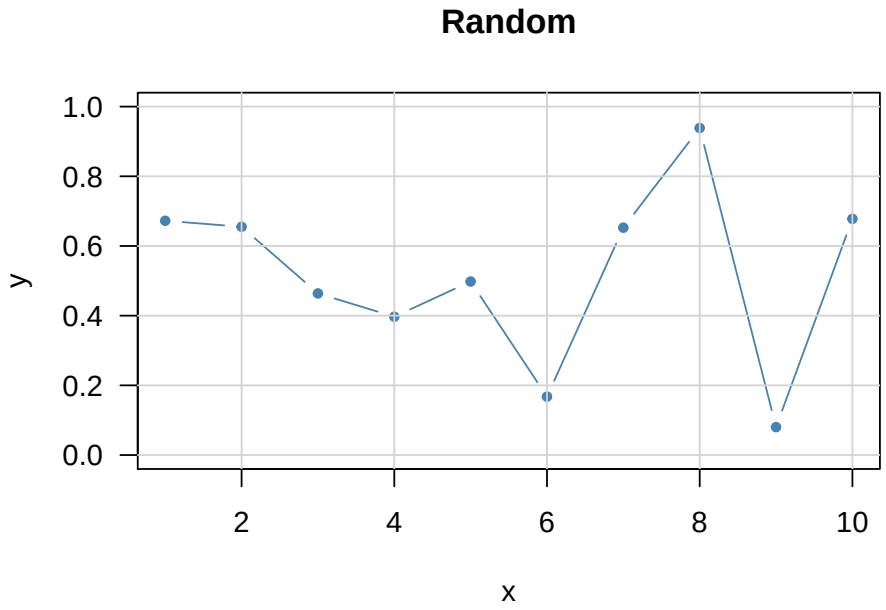
- An *unordered collection* of name-value pairs, providing a *primary context* for an operation or computation in **R**. They are objects in their own right, and may be manipulated as such.
- They are like `lists`, in that their values may be extracted or replaced by *name* using either the `$name` or `[["name"]]` operators, (but *not* by numerical indices!).
- They are *unlike* lists in that they are *not copied* on assignment. This makes them tricky!
- They are internally represented a hash tables, making them very efficient for access and replacement.
- Most junior **R** programmers are unaware of the concept, usually without negative results.

"Hello world" examples.

```
e1 <- new.env()
e1$x <- 1:10
e1$y <- runif(10)
ls(envir = e1)
```

```
[1] "x" "y"
```

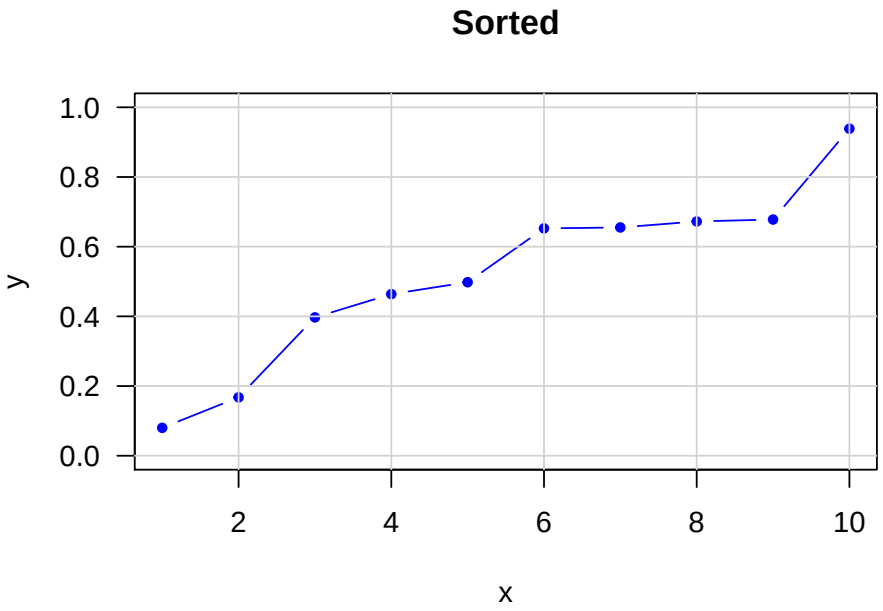
```
evalq(plot(x, y, type = "b", ylim = 0:1,
           col = "steel blue", main = "Random"), e1); grid(lty="solid")
```



```r
e2 <- e1
e2$y <- sort(e2$y)
form <- y ~ x
environment(form) <- e1  ## yes, e1.  This is not a typo ...
form  ## enclosing environment given an address, not a name.
```

```
y ~ x
<environment: 0x563ccf044228>
```

```r
plot(form, ylim=0:1, col="blue", type="b", main="Sorted"); grid(lty="solid")
```



An example. The Box-Cox family of transformations.

$$y^{(\lambda)} = \begin{cases} \frac{y^\lambda - 1}{\lambda} & \text{if } \lambda \neq 0 \\ \log y & \text{if } \lambda = 0 \end{cases}$$

```
tBoxCox <- local({  ## An explicit enclosing environment...
  t1 <- function(y, lambda) (y^lambda - 1)/lambda ## |lambda| > eps
  t2 <- function(y, lambda) {  ## |lambda| <= eps, use power series...
    logy <- log(y)
    ly <- lambda*logy
    logy*(1 + ly/2*(1 + ly/3*(1 + ly/4*(1 + ly/5))))
  }
  function(y, lambda, eps = 1/250)
      with(data.frame(y = y, lambda = lambda),  ## ensure equal length
           ifelse(abs(lambda) > eps, t1(y, lambda), t2(y, lambda)))
})
```

Application: simultaneous testing of transformations

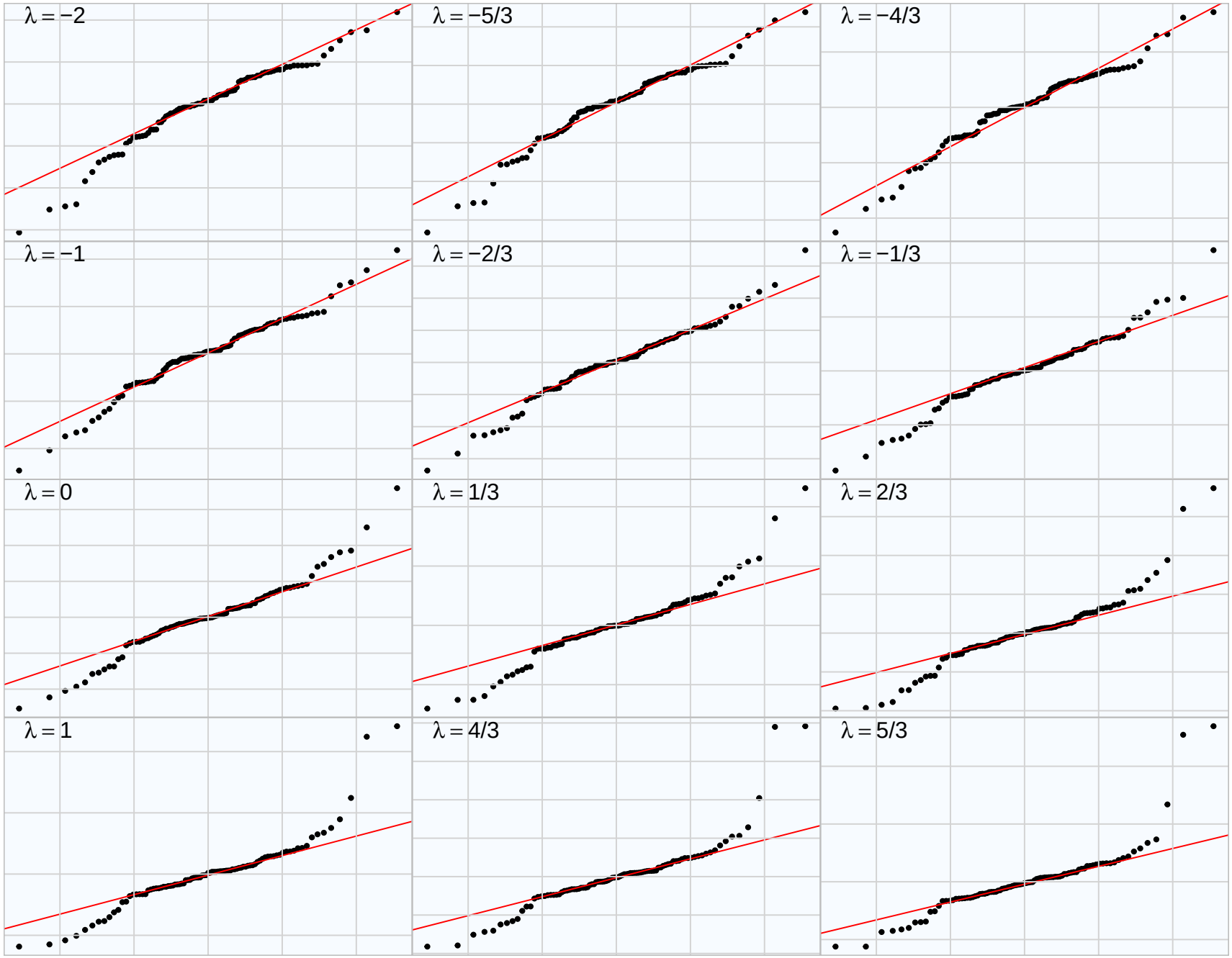
```
library(fractional)
(lambda <- seq(-2, 5/3, length.out = 12))
```

```
 [1] -2.0000000 -1.6666667 -1.3333333 -1.0000000 -0.6666667 -0.3333333
 [7]  0.0000000  0.3333333  0.6666667  1.0000000  1.3333333  1.6666667
```

```
tst <- lm(outer(MPG.city, lambda, tBoxCox) ~ Weight+Type+Origin, Cars93)
dim(rs <- resid(tst))  ## residual matrix
```

```
[1] 93 12
```

```
par(mfrow=n2mfrow(length(lambda)), mar=rep(0,4), bg=alpha("alice blue", 0.5))
for(j in seq_along(lambda)) {
  qqnorm(rs[,j], axes = FALSE, ann = FALSE)
  qqline(rs[,j], col = "red")
  grid(lty = "solid")
  par(usr=c(0,1,0,1))
  text(0.05, 0.95, bquote(lambda == .(as.character(fractional(lambda[j])))),
       cex = 1.5, adj = 0)
  box(col = "grey")
}
```


λ = −2
λ = −5/3
λ = −4/3
λ = −1
λ = −2/3
λ = −1/3
λ = 0
λ = 1/3
λ = 2/3
λ = 1
λ = 4/3
λ = 5/3

Linear and spline interpolation.

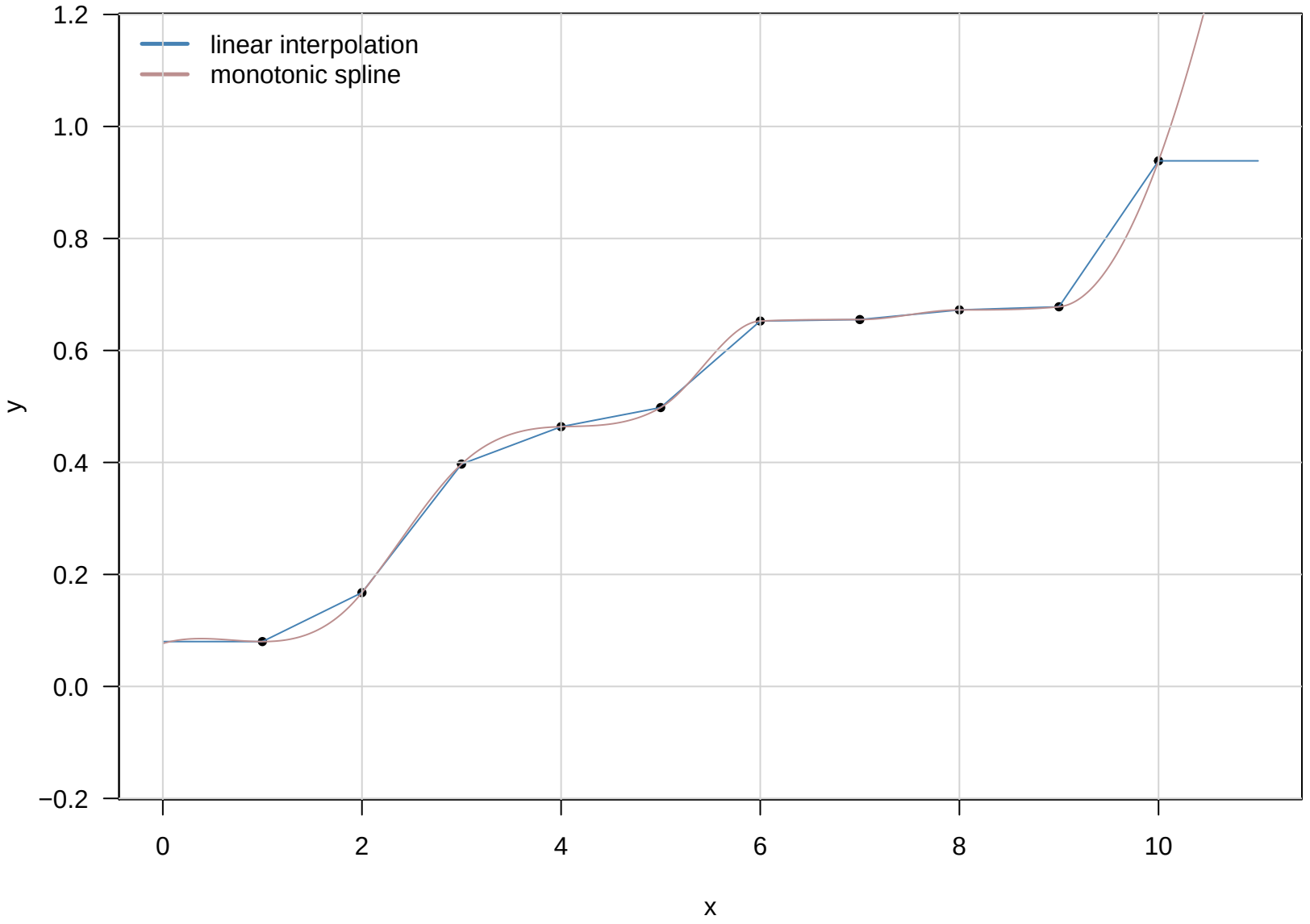
```
with(e1, {  ## works with environments too!
  linInt <<- approxfun(x, y, rule = 2)  ## "flat" extrapolation
  splInt <<- splinefun(x, y, method = "hyman") ## monotonic spline
})
linInt
```

```
function (v)
.approxfun(x, y, v, method, yleft, yright, f, na.rm)
<bytecode: 0x563cd08e9810>
<environment: 0x563cd08e8e70>
```

```
ls(envir = environment(linInt))  ## in the sack...
```

```
[1] "f"      "method" "na.rm"  "x"      "y"      "yleft"  "yright"
```

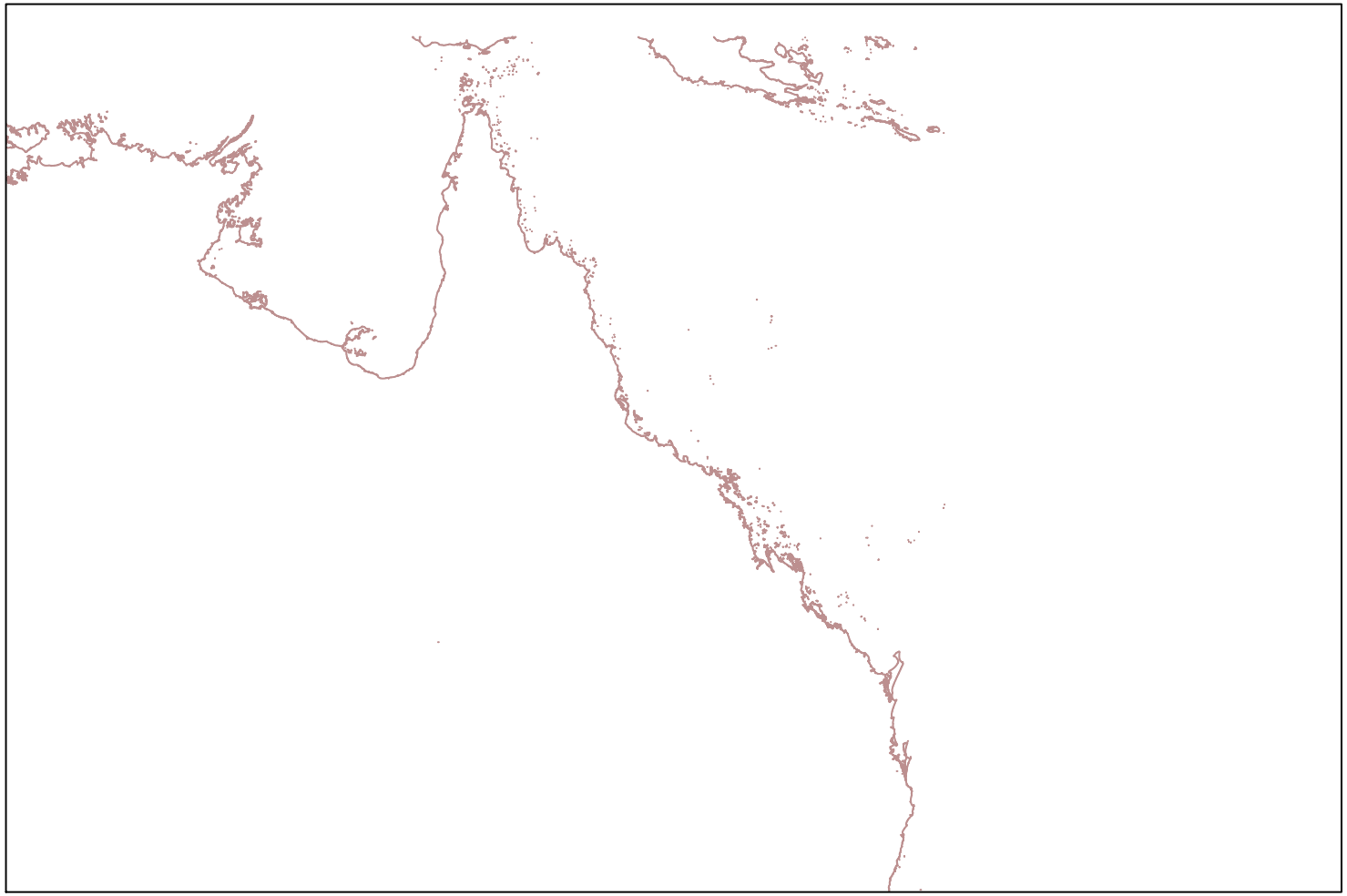
```
plot(form, xlim = c(0, 11), ylim = c(-0.15, 1.15))
curve(linInt, add = TRUE, col="steel blue", n = 500)
curve(splInt, add = TRUE, col="rosy brown", n = 500)
legend("topleft", c("linear interpolation", "monotonic spline"),
       bty = "n", lty = "solid", lwd = 2.5,
       col = c("steel blue", "rosy brown"))
grid(lty="solid")
```


linear interpolation
monotonic spline
1.2
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
y
0
2
4
6
8
10
x

A more practical use: storing data with a function.

```
str(Oz)

List of 2
 $ x: num [1:439470] NA 117 117 117 117 ...
 $ y: num [1:439470] NA -35 -35 -35 -35 ...

Aus <- local({
  Oz <- Oz  ## keep a copy here!

  function (add = FALSE, xlim = c(112.913865, 154.417058),
                         ylim = c(-43.738846, -9),
            xlab = "", ylab = "", axes = FALSE, ...) {
### the xy-list 'Oz' is held in the local environment of this function
    if (!add) {
      plot(xlim, ylim, asp = 1, type = "n", xlab = xlab, ylab = ylab,
           axes = axes, ...)
    }
    lines(Oz, ...)
  }
})

ls(envir = environment(Aus))
```

```
[1] "Oz"

english(object.size(Aus))                    ## not the full story!

twelve thousand five hundred and thirty-six bytes

english(object.size(serialize(Aus, NULL))) ## more honest!

seven million forty-eight thousand two hundred and forty-eight bytes

## what it actually does!
Aus(xlim = c(140, 155), ylim = c(-30, -9), col = "rosy brown")
box()
```



Super-assignment

- The <<- assignment operator, assigns *outside* the environment in which it is invoked[a].
- An assignment such as

```
x <<- sqrt(y)
```

  would
  - look for `y` in the current environment (and from there back though its *lexical scoping*) to form the square roots, but
  - look for an `x` *outside* the current environment to put the result.

    If it found none, the value would be put in the *working environment* (making it dangerous!)

[a] This makes it a side-effect, and side-effects are potentially dangerous.

```
fn <- local({
  x <- 0      ## to receive
  n <- 0      ## super-assigned values
  function(y = x) {
    x <<- y
    n <<- n + 1
    return(c(No = n, Value = x, Square = y^2))
  }
})
fn()
```

```
    No  Value Square
     1      0      0
```

```
fn(3:5)
```

```
     No  Value1  Value2  Value3 Square1 Square2 Square3
      2       3       4       5       9      16      25
```

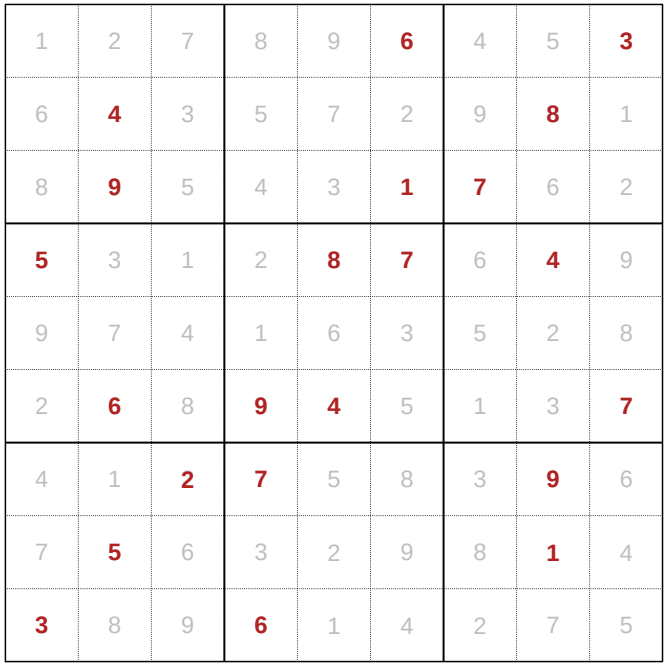
```
fn()
```

```
     No  Value1  Value2  Value3 Square1 Square2 Square3
      3       3       4       5       9      16      25
```

2 How to spoil a Sudoku puzzle using R

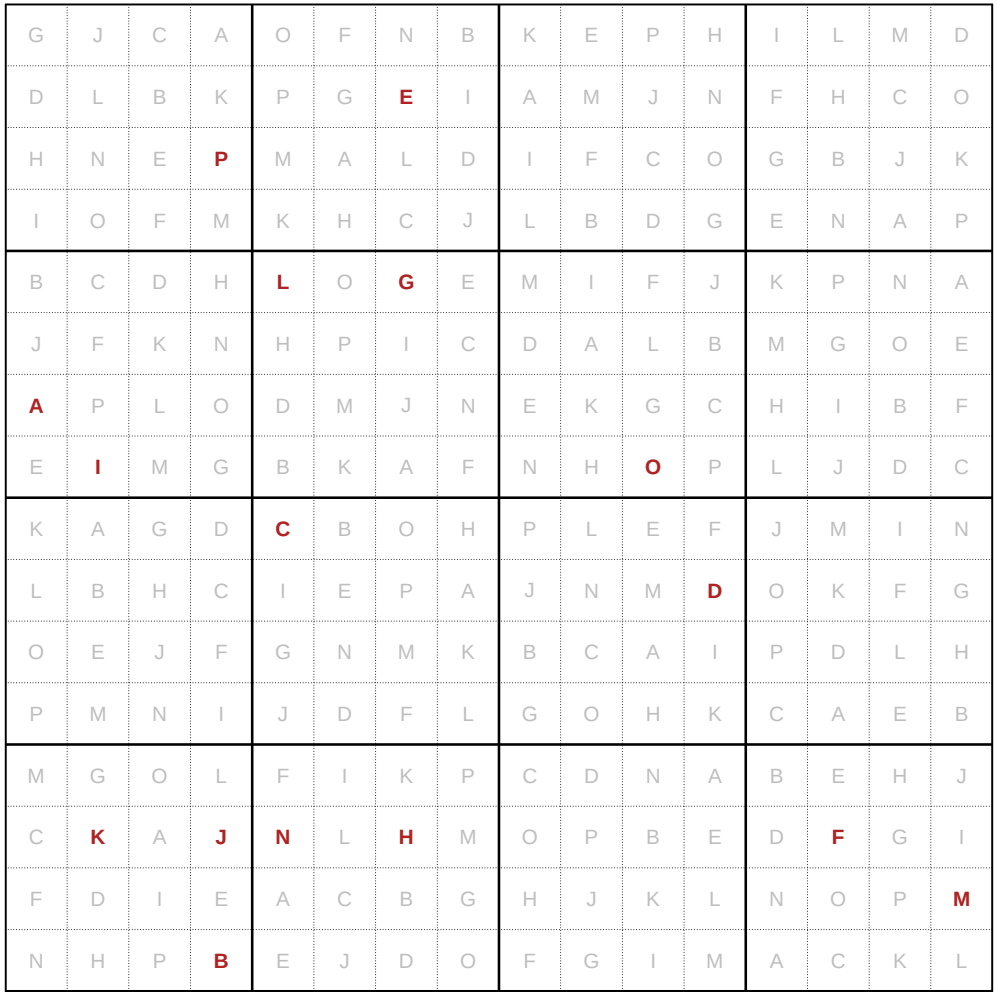
Example:

```
library(sudokuAlt)
fetchAUGame(difficulty = "tough") %>% solve %>% plot
```



A bigger example, $n = 4$. This is a generalized Latin Square design:

```
seedGame(4) %>% solve %>% plot
```



Strategy:

- Represent the game externally as an $n^2 \times n^2$ matrix, numeric or character, but internally as a *numerical* array of dimension $n \times n \times n \times n$, with empty cells `NA`.

  Set up tools for converting from one to the other.

- Set up tools for checking if the *Sudoku* constraints have been violated (for any complete game, or any tentative solution).

- The main process is recursive. The function `findSolution` is given a progressively solved game and does the following:
  - Check the game for validity. If invalid, return `FALSE`, otherwise continue.
  - Check if the game has any missing cells. If not, return it as the solution. Otherwise:
    * Check each missing cell for possible entries, and count them.
    * If any cell has *no* possible entry, the game is unsolvable; return `FALSE`.

    * If there are any cells with only *one* possible entry, fill them, and check the game.
      If invalid, the game is unsolvable; return `FALSE`.
      If still valid, resume the check for missing cells.
      If there are none, the game is solved, and return the solution.
    * If here are cells with multiple possible entries, choose one of them with the fewest possible. Loop over them, as follows.
      Fill the cell with the first possibility, and recursively call `findSolution` to see if there is a solution. If so return it.
      If not, proceed to the next possibility, and try again.
      If no possibility leads to a solution, there is none. Return `FALSE`.

– If a solution has been found by `findSolution` convert it back to a matrix and return it; if none has been found the game is unsolvable: return `NULL`.

The code. This speaks for itself.

```
solveGame <- function(game) {
  n <- as.integer(round(sqrt(nrow(game))));  stopifnot(n > 1 && n < 6)

  set <- if(n <= 3) as.character(1:n^2) else LETTERS[1:n^2] ## legals
  storage.mode(game) <- "character"
  is.na(game[!(game %in% set)]) <- TRUE
  nSet <- 1:n^2  ## work in integers

  toMatrix <- function(game) {  ## inverse of toArray
    game[] <- set[game]
    dim(game) <- c(n, n)^2
    game
  }

  toArray <- function(mat) {  ## inverse of toMatrix
    array(as.integer(match(mat, set)), dim = c(n,n,n,n))
  }

  conflict <- function(section)
      any(duplicated(na.omit(as.vector(section))))
```

```
invalid <- function(game) {
  for(i in 1:n) for(j in 1:n) {
    if(conflict(game[,i,,j]) ||  ## 'same block'
       conflict(game[i,j,,]) ||  ## 'same row'
       conflict(game[,,i,j]))    ## 'same column'
        return(TRUE)
  }
  FALSE
}
findSolution <- function(game) {
  if(invalid(game)) return(FALSE)  ## dead end.  Go back.
  while(anyNA(game)) {  ## anyNA() is only  in R 3.0.2 and later.
    holes <- which(is.na(game), arr.ind = TRUE)  ## a splendid trick!
    nr <- nrow(holes)
    fills <- vector("list", nr)
    lengths <- integer(nr)
    for(j in 1:nr) {
      i <- holes[j,]
      fills[[j]] <- setdiff(nSet, c(game[    ,i[2],    ,i[4]],
                                    game[i[1],i[2],    ,    ],
                                    game[    ,    ,i[3],i[4]]))
```

```
      lengths[j] <- length(fills[[j]])
      if(lengths[j] == 0) return(FALSE)
    }
    if(any(h <- which(lengths == 1))) {
      game[holes[h,,drop = FALSE]] <- unlist(fills[h])
      if(invalid(game)) return(FALSE)
    } else {  ## only holes with multiple alternatives
      m <- which.min(lengths)
      entries <- fills[[m]]
      pos <- holes[m,,drop = FALSE]
      for(e in entries) {
        game[pos] <- e
        h <- findSolution(game)   ## recursive call
        if(!isFALSE(h)) return(h) ## Bingo!
      }
      return(FALSE)  ## dud game, no solutions!
    }
  }
  game  ## should never reach this point of the code.
}

## the business starts here
```

```
  solution <- findSolution(toArray(game))
  if(isFALSE(solution)) NULL else
  structure(toMatrix(solution), game = game, class = "sudoku")
}
```

The short form, *solve(game)*, is made available by writing a method for the *S3* generic function, *solve*, in the *base* package. The method simply uses *solveGame*.

```
base::solve

function (a, b, ...)
UseMethod("solve")
<bytecode: 0x563ccfe7a858>
<environment: namespace:base>

sudokuAlt:::solve.sudoku

function (a, ...)
{
    solveGame(a)
}
<bytecode: 0x563cce49e848>
<environment: namespace:sudokuAlt>
```

Session information

Date: 2021-01-29

- R version 4.0.3 (2020-10-10), `x86_64-pc-linux-gnu`
- Locale: `LC_CTYPE=en_AU.UTF-8`, `LC_NUMERIC=C`, `LC_TIME=en_AU.UTF-8`, `LC_COLLATE=en_AU.UTF-8`, `LC_MONETARY=en_AU.UTF-8`, `LC_MESSAGES=en_AU.UTF-8`, `LC_PAPER=en_AU.UTF-8`, `LC_NAME=C`, `LC_ADDRESS=C`, `LC_TELEPHONE=C`, `LC_MEASUREMENT=en_AU.UTF-8`, `LC_IDENTIFICATION=C`
- 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, fractional 0.1.3, 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, sudokuAlt 0.2-1, 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, 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