Department of Statistics
STATS 782 Statistical Computing
Assignment 4(2022.1)

Total: 50 marks **Due: 23:59 NZT, Thursday 2 June 2022**

1. Please read these instructions carefully. Further instructions might be posted on the class webpage.
2. Upload your soft copy (assignment source) to Canvas: the file should end in `.Rmd`, or possibly `.R` or `.Rnw`. The marker may run or knit your R code, so include your name and ID in all files. The file names should contain your UPI. RMarkdown is strongly recommended.
3. Upload the source package tar ball (typically `pcp_1.0.tar.gz`) for the package you created for Question 2.
4. Also upload your `.pdf` to Canvas too. **Note the time difference between countries**.
5. Coversheet: please make sure you do **one** of the following else your assignment will not be marked:
   (a) Sign the Cover Sheet and combine with your assignment document (pdf or Word) into a single file before submission, OR
   (b) Type or write for the following at the beginning of your assignment: Your name (as it appears in Canvas), your UPI, and the following statement: "I have read the declaration on the cover sheet and confirm my agreement with it."
6. Include everything in your report: R code (tidied up), outputs (including error/warning messages), and your explanations (if any). Please comment on almost all of your output, especially parts that need human interpretation, else marks will be deducted. That is, you need to convince the marker that you understand what the data or solution is saying.
7. Print some intermediate results to show how your code works step by step, if not obvious. Comment your code if appropriate, e.g., for functions, blocks of code, and key variables.
8. Type `help.start()` when you open R. You need to use the online help to find details and functions that may not be covered directly in the coursebook. This requires maturity; we cannot cover everything in class or the coursebook.
9. Your mark for this assignment will depend on getting the right answer, the elegance/efficiency of your approach, and the tidiness and documentation of your code/report. Avoid copy/paste - use programmatic ways to repeat things if needed. **Marks (up to 7) will be deducted for messy code, etc**.
10. This PDF file may contain colour that is important to see.

1. **[31 marks]** This question uses R **S3** object system.

   Define a class `"pgon"` which represents a single polygon in a plane. It shall be defined by a set of points represented by coordinates `x` and `y` (numeric vectors of the same length) where the first elements of those vectors specify the coordinates of the first point of the polygon, the second elements the second point etc. The values in the vectors must be finite and may not contain missing values.

   (a) Define a constructor `pgon(x, y)` creating an object of the above class `"pgon"`. Make sure the returned object is valid and has at least three points. Recyling rule should be applied to the arguments `x` and `y` if necessary. [4 marks]

```
> pgon <- function(x, y) {
      if (!is.numeric(x) || !all(is.finite(x)) ||
          !is.numeric(y) || !all(is.finite(y)))
          stop("`x' and `y' must be finite, numeric vectors")
      n <- length(x)
      if (length(y) > n) {
          n <- length(y)
          x <- rep(x, length.out=n)
      } else if (length(y) < n)
          y <- rep(y, length.out=n)
      if (n < 3)
          stop("A polygon must have at least three points")

      structure(list(x=x, y=y), class="pgon")
  }
```

   Test with:

```
> p4 <- pgon(c(0,1,1), c(0,0,1,1))
> a  <- seq(0, 2*pi, length.out=9)[1:8]
> p8 <- pgon(sin(a), cos(a))
```

   (b) Define R S3 accessor function `pts` and the corresponding method for the class `pgon` to return the points as a matrix with columns `x` and `y`. [3 marks]

```
> pts <- function(x) UseMethod("pts")
> pts.pgon <- function(x) cbind(x=x$x, y=x$y)
```

```
> pts(p4)

     x y
[1,] 0 0
[2,] 1 0
[3,] 1 1
[4,] 0 1
```

   (c) Write a `length` method for the class `"pgon"` which returns the number of points in the polygon. [1 mark]

```
> length.pgon <- function(x) nrow(pts(x))
```

```
> length(p8)

[1] 8
```

(d) Write a `print` method to display the objects. If there are more than 4 points, show the first 4 points and `[...]` as the last line, e.g.: [4 marks]

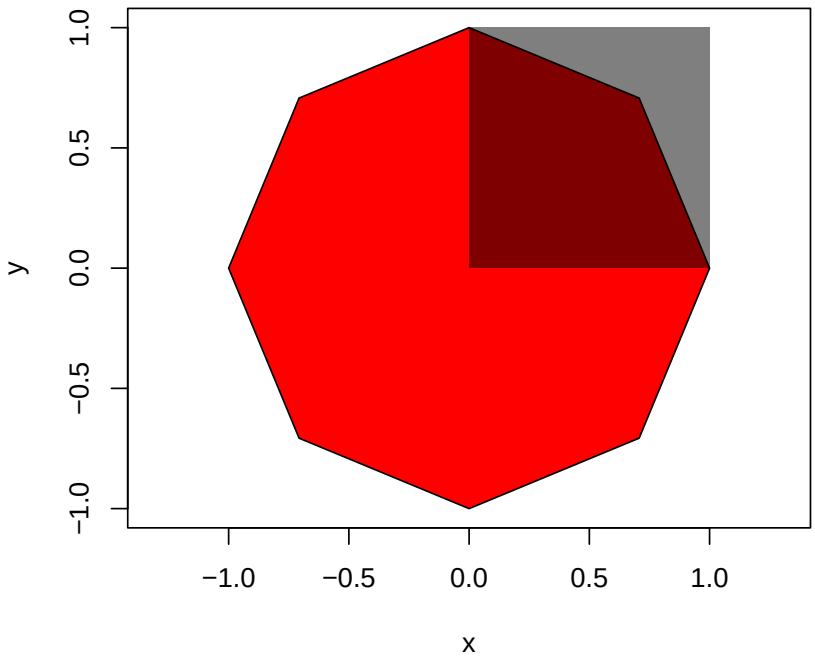
```
> .truncated.print <- function(x, ...)
    if (nrow(x) > 4) {
      print(x[1:4,], ...)
      cat("[...]\n")
    } else print(x, ...)
>
> print.pgon <- function(x, ...) {
    cat("Polygon of ", length(x), " points.\n", sep='')
    .truncated.print(pts(x), ...)
    invisible(x)
  }
```

```
> p8

Polygon of 8 points.
             x             y
[1,] 0.0000000  1.000000e+00
[2,] 0.7071068  7.071068e-01
[3,] 1.0000000  6.123234e-17
[4,] 0.7071068 -7.071068e-01
[...]
```

(e) Implement the `plot` method for the `"pgon"` class. You should allow for a logical argument `add` which determines whether a new plot will be created (`add=FALSE`, default) or the polygon added to an existing plot (`add=TRUE`). If a new plot is created, the parameter `asp` should be honoured for setting up the new plot and should default to 1. Other additional graphical parameters should be passed to the function drawing the polygon. Use the following code to reproduce the figure below: [4 marks]

```
> ## define the method, it must start with x, y (even
> ## if y is not used), because the generic is defined
> ## as plot(x, y, ...).
> ## asp= must be explicit since ... is passed to
> ## polygon() and not plot()
> plot.pgon <- function(x, add=FALSE, asp=1, ...) {
    if (!add)
      plot(range(x$x), range(x$y), asp=asp, type='n',
         xlab="x", ylab="y")
    polygon(x$x, x$y, ...)
  }
>
> ## Note: as.data.frame(pts(x)) can be used instead of (x$x, x$y)
```

```
> par(mar=c(4, 4, 0.1, 0.1))
> plot(p8, col="red")
> plot(p4, col="#00000080", border=NA, add=TRUE)
```



(f) Implement addition and subtraction on the `pgon` class objects as follows:

- The only objects that can be added or subtracted to/from the `pgon` objects are numeric vectors of length two where the first element is interpreted as `x` and the second element as `y`.
- Addition causes the the polygon to be translated (moved) by the values of the vector: given polygon points (x_i, y_i) and the vector (x, y) the result is a polygon with points $(x_i + x, y_i + y)$
- Unary minus reflects the polygon in the origin: $(x_i, y_i) \to (-x_i, -y_i)$.
- Binary minus follows the rule $a - b = a + (-b)$.

```
>  is.pgon <- function(x) inherits(x, "pgon")
>  is.pvec <- function(x) is.numeric(x) && length(x) == 2 && all(is.finite(x))
>
>  `+.pgon` <- function(e1, e2) {
     ## unary is identity
     if (missing(e2)) e1
     else if (is.pgon(e1) && is.pvec(e2))
           pgon(e1$x + e2[1], e1$y + e2[2])
     else if (is.pvec(e1) && is.pgon(e2))
           e2 + e1
     else
       stop("+ is only defined for pgons and numeric vectors of length two")
   }
>
>  `-.pgon` <- function(e1, e2) {
     if (missing(e2)) { ## unary minus is reflection
       e1$x <- -e1$x
       e1$y <- -e1$y
       e1
     } else ## binary
       e1 + (-e2)
   }
>
>  ## NOTE: you cannot use Ops, because there is no commonality
>  ## in implemntation (recall Lab 9) and it must fail for
```

```
>  ## unsupported operators.
```

Any other type combinations are not allowed. You can test your solution against these examples: [5 marks]

```
> -p4

Polygon of 4 points.
      x  y
[1,]  0  0
[2,] -1  0
[3,] -1 -1
[4,]  0 -1

> p4 + c(1,2)

Polygon of 4 points.
     x y
[1,] 1 2
[2,] 2 2
[3,] 2 3
[4,] 1 3
```

```
> p4 - pts(p8)[1,]

Polygon of 4 points.
     x  y
[1,] 0 -1
[2,] 1 -1
[3,] 1  0
[4,] 0  0

> c(1, 1) - p4

Polygon of 4 points.
     x y
[1,] 1 1
[2,] 0 1
[3,] 0 0
[4,] 1 0
```

(g) Define a subclass `"colygon"` of `"pgon"` which adds another component `"col"` (string) with the default `"grey"` and the corresponding constructor `colygon(x, y, col)`. It should also be possible to create a 'colygon' object from an existing 'pgon' object, i.e., `colygon(p8, col="red")` should work with `p8` being of class `"pgon"`. Then define a `print` method for `"colygon"` which also displays the colour. [4 marks]

```
> colygon <- function(x, y, col="grey") {
    if (!is.character(col) || length(col) != 1)
      stop("`col' must be a string")
    ## if x is not already a pgon call superclass' contructor
    o <- if (is.pgon(x)) x else pgon(x, y)
    ## add colour
    o$col <- col
    ## and our class as subclass
    class(o) <- c("colygon", class(o))
    o
  }
> print.colygon <- function(x, ...) {
    cat("Coloured polygon of ", length(x), " points and colour ", x$col, ".\n", sep='')
    .truncated.print(pts(x), ...)
    invisible(x)
  }
```

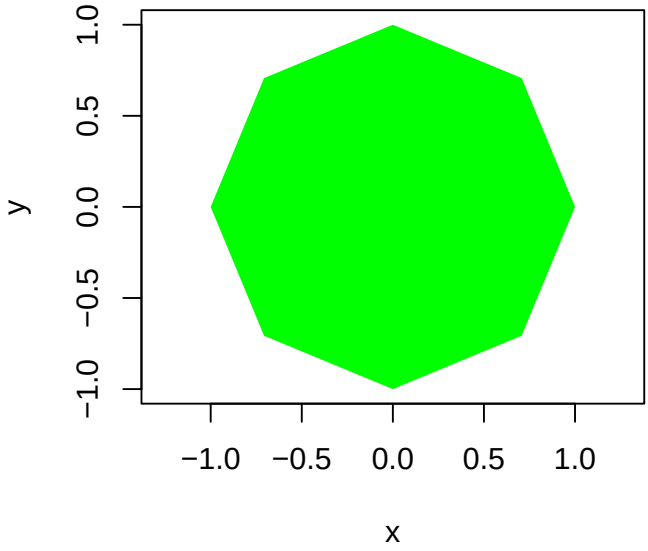
Test with:

```
> (l3 <- colygon(c(-1, 0, 1), c(-1, 1), col="yellow"))

Coloured polygon of 3 points and colour yellow.
      x  y
[1,] -1 -1
[2,]  0  1
[3,]  1 -1

> (l8 <- colygon(p8, col="green"))
```

```
Coloured polygon of 8 points and colour green.
             x             y
[1,] 0.0000000  1.000000e+00
[2,] 0.7071068  7.071068e-01
[3,] 1.0000000  6.123234e-17
[4,] 0.7071068 -7.071068e-01
[...]
```

(h) Do whatever is necessary such that the `plot` function will draw an object of the class `"colygon"` by filling the polygon with the contained colour and no borders. Test it with `plot(l8)`. [3 marks]

```
> ## we simply call the superclass' method with extra col and border
> plot.colygon <- function(x, ...)
    NextMethod(,, col=x$col, border=NA, ...)
>
>   ## or plot.pgon(x, col=x$col, border=NA, ...)
>
> plot(l8)
```



(i) Run the following code:

```
> l8 + c(0, 1)

Polygon of 8 points.
             x         y
[1,] 0.0000000 2.0000000
[2,] 0.7071068 1.7071068
[3,] 1.0000000 1.0000000
[4,] 0.7071068 0.2928932
[...]

> -l8

Coloured polygon of 8 points and colour green.
             x             y
[1,]  0.0000000 -1.000000e+00
```

```
[2,] -0.7071068 -7.071068e-01
[3,] -1.0000000 -6.123234e-17
[4,] -0.7071068  7.071068e-01
[...]
```

Depending on your implementation of the `+` and `-` operators you should see results that are either of class `"pgon"` or `"colygon"`. *Your output may be different from the above.* Explain the class of each of the operator results for *your* code. In case they are not both `"colygon"` write R code which makes sure that the results of both will be of class `"colygon"`. Try to define as few methods as possible and do not modify previous methods for `"pgon"`. [3 marks]

Answer

We have not implemented operator methods for `"colygon"` so the superclass' methods will be used. The results will be of class `"pgon"` if you used constructors in the operator implementations (here we used it for binary `+`), but they will be of class `"colygon"` if you returned a modified `e1` or `e2` object instead (as we did for the reflection). Both are valid soultions. To make sure that all operators are correct, we can let the superclass do the work and only re-attach the `col` component from the `"colygon"` object by defining the `Ops` group generic.

```
> `Ops.colygon` <- function(e1, e2)
    colygon(NextMethod(), col=if (inherits(e1, "colygon")) e1$col else e2$col)
>
> l8 + c(0, 1)

Coloured polygon of 8 points and colour green.
             x         y
[1,] 0.0000000 2.0000000
[2,] 0.7071068 1.7071068
[3,] 1.0000000 1.0000000
[4,] 0.7071068 0.2928932
[...]

> -l8

Coloured polygon of 8 points and colour green.
              x             y
[1,]  0.0000000 -1.000000e+00
[2,] -0.7071068 -7.071068e-01
[3,] -1.0000000 -6.123234e-17
[4,] -0.7071068  7.071068e-01
[...]
```

2. **[19 marks]** The file `source.R` contains R code which plots a simple Parallel Coordinates Plot. It consists of private functions `norm1`, `normN` and a public function `pcp` which draws the plot from a supplied list of variables.

   The purpose of this question is to create an R package. The list of tasks below defines the marks for each subtask and the recommended order, but at the end of it you are expected to upload the full package `pcp_1.0.tar.gz` to Canvas which will be your submission for this question. You are submitting your final work not individual steps.

   (a) Edit `source.R` and add comments before every complete code line (multi-line function calls count as one line), explaining what each line does. Your task is to explain the purpose, not to translate the R code to English. [3 marks]

   (b) Create a valid R package named `pcp` from the source file. Make sure all your comments from the previous question are included! Update `DESCRIPTION` to add all important information, you can use `GPL-2` as the license. [2 marks]

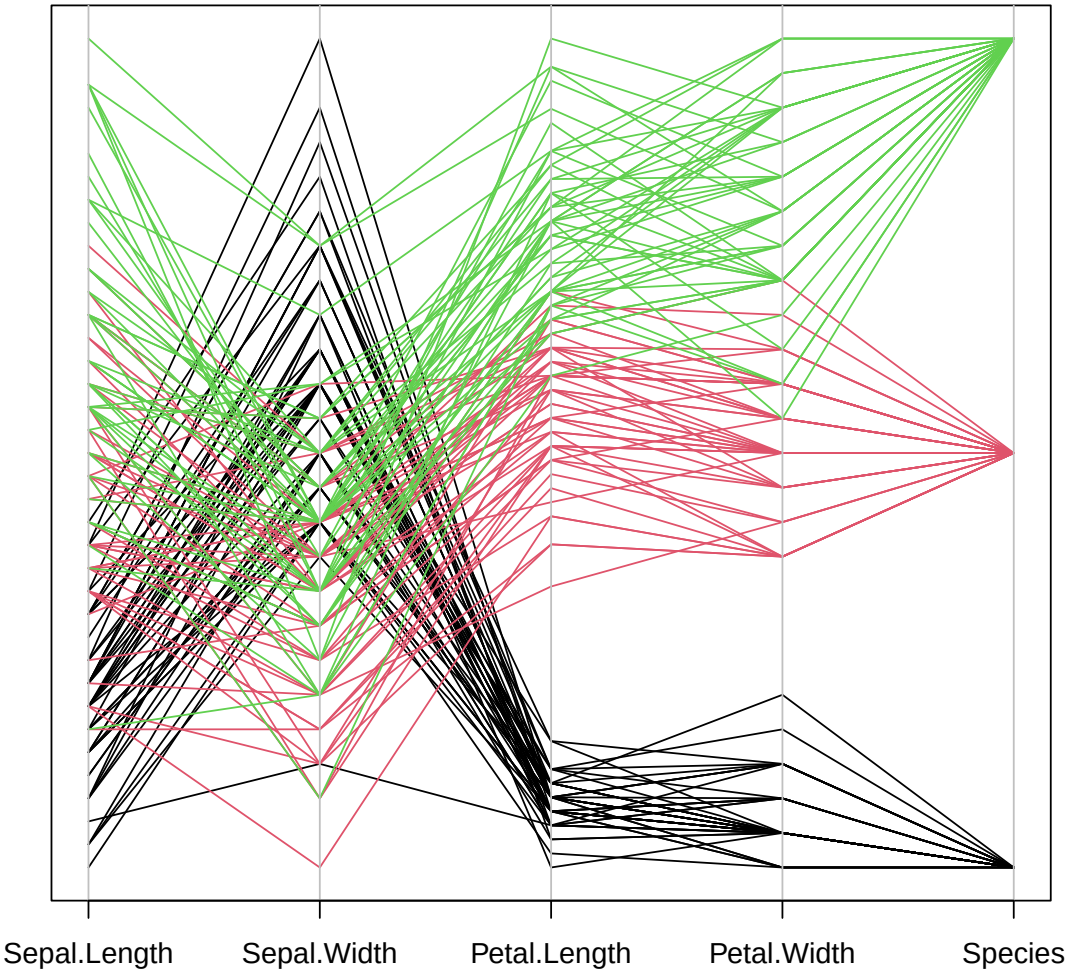
   (c) Edit `NAMESPACE` to make sure only the public function is exported. Make sure functions from other packages you use are imported and adjust `DESCRIPTION` accordingly if necessary. [2 marks]

```
== DESCRIPTION ==
Package: pcp
Type: Package
Title: Parallel Coordinates Plot
Version: 1.0
Author: Simon Urbanek
Maintainer: Simon Urbanek <simon.urbanek@R-project.org>
Description: Draws a simple parallel coordinates plot.
Imports: graphics
License: GPL-2


== NAMESPACE ==
export("pcp")
importFrom("graphics", "abline", "axis", "matplot")
```

   (d) Create complete documentation for the public function `pcp`. Describe every argument as well as the return value. (You can remove `pcp-package.Rd` file if you created the package using a skeleton, because your function has the same name as the package so we don't need two documentation pages.) [3 marks]

   (e) Add a working example to the `pcp` documentation which uses the function based on the `iris` dataset in R. Colour the groups by the `Species` variable. [3 marks]

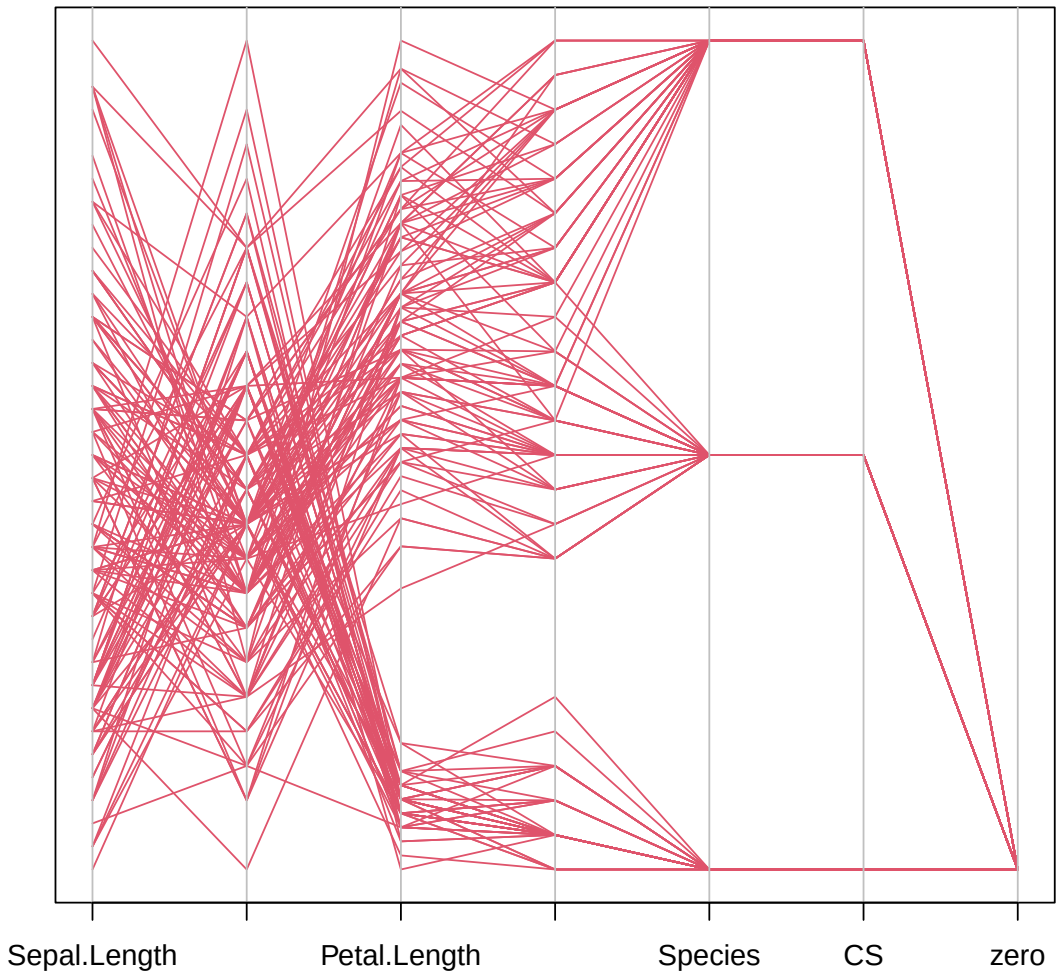
```
> library(pcp)
> example(pcp)


pcp> pcp(iris, col=iris$Species)
```



(f) Run `R CMD check` on your package and resolve any `ERROR`s, `WARNING`s or `NOTE`s by addressing any issues the check has found (with the exception of "PDF version of manual" - if you don't have TeX available you can ignore that error). [2 marks]

(g) Create a subdirectory `tests` in the package and a file `test.R` therein with code that tests the `pcp` function. Make sure your test covers 100% of the package, i.e. after you run the code in `test.R` every line of the functions defined in the package has been executed at least once. [4 marks]
Hint: to exercise code which generates errors, use `try(pcp(...), silent=TRUE)` to catch the error.

```
> library(pcp)
>
> ## factor, numeric, character and constant
> pcp(c(iris, list(CS=as.character(iris$Species), zero=0)))
>
> try(pcp("foo"), silent=TRUE)  ## non-list
> try(pcp(list()), silent=TRUE) ## too few variables
> try(pcp(c(iris, NA)), silent=TRUE) ## non-finite
```



Assuming your package source is in the `pcp` directory you can check the coverage by installing the `"covr"` package from CRAN and calling:

```
> library(covr)
> package_coverage("pcp", "all")

pcp Coverage:  100.00%
R/source.R: 100.00%
```

You can also see detailed report on which lines are covered and which are not by running

```
> report(package_coverage("pcp", "all"))
```

```
== pcp/tests/test.R ==
library(pcp)

## factor, numeric, character and constant
pcp(c(iris, as.character(iris$Species), 0))
```

```
try(pcp("foo"), silent=TRUE)  ## non-list
try(pcp(list()), silent=TRUE) ## too few variables
try(pcp(c(iris, NA)), silent=TRUE) ## non-finite

== pcp/R/source.R ==
## normalise one vector
norm1 <- function(x) {
    ## convert character vectors to factors
    if (is.character(x)) x <- factor(x)
    ## convert factors to their integer indices
    if (is.factor(x)) x <- as.integer(x)
    ## check if the converted x is non-numeric or
    ## has any non-finite elements (including NAs)
    if (!is.numeric(x) || !all(is.finite(x)))
        ## raise an error if it does
        stop("All variables must finite numerics, factors or strings")
    ## make sure the lowest value is 0 (works, becasue x is finite)
    x <- x - min(x)
    ## if the vector is not constant 0 make sure the highest value is 1,
    ## otherwise just return 0s (to avoid division by 0)
    if (any(x > 0)) x / diff(range(x)) else x
}

## normalize a list of variables
normN <- function(data)
    ## normalise each variable and return the result
    lapply(data, norm1)

## draws a parallel coordinates plot
pcp <- function(data, col=2, lwd=1, lty=1, ...) {
    ## data must be a list, raise an error orterwise
    if (!is.list(data))
        stop("`list' must be a list of variables!")
    ## data must contain at least two variables
    if (length(data) < 2)
        stop("Need at least two variables")
    ## normalise all variables
    data <- normN(data)
    ## compute a vector containing the lengths of the variables
    n <- lengths(data)
    ## find the size N of the longest variable
    N <- max(n)
    ## if the variables don't have the same length, apply
    ## recyling rule by extending them to the length N
    if (!all(n == N))
        data <- lapply(data,
                       function(x) rep(x, length.out=N))
    ## convert all variables into a matrix with variables in columns
    ## (possible since they now have the same length and are all numeric)
    m <- matrix(unlist(data), N)
    ## set column names from the data variable names
    colnames(m) <- names(data)
    ## draw a line-plot of all the rows as lines and variables along the x axis
    ## pass-though ... from the function. Do not draw either axis label
    matplot(t(m), type='l', col=col, lwd=lwd, lty=lty,
            xlab='', ylab='', xaxt='n', yaxt='n', ...)
    ## add vertical lines at the variables
    abline(v = seq_along(data), col="#c0c0c0")
    ## add bottom axis label with variable names
```

```
    axis(1, seq_along(data), names(data))
    ## return the normalised and recycled variables
    invisible(data)
}

R CMD build pcp
* checking for file ‘pcp/DESCRIPTION’ ... OK
* preparing ‘pcp’:
* checking DESCRIPTION meta-information ... OK
* checking for LF line-endings in source and make files and shell scripts
* checking for empty or unneeded directories
* building ‘pcp_1.0.tar.gz’

R CMD INSTALL pcp_1.0.tar.gz

R CMD check pcp_1.0.tar.gz
* using log directory ‘pcp.Rcheck’
* using R version 4.2.0 (2022-04-22)
* using platform: x86_64-apple-darwin17.0 (64-bit)
* using session charset: UTF-8
* checking for file ‘pcp/DESCRIPTION’ ... OK
* checking extension type ... Package
* this is package ‘pcp’ version ‘1.0’
* checking package namespace information ... OK
* checking package dependencies ... OK
* checking if this is a source package ... OK
* checking if there is a namespace ... OK
* checking for executable files ... OK
* checking for hidden files and directories ... OK
* checking for portable file names ... OK
* checking for sufficient/correct file permissions ... OK
* checking whether package ‘pcp’ can be installed ... OK
* checking installed package size ... OK
* checking package directory ... OK
* checking DESCRIPTION meta-information ... OK
* checking top-level files ... OK
* checking for left-over files ... OK
* checking index information ... OK
* checking package subdirectories ... OK
* checking R files for non-ASCII characters ... OK
* checking R files for syntax errors ... OK
* checking whether the package can be loaded ... OK
* checking whether the package can be loaded with stated dependencies ... OK
* checking whether the package can be unloaded cleanly ... OK
* checking whether the namespace can be loaded with stated dependencies ... OK
* checking whether the namespace can be unloaded cleanly ... OK
* checking dependencies in R code ... OK
* checking S3 generic/method consistency ... OK
* checking replacement functions ... OK
* checking foreign function calls ... OK
* checking R code for possible problems ... OK
* checking Rd files ... OK
* checking Rd metadata ... OK
* checking Rd cross-references ... OK
* checking for missing documentation entries ... OK
* checking for code/documentation mismatches ... OK
* checking Rd \usage sections ... OK
* checking Rd contents ... OK
* checking for unstated dependencies in examples ... OK
```

```
* checking examples ... OK
* checking for unstated dependencies in ‘tests’ ... OK
* checking tests ...
 OK
* checking PDF version of manual ... OK
* DONE
Status: OK
```