R essentials for model calibration

Willem Vervoort

10 July 2017

Contents

THE UNIVERSITY OF SYDNEY

International Research Institute for Climate and Society
Earth Institute | Columbia University

Instituto Nacional de Investigación Agropecuaria
URUGUAY

Introduction

This is an introduction to R written for the "How do I use satellite and global reanalysis data for hydrological simulations in SWAT?" workshop in Montevideo between 7 - 11 August 2017, jointly organised by the University of Sydney, IRI and INIA.
This work is based on earlier "introduction to R" practicals in units of study at the University of Sydney and an introduction to R course taught in Guernavaca, Mexico in 2014. It also builds on many of the introduction to R literature on the internet.
This course is not a complete introduction, and more in depth knowledge on R and the use of R can be gained from many courses on-line and by basic practice.
I hope that this course offers sufficient depth to help you engage in the rest of the course, which uses R for analysis of the output.

This course covers simple R, basic statistics, data frame operations, reading in files and a plotting. The final part covers calibration of a simple hydrological model using the package hydromad, which is based on the unit LWSC3007 at the University of Sydney.

R as a modelling environment

The origins of R are in statistics, so this is hat R does best. However, over time, it has proven to be a flexible language that can also be used quite effectively for programming and data science.

some intro about R

Spanish documentation on R is extensively available, for example: Introduction to R and via datacamp.

In this introduction I also have also relied on R for Data Science, in particular chapter 4.

Basic R

In its most basic form, R is a calculator

```
3*5
```

```
## [1] 15
```

```
50/100 + 0.1
```

```
## [1] 0.6
```

```
10 - 20
```

```
## [1] -10
```

The basic structure of R is based on objects, which are named. R is case sensitive, so keep this in mind. The main object we will use here is a *dataframe* or its modern variant the *tibble*.

R uses "<-" to assign a value (or another object) to an object

```
# assign
x <- 5
y <- 2
```

You can call up what is stored in the object (inspect) again by just typing its name:

```
x
```

```
## [1] 5
```

These objects will show up in the "Environment" window in Rstudio, or you can use `ls()` in the console to list the objects. The function `c()` can be used to stick things together into a vector. Redo the below commands in your own script.

```
# a vector
x = c(1,2,5,7,8,15,3,12,11,19)
# another vector
y = 1:10
# you have now two objects
ls()
```

```
## [1] "root" "x"    "y"
```

```
# you can add, multiply or subtract
z = x + y
z
```

```
##  [1]  2  4  8 11 13 21 10 20 20 29
```

```
zz = x * y
zz
```

```
##  [1]   1   4  15  28  40  90  21  96  99 190
```

```
zzz = x - y
zzz
```

```
##  [1]  0  0  2  3  3  9 -4  4  2  9
```

```
foo = 0.5*x^2 - 3*x + 2
foo
```

```
##  [1]  -0.5  -2.0  -0.5   5.5  10.0  69.5  -2.5  38.0  29.5 125.5
```

A dataframe

A dataframe is a bit more complex, and here is a simple demonstration of its power.

```
Rainfall <- data.frame(City = c("Montevideo","New York", "Amsterdam","Sydney", "Moscow", "Hong Kong"),
                       Rain_mm = c(950, 1174, 838, 1215, 707, 2400))
Rainfall
```

```
##         City Rain_mm
## 1 Montevideo     950
## 2   New York    1174
## 3  Amsterdam     838
## 4     Sydney    1215
## 5     Moscow     707
## 6  Hong Kong    2400
```

As you can see a data.frame can mix character columns (City) and numeric columns (Rain_mm). Here I used `c()` to generate vectors which I put in the columns. In addition, the columnns have names, which you can access using `colnames()`: City, Rain_mm

Once you have a dataframe, you can access parts of the dataframe or manipulate the dataframe.

```
# call a column
Rainfall$City
```

```
## [1] Montevideo New York   Amsterdam  Sydney     Moscow     Hong Kong
## Levels: Amsterdam Hong Kong Montevideo Moscow New York Sydney
```

```
# or
Rainfall["City"]
```

```
##         City
## 1 Montevideo
## 2   New York
## 3  Amsterdam
## 4     Sydney
## 5     Moscow
## 6  Hong Kong
```

```
# or
Rainfall[,1]
```

```
## [1] Montevideo New York   Amsterdam  Sydney     Moscow     Hong Kong
## Levels: Amsterdam Hong Kong Montevideo Moscow New York Sydney
```

```
# find a row
Rainfall[Rainfall["City"]=="Montevideo"]
```

```
## [1] "Montevideo" " 950"
```

```
# see the first two rows
Rainfall[1:2,]
```

```
##         City Rain_mm
## 1 Montevideo     950
## 2   New York    1174
```

```
# find a subset
lots <- Rainfall[Rainfall["Rain_mm"] > 1000,]
lots
```

```
##        City Rain_mm
## 2  New York    1174
## 4    Sydney    1215
## 6 Hong Kong    2400
```

Exercise

Using the above examples, can you do the following?

- Extract the column with the rainfall values?
- Extract the row with the annual rainfall at Amsterdam?
- Which cities have rainfall below 1500 mm?

The working directory

Generally R works from a "working directory". This is the directory on disk where it expects to find files or write files to. You can set this in Rstudio via the menu item "Session" –> "Set working directory", but you can also set this in code. Setting the working directory is useful when you want to access data in files on your computer or the network.

The basic function to use is `setwd("path/to/file")`. The thing to note is that in the path description you have to use "forward /" rather than the standard windows "backward".

```
# set the working directory
setwd("C:/users/rver4657/Documents")

# see some of the files
dir()[1:10]
```

```
##  [1] "~$REGMIP.docx"
##  [2] "~$tes EG1.docx"
##  [3] "~$tes EGU.docx"
##  [4] "20110309_scan2.jpg"
##  [5] "20140702_LetterSebInternship.docx"
##  [6] "20160316_Abstract.docx"
##  [7] "20161203_SintgedichtDelft.txt"
##  [8] "aansprakelijkheidsverzekeringen.txt"
##  [9] "Custom Office Templates"
```

```
## [10] "Default.rdp"
```

Reading data from different sources

There are a multitude of functions to read data from the disk into the R memory, I will demonstrate only a few here.

Because a lot of data is stored in comma delimited txt files (such as Excel exports), using `read.csv()` is a good standard option.

Here I am reading in some monthly data from the Concordia station in the Uruguay river in Argentina. This data was originally downloaded from the Global River Discharge Database

```
UR_flow <- read.csv("data/UruguayRiver_ConcordiaSt.csv")
# check the first few lines (6 by default)
head(UR_flow)
```

```
##   Year Month Flow
## 1 1969     1 7888
## 2 1969     2 5951
## 3 1969     3 4296
## 4 1969     4 4173
## 5 1969     5 4539
## 6 1969     6 4857
```

Exercise

- Can you read in the file: "Parana_CorrientesSt.csv"?

Packages and library

Much of the power in R comes from the fact that it is open source and this means many people write new code and share this code. The formal way to do this is via "packages", which, once checked and endorsed by the R community, appear in the CRAN repository as a **package**.

Here we might want to use some of the features in the package tidyverse. The other package we will use later is the package zoo

There are two components to using packages. The first is to make sure that the package is installed, for which we can use the function `install.packages()`. Note that the name of the package is a *string* so needs to between quotes `""`.

```
install.packages("tidyverse")
```

If the package is installed in your personal library, you will need to load the package in R using `require()` or `library()`. There are subtle differences between these two functions, but they are currently not that important. Check the help files.

```
require(tidyverse)
```

```
## Loading required package: tidyverse
```

```
## Loading tidyverse: tibble
## Loading tidyverse: tidyr
## Loading tidyverse: readr
## Loading tidyverse: purrr
```

```
## Loading tidyverse: dplyr
## Conflicts with tidy packages ----------------------------------------------
## filter(): dplyr, stats
## lag():    dplyr, stats
```

Exercise

- Can you load the package zoo?

summarising data

It is often important to summarise data, for example we might want to know the average monthly flow or the standard deviation of flow. R of course have several functions to deal with this.

standard statistical functions

Here are some simple examples of standard statistical functions `mean`, `sd` and `cor` (and of course there are many more).

```
# average monthly flow
mean(UR_flow$Flow)
```

```
## [1] 5456.553
```

```
# st dev average flow
sd(UR_flow$Flow)
```

```
## [1] 3491.968
```

```
# subset two years and correlate
flow1969 <- UR_flow[UR_flow$Year==1969,]
flow1970 <- UR_flow[UR_flow$Year==1970,]

cor(flow1969$Flow,flow1970$Flow)
```

```
## [1] -0.3164468
```

using aggregate()

Another useful function is `aggregate()`, which allows you to apply a function over data frame and particular across different factors. Here is an example of summing the Uruguay river flow by year.

```
# aggregate to annual flow
(annual_flow <- aggregate(UR_flow,list(Year=UR_flow$Year),sum))
```

```
##    Year  Year Month   Flow
## 1  1969 23628    78  53753
## 2  1970 23640    78  52130
## 3  1971 23652    78  66648
## 4  1972 23664    78  99562
## 5  1973 23676    78 103070
## 6  1974 23688    78  47130
## 7  1975 23700    78  68075
```

```
## 8  1976 23712    78  53500
## 9  1977 23724    78  73650
## 10 1978 23736    78  41700
## 11 1979 23748    78  61047
```

Note that the parentheses around the statement means that the result of the statement is printed.

Exercise

- Can you calculate the standard deviation of the monthly flow by year?

if else and for loops

Get this from Tom's older stuff Loops are essential in programming. There are a range of different types of loops, but here I will only demonstrate "if else" and "for". I have always been told that all other loops are just derivatives or short cuts.
An "if else" loop allows you to program a switch in the code.
Basically it is used to evaluate an expression if the statement is TRUE and to evaluate another expression if the statement is FALSE:
if (comparison) do this, else do that
You might call the above line "pseudo code". It is sometimes handy to first write something in pseudo code, basically you want to write in broad language what you want to happen.
You could also use only the "if" part, which means nothing happens if the statement is FALSE.

As an example I want to do: if (a data frame has more than 10 rows) write data frame is LONG, else write data frame is SHORT

```
#We will first generate the data frame:
x <- UR_flow
# now wrote loop
if (nrow(x) > 10) {
    print("the dataframe is LONG")
} else {
    print("the dataframe is SHORT")
}
```

```
## [1] "the dataframe is LONG"
```

There are a few things to note here. I use the statement `nrow()` to check how many rows the data frame has. I use the statement `print()` to write something to the screen.

Exercise

- Try generating another data frame x and rerun the program, or change the program to get it to say "the dataframe is short".

R also has an `ifelse()` command. This is a vectorized version of the if command - which means that it can be used on vectors of data - the command is applied to each element (or value) of the vector in turn. The if command only evaluates single values.
Using the ifelse command will return a vector of values, the same length as the longest argument in the expression.
Wherever possible, it is preferable to use the ifelse command rather than using the if command in combination with a loop - writing the program is more efficient and R evaluates vectorised functions more efficiently than it does loops. Here is an example which changes the program above

```
x <- UR_flow
# add a column which identifies whether the flow < 5000
x[,4] <- ifelse(x[,3] > 5000, "large", "small")
# this creates a third column
tail(x,10)
```

```
##     Year Month  Flow    V4
## 123 1979     3  1699 small
## 124 1979     4  1650 small
## 125 1979     5  5052 large
## 126 1979     6  2619 small
## 127 1979     7  3776 small
## 128 1979     8  6134 large
## 129 1979     9  3275 small
## 130 1979    10 15692 large
## 131 1979    11 12244 large
## 132 1979    12  7380 large
```

I check in the second column of the data.frame whether the flows are greater than 5000 or not. I then write in the third column whether they are large or small numbers. A more complex (nested) ifelse version would be:

```
x[,5] <- ifelse(x[,3] > 2500,ifelse(x[,3] > 10000, "large", "intermediate"), "small")
tail(x,10)
```

```
##     Year Month  Flow    V4           V5
## 123 1979     3  1699 small        small
## 124 1979     4  1650 small        small
## 125 1979     5  5052 large intermediate
## 126 1979     6  2619 small intermediate
## 127 1979     7  3776 small intermediate
## 128 1979     8  6134 large intermediate
## 129 1979     9  3275 small intermediate
## 130 1979    10 15692 large        large
## 131 1979    11 12244 large        large
## 132 1979    12  7380 large intermediate
```

You can try out some of your own versions of this

The "for" loop, getting the program to do something repeatedly

Loops are used to repeat a set of commands. Normally, there will be a variable which changes value in each successive loop through the commands. Reference to this changing value results in differences in output from successive iterations.

The for loop is used when the number of required iterations is known before the loop begins. It is used in the following way: for (name in expression1) {expression2}

- name is the name of the loop variable. Its value changes during each iteration, starting with the first value and ending with the last value in expression1.
- expression1 is a vector expression (often a sequence, such as 1:10).
- expression2 is a command or group of commands that are repeatedly evaluated. It usually contains references to name, which result in changes to the value of the expression as the value of name changes.

Here is the classic example of a loop

```
# Hello world
for (i in 1:5) {
```

```
    print(paste(i, "hello world"))
}
```

```
## [1] "1 hello world"
## [1] "2 hello world"
## [1] "3 hello world"
## [1] "4 hello world"
## [1] "5 hello world"
```

Note the use of `paste()` to combine character vectors.
Here is another simple loop that tells you the first 5 values of the flow data.

```
for (i in 1:5) {
  print(paste(UR_flow$Flow[i], "is the flow (ML/day)"))
}
```

```
## [1] "7888 is the flow (ML/day)"
## [1] "5951 is the flow (ML/day)"
## [1] "4296 is the flow (ML/day)"
## [1] "4173 is the flow (ML/day)"
## [1] "4539 is the flow (ML/day)"
```

```
# or more complex:
for (i in 1:5) {
  print(paste("in Year", UR_flow$Year[i],"and month",
              UR_flow$Month[i],
              "the flow is", UR_flow$Flow[i], "(ML/day)"))
}
```

```
## [1] "in Year 1969 and month 1 the flow is 7888 (ML/day)"
## [1] "in Year 1969 and month 2 the flow is 5951 (ML/day)"
## [1] "in Year 1969 and month 3 the flow is 4296 (ML/day)"
## [1] "in Year 1969 and month 4 the flow is 4173 (ML/day)"
## [1] "in Year 1969 and month 5 the flow is 4539 (ML/day)"
```

You can also nest loops, that is, embed one loop into another. Here is an example that prints both the year and the flow using the column names in the dataframe.

```
for (i in 1:5) {
  for (j in c(1,3)) {
  print(paste(UR_flow[i,j], colnames(UR_flow)[j]))
  }
}
```

```
## [1] "1969 Year"
## [1] "7888 Flow"
## [1] "1969 Year"
## [1] "5951 Flow"
## [1] "1969 Year"
## [1] "4296 Flow"
## [1] "1969 Year"
## [1] "4173 Flow"
## [1] "1969 Year"
## [1] "4539 Flow"
```

Exercise

- Write another program that includes a loop and a logical test

Comparison and Logical Operators

Comparison operators return a true or false value:

- == Equal to
- Greater than
- = Greater than or equal to
- < Less than
- <= Less than or equal to

Comparison operators can be combined with logical operators to describe more complex conditions.

Logical operators:

- ! Not
- | or (used for vectors, with the ifelse command)
- || or (used for single values)
- & and (used for vectors, with the ifelse command)
- && and (used for single values)

Exercise

Write a small program that uses comparison operators and a logical operator. # Plotting using ggplot and using zoo

R has many different plotting options, but recently ggplot2 appears to be the preferred option. One of the reasons for the support for ggplot2 is because it can also be used in Python.

As a start I will first modify the UR_flow data to make this into a "zoo" data frame, which has a timeseries based index and helps with plotting.

The function basically takes the data and links it to an index (such as the date). I first construct a vector of dates using the month and year from `UR_flow`.

```
require(zoo)
```

```
## Loading required package: zoo
```

```
##
## Attaching package: 'zoo'
```

```
## The following objects are masked from 'package:base':
##
##     as.Date, as.Date.numeric
```

```
UR_flow$Dates <- as.Date(paste(UR_flow$Year,UR_flow$Month,"01",sep="-"))
UR_flow_z <- zoo(UR_flow$Flow, order.by = UR_flow$Dates)
```

Once we have this zoo data frame, it can be plotted quite easily with the basic plotting package.

```
plot(UR_flow_z)
```

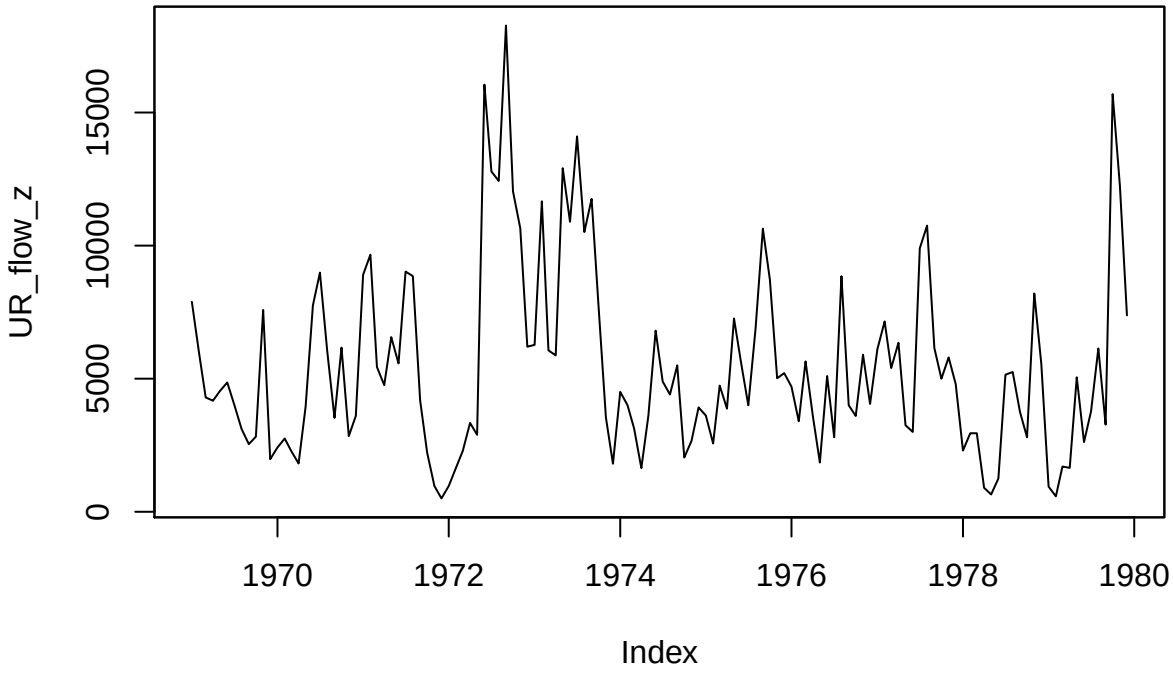


Figure 1: Demonstration of the simple plotting package in R

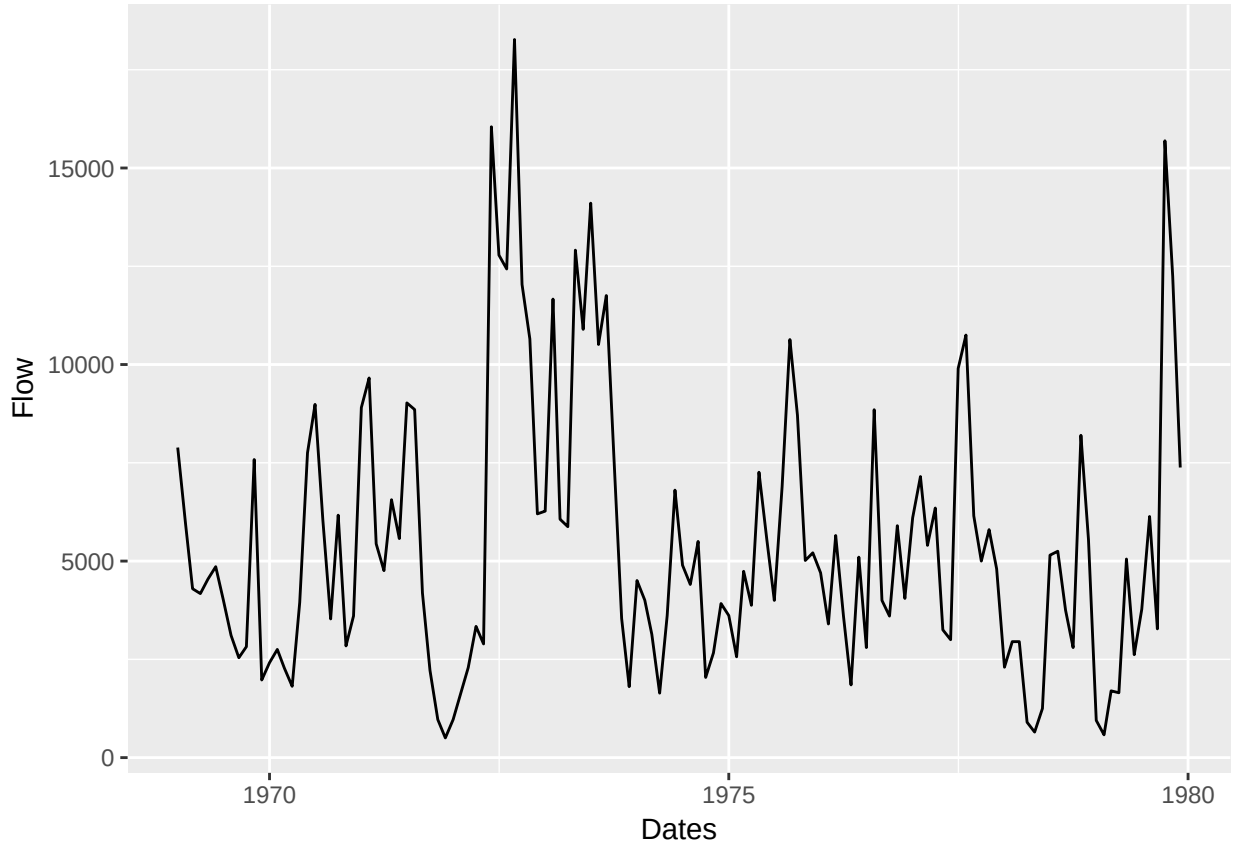


Figure 2: Demonstration of the ggplot2 package in R

Using ggplot2 is a bit more involved, and does not work well directly with zoo. So we need to go back to UR_flow.

```
p <- ggplot(UR_flow, aes(x=Dates, y = Flow)) + geom_line()
print(p)
```

Writing functions in R

Until now you have used several functions in R that are part of packages or part of "core" R. However, another powerful element in R is the ability to write your own functions. There are two major advantages with writing functions:

1. They are easy to test, as they are contained. This is especially true if keep functions short.
2. They are short cuts and repeatable and therefore limit the possibility of typos.

Let's go back to the "hello world" example that we used in a loop earlier. We can write the same example in a function.
The first thing to do is to decide which inputs we want the function to use to create the output. In this case I suggest we might want to change how many times the function produces output (which was 5 in the earlier example) and the actual output text, which was "hello world" in the original function.

The basic structure of a function is:

```
NameOfFunction <- function(input1, input2,...) {
```

```
doSomething <- ....
return(doSomething)
}
```

Here is the hello world example:

```
HW <- function(n, outtext) {
 for (i in 1:n) {
    print(outtext)
  }
  # return("nothing")
}

# test
HW(5, "Hello World")
```

```
## [1] "Hello World"
## [1] "Hello World"
## [1] "Hello World"
## [1] "Hello World"
## [1] "Hello World"
```

```
# switch input by naming
HW(outtext ="I can switch the inputs", n = 3)
```

```
## [1] "I can switch the inputs"
## [1] "I can switch the inputs"
## [1] "I can switch the inputs"
```

Note that in this case the function produces output as part of its execution rather than returning an actual value (which is why I commented out the `return` statement). In the first example, you can see that you don't have to name the inputs if you keep the inputs in the same order as the defined function. R assumes that you mean `n = 5` and `outtext = "Hello world"`. In the second example I show that you can switch the inpurs if you name them and that the function allows you to choose different inputs.

Exercise

- Can you write a function that calculates y = a*x + b for different values of x, a and b?
- Make the function return the output using return()

R for hydrological modelling (hydromad example)

The below example is a bit more complex but demonstrates the power of packages, and how you can use R to do hydrological modelling using the package hydromad.

The R package Hydromad is an add-in into R that deals specifically with hydrological modelling: http://hydromad.catchment.org/. It incorporates a range of models, and provides tools for calibration, validation and error checking. Installation of Hydromad can be a bit tricky, ask Willem for help if you want to install it on your personal computer:

- In R, install a series of packages: 'install.packages(c("zoo", "latticeExtra", "polynom", "car", "Hmisc"))'
- Follow the instructions on: http://hydromad.catchment.org/#installation to install the main package

On difference in the plotting is that hydromad uses the package lattice rather than ggplot2 (which I demonstrated earlier). Maybe in the future this can be changed to ggplot2, but currently this is not a priority. This means that you will see the functions `xyplot()` and `levelplot` used.

Once the package is installed, load it:

```
require(hydromad)
```

```
## Loading required package: hydromad
## Loading required package: lattice
## Loading required package: latticeExtra
## Loading required package: RColorBrewer
##
## Attaching package: 'latticeExtra'
## The following object is masked from 'package:ggplot2':
##
##     layer
## Loading required package: polynom
## Loading required package: reshape
##
## Attaching package: 'reshape'
## The following object is masked from 'package:dplyr':
##
##     rename
## The following objects are masked from 'package:tidyr':
##
##     expand, smiths
```

And we will use the data from the Cotter river that come with the package

```
data(Cotter)
xyplot(Cotter)
```

GR4J

GR4J is a simple conceptual rainfall-runoff model, developed in France. The associated paper is: Perrin, C., Michel, C., Andr?assian, V., 2003. Improvement of a parsimonious model for streamflow simulation. Journal of Hydrology, 279(1-4): 275-289.

The power of GR4J is that it only has 4 parameters, which you could also consider its limitation. However, the strategic choices within the structure means that the model is highly flexible and easy to fit to all kinds of streamflow data.

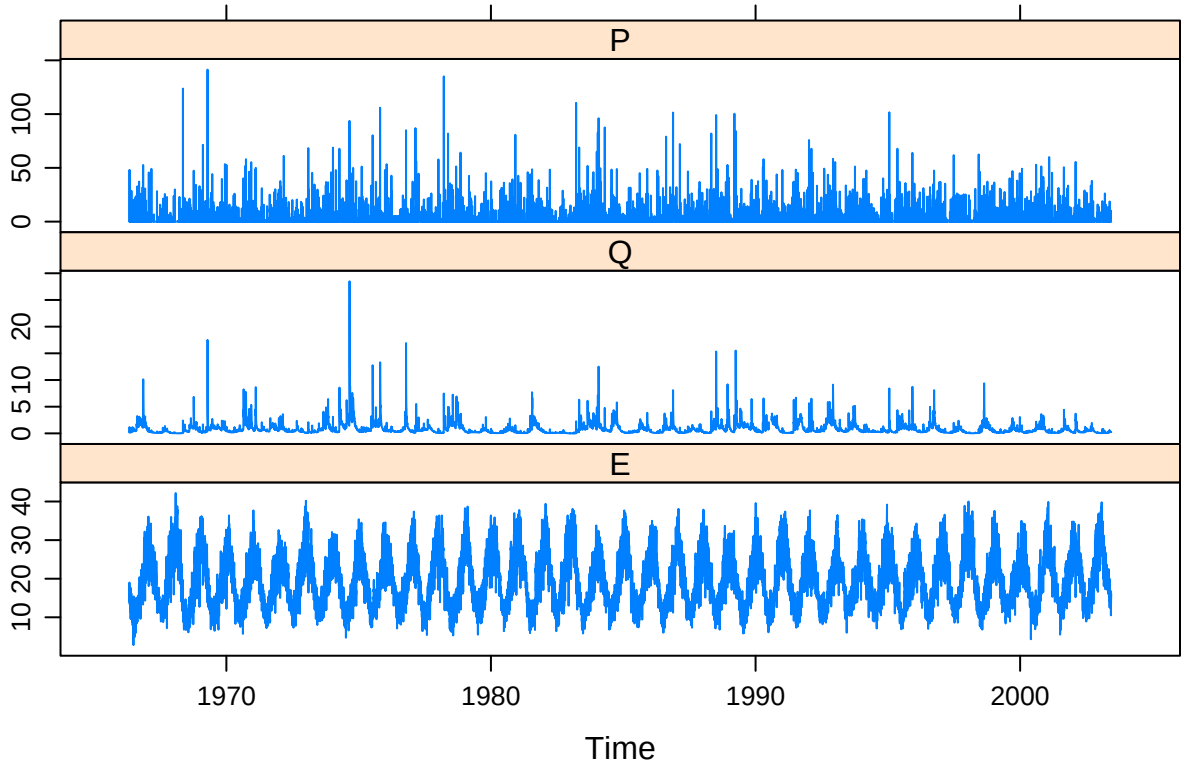
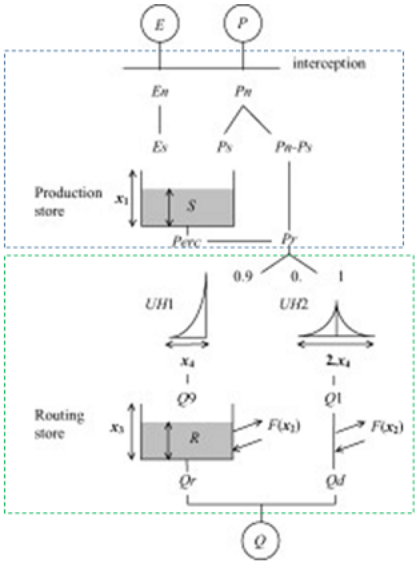


Figure 3: plot of the data in the Cotter data set



SMA

Routing

- Interception and unit hydrographs
- Unit hydrograph concept for runoff
- Losses are possible from the routing
- 4 parameters (x1, x2, x3 and x4
- Non-linear responses

Defining the model

We now need to specify our first model. The beauty of hydromad is that there actually a range of different models that can be applied and these are discussed in much more detail in Andrews et al. (2011). We will just go through the specification of GR4j. The first model we will define is purely for illustration and is actually never fitted.

Note how in the hydromad function the soil moisture accounting (the water balance) component is defined by `sma` which in this case is `gr4j` and the routing component is defined separately as `routing = "gr4jrouting"` .
You could include different `sma` and different `routing`. The etmult is a multiplier to transform maximum temperature into potential ET (PET). This is because maximum daily temperature and PET are highly correlated with a slope of approximately 0.15.

```
# specify a model structure (GR4J) with some arbritrary parameters
CMod <- hydromad(Cotter[1:1000,], sma="gr4j", routing="gr4jrouting",etmult=0.15,
                 x1 = 665, x2 = 10, x3 = 90, x4 = 3.8,
                S_0 = 0.6, R_0 = 0.7)
# in other words, this model is purely arbitrary to demonstrate
# the different components
xyplot(predict(CMod, return_state = TRUE, return_components = TRUE),
       strip = FALSE, strip.left = TRUE)
```

```
# plot this to show the components
```

This figure indicates the different components of the GR4J model. This includes the "effective rainfall" U, which is the rainfall minus the ET, the storage levels in the "production store" or soil bucket, the predicted ET from the soil bucket, the fast flow Xr and the slow flow Xd and finally the levels in the groundwater store. Note the definition of the parameters x1, x2, x3 and x4 which relate to processes in the model, and it might be worth considering how these influence the model.

You can try changing the parameters and see if it changes the model.
We can also ask for the parameters and the information that we have stored in the model object:

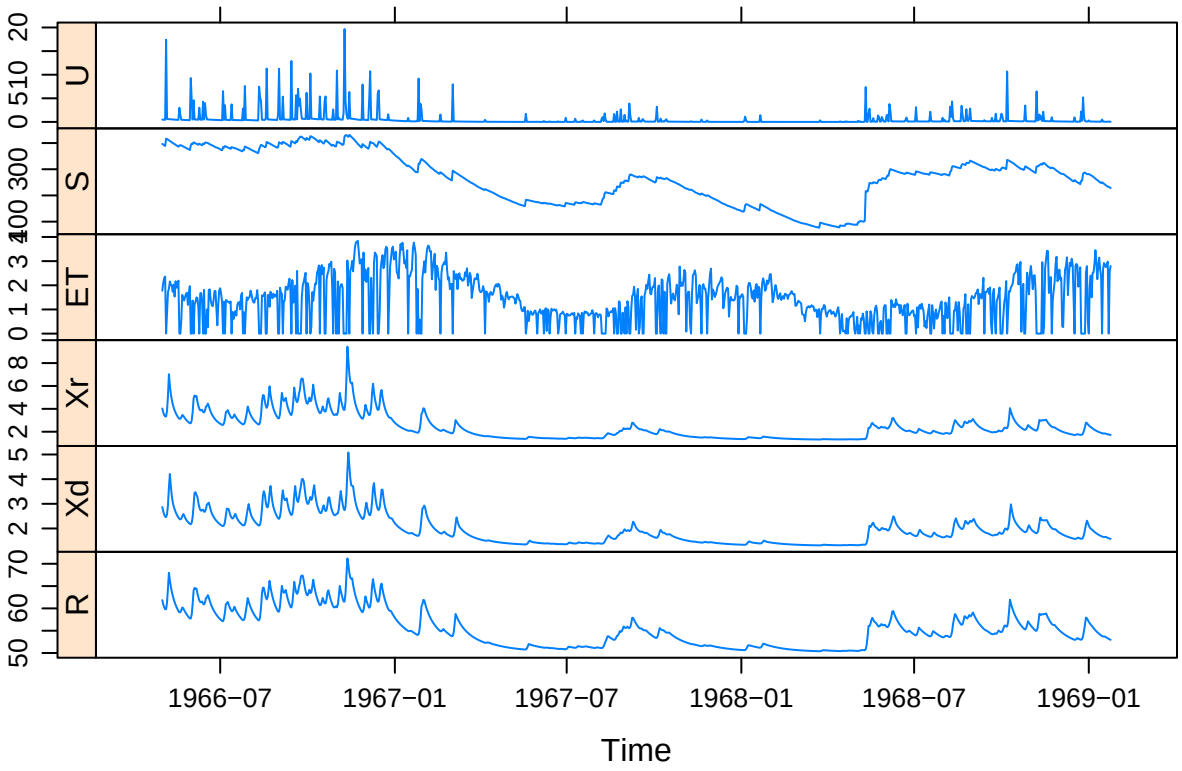


Figure 4: Plot of GR4J with hypothetical parameters

```
print(CMod)
```

```
##
## Hydromad model with "gr4j" SMA and "gr4jrouting" routing:
## Start = 1966-05-01, End = 1969-01-24
##
## SMA Parameters:
##     x1  etmult     S_0
## 665.00    0.15    0.60
## Routing Parameters:
##   x2    x3    x4   R_0
## 10.0  90.0   3.8   0.7
```

The first part of the output from the print statement indicates the SMA parameters that we have entered in the model. Here x1 is set to 665 mm and again the et multiplier is set to 0.15 to convert Maximum Temperature to potential ET. S_0 is just the initial storage level as a fraction of x1, this means the initial S is 0.6*665, you can check this on the figure. The default ranges for all parameters for all models can be found by entering: `hydromad.options()`

The routing parameters relate to the x2 through x4 parameters and are listed in the second part of the output.

```
coefficients(CMod)
```

```
##     x2     x3     x4     x1 etmult    S_0    R_0
##  10.00  90.00   3.80 665.00   0.15   0.60   0.70
```

optimisation using hydromad

We now want to calibrate the model on some data, we will just take the first part of the data so we will use about 7 years

```
# split the data to use for calibration
Data_Cal<- window(Cotter, start = "1970-01-01",end = "1975-12-31")
```

You can now start fitting the model, but this first means we have to give the model some ranges to fit for the parameters rather than single values. I have decided to fit all parameters. The choice of the ranges is somewhat arbitrary, but I have increased the range for x1 to be quite high.

```
CMod <- hydromad(Data_Cal, sma="gr4j", routing="gr4jrouting",
  etmult=c(0.05,0.5),x1 = c(1000,3000), x2 = c(-3,20),
  x3 =c(5,500), x4 = c(0.5,10), S_0 = 0.5, R_0 = 0.5)
```

Then using the function `fitByOptim` we can calibrate the model

```
CotterFit <- fitByOptim(CMod,objective=~hmadstat("r.squared")(Q,X),
                        samples=1000,method="PORT")
```

This fits the model using the "optim" routine in R to optimise the parameters in the model (see `?optim`). `optim()` is quite a complex function, but very useful for all types of optimisation problems. The method `PORT` is one of the specific methods that can be used, but different other methods can be specified. Felix Andrews (Andrews et al. 2011) suggests that this routine is preferred, but you could try another method. The standard `nelder mead` is a regular least squares optimisation.

The fitting gives some output about the iterations it runs through and also possible warnings. You can ask for some information about the model:

```
summary(CotterFit)
```

```
##
## Call:
## hydromad(DATA = Data_Cal, etmult = 0.113754, x1 = 1000, x2 = -0.585523,
##     x3 = 118.879, x4 = 0.822519, S_0 = 0.5, R_0 = 0.5, sma = "gr4j",
##     routing = "gr4jrouting")
##
## Time steps: 2091 (0 missing).
## Runoff ratio (Q/P): (1.191 / 3.18) = 0.3747
## rel bias: 0.004701
## r squared: 0.8071
## r sq sqrt: 0.7538
## r sq log: 0.7087
##
## For definitions see ?hydromad.stats
```

This returns the models fitted values and the different goodness of fit stats. Note that 80.71 percent of the variation in the streamflow is being explained by this model based on the r^2, which is really the NSE (Nash Sutcliffe Efficiency).

The other statistics are the relative bias (basically the mean error divided by the mean flow) which is about 0.0047009. The r.sq.sqrt is the r^2 based on a square root transformation of the data, while the r.sq.log is the r^2 based on a logarithmic transformation of the data. It is suggested that these two statistics are more indicative of the fit on the low flows, while the r^2 is more indicative of the fit of the flow peaks (Bennett et al., 2013). So in this case the model seems to fit the low flows better than the high flows.

The fitted coefficients can be extracted separately by asking:

```
coef(CotterFit)
```

```
##           x2           x3           x4           x1       etmult
##   -0.5855234  118.8785499    0.8225190 1000.0000000    0.1137536
##          S_0          R_0
##    0.5000000    0.5000000
```

Now we need to check how good our fit actually is. We can plot the observed data with the predicted output and the rainfall to see how it looks visually, but this is just showing how we fitted the data.

```
# plot observed vs modelled with the rainfall (Figure 5)
xyplot(CotterFit, with.P=TRUE, xlim=as.Date(c("1970-01-01", "1975-01-01")))
```

```
## Warning in formals(fun): argument is not a function
```

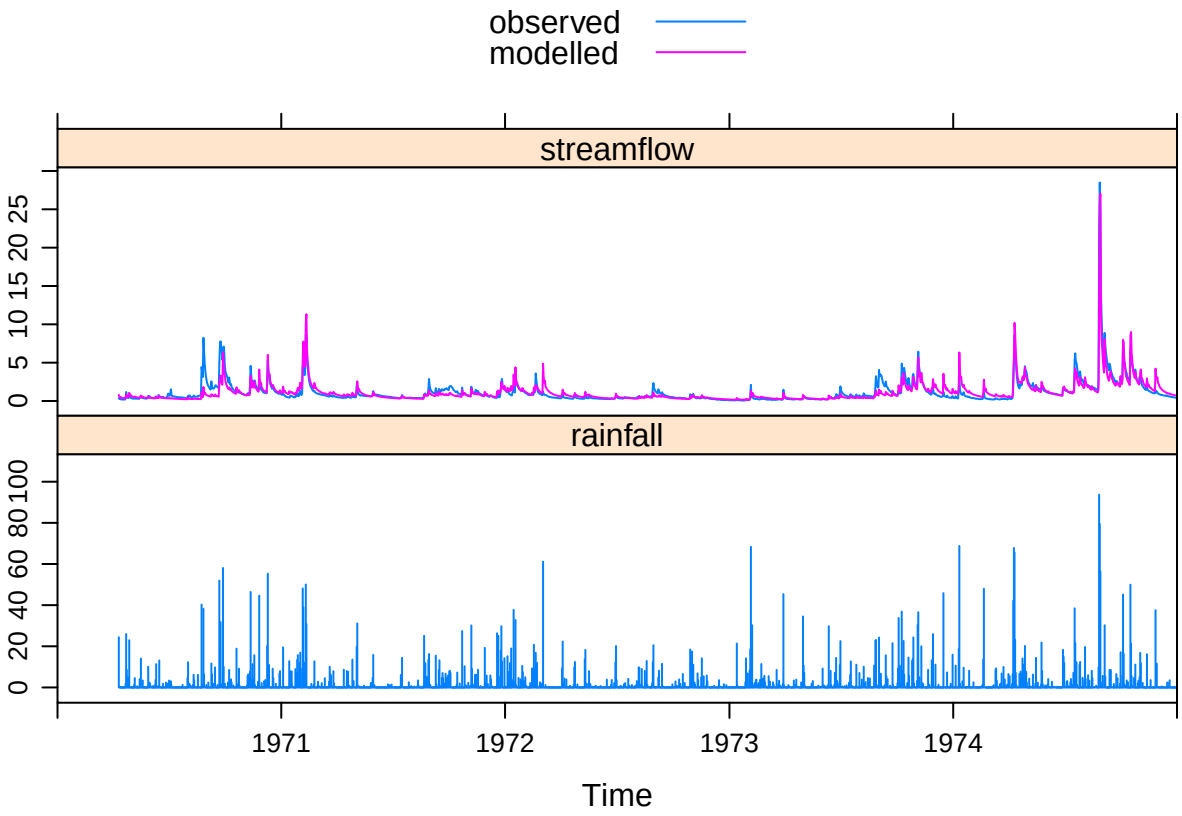


Figure 5: Predicted and observed flow for Cotter for the calibration period

END OF DOCUMENT