A primer on R for hydrologists

R.D. (Dan) Moore[1] and Dave Hutchinson[2]

[1] Department of Geography, University of British Columbia

[2] Environment Canada, Vancouver, BC

2016 August 20

Introduction

The objective of this article is to provide information to assist novice users in getting up and running with R using hydrologically relevant examples. The article ends with an example to illustrate the use of R for infilling missing weather data, a common task in data processing prior to undertaking hydrologic analyses.

How to download and run R

R can be downloaded either as source code, which can be modified prior to compiling, or as pre-compiled binaries for a variety of operating systems, including Unix, Windows and Mac OS. A network of servers worldwide store identical, up-to-date versions of R code, packages, and documentation. These servers are collectively known as the Comprehensive R Archive Network, or CRAN. In western Canada, the appropriate mirror link to download R is http://cran.stat.sfu.ca/.

The basic installation includes a number of packages, which are collections of functions. The *base* and *stats* packages are automatically "attached" to the workspace at the beginning of a user session, and the functions within them are directly available to the user. There are other packages included in the basic installation that must be explicitly attached to the workspace using the *library*( ) function. Of particular interest is the *MASS* package, which provides functions and data sets to support the classic text by Venables and Ripley (2002), *Modern Applied Statistics with S*. There is also a plethora of user-contributed packages that can be downloaded and installed for use.

Once installed, R can be run in several ways. The most direct is to use the command-line interface, in which commands are typed in at the command prompt (indicated by ">"). This approach works well for doing basic calculations (i.e., in place of a calculator) and for typing short commands. However, most applications of R will involve long commands that incorporate lists of options (e.g., to specify plotting options within a graph) and/or multiple lines of code.

One alternative approach to the direct use of the command line is to type the code into a text editor such as Notepad in Windows, emacs in Unix or TextEdit in Mac OS X. You can then select the lines of code you want to run, then copy and paste them in at the command line. For users of Windows-based systems, Notepad++ is a powerful alternative to Notepad. Like R, Notepad++ is available for free under the GNU General Public Licence. Within Notepad++, you can open multiple files at the same time, and it supports code-folding and syntax highlighting for a range of programming languages. The latter features are a great help when writing code. For example, the syntax highlighting will identify the beginning and end of loops and grouped statements, which can be a great help when coding nested loops or other structures. Code-folding provides the ability to compress or expand these structures during program development, which allows the programmer to focus on overall program flow and logic. Notepad++ can be downloaded via the following link: http://notepad-plus-plus.org/.

The use of Notepad++ with R can be further enhanced by using the NppToR plug-in. For example, with NppToR running, one can, by pressing the F8 key or CTRL + F8, submit the current line or the entire code for processing by R. NppToR is available for free under the GNU General Public Licence via the following link: http://sourceforge.net/projects/npptor/.

A number of open-source GUIs have been developed for R, including R Commander, Tinn-R and RStudio.[1] Of the open-source GUIs, RStudio is the most popular among the hydrologists we know who use R, and it is available for Windows, Linux and Mac OS X operating systems. It can be downloaded via the following link: http://www.rstudio.com/.

When writing R code, it is important to remember that the code is case sensitive. That is, for example, a variable named "Qratio" will be considered different from a variable named "qratio."

[1] http://en.wikipedia.org/wiki/R_(programming_language)#cite_note-18

For guidance on naming conventions in R code, and other aspects of formatting code for readability, one can refer to the Google Style Guide for R code:

http://google-styleguide.googlecode.com/svn/trunk/Rguide.xml

Data types and structures within R

R supports three basic data types: numeric, character and factor. Numeric values include both integer and non-integer values. Character values can include, for example, the names of monitoring stations within a data set. Factor values are used to represent categorical variables and are particularly useful for including these variables in statistical analyses. For example, a hydrologist might be interested in comparing April 1 snowpack water equivalent (SWE) between El Niño and La Niña years. In this analysis April 1 SWE would be a numeric variable and the El Niño state a factor variable.

R provides a number of functions for converting data types. For example, the *as.factor*( ) function can be used to convert string or numeric variables into factors; the *as.character*( ) function converts numeric or factor variables to characters; and the *as.numeric*( ) can convert suitable character or factor values into numerical values.

Dates and times can be represented in a variety of ways within R. For example, one could represent a date-time value as a set of numeric values representing the year, month, day, hour, minute and second. Alternatively, it could be represented as a single character value (e.g., "Sept. 8, 2013, 12:45:10") or as a single numeric value representing the time in seconds relative to some base date-time. R includes functions for converting among these various representations. We will devote a separate article to the topic of date-time representation within R, particularly in relation to plotting time-series graphs.

R functions are built around a hierarchy of data structures: vectors, matrices, arrays, data frames and lists. Vectors are structures that contain one or more values, all of which must be of the same type. A typical data set in hydrology might include a set of vectors having the same length, each representing a different variable. For example, a data set might include a vector containing a character variable named "Date" and another containing a numeric variable named "Q." A simple

way to create vectors is to use the "concatenate" function, symbolized by *c*(). For example, we could create a small data set of two vectors by typing the following code at the command prompt:

```
> Date <- c("2010-11-2", "2010-11-3", "2010-11-4", "2010-11-5")
> Q <- c(18.5, 26.9, 23.5, 22.1)
```

Note that "<-" is an assignment operator. An equal sign ("=") can also be used, but is not endorsed by the general R user community. The statements above create two vectors, each with length = 4, that represent a set of dates (as character values) and the mean daily discharge on those dates.

Matrices are rectangular structures with rows and columns. All elements within a matrix must be of the same type. Examples of matrices in hydrology include the values of elevation in a gridded digital elevation model or a data set comprising multiple time series of streamflow, each column representing a different station and each row a different date-time.

Arrays are an extension of matrices that can have more than two dimensions. For example, a gridded climate data product might include time series of daily air temperature at each of a number of grid points over a region. This data product could be stored as a three-dimensional array, with two of the dimensions representing the location of the grid point (e.g., as latitude and longitude) and the third representing time.

Data frames are two-dimensional data structures in which the columns represent different variables and the rows represent different cases or records (e.g., dates or sampling locations). Unlike matrices, the columns can be of different types. Data frames thus allow different variable types to be combined. For example, the variables Date and Q could be combined in a data frame named "df" as follows:

```
> df <- data.frame(Date, Q)
```

Within a data frame, all the columns must have the same length. The *length()* and *dim()* functions can be used to determine the length of a vector or the dimensions of a matrix or array, respectively.

Matrices can be converted to data frames using *as.data.frame*(), and data frames can be converted to matrices using *as.matrix*(). The conversion from matrix to data frame will preserve the data type. However, the conversion from data frame to matrix depends upon the uniqueness of data types stored within the data frame. If the data frame consists of more than one data type, the conversion to matrix will assume the default data type of character to preserve values.

Lists are the most general data structure. They can contain other objects with no restrictions on data type or length, including data frames, vectors, matrices and other lists. Lists are typically used to contain the output from an analysis. For example, the output from a linear regression analysis is a list that contains, among other quantities, the estimated slope and intercept, their standard errors, the coefficient of determination and the residuals from the regression.

Many packages create specialized data classes and objects. For example, the *sp* package provides data classes and methods for handling, processing and displaying spatial data, including points, lines, polygons and grids.

Selecting subsets of data structures

R uses several constructs to access individual elements or subsets of data structures. For vectors, matrices, and arrays, the most common method to access individual elements or subsets is to use square brackets, as in:

```
> x[i]      # to access the ith element of a vector
> x[1:2]    # to access the first two elements of a vector
> x[4:7]    # to access elements 4, 5, 6 and 7
> y[i, j]   # to access the ith row and jth column of a matrix
> y[, j]    # to access the jth column of a matrix
> y[i, ]    # to access the ith row of a matrix
```

Using a negative index (or sequence of negative index values) will cause values to be left out of the structure. Some examples follow:

```
> x[-1]     # to access all but the first element of a vector
```

```
> x[-1:-2]  # to access all but the first two elements of a vector

> y[-1, ]   # to access all but the first row of a matrix
```

The following code illustrates how to generate a lagged version of a vector by deleting the last value of the vector and then adding a missing value (NA) to the beginning:

```
> n <- length(x)

> xlag <- c(NA, x[-n])
```

Subsets of a vector can be accessed based on values in another vector. For example, if you have two vectors of equal length, "Q" containing a time series of discharge and "Year" containing the year, you can extract subsets of Q as follows:

```
> Q[Year == 1999] # to extract streamflow for the year 1999

> Q[Year < 1999]  # to extract streamflow for the years prior to 1999
```

In the first line of code, above, note that the double equal sign ("==") is used to test for equality. To access individual components of lists (or data frames), R uses double square brackets (e.g., [[ ]]) or the dollar sign ('$'). For example, the column containing a variable named "Date" within a data frame named "df" could be accessed by one of the following two expressions:

```
> df$Date         # using $ for obtaining the column
```

```
> df[[‘Date’]]    # using [[ ]] for obtaining the column
```

Subsets of data frames can be extracted using the *subset*( ) function. For example, one could extract data from a data frame named "df" for the month of June as follows (assuming the data frame contains a numeric variable named "Month"):

```
> dfJune = subset(df, Month == 6)
```

Handling missing data

Missing data are a fact of life in most hydrologic data sets. In R, such values are coded as NA. Many functions, when applied to a set of values that include any NA values, will return the value NA unless there is an option within the function to specify alternative handling of NA values.

For example, suppose one has the following vector of data and applies the mean function:

```
> a <- c(1, 3, 8, 15, 24, NA)

> a.mean <- mean(a)
```

The value of *a.mean* will be NA by default. To compute the mean of the non-NA values of *a*, one must use the "na.rm" option:

```
> a.mean <- mean(a, na.rm = TRUE)
```

When na.rm is TRUE, the function will remove the NA values prior to computing the mean. Thus, the value of *a.mean* would be 10.2. The example provided at the end of the article illustrates the handling of NA values when performing linear regression.

Understanding and working with "vectorized" functions

An important feature of R's computational functions is that they are "vectorized," which facilitates the writing of compact code, particularly for operations involving summations. In fact, the clever use of vectorized code can help the user avoid using looping structures, which slow execution. As an illustration of vectorized code, suppose the user has defined two vectors, a and b, and then computes their sum and product:

```
> a <- c(1, 3, 8, 15, 24)

> b <- c(6, 5, 2, 12, 9)

> c <- a + b

> d <- a*b
```

The resulting vector, *c*, would be computed by the element-wise addition of the elements of *a* and *b*, resulting in the vector *c* containing the elements 7, 8, 10, 27, 33. Similarly, the vector *d* would contain the element-wise products of the elements in *a* and *b*: 6, 15, 16, 180, 216. Taking advantage of this of vectorized code, one could compute the standard deviation of a vector, *x*, as follows:

```
> n <- length(x)     # length of vector

> m <- mean(x)       # mean of vector

> stdev <- sqrt(sum((x - m)^2)/(n - 1))
```

The expressions above have used the built-in functions *length*( ), *mean*( ), *sum*( ) and *sqrt*( ), the latter of which computes the square root of the argument. This example is, of course, trivial because R already provides the *sd*( ) function for computing standard deviations directly. The example above also illustrates the incorporation of comments in R code. Anything following the hash symbol (#) on a line is treated as a comment and not processed. If a hash symbol is in the first column, the entire line is treated as a comment.

A powerful example of vectorized code is the *ifelse*( ) function, which is similar to the *if*( ) function in Excel. In many applications, streamflow data should be analysed by water year (Oct. 1 to Sept. 30) rather than calendar year. If the data set contains the variables Year and Month (represented as numbers, with Month = 1 for January), the water year (WY) for each case in the file can be computed simply by inserting the following line of code:

```
> WY <- ifelse(Month > 9, Year + 1, Year)
```

User-defined functions and running code from a separate file

One really useful aspect of R is the ability to create user-defined functions to generate re-usable code. For example, in many analyses of time series data, it is necessary to generate lagged

versions of a variable. Box 1 includes a user-defined function to create a version of a variable that is lagged by "k" time steps. Note that the body of the function is contained within curly braces "{" and "}". There are two arguments to the function: a vector "x" to be lagged, and the number of time steps to be lagged, "k". The function can accommodate positive lags (the copy is lagged by *k* time steps relative to the original) and negative lags (the copy is advanced *k* time steps). The function illustrates the use of the *if* and *else* statements for conditional execution.

Once the code containing the function has been run, the function will be attached to the work space and available for use. The following code illustrates the application of the function from the command line:

```
> y <- seq(1, 10, 1)    # create a vector of digits from 1 to 10

> ylag4 <- xlag(y, 4)   # create a version lagged four time steps

> yadv4 <- xlag(y, -4)  # create a version advanced four time steps
```

The result of the second line is a vector named *ylag4* containing the following values: NA, NA, NA, NA, 1, 2, 3, 4, 5, 6. The result of the third line is a vector named *yadv4* containing the following values: 5, 6, 7, 8, 9, 10, NA, NA, NA, NA. Note that there is a *lag*() function in the *stats* package that is equivalent to the user-defined *xlag*() function, except that it works only on data structures of the *ts* (time series) class.

Another useful feature is the ability to run code in a file using the *source*() function. If you are working from the command line, you can run all the code contained in a file as follows:

```
> source("pathname")
```

where *pathname* is the path to the file, including the root and sub-directories as appropriate. You can also execute the *source*() command from within an R script. This can be useful if you have a set of functions to be used within one or more R scripts. The code for the functions can be bundled within a single text file and then executed from another script to attach the functions for use within the work space. Another useful application of the *source*() command is that it allows one to separate different components of a project's workflow. For example, scripts for pre-

processing data could be in one file, scripts to conduct an analysis could be in another, and scripts to generate report-quality graphs could be in another. All of these could be executed from a master script via calls to the *source*() function.

Where to get help

Learning R can be a challenge. Fortunately, a range of resources are available, including a number of manuals that are available for free via links on the R web site (http://cran.r-project.org/). Of particular interest to novice R users are *An Introduction to R* by Venables et al. (2013), *R for Beginners* by Paradis (2005), *Using R for Data Analysis and Graphics - Introduction, Examples and Commentary* by Maindonald (2008), and the *R Reference Card* (Version 1 by Short, 2004; Version 2 by Baggott, 2012).

If you know the name of the command that you are seeking help about, you can view the built-in help page by using one of the following commands (used here for the function *lm*()):

```
>?lm

>help(lm)
```

If you do not know the name of the function, a broader search within the help pages can be conducted using key words. For example, use one of the two following commands to find functions with the term "skewness" in the help page:

```
>??skewness

>help.search("skewness")
```

Note, however, that the above searches will only find instances of "skewness"; they will not locate instances of "skew" in the help pages. In our experience, the built-in help pages are most useful to users after they have gained some experience with R.

The mailing lists (http://www.r-project.org/mail.html) provide another source of help. Users can send questions to the lists via email to elicit responses from the R user community. Be forewarned that many members of the R community can be blunt in their responses, especially when it is clear that the author of the question has not done his or her homework before posting. Be sure to read the posting guide (http://www.r-project.org/posting-guide.html) before sending

any questions to the list. An invaluable source of information is the Rseek search engine (tp://www.rseek.org/), which will search a number of R-related sites, including the help pages and the mail lists.

Example: using linear regression to infill missing air temperature data

A common step in the processing of data in preparation for hydrologic analyses is to estimate or infill missing values, especially for weather data. A range of approaches can be used. If one or more weather stations are relatively close to the station being used in the analysis, a simple but popular approach is to develop regression relations between weather variables at the station of interest as a function of the corresponding variables at nearby stations. The script presented in Boxes 2 to 4 illustrate the use of linear regression to infill missing data at a high-elevation station (Cypress Bowl, located north of Vancouver in the North Shore mountains) as a function of a low-elevation station (Vancouver International Airport). The data files and the script are available via the following web site:

www.geog.ubc.ca/~rdmoore/Rcode.htm

The first part of the script (Box 2) causes the data files, in Canadian Climate Centre format, to be read in from a web site and manipulated into data frames using the *read.msc*( ) function in the user-contributed *seas* package[2]. The data frames are then merged using the *merge*( ) function to line up the time series with a common time base. The script then computes the mean daily air temperatures at the two stations and converts the date variable from character to an ISO date-time variable. It then extracts the month from the ISO date-time variable to allow analyses to be conducted separately for each month.

The second part of the script (Box 3) fits a linear regression relation between mean daily air temperature at Cypress Bowl as a function of mean daily air temperature at YVR using all of the

[2] Packages are available from CRAN, the Comprehensive R Archive Network. Type `install.packages('seas')` at the R command prompt to install the package (your computer must be connected to the internet).

data. It generates a scatterplot and a box plot of the residuals by month. As can be seen in Figure 1, there is a seasonal pattern to the residuals, indicating that separate regressions by month would be more appropriate. These are illustrated in Figure 2.

As mentioned earlier, the result of a function like *lm*( ) is a list. In this specific example, the object is named *mod.lm.* This list contains a number of results from the linear model fit, including estimated coefficient values and their standard errors, p values, and the coefficient of determination. These can be displayed using the command *summary*(*mod.lm*). If you want to extract any of these values for further analysis, you can store the summary in another object as follows:

```
> mod.lm.sum <- summary(mod.lm)
```

Then, for example, to extract the fitted coefficients and store them in a vector named *b*, and the standard errors of the coefficients into a vector named *sb*, use the following commands:

```
> b <- mod.lm.sum$coef[, 1]

> sb <- mod.lm.sum$coef[, 2]
```

To see a listing of all the quantities that can be accessed, type

```
> names(mod.lm.sum)
```

Alternatively, a number of results can be extracted from *mod.lm* using the *objectname*$*component* construction. For example, to extract the estimated coefficients into a vector named *b*, type:

```
> b <- mod.lm$coef
```

Other quantities that can be extracted include the residuals (*mod.lm*$*residuals*) and the fitted (predicted) values (*mod.lm*$*fitted.values*). To see listing of all of the quantities that can be accessed, use the command:

```
> names(mod.lm)
```

An alternative, and more detailed, approach to view the structure of an object is the *str*( ) function.

The third part of the script (Box 4) fits a linear model with an interaction term between the mean daily air temperature at YVR and month, which requires the conversion of month from a numeric to a factor variable using *as.factor*( ). This model effectively fits a separate regression for each month. In this application of the *lm*( ) function, the *na.action* option is set to *na.exclude*. When this option is used, the subsequent application of the *predict*( ) and *residuals*( ) functions to generate predicted values and residuals will pad out these vectors with NA values so that they have the same length as the vectors for observed values and the predictor variables. The default option for *na.action* is *na.omit*. When the *na.omit* option is used in the application of *lm*( ), predicted values are not padded.

The final part of the script in Box 4 uses the *predict.lm*( ) function to generate predicted values for all values of the variable *x*, not just those values used to fit the regression. The script uses the *ifelse*( ) function to create a new vector to hold the infilled time series (*yfill*), in which NA values in the observed time series are replaced by values predicted by the regression The script generates time series graphs of the observed and infilled (Figure 4).

Summary

This article has provided information required to download and run R, and has explained basic aspects of data types and syntax to assist novice users in using R. An example of a realistic data processing application has been provided to illustrate a range of the functions that are useful in analysing hydrologic data sets. For example, a *for* loop is used to analyse the data by month, with index values being used to select the appropriate subsets of the data for each month. The *expression* statement is used in *plot*( ) commands to include subscripts, superscripts and special symbols in axis labels. It can also be used for legends, or for any text added to a figure. Plotting windows are set up for multi-panel graphs using *par* options.

Box 1. A user-defined function to create a version of a variable lagged or advanced by "k" time steps.

```
xlag = function(x, k) {

  # x is a time series vector to be lagged/advanced
  # k is the lag (positive for lag, negative for advance
  # the function returns the lagged/advanced time series

  n = length(x)
  if (k > 0) {
    # lag time series
    xlk <- c(rep(NA, k), x[1:(n - k)])
  } else if (k < 0) {
    # advance time series
    xlk <- c( x[abs(k-1):n], rep(NA, abs(k)) )
  } else {
    # k = 0, return original series
    xlk <- x
  }
  return(xlk)
}
```

Box 2. The first part of a script to infill missing air temperature data. This component reads in the data and computes mean daily air temperature for two weather stations, Vancouver International Aiport and Cypress Bowl.

```
# infill-example.r
#
# Code to infill missing air temperature data at Cypress weather station
# (elevation = 930 m, station id = 1102253) based on the record at YVR
# (elevation = 4.30 m, station id = 1108447) using linear regression.
#
# This code first explores the use of a single regression for the whole
# year, then fits a model including the interaction with Month, which
# effectively fits separate regressions by month.
#
# NB: The "seas" package must be installed to run the script.
#
# 2013 Dec 4 - RD Moore
################################################################################

# start by clearing out workspace
rm(list = ls())

# load package "seas" to parse Canadian Climate Centre files
library(seas)

# read daily met data for YVR and Cypress into data frames and then merge
yvr.file = "http://www.geog.ubc.ca/~rdmoore/rcode/YVR_DLY04.txt"
cyp.file = "http://www.geog.ubc.ca/~rdmoore/rcode/Cypress_DLY04.txt"
yvr <- read.msc(file = yvr.file, flags = TRUE)
cyp <- read.msc(file = cyp.file, flags = TRUE)
yvrcyp <- merge(yvr, cyp, by = "date", all.x = TRUE,
                suffixes = c(".yvr", ".cyp"))

# convert the date character string into an ISO date-time variable
t.ISO <- strptime(yvrcyp$date, format = "%Y-%m-%d")
Month <- t.ISO$mon + 1   # extract month from t.ISO

# compute daily mean temperature for YVR (x) and Cypress (y)
x <- 0.5*(yvrcyp$t_max.yvr + yvrcyp$t_min.yvr)
y <- 0.5*(yvrcyp$t_max.cyp + yvrcyp$t_min.cyp)
```

Box 3. The second part of a script to infill missing air temperature data. This component explores the use of a linear regression fitted to the entire data set, then fits separate regressions for each month.

```
#############################################################################
# fit a single regression to all the data
#############################################################################

# set up plot window for two graph panels
# first save default plot parameters

old.par = par()
windows(width = 7, height = 4) # open new window
par(mfrow = c(1, 2)) # one row, two columns

# scatterplot using different symbols for each month
plot( x, y,
  type = "p", pch = Month,
  xlab = expression("YVR "*italic(T[mean])*" ("*degree*"C)"),
  ylab = expression("Cypress "*italic(T[mean])*" ("*degree*"C)")
)

# fit a regression line using na.exclude in lm() to account for missing data
mod.lm = lm(y ~ x, na.action = na.exclude)
abline(mod.lm, col = "red")  # adds regression line to existing scatterplot
ypred = predict(mod.lm)
yres = residuals(mod.lm)

# boxplot of residuals by month
boxplot(yres ~ Month,
  ylab = expression(italic(y)*" - "*italic(hat(y))*" ("*degree*"C)"),
  xlab = "Month"
)
abline(h = 0, lty = 3) # add horizontal line at y = 0 for visual reference


#############################################################################
# include interaction between x and Month in linear model to fit
# separate regressions; generate scatterplots by month;
# use na.exclude to account for NA values
#############################################################################

mod.lm.2 = lm(y ~ x*as.factor(Month), na.action = na.exclude)
windows(width = 7, height = 7)
par(mfrow = c(4, 3), mar = c(4,4,1,1))
for (im in 1:12) {
  plot( x[Month == im], y[Month == im],
    xlab = expression("YVR "*italic(T[mean])*" ("*degree*"C)"),
    ylab = expression("Cypress "*italic(T[mean])*" ("*degree*"C)")
  )
  abline( lm(y[Month == im] ~ x[Month == im]), col = "red" )
  legend( "topleft", bty = "n", legend = month.abb[im])
}
```

Box 4. The third part of a script to infill missing air temperature data. This component applies the fitted model and then uses predicted values to replace NA values.

```
#############################################################################
# generate infilled data set by replacing NA values with predicted values
#############################################################################

# predict y using model for all values of x
ypred2 <- predict.lm(mod.lm.2, new = data.frame(x))

# replace NA values in y with ypred2
yfill <- ifelse(is.na(y), ypred2, y)

# restrict time period for plotting
t1 = ISOdatetime(1984, 1, 1, 0, 0, 0)
t2 = ISOdatetime(2001, 12, 31, 0, 0, 0)
yplot = yfill[t.ISO > t1 & t.ISO < t2]
tplot = t.ISO[t.ISO > t1 & t.ISO < t2]

# create tick locations for time axis
year.plot = seq(1984, 2003, 1)
year.tick = ISOdatetime(year.plot, 1, 1, 0, 0, 0)

par(old.par) # restore original graphing parameters
windows(width = 7, height = 4)
plot( tplot, yplot, type = "l", col = "gray",
  ylim = c(-20, 30), # y axis limits
  xlim = c(t1, t2),  # x axis limits
  ylab = expression("Cypress "*italic(T[mean])*" ("*degree*"C)"),
  xlab = "", xaxt = "n" # suppress x axis
)

# add x axis
axis(side = 1, at = year.tick, labels = as.character(year.plot))

# plot actual time series over infilled series
lines(t.ISO, y, col = "black")

# add legend to bottom of graph, no box, two columns
legend("bottom", ncol = 2, bty = "n",
  lty = 1, col = c("black", "gray"), legend = c("observed", "infilled")
)
```

Figure 1. Scatterplot of mean daily air temperature at Cypress Bowl as a function of mean daily air temperature at Vancouver International Airport (YVR) (left), and boxplot of regression residuals by month (right). In the scatterplot, different symbols are used for each month, and the red line is the fitted regression relation.

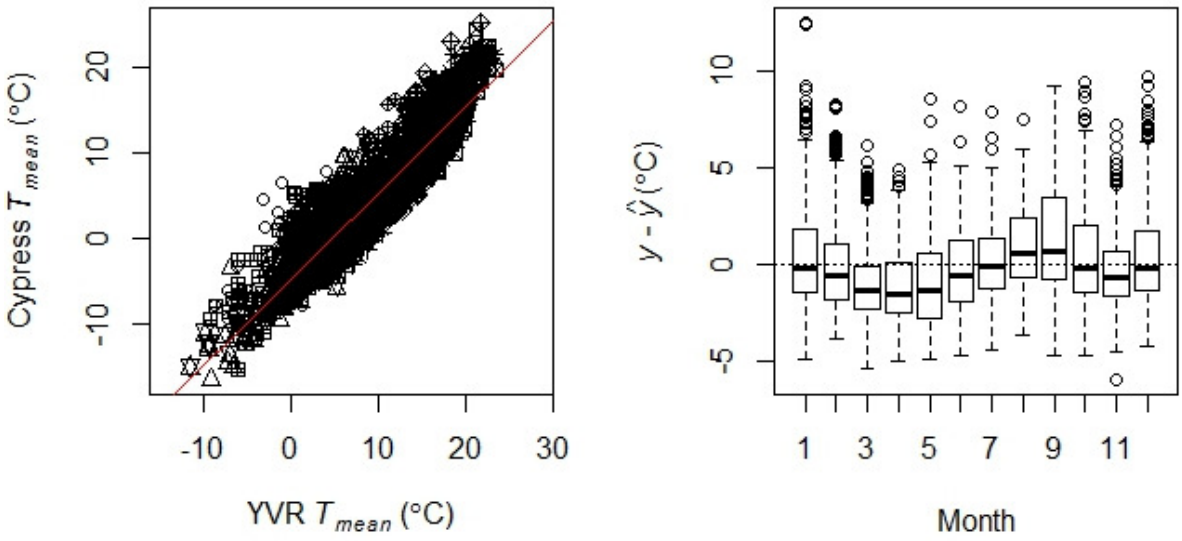

Figure 2. Scatterplot of mean daily air temperature at Cypress Bowl as a function of mean daily air temperature at Vancouver International Airport (YVR) for each month. The red lines are the fitted regression relations.

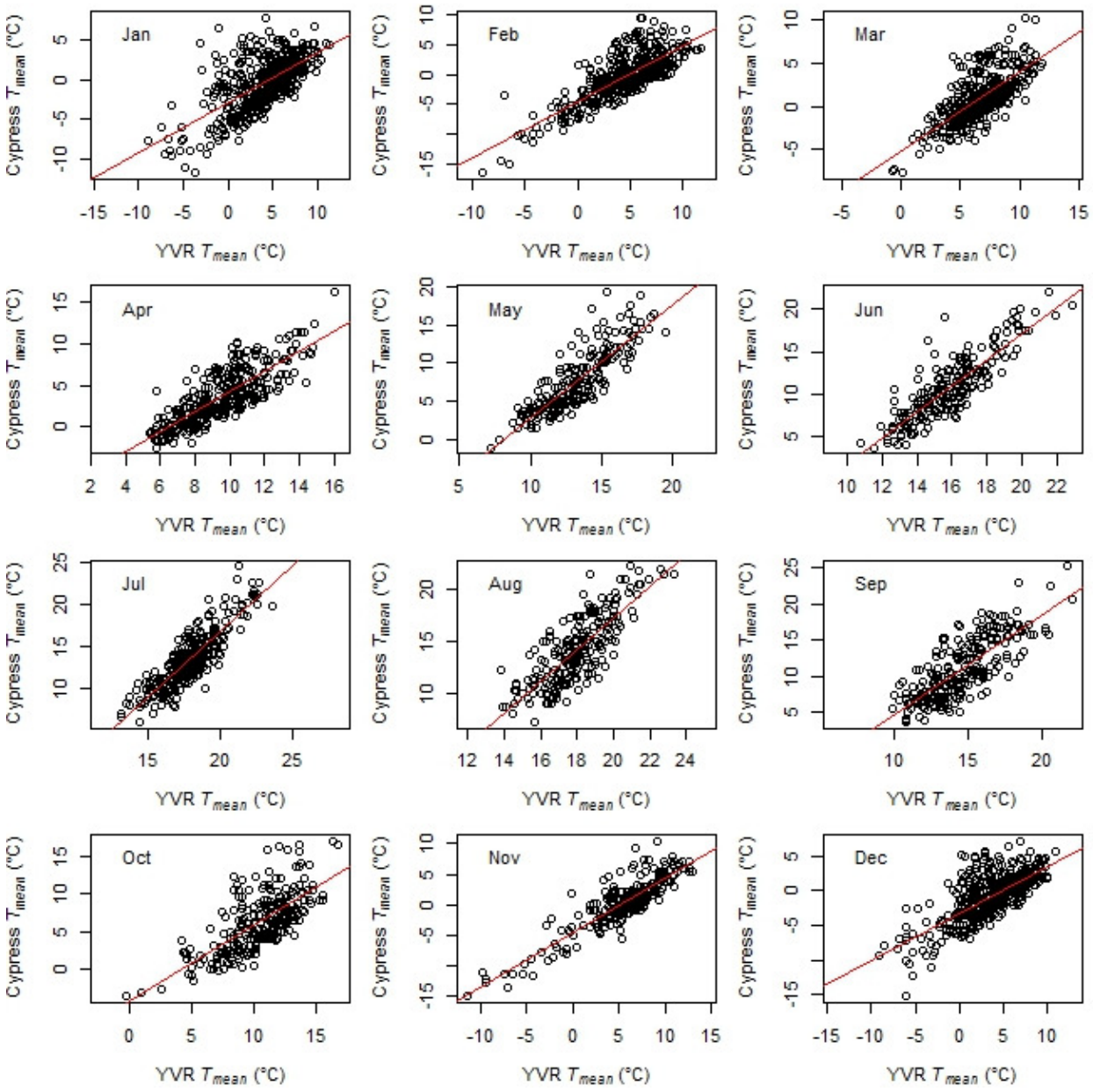

Figure 3. Time series of mean daily air temperature at Cypress Bowl. Black line is observed; grey line is infilled.