

Starting with R and RStudio: A brief introduction for civil engineering courses

After completing this short tutorial, you will be able to:

- Install packages and load libraries into R
- Read data into R from csv or xlsx files
- Perform calculations using R
- Build a simple function, with a *for* loop
- Understand different data types: lists and data frames
- Create a plot with professional axis labels and a legend

This tutorial uses two input data files for water properties and atmospheric properties: **water_properties_SI.csv** and **standard_atmosphere_SI.csv**. If not provided, they can be found [here](#). They will be copied into your working directory (defined below).

The first six pages of this introduction draw from two excellent sources: Lovelace and Cheshire, (2014) and Cleveland (2018). R is an open-source scripting language. Open-source means anyone can contribute to the development of the code, and many thousands of people have. It also means it is free to use, so in addition to being installed on the engineering computer system, you can install it on your personal computer (using any operating system). As you will see, it looks very much like Matlab in many ways, and can do almost anything you can imagine. It is widely used in academic and professional settings around the world.

It is a scripting language, meaning you execute commands and the output can be used in following commands. What follows is a short run-through of an exercise to illustrate some of the capabilities of R using RStudio. RStudio is an interface for using R that is also free and makes it much more accessible. If you are installing these on your own computer, first install R (from <https://cran.r-project.org/>) then install RStudio (<https://www.rstudio.com/>).

Before continuing, watch a Youtube video describing R and RStudio, like [this one](#). After watching that, you'll be familiar with the interface and basic functions. Other online references (noted in the video) that are helpful are [Quick-R](#), [R-bloggers](#), and [stackoverflow](#). Google searches for R techniques often land at one of these.

Remember that, as with any programming language, there are often many ways to produce the same output in R. The code presented in this document is not the only way to do things. You should play with the code to gain a deeper understanding of R. Do not worry, you cannot 'break' anything using R and all the input data can be re-loaded if things do go wrong.

Table 1 shows water properties from the *water_properties_SI.csv* file, similar to standard reference texts.

Table 1 - Water properties, from MWH's Water Treatment: Principles and Design, Third Edition, Wiley & Sons.

Physical properties of water (SI units)

Temperature T (°C)	Specific Weight γ (kN/m ³)	Density ^a ρ (kg/m ³)	Dynamic Viscosity ^b μ ($\times 10^{-3}$ kg/m·s)	Kinematic Viscosity ν ($\times 10^{-6}$ m ² /s)	Surface Tension ^c σ (N/m)	Modulus of Elasticity ^a E ($\times 10^9$ N/m ²)	Vapor Pressure P _v (kN/m ²)
0	9.805	999.8	1.781	1.785	0.0765	1.98	0.61
5	9.807	1000.0	1.518	1.519	0.0749	2.05	0.87
10	9.804	999.7	1.307	1.306	0.0742	2.10	1.23
15	9.798	999.1	1.139	1.139	0.0735	2.15	1.70
20	9.789	998.2	1.002	1.003	0.0728	2.17	2.34
25	9.777	997.0	0.890	0.893	0.0720	2.22	3.17
30	9.764	995.7	0.798	0.800	0.0712	2.25	4.24
40	9.730	992.2	0.653	0.658	0.0696	2.28	7.38
50	9.689	988.0	0.547	0.553	0.0679	2.29	12.33
60	9.642	983.2	0.466	0.474	0.0662	2.28	19.92
70	9.589	977.8	0.404	0.413	0.0644	2.25	31.16
80	9.530	971.8	0.354	0.364	0.0626	2.20	47.34
90	9.466	965.3	0.315	0.326	0.0608	2.14	70.10
100	9.399	958.4	0.282	0.294	0.0589	2.07	101.33

Now suppose you need to estimate the density of water at 44 °C. The two pairs of temperature and density that surround 44 °C are (40 °C; 992.2 kg/m³) and (50 °C; 988.0 kg/m³). So, to estimate the unknown density using linear interpolation you can apply the following:

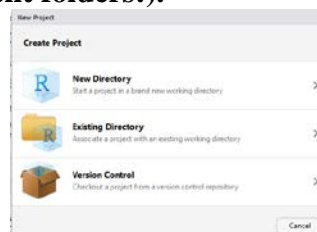
$$\rho' = 992.2 + (988.0 - 992.2) \frac{(44 - 40)}{(50 - 40)} = 990.52$$

This may need to be done many times in the course of hydraulics calculations, so automating it with a script will be very useful. Steps to do that follow:

Open a new project in RStudio and testing commands

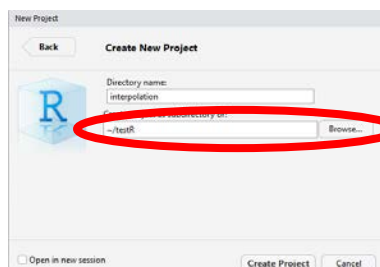
→ **IMPORTANT!** If you are using R on a university lab computer (or other system where you do not have administrative privileges), follow the steps in **Appendix 1** prior to continuing.

Open RStudio, and start the new project in a new directory (this is a good practice – **different projects should always be in different folders!**).

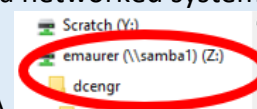


Use RStudio's projects to organize your R work. Place your files into sub-folders (e.g. code, input-data, figures).

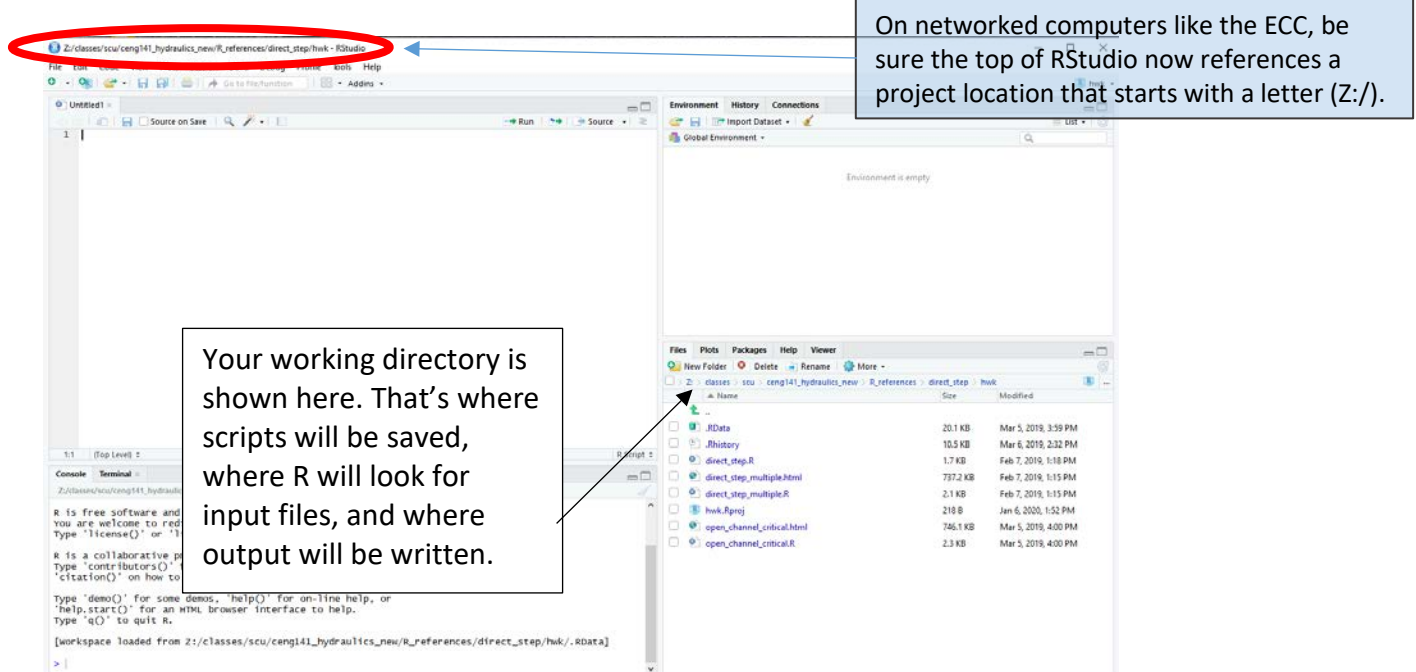
Then create the new directory name (which becomes the project name):



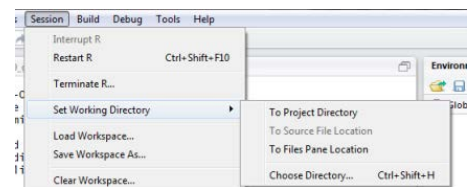
Browse to the folder in which the new directory will be created. If you are on a networked system like the ECC, navigate to **Z:\dcengr** and then to the root folder for the new project directory.



Now you'll see the blank RStudio interface you'll recognize from the video.



You can use the 'Session' menu to change the working directory if you need to:



Now type a couple of lines of code into the console:

```
#test commands
x <- 1
print(x)
```

R commands are all shaded in this document. They can be copied and pasted into R to reproduce everything that follows. Sometimes symbols like "<-" or quotation marks need to be deleted and retyped, though.

Notice the <- is used instead of = to assign values to a variable in typical R code (either <- or = will work, though). Lines starting with # are comments to make the code understandable, but are ignored by R. Adding comments to your code is essential; make these meaningful so you remember what the code is doing when you revisit it at a later date. A few pointers to create good code (for example your R script) and documentation.

- At the beginning include a description of what the script does.
- Load the required packages and describe in a couple of words what each is for.
- Break your code into logical pieces and include comments at the start of each.
- If you have functions defined, place them at the beginning of the script
- Use consistent variable name and code style.
- If you repeat commands more than twice, use a loop or an *apply* command instead.

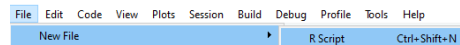
If you will hand in your code and what it produces as a deliverable product, see Appendix 2.

If you require help on any function, use the help command, e.g. `help(plot)` or the help through RStudio. Because R users love being concise, this can also be written as `?plot`. Feel free to use it at any point you would like more detail on a specific function (although R's help files are famously cryptic for the un-initiated). Help on more general terms can be found using the `??` symbol. To test this, try typing `??regression`.

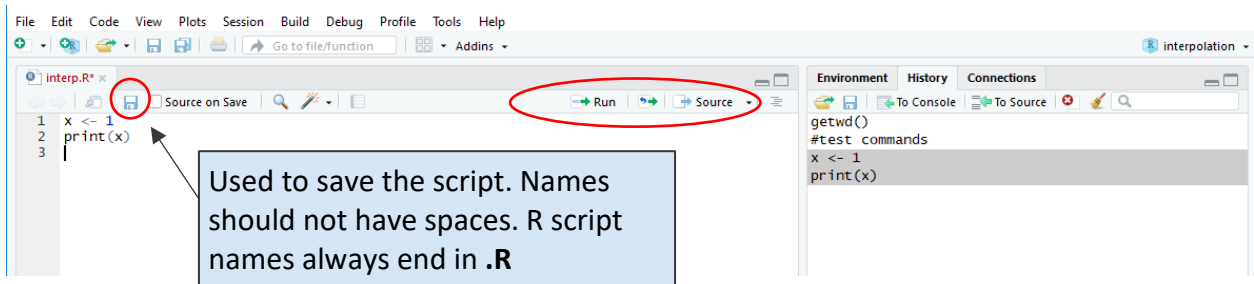
Starting an R script

Now click the History tab in the upper right, highlight a couple of the commands you just entered, and press "To Source."

A new R script can also be opened with File→New File→R Script.



The commands will pop into the upper left panel, which is where you'll create your R script. Click the save icon for the new script and rename it from Untitled to something like "interp.R" (always end R scripts with ".R"). The new name appears on the tab.



You can highlight those lines in the script and click "Run" to execute them, which is a good way to try out your code. If you click "Source" it runs all of the commands in the R script. Try that out and see what happens. Now delete those two lines of code as they aren't very useful.

Returning to our example, the first thing to do for interpolating values is to get the data from the table into R. That could be done by typing them in like this (notice the "c" in front tells R to concatenate the values into a list):

```
temps <- c(0,5,10,15,20,25,30,40,50,60,70,80,90,100)
densities <- c(999.8,1000,999.7,999.1,998.2,997,995.7,992.2,988,983.2,977.8,971.8,965.3,958.4)
```

You can copy and paste these into your R script, highlight them and click Run to create these variables. You can look at these in the Environment tab on the upper right panel:

Values	
densities	num [1:14] 1000 1000 1000 999 99...
temps	num [1:14] 0 5 10 15 20 25 30 40...

Now any one of these can be accessed by an index number (in R the first item in a list is numbered 1, as you might expect):

```
densities[2]
```

```
[1] 1000
```

You can make these look like a table by binding them together as columns with `cbind`, and to be sure they have the same number of elements:

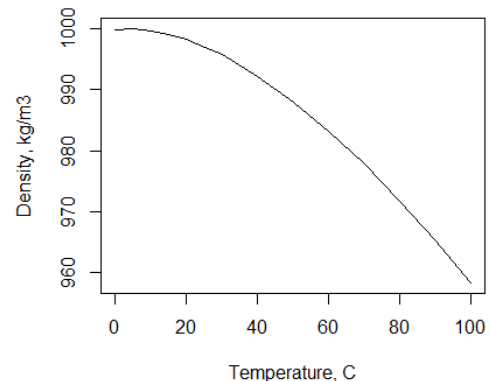
```
> cbind(temps,densities)
      temps densities
[1,]      0    999.8
[2,]      5   1000.0
[3,]     10    999.7
[4,]     15    999.1
[5,]     20    998.2
[6,]     25    997.0
[7,]     30    995.7
[8,]     40    992.2
[9,]     50    988.0
[10,]    60    983.2
[11,]    70    977.8
[12,]    80    971.8
[13,]    90    965.3
[14,]   100    958.4
```

A first plot

How does density vary with temperature? Plot it to see, by executing this command:

```
plot(temps,densities,xlab="Temperature, C",ylab="Density, kg/m3", type="l")
```

The plot will appear in the lower right panel:



That's helpful, but not something that can be turned in.

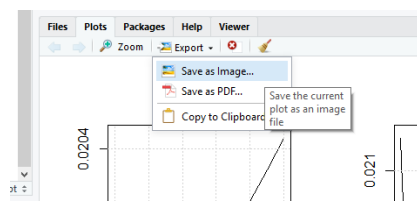
To plot the degree symbol, superscripts, and other special characters is easy (for example, try a google search of "R plot degree symbol").

One way to create better annotation uses *expression*:

```
xlbl <- expression("Temperature, " (degree*C))
ylbl <- expression("Density," ~rho~ (kg/m^{3}))
plot(temps,densities,xlab=xlbl,ylab=ylbl, type="l")
```

Try that to see what happens. Greek letters are simply spelled out; superscripts are preceded by ^, subscripts are enclosed by [], the tilde (~) adds a space. You can of course change axis limits, fonts, colors, etc. Try adding gray gridlines with `grid()` to make the plot more readable.

When you have a final plot you want to include in another report (importing into Word, for example), you can export it from RStudio using the Export button, which allows you to save a plot as a JPG, PNG, PDF, or any of many other options.



Writing a function (optional)

Rather than write a single equation for interpolation, it is far better to write it as a function. In this case it can look like the following (again, this is adapted from the T.G. Cleveland reference):

```
#The interpolation function
interp_value <- function (x,xvector,yvector) {
  xvlength <- length (xvector)
  yvlength <- length (yvector)
  for (i in 1:( xvlength -1) ){
    if( (x >= xvector [i]) & (x <= xvector [i +1]) ){
      result = yvector [i]+( yvector [i+1] - yvector [i]) *(x - xvector [i])/
        (xvector [i+1] - xvector [i])
      return (result)
    }
  }
}
```

The command `dens <- interp_value(44,temps,densities)` then produces:

```
> dens
[1] 990.52
```

What would happen if your temperature and density vectors had different numbers of values in them? What if you put in a value outside the range of temperatures in the xvector? These sorts of cases should be caught by a good script, so a final R script would need to include quite a few *if* statements to anticipate these situations.

Reading in data from a csv file or a xlsx spreadsheet

Rather than type in the table values, usually you'll be able to import a csv (comma delimited) file (used in the example below), or from Excel files using the **readxl** package.

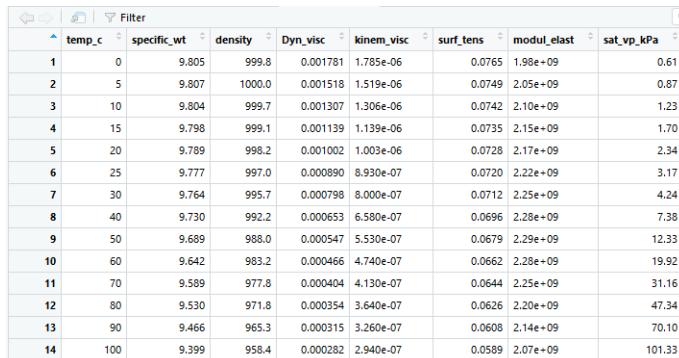
Copy the file **water_properties_SI.csv** into your R working directory. It is a csv file of the water properties table and looks like this:

```
temp_c,specific_wt,density,Dyn_visc,kinem_visc,surf_tens,modul_elast,sat_vp_kPa
0,9.805,999.8,0.001781,0.000001785,0.0765,1980000000,0.61
5,9.807,1000,0.001518,0.000001519,0.0749,2050000000,0.87
10,9.804,999.7,0.001307,0.000001306,0.0742,2100000000,1.23
15,9.798,999.1,0.001139,0.000001139,0.0735,2150000000,1.7
20,9.789,998.2,0.001002,0.000001003,0.0728,2170000000,2.34
25,9.777,997,0.00089,0.000000893,0.072,2220000000,3.17
30,9.764,995.7,0.000798,0.000000798,0.0712,2250000000,4.24
40,9.73,992.2,0.000653,0.000000653,0.0696,2280000000,7.38
50,9.689,988,0.000547,0.000000547,0.0679,2290000000,12.33
60,9.642,983.2,0.000466,0.000000466,0.0662,2280000000,19.92
70,9.589,977.8,0.000404,0.000000404,0.0644,2250000000,31.16
80,9.53,971.8,0.000354,0.000000354,0.0626,2200000000,47.34
90,9.466,965.3,0.000315,0.000000315,0.0608,2140000000,70.1
100,9.399,958.4,0.000282,0.000000282,0.0589,2070000000,101.33
```

This can be read directly into R (in the statement below `head=TRUE` means the first row is a header with the names of each column; `sep=","` tells R commas are used to separate columns):

```
watertable <- read.csv(file="water_properties_SI.csv",head=TRUE,sep=",")
```

Notice it is stored in a different format than a variable – it is a *data frame*, but basically think of it as rows and columns with column names. Click on the name of the new table in the Environment to see what it looks like in R:



	temp_c	specific_wt	density	Dyn_visc	kinem_visc	surf_tens	modul_elast	sat_vp_kPa
1	0	9.805	999.8	0.001781	1.785e-06	0.0765	1.98e+09	0.61
2	5	9.807	1000.0	0.001518	1.519e-06	0.0749	2.05e+09	0.87
3	10	9.804	999.7	0.001307	1.306e-06	0.0742	2.10e+09	1.23
4	15	9.798	999.1	0.001139	1.139e-06	0.0735	2.15e+09	1.70
5	20	9.789	998.2	0.001002	1.003e-06	0.0728	2.17e+09	2.34
6	25	9.777	997.0	0.000890	8.930e-07	0.0720	2.22e+09	3.17
7	30	9.764	995.7	0.000798	8.000e-07	0.0712	2.25e+09	4.24
8	40	9.730	992.2	0.000653	6.580e-07	0.0696	2.28e+09	7.38
9	50	9.689	988.0	0.000547	5.530e-07	0.0679	2.29e+09	12.33
10	60	9.642	983.2	0.000466	4.740e-07	0.0662	2.28e+09	19.92
11	70	9.589	977.8	0.000404	4.130e-07	0.0644	2.25e+09	31.16
12	80	9.530	971.8	0.000354	3.640e-07	0.0626	2.20e+09	47.34
13	90	9.466	965.3	0.000315	3.260e-07	0.0608	2.14e+09	70.10
14	100	9.399	958.4	0.000282	2.940e-07	0.0589	2.07e+09	101.33

You can access each column by appending a \$ and the column name to the table name: `watertable$density` returns the densities. Likewise you can pick out one value by appending an index value like `[3]`. One advantage of data frames is that you can choose a value from one column using a condition of another. To get the density associated with a temperature of 40°C: `watertable[watertable$temp_c == 40,]$density`

You can play around with different formulations of these commands to see how you could rewrite the interpolation function to work with any water property, not just density.

Of course, we're not the first to try interpolating values, which means there are packages to do this. The base R installation includes the *approx* function (which returns two values: appending \$x returns the input lookup value 44.0, appending \$y gives the interpolated density value 990.52):

```
approx(watertable$temp_c, watertable$density, 44.0)$y
```

```
[1] 990.52
```

What if you wanted to look up 10 values? You can do them all at once, skipping the *for* loop.

```
ten_temps <- c(5,10,15,20,25,30,35,40,45,50)
#or more simply:
ten_temps <- seq(from=5, to=50, by=5)
ten_densities <- approx(watertable$temp_c, watertable$density, ten_temps)$y
which produces:
```

```
> ten_densities
```

```
[1] 1000.00 999.70 999.10 998.20 997.00 995.70 993.95 992.20 990.10 988.00
```

Creating new values based on other variables – using a loop or array math

Here we'll start with reading in a file of the standard atmosphere properties, similarly to what was done earlier (ensure the file is in the working directory before executing this command):

```
stdatmosphere <- read.csv(file="standard_atmosphere_SI.csv", head=TRUE, sep=",")
```

The elevations (or altitude) can be placed into a new array:

```
elevs <- stdatmosphere$altitude_m
```


While pressure is already in the standard atmosphere table, as an example we will calculate new estimates of the air pressure using the approximation equation (Allen et al., *FAO Irrigation and Drainage Paper*. No. 56, 1998) where P is in kPa, Z is in meters:

$$P = 101.3 \left(\frac{293 - 0.0065Z}{293} \right)^{5.26}$$

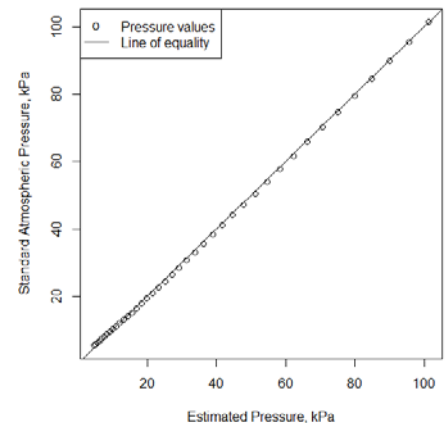
To create a new estimated pressure for each elevation, you could build a *for* loop and run it for each elevation to estimate the pressure. R is smart enough to accept a list of values as input to an equation and will produce a list of results from it:

```
pressure_estimated <- 101.3*((293.0 - 0.0065*elevs)/293.0)^5.26
```

The new *pressure_estimated* variable has the same number of elements as *elevs* (and every other column in *stdatmosphere* data frame, so many plots could be made).

One example is shown here, plotted using the techniques from earlier, with $x=pressure_estimated$ and $y=stdatmosphere\$pressure_kPa$.

The line of equality, on which all points would fall if the approximation were perfect, is drawn on the plot by adding `abline(a=0, b=1)` after the plot command).



The legend in the plot was added by the intuitive command `legend`, similar to this:

```
legend('topleft', c('Pressure values', 'Line of Equality'), lty=c(NA, 1), pch=c(1, NA), bg='white')
```

This was tricky because of the mix of points and lines for symbols, where the line types were specified with a list where the first value was *NA* (meaning *not available*), and likewise since the second legend symbol is only a line, the second symbol was specified as *NA*. You should try to re-create this plot.

Quitting an RStudio session

When you quit from RStudio, you have the option to save your workspace (by saving an image to a file called *.Rdata* in your working directory), which can be restored next time you open RStudio. It should also prompt you to save your R scripts, which you should do.

References

- Cleveland, T.G., 'Fluid Mechanics Computations in R: A Toolkit to Accompany CE 3305', Texas Tech University (2018).
- Lovelace, Robin, and James Cheshire. 'Introduction to Visualising Spatial Data in R'. *Comprehensive R Archive Network*, no. 03 (2014).

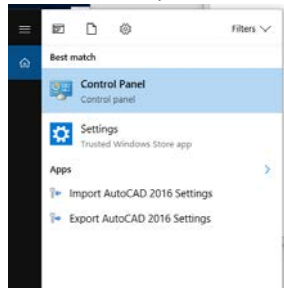
Appendix 1 – Installing Packages on Engineering Computers

To test whether a package is installed, try to load it using *library*. For example, to test if the package *readxl* exists, **open RStudio** and in the pane that has the command console, at the “>” prompt type `library(readxl)` and hit Enter. If there is no output from R, this is good news: it means that the library has already been installed on your computer. *Once a package is installed on a computer (or as a user in the engineering lab) it does not need to be installed again.*

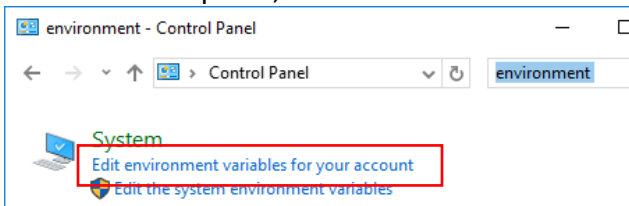
If there is error message, it needs to be installed using *install.packages*. For example on a personal computer (but not the networked ECC computers) you would type `install.packages("readxl")`, and then the `library(readxl)` command would work and the functions in the package would be available. The package will download from CRAN (the Comprehensive R Archive Network); if you are prompted to select a ‘mirror’, select one that is close to your home. **To do this on a networked system like in the lab, you first need to set up a few things, as follows.**

On networked drives, R will look for certain places where it can download libraries and packages. R can work with drives that start with a letter (such as Z:\) but cannot work with network addresses (like `\\machine\...`). On the engineering lab computers, Z:\ and `\\samba1\username` are equivalent, so you just need to tell Windows and R that these are the same. R looks for certain locations based on something called *environment variables*, so you have to set that.

In Windows, click the search button and look for Control Panel.

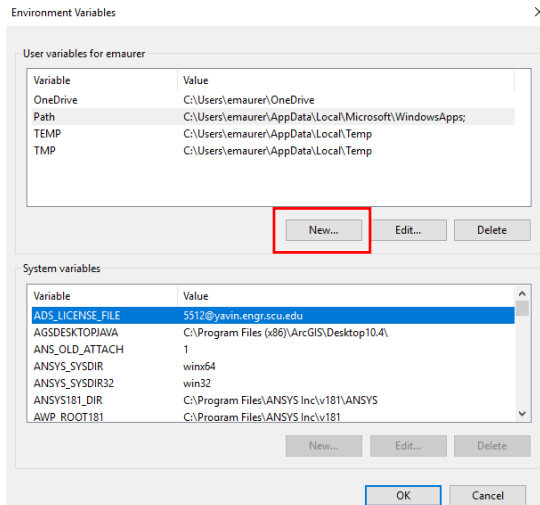


On the control panel, search for environment variables:



Click the “Edit the System Variables for your account.”

In the top window for *User Variables*, click **New**.

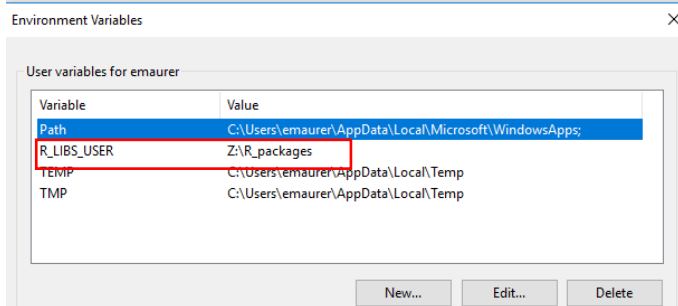


Add the following name and value:

Variable name: R_LIBS_USER

Variable value: Z:\R_packages

Click OK. Now it should appear in your list of user environmental variables.



R creates the directory defined by Value and installs packages there. You may use another path if you prefer – R will just install packages where you define.

Click OK to save the variable and OK again to exit the environment variable window. You will need to **log out and log back in again** to activate this variable.

You can now test to see if you can install a library, and hopefully will see a “success” message, something like this:

```
> install.packages("readxl")
Installing package into 'C:/Users/EdMaurer/documents/R/win-library/3.6'
(as 'lib' is unspecified)
trying URL 'https://cloud.r-project.org/bin/windows/contrib/3.6/readxl_1.3.1.zip'
content type 'application/zip' length 1524653 bytes (1.5 MB)
downloaded 1.5 MB

package 'readxl' successfully unpacked and MD5 sums checked

The downloaded binary packages are in
C:/Users/EdMaurer/AppData/Local/Temp/Rtmpw0D0zm/downloaded_packages
> |
```

You'll see when one package needs another package to work (a 'dependency') it automatically looks for it and installs it if needed.

Appendix 2 – Exporting your script and output (including plots) to a pdf file with knitr

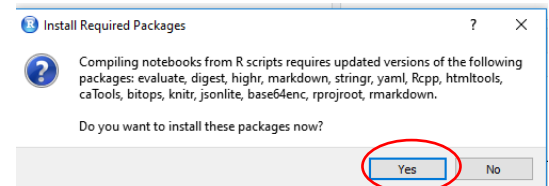
⇒ Before this can be done on ECC computers, you must create a user environment variable for R following the steps in Appendix 1. You only need to do that the first time you use *knitr* (or install any other R package). **If you haven't already, do that first, then return to this.**

To ensure a proper header appears with your name, the lab title, and the date, add something like this to the very top of your script:

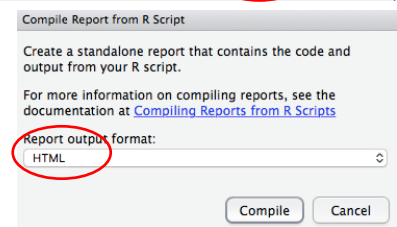
```
#' ---
#' title: "LAB 6 - calculating flow through a cross-section"
#' author: "Ed Maurer"
#' date: "Sept 24, 2018"
#' ---
```

Each of these lines begins with `#'` (with the single quote). That tells knitr it is a special (“markdown”) comment, which can create a nice document. Place lines like this (starting with `#'`) anywhere in a document to add text. Add headers with `#' #` for h1, `#' ##` for h2, etc. A line starting with `#' *` creates a bulleted list (add 4 more spaces per level of indentation). Add hyperlinks with `#' [text to appear](http://link.html)`. More details are [here](#).

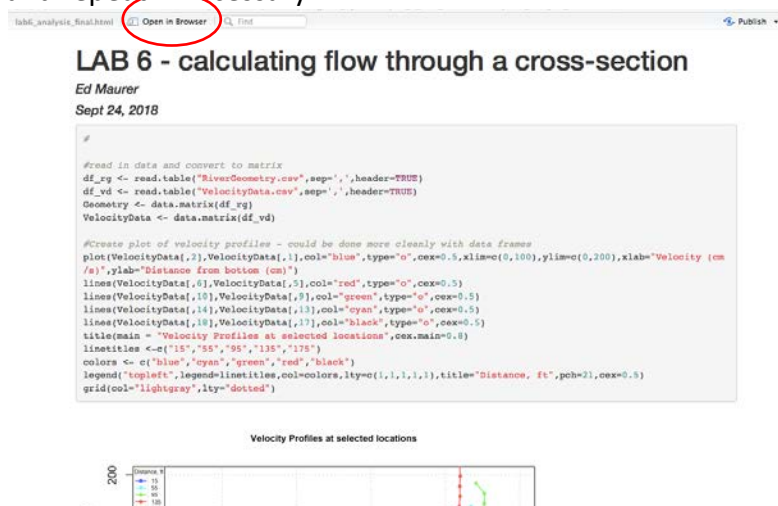
From the RStudio main menu select File → Knit Document. If this is the first time you have used the knitr function, you'll see this notice about installing packages. Say Yes and it will install them.



On the window that appears, select **HTML** output format then **Compile**.



What appears next is an html page that has your code interspersed with the output it created (graphs, tables, etc.). This is a very helpful way to share code, and it also allows anyone to see what your code produced at each step. Ensure it looks clean and professional – return to the script to add more comments or other features and repeat if necessary.




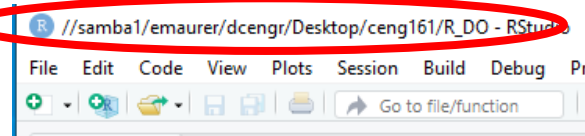
Click “Open in Browser” to use your default browser, from which you can print this to produce a pdf file of your work. If you encounter errors, check Appendix 3 for some possible solutions.

Appendix 3 – Fixing common problems when using R/Rstudio on networked (ECC) computers

3.1) Failure to produce html/pdf output using “knit document” on ECC lab computers (Windows)

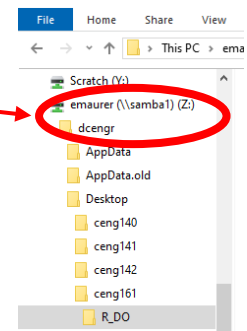
Sometimes on networked computers (like the ECC lab) R/RStudio will fail to create an html version of your script and output when using the “knit document” menu item. If the error message says something about “**pandoc**” there is a simple fix that works in many cases. The issue is often related to saving the R project to your Desktop or My Documents location. That is a fine place to save the files, but how you access them makes a difference.

For example, if you click on a folder on the desktop:  And then navigate to a Rproj file, you can double-click it and it will open in RStudio.

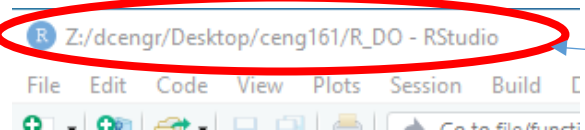


Note that the path to the project starts with “//” indicating a network drive. While **pandoc** can work with these it doesn’t do so easily. If you run “knit document” on this it will likely fail.

A simple fix is to open Windows Explorer and navigate to **Z:\dcengr** and then to your folder with the R project.



Double-clicking on the Rproj file from here opens it in RStudio, as before notice in RStudio the path to the project now starts with a letter (**Z:/**). Now the “knit document” command should work.

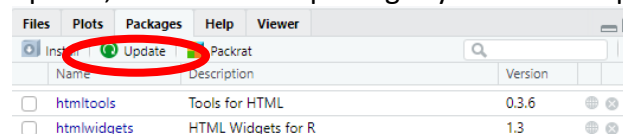


The top of RStudio should now reference a project location that starts with a letter (Z:/).

Another fix is to just work from a USB drive, which will always have a letter assigned to it.

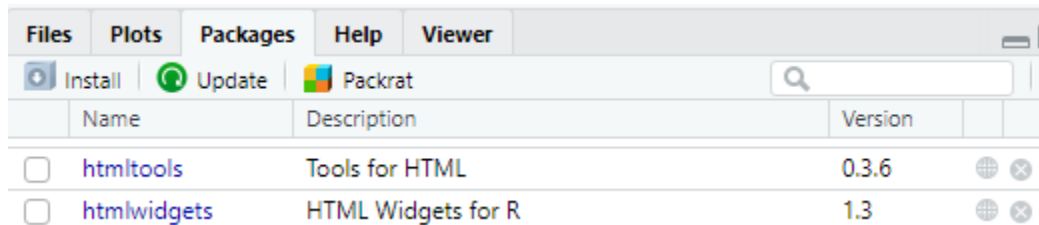
3.2) Packages are incompatible with current R version

As packages evolve or R is updated, packages sometimes need to be updated. If you receive a warning indicating this, in RStudio simply go to the packages tab in the lower right panel, click update, and select the packages you need to update.







3.3) Knit Document produces an error that package needs to be reinstalled

```
> source("../data/Chapter 3/Design/Design/Design/R_00/00_01.R")  
Error: package 'htmltools' was installed by an R version with different internals; it needs to be reinstalled for use with this R version  
> |
```



The screenshot shows the RStudio interface with the 'Packages' tab selected. It displays a table of installed packages. The table has columns for 'Name', 'Description', and 'Version'. Two packages are listed: 'htmltools' (version 0.3.6) and 'htmlwidgets' (version 1.3). Both have checkboxes in the first column, which are currently unchecked. To the right of the version numbers are icons for refreshing and deleting the package.

	Name	Description	Version	
<input type="checkbox"/>	htmltools	Tools for HTML	0.3.6	 
<input type="checkbox"/>	htmlwidgets	HTML Widgets for R	1.3	 

At the console, type commands to remove and re-install the package. This example is for the *htmltools* package, which was the source of the error above.

```
> remove.packages("htmltools")  
Removing package from 'Z:/R_packages'  
(as 'lib' is unspecified)  
> install.packages("htmltools")
```