Tutorial 2: manipulating data classes

In this tutorial we will work with the data you collected in last week's laboratory. This should give you a good example of the difficulties encountered when using outputs from a datalogger. You should already have your own copy of the files but if needed the dataset used here is available from the monitoring webpage. It corresponds to the data collected from the CR3000 data logger at a rate of one sample per minute (Pract2_CR3000_Table1.dat).

➔ As for Tutorial 1, open a new script (file/new script) and start by cleaning the workspace:

```
# R tutorial number 2, 13 Oct 2009
# start with a clean work space
rm(list=ls())
```

➔ We then need to read in the data using the ***read.table*** command. Remember that when invoking this command the path to your data file location has to be specified. This is often a source of problems and you need to be careful about several common mistakes:

- Paths in R are specified using a "/" and not a "\" like in Windows.
- Absolute paths (i.e., starting from the drive location) are always better to use.
- **In the Master's computer room extensions to files** (such as .txt) are put by default depending and the application used and they **are hidden** (see R FAQs page on the web). Therefore if you explicitly specify the extension you might end up with it being written twice (e.g. file.txt.txt) without realizing it. This will of course cause a mismatch between what you ask R to look for and what the file is actually called.

One important difference with Tutorial 1 is that the columns in the data file are delimited by comas (",") rather than just a blank. This needs to be added as an input to the ***read.table*** command, along with the number of lines to skip (4 lines of header here).

```
# read data from the 1 minute file and store into a table
# note that the file has 4 lines of header which we skip
# and that columns are separated by comas (",")
TABLE <-
read.table("C:/MEC/R_tutorial/Tutorial_2/Pract2_CR3000_Table1.dat
",skip=4,sep=",")
```

To check that your path has been found by R you should save your script and source it at this point. You can also copy & paste the last line into the console. If the following message appears in blue you have not specified the path correctly (in this example it should be Tutorial_2 and not Tutorial2)

Error in file(file, "r") : cannot open the connection
In addition: Warning message:
In file(file, "r") :
cannot open file 'C:/MEC/R_tutorial/Tutorial2/Pract2_CR3000_Table1.dat': No such file or directory

If no error message appears you have successfully loaded your data into the array called TABLE. By typing TABLE into the console you should see all your data being printed (100 samples). You will notice that separation was made with regards to the comas and not the blank spaces in the file and therefore the whole timestamp appears in one single column (column 1).

The advantage of the decimal time notation introduced in Tutorial 1 should now appear more obvious when looking at the time stamp provided here: how would you for instance plot the evolution of the temperature as a function of this timestamp? You could use the minutes to start but then you quickly face problems when changing hour.

➔ We will therefore compute the decimal time from the information in this timestamp. First we need to separate the different components and store them in different vectors. A convenient way to do so is to convert the timestamp to a character string[1] and then select only the substrings we actually need. The command to convert to a character string is ***as.character*** and the one to select a substring is ***substr***. The use of ***as.character*** is straight forward. ***substr*** expects three arguments: the name of the original character string, the index of the element from which to start the substring and the one where to stop.

```
# Convert the timestamp (first column) into a character string
TIMESTAMP <- as.character(TABLE[,1])

# Check the result
print(TIMESTAMP[1])

# Now select only the substrings we want
YEAR <- substr(TIMESTAMP,1,4)
MONTH <- substr(TIMESTAMP,6,7)
DAY <- substr(TIMESTAMP,9,10)
HOUR <- substr(TIMESTAMP,12,13)
MINUTE <- substr(TIMESTAMP,15,16)
SECOND <- substr(TIMESTAMP,18,19)
```

You can check the success of the commands by printing any of the vectors created.

➔ All these samples were taken on the same day (DOY 280) so all decimal times will be between 280 and 281. To compute the decimal value we need to convert the character strings back into numbers which is done with the ***as.real*** command[2]. The following line then gives the value of the decimal time:

```
# Compute decimal time from hour and minute vectors
DOY<-280
DECTIME <- DOY+as.real(HOUR)/24+as.real(MINUTE)/(24*60)
```

Note that for a longer dataset you would not be able to fix the DOY to 280 like it is done here. Instead you would need to convert the YEAR, MONTH and DAY information using both the as.Date and format commands. Here is an example of how the same sequence of commands can be done more generically. First we create a DATE character string by pasting the YEAR, MONTH and DAY strings together ("20091007") and then we convert it to the class know by R as a date using the ***as.Date*** command. Note that to avoid any misinterpretation of the date we explicitelly define the format it is to be read in ("%Y%m%d"). The ***format*** command finally enables you to switch from one date format to another, and here we specify that we want the Day of Year notation ("%j"):

[1] Each object in R belongs to a class (character, real, integer, etc ...). Have a look at section 2 of the "An introduction to R" manual on the R webpage for a complete definition (http://www.r-project.org/ and select Manuals).
[2] Note that all commands needed to convert from one class to another start in the same way (as.character, as.real, as.integer, etc...). You can also use the more generic command ***as*** which requires the classes as arguments (see ?as).

```
DATE<-paste(YEAR,MONTH,DAY,sep="")
DOY <- format(as.Date(DATE,"%Y%m%d"),"%j")
DECTIME <- DOY+as.real(HOUR)/24+as.real(MINUTE)/(24*60)
```

➔ We can now store the temperature data (column 5) and plot its evolution as a function of the decimal time. Horizontal lines corresponding to the mean temperature value during the period as well as the mean +/- one standard deviation are also plotted using the ***abline*** command.

```
#Store temperature data
TEMP <- TABLE[,5]

#Plot temperature evolution as a function of the decimal time
plot(DECTIME,TEMP,type="b",xlab="Time (d)",ylab="Temperature
(C)")

#add lines for the mean and the mean +/- sd values
abline(h=mean(TEMP),col="red")
abline(h=mean(TEMP)+sd(TEMP),lty="dashed",col="blue")
abline(h=mean(TEMP)-sd(TEMP),lty="dashed",col="blue")

# add legends
legend(min(DECTIME),max(TEMP),lty="solid",col="red" ,"Mean
value",bty="n")
legend(min(DECTIME),max(TEMP)-0.02,lty="dashed",col="blue","Mean
+/- one standard deviation ",bty="n")
```

➔ Now we want to identify the time at which the minimum temperature occured. The best way to do that in R is via the very powerful command ***which***. When applied to a vector or an array this commands returns the indexes satisfying a given condition. In this case we ask for the index of the element corresponding to the minimum temperature value. Note that the condition is specified using two equal signs. This is the convention when using a TRUE/FALSE type of condition and the omission of the second "=" is a common source of error.

```
# find the index which corresponds to the minimum temp
INDEX_OF_MIN <- which(TEMP==min(TEMP))
```

It is then possible to retrieve other information related to this index, such as the decimal time (or the corresponding minute):

```
# get corresponding decimal time
TIME_OF_MIN <-DECTIME[INDEX_OF_MIN]
```

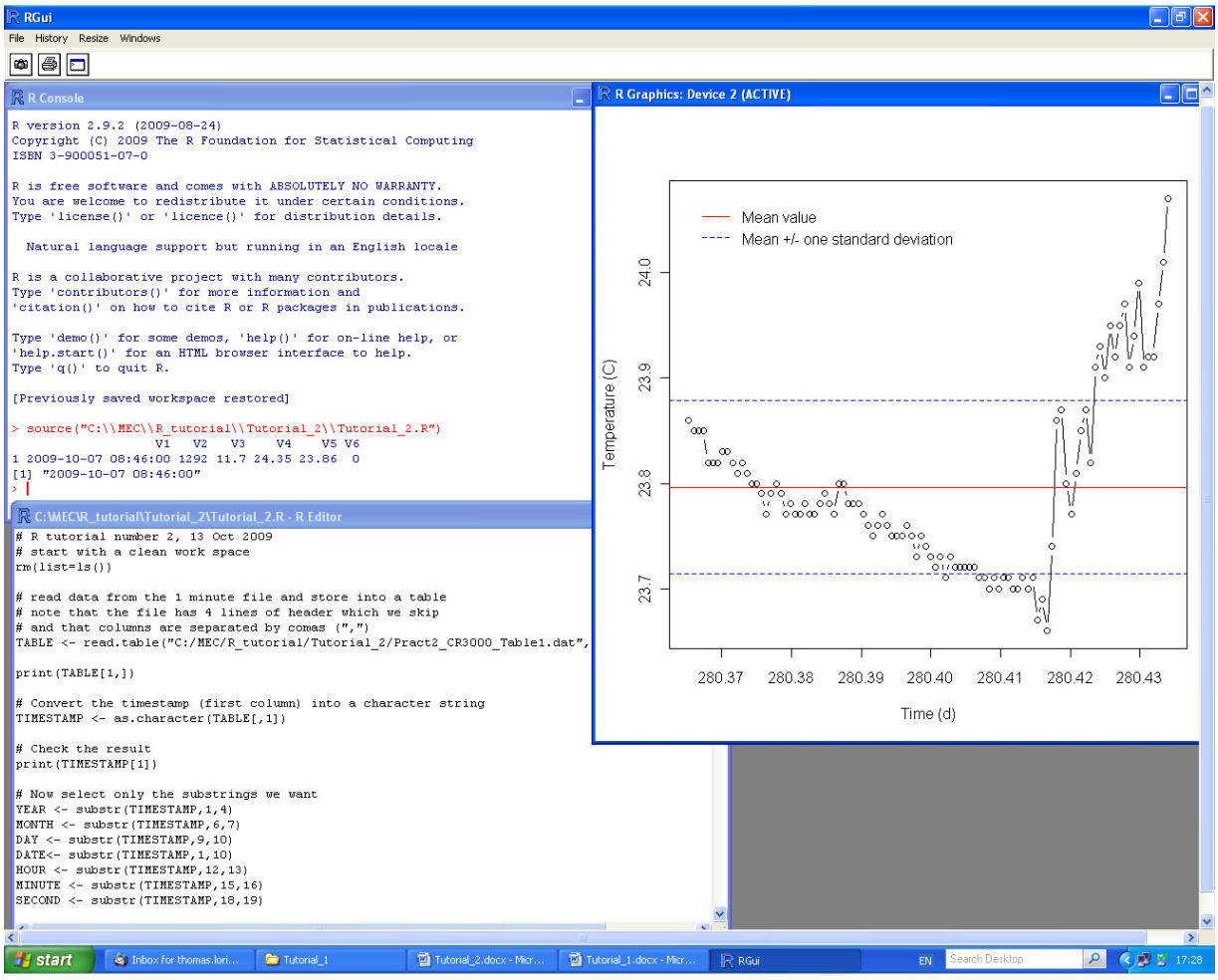
➔ You might want to pass this data to someone else or just keep a track of the decimal time for yourself without having to re-run the script. We will therefore write the decimal time and temperature data to a new text file. This is done with the write command of R which also requires the path to the data file you want to write to. Note that this file does not need to already exist, R will create it if needed. An important argument to this command is the "append=" switch which determines whether to append the data to the file or overwrite existing one.

In this case we start by writing the file header ("%Dectime Temperature"), overwriting any existing line and then we append the actual data. The paste command is needed to generate the lines to write out. This is also how you can specify the column separator ("sep=").

```
# Write out the data to a text file
# First write header info
write("%Dectime
Temperature","C:/MEC/R_tutorial/Tutorial_2/Temp_Dectime.txt",
append=FALSE)
#Then write the actual data lines
write(pasteDECTIME,TEMP,sep="
"),"C:/MEC/R_tutorial/Tutorial_2/Temp_Dectime.txt",append=TRUE)
```

Upon completion you should have a new file in the specified directory. By scanning the file you will see that the dectime data has more digit that we really need. You can force R to round the values before to write them out using the following command instead of the previous one:

```
write(paste(round(DECTIME,4),TEMP,sep="
"),"C:/MEC/R_tutorial/Tutorial_2/Temp_Dectime.txt",append=TRUE)
```



Any suggestions or revisions to this document please email: **thomas.loridan@kcl.ac.uk**