Multiparameter Controllers

Henrik Madsen, Pierre-Julien Trombe, Philip Delff

October 2, 2009

Contents

Chapter 1

Introduction to R and Mysql

While many tasks that used to be performed using relational databases can be easily implemented in R, there are some situations where using the power of R nicely complements the capabilities of a relational database. These latter are used to make working with very large data sets easier. Along with the increasing volume of data that we acquire and store, relational databases are gaining popularity among the scientific community.

There are 2 principal ways to connect with databases and R. The first uses the ODBC (Open DataBase Connectivity) facility available on many computers. The second uses the DBI package of R along with a specialized package for the particular database needed to be accessed (the corresponding package is called RMYSQL in the case of the present project). Common rules state that:

- if there is a specialized package available for the database used, the corresponding DBI-based package may give better performance than the ODBC approach.
- on the other hand, f one uses a database fr which a specialized package is not available, using ODBC may be the only option.

1.1 A brief guide to SQL

Since a single server may hold more than one database each with potentially many tables, and since each table can contain many columns (variables), it may be useful to examine exactly what is available in a database before

starting to work with it. The table below shows some common tasks and the SQL statements to execute them:

Task	SQL Query
Find names of available databases	SHOW DATABASES
Find names of tables in a database	SHOW TABLES IN database_name
Find names of columns in a table	SHOW COLUMNS IN table_name
Find the types of columns in a table	DESCRIBE table_name
Change the default database	USE database_name

When using command-line, each SQL statement must end in a semi-colon but this is not required when using the RMySQL interface.

1.2 Basics of SQL

The first step to understanding SQL is to realize that, unlike R, it is not programming language; operations in SQL are performed using individual queries witout loops or control statements. The most important SQL command is SELECT. Since queries are performed using single statements, the syntax of the SELECT command can be quite long:

```
SELECT column_names or field_names
FROM table_name
WHERE condition
GROUP BY column_names
HAVING condition
ORDER BY column (ASC — DESC)
LIMIT offset, count;
```

Fortunately, most of the clauses in the SELECT statement are optional. In fact, many queries will simply retrieve all of the data in a particular table through the following command:

```
SELECT * FROM table_name;
```

The asterisk (*) means "all the columns in the tabe". Alternatively, a comma-separated list of variables or expressions can be supplied:

```
SELECT var1, var2, ...
FROM table_name;
```

1.3 Modifying Database records

Though there are many a lot of SQL commands to manipulate data stored in relational databases, we give special focus to the following ones: UPDATE, DELETE and DROP. They allow to modify database records. It is worth noticing that these commands take effect on the database as soon as they are issued, so it is a good idea to have a backup of the data in the database before using these commands. But since no backup was possible on the SQL server at IMM (at least in the beginning), these commands were not used extensively in the present project but could turn out very useful for other projects.

To change the values of selected records in a database, the UPDATE command can be used. The format of the UPDATE statement is:

```
UPDATE table_name SET var=value
WHERE condition
LIMIT n;
```

To change more than one variable's value, the *var=value* specification can be replaced with a comma-separated list of variable/value pairs. The WHERE and LIMIT specifications are optional. If a LIMIT specification is provided, only that many records will be be considered for updating, even if some of the chosen records will not actually be modified.

To completely remove a record, the DELETE statement can be used. The basic syntax is as follows:

```
DELETE FROM table_name
WHERE condition
LIMIT n;
```

Without a WHERE clause, all of the records of the database will be removed, so this statement should be used with caution. If a LIMIT specification is provided, it will be based on observations matching the condition of the WHERE clause, if one is specified.

To completely remove an entire table or database, the DROP statement can be used:

```
DROP TABLE table_name;
```

or

DROP DATABASE database_name;

When using the DROP command, an error will be reported if the table or database to be droppped does not exist. To avoid this, the IF EXISTS can be added to the DROP statement as in:

DROP DATABASE IF EXISTS database_name;

1.4 ODBC

The ODBC (Open DataBase Connectivity) facility allows access to a variety of databases through a common interface. In R, the RODBC package, available from CRAN, is used to access this capability. ODBC was originally developed on Windows, and the widest variety of ODBC connectors will be available on that plateform. However, both Linux and Mac OS X also provide database connectivity through ODBC. If one needs to use a database in R that is not directly supported, RODBC will probably be the best choice, as many database manufacturers provide ODBC connectors for their products. The first step in using RODC is to set up a DSN (Data Source Name). It is a file which is configurated to provide all the necessary information to connect and access the database: server, username, password, and databae. The second step consists in loading the RODBC package of R and creating a connection by simply passing the DSN to the *odbcConnect* function as follows:

```
> library(RODBC)
> con=odbcConnect('dsn_name')
```

Additional keywords defining the connection can be provided in the DSN argument: server,user, password, port and database. For instance:

```
> con=odbcConnect('dsn_name;password=xxxx;user=immpjt')
```

Once the user has got a connection to the ODBC source, the sqlQUery function allows any valid SQL query to be sent to the connection. This will be the case even if SQL is not the native language of the underlying database. To prevent unnecessary resource use, the *odbcClose* function should be passed to close any ODBC connection objects when they are no longer needed.

1.5 Accessing a MySQL Database

One of the most popular databases used with R is MySQL. This freely available database (http://mysql.com) runs on a variety of plateforms and is relatively easy to configure and operate.

The first in accessing the a MySQL database is loading the MySQL package. This package will automatically load the required DBI package, which provides a common interface across different databases. Next, the MySQL driver is loaded vi the dbDriver function, so that the DBI interface will know what type of database it is communicating with:

```
> library(RMySQL)
> drv= dbDriver('MySQL')
```

Now, the specifications of the database connection can be provided through dbConnect function. These include the database name, the database username and password, and the host on which the database is running. Id the database is running on the same machine as the R session, the hostname can be omitted. For example, to access a database called "bldg_th", via a user name of "TI_user" and a password of "secret" on the host "pjtsqlsrv.imm.dtu.dk", the following call to dbConnect could be used:

```
> con=dbConnect(drv,dbname='bldg_th', user='TI_user'
    password='secret', host='pjtsqlsrv.imm.dtu.dk')
```

The calls to *dbDriver* and *dbConnect* need only to be made once for an entire session. One can close an unused DBI connection object to dbDisconnect.

SQL queries make requests for some or all of the variables in one or more database tables. In most cases, a single call to *dbGetQuery* can be used to send a query to the database. For example, suppose that

```
> con=dbConnect(con,'SELECT * FROM table_name')
```

Any valid SQL query can be passed to a database by this method.

Chapter 2

Data - description and preparation

2.1 Data overview

2.2 Detection of errors in data

2.2.1 Outliers

2.2.2 Constant sub-sequences

Data from automatic physical measurements often contain subsequences of constant measurements. These are depending on the origin of the data likely to be caused by faulty measurements and must in some cases be removed. Errors may cause some data loggers to keep reporting the same measurement while in fact they are not working.

Care must however be taken every time such a subsequence is removed. A set point for an indoor climate controller can very well be constant for days, or even never be touched. This could be a bad control strategy but is unlikely reflecting faulty data loggers.

A tool has been developed to get an overview of the appearance of such subsequences and to remove them if wished by the user. Figure 2.1 demonstrates this feature applied on the recorded ambient temperature.

From the plot of the raw data, constant subsequences are easily seen in the beginning and towards the end of the data, and these have been removed

by the designed routine. But moreover, it's seen that the clear outliers have also been removed. This may come as a surprise, but is because there are more than one point, and these outliers are thus constant subsequences too.

2.3 Resampling and smoothing

Once data has been cleaned for outliers and other erroneous behavior, it needs to be transformed into a representation that can be used in a *multivariate* analysis. In such an analysis one measurement should no longer be thought of as one number in one of the more than 1000 given time series but rather as one point of as many dimensions as original time series. Hence the original time series must be synchronized such that they can be merged into a multivariate time series with one equidistant time stamps. As an example, consider for instance measurements of a room temperature and measurements of the ambient temperature forming two separate time series. Now one wants to estimate the correlation between the two variables. If the data loggers have recorded with exactly the same frequency and without offset, this task is simple. But what is more likely is that the two are recorded differently with respect to both frequency and offset, and points in the two time series cannot be directly compared since they are not recorded at the same time.

Some kind of interpolation can then be applied on one or both of the time series. In this work, *kernel estimation* has been applied to re-sample all measurements. Kernel estimation is a weighted average in a "sliding window". The kernel that has been used is the Epanechnikov kernel with a total width of one hour such that measurements that are more than half an hour from the point to be estimated are neglected.

2.4 Electricity consumption

Three different time series of electricity consumption have been compared. Two of them are available from the database, they are Electricity to technical installations (time series 20), electricity demand for common installations provided by Dong (time series 981), and manual readings of the electricity meter (not in database).

The data from Dong and the manual readings were inspected, and no outliers or suspicious behavior was found. The series from Dong has a sample period

of 1 hour, starts the first of August 2007 and ends on August 31, 2008. June 2008 is completely missing,but apart from that no datapoints are missing. The series is represents power consumption of unit kW. The meter readings are approximately monthly and represent the cumulative consumption. The readings are done from the end of June 2007 until the beginning of September 2008.

A plot of the measured electricity consumption (time series 20) is shown in Figure 2.3. As shown, observations are recorded over about six months, and contain three periods of missing values of about 15 days each.

Figure 2.4 only shows data from five days making it easier to understand the diurnal behavior of the series. It is seen that the time series represents the cumulative consumption. But it seems that every day, some weird measurements appear.

The periodic jumps before midnight in the measured data from the database are believed to be errors due to re-calibration. Based on a histogram of the data, a threshold value (50 kWh) was decided to distinguish between difference that can be explained by consumption and larger jumps. The jumps were then removed.

After removing the jumps in this time series, the three different series are compared in Figure 2.5.

The results are 25-50% less than before removing the outliers. But it is still far greater than what is read on meters.

This data can be obtained applying the commands:

```
> fetchTS(20)
> ts_20$vaerdi[which(diff(ts_20$vaerdi)>50)+1] <- NA
```

The first command fetches the raw data from the database. `fetchTS()` has substituted `uploadTS()` because of the confusing name, but `uploadTS()` links to `uploadTS()` and therefore does the same. The second command removes all the sudden jumps of more than 50 kWh. In **R** a missing value is denoted `NA` (Not Assigned).

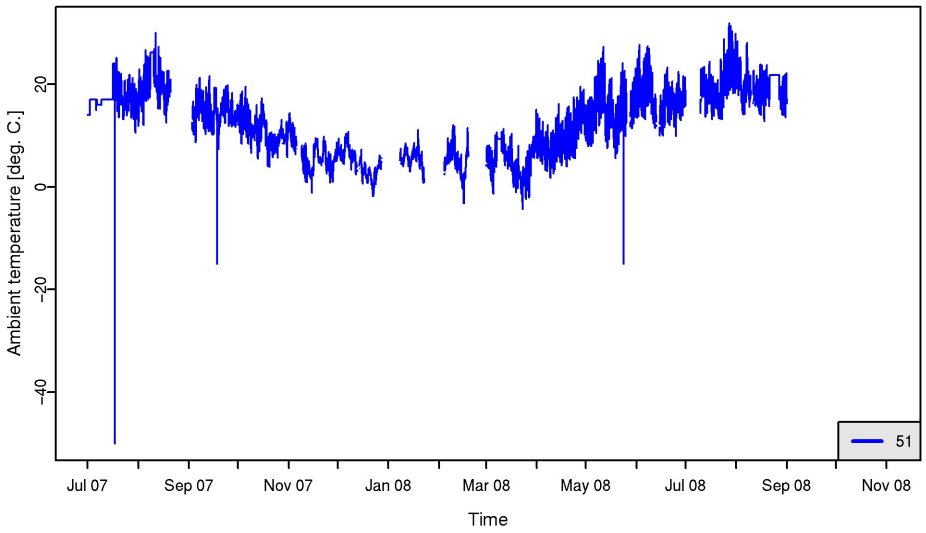
It is seen that the data provided by Dong suggests an electricity consumption of up to 12 times as much as what the meter readings suggest, and up to 5 times as much as what the measurements from the database suggests.

2.5 District Heating

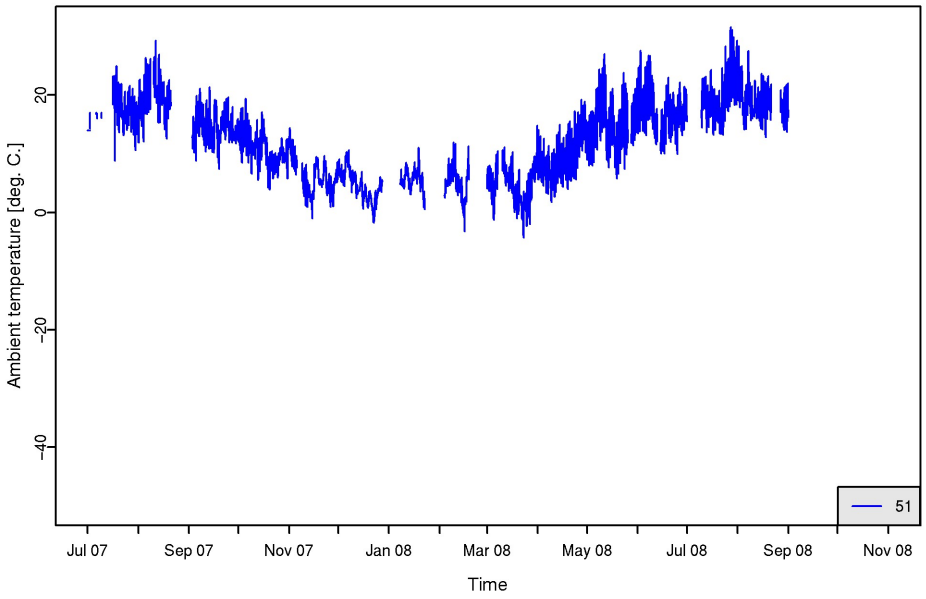
Both a plot of the full time series of district heat demand and an extract is shown in Figure 2.6. This time series is seen not to have the same periodic jumps as the electricity.

The data from the database is now compared with the observations manually read from the meter. These time series are shown in Figure 2.7

These measures are comparable, and they only differ by a factor two in the last period where data is only available for 8 days from the database.



(a) The raw data.



(b) The data after the constant subsequences have been detected and removed.

Figure 2.1 *A time series that have been cleaned for constant subsequences.*

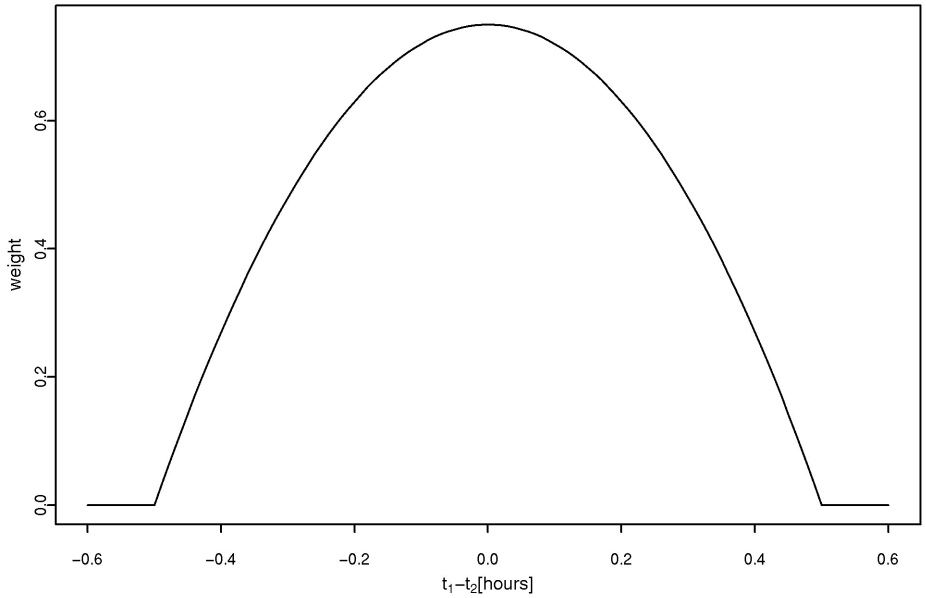


Figure 2.2 *The Epanechnikov kernel of width one hour.*

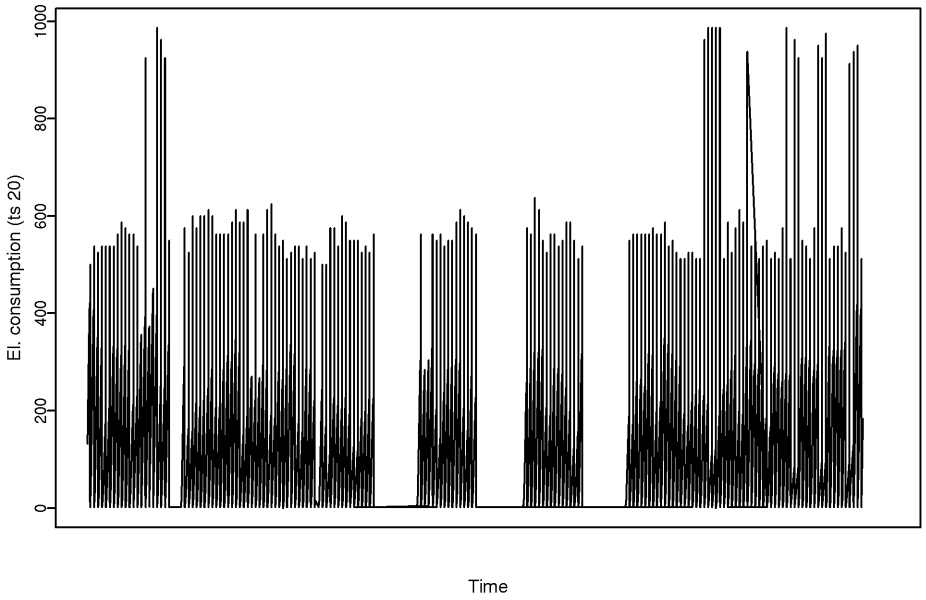


Figure 2.3 *The electricity demand in the database. Time series 20.*

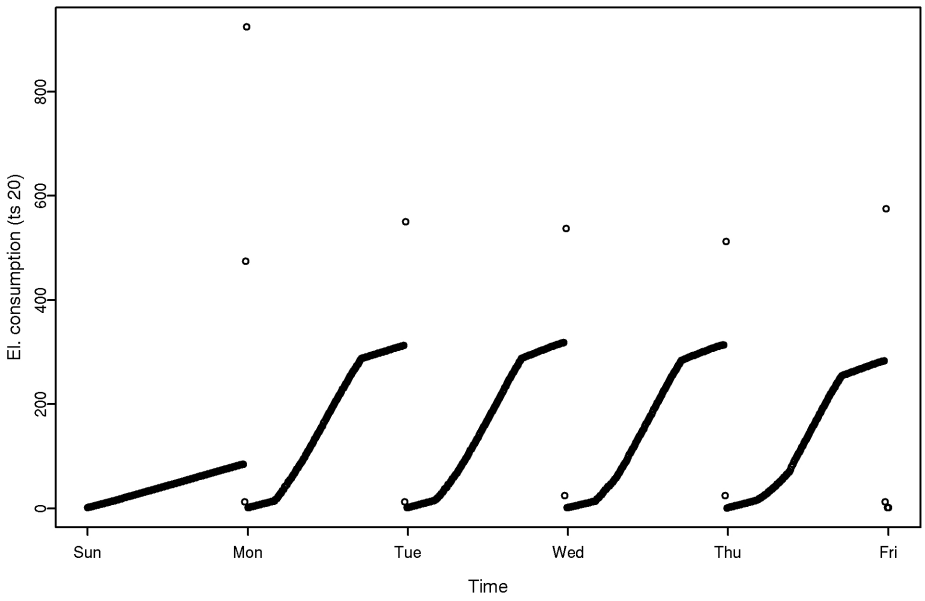
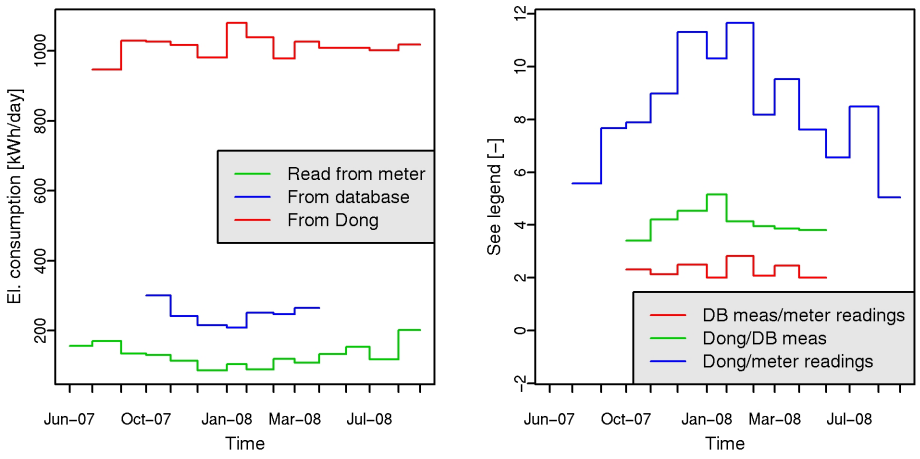


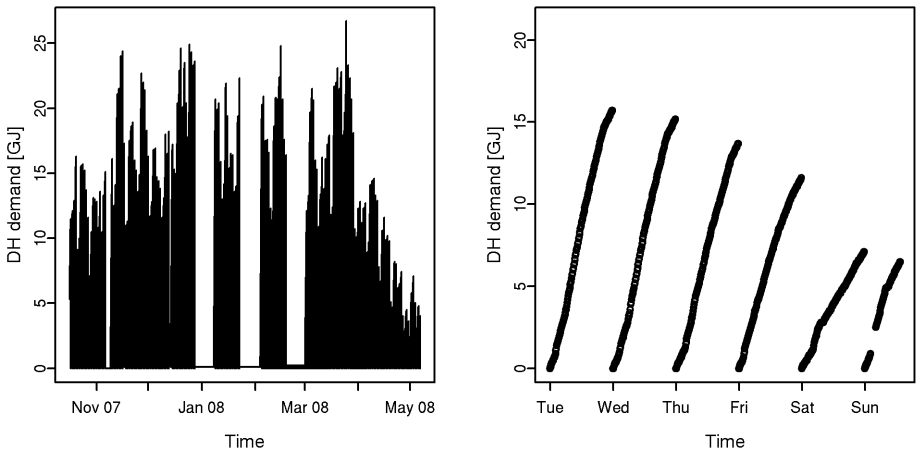
Figure 2.4 *The electricity demand in the database. The period shown is from 2008-04-13 to 2008-04-17.*



(a) Estimates of the daily electricity consumption based on the meter readings and the modified data from the database.

(b) The proportion between the two time series where defined.

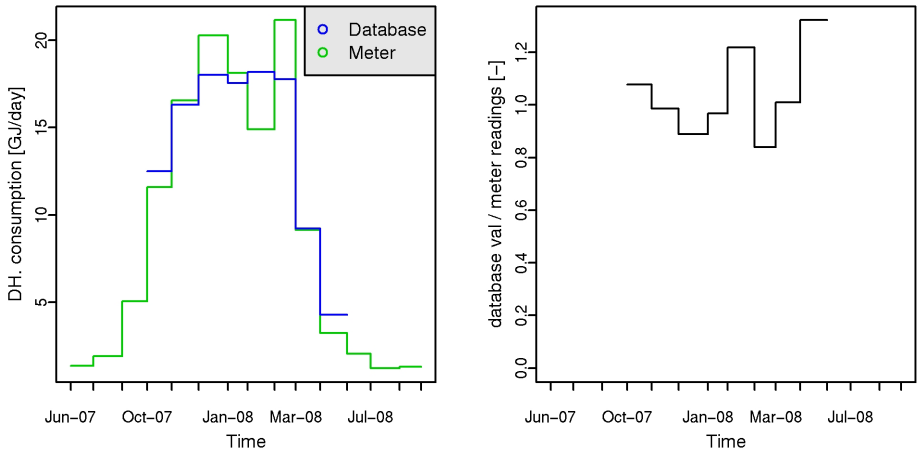
Figure 2.5 *Comparisson of the electricity demands registered in the database and the meter readings.*



(a) The full time series of the district heating demand.

(b) An extract of the district heating demand. 2007-10-22 to 2007-10-28.

Figure 2.6 *District heating demand. Time series 21.*



(a) Estimates of the daily district heat consumption based on the two different time series.

(b) The proportion between the two time series where defined.

Figure 2.7 *Comparison of the district heating demands registered in the database and the meter readings.*

Chapter 3

Multivariate statistics

3.1 Principal component analysis

3.1.1 Heat and cooling demands and electricity consumption

Principal component analysis (PCA) is a statistical method to analyse correlation of multivariate data. The observed variables compose a basis spanning the space of the observation. Principal component analysis is a method to formulate a new basis which components are ordered by their ability to explain the variance observed in the data. This new basis provide information about both which variables of major importance in relation to the variance in the data set and the correlation between the different variables.

Figure 3.1 shows the result of PCA applied on four consumption variables, namely district heat consumption, electricity consumption of technical installations, and the electricity consumption by both cooling systems.

3.1.2 Temperature variation throughout the building

Temperatures in the building are expected to be strongly correlated. First, the mean temperatures in the different rooms are considered in Figure 3.3.

A few mean values a considerably distinct from the rest, and these time series should be investigated. Rooms with low mean temperatures are especially room number 103, 313, 433, 436, and those with particularly high average temperatures are room number 221, 225, and 328. Room 328 is quite extreme, which is because this is a server room.

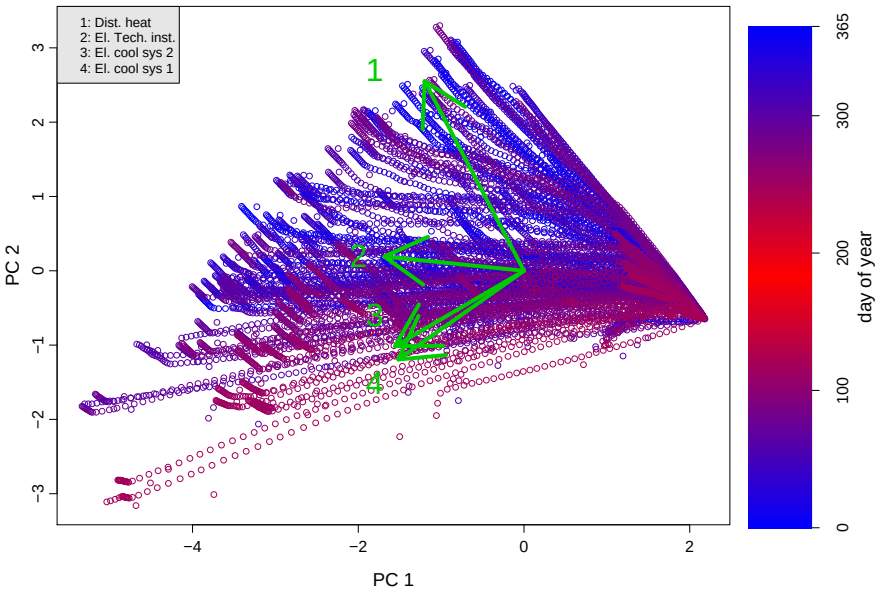


Figure 3.1 *The data transformed with PCA. The green arrows are the directions of the four original variables. The legend lists these ranked by angle from the (1,0) (straight right) and against the clock. The color*

Figure 3.4 shows the first ten principal components' ability to explain the variance of the room temperatures measured by the TAC system.

The percentages shown over the components are the cumulative share of variance described. It is seen that the first principal component explains more than half of the variance, and that the two first components explain 62%. It is therefore very relevant to investigate how the different time series influence on these two components. The observations projected into the plane spanned by the two first components are shown in Figure 3.5. First of all, it is seen that most of the room temperatures are positively correlated in the first component. This is as expected. The first principle component can be interpreted as the mean of most of the room temperatures.

Secondly, some rooms almost do not influence on the first principal component. These are especially rooms 102, 103, and 313 which were all found to have average temperatures below the 95% confidence interval for the average room temperatures. These rooms are not only cooler than the others, they also seem to be controlled differently. Rooms 433 and 436 were also remarked as relatively cool rooms from Figure 3.3. Even though room 433

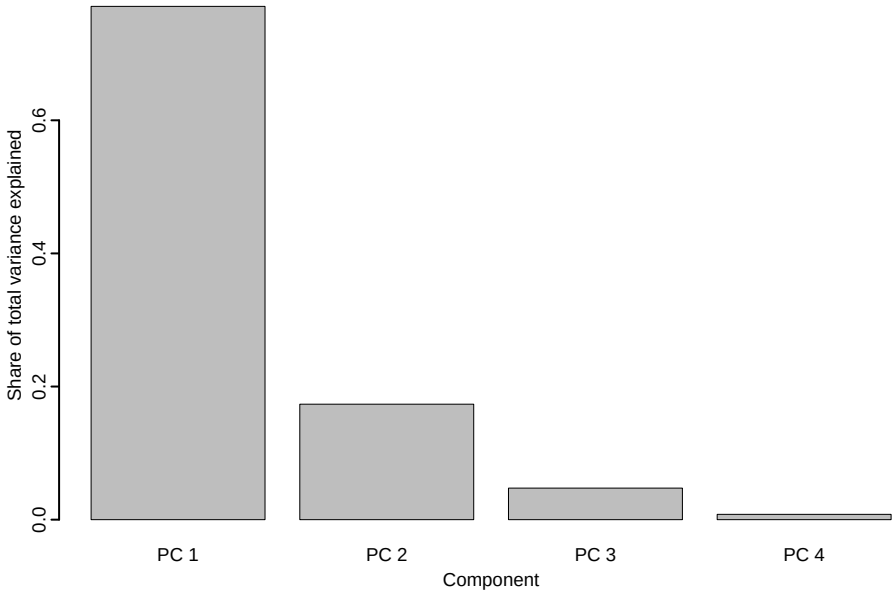


Figure 3.2 *Test figure 2*

does not have a large influence on the first component, it does not seem particularly interesting in relation to the second component either. Finally, it may come as a surprise that the particularly warm rooms - and especially room 328 - do not distinct in the two first principal components.

Figure 3.6 shows a biplot where the observations are projected onto the third and fourth principal components.

3.2 Cluster analysis

3.3 ARMAX models

Autoregressive Moving Average processes with eXogenous inputs (ARMAX) are random processes that are linearly dependent on both history of the processes themselves, the errors commited by the model, and on other processes [**?**]. For short notation, an armax model is denoted by the order of each model part. The order denotes the number of lags (historical sampling periods) the process depend on. E.g. an armax model depending on two

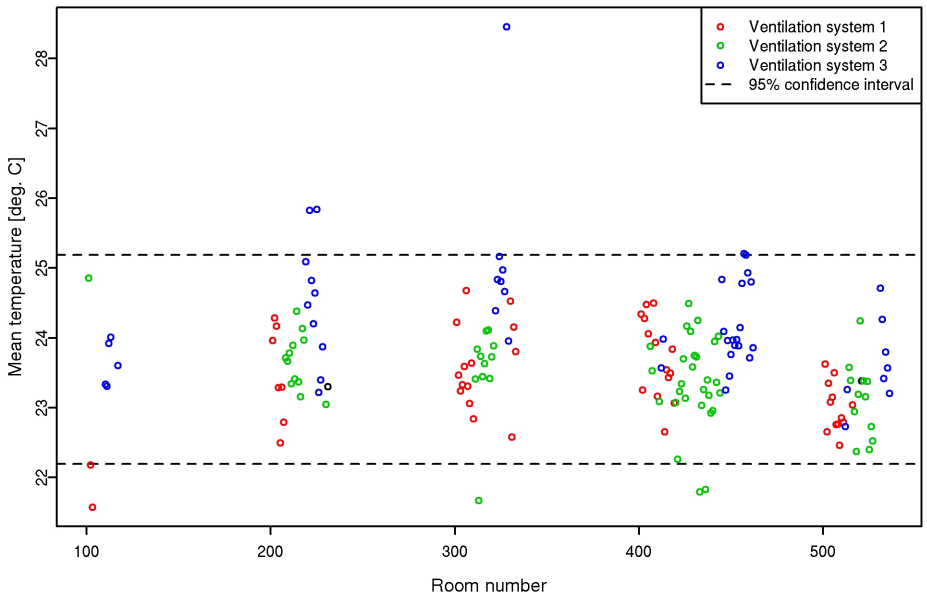


Figure 3.3 *The mean room temperatures with emperical confidence intervals.*

lags of the process' own state, one lag of the prediction error, and three lags of an exogenous process is denoted an ARMAX(2,1,3) model (having six parameters in total).

Since this group of models can have several parameters, care must be taken not to *overfit* data. Overfitting refers to fitting data better than what the model physically explains. By estimateing the parameters in such models *recursively*, this problem is overcome because future observations are never used for predictions from which the residuals are calculated.

Moreover, the estimation can be done *adaptively*. The weight function used in the estimation is exponentially decaying with age of data. Hence, changes in the system behaviour will lead to changes in the parameter estimates. Exponential smoothing adds another parameter to the model, i.e. the *forgetting factor*.

3.3.1 An ARMAX model of the atrium room temperature

The measured room temperature is shown in the upper part of Figure 3.7. Except for four holes of around 15 days each, 10 months of measurements

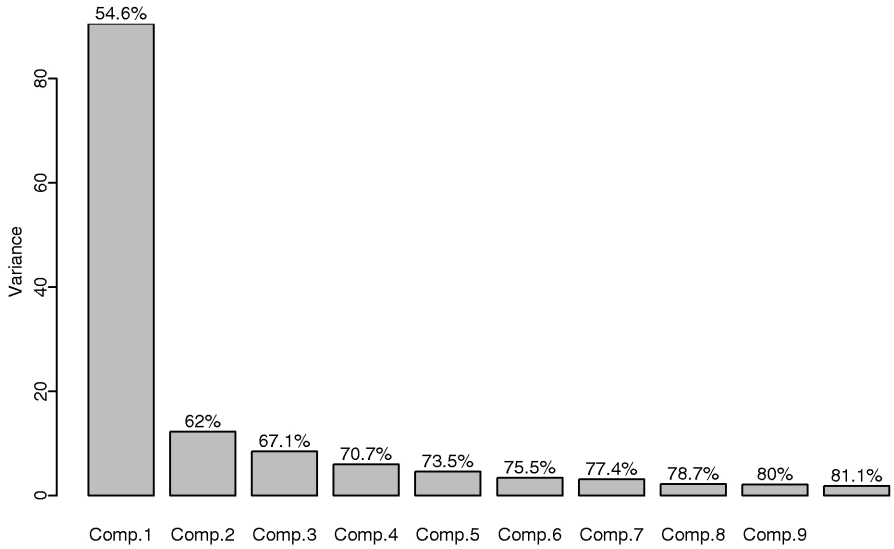


Figure 3.4 *The variance of the room temperatures explained by the 10 first principal components.*

are available. All room temperatures are between 20 and 25 degrees, and they seem to be a little higher at summer than winter. This is as expected.

In the lower plot, the position of the heating valve, the calculated and the measured supply temperature, and the ambient temperature are shown. The position of the heating valve should be interpreted as how open the valve is, from 0 to 100%. At a first look at the heating valve position, the behavior seems to be very different in the period October 2007 to mid-April 2008 from the rest of the considered period. In this sub-period, the variance of the valve position is much smaller, the position only varying a few percents. In the rest of the considered period it the position changes rapidly between 0 and approximately 50% or between approximately 50% and 1. Looking at the temperature graphs, this change of behavior could be a result of some temperature- dependent control. This in itself may be inappropriate since the variance of the room temperature also seem to be larger when the position of the heating valve fluctuates more.

Different models have been tried, and the best performing one was an ARMAX(2,1,2) with the heat valve position as the only exogenous process. The upper plot in Figure 3.8 shows the residuals of this model applied on the

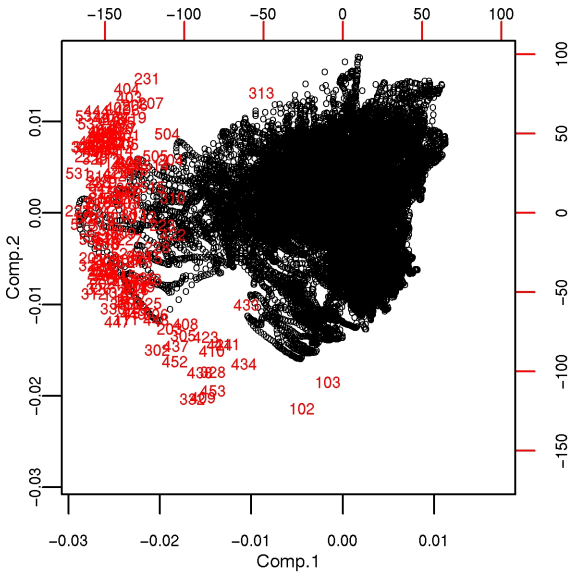


Figure 3.5 *A biplot of the two first principal components. The black points are the room temperature series projected into the two first principal components. The red room numbers represent the different time series average directions and lengths in the plane spanned by the two first principal components.*

room temperature in the atrium. Moreover, an adaptively estimated normal confidence interval is plotted, and red marks are shown in the bottom when the residuals exceed the confidence interval [**?**]. It may be surprising that apparently many more than 0.005% of the observations exceed the confidence interval. The confidence interval is based on the assumption of the errors following the normal distribution which is a bad assumption. Especially because of rapidly changing effects, i.e. heating or ventilation changes, or maybe even errors in data.

From this, one can identify where the system rapidly changes behaviour causing residual outliers. In the lower plot in Figure 3.8 the trace of the parameter estimates are shown. One must have in mind that these parameter estimates are multiplied with explanatory variables of different magnitudes. The size of the estimate can therefore not be interpreted as their influence on the predictions.

Next step from this example is to investigate the control of the heating

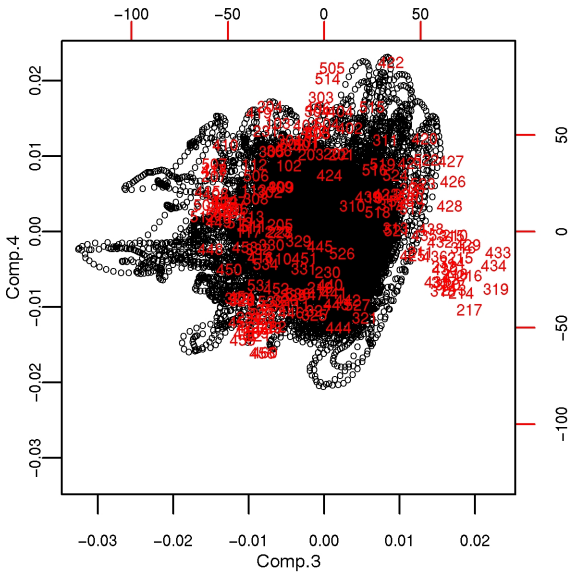


Figure 3.6 *A biplot of the third and the fourth principal components. The black points are the room temperature series projected into these two components. The red room numbers represent the different time series average directions and lengths in the plane spanned by the conidered components.*

valve position to see if a more appropriate control strategy could be used. Moreover, other control variables should be investigated where the position of the heating valve does not seem to explain the outliers.

3.4 Problems and notes

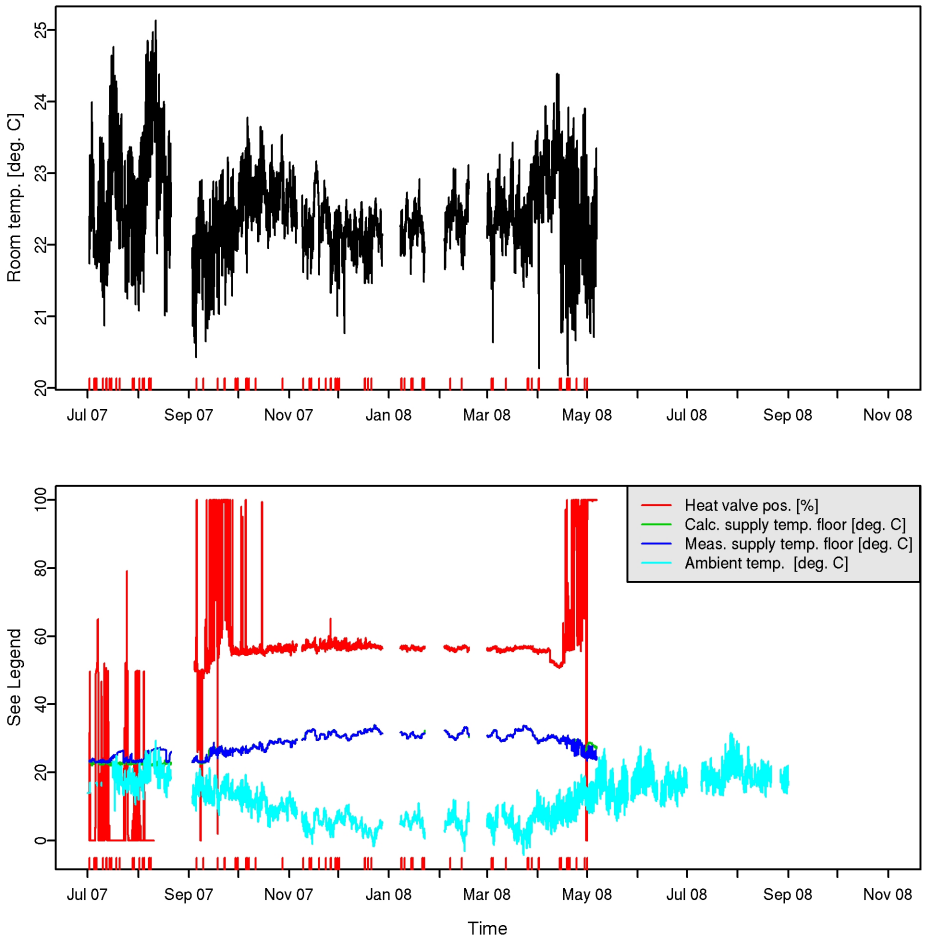


Figure 3.7 *First, the room temperature which is modelled, then some possible explanatory variables.*

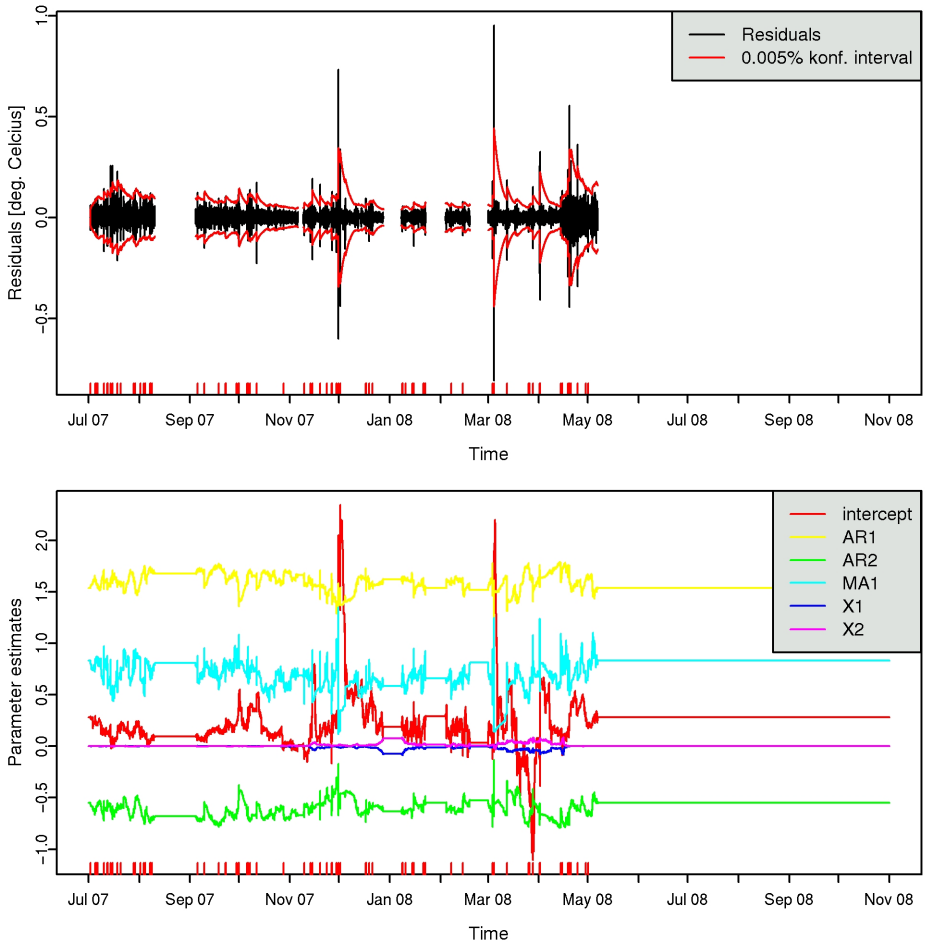


Figure 3.8 *Top: Residuals for the ARMAX model and a prediction interval. The observations exceeding this interval are marked with red at the bottom axis. In the bottom plot, the parameter estimates are tracked. When large prediction errors are committed, the parameter estimates change more.*

Chapter 4

Conclusions

Appendix A

A tutorial on R and tools to visualize data

This tutorial provides a brief introduction to R (www.cran.r-project.org) and some R functions developed to visualize data and export graphics in the framework of the "Multiparameter Controllers" project.

Since the programs are running in connection with a copy of the database hosted on a server at IMM-DTU, one first needs to log on to the IMM server via the ThinLinc client which is used to virtualize desktops. The application file to install it can be freely downloaded at www.cendio.com.

A.1 Log on to the IMM server

Once the ThinLinc Client has been installed and started (the startup interface is seen in Figure A.1), the user must enter:

- the server name: thinlinc.imm.dtu.dk
- a name/login
- a password

It takes a few seconds to log on to the IMM-DTU server. If everything works correctly, a window virtualizing a UNIX desktop environment is opened (Figure A.2).

Figure A.1 *ThinLinc client login.*

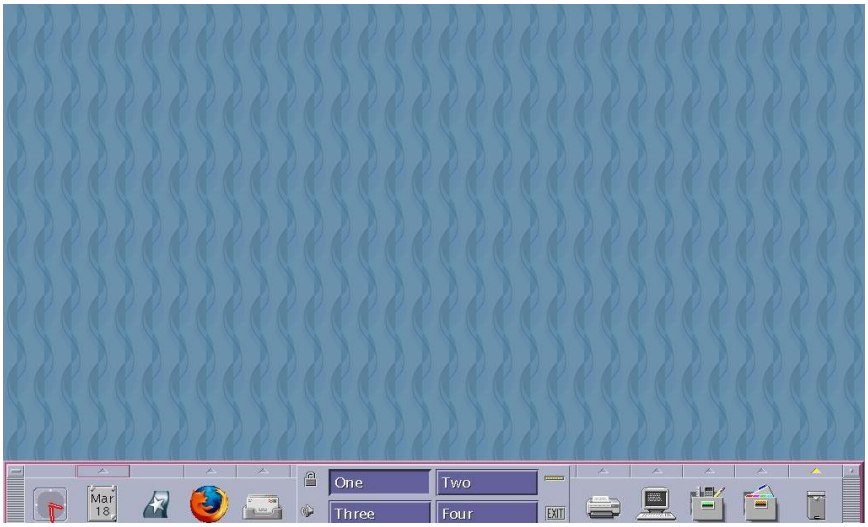


Figure A.2 *Unix desktop environment.*

The user can easily switch from the virtual UNIX desktop to the desktop of his current operating system by pressing the windows key on his keyboard and then minimizing the virtual windows or by turning the "full screen" mode off (Figure A.3).

The user accounts relevant to the project have been equipped with a folder called "MultiParamContr" (Figure A.4). It contains different sub-folders with different purposes. One is called "`common`" and is maintained by Pierre-Julien and Philip. Don't save anything in it since you will probably lose it

Figure A.3 *Minimizing the Unix session.*

when the directory is updated. In "common/doc" you will find the documentation related to the project (this tutorial, descriptions of data, etc).

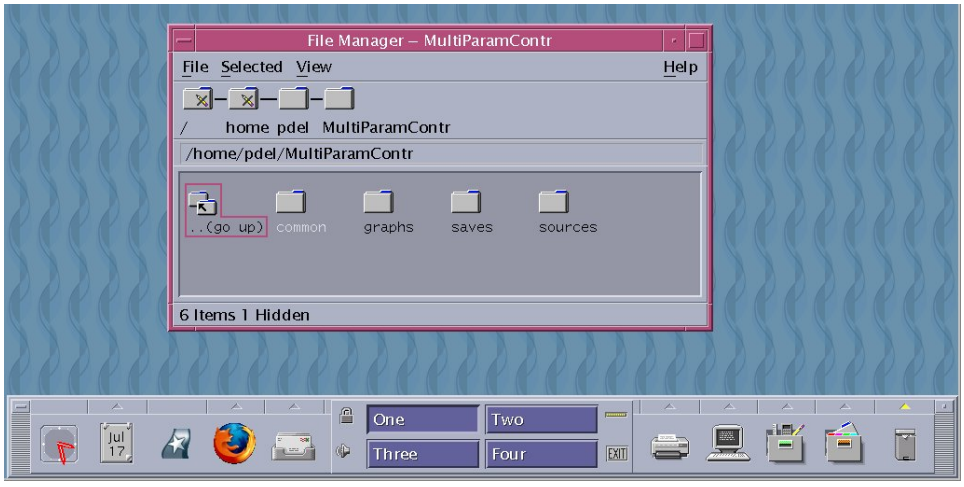


Figure A.4 *The folder used for the Multiparameter Controllers project.*

To reach the file system at IMM from a Windows PC, one can use WinSCP (www.winscp.net). It's an easy-to-use client, similar to many ftp clients. Connect to thinlinc.imm.dtu.dk with the same login as when using Thinlinc.

A.2 Starting R

It is straight forward to start R. First, the user opens a **hms1** terminal window in the list of available terminal windows (Figure A.5).

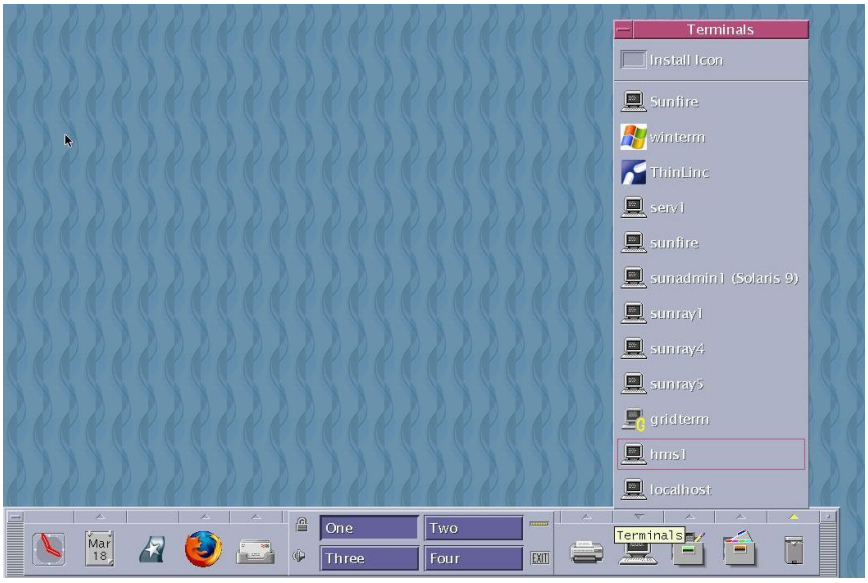


Figure A.5 *Starting a terminal. Notice that it must be on hms1.*

Now, in the hms1 terminal window, type "`Rmultiparam`". This a script made to set up the necessary environment to work with the data and R functions created for that. The command will result in both opening the file the user uses to save his command lines (in a separate window) and R starting (Figure A.6).

A.3 The R interface

R is an advanced collection of software for numeric mathematical modeling and statistics. It can produce graphics of very high quality. To get started with R, one can have a look at one or more of the following:

- http://cran.r-project.org/doc/manuals/R-intro.html The official howto.
- Another R introduction.
- http://cran.r-project.org/doc/contrib/Verzani-SimpleR.pdf Another howto with a lot of nice features.

Technically, R is an expression language which means that the user has to write command lines to call functions and to create object/variables. If a

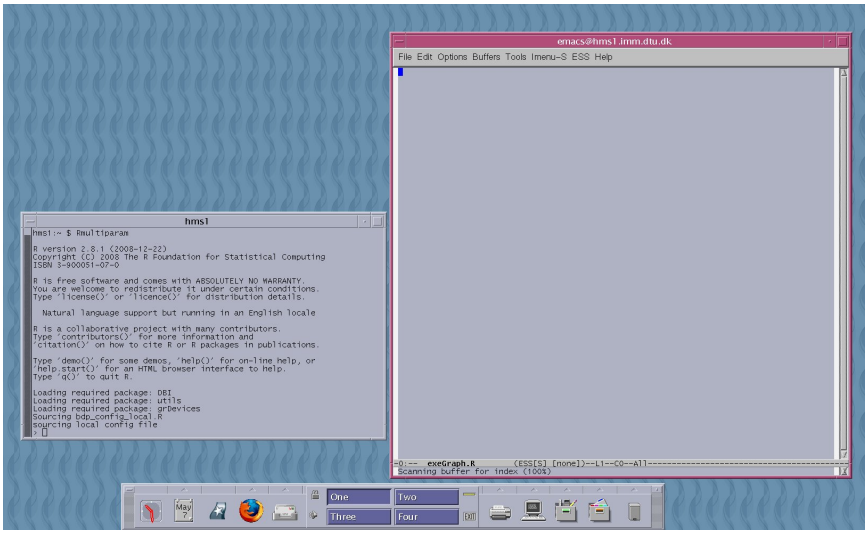


Figure A.6 *R is launched with the "`Rmultiparam`" command taking care of the initialization.*

command is not complete at the end of a line, R will give a different prompt (instead of the usual ">"), by default "+" on second and subsequent lines and continue to read input until the command is syntactically complete.

Remark: R provides a mechanism for recalling and re-executing previous commands. The vertical arrow keys on the keyboard can be used to scroll forward and backward through a command history. Once a command is located in this way, the cursor can be moved within the command using the horizontal arrow keys, and commands can be modified.

To create a vector, one can use the `c()` function. For instance:

```
> c(2,5,6)
[1] 2 5 6
```

There are 2 ways to define a sequence of numbers: for instance, `c(1,2,3,4,5)` can be summarized as `c(1:5)`, or just `1:5`. In the following, this notation will be used.

A.4 Importing data from the database

To extract data from the database and import them into the R environment, a function called `fetchTS()` has been created. It takes the following arguments:

`id.ts` A number (or a vector of numbers) corresponding to a time series identification number in the database.

`sync` (optional) If `sync=TRUE`, kernel smoothing will be applied to the time series. This ensures that different time series will share time stamps. Notice that smoothing is also low-pass filtering. This means that fast dynamics are damped.

The user can import one time series at a time,

```
> fetchTS(991)
Time series 991 has been successfully imported
> fetchTS(434)
Time series 434 has been successfully imported
> fetchTS(73)
Time series 73 has been successfully imported
```

Or many time series at a time using the operator `c()` to create a vector of identification numbers:

```
> fetchTS(c(991,434,73,74,78,79,22,25))
  Time series 991 has been successfully imported
  Time series 434 has been successfully imported
  Time series 73 has been successfully imported
  Time series 74 has been successfully imported
  Time series 78 has been successfully imported
  Time series 79 has been successfully imported
  Time series 22 has been successfully imported
  Time series 25 has been successfully imported
```

Remark: the serial order of the identification numbers does not matter, for each time series imported, a confirmation statement informs the user that the operation has succeeded, there is no restriction on the number of time series (= length of the vector) to be imported but the user must be informed that if too many time series (> 30) have been imported, it can significantly affect/slow down the performance of the software.

A list of the time series already imported into the R environment can be obtained by calling the `listTS()` function which does not take any arguments:

```
> listTS()

 Time series already imported into the R environment:
            22    19780-0100-KØC1-EM1_FORBR_IDAG_LOG_X
            25    19780-0100-KØC2-EM1_FORBR_IDAG_LOG_X
            434    19780-0103-VEN3-IRR324-RT01_LOG_X
            73    19780-0100-VEN2-VI1_P_LOG_X
            74    19780-0100-VEN2-VU1_P_LOG_X
            78    19780-0100-VEN3-MK1_LOG_X
            79    19780-0100-VEN3-MK2_LOG_X
            991    1243031_temp
>
```

Invoking `listTS()` generates a list of time series which can be identified by the combination of the number and the name defined in the database.

It is possible to remove a time series which will not be used any longer in the session by invoking the `removeTS()` function in the same way as the `fetchTS()` function:

```
> removeTS(c(73,434))
> listTS()

 Time series already imported into the R environment:
            22    19780-0100-KØC1-EM1_FORBR_IDAG_LOG_X
            25    19780-0100-KØC2-EM1_FORBR_IDAG_LOG_X
            74    19780-0100-VEN2-VU1_P_LOG_X
            78    19780-0100-VEN3-MK1_LOG_X
            79    19780-0100-VEN3-MK2_LOG_X
            991    1243031_temp
>
>removeTS(-1)
>listTS()
>
```

Remark: By passing -1 as an argument, the `removeTS()` removes any time series previously imported

A.5 Errors

If there is no time series in the database corresponding to the identification number in the arguments, an error message is returned:

```
  > fetchTS(1200)
Error in typeTS(id.maal, id.tac, id.dong, id.ik.log)
   ERROR: identification number 1200  is not correct
>
```

If there is an error in the syntax of the command line (the user forgets to call the vector operator `c()` when importing many time series at a time for instance), an error message will be generated by the program:

```
> fetchTS(991,434,73,74,78,79,22,25)
Error in fetchTS(991,434,73,74,78,79,22,25)
   unused argument(s) (434,73,74,78,79,22,25)
>
```

A.6 Getting information about a time series

Basic statistics are provided by the function, `summaryTS()`. A basic example:

```
> summaryTS(c(80,87,93))

   N.obs N.hours s.p. [min]  Mean Median    sd   <15    <19
      >26    >28
80 74922     311          5  2.50   0.00  8.46  0.955  0.965
    0.024  0.022
87 74922     311          5 25.31   0.00 39.07  0.645  0.667
    0.303  0.292
93 74922     311          5 10.03   8.97  5.91  0.764  0.929
    0.009  0.000
```

`N.obs` is the number of considered observations, `N.hours` is the number of hours over which the observations are spanning. `s.p.` is "sample period", and the unit is minutes. Mean, median and standard deviation then follow before the fractions of the observations that are less than or greater than different values. The default fractions shown are "less than 15", "less than 19", "greater than 26", and "greater than 28". These can be specified with the "lt" (less than) and "gt" (greater than) options.

```
> summaryTS(c(93,80,87),lt=c(21),gt=c(31))

   N.obs N.hours s.p. [min] Mean Median    sd    <21    >31
80 74922     311          5 2.50   0.00  8.46  0.969  0.019
87 74922     311          5 25.31  0.00 39.07  0.678  0.278
93 74922     311          5 10.03  8.97  5.91  0.963  0.000
```

Remark: The output from `summaryTS()` may be too wide to fit the R console and will therefore split into more lines which can be confusing. If needed, the width used by the R console can be changed to for instance 100 characters (default is 80) with the command

```
> options(width=100)
```

The output can also be written to a "comma-seperated values" (csv) file which can be imported to other software. This is done with the option `csv` which must be assigned with a string.

```
> summaryTS(c(93,80,87),lt=c(21),gt=c(31),csv=''csvfile.csv'')
```

A.7 Graphics

Generating graphics is a multifold procedure. The user first initializes graphical settings such as the number of sub-windows and the titles/ X axis labels / Y axis labels for each sub-window. Then, the user defines the time series to be plotted, a time interval and whether he wants to save the graphic as a JPEG file.

A.7.1 Graphical setting initialization

Invoking the ``optgraphTS()'' function requires several arguments to be passed in the right order:

`nb.wind` The number of sub-windows of the plot.

`title` A title for each sub-window (a character string or a vector of character string).

`X.label` X axis label for each sub-window (a character string or a vector of character string).

`Y.label` Y axis label for each sub-window (a character string or a vector of character string).

Below come a few examples (lines starting with # are comments):

```
# 1 window, no title, no label
> opt <- optgraphTS(1,"","","")
# 2 sub-windows, no title, no label
> opt <- optgraphTS (2,"","","")
# 1 window, title & both X and Y labels
> opt <- optgraphTS (1,"April 2008","Time","Temperature")
# 2 sub-windows, same title & labels for both windows
> opt <- optgraphTS (2,"April 2008","Time","Temperature")
# 2 sub-windows, same title & X labels but different Y labels
> opt <- optgraphTS (2,"April 2008","Time",c("Temperature","
    Humidity"))
```

A.7.2 Plotting a time series

Generating graphics requires to call the `graphTS()` function and pass the following arguments:

`id.ts` a time series identification number (or a vector of identification numbers), the user has to make sure that the time series corresponding to that/those number(s) has/have already been imported into the R environment (via the `fetchTS()` function).

`nb.wind` The number of sub-windows of the plot (it has to be the same as in the `optgraphTS()` function).

`ts.wind` A vector of integers which represent the windows in which the time series will be plottet. Eg, if plotting three different time series, and the first one should be plottet alone in the first plot, and the other two together in the second, use `ts.wind=c(1,2,2)`.

`interval` a vector of time interval of the form `c(yyyymmdd,yyyymmdd)`

- c(0,0): default time interval, plot the time series from 1 July 2007 to 31 October 2008.
- c(20080501,20080501): plot the time series over 1 May 2008.
- c(20080501,20080531): plot the time series from 1 to 31 May 2008.

`opt` The name of the object which contains the graphical settings (allocated by calling the optgraphTS function).

`saveGraph` Set equal to `TRUE` (save the graph as a jpeg file) or `FALSE` (do not save any graph).

`grid` If set to `TRUE`, a grid will be drawn in the figure.

`legendpos` By default, the position of the legend is calculated from the plot and tried to be put at the right border of the plot. If this fails, the user can in stead use `legendpos = "string"`, where string is one of `"bottomright"`, `"bottom"`, `"bottomleft"`, `"left"`, `"topleft"`, `"top"`, `"topright"`, `"right"`. The legend will then be put inside the plot but at the border at the position described by the string.

`yaxt` In some cases, the ordinate axis is not well rendered. In such case, try setting this option to "axis".

Some restrictions have been set up:

- The number of windows is limited to 4 (nb_wind < 4).
- The number of time series to be plotted per sub-window is limited to 10.

Below come a few examples:

```
> opt_1 <- optgraphTS(1,"","","")
> graphTS(434,1,1,c(0,0),opt_1,T)
```

```
> opt_2 <- optgraphTS(1,"Room 324 (TAC Measurements)","Year
    2008","Temperature")
> graphTS(434,1,1,c(20080424,20080430),opt_2,T)
```

```
> opt_3 <- optgraphTS(1,"Room 324 (TAC vs. IK Measurements)","
    Year 2008","Temperature")
> graphTS(c(991,434),1,c(1,1),c(20080424,20080430),opt_3,T)
```

```
> opt_4 <- optgraphTS(4,c("Room 324 (TAC vs. IK Measurements)
    ","","","",""), "Year 2008",c("Temperature","","","",""))
> graphTS(c(991,434,73,74,79,78,25,22),4,c(1,1,2,2,3,3,4,4),c
    (20080424,20080428), opt_4,T)
```

```
> opt_5 <- optgraphTS(3,c("VEN3-MV1","VEN3-RV1","VEN3-UT1")
    ,"24-28 April 2008","")
> graphTS(c(80,87,93),3,c(1,2,3),c(20080424,20080428), opt_5,T)
```

Listing A.1 *An example of plotting three time series in three different figures. The range of interest is narrowed down. The resulting figure is seen in Figure A.11*

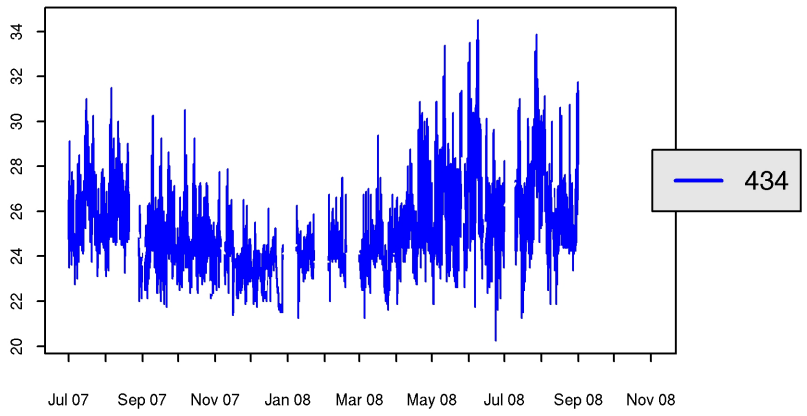


Figure A.7 *Resulting graph from the first example on the use of* `graphTS()`.

The name of the generated graphic file is a combination of the identification number and the time interval. For instance:

Function call	Name of the saved file
optgraphTS(434,1,1,c(0,0),opt_1,S)	fig_434_20070701_20081101.jpeg
optgraphTS(c(434,991),1,c(1,1),c(0,0), opt_1,S)	fig_434_991_20070701_20081101.jpeg
optgraphTS(c(434,991,22),2,c(1,1,2), c(0,0),opt_1,S)	fig_434_991_22_20070701_20081101.jpeg
optgraphTS(c(434,991,22),2,c(1,1,2), c(20080424,20080428),opt_1,S)	fig_434_991_22_20080424_20080428.jpeg

To display graphics files, click on the Home Folder icon on the left hand side of the bin icon and the path: /home/??/MultiParamContr/graphs ("??" stands for the login name of the user, "pdel" in Figure A.4 for instance).

A.8 Estimating distribution functions

You may want to have a more complete view of the distributions of the data in a time series. For this, you can use the `cdfTS()` function. It uses

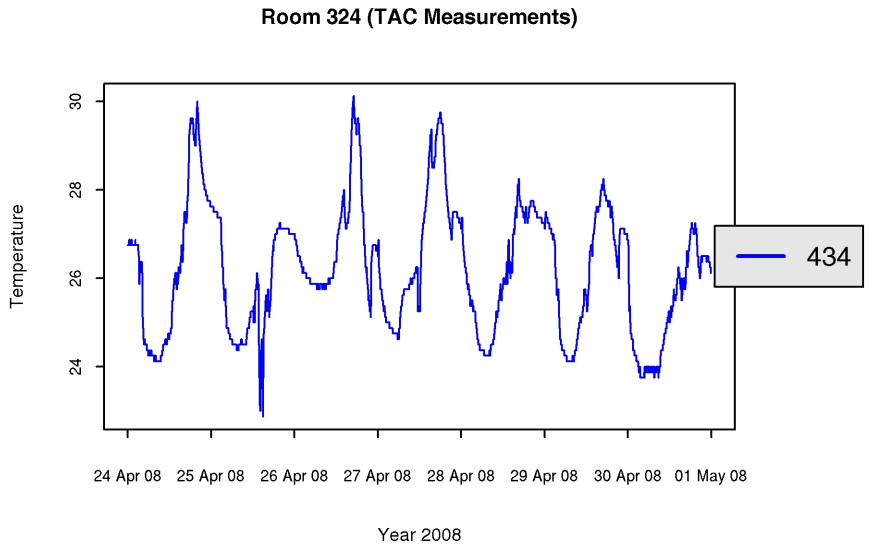


Figure A.8

the "opt" argument like `graphTS()`. So, first the title and axis labels are defined. These graphical parameters must be provided when generating a plot. Here for time series 21:

```
> opt_cdf <- optgraphTS(1,"Estimated CDF for time series 21",
  "Consumption", "P(C<c)")
> fetchTS(21)
> cdfTS(21,opt=opt_cdf)
```

Listing A.2 *Estimating the cumulative distribution function for time series 21 and plotting it. Plot shown in Figure A.13.*

The function takes the following optional arguments:

`savename` If present, the plot will be written to a jpeg file in `/MultiParamContr/graphs/` with the name given, appended with .jpeg if needed.

`start` The lower value of the range of interest. This can be useful if outliers are present in the lower region.

`end` The upper value of the range of interest. This can be useful if outliers are present in the higher region.

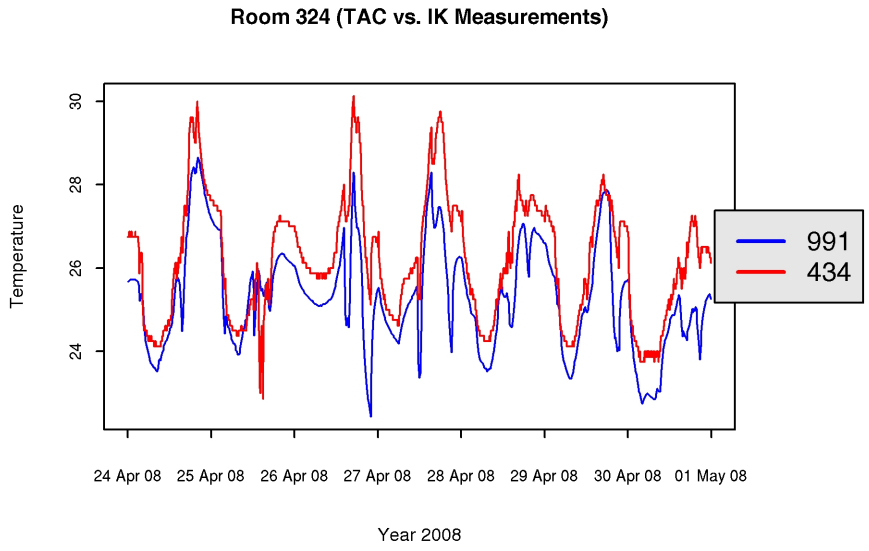


Figure A.9

`step` The distance between the values of the explanatory variable for which the cdf is estimated.

`interval` As for the function `graphTS()`, the time interval that must be considered. c(0,0) is default and means the full period.

`nb.sub` Number of windows (like in `graphTS()`). Default is 1.

`ts.wind` A vector describing in which windows the different time series will be plotted. The default is that all are plotted in the first window.

An example with two time series and some more options used:

```
> opt_cdf2 <- optgraphTS(1,"Estimated CDF for time series 21 and
    991",
  "Consumption", "P(C<c)")
> cdfTS(id.ts=c(21,991),nb.sub=1,ts.wind=c(1,1),start=10,step=1,
   end=25,opt=opt_cdf2, savename="cdf_21_991")
```

Listing A.3 *The CDF's of two different time series are estimated and plotted together. The resulting plot is seen in Figure A.14*

Figure A.10

A.9 Filtering a time series with specific days/hours

`filterTS()` is a function to strip a time series to only include observations from specific time of week and day. Default is Monday to Friday from 8 a.m. to 5 p.m. To obtain this for the time series number 21 and 991, just type:

```
> filterTS(c(21,991))
```

However, two options, days and hours, control the filtering and can be adjusted by the user. Notice that for days, day 1 is Monday, while day 6 is Saturday, and day 0 is Sunday. Regarding hours, the hours given are the hours numbers. For instance hours=16 corresponds to the time period from 4 p.m. to 5 p.m.

To filter on Tuesdays, from 10 a.m. to 11 a.m:

```
> filterTS(c(21,991),days=c(2),hours=c(10))
```

To filter on Sundays and Mondays, from 3 a.m. to 4 a.m:

```
> filterTS(c(21,991),days=c(0,1),hours=c(3))
```

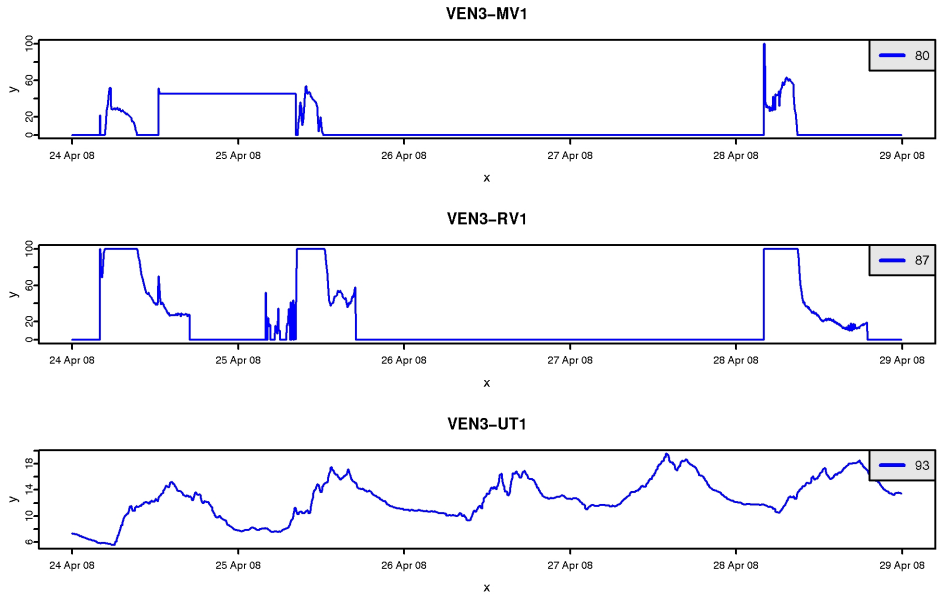


Figure A.11 *Figure produced with the code in Listing A.1.*

Figure A.12

To filter on Mondays, Tuesdays and Thursdays, from 3 a.m. to 4 a.m:

```
> filterTS(c(21,991),days=c(1,2,4),hours=c(3))
```

To filter from Mondays to Fridays, from 3 a.m. to 4 a.m:

```
> filterTS(c(21,991),days=c(1,2,3,4,5),hours=c(3))
```

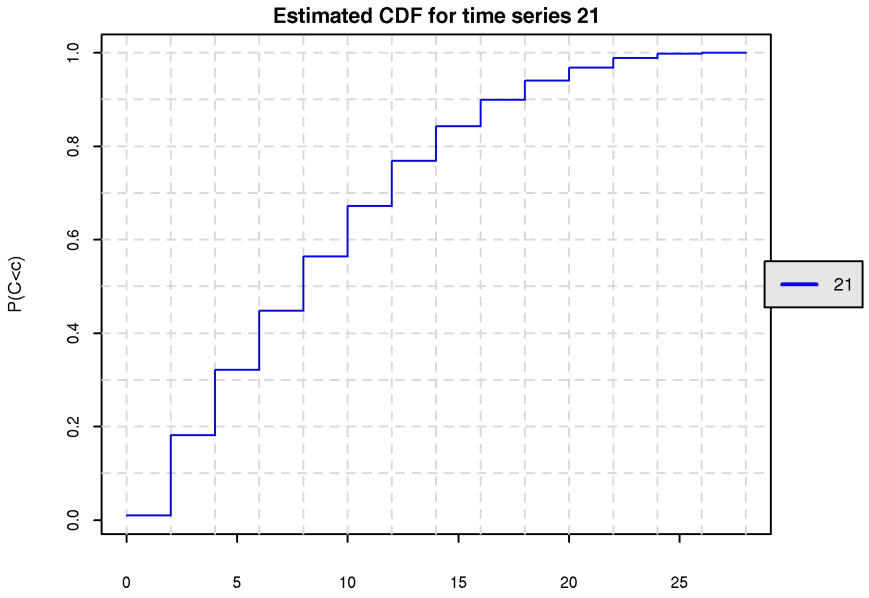


Figure A.13 *The estimated cumulative distribution function for time series 21. Code to produce it in Listing A.2.*

or

```
> filterTS(c(21,991),days=c(1:5),hours=c(3))
```

To consider only a certain period, you can use the option, `interval`. The default is returning data from all of the interval covered by the time series. `interval` uses the same syntax as in `graphTS()`.

To filter from Mondays to Fridays, from 8 a.m. to 12 A.m and from 2 P.m. to 5 P.m:

```
> filterTS(c(21,991),days=c(1:5),hours=c(8:11,14:16))
```

Notice that `filterTS()` modifies the relevant time series directly. The time series then only consists of the mentioned observations. If further work is going to be carried out on the full time series, the user can invoke the `fetchTS()` to restore them.

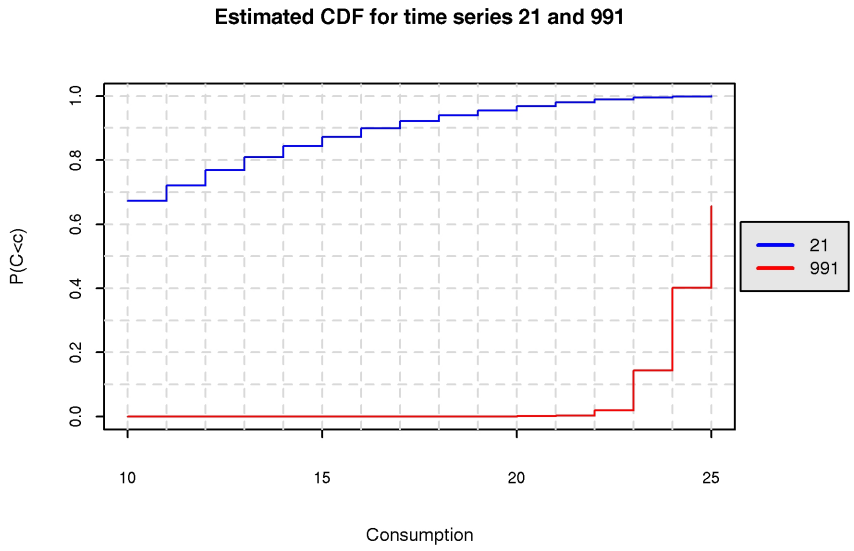


Figure A.14 *Estimate of two different time series. The ranges of the series differ and depend on chosen units. The chosen range in this example does not cover any of the two ranges of the time series. Code generating the plot shown in Listing A.3.*

A.10 Saving the session and exiting R

To save your session such that command history, imported data etc. will automatically be re-read when starting R next time, you have two functions. One is save.image() which you can invoke at any time. When quitting R, you can also just use `q()` and answer `y` to saving your session:

```
> q()
Save workspace image? [y/n/c]: y
```

Remember that this has nothing to do with the text file(s) you are editing with Emacs. Remember also to save this/these from emacs before disconnecting.