

Brief User's Guide: Dseplus extensions to DSE

Paul Gilbert

June 23, 2005

Copyright 1993-2004, Bank of Canada.

The user of this software has the right to use, reproduce and distribute it. The Bank of Canada makes no warranties with respect to the software or its fitness for any particular purpose. The software is distributed by the Bank of Canada solely on an "as is" basis. By using the software, user agrees to accept the entire risk of using this software.

The software documented in this guide is available on the the Comprehensive R Archive Network (CRAN) <<http://cran.r-project.org>> or at <<http://www.bank-banque-canada.ca/pgilbert>>. Please check for new versions.

Parts of this Guide are generated automatically using the R Sweave utilities (see F. Leisch, R News v2/3, Dec. 2002, p 28-31), so the examples should all work, but the formatting of examples may have resulted in some line truncation. For each package, the text and examples in this guide are included in the distributed package subdirectory `inst/doc/*.Stex`. Please check that file if there is any doubt about the example text. Also, screen graphics are different (often with better sized fonts) than those generated in the text. This is a problem that I hope to fix sometime.

I regularly use the code with R on Linux and sometimes on Solaris. There is an extensive set of tests which is run on all R test platforms for packages distributed on CRAN. Please report any errors you find. In the past, the code has also worked with Splus 3.3 on Solaris, but I no longer check this. There are known problems with Splus 5.

Caveat: This software is the by-product of ongoing research. It is not a commercial product. Limited effort is put into maintaining the documentation (but the R tools do automatically check that all functions and their arguments are documented in the help system, and all examples work). This guide may have references to functions which do not yet work and/or have not been distributed, and the documentation may not correspond to the current capabilities of the functions (but please report these problems if you find them). While the software does many standard time-series things, it is really intended for doing some non-standard things. The main difference between DSE and most widely available software is that DSE is designed for working with multivariate time series and for studying estimation techniques and forecasting models.

Constructive suggestions and comments are welcomed. I can be reached at <pgilbert@bank-banque-canada.ca> or by phone at (613) 782-7346.

The Users Guide is divided into sections corresponding to the packages in the dseplus bundles. A copy of the section for each package is also included with the package.

DSE Bundle Contents

1 dse1 Guide

1.1 Defining a TSdata Structure

1.2 ARMA and State-Space TSmodels

1.3 Model Estimation

2 dse2 Guide

2.1 Forecasting

2.2 Evaluation of Forecasting Models

2.3 Evaluating Estimation Methods

2.4 Adding New TSdata Classes

2.5 Adding New TSmodel Classes

3 tframe Guide

3.1 tframe Functions

4 Mini-Reference

DSEplus Bundle Contents (not included in DSE guide)

5 juice Guide

5.1 Juice Functions

6 curve Guide

6.1 Curvature Calculations

7 monitor Guide

7.1 Cookbook for Monitoring Models

8 padi Guide

8.1 PADI Data Retrieval

9 dsepadi Guide

9.1 TS PADI Data Retrieval

10 syskern Guide

10.1 syskern Functions

Other Related Package Contents (not included in DSE guide)

11 setRNG Guide

11.1 setRNG Functions

12 CDNmoney Data

13 tsfa Guide

13.1 tsfa Functions

Preamble

1 Introduction to Dseplus

Please see the DSE Guide for a more general introduction

Examples of the use of several functions are illustrated in the files in the demo subdirectories. (In R see *demo()*)

dse Bundle

2 Juice Functions

In R, the functions in this package are made available with

```
> library("juice")
> ls()
```

As of R-2.1.0 the code from the vignette that generates this guide can be loaded into an editor with *edit(vignette("juice"))*. This uses the default editor, which can be changed using *options()*. Also, it should be possible to view the pdf version of the guide for this package with *print(vignette("juice"))* and the guide for the dse bundle with *print(vignette("dse-guide"))*.

Not yet.

3 curve Functions

In R, the functions in this package are made available with

```
> library("curve")
> ls()
```

As of R-2.1.0 the code from the vignette that generates this guide can be loaded into an editor with *edit(vignette("curve"))*. This uses the default editor, which can be changed using *options()*. Also, it should be possible to view the pdf version of the guide for this package with *print(vignette("curve"))* and the guide for the dse bundle with *print(vignette("dse-guide"))*.

Not yet.

4 syskern Functions

In R, the functions in this package are made available with

```
> library("syskern")
```

As of R-2.1.0 the code from the vignette that generates this guide can be loaded into an editor with `edit(vignette("syskern"))`. This uses the default editor, which can be changed using `options()`. Also, it should be possible to view the pdf version of the guide for this package with `print(vignette("syskern"))` and the guide for the dse bundle with `print(vignette("dse-guide"))`.

Not yet.

5 TS PADI Data Retrieval

In R, the functions in this package are made available with

```
> library("padi")
```

As of R-2.1.0 the code from the vignette that generates this guide can be loaded into an editor with `edit(vignette("padi"))`. This uses the default editor, which can be changed using `options()`. Also, it should be possible to view the pdf version of the guide for this package with `print(vignette("padi"))` and the guide for the dse bundle with `print(vignette("dse-guide"))`.

This section describes utilities for retrieving data from an online database. This has been implemented using the TS PADI interface. The examples use series names which are specific to the Bank of Canada.

Building a database plug will typically require some programming effort. This effort can be reduced by using a standardized interface. Code and a description of a prototype of a standard for a Time Series Protocol for Application - Database Interface (TS PADI) is available at <http://www.bank-banque-canada.ca/pgilbert>. The code includes a working interface to a Fame database.

(This interface is getting to be fairly old and, although it still works, some of the underlying code should probably be replaced with a newer approach to standardized interfaces.)

Data is retrieved with a description which gives an indication of where the data comes from, which series are model inputs and which are model outputs, any transformations which should be applied to the data, and some padding information indicating whether the series should be padded with NAs to the length of the longest available series or truncated to the subset where all data is available for all series. Data is retrieved by using the generic function `freeze()` on the description. When `freeze()` is applied to an object which is already time series data then the data is simply returned. When applied to a data description object the data is retrieved from the data base. Most of the functions in the DSE library use the function `freeze()` on data, so data descriptions can be used interchangeably with data. For model estimation purposes it is usually desirable to retrieve the data and work with a fixed data set, but once a model is established and is routinely used with newly available data then the data description is more convenient.

The following simple example specifies the series *V14182897* from the *ets* server as the single output series.

```
> if (checkPADIServer("ets")) manuf.prod <- getpadi("V14182897",
  server = "ets")
```

Here is a multivariate example:

```
> if (checkPADIServer("ets")) eg.data <- getpadi(c("P100000", "I37026",
  "b1627", "b14013", "b4237", "D767608", "b3400", "M.BCPI",
  "M.JQIND", "M.CUSA0"), server = "ets")
```

6 dsepadi Functions

In R, the functions in this package are made available with

```
> library("dsepadi")
```

and most of the examples also need

```
> require("padi")
```

As of R-2.1.0 the code from the vignette that generates this guide can be loaded into an editor with `edit(vignette("dsepadi"))`. This uses the default editor, which can be changed using `options()`. Also, it should be possible to view the pdf version of the guide for this package with `print(vignette("dsepadi"))` and the guide for the dse bundle with `print(vignette("dse-guide"))`.

This section describes utilities for retrieving data from an online database. This has been implemented using the TS PADI interface. The examples use series names which are specific to the Bank of Canada.

Data is retrieved with a description which gives an indication of where the data comes from, which series are model inputs and which are model outputs, any transformations which should be applied to the data, and some padding information indicating whether the series should be padded with NAs to the length of the longest available series or truncated to the subset where all data is available for all series. Data is retrieved by using the generic function `freeze()` on the description. When `freeze()` is applied to an object which is already time series data then the data is simply returned. When applied to a data description object the data is retrieved from the data base. Most of the functions in the DSE library use the function `freeze()` on data, so data descriptions can be used interchangeably with data. For model estimation purposes it is usually desirable to retrieve the data and work with a fixed data set, but once a model is established and is routinely used with newly available data then the data description is more convenient.

The following simple example specifies the series *I37005* from the *ets* server as the single output series, and gives it a more descriptive name. No data transformations are performed.

```
> eg2.DSE.data.names <- TSPADIdata(output = "I37005", output.names = "manuf.prod.",
  server = "ets")
```

Setting `output.names` is optional. If they are set then they will be used in many printing and plotting routines. The following line then returns the data.

```
> if (require("padi") && checkPADIServer("ets")) eg2.DSE.data <- freeze(eg2.DSE.data.names)
```

The following example specifies one input and one output series.

```
> eg3.DSE.data.names <- TSPADIdata(input = "lfsa455", input.transforms = "percentChange",
  input.names = "manuf.emp.", output = "i37005", output.names = "manuf.prod.",
  output.transforms = "percentChange", pad.start = F, pad.end = T,
  server = "ets")
> if (require("padi") && checkPADIServer("ets")) eg3.DSE.data <- freeze(eg3.DSE.data.names)
```

Here is a multivariate example:

```
> JofF.VAR.data.names <- TSPADIdata(input = "B14017", input.transforms = "diff",
  input.names = "R90", output = c("P100000", "I37026", "b1627",
  "b14013", "b4237", "D767608", "b3400", "M.BCPI", "M.JQIND",
  "M.CUSA0"), output.transforms = c("percentChange", "percentChange",
  "percentChange", "diff", "diff", "percentChange", "percentChange",
  "percentChange", "percentChange", "percentChange"), output.names = c("CPI",
  "GDP", "M1", "RL", "TSE300", "employment", "PFX", "com. price ind.",
  "US ind. prod.", "US CPI"), server = "ets")
> if (require("padi") && checkPADIServer("ets")) JofF.VAR.data <- freeze(JofF.VAR.data.names)
```

This is the example used in Gilbert (1995), except that `freeze()` will retrieve that latest version of the data, and P484549 has been discontinued is replaced with P100000 (CPI, 1992=100).

The variables `pad`, `pad.start`, and `pad.end` control what happens at the beginning and end of multivariate data when all series are not available for the same periods. If `pad.start` is TRUE then NAs are placed at the beginning of series if data is not available, so the multivariate series starts with the first available data. If `pad.start` is FALSE then the beginning is truncated so that the first multivariate data point contains values for all variables. Similarly, `pad.end` works with the last periods of the series. `pad` can be used in place of `pad.start` and `pad.end`.

Most estimation routines require a complete data set for all variables (`pad=F`), but for many purposes it is useful to have all the data. The function `trimNA` takes a complete data set and removes padding at both ends. This is a convenient way to remove NAs from the beginning and end before estimation. The function `tfwindow` can also be used to truncate series to a desired sample period.

An alternate form for specifying the data names can be given using the function `TSPADIdata2`:

```
> alt.JofF.VAR.data.names <- TSPADIdata2(input = list(c("ets",
  "", "B14017", "diff", "R90")), output = list(c("ets", "",
  "P100000", "percentChange", "CPI"), c("ets", "", "I37026",
```

```

"percentChange", "GDP"), c("ets", "", "b1627", "percentChange",
"M1"), c("ets", "", "b14013", "diff", "RL"), c("ets", "",
"b4237", "diff", "TSE300"), c("ets", "", "D767608", "percentChange",
"employment"), c("ets", "", "b3400", "percentChange", "PFX"),
c("ets", "", "M.BCPI", "percentChange", "com. price ind."),
c("ets", "", "M.JQIND", "percentChange", "US ind. prod."),
c("ets", "", "M.CUSA0", "percentChange", "US CPI"))

```

The result is the same but this form may be more convenient in some circumstances. For each series the character strings indicate the server, additional server information, the series identifier, any transformation, and finally a series description. The order of these strings is important. The additional server information may be empty, as above, but cannot be omitted. For some servers it may be used to pass information such as a source database. If no data transformation is to be done then the third string should be empty ("").

A smaller example, also used in Gilbert (1995), is given by:

```

> eg4.DSE.data.names <- TSPADIdata(input = "B14017", input.transforms = "diff",
  input.names = "R90", output = c("P100000", "I37026", "D767608",
    "b3400"), output.transforms = c("percentChange", "percentChange",
    "percentChange", "percentChange"), output.names = c("CPI",
    "GDP", "employment", "PFX"), server = "ets")
> if (require("padi") && checkPADIServer("ets")) eg4.DSE.data <- freeze(eg4.DSE.data.names)

```

7 Cookbook for Monitoring Models

In R, the functions in this package are made available with

```
> library("monitor")
```

As of R-2.1.0 the code from the vignette that generates this guide can be loaded into an editor with `edit(vignette("monitor"))`. This uses the default editor, which can be changed using `options()`. Also, it should be possible to view the pdf version of the guide for this package with `print(vignette("monitor"))` and the guide for the dse bundle with `print(vignette("dse-guide"))`.

This section gives a brief recipe for building short term forecasting models. It is intended to be self-contained although there are references to other sections for additional information.

The term "monitoring" comes from the fact that one is often trying to monitor the current state of the economy based on data from prior periods, since there is typically some lag before statistical agencies release data for the current period. The steps, explained in more detail below, are:

- 1/ specify the data series to use in the model
 - 2/ estimate a model and confirm that it is reasonable
 - 3/ repeat 1 and 2 if other series are to be considered for competing models
- (beware that fishing can be dangerous)

- 4/ run the monitoring program to produce forecasts and optionally
- 5/ set up an automatic program to run the monitoring program and distribute results

This library use the TS PADI interface explained in more detail in an appendix. For example purposes it is assumed that the data can be retrieved from an "economic time series" (ets) server. The examples use names of series which are used internally at the Bank of Canada and are probably not available elsewhere. Start S/R and open a graphics window with

```
> x11()
```

If running remotely it may be necessary to use an argument like "-display YourWorkstation:0.0" to display on your workstation. A few more details on running S/R are given in Section 2 of this guide.

7.1 Step 1- specify the data

The data is specified in an variable which indicates the name of the series, the source, any transformations which should be applied, and possibly some other options. For more details see the section on *TSdata* in the guide for *dse1*. An example of a model which contains two outputs and no inputs is

```
> if (require("dsepadi")) cbps.manuf.data2.ids <- TSPADIdata2(output = list(c("ets",
  "", "i37013", "percentChange", "cbps.prod."), c("ets", "",
  "i37005", "percentChange", "manuf.prod.")), pad.start = FALSE,
  pad.end = TRUE, end = c(1997, 9))
```

With the above, the data will be converted to percent change when it is read from the database. The default behaviour for data retrieval is to trim all series to the same length. The length is such that there are no missing values on the ends. *pad.start* and *pad.end* can be used to modify this behaviour. With *pad.end=TRUE* all series are padded on the end with NAs to give a length which will include the most recent data value from any series. This is preferred for forecasting but the NAs have to be trimmed with *trimNA* for estimation procedures. The data is actually retrieved from the database with

```
> if (require("padi") && checkPADIservers("ets")) cbps.manuf.data2 <- freeze(cbps.manuf.data2)
```

This example and others below will not work without a database server that provides the indicated data. The *if* in the above allows automatic example checking to work (at the Bank of Canada).

The following example specifies one input series and one output series. It uses an alternate constructor (*TSPADIdata* vs. *TSPADIdata2*) which takes arguments in a different format. (The result is the same but different styles sometimes seem more convenient.)

```

> manuf.data.ids <- TSPADIdata(input = "lfsa455", input.transforms = "percentChange",
  input.names = "manuf.emp.", output = "V141828975", output.transforms = "percentChange",
  output.names = "manuf.prod.", server = "ets", pad.start = FALSE,
  pad.end = TRUE)
> if (require("padi") && checkPADIServer("ets")) manuf.data <- freeze(manuf.data.ids)

```

The data can be plotted with

```

> if (require("padi") && checkPADIServer("ets")) tfplot(manuf.data)

```

In this example the plot shows missing data in the middle. In this somewhat unusual case it is necessary to trim the beginning of the data set to remove the portion up to the end of the missing data. This could be done with

```

> if (require("padi") && checkPADIServer("ets")) manuf.data <- tfwindow(manuf.data,
  start = c(1976, 2), warn = FALSE)

```

However, the trimming would have to be repeated each time the data is updated from the database, which is especially inconvenient for automatic procedures described further below. A better way is to set the starting period for retrieved data with

```

> manuf.data.ids <- modify(manuf.data.ids, start = c(1976, 2))

```

then when data is retrieved with

```

> if (require("padi") && checkPADIServer("ets")) manuf.data <- freeze(manuf.data.ids)

```

it will start after the missing data. The start can also be specified with the argument `start` for the function `TSPADIdata`.

A more detailed plot of the last portion of the data can be produced with

```

> if (require("padi") && checkPADIServer("ets")) tfplot(manuf.data,
  start = c(1995, 11))

```

Note the "." after `start` is part of the name of the argument. It is often not necessary since truncated arguments usually match without problem, but is required in the case of `tfplot` so that the argument is not confused with the function `start`. To specify and retrieve data with two input series and one output series

```

> cbps.manuf.data.ids <- TSPADIdata(input = c("lfsa462", "lfsa455"),
  input.transforms = "percentChange", input.names = c("cbps.emp.",
  "manuf.emp"), output = "i37013", output.transforms = "percentChange",
  output.names = "cbps.prod.", start = c(1976, 2), server = "ets",
  db = "", pad.start = FALSE, pad.end = TRUE)
> if (require("padi") && checkPADIServer("ets")) cbps.manuf.data <- freeze(cbps.manuf.data.ids)

```

To specify and retrieve data with one input variable and two output variable

```

> cbps.manuf.data3.ids <- TSPADIdata(input = "lfsa462", input.transforms = "percentChange",
  input.names = "cbps.emp.", output = c("i37013", "i37005"),
  output.transforms = c("percentChange", "percentChange"),
  output.names = c("cbps.prod.", "manuf.prod."), start = c(1976,
    2), server = "ets", db = "", pad.start = FALSE, pad.end = TRUE)
> if (require("padi") && checkPADIServer("ets")) cbps.manuf.data3 <- freeze(cbps.manuf.data3)

```

Setting start is only necessary because of this rather unusual case where there are missing values in the middle of one series

7.2 Step 2 - estimate model

At this point it may be useful to make S/R prompt for a return before each new graph is produced. This is done with

```

> par(ask = TRUE)

```

A model can be estimated with various estimation techniques, some of which are described in Section 6. For example:

```

> if (require("padi") && checkPADIServer("ets")) manuf.model <- bft(trimNA(manuf.data))

```

This uses a "brute force technique" described in Gilbert (1995). It might take some time to run. It uses a default maximum number of lags of 12. The estimation is faster if a smaller number of lags is specified using

```

> if (require("padi") && checkPADIServer("ets")) manuf.model <- bft(trimNA(manuf.data),
  max.lag = 5)

```

By default the bft procedure prints information as it proceeds. This can be stopped using

```

> if (require("padi") && checkPADIServer("ets")) manuf.model <- bft(trimNA(manuf.data),
  verbose = FALSE, max.lag = 5)

```

To display the parameters of the estimated model just type the name of the variable in which it was stored:

```

> if (require("padi") && checkPADIServer("ets")) manuf.model

```

and to plot it:

```

> if (require("padi") && checkPADIServer("ets")) {
  tfplot(manuf.model)
  tfplot(manuf.model, start = c(1990, 1))
  tfplot(manuf.model, start = c(1995, 1))
}

```

Models for the other specified data sets can be estimated in the same way:

```

> if (require("padi") && checkPADIServer("ets")) {
  cbps.manuf.model <- bft(trimNA(cbps.manuf.data), verbose = FALSE)
  tfplot(cbps.manuf.model)
  tfplot(cbps.manuf.model, start = c(1995, 1))
}

```

To forecast with the model using all available data (This example is artificially truncated with `tfwindow` because some of the data has been discontinued.)

```

> if (require("padi") && checkPADIServer("ets")) {
  z <- forecast(TSmodel(manuf.model), tfwindow(manuf.data,
    end = c(1995, 1), warn = FALSE), conditioning.inputs = tfwindow(inputData(manuf.da
    end = c(1996, 12), warn = FALSE))
  tfplot(z, start = c(1995, 1))
}

```

To see the forecast use

```

> if (require("padi") && checkPADIServer("ets")) {
  forecasts(z)[[1]]
  tfwindow(forecasts(z)[[1]], start = c(1996, 3), warn = FALSE)
}

```

Forecasting is discussed in the `dse2` Guide.

To evaluate how well the model does at forecasting, look at the covariance of the forecast error at different horizons with

```

> if (require("padi") && checkPADIServer("ets")) {
  fc <- forecastCov(manuf.model)
  tfplot(fc)
}

```

It is also good to consider how well the forecast does relative to a zero and a trend forecast:

```

> if (require("padi") && checkPADIServer("ets")) {
  fc <- forecastCov(manuf.model, zero = TRUE, trend = TRUE)
  tfplot(fc)
}

```

The above forecast error analysis is done within the sample which was used for estimating the model. An out-of-sample forecast error analysis is typically a better indication of how well the model will really do. This can be done by using `tfwindow` to truncate the data to a subset for estimation and then evaluate the forecast error on the remainder. Another compromise, which is attractive when short data sets are involved, is to do an out-of-sample evaluation of the performance of an estimation procedure, and then hope that the procedure will continue to estimate good models when the whole data set is used.

```

> if (require("padi") && checkPADIServer("ets")) {
  outfc <- outOfSample.forecastCovEstimatorsWRTdata(trimNA(manuf.data),
    estimation.sample = 0.5, estimation.methods = list(bft = list(verbose = FALSE),
      estVARXls = NULL), trend = TRUE, zero = TRUE)
  tfplot(outfc)
}

```

The bft procedure is generally fairly good but it can sometimes be out performed by a simple least squares estimation, especially for univariate models. Its real strength is for multivariate models:

```

> if (require("padi") && checkPADIServer("ets")) {
  outfc <- outOfSample.forecastCovEstimatorsWRTdata(trimNA(cbps.manuf.data3),
    estimation.sample = 0.5, estimation.methods = list(bft = list(verbose = FALSE),
      estVARXls = NULL), trend = TRUE, zero = TRUE)
  tfplot(outfc)
}

```

More details are given in Section 8.

Once a model has been chosen it can be re-used, rather than re-estimating each time there is a new data point. This is done by extracting the model from the object returned by the estimation procedure. This object is a model with data and some estimation information. If you want to use different data then the data needs to be retrieved again using the variable which indicates the source. For example

```

> if (require("padi") && checkPADIServer("ets")) new.data <- freeze(manuf.data.ids)

```

To run the model and get one-step-ahead predictions with the new data use

```

> if (require("padi") && checkPADIServer("ets")) z <- l(TSmodel(manuf.model),
  trimNA(new.data))

```

Or the data retrieval can be done in the same step with

```

> if (require("padi") && checkPADIServer("ets")) {
  z <- l(TSmodel(manuf.model), trimNA(tfwindow(freeze(manuf.data.ids),
    start = c(1976, 2), warn = FALSE)))
  tfplot(z)
  tfplot(z, start = c(1995, 8))
}

```

Forecasts more than one-step-ahead require input series up to the horizon for which the forecast is to be produced. To run the model and get forecasts when more input than output data is available [tfwindow(..., end=c(1996,1)) is used in this example to simulate the situation. The data series have been terminated, so this example needs to be redone.]:

```

> if (require("padi") && checkPADIServer("ets")) {
  z <- forecast(TSmodel(manuf.model), tfwindow(trimNA(new.data),
    end = c(1996, 1), warn = FALSE), conditioning.inputs = trimNA(inputData(new.data)))
  tfplot(z, start = c(1995, 6))
}

```

The effect of this is to trim NAs from input separately from output so that input will not be truncated to the same ending period as output. If you actually want the numbers rather than plots of the data use

```

> if (require("padi") && checkPADIServer("ets")) forecasts(z)[[1]]
or
> if (require("padi") && checkPADIServer("ets")) tfwindow(forecasts(z)[[1]],
  start = c(1996, 2), warn = FALSE)

```

will print values starting in the second period of 1996.

The horizon for a model with no inputs is determined by the argument horizon, which has a default value of 36. For a model which requires input (conditioning) data, the horizon for the forecast is determined by the input data, conditioning.inputs or conditioning.inputs.forecasts. If none of these are supplied then the argument horizon is used to replicate the last period of input data to the indicated horizon.

At the Bank of Canada PADI is an interface to a Fame server. The forecast data can be put into a Fame database with

```

> if (require("padi") && ("fame.server" == PADIServerProcess())) putpadi(forecasts(z)[[1]],
  dbname = "nameofdatabase.db", series = seriesNamesOutput(z))

```

In the above

```

> if (require("padi") && ("fame.server" == PADIServerProcess())) seriesNamesOutput(z)

```

extracts a character vector of the series names.

7.3 Step 3 - reconsider the data and model

The performance of alternative models on a given data set can be compared by looking at the forecast error covariance from forecastCov. Repeat the required parts of steps one and two and choose the model which does best at the horizons of interest. Sometimes the real purpose of a monitoring model is just to forecast one series (the series of primary interest). Other series are included only because they provide additional information for forecasting the series of primary interest. One disadvantage of including additional series is that it increases the number of parameters which must be estimated, and thus reduces the quality of the estimates. At this step you should reconsider what series are included for the model. Choose the model which does best on the series of primary interest (but see also "Juice Functions").

7.4 Step 4 - run the monitoring

During the S session, variables (e.g. models and data) are saved in a subdirectory .Data below the directory where you started S. (In R they are in the file .RData.) The variables will be available the next time S/R is started from the same subdirectory. One danger is that you can overwrite an existing variable just by assigning a new value to the name. Once you have a model to use for forecasting it is a good idea to save it in a separate file so it will not be lost by accident. The model `manuf.model` and the corresponding data identifiers can be saved in the file "manuf.model.definition" with

```
> if (require("padi") && ("fame.server" == PADIserverProcess())) dump(c("manuf.model",
  "manuf.data.ids"), file = "manuf.model.definition")
```

If necessary they can then be retrieved with

```
> if (require("padi") && ("fame.server" == PADIserverProcess())) source("manuf.model.definition")
```

The model can be run to produce a forecast and mail the results to a list of recipients. The function to do this compares the current data to a previous copy of the data in order to determine if an updated forecast should be run. The comparison data is first initialized with

```
> if (require("padi") && checkPADIserver("ets")) manuf.previous.data <- freeze(manuf.data.ids)
```

then in order to make the data look like it has changed

```
> if (require("padi") && checkPADIserver("ets")) outputData(manuf.previous.data)[1,
  1] <- NA
```

and to run the forecast and E-mail the results

```
> if (FALSE) {
  r <- simpleMonitoring(manuf.model, manuf.data.ids, manuf.previous.data,
    mail.list = "pgilbert@bank-banque-canada.ca", message.title = " Manufacturing Monitoring",
    message.subject = "Manufacturing Monitoring", show.start = c(0,
      -3), report.variables = seriesNames(manuf.data.ids),
    data.sub.heading = "%chg %chg", message.footer = " f - forecast value",
    data.tag = " ", forecast.tag = "f")
}
```

The status of the result can be checked with

```
> if (FALSE) r$status
```

and the comparison data should also be updated with

```
> if (FALSE) manuf.previous.data <- r$data
```

Especially for debugging purposes it is often useful to keep a more complete record of the data and model used to produce the forecast. This can be done with the `simpleMonitoring` argument `save.as` which can be set to specify a file name. Setting `save.as=paste("Manufacturing.monitoring.", make.names(date()), sep="")` in the above would make a file name which includes a time stamp. Also, setting the argument `run.again=TRUE` will run the forecast without checking to see if the data has been updated.

The argument `mail.list` allows the output to be mailed to a list of recipients, but it may be more convenient to mail the result to a list server which can be used for distribution purposes. This may be easier to maintain, as the list server list of recipients can be changed at any time (and in automatic mode described next the program does not have to be restarted.)

7.5 Step 5 - automatic program to run the monitoring

To run the above and e-mail forecast directly from the Unix command prompt a shell script can be set up as follows:

Below it is assumed this is in a file called `manufacturing`. To run this automatically every 20 minutes from 7am to 10am the script

could be put in a file `monitoring.daemon` and then this can be started at the Unix prompt with the command

```
unix prompt: monitoring.daemon manufacturing
```

The disadvantage of this approach is that the overhead for starting `Splus` is fairly heavy and it may be difficult to use your computer for much else from 7am to 10am. (R may be better in this respect.) If you have direct access to the files used for the database then the script could be modified to check time stamps on the files and only run if the file date has changed. If database files are used to store many series, and not all are updated at the same time, then the savings will not be much. At the Bank of Canada another script called `Data.trigger.daemon` can be used to run a `Fame` procedure to check if the particular series have been updated, and then run `manufacturing` only in that case.

8 Appendix

There is no appendix yet.