Brief User's Guide: Dynamic Systems Estimation Library

Paul Gilbert

February 10, 2003

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.

This draft of the Guide reflects many changes but should be considered a work in progress at this time. It is being converted to use the R Sweave utilities (see F. Leisch, R News v2/3, Dec. 2002, p 28-31), but there is still work left to do in that conversion. The graphics, section numbering, and table of contents all need improvement, but in particular, 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.

I regularly test the code with R on Solaris and Linux and sometimes with Splus 3.3 on Solaris. There are known problems with Splus 5. Please report errors you find.

Caveat: This software is the by-product of ongoing research. It is not a commercial product. Limited effort is put into maintaining the documentation. There may be 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. While the software does many standard time-series things, it is really intended for doing some non-standard things. The main difference between this library and many widely available packages is that the library 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 at <PaulGilbert@ Ottawa. com> or by phone at (613) 782-7346.

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

Preamble

DSE Bundle Contents

DSEplus Bundle Contents (not included in DSE guide)

Other Related Package Contents (not included in DSE guide)

Preamble

1 Introduction to DSE

This library was originally designed with linear, time-invariant autoregressive moving-average (ARMA) models and state-space (SS) models in mind. These remain the most well developed models in the library and provide the basis for most of the examples in this guide. However, the library implements object oriented methods for studying new estimation techniques and other kinds of time series models. Methods are implemented for studying Troll (Intex Solutions, Inc.) models (currently broken) and some neural net architectures are being explored. These provide examples for implementing new model objects and estimation methods. Users are encouraged to consider specific representations used in this guide as examples in the context of the library's broader objectives.

In order to provide examples the library also implements some estimation techniques and methods for converting among various representations of time series models. Many functions for the usual diagnostics which are preformed with time series data and models are included as well. Additional information on specific functions is available through the help facility. For details of some of the underlying theory of ARMA and SS model equivalence and examples of some of the capabilities of the library see Gilbert (1993). For examples of using the library to evaluate estimation methods see Gilbert (1995). Examples of the use of several functions are illustrated in the files in the demo subdirectories. (In R see *demo()*)

2 Getting Started with S/R

This library works with the Splus 3.3 (MathSoft <http:www.insightful.com>) version of the S language and recent versions of the R language (Ihaka and Gentleman, 1996) <http://cran.r-project.org>. The notation S/R will be used to indicate both languages and S or R will be used when a remark is specific to one or the other. Splus will be indicated when a remark may be specific to Splus but not to S in general. Italics will be used to indicate examples as well as functions and objects, and () will frequently be added to function names to help distinguish them as such. Anything entered after a # is a comment in S/R.

This guide explains certain aspects of S/R in order to make the library accessible to users unfamiliar with S/R. Knowledge of the S/R

language is extremely useful. Users are referred to Becker, Chambers and Wilks (1988, commonly known as The Blue Book), Venables and Ripley (various editions), Ripley (1994), Burns(...), Krause and Olson(...), or to the user manuals for their implementation. Users already familiar with S/R should ignore the simplified explanations given in this section and skip directly to the next section.

An important point is that S/R functions take arguments in brackets (...), even when there are no argument. So, for example, the function to get out of S/R and back to the operating system is q(). Values are assigned to S/R variables with the two character symbol "<-". Also, it is important to be aware that upper and lower case letters are different. Most examples in this guide show only the user input, not the computer output.

If DSE is not installed on your system, please use the usual R package installation procedures or see installation.txt in the source distribution. Once S/R is started the DSE library must be made available. In S this is done with *library("DSE", first=T)* and *load.DSE.fortran()* or in R by either *library()* or *require()* for each package. For example, to use the *dse1* package:

```
library("dse1")
```

Other required packages will be automatically attached. You should consider putting these lines in your .First function, which is automatically executed each time you start. (This is especially advisable in Splus as some of the more computationaly intensive functions in this library spawn separate Splus sessions to speed calculations.)

Descriptions of functions and objects are available in the help system. This is integrated with the R help system (started with *help.start()*). HTML help is also available in S and can be viewed with a web browser. From an S session this can be started with the function *help.start.DSE()*, which tries to start Netscape by default, but any browser can be specified with *help.start.DSE(browser= "my.browser")*. The string passed as *browser* should be a system command for starting a web browser.

3 General Outline of DSE Objects and Methods

The library implements three main classes of objects: TSdata, TSmodel, and TSestModel. These are respectively, representations of data, models, and models with data and estimation information.

TSdata is an object which contains a (multivariate) time series object called output and optionally another called input. Methods for defining the general version of this class of object are described in the next section and more details are provided in the help for TSdata. Input and output correspond to what are often labelled x and y in econometrics and time series discussions of ARMA models. These are sometimes called exogenous and endogenous variables, though those terms are often not correct for these models. Statistically, output is the variable which is modelled and input is the conditioning data. From a practical and computational point of view, the model forecasts output data and input data must always be supplied. In particular, to forecasts multiple periods into the future requires supplying input data for the future so that the model can calculate outputs. The terms input and output are commonly used in the engineering literature, and often correspond to a control variable and the output from a physical system. However, the causal interpretation in this context is not always appropriate for other uses of time series models. In addition, even when a causal direction is known or assumed, it is not always desirable to define the exogenous variable as an input. If the model is to give forecasts into the future then it may be better to define exogenous variables as outputs and let the model forecast them, unless better forecasts of the exogenous variables are available from other sources. One context in which an input variable is important is to examine policy scenarios. In this context the policy variable is defined as the input and forecasts are produced conditioned on different assumptions about the policy.

TSmodel objects are models which are arranged to use TSdata. These objects always have another specific class indicating the type of model. The ARMA and SS constructor methods for ARMA TSmodels and state-space TSmodels are described in a section below. Other specific classes of TSmodels can be defined and many of the methods in this library will work with these new models, as long as they use TSdata and have a few important methods implemented. More details on defining other classes of models are given in another section of this guide. Details on the representation of models are provided in the help for TSmodel and the help for specific model constructors.

TSestModel objects are objects which contain TSdata, a TSmodel, and some statistical information generated by *l(model, data)*. The *l()* method originally meant likelihood, but the method returns the one-step-ahead predictions and other information based on those predictions. Methods for studying one-step-ahead model forecasts extract the predictions from these objects. Other methods treat *TSestModel* objects as a simple way to group together a model and data. For

example, methods for studying multi-step forecasts need to generate the forecasts, so they do not use the predictions in the *TSestModel* object. More detail about *TSestModel* objects is available in the help system.

The default method for *TSdata()* constructs a *TSdata* object, as will be described in the next section. The generic methods *TSmodel()* and TSdata() can also be used to extract the TSmodel or TSdata object from another object (such as a *TSestModel*).

The functions in this library can be used by starting with data and estimating a model, or by starting with a model and producing simulated data. The section on *TSdata* starts with data, but it would be equally possible to start with models as described in the sections on ARMA and State-Space *TSmodels*.

dse Bundle

4 dse1 Guide

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

```
> library("dse1")
```

Several data sets are included with this library and will be used in examples in this guide. In S these are available when the library is attached. In R they are made available by

```
> data(package = "dse1")
> data(eg1.DSE.data, package = "dse1")
> data(egJofF.1dec93.data, package = "dse1")
```

4.1 Defining a TSdata Structure

This section describes how to construct a *TSdata* structure if you have other data you would like to use. Section 10 discusses adding new kinds of *TSdata* classes. Some installations may have an online database and it may be possible to connect directly to this data. See the padi and dsepadi packages as on one possibility for doing this.

For many people the situation will be that the data is in some ASCII file. This can be loaded into session variables with a number of standard S/R functions, the most useful of which are probably *scan()* and *read.table()*. Following is an example which reads data from an ASCII

file called "eg1.dat" and puts it in the variable called *eg1.DSE.data* (which is also one of the available data sets). The file has five columns of numbers and 364 rows. The first column just enumerates the rows and is discarded.

This matrix can be used to form a *TSdata* object by

The matrix and the resulting *TSdata* object do not have a good time scale associated with points. A better time scale can be added by

```
> eg1.DSE.data <- tframed(eg1.DSE.data, list(start = c(1961, 3),
      frequency = 12))
```

There are several different possibilities for representing time in S/R objects. The most common is the ts matrix object, which is used in the above default tframed method. (ts is a class in R. In S it is not a class of object, but the default representation of time series which existed before classes were introduced.) The above tframed method and ts can also be used directly on the matrix before the TSdata object is formed. However, [,] in Splus results in the time scale being lost, so it would need to be reassigned to the input and output matrices of the TSdata object. The methods from the tframe library are used extensively in the DSE library because they provide a common way to proceed in Splus and R, extend to other time representations in addition to ts, and provide a mechanism for extending methods to other objects like TSdata and TSmodels.

Names can be given to the series with

```
> seriesNamesInput(eg1.DSE.data) <- "R90"
> seriesNamesOutput(eg1.DSE.data) <- c("M1", "GDP12", "CPI")
```

Setting the series names is not necessary but many functions can use the names if they are available. (This overlaps somewhat with S/R dimnames, but is the preferred method in this library as it extends to data which is not a matrix.) The *TSdata* object with elements input and output is the structure which the functions in this library expect. More details on this structure are available in the help for TSdata. The input and output elements can be defined in a number of different ways and new representations can be fairly easily added. For example, when the data is on a remote database as used by TSPADI, the S/R object is just a description of where to get the data, rather than the data itself. In this case the *freeze()* function is used automatically by many functions in the DSE library in order to get a copy of the data when calculations are to be performed.

Once data is available a model can be estimated:

```
> model1 <- est.VARX.ls(eg1.DSE.data)
> model2 <- est.SS.Mittnik(eg1.DSE.data, n = 4)
```

(Note: these models are not the same as those reported in Gilbert,1993. In that paper a variant of *est.VARX.ar* was used.) The scale of the series in *eg1.DSE.data* are very different, with the result that the covariance matrix of the residuals from the estimation is nearly singular. This is detected during the calculation of residual statistics. Statistics are then calculated using only the non-degenerate subspace and a warning message is printed. A better model might be obtained if the data were scaled differently.

Information about the estimated models can be displayed, for example:

```
> summary(model1)
> summary(model2)
> model1
> model2
> stability(model1)
> stability(model2)
> information.tests(model1, model2)
```

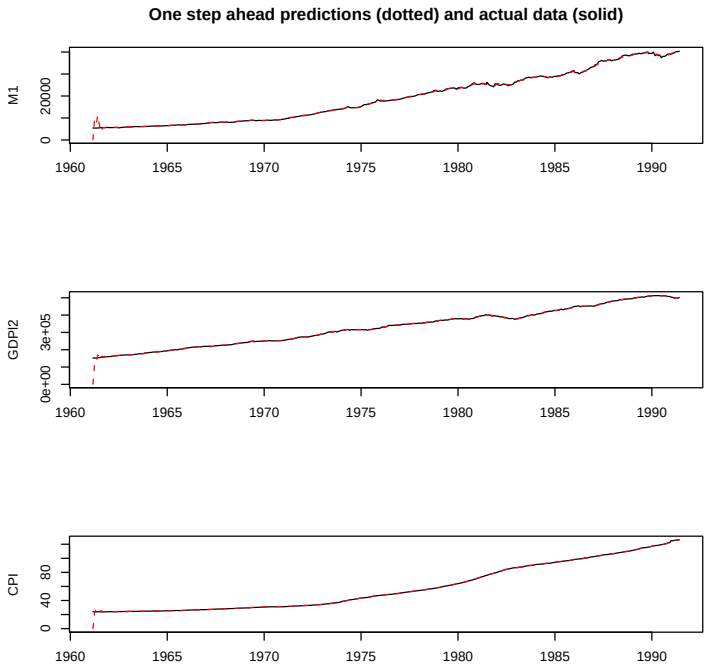
Typing the name of an object in S/R results in the object being printed. To display plots it is first necessary to open a graphics window:

```
> x11()
```

Once a graphics display is active then plots can be viewed:

```
> tfplot(model1)
> tfplot(model2)
> tfplot(eg1.DSE.data)
> check.residuals(model1)
> check.residuals(model2)
```

The function *tfplot* produces separate graphs for each series. The first



***tfplot* command produces this graphic:**

Note that initial conditions have been set to zero, but the effect of this dies out quickly. (Also note that the graph labels may be slightly different depending on which version DSE and of R or S you are using.)

4.2 ARMA and State Space TSmodels

Specifying ARMA and SS models is described below, but first their definition is outlined. The linear time-invariant ARMA representation is

$$A(L)y_t = B(L)e_t + C(L)u_t \tag{1}$$

where y_t is a p dimensional vector of observed output variables, u_t is an m dimensional vector of input variables, e_t is a p dimensional unobserved disturbance vector process and A, B and C are matrices of the appropriate dimension in the lag (back shift) operator L. VAR models can be thought of as a special case of ARMA models with B(L)=I. ARIMA models are also a special case of ARMA models.

Note that the time convention here implies that the input variable u_t can influence the output variable y_t in the same time period. This convention is not always used in time-series models but is important for economics data, especially at annual frequencies.

A linear time-invariant state space representation in innovations form is given by

$$\begin{aligned} z_t &= Fz_{t-1} + Gu_t + Ke_{t-1} \\ y_t &= Hz_t + e_t \end{aligned} \tag{2}$$

where z_t is the unobserved underlying n dimensional state vector, F is the state transition matrix, G, the input matrix, H, the output matrix, and K, the Kalman gain. The library also has some limited capabilities to work with the more general non-innovations form

$$\begin{aligned} z_t &= Fz_{t-1} + Gu_t + Qn_t \\ y_t &= Hz_t + Re_t \end{aligned} \tag{3}$$

where n_t is the system noise, Q, the system noise matrix, and R the output (measurement) noise matrix.

Models are specified by setting up the arrays that define the model and grouping them into a TSmodel object. Here is an example ARMA model with two series, a second order AR polynomial, a first order MA polynomial and no exogenous variable:

```
> AR <- array(c(1, .5, .3, 0, .2, .1, 0, .2, .05, 1, .5, .3),
              c(3, 2, 2))
> MA <- array(c(1, 0.2, 0, 0.1, 0, 0, 1, 0.3), c(2, 2, 2))
> arma <- ARMA(A = AR, B = MA, C = NULL)
> rm(AR, MA)
> arma
> stability(arma)
> data.arma.sim <- simulate(arma)
> arma <- l(arma, data.arma.sim)
> summary(arma)
> tfplot(data.arma.sim)
> tfplot(arma)
```

Note that arrays are filled in the order of their dimensions, which may not be what you expect. The internal representation of TSmodels may be described in the help for the specific model constructors, but in general it should be considered "opaque" and an understanding of the internal data structure should not be necessary to use the

models. The function *l()* evaluates the model with the simulated data. Functions generally use default values for some arguments. For example, the length of the simulation and the covariance of the noise can be specified. The above example uses the default values. See the help on simulate for more details. In the example above, arma is initially assigned an object of class TSmodel, but it is then re-assigned the value returned by *l()*, which is an object of class TSestModel. Also, many functions work with different classes of objects, and do different things depending on the class of the argument. The function *tfplot()* works with objects of class *TSdata* and *TSestModel*.

Here is an example of a state space model:

```
> f <- array(c(0.5, 0.3, 0.2, 0.4), c(2, 2))
> h <- array(c(1, 0, 0, 1), c(2, 2))
> k <- array(c(0.5, 0.3, 0.2, 0.4), c(2, 2))
> ss <- SS(F = f, H = h, K = k)
> print(ss)
> stability(ss)
> data.ss.sim <- simulate(ss)
> ss <- l(ss, data.ss.sim)
> summary(ss)
> tfplot(ss)
```

Data which has been generated with simulate is a TSdata object and can be used with estimation routines. This provides a convenient way to generate data for estimation algorithms, but remember that estimation will not necessarily get back to the model you start with, since there are equivalent representations (see Gilbert, 1993). However, a good estimate will get close to the likelihood and predictions of the original model.

Here is an example of changing between state space and ARMA representations using the models defined in the previous example:

```
> ss.from.arma <- l(to.SS(arma), data.arma.sim)
> arma.from.ss <- l(to.ARMA(ss), data.ss.sim)
> summary(ss.from.arma)
> summary(arma)
> summary(arma.from.ss)
> summary(ss)
> stability(arma)
> stability(ss.from.arma)
```

The function *roots()* is used by *stability()* and can be used by itself to return the roots but not evaluate their magnitude footnoteBy default

the roots of an ARMA model are calculated by converting the model to state space form, for reasons explained in Gilbert (2000). By specifying by.poly=T the method can be changed to use an expansion of the polynomial determinant.. When their arguments are *TSmodel*s the functions *to.SS()* and *to.ARMA()* return objects of class *TSmodel* which are not assigned to a variable in the above example, but used in the evaluation of *l()*. The models are returned as part of the *TSestModel* returned by *l()*.

4.3 Model Estimation

The example data *eg1.DSE.data* and *egJofF.1dec93.data* are available with the DSE library and are used in examples in this section.

To estimate an AR model with the default number of lags:

```
> model.eg1.ls <- est.VARX.ls(trim.na(eg1.DSE.data))
```

In this example trim.na removes NA padding from the ends of the data, since the estimation method cannot handle missing values. This padding may not be present, depending on how the data was retrieved. This data is highly correlated and highly parameterized models result in a degenerate covariance matrix. When this happens a warning is produced in this and other examples.

It is also possible to select a subsample of the data:

```
> subsample.data <- tfwindow(eg1.DSE.data, start = c(1972, 1),
                                           end = c(1992, 12))
```

This creates a new variable with data starting in January 1972 and ending in December 1992. The S/R function window also usually works, however the function tfwindow is typically used in the DSE library and this guide because it has occasionally been necessary to correct some problems with window. Various functions can be applied to the estimation result

```
> summary(model.eg1.ls)
> print(model.eg1.ls)
> tfplot(model.eg1.ls)
> check.residuals(model.eg1.ls)
```

Other estimation techniques are available

```
> model.eg1.ar <- est.VARX.ar(trim.na(eg1.DSE.data))
> model.eg1.ss <- est.SS.from.VARX(trim.na(eg1.DSE.data))
```

```
> model.eg1.bft <- bft(trim.na(eg1.DSE.data))
> model.eg1.mle <- est.max.like(est.VARX.ls(trim.na(eg1.DSE.data),
                                            max.lag = 1))
```

***tfplot* can put multiple similar objects on a plot**

```
> tfplot(model.eg1.ls, model.eg1.ar)
> tfplot(model.eg1.ls, model.eg1.ar, start. = c(1990, 1))
```

Because of this, the argument *start.* requires a period at the end so it is not confused with an object to be plotted.

Most of the estimation techniques have several optional parameters which control the estimation. Consult the help for the individual functions. *est.max.like* extracts data from a TSestModel and uses the model structure and initial parameter values for the estimation. (Note: Maximum likelihood estimation can be very slow and may not converge in the default number of iterations. It also tends to over fit unless used with care, so that out-of-sample performance is not good. I do not generally recommend it, although it does offer possibilities for constraining the structure in specific ways (e.g. fixing some model matrix entries to zero or one). You might consider comparing mle to other estimation techniques using functions discussed in the following sections.) In the above *est.max.like* example a smaller (one lag) model is used. Be prepared for the estimation to take some time when models have a large number of parameters.

An important point to note is that the one-step-ahead predictions and related statistics returned by these estimation techniques are calculated by evaluating l(model, data) as the final step after the model has been estimated. This can give different results than might be expected using the estimation residuals, particularly with respect to initial condition effects. (For stable models initial condition effects should not be too important. If they are an important factor check the documentation for specific models regarding the specification of initial conditions.)

Also remember when estimating a model that, if you want to predict future values of a variable, it will need to be an output in the TSdata object.

For the next example a four variable subset of the data in *egJofF.1dec93.data* will be used. This subset is extracted by

```
> eg4.DSE.data <- egJofF.1dec93.data
> output.data(eg4.DSE.data) <- output.data(eg4.DSE.data,
                                    series = c(1, 2, 6, 7))
```

which selects the 1st, 2nd, 6th, and 7th series of the output data. The following uses the currently preferred automatic estimation procedure:

```
> model.eg4.bb <- est.black.box(trim.na(eg4.DSE.data), max.lag = 3)
```

An optional argument *verbose=F* will make the function print much less detail about the steps of the procedure. The optional argument, *max.lag=3*, specifies the maximum lag which should be considered. The default *max.lag=12* may take a very long time for models with several variables. *est.black.box* currently uses *est.black.box*4, also known as *bft(..., standardize=T)* which is called the brute force technique in Gilbert (1995).

The traditional model information criteria tests can be performed to compare models:

```
> information.tests(model.eg1.ar, model.eg1.ss)
```

An arbitrary number of models can be supplied. The generated table lists several information criteria. For state space models the calculations are done with both the number of parameters (the number of unfixed entries in the model arrays) and the theoretical parameter space dimension. See Gilbert (1993, 1995) for a more extensive discussion of this subject.

Note that converting among representations produces input-output equivalent models, so that predictions, prediction errors, and any statistics calculated from these, will be the same for the models. However, different estimation techniques produce different models with different predictions. So, *est.VARX.ls(data)* and *to.SS(est.VARX.ls(data))* will produce equivalent models, and *est.SS.Mittnik(data)* and *to.ARMA(est.SS.Mittnik(data))* will produce equivalent models, but the first two will not be equivalent to the second two.

5 dse2 Guide

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

```
> library("dse2")
```

The next code lines are here to initialize results from examples in dse1 that are used in dse2 examples.

```
> data(egJofF.1dec93.data, package = "dse1")
> eg4.DSE.data <- egJofF.1dec93.data
> eg4.DSE.model <- est.VARX.ls(eg4.DSE.data)
> output.data(eg4.DSE.data) <- output.data(eg4.DSE.data,
                                           series = c(1, 2, 6, 7))
> eg4.DSE.model <- est.VARX.ls(eg4.DSE.data)
> new.data <- TSdata(
   input = ts(rbind(input.data(eg4.DSE.data), matrix(0.1, 10, 1)),
              start = start(eg4.DSE.data),
              frequency = frequency(eg4.DSE.data)),
   output = ts(rbind(output.data(eg4.DSE.data), matrix(0.3, 5, 4)),
              start = start(eg4.DSE.data),
              frequency = frequency(eg4.DSE.data)))
> if (require("padi") & require("dsepadi"))
    eg4.DSE.data.names <- TSPADIdata(
       input = "B14017", input.transforms = "diff", input.names = "R90",
       output = c("P100000", "V2036138", "V2062811", "b3400"),
           output.transforms = c("percent.change",
                "percent.change", "percent.change", "percent.change"),
           output.names = c("CPI", "GDP", "employment", "PFX"),
      server = "ets")
```

5.1 Forecasting

The TSestModel object returned by estimation is a TSmodel with TSdata and some estimation information. To use different data, the new data needs to be in a variable which is a TSdata object. For example, suppose a model is estimated by

```
> eg4.DSE.model <- est.VARX.ls(eg4.DSE.data)
```

and suppose new data becomes available. If you have direct database access this might be done with something like

```
> if (require("padi") && checkPADIserver("ets"))
    new.data <- freeze(eg4.DSE.data.names)
```

If database access is not available then, for example purposes, new.data can be generated with

```
> new.data <- TSdata(
    input = ts(rbind(input.data(eg4.DSE.data), matrix(0.1, 10, 1)),
               start = start(eg4.DSE.data),
               frequency = frequency(eg4.DSE.data)),
```

```
output = ts(rbind(output.data(eg4.DSE.data), matrix(0.3, 5, 4)),
            start = start(eg4.DSE.data),
            frequency = frequency(eg4.DSE.data)))
```

This simply appends ten observations of 0.1 onto the input and five observations of 0.3 onto the outputs. The function ts assigns time series attributes which are taken from *eg4.DSE.data*. The model can be evaluated with the new data by

```
> z <- l(TSmodel(eg4.DSE.model), trim.na(new.data))
```

Recall that *TSmodel()* extracts the *TSmodel* from the *TSestModel*. If database access is available the above can be done in one step:

```
> if (require("padi") && checkPADIserver("ets"))
   z <- l(TSmodel(eg4.DSE.model), trim.na(freeze(eg4.DSE.data.names)))
```

***trim.na* on a *TSdata* object removes NAs from the ends and truncates both input and output to the same sub-sample. *l()* does not easily give forecasts beyond the period where all data is available. (Optional arguments can be used to achieve this, but the function forecast is more convenient.)**

Forecasts are conditioned on input so it must be supplied for periods for which forecasts are to be calculated. (That is, input is not forecast by the model.) When more data is available for input than for output, as in new.data generated above, then *forecast()* will use input data and produce a forecast of output.

```
> z <- forecast(TSmodel(eg4.DSE.model), new.data)
```

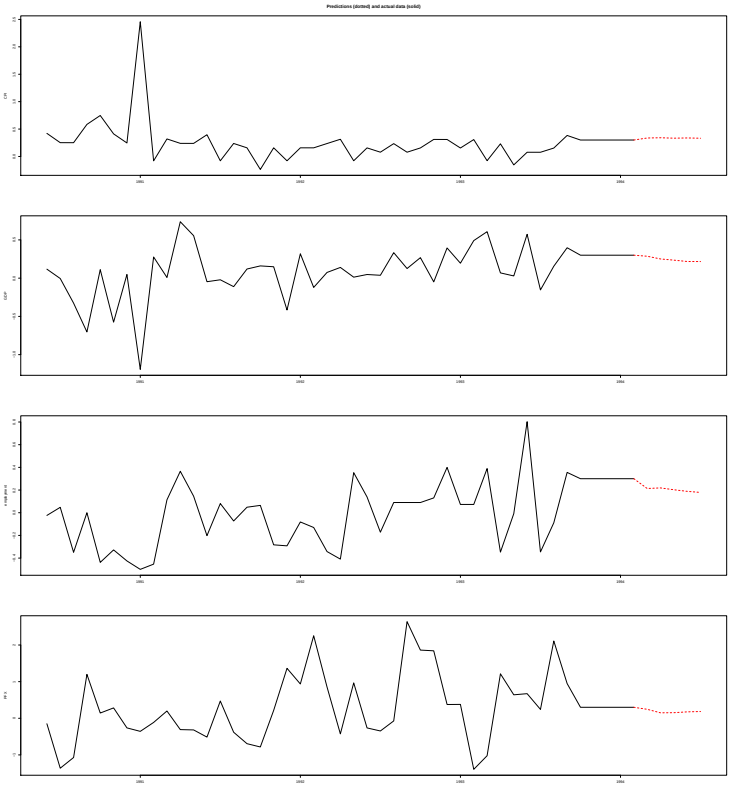
The input data can also be specified as a separate argument. For example, the same result will be achieved with

```
> z <- forecast(TSmodel(eg4.DSE.model), trim.na(new.data),
                conditioning.inputs = input.data(new.data))
```

The conditioning.inputs override input in the TSdata supplied in the second argument to the function.

To see plots of the forecasts use

```
> tfplot(z, start = c(1990, 6))
```



Sometimes a forecast for input data comes from another source, perhaps another model. Rather than construct the *conditioning.inputs* as described above, another way to combine this forecast with the historical input data is to use the argument *conditioning.inputs.forecasts*:

```
> z <- forecast(eg4.DSE.model,
           conditioning.inputs.forecasts = matrix(0.5, 6, 1))
```

This would use the input data from *eg4.DSE.model* and append 6 periods of 0.5 to it.

```
> if (require("padi") && checkPADIserver("ets"))
   z <- forecast(TSmodel(eg4.DSE.model),
                 freeze(eg4.DSE.data.names),
              conditioning.inputs.forecasts = matrix(0.5, 6, 1))
```

retrieves new data and appends 6 periods of 0.5 to the input series

Some generic functions which work with the structure returned by forecast:

```
> summary(z)
> print(z)
> tfplot(z)
> tfplot(z, start = c(1990, 1))
```

If you actually want the numbers from the forecast they can be extracted with

```
> forecasts(z)[[1]]
```

The *[[1]]* indicates the first forecast (in this example there is only one, but the same structures are used for other purposes discussed below. To see a subset of the data use *tfwindow*:

```
> tfwindow(forecasts(z)[[1]], start = c(1994, 1))
```

This prints values starting in the first period of 1994.

The horizon for the forecast is determined by the available input data (*conditioning.inputs* or *conditioning.inputs.forecasts*). If neither of these are supplied then the argument *horizon*, which has a default value of 36, is used to replicate the last period of data to the indicated horizon. For models with no input variables the argument horizon controls the length of the forecast.

5.2 Evaluating Forecasting Models

How well does the model do at forecasting? The first thing to check is that model forecasts actually track the data more or less. The generic function tfplot() works with results from the following functions. Recall that the function l() applies a TSmodel to TSdata and returns a TSestModel which includes one-step ahead forecasts. It can be used with any TSmodel and TSdata of corresponding dimension. So

```
> z <- l(TSmodel(eg4.DSE.model), new.data)
```

applies the previously estimated model to the new data, and

```
> tfplot(z)
```

would plot the one-step ahead forecasts. The function forecast discussed in the previous section calculates multi-step ahead forecasts from the end of the data. For evaluating forecasting models it is more useful to calculate forecasts within the sample of available data. This is for two reasons. First, the forecast can be compared against the actual outcome. Second, if the model has an input then the forecast

is conditioned on it. If data is available then the actual input data can be used. (But beware that this is not a true test of the model's ability to forecast if the whole sample has been used to estimate the model.) There are two methods to calculate multi-step ahead forecasts within the data sample. featherForecasts produces multiple period ahead forecasts beginning at specified periods. The name comes from the fact that the graph sometimes looks like a feather (although it will not if the forecasts are good).

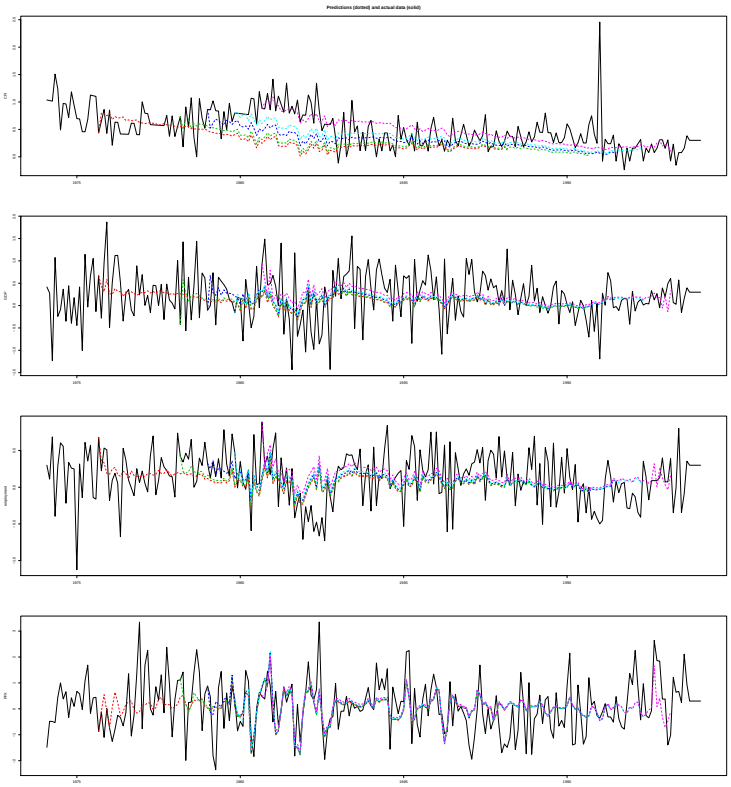
```
> z <- featherForecasts(TSmodel(eg4.DSE.model), new.data)
> tfplot(z)
```

In the example above the forecasts begin by default every tenth period. In the following example the forecasts begin at periods 20, 50, 60, 70 and 80 and forecast for 150 periods.

```
> z <- featherForecasts(TSmodel(eg4.DSE.model), new.data,
              from.periods = c(20, 50, 60, 70, 80), horizon = 150)
```

The plot looks like this:

```
> tfplot(z)
```

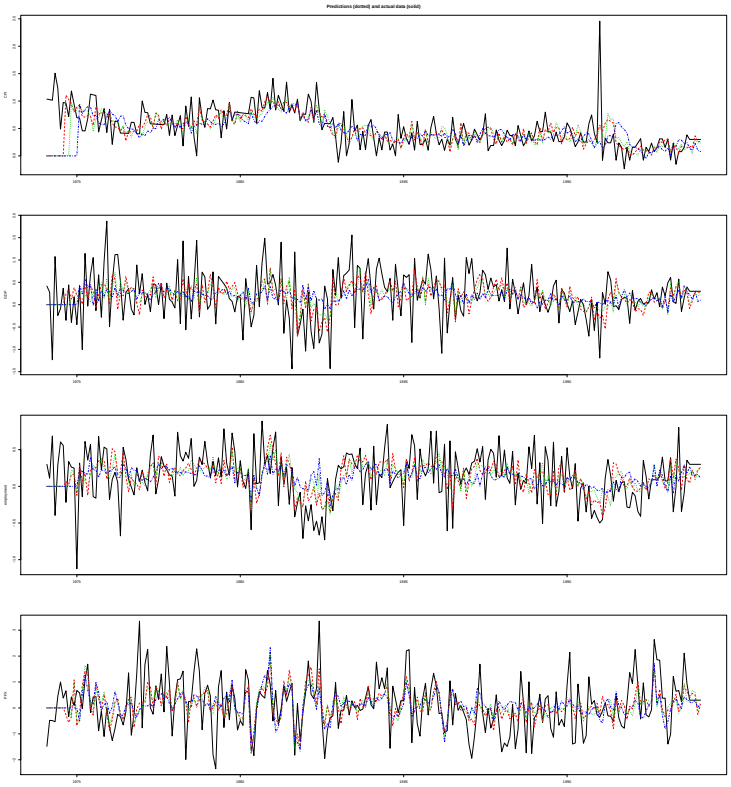


The second method, *horizonForecasts*, produces forecasts from every period for specified horizons.

```
> z <- horizonForecasts(TSmodel(eg4.DSE.model), new.data,
            horizons = c(1, 3, 6))
```

produces forecasts 1, 3 and 6 steps ahead. The plot looks like this:

```
> tfplot(z)
```



The result is aligned so that the forecast for a particular period is plotted against the actual outcome for that period. Thus, in the last example, the plot will show the data for each period along with the forecast produced from 1, 3, and 6 periods prior. This plot is particularly useful for illustrating when models do well and when they do not. A common experience with economic data is that models do well during periods of expansion and contraction, but miss the turning points. The forecast covariance, to be discussed next, averages over all periods. It is quite possible that a model can indicate turning points well but not do so well on average, and thus be overlooked if

only forecast covariance is considered. It is always useful to keep in mind the intended use of the model.

The numbers which generate the above plot can be extracted from the result of horizonForecasts with forecasts(). This gives an array with the first dimension corresponding to the horizons and the time frame aligned to correspond to the data. So forecasts(z)[2,30,] from the above example will be the prediction made for the 30th period from 3 periods previous (the second element indicated in horizons is 3) and forecasts(z)[3,30,] will be the prediction made for the 30th period from 6 periods previous (horizons[3] is 6). Remember that these forecasts are conditioned on the supplied input data, which means that the output variables here are forecast 1, 3 and 6 periods ahead, but true, not forecasted, input data is used.

If the forecasts look reasonable then examine the forecast errors more systematically. The following calculates the forecast covariances at different horizons.

```
> fc <- forecastCov(TSmodel(eg4.DSE.model), data = eg4.DSE.data)
> tfplot(fc)
> tfplot(forecastCov(TSmodel(eg4.DSE.model), data = eg4.DSE.data,
      horizons = 1:4))
```

The last example calculates for horizons from 1 to 4 rather than the default 1 to 12. To see how the model forecasts relative to a zero forecast and a trend forecast:

```
> fc <- forecastCov(TSmodel(eg4.DSE.model), data = eg4.DSE.data,
      zero = T, trend = T)
> tfplot(fc)
```

This is a very useful check (and often very humbling).

You can also get out-of-sample forecast covariances. This will be discussed in the next section.

There is not yet implemented in DSE any measure of forecast errors which can be compared across models - inevitably the covariance of the error is smaller for less variable series and is also affected by scaling of the series. This may just mean that the series is easier to predict or has a different scale, not that the forecast equation is more brilliant. MAPE may be implemented sometime.

5.3 Evaluating Estimation Methods

One way to test estimation techniques is to specify a "true" model which is used to produce simulated data and then examine how well an estimation technique finds the true model. This is not as general as theoretical results, since it is really only valid at the "true" parameter values and for the sample size tested, however, it can be illustrative and theoretical results for small samples are very difficult to obtain. It also provides a very good cross check of the simulation and estimation code. Also, equivalent representations may have effects which are not yet fully appreciated in the literature. The following models from Gilbert (1995) will be used to illustrate.

```
> mod1 <- ARMA(A = array(c(1, -0.25, -0.05), c(3, 1, 1)),
               B = array(1, c(1, 1, 1)))
> mod2 <- ARMA(A = array(c(1, -0.8, -0.2), c(3, 1, 1)),
               B = array(1, c(1, 1, 1)))
> mod3 <- ARMA(A = array(c(1, -0.06, 0.15, -0.03, 0, 0.02, 0.03,
      -0.02, 0, -0.02, -0.03, -0.02, 0, -0.07, -0.05, 0.12, 1,
      0.2, -0.03, -0.11, 0, -0.07, -0.03, 0.08, 0, -0.4, -0.05,
      -0.66, 0, 0, 0.17, -0.18, 1, -0.11, -0.24, -0.09), c(4, 3,
      3)),
             B = array(diag(1, 3), c(1, 3, 3)))
```

mod2 has a unit root, as can be verified with roots(mod2) or stability(mod2).

The function MonteCarloSimulations runs simulate repeatedly to give many data samples.

```
> z <- MonteCarloSimulations(mod1, simulation.args = list(sampleT = 100))
> tfplot(z)
> distribution(z)
```

Usually it is not necessary to use MonteCarloSimulations and actually save all the simulations since the seed and other information about the random number generator (RNG) can be used to reproduce the samples. Thus functions for testing estimation methods can produce the same samples when they are needed.

The function EstEval simulates and then estimates models:

```
> e.ls.mod1 <- EstEval(mod1, replications = 100,
      simulation.args = list(sampleT = 100, sd = 1),
      estimation = "est.VARX.ls",
      estimation.args = list(max.lag = 2),
      criterion = "TSmodel",
```

```
    rng = list(kind = "default", normal.kind = "default",
            seed = c(13, 44, 1, 25, 56, 0, 6, 33, 22, 13, 13, 0)))
```

In this example simulation and estimation will be repeated 100 times with samples of size 100 and the standard deviation of the model noise will be set to 1. *simulation.args* are passed to the function simulate, which may take different arguments depending on the class of the model. Estimation is done with the function *est.VARX.ls* and *estimation.args* are passed to it. The argument criterion specifies what should be returned from the estimation. In this case the model is returned (An object of class TSmodel) but not additional information as is usually returned in the object TSestModel. It is also possible to specify coef or roots to return only that specific information, but that information can be extracted from the TSmodel as illustrated below. In general EstEval will work with any estimation method which will take the results of simulate applied to the supplied model and returns something that criterion can extract. That is, if criterion(estimation(simulate(model))) returns something (with criterion and estimation replaced by the functions you supply and model replaced by the model you supply), then EstEval should work with your functions. This does not mean that plots described below will necessarily work or make sense.

The argument rng is optional here and in all the examples below. If supplied, the RNG and seed will be set. This is useful if an experiment is to be reproduced. Using Splus 3.2 and 3.3 the settings indicated in this section will reproduce the results in Gilbert (1995). It is possible to generate similar random experiments in S and in R, but not using the Splus default generator. If the argument rng above is given as

```
> rng = list(kind = "Wichmann-Hill", seed = c(979, 1479, 1542),
      normal.kind = "Box-Muller")
```

then the uniform RNG is set to Wichmann-Hill, the normal transformation is set to Box-Muller, and the initial seed is set. With the RNG set in this way both Splus and R will produce similar results. These settings are reset to their previous values when the function completes. They can be set so that they do not revert using the function

```
> set.RNG(kind = "Wichmann-Hill", seed = c(979, 1479, 1542),
          normal.kind = "Box-Muller")
```

The argument seed is optional (and other values can be supplied but they should be consistent with the generator). An initial seed will be generated if it is omitted.

The following uses mod2 as the true model.

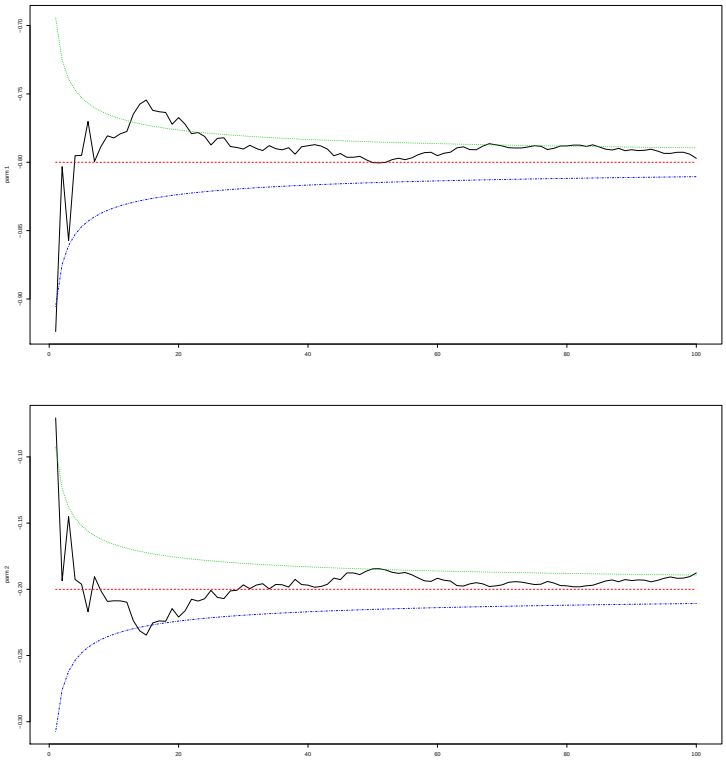
```
> e.ls.mod2 <- EstEval(mod2, replications = 100,
       simulation.args = list(sampleT = 100, sd = 1),
      estimation = "est.VARX.ls", estimation.args = list(max.lag = 2),
      criterion = "TSmodel")
```

To plot a line chart of the cumulative average of the estimated parameters use coef to extract the parameters (coefficients) from the TSmodel:

```
> par(mfcol = c(2, 1))
> tfplot(coef(e.ls.mod1))
```

The plot from mod2 looks like this:

```
> tfplot(coef(e.ls.mod2))
```

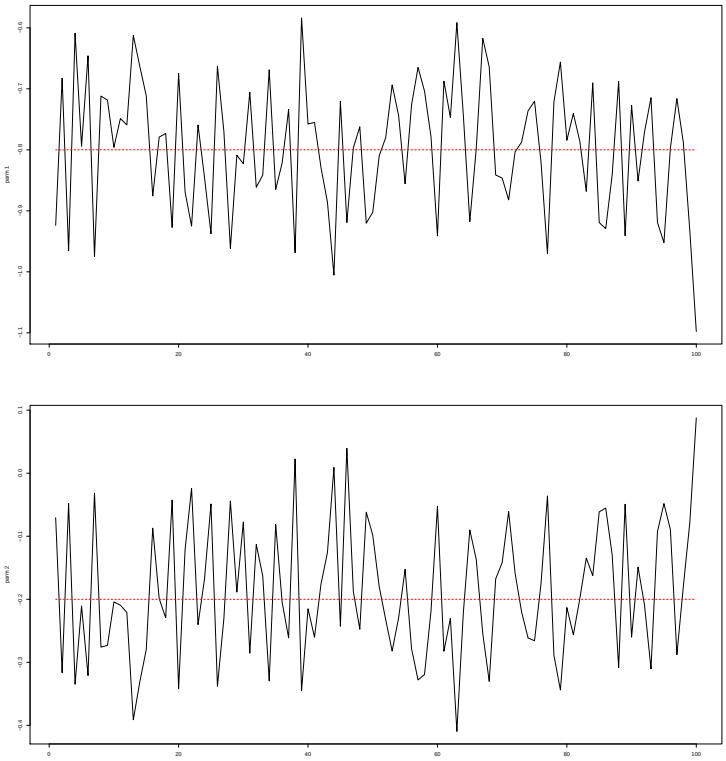


The straight line indicates the true value. To plot a line chart of the estimated parameters use coef to extract the parameters from the TSmodel:

```
> par(mfcol = c(2, 1))
> tfplot(coef(e.ls.mod1), cum = F, bounds = F)
```

bounds controls whether or not estimated one standard deviation bounds are plotted. The plot from mod2 looks like this:

```
> tfplot(coef(e.ls.mod2), cum = F, bounds = F)
```

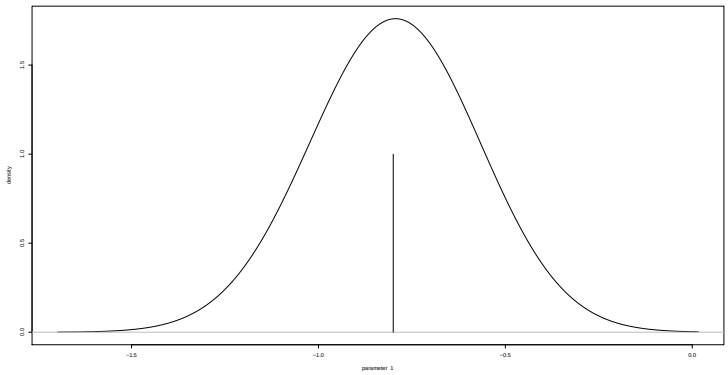


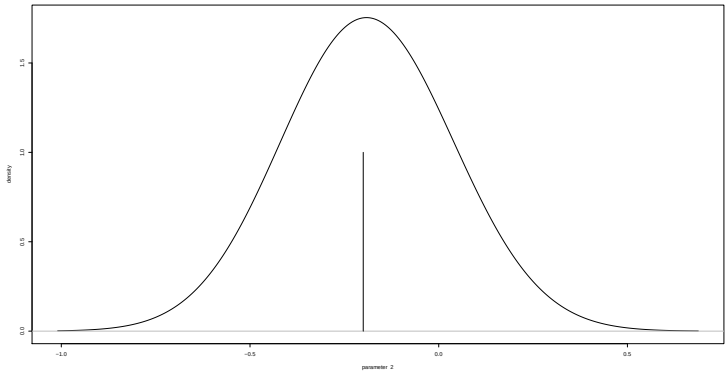
To plot the distribution of estimates:

```
> distribution(coef(e.ls.mod1), bandwidth = 0.2)
```

The plot from mod2 looks like this:

```
> distribution(coef(e.ls.mod2), bandwidth = 0.2)
```



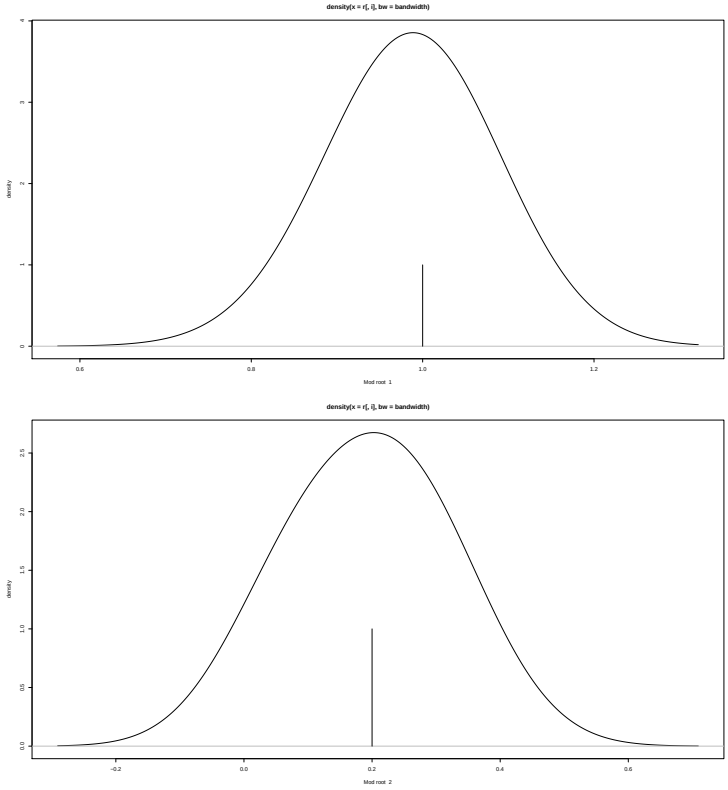


To plot the roots of the estimated model use roots to extract the roots from the TSmodel:

```
> e.ls.mod1.roots <- roots(e.ls.mod1)
> plot(e.ls.mod1.roots)
> plot(e.ls.mod1.roots, complex.plane = F)
> plot(roots(e.ls.mod2), complex.plane = F)
> distribution(e.ls.mod1.roots, bandwidth = 0.2)
```

bandwidth is an argument passed to the kernel estimator used to generate the plot. The plot from mod2 looks like this:

```
> distribution(roots(e.ls.mod2), bandwidth = 0.1)
```



Some attention to the equivalence of different model representations is necessary when evaluating estimation methods. For example, if the state space equivalent of a VAR model is used as the true model for simulation and *est.VARX.ls* is used for estimation then parameter estimates will be very different from those of the state space model (but root estimates should still be similar). Many estimation techniques may also do some model selection (such as est.black.box does), so the returned models may have different numbers of parameters and/or lags.

Evaluating models based on their forecast performance avoids some of these difficulties. In any case, since forecasting is often the end objective, it is useful to evaluate models directly on their forecasting performance. The function *forecastCovEstimatorsWRTtrue()* evaluates estimation methods using a given true model for simulation. It calculates the covariance of forecast errors of the estimated models relative to the output of the true model:

```
> pc <- forecastCovEstimatorsWRTtrue(mod3,
     estimation.methods = list(est.VARX.ls = list(max.lag = 6)),
```

```
      est.replications = 2,
      pred.replications = 10,
      rng = list(kind = "default", normal.kind = "default",
          seed = c(53, 41, 26, 39, 10, 1, 19, 25, 56, 32, 28, 3)))
```

The names of the elements in the list *estimation.methods* specify the estimation methods and their value is a list of the arguments to the method. If no arguments are required then the value should be specified as NULL. The covariance for forecasts of zero and a simple trend are also calculated. These are useful benchmarks. *est.replications* controls the number of times a sample is generated and used for estimating a model with each estimation method. *pred.replications* controls how many times the forecasts from the estimated model are compared with output from the true model. Thus the total number of simulations is est.replications + est.replications * pred.replications, so 22 in the above example.

A similar function is available which applies a model reduction procedure after the estimation:

```
> pc.rd <- forecastCovReductionsWRTtrue(mod3,
      estimation.methods = list(est.VARX.ls = list(max.lag = 3)),
      est.replications = 2,
      pred.replications = 10,
      rng = list(kind = "default", normal.kind = "default",
             seed = c(29, 55, 47, 18, 33, 1, 15, 15, 34, 46, 13, 2)))
```

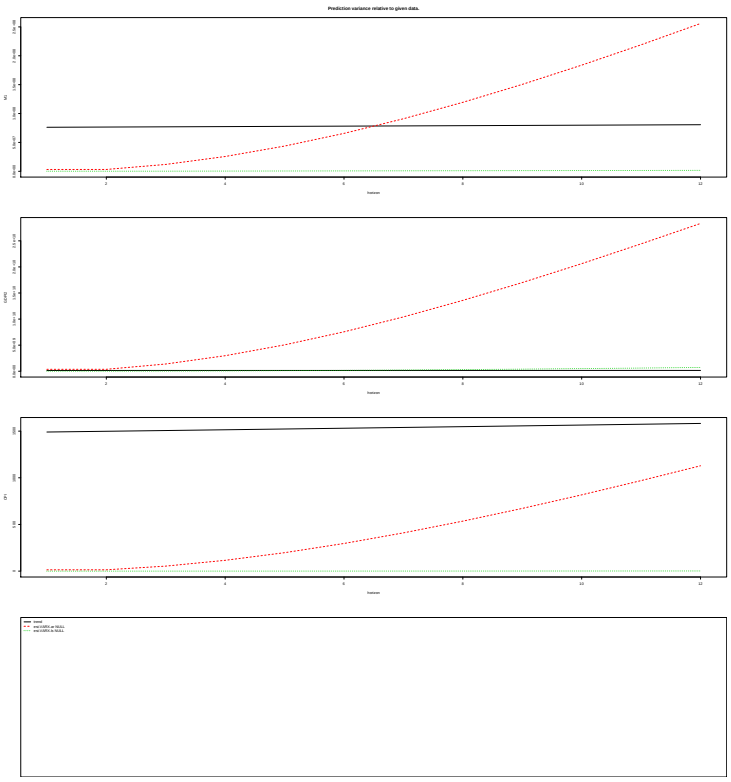
The reduction procedure used is reduced.models.Mittnik. An optional argument criteria can be specified. This controls the model selection criteria used by the reduction technique.

It is possible to compare different estimation techniques on the basis of their out-of-sample forecasting error with respect to a data sample. In the following example estimation.sample controls the portion of the sample used for estimation. It can be a fraction indicating a portion of the sample, or it can be an integer in which case it will be treated as the number of periods to use for estimation.

```
> data(eg1.DSE.data, package = "dse1")
> z <- out.of.sample.forecastCovEstimatorsWRTdata(trim.na(eg1.DSE.data),
      estimation.sample = 0.5,
      estimation.methods = list(est.VARX.ar = NULL, est.VARX.ls = NULL),
      trend = T)
```

The plot looks like this:

```
> tfplot(z)
```



In the example below the number of lags is limited (the default is 12 for *est.black.box4*) and printing of intermediate results is suppressed.

```
> z <- out.of.sample.forecastCovEstimatorsWRTdata(trim.na(eg1.DSE.data),
          estimation.sample = 0.5,
          estimation.methods = list(
               est.black.box4 = list(max.lag = 3, verbose = F),
               est.VARX.ls = list(max.lag = 3)),
          trend = T,
          zero = T)
> tfplot(z)
```

The object returned by out.of.sample.forecastCovEstimatorsWRTdata() contains the estimated models so it is possible to extract the models and use l, horizonForecasts and featherForecasts. In the above example the model estimated with est.black.box4 is the first model and that estimated with est.VARX.ls is the second, so

```
> zz <- horizonForecasts(TSmodel(z, select = 1), TSdata(z),
          horizons = c(1, 3, 6))
```

would generate an object with the actual forecasts for the model estimated with est.black.box4 (rather than the covariance of the forecast errors) and forecasts(zz)[3,30,] will then be the prediction made for the 30th period from 6 (the third element of horizons) periods previous. The generic function horizonForecasts() can also be applied directly to z and the appropriate information will be extracted to generate forecasts for all the estimated models.

5.4 Adding New TSdata Classes

Data used by functions in this library are objects of class TSdata. The default methods assume that this is a list with an element output and optionally an element input, each of which is a (multivariate) time series object. New classes of time series can be defined and the DSE library should work as long as the methods describe in the tframe library are implemented for the new time series class. This usually will not require any changes to TSdata methods (or anything else in the DSE library). The time series class tfPADIdata defined in the tframe library is an object which does not contain data, but only a description of where to get the data. The generic function freeze() calls freeze.tfPADIdata() which uses the location descriptor in order to get a fixed copy of the data as a time series matrix.

More generally, it is possible to define new specific classes of TSdata. The TSPADIdata object described in the appendix on database interfaces is an object of class TSdata and specific class TSPADIdata. The input and output for this class are time series location descriptors of class tfPADIdata. Many functions in this library require matrices for input and output in order to do calculations. In this case they use the function freeze() before doing any calculations. The method freeze.TSPADIdata() uses freeze.tfPADIdata() on each element.

5.5 Adding New TSmodel Classes

Models used in the library are of class "TSmodel" with secondary classes to indicate specific types of models. The original library supported subclass "ARMA" and "SS". The current version also support subclass "troll". (*** The interface for running troll models is broken at present. Another, more easily available example is under construction) To run models in this subclass requires the Troll software from Intex Solutions, Inc. It also requires the TSPADI interface. The main methods which will be necessary for a new class of models "xxx" are

print.xxx, is.xxx, l.xxx, simulate.xxx, seriesNamesInput.xxx, seriesNamesOutput.xxx, check.consistent.dimensions.xxx, and MonteCarloSimulations.xxx. Also, the method to.xxx is useful for converting models from existing classes to this new class where possible. Models should inherit from TSmodel.

The troll class of models is fairly interesting from a programming perspective, since the data is not native to S/R and the models are not run within S/R. One reason for wanting to do this is to use all of the other tools in the library to analyze models which have already been built and are running in other environments. Troll has very good algorithms for running "forward looking models" which are currently popular in economics. The tools in the DSE library (e.g. functions for analyzing forecasting properties) can be used as if the troll models were run directly in S/R, even though they are actually run with completely separate software.

The troll TSmodels provide an example of how to implement additional classes of models.

6 tframe Functions

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

```
> library("tframe")
```

User Guide documentation for the tframe functions is not yet done. See the help documention instead.

7 setRNG Functions

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

```
> library("setRNG")
```

User Guide documentation for the setRNG functions is not yet done. See the help documention instead.

8 Appendix I: Mini-Reference

Following is a short list of some of the functions. The online help contains more details on all functions, while the guides for each package contain more complete descriptions.

OBJECTS

- ARMA - define an ARMA TSmodel
- SS - define a state-space TSmodel
- TSdata - an input/output time series data structure
- TSestModel - a TSmodel estimated with TSdata

MODEL INFORMATION

- print - display model arrays
- summary - summary information about a model
- tfplot - plot data or model predictions.

MODEL PROPERTIES

- McMillan.degree - calculate the McMillan degree of a model
- roots - calculate the roots of a model
- stability - check stability of model

MODEL CONVERSION

- to.SS - convert to an equivalent state space innovations representation
- to.ARMA - convert to an ARMA representation

SIMULATION, ONE-STEP PREDICTIONS & RELATED STATISTICS

- simulate - Simulate a model to generate artificial data.
- l - evaluate a TSmodel with TSdata and return a TSestModel object
- smoother - calculate smoothed state for a state space model.
- check.residuals - distribution, autocorrelation and partial autocorrelation of residuals
- information.tests - print model selection criteria

MODEL ESTIMATION & REDUCTION

- est.VARX.ls - estimate VAR model with exogenous variable using OLS
- est.VARX.ar - estimate VAR model with exogenous variable using autocorrelations
- est.SS.from.VARX - estimate a VARX model and convert to state space
- est.SS.Mittnik - estimate state space model using Mittnik's markov parameter technique
- est.max.like - Maximum likelihood estimation of models.
- est.black.box - estimate and find the best reduced model
- est.black.box4 - estimate and find the best reduced model by techniques in Gilbert (1995), also referred to as bft
- reduction.Mittnik - nested-balanced state space model reduction by svd of Hankel generated from a model

FORECAST AND FORECAST EVALUATION

- forecast - generate a forecast from given model and data.
- featherForecasts - forecast from specified periods
- horizonsForecasts - forecast specified periods ahead
- forecastCov - calculate covariance of multi-period ahead forecasts

ESTIMATION EVALUATION

- EstEval - evaluate specified estimation techniques using a given true model
- out.of.sample.forecastCovEstimatorsWRTdata - evaluate specified estimation techniques using a given data set