dse2 Guide

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

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

As of R-2.1.0 the code from the vignette that generates this guide can be loaded into an editor with *edit(vignette("dse2"))*. 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("dse2"))* and the guide for the dse bundle with *print(vignette("dse-guide"))*.

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 <- estVARXls(eg4.DSE.data)
> outputData(eg4.DSE.data) <- outputData(eg4.DSE.data, series = c(1,
      2, 6, 7))
> eg4.DSE.model <- estVARXls(eg4.DSE.data)
> new.data <- TSdata(input = ts(rbind(inputData(eg4.DSE.data),
      matrix(0.1, 10, 1)), start = start(eg4.DSE.data), frequency = frequency(eg4.DSE.data))
      output = ts(rbind(outputData(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", "v37426"), output.transforms = c("percentChange",
          "percentChange", "percentChange", "percentChange"), output.names = c("CPI",
          "GDP", "employment", "PFX"), server = "ets")
```

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 <- estVARXls(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(inputData(eg4.DSE.data),
      matrix(0.1, 10, 1)), start = start(eg4.DSE.data), frequency = frequency(eg4.DSE.data))
      output = ts(rbind(outputData(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), trimNA(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),
      trimNA(freeze(eg4.DSE.data.names)))
```

trimNA 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)
```

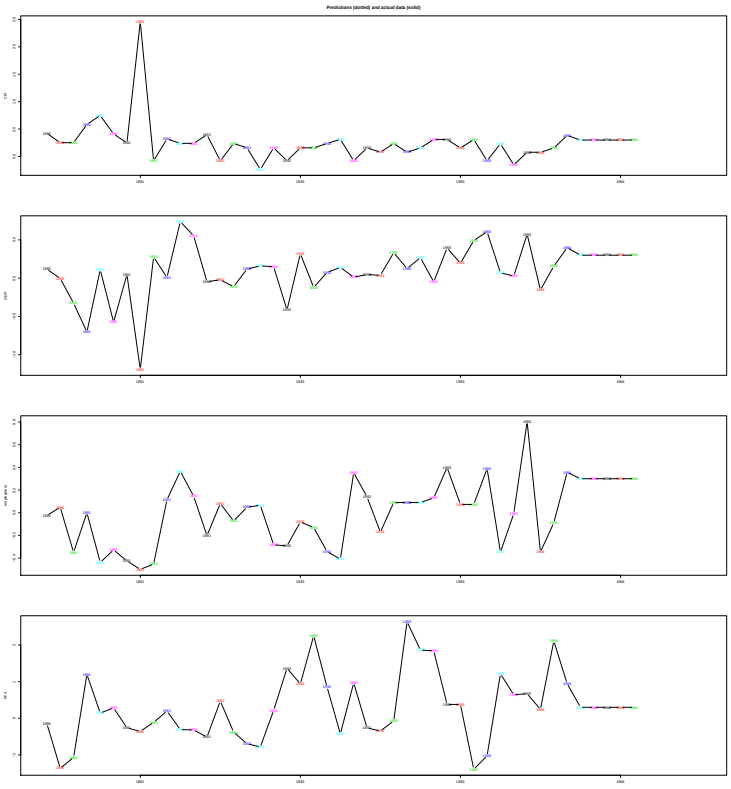
The input data can also be specified as a separate argument. For example, the same result will be achieved with

```
> z <- forecast(TSmodel(eg4.DSE.model), trimNA(new.data), conditioning.inputs = inputData(ne
```

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), warn = FALSE)
```

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.

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).

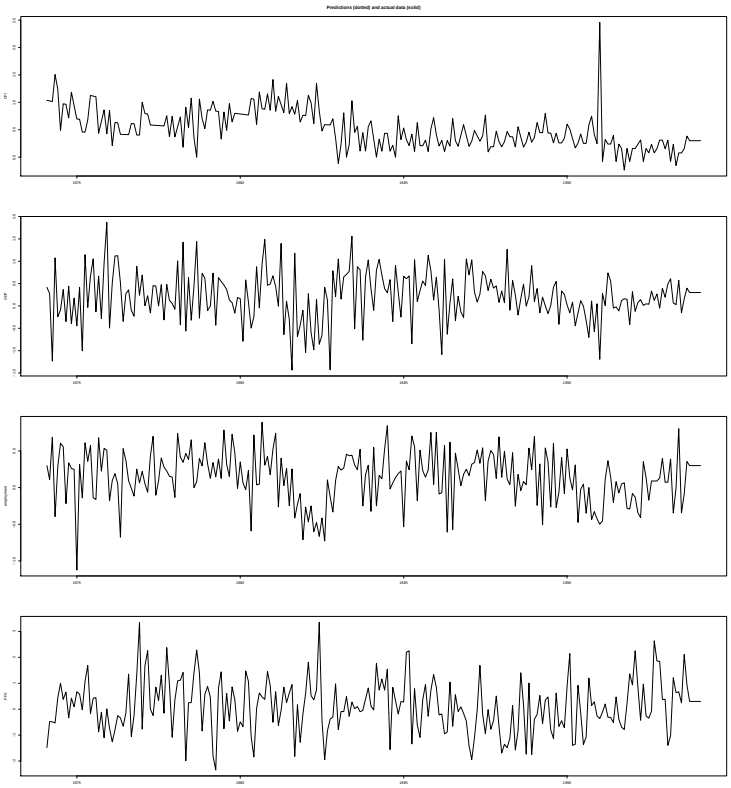
```
> z <- featherForecasts(TSmodel(eg4.DSE.model), new.data)
> tfplot(z)
```

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)
```

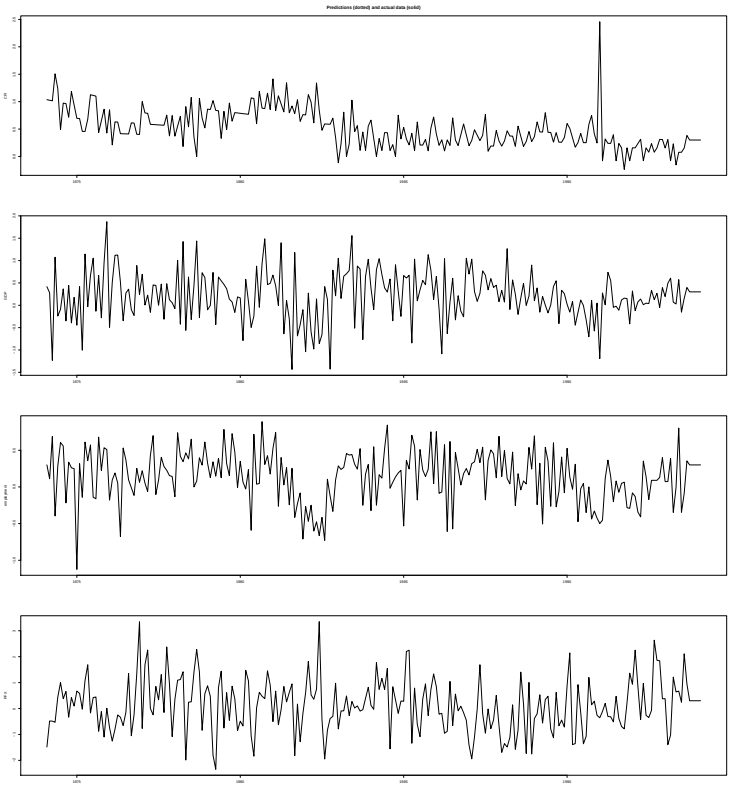


The second method, *horizonForecasts*, produces forecasts from every period for specified horizons.

```
> 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.

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 = "estVARXls", estimation.args = list(max.lag = 2),
      criterion = "TSmodel")
```

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 *estVARXls* 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.

An optional argument *rng* can be specified here and in 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 by comments in the examples 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

```
> setRNG(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. Typically the seed should be set only when trying to reproduce previous results.

The following uses mod2 as the true model.

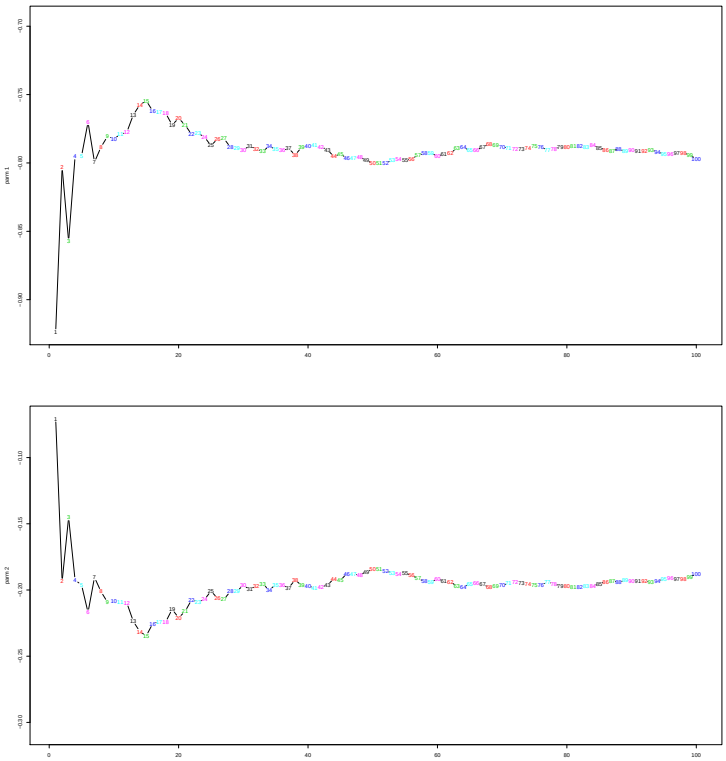
```
> e.ls.mod2 <- EstEval(mod2, replications = 100, simulation.args = list(sampleT = 100,
      sd = 1), estimation = "estVARXls", estimation.args = list(max.lag = 2),
      criterion = "TSmodel")
```

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))
```

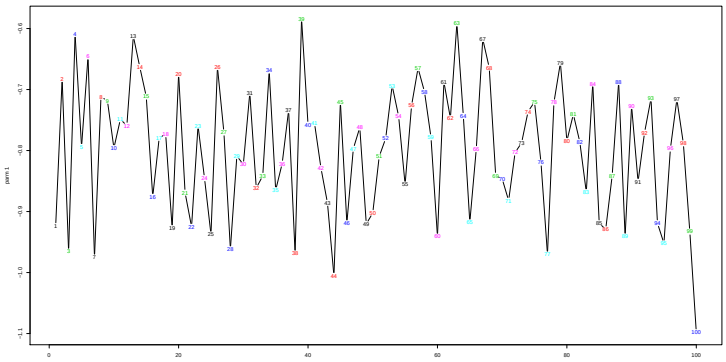


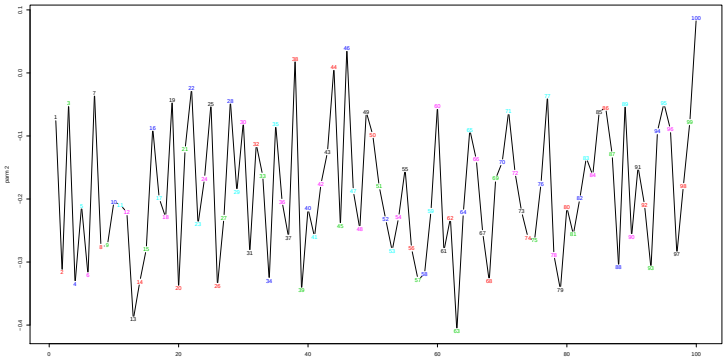
The straight line indicates the true value. To plot a line chart of the estimated parameters use coef to extract the parameters from the TSmodel:

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

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

```
> tfplot(coef(e.ls.mod2), cumulate = FALSE, bounds = FALSE)
```



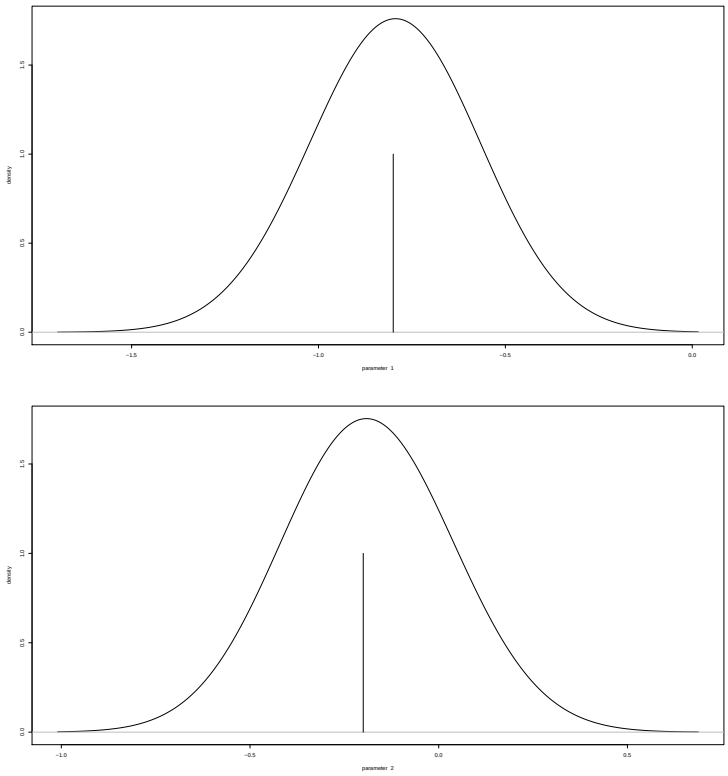


To plot the distribution of estimates:

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

The plot from mod2 looks like this:

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

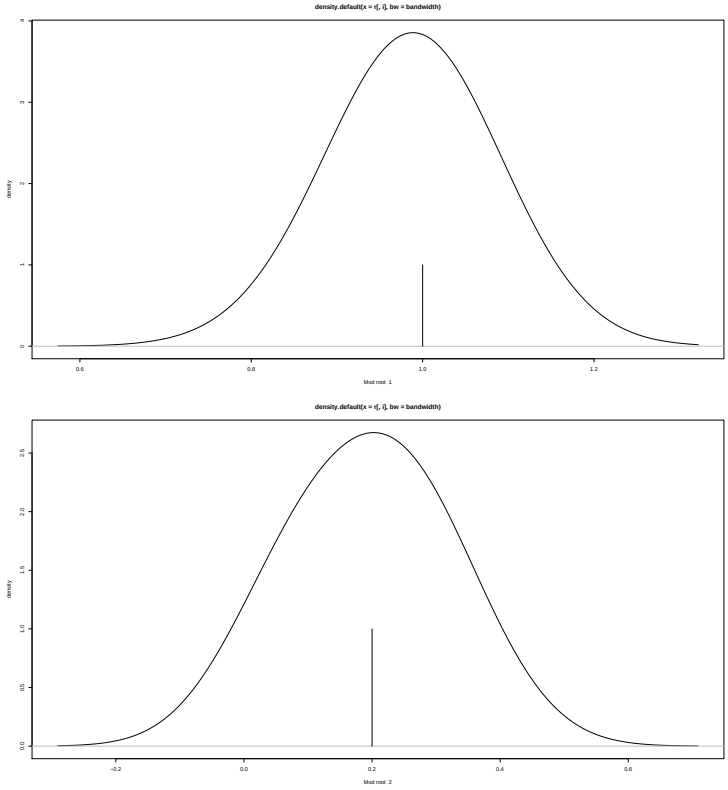


To plot the roots of the estimated model use roots to extract the roots from the TSmodel:

```
> 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 *estVARXls* 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 estBlackBox 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(estVARXls = list(max.la
      est.replications = 2, pred.replications = 10)
```

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 covari-

ance 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(estVARXls = list(max
      est.replications = 2, pred.replications = 10)
```

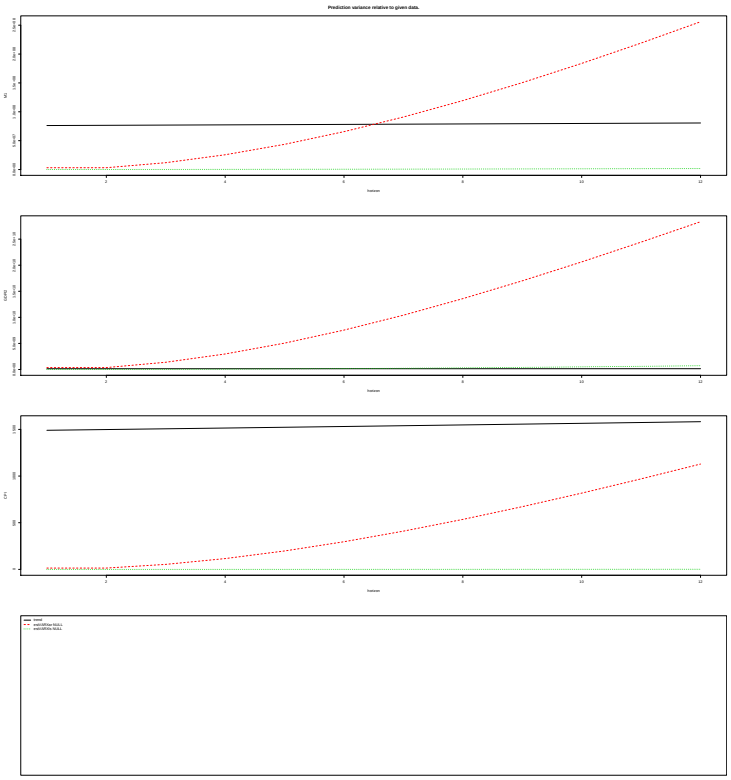
The reduction procedure used is MittnikReducedModels.. An optional argument criteria can be specified. This controls the model selection criteria used by the reduction technique.

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 <- outOfSample.forecastCovEstimatorsWRTdata(trimNA(eg1.DSE.data),
      estimation.sample = 0.5, estimation.methods = list(estVARXar = NULL,
          estVARXls = 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 *estBlackBox4*) and printing of intermediate results is suppressed.

```
> z <- outOfSample.forecastCovEstimatorsWRTdata(trimNA(eg1.DSE.data),
      estimation.sample = 0.5, estimation.methods = list(estBlackBox4 = list(max.lag = 3,
          verbose = F), estVARXls = list(max.lag = 3)), trend = T,
      zero = T)
> tfplot(z)
```

The object returned by outOfSample.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 estBlackBox4 is the first model and that estimated with estVARXls 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 estBlackBox4 (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.

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 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, checkConsistentDimensions.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 envi-

ronments. 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.