```
nonlin decay tight
compute decay=.25,tight=.05
declare real sse
*
find(trace) min sse
   *
   * Reject out-of-range values. (Set the value to NA and skip the
   * remaining calculation.)
   *
   if decay<0.0.or.tight<=0.0 {
      compute sse=%na
      next
   }
   @ForecastLoss(from=fstart,to=fend,steps=nsteps,$
      tight=tight,decay=decay,seed=21439) sse
end find
```

The optimized value for TIGHT ends up being quite small, indicating that for this series, the best forecasting autoregression is basically a random walk.

```
1.  DECAY                0.2247858542
2.  TIGHT                0.0026886316
```

6.6 VAR with Informative Prior (BVAR)

When applied to a univariate autoregression, the standard deviations in the Minnesota prior take the simple form:

$$\gamma l^{-d}$$

There are just two hyperparameters, and the prior isn't affected by the scale of the variable. A VAR will be more complicated. We need to adjust for the scales of the variables; if, for instance, interest rates are in percentages, the coefficients will be a factor of 100 smaller than if they're in decimals. We also need to incorporate into the prior the notion that "own" lags of the typical variable have most of the explanatory power.

To adjust for scale, the Minnesota prior starts with an (OLS) univariate autoregression (including any deterministic components) on each of the dependent variables. The standard errors of estimate of these regressions (called s_i for equation i) are used to rescale the standard deviation of the prior for a coefficient on a lag of variable j in equation i by s_i/s_j. While somewhat arbitrary, this has stood the test of time. It handles issues of scale correctly (you'll get exactly the same results if you rescale variables), it doesn't add any extra hyperparameters to the structure and it seems to work fine in practice. The univariate autoregressions are used rather than the OLS VAR because it's quite possible that an unrestricted VAR may have almost no (or even negative) degrees of freedom if there are many endogenous variables.

In the RATS manual, the standard error function takes the general form:

$$S(i,j,l) = \frac{\gamma l^{-d} f(i,j) s_i}{s_j}$$

$f(i,j)$ is the relative tightness of variable j in equation i. This could be a general $M \times M$ function, but the standard here is to make

$$f(i,i) = 1; f(i,j) = w \text{ if } i \neq j$$

where w is a new hyperparameter. The closer this is to zero, the more the lags of "other" variables are shut down. (The means of the priors on all these other variables are zero).

We now have a SUR model with an informative prior. Because the underlying model is linear, we can do a single cross product matrix to get all the statistics required to compute the likelihood. There's one obstacle standing in the way of working with the model using standard sampling techniques: the possibility of the size overwhelming the capacity of the computer.

With a flat prior, the VAR is easy to handle because the (potentially) huge covariance matrix for the coefficients factors into smaller parts:

$$\Sigma \otimes \left(\sum x_t' x_t \right)^{-1}$$

The number of calculations in a matrix inversion or Choleski factorization go up with the cube of the matrix size. Reducing an $Mk \times Mk$ inversion to $M \times M$ and $k \times k$ reduces the number of calculations by roughly M^3 (assuming $k \gg M$). For a six variable model, that's 216. Although the precision matrix coming from the data is still the very nicely formed:

$$\hat{\mathbf{H}} = \mathbf{H} \otimes \sum x_t' x_t$$

the precision matrix from the prior (**H**) is a general diagonal matrix and we need to compute

$$\left(\hat{\mathbf{H}} + \mathbf{H} \right)^{-1} \tag{6.17}$$

There is no simple way to compute that other than taking the full inverse. In a small enough system, that's certainly not unreasonable, but it *is* for the much larger VAR's that are often used in macroeconomic forecasting.

An alternative is to treat **H** as a diagonal matrix. Since the priors are independent across equations, this makes (6.17) a matrix that's non-zero only on the blocks on the diagonal, so it can be handled equation by equation. We wouldn't expect that to produce that much of a difference compared with doing the whole system; after all, the maximizing point for the likelihood is OLS equation by equation and the priors are independent. If we're interested in the model solely for forecasting (not analyzing an SVAR), it's the lag coefficients that really matter.

Any of the standard forms of Minnesota prior can be implemented in RATS by adding the SPECIFY instruction to the system definition. When you estimate the system using **ESTIMATE**, it does the calculation of the s_i scale factors as described above. Note that this makes the prior somewhat sensitive to the data range over which you estimate the system. The point estimates are computed using what's known as *mixed estimation*. This is basically the natural conjugate prior done somewhat informally. The formal natural conjugate prior for a single equation would have the precision from the likelihood as

$$h\sum x_t'x_t$$

and the precision from the prior as

$$h\left(\mathbf{H}^*\right)$$

where $\mathbf{H}^*$ is the prior precision matrix adjusted for multiplication by h. In example 2-1, that adjustment was to divide by the prior mean for h. In mixed estimation, it's usually done by multiplying by the variance of an OLS estimate of the equation (same as dividing by the precision); in RATS, that multiplier is s_i^2. Note that the precision is the reciprocal of the standard deviation squared, so if we write the precision of variable j in equation i at lag l as

$$H(i,j,l)=\frac{l^{2d}s_j^2}{\gamma^2 f(i,j)^2 s_i^2}$$

then the $\mathbf{H}^*$ used for mixed estimation just cancels out the s_i^2 in the denominator.

Thirty years ago, when VAR's were first being introduced, there was no serious option for analyzing a BVAR more formally. Even the simple Monte Carlo integration with flat priors could take an hour for 1000 draws. Now, it's possible to handle medium-sized models with full systems estimation. Since the Minnesota prior is still widely used, it would be helpful to be able to strip the setup out of the **ESTIMATE** instruction. There are several procedures which can do that: **@BVARBuildPrior** and **@BVARBuildPriorMN**. **@BVARBuildPriorMN** calls **@BVARBuildPrior** many times to construct the prior. In our example, we build the prior with:

```
@BVARBuildPriorMN(model=varmodel,tight=.1,other=.5) $
  hbpriors hpriors
@BVARFinishPrior hbpriors hpriors bprior hprior
```

The **@BVARBuild** routines generate the precision and the precision x the mean. (That's the easiest form to generate). **@BVARFinishPrior** converts this information back into precision and mean. Note that **@BVARBuildPrior** can be used to add *dummy observation* elements to a prior.

The example program generates out-of-sample simulations. This allows a full accounting for all the uncertainly in forecasts: both the uncertainty regarding

the coefficients (handled by Gibbs sampling) and the shocks during the forecast period.

```
compute %modelsetcoeffs(varmodel,bdraw)
simulate(model=varmodel,cv=sigmad,results=simresults,steps=nstep)
do i=1,nvar
   set forecast(i)   fstart fend = forecast(i)+simresults(i)
   set forestderr(i) fstart fend = forestderr(i)+simresults(i)**2
end do i
```

The **SIMULATE** instruction does the forecasts with randomly drawn shocks. We can't really do this with the equation-at-a-time methods because this depends upon the draw for the covariance matrix. (We can get point forecasts, but not the "cloud"). The `forecast` and `forecastderrs` series accumulate the sum and sum of squares of the forecasts. Outside the loop, these are converted into the mean and standard errors by standard calculations:

```
do i=1,nvar
   set forecast(i) fstart fend     = forecast(i)/ndraws
   set forestderr(i) fstart fend = $
     sqrt(forestderr(i)/ndraws-forecast(i)**2)
end do i
```

6.7 RATS Tips and Tricks

The function %CLOCK

`%clock(draw,base)` is like a "mod" function, but gives values from 1 to `base`, so `%clock(draw,2)` will be 1 for odd values of `draw` and 2 for even values.

The %MODEL function family

A `MODEL` is an object which organizes a collection of linear equations or formulas. The family of functions with names beginning with `%MODEL` take information out of and put it into a **MODEL**. These are all included in the *Model* category of the *Functions* wizard.

The most important of these apply to the case where the model is a set of linear equations like a VAR. For those, you can get and reset the coefficients of all equations at one time:

%modelgetcoeffs(model) returns a matrix of coefficients, with each column giving the coefficients in one of the equations.

%modelsetcoeffs(model,b) resets the coefficients of all equations in the model. The input matrix `b` can either have the same form as the one returned by `%modelgetcoeffs` (one column per equation) or can be a single "vec'ed" coefficient vector, with equation stacked on top of equation.