Multivariate Autoregressive Modeling for the Environmental Sciences

E. E. Holmes, M. D. Scheuerell, and E. J. Ward

2022-02-21

Contents

Preface

This is material that was developed as part of a course we teach at the University of Washington on applied time series analysis for fisheries and environmental data. You can find our lectures on our course website ATSA.

Book package

The book uses a number of R packages and a variety of fisheries data sets. The packages and data sets can be installed by installing our **atsalibrary** package which is hosted on GitHub:

```
library(devtools)
devtools::install_github("atsa-es/atsalibrary")
```

Authors

The authors are United States federal research scientists. This work was conducted as part of our jobs at the Northwest Fisheries Science Center (NWFSC), a research center for NOAA Fisheries, and the United States Geological Survey, which are United States federal government agencies. E. Holmes and E. Ward are affiliate faculty and M. Scheuerell is an associate professor at the University of Washington.

Links to more code and publications can be found on our academic websites:

- http://faculty.washington.edu/eeholmes
- http://faculty.washington.edu/scheuerl
- http://faculty.washington.edu/warde

Citation

Holmes, E. E., M. D. Scheuerell, and E. J. Ward. Multivariate Autoregressive Modeling for the Environmental Sciences. NOAA Fisheries, Northwest Fisheries Science Center, 2725 Montlake Blvd E., Seattle, WA 98112. Contacts eeholmes@uw.edu, eward@uw.edu, and scheuerl@uw.edu.

Chapter 1

Template

Start with one paragraph overview.

A script with all the R code in the chapter can be downloaded here. The Rmd for this chapter can be downloaded here.

Data and packages

All the data used in the chapter are in the **MARSS** package. Install the package, if needed, and load to run the code in the chapter.

```
library(MARSS)
```

1.1 Introduction

Introdution. Here's how to do an equation with numbering.

$$\begin{aligned} \mathbf{y}_t &= \mathbf{Z}\mathbf{x}_t + \mathbf{a} + \mathbf{v}_t \text{ where } \mathbf{v}_t \sim \text{MVN}(0, \mathbf{R}) \\ \mathbf{x}_t &= \mathbf{x}_{t-1} + \mathbf{w}_t \text{ where } \mathbf{w}_t \sim \text{MVN}(0, \mathbf{Q}) \end{aligned} \tag{1.1}$$

1.2 New section

New section. Example of an equation with matrix.

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix}_t = \begin{bmatrix} z_{11} & z_{12} & z_{13} \\ z_{21} & z_{22} & z_{23} \\ z_{31} & z_{32} & z_{33} \\ z_{41} & z_{42} & z_{43} \\ z_{51} & z_{52} & z_{53} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}_t + \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \end{bmatrix} + \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{bmatrix}_t . \tag{1.2}$$

and the process model would look like

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}_t = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}_{t-1} + \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}_t \tag{1.3}$$

The observation errors would be

$$\begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{bmatrix}_t \sim \text{MVN}\left(\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} & r_{15} \\ r_{12} & r_{22} & r_{23} & r_{24} & r_{25} \\ r_{13} & r_{23} & r_{33} & r_{34} & r_{35} \\ r_{14} & r_{24} & r_{34} & r_{44} & r_{45} \\ r_{15} & r_{25} & r_{35} & r_{45} & r_{55} \end{bmatrix} \right) \tag{1.4}$$

And the process errors would be

$$\begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}_t \sim \text{MVN}\left(\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} q_{11} & q_{12} & q_{13} \\ q_{12} & q_{22} & q_{23} \\ q_{13} & q_{23} & q_{33} \end{bmatrix} \right) . \tag{1.5}$$

1.3 Section 3

1.4 Discussion

For your homework this week, we will continue to investigate common trends in the Lake Washington plankton data.

1. Fit other DFA models to the phytoplankton data with varying numbers of trends from 1-4 (we fit a 3-trend model above). Do not include any covariates in these models. Using `R="diagonal and unequal"` for the observation errors, which of the DFA models has the most support from the data?

   Plot the model states and loadings as in Section 1.3. Describe the general patterns in the states and the ways the different taxa load onto those trends.

2. How does the best model from Question 1 compare to a DFA model with the same number of trends, but with `R="unconstrained"`?

   Plot the model states and loadings as in Section 1.2. Describe the general patterns in the states and the ways the different taxa load onto those trends.

   Also plot the the model fits as in Section 1.3. Do they reasonable? Are there any particular problems or outliers?

3. Fit a DFA model that includes temperature as a covariate and 3 trends (as in Section **??**), but with`R="unconstrained"`? How does this model compare to the model with `R="diagonal and unequal"`? How does it compare to the model in Question 2?

   Plot the model states and loadings as in Section 1.2. Describe the general patterns in the states and the ways the different taxa load onto those trends.

Chapter 2

Univariate state-space models

This chapter will show you how to fit some basic univariate state-space models using the {MARSS} package, the `StructTS()` function, and JAGS and Stan code. This chapter will also introduce you to the idea of writing AR(1) models in state-space form.

A script with all the R code in the chapter can be downloaded here. The Rmd for this chapter can be downloaded here.

Data and packages

All the data used in the chapter are in the {MARSS} package. The other required packages are {stats} (normally loaded by default when starting R), {datasets} and {forecast}. Install the packages, if needed, and load:

```
library(stats)
library(MARSS)
library(forecast)
library(datasets)
```

To run the JAGS code example (optional), you will also need JAGS installed and the {R2jags}, {rjags} and {coda} R packages. To run the Stan code example (optional), you will need the {rstan} package.

2.1 Fitting with {MARSS}

The {MARSS} package fits multivariate auto-regressive models of this form:

$$\begin{aligned} \mathbf{x}_t &= \mathbf{B}\mathbf{x}_{t-1} + \mathbf{u} + \mathbf{w}_t \text{ where } \mathbf{w}_t \sim \text{N}(0, \mathbf{Q}) \\ \mathbf{y}_t &= \mathbf{Z}\mathbf{x}_t + \mathbf{a} + \mathbf{v}_t \text{ where } \mathbf{v}_t \sim \text{N}(0, \mathbf{R}) \\ \mathbf{x}_0 &= \mu \end{aligned} \tag{2.1}$$

To fit your time series model with the {MARSS} package, you need to put your model into the form above. The $\mathbf{B}$, $\mathbf{Z}$, $\mathbf{u}$, $\mathbf{a}$, $\mathbf{Q}$, $\mathbf{R}$ and μ are parameters that are (potentially) estimated. The $\mathbf{y}$ are your data. The $\mathbf{x}$ are the hidden state(s). Everything in bold is a matrix; if it is a small bolded letter, it is a matrix with 1 column.

Important: In the state-space model equation, $\mathbf{y}$ *is always the data and* $\mathbf{x}$ *is a hidden random walk estimated from the data.*

A basic `MARSS()` call looks like `fit=MARSS(y, model=list(...))`. The argument `model` tells the function what form the parameters take. The list has the elements with the names: `B U Q` etc. The names correspond to the parameters with the same names in Equation (2.1) except that μ is called `x0`. `tinitx` indicates whether the initial $\mathbf{x}$ is specified at $t = 0$ so $\mathbf{x}_0$ or $t = 1$ so $\mathbf{x}_1$.

Here's an example. Let's say we want to fit a univariate AR(1) model observed with error. Here is that model:

$$\begin{aligned} x_t &= bx_{t-1} + w_t \text{ where } \mathbf{w}_t \sim \text{N}(0, q) \\ y_t &= x_t + v_t \text{ where } v_t \sim \text{N}(0, r) \\ x_0 &= \mu \end{aligned} \tag{2.2}$$

To fit this with `MARSS()`, we need to write Equation (2.2) as Equation (2.1). Equation (2.1) is in matric form. In the model list, the parameters must be written exactly like they would be written for Equation (2.1). For example, `1` is the number 1 in R. It is not a matrix:

```
class(1)
```

```
[1] "numeric"
```

If you need a 1 (or 0) in your model, you need to pass in the parameter as a 1×1 matrix: `matrix(1)`.

With that mind, our model list for Equation (2.2) is:

```
mod.list <- list(B = matrix(1), U = matrix(0), Q = matrix("q"),
    Z = matrix(1), A = matrix(0), R = matrix("r"), x0 = matrix("mu"),
    tinitx = 0)
```

We can simulate some AR(1) plus error data like so

```
q <- 0.1
r <- 0.1
n <- 100
y <- cumsum(rnorm(n, 0, sqrt(q))) + rnorm(n, 0, sqrt(r))
```

And then fit with `MARSS()` using `mod.list` above:

```
fit <- MARSS(y, model = mod.list)
```

```
Success! abstol and log-log tests passed at 16 iterations.
Alert: conv.test.slope.tol is 0.5.
Test with smaller values (<0.1) to ensure convergence.

MARSS fit is
Estimation method: kem
Convergence test: conv.test.slope.tol = 0.5, abstol = 0.001
Estimation converged in 16 iterations.
Log-likelihood: -65.70444
AIC: 137.4089   AICc: 137.6589

      Estimate
R.r     0.1066
Q.q     0.0578
x0.mu  -0.2024
Initial states (x0) defined at t=0

Standard errors have not been calculated.
Use MARSSparamCIs to compute CIs and bias estimates.
```

If we wanted to fix $q = 0.1$, then $\mathbf{Q} = [0.1]$ (a 1×1 matrix with 0.1). We change `mod.list$Q` and re-fit:

```
mod.list$Q <- matrix(0.1)
fit <- MARSS(y, model = mod.list)
```

2.2 Examples using the Nile river data

We will use the data from the Nile River (Figure **??**). We will fit different flow models to the data and compare the models with AIC.

```
library(datasets)
dat <- as.vector(Nile)
```

2.2.1 Flat level model

We will start by modeling these data as a simple average river flow with variability around some level μ.

$$y_t = \mu + v_t \text{ where } v_t \sim \text{N}(0, r) \tag{2.3}$$

where y_t is the river flow volume at year t.

We can write this model as a univariate state-space model as follows. We use x_t to model the average flow level. y_t is just an observation of this flat x_t. Work through x_1, x_2, ... starting from x_0 to convince yourself that x_t will always equal μ.

$$\begin{gathered} x_t = 1 \times x_{t-1} + 0 + w_t \text{ where } w_t \sim \text{N}(0, 0) \\ y_t = 1 \times x_t + 0 + v_t \text{ where } v_t \sim \text{N}(0, r) \\ x_0 = \mu \end{gathered} \tag{2.4}$$

The model is specified as a list as follows:

```
mod.nile.0 <- list(B = matrix(1), U = matrix(0), Q = matrix(0),
    Z = matrix(1), A = matrix(0), R = matrix("r"), x0 = matrix("mu"),
    tinitx = 0)
```

We then fit the model:

```
kem.0 <- MARSS(dat, model = mod.nile.0)
```

Output not shown, but here are the estimates and AICc.

```
c(coef(kem.0, type = "vector"), LL = kem.0$logLik, AICc = kem.0$AICc)
```

```
       R.r      x0.mu         LL       AICc
28351.5675   919.3500  -654.5157  1313.1552
```

2.2.2 Linear trend in flow model

Figure **??** shows the fit for the flat average river flow model. Looking at the data, we might expect that a declining average river flow would be better. In MARSS form, that model would be:

$$\begin{aligned} x_t &= 1 \times x_{t-1} + u + w_t \text{ where } w_t \sim \text{N}(0,0) \\ y_t &= 1 \times x_t + 0 + v_t \text{ where } v_t \sim \text{N}(0,r) \\ x_0 &= \mu \end{aligned} \tag{2.5}$$

where u is now the average per-year decline in river flow volume. The model is specified as follows:

```
mod.nile.1 <- list(B = matrix(1), U = matrix("u"), Q = matrix(0),
    Z = matrix(1), A = matrix(0), R = matrix("r"), x0 = matrix("mu"),
    tinitx = 0)
```

We then fit the model:

```
kem.1 <- MARSS(dat, model = mod.nile.1)
```

Here are the estimates, log-likelihood and AICc:

```
c(coef(kem.1, type = "vector"), LL = kem.1$logLik, AICc = kem.1$AICc)
```

```
         R.r          U.u        x0.mu           LL         AICc
22213.595453    -2.692106  1054.935067  -642.315910  1290.881821
```

Figure **??** shows the fits for the two models with deterministic models (flat and declining) for mean river flow along with their AICc values (smaller AICc is better). The AICc for the model with a declining river flow is lower by over 20 (which is a lot).

2.2.3 Stochastic level model

Looking at the flow levels, we might suspect that a model that allows the average flow to change would model the data better and we might suspect that there have been sudden, and anomalous, changes in the river flow level. We will now model the average river flow at year t as a random walk, specifically an autoregressive process which means that average river flow is year t is a function of average river flow in year $t-1$.

$$\begin{aligned} x_t &= x_{t-1} + w_t \text{ where } w_t \sim \text{N}(0, q) \\ y_t &= x_t + v_t \text{ where } v_t \sim \text{N}(0, r) \\ x_0 &= \mu \end{aligned} \tag{2.6}$$

As before, y_t is the river flow volume at year t. x_t is the mean level. The model is specified as:

```
mod.nile.2 = list(B = matrix(1), U = matrix(0), Q = matrix("q"),
    Z = matrix(1), A = matrix(0), R = matrix("r"), x0 = matrix("mu"),
    tinitx = 0)
```

We could also use the text shortcuts to specify the model. Because $\mathbf{R}$ and $\mathbf{Q}$ are 1×1 matrices, "unconstrained", "diagonal and unequal", "diagonal and

equal" and "equalvarcov" will all lead to a 1×1 matrix with one estimated element. For **a** and **u**, the following shortcut could be used:

```
A <- "zero"
U <- "zero"
```

Because $\mathbf{x}_0$ is 1×1, it could be specified as "unequal", "equal" or "unconstrained".

```
kem.2 <- MARSS(dat, model = mod.nile.2)
```

Here are the estimates, log-likelihood and AICc:

```
c(coef(kem.2, type = "vector"), LL = kem.2$logLik, AICc = kem.2$AICc)
```

```
       R.r         Q.q       x0.mu          LL        AICc
15065.6121  1425.0030  1111.6338  -637.7631  1281.7762
```

2.2.4 Stochastic level model with drift

We can add a drift to term to our random walk; the u in the process model (x) is the drift term. This causes the random walk to tend to trend up or down.

$$\begin{aligned} x_t &= x_{t-1} + u + w_t \text{ where } w_t \sim \text{N}(0, q) \\ y_t &= x_t + v_t \text{ where } v_t \sim \text{N}(0, r) \\ x_0 &= \mu \end{aligned} \tag{2.7}$$

The model is then specified by changing `U` to indicate that a u is estimated:

```
mod.nile.3 = list(B = matrix(1), U = matrix("u"), Q = matrix("q"),
    Z = matrix(1), A = matrix(0), R = matrix("r"), x0 = matrix("mu"),
    tinitx = 0)
```

```
kem.3 <- MARSS(dat, model = mod.nile.3)
```

Here are the estimates, log-likelihood and AICc:

```
c(coef(kem.3, type = "vector"), LL = kem.3$logLik, AICc = kem.3$AICc)
```

```
         R.r          U.u          Q.q        x0.mu           LL         AICc
15585.278194    -3.248793  1088.987455  1124.044484  -637.302692  1283.026436
```

Figure ?? shows all the models along with their AICc values.

2.3 The StructTS function

The `StructTS` function in the {stats} package in R will also fit the stochastic level model:

```
fit.sts <- StructTS(dat, type = "level")
fit.sts
```

```
Call:
StructTS(x = dat, type = "level")

Variances:
  level  epsilon
   1469    15099
```

The estimates from `StructTS()` will be different (though similar) from `MARSS()` because `StructTS()` uses $x_1 = y_1$, that is the hidden state at $t = 1$ is fixed to be the data at $t = 1$. That is fine if you have a long data set, but would be disastrous for the short data sets typical in fisheries and ecology.

`StructTS()` is much, much faster for long time series. The example in `?StructTS` is pretty much instantaneous with `StructTS()` but takes minutes with the EM algorithm that is the default in `MARSS()`. With the BFGS algorithm, it is much closer to `StructTS()`:

```
trees <- window(treering, start = 0)
fitts <- StructTS(trees, type = "level")
fitem <- MARSS(as.vector(trees), mod.nile.2)
fitbf <- MARSS(as.vector(trees), mod.nile.2, method = "BFGS")
```

Note that `mod.nile.2` specifies a univariate stochastic level model so we can use it just fine with other univariate data sets.

In addition, `fitted(fit.sts)` where `fit.sts` is a fit from `StructTS()` is very different than `fit.marss$states` from `MARSS()`.

```
t = 10
fitted(fit.sts)[t]
```

```
[1] 1162.904
```

is the expected value of y_{t+1} (in this case y_{11} since we set $t = 10$) given the data up to y_t (in this case, up to y_{10}). It is called the one-step ahead prediction.
We are not going to use the one-step ahead predictions unless we are forecasting or doing cross-validation.

Typically, when we analyze fisheries and ecological data, we want to know the estimate of the state, the x_t, given ALL the data. For example, we might need an estimate of the population size in year 1990 given a time series of counts from 1930 to 2015. We don't want to use only the data up to 1989; we want to use all the information. `fit.marss$states` from `MARSS()` is the expected value of x_t given all the data. For the stochastic level model, that is equal to the expected value of y_t given all the data except y_t.

If you needed the one-step predictions from `MARSS()`, you can get them from the Kalman filter output:

```
kf = print(kem.2, what = "kfs")
kf$xtt1[1, t]
```

Passing in `what="kfs"` returns the Kalman filter/smoother output. The expected value of x_t conditioned on y_1 to y_{t-1} is in `kf$xtt1`. The expected value of x_t conditioned on all the data is in `kf$xtT`.

```
Loading required package: lattice

Loading required package: survival

Loading required package: Formula

Attaching package: 'Hmisc'

The following objects are masked from 'package:dplyr':

    src, summarize

The following objects are masked from 'package:base':

    format.pval, units
```

2.4 Comparing models with AIC and model weights

To get the AIC or AICc values for a model fit from a MARSS fit, use `fit$AIC` or `fit$AICc`. The log-likelihood is in `fit$logLik` and the number of estimated parameters in `fit$num.params`. For fits from other functions, try `AIC(fit)` or look at the function documentation.

Let's put the AICc values 3 Nile models together:

```
nile.aic = c(kem.0$AICc, kem.1$AICc, kem.2$AICc, kem.3$AICc)
```

Then we calculate the AICc minus the minus AICc in our model set and compute the model weights. ΔAIC is the AIC values minus the minimum AIC value in your model set.

```
delAIC <- nile.aic - min(nile.aic)
relLik <- exp(-0.5 * delAIC)
aicweight <- relLik/sum(relLik)
```

And this leads to our model weights table:

```
aic.table <- data.frame(AICc = nile.aic, delAIC = delAIC, relLik = relLik,
    weight = aicweight)
rownames(aic.table) <- c("flat level", "linear trend", "stoc level",
    "stoc level w drift")
```

Here the table is printed using `round()` to limit the number of digits shown.

```
round(aic.table, digits = 3)
```

```
                     AICc delAIC relLik weight
flat level         1313.155 31.379  0.000  0.000
linear trend       1290.882  9.106  0.011  0.007
stoc level         1281.776  0.000  1.000  0.647
stoc level w drift 1283.026  1.250  0.535  0.346
```

One thing to keep in mind when comparing models within a set of models is that the model set needs to include at least one model that can fit the data reasonably well. Reasonably well' means the model can put a fitted line through the data. Can't all models do that? Definitely, not. For example, the flat-level model cannot put a fitted line through the Nile River data. It is simply impossible. The straight trend model also cannot put a fitted line through the flow data. So if our model set only included flat-level and straight trend, then we might have said that the straight trend model isbest' even though it is just the better of two bad models.

2.5 Basic diagnostics

The first diagnostic that you do with any statistical analysis is check that your residuals correspond to your assumed error structure. We have two types of errors in a univariate state-space model: process errors, the w_t, and observation errors, the v_t.
They should not have a temporal trend. To get the residuals from most

types of fits in R, you can use `residuals(fit)`. `MARSS()` calls the v_t, "model residuals", and the w_t "state residuals". We can plot these using the following code (Figure **??**).

```
par(mfrow = c(1, 2))
resids <- residuals(kem.0)
plot(resids$model.residuals[1, ], ylab = "model residual", xlab = "",
    main = "flat level")
```

```
Warning in min(x): no non-missing arguments to min; returning Inf
```

```
Warning in max(x): no non-missing arguments to max; returning -Inf
```

```
Warning in min(x): no non-missing arguments to min; returning Inf
```

```
Warning in max(x): no non-missing arguments to max; returning -Inf
```

```
Error in plot.window(...): need finite 'xlim' values
```

```
abline(h = 0)
plot(resids$state.residuals[1, ], ylab = "state residual", xlab = "",
    main = "flat level")
```

```
Warning in min(x): no non-missing arguments to min; returning Inf
```

```
Warning in min(x): no non-missing arguments to max; returning -Inf
```

```
Warning in min(x): no non-missing arguments to min; returning Inf
```

```
Warning in max(x): no non-missing arguments to max; returning -Inf
```

```
Error in plot.window(...): need finite 'xlim' values
```

```r
abline(h = 0)
```

```
Warning in min(x): no non-missing arguments to min; returning Inf

Warning in max(x): no non-missing arguments to max; returning -Inf

Warning in min(x): no non-missing arguments to min; returning Inf

Warning in max(x): no non-missing arguments to max; returning -Inf

Error in plot.window(...): need finite 'xlim' values

Warning in min(x): no non-missing arguments to min; returning Inf

Warning in min(x): no non-missing arguments to max; returning -Inf

Warning in min(x): no non-missing arguments to min; returning Inf

Warning in max(x): no non-missing arguments to max; returning -Inf

Error in plot.window(...): need finite 'xlim' values

Warning in min(x): no non-missing arguments to min; returning Inf

Warning in min(x): no non-missing arguments to max; returning -Inf

Warning in min(x): no non-missing arguments to min; returning Inf

Warning in max(x): no non-missing arguments to max; returning -Inf

Error in plot.window(...): need finite 'xlim' values

Warning in min(x): no non-missing arguments to min; returning Inf

Warning in min(x): no non-missing arguments to max; returning -Inf
```

```
Error in plot.window(...): need finite 'xlim' values

Warning in min(x): no non-missing arguments to min; returning Inf

Warning in min(x): no non-missing arguments to max; returning -Inf

Warning in min(x): no non-missing arguments to min; returning Inf

Warning in max(x): no non-missing arguments to max; returning -Inf

Error in plot.window(...): need finite 'xlim' values

Warning in min(x): no non-missing arguments to min; returning Inf

Warning in min(x): no non-missing arguments to max; returning -Inf

Warning in min(x): no non-missing arguments to min; returning Inf

Warning in max(x): no non-missing arguments to max; returning -Inf

Error in plot.window(...): need finite 'xlim' values
```

The residuals should also not be autocorrelated in time. We can check the autocorrelation with the function `acf()`. We won't do this for the state residuals for the flat level or linear trends since for those models $w_t = 0$. The autocorrelation plots are shown in Figure **??**. The stochastic level model looks the best in that its model residuals (the v_t) are fine but the state model still has problems. Clearly the state is not a simple random walk. This is not surprising. The Aswan Low Dam was completed in 1902 and changed the mean flow. The Aswan High Dam was completed in 1970 and also affected the flow. You can see these perturbations in Figure **??**.

```
par(mfrow = c(2, 2))
resids <- residuals(kem.0)
acf(resids$model.residuals[1, ], main = "flat level v(t)")
```

```
Error in ts(x): 'ts' object must have one or more observations
```

```
resids <- residuals(kem.1)
acf(resids$model.residuals[1, ], main = "linear trend v(t)")
```

```
Error in ts(x): 'ts' object must have one or more observations
```

```
resids <- residuals(kem.2)
acf(resids$model.residuals[1, ], main = "stoc level v(t)")
```

```
Error in ts(x): 'ts' object must have one or more observations
```

```
acf(resids$state.residuals[1, ], main = "stoc level w(t)", na.action = na.pass)
```

```
Error in ts(x): 'ts' object must have one or more observations
```

```
Error in ts(x): 'ts' object must have one or more observations
```

```
Error in ts(x): 'ts' object must have one or more observations
```

```
Error in ts(x): 'ts' object must have one or more observations
```

```
Error in ts(x): 'ts' object must have one or more observations
```

2.6 Fitting with JAGS

Here we show how to fit the stochastic level model, model 3 Equation (2.7), with JAGS. This is a model where the level is a random walk with drift and the Nile River flow is that level plus error.

```
library(datasets)
y <- as.vector(Nile)
```

This section requires that you have JAGS installed and the {R2jags}, {rjags} and {coda} R packages loaded.

```
library(R2jags)
library(rjags)
library(coda)
```

The first step is to write the model for JAGS to a file (filename in `model.loc`):

```
model.loc <- "ss_model.txt"
jagsscript <- cat("
   model {
   # priors on parameters
   mu ~ dnorm(Y1, 1/(Y1*100)); # normal mean = 0, sd = 1/sqrt(0.01)
   tau.q ~ dgamma(0.001,0.001); # This is inverse gamma
   sd.q <- 1/sqrt(tau.q); # sd is treated as derived parameter
   tau.r ~ dgamma(0.001,0.001); # This is inverse gamma
   sd.r <- 1/sqrt(tau.r); # sd is treated as derived parameter
   u ~ dnorm(0, 0.01);

   # Because init X is specified at t=0
   X0 <- mu
   X[1] ~ dnorm(X0+u,tau.q);
   Y[1] ~ dnorm(X[1], tau.r);

   for(i in 2:TT) {
   predX[i] <- X[i-1]+u;
```

```
   X[i] ~ dnorm(predX[i],tau.q); # Process variation
   Y[i] ~ dnorm(X[i], tau.r); # Observation variation
   }
   }
   ",
    file = model.loc)
```

Next we specify the data (and any other input) that the JAGS code needs. In this case, we need to pass in `dat` and the number of time steps since that is used in the for loop. We also specify the parameters that we want to monitor. We need to specify at least one, but we will monitor all of them so we can plot them after fitting. Note, that the hidden state is a parameter in the Bayesian context (but not in the maximum likelihood context).

```
jags.data <- list(Y = y, TT = length(y), Y1 = y[1])
jags.params <- c("sd.q", "sd.r", "X", "mu", "u")
```

Now we can fit the model:

```
mod_ss <- jags(jags.data, parameters.to.save = jags.params, model.file = model.loc,
    n.chains = 3, n.burnin = 5000, n.thin = 1, n.iter = 10000,
    DIC = TRUE)
```

We can then show the posteriors along with the MLEs from MARSS on top (Figure **??**) using the code below.

```
attach.jags(mod_ss)
par(mfrow = c(2, 2))
hist(mu)
abline(v = coef(kem.3)$x0, col = "red")
hist(u)
abline(v = coef(kem.3)$U, col = "red")
hist(log(sd.q^2))
abline(v = log(coef(kem.3)$Q), col = "red")
hist(log(sd.r^2))
abline(v = log(coef(kem.3)$R), col = "red")
```

```
detach.jags()
```

To plot the estimated states (Figure **??**), we write a helper function:

```
plotModelOutput <- function(jagsmodel, Y) {
    attach.jags(jagsmodel)
    x <- seq(1, length(Y))
    XPred <- cbind(apply(X, 2, quantile, 0.025), apply(X, 2,
        mean), apply(X, 2, quantile, 0.975))
    ylims <- c(min(c(Y, XPred), na.rm = TRUE), max(c(Y, XPred),
        na.rm = TRUE))
    plot(Y, col = "white", ylim = ylims, xlab = "", ylab = "State predictions"
    polygon(c(x, rev(x)), c(XPred[, 1], rev(XPred[, 3])), col = "grey70",
        border = NA)
    lines(XPred[, 2])
    points(Y)
}
```

```
plotModelOutput(mod_ss, y)
lines(kem.3$states[1, ], col = "red")
lines(1.96 * kem.3$states.se[1, ] + kem.3$states[1, ], col = "red",
    lty = 2)
lines(-1.96 * kem.3$states.se[1, ] + kem.3$states[1, ], col = "red",
    lty = 2)
title("State estimate and data from\nJAGS (black) versus MARSS (red)")
```

2.7 Fitting with Stan

Let's fit the same model with Stan using the {rstan} package. If you have not already, you will need to install the {rstan} package. This package depends on a number of other packages which should install automatically when you install {rstan}.

```
library(datasets)
library(rstan)
y <- as.vector(Nile)
```

First we write the model. We could write this to a file (recommended), but for this example, we write as a character object. Though the syntax is different from the JAGS code, it has many similarities. Note, unlike the JAGS, the Stan does **not allow** any NAs in your data. Thus we have to specify the location of the NAs in our data. The Nile data does not have NAs, but we want to write the code so it would work even if there were NAs.

```
scode <- "
data {
  int<lower=0> TT;
  int<lower=0> n_pos; // number of non-NA values
  int<lower=0> indx_pos[n_pos]; // index of the non-NA values
  vector[n_pos] y;
}
parameters {
  real x0;
  real u;
  vector[TT] pro_dev;
  real<lower=0> sd_q;
  real<lower=0> sd_r;
}
transformed parameters {
  vector[TT] x;
  x[1] = x0 + u + pro_dev[1];
  for(i in 2:TT) {
    x[i] = x[i-1] + u + pro_dev[i];
  }
}
model {
  x0 ~ normal(y[1],10);
  u ~ normal(0,2);
  sd_q ~ cauchy(0,5);
  sd_r ~ cauchy(0,5);
```

```
  pro_dev ~ normal(0, sd_q);
  for(i in 1:n_pos){
    y[i] ~ normal(x[indx_pos[i]], sd_r);
  }
}
generated quantities {
  vector[n_pos] log_lik;
  for (i in 1:n_pos) log_lik[i] = normal_lpdf(y[i] | x[indx_pos[i]], sd_r);
}
"
```

Then we call `stan()` and pass in the data, names of parameter we wish to have returned, and information on number of chains, samples (iter), and thinning. The output is verbose (hidden here) and may have some warnings.

```
# We pass in the non-NA ys as vector
ypos <- y[!is.na(y)]
n_pos <- sum(!is.na(y))  #number on non-NA ys
indx_pos <- which(!is.na(y))  #index on the non-NAs
mod <- rstan::stan(model_code = scode, data = list(y = ypos,
    TT = length(y), n_pos = n_pos, indx_pos = indx_pos), pars = c("sd_q",
    "x", "sd_r", "u", "x0"), chains = 3, iter = 1000, thin = 1)
```

We use `extract()` to extract the parameters from the fitted model. The estimated level is `x` and we will plot that with the 95% credible intervals.

```
pars <- rstan::extract(mod)
pred_mean <- apply(pars$x, 2, mean)
pred_lo <- apply(pars$x, 2, quantile, 0.025)
pred_hi <- apply(pars$x, 2, quantile, 0.975)
```

Here is a `ggplot()` version of the plot.

```
library(ggplot2)
nile <- data.frame(y = y, year = 1871:1970)
h <- ggplot(nile, aes(year))
```

```
h + geom_ribbon(aes(ymin = pred_lo, ymax = pred_hi), fill = "grey70") +
    geom_line(aes(y = pred_mean), size = 1) + geom_point(aes(y = y),
    color = "blue") + labs(y = "Nile River level")
```

We can plot the histogram of the samples against the values estimated via maximum likelihood.

```
par(mfrow = c(2, 2))
hist(pars$x0)
abline(v = coef(kem.3)$x0, col = "red")
hist(pars$u)
abline(v = coef(kem.3)$U, col = "red")
hist(log(pars$sd_q^2))
abline(v = log(coef(kem.3)$Q), col = "red")
hist(log(pars$sd_r^2))
abline(v = log(coef(kem.3)$R), col = "red")
```

2.8 A random walk model of animal movement

A simple random walk model of movement with drift (directional movement) but no correlation is

$$x_{1,t} = x_{1,t-1} + u_1 + w_{1,t}, \quad w_{1,t} \sim \mathrm{N}(0, \sigma_1^2) \tag{2.8}$$

$$x_{2,t} = x_{2,t-1} + u_2 + w_{2,t}, \quad w_{2,t} \sim \mathrm{N}(0, \sigma_2^2) \tag{2.9}$$

where $x_{1,t}$ is the location at time t along one axis (here, longitude) and $x_{2,t}$ is for another, generally orthogonal, axis (in here, latitude). The parameter u_1 is the rate of longitudinal movement and u_2 is the rate of latitudinal movement. We add errors to our observations of location:

$$y_{1,t} = x_{1,t} + v_{1,t}, \quad v_{1,t} \sim \mathrm{N}(0, \eta_1^2) \tag{2.10}$$

$$y_{2,t} = x_{2,t} + v_{2,t}, \quad v_{2,t} \sim \mathrm{N}(0, \eta_2^2), \tag{2.11}$$

This model is comprised of two separate univariate state-space models. Note that y_1 depends only on x_1 and y_2 depends only on x_2. There are no actual

interactions between these two univariate models. However, we can write the model down in the form of a multivariate model using diagonal variance-covariance matrices and a diagonal design ($\mathbf{Z}$) matrix. Because the variance-covariance matrices and $\mathbf{Z}$ are diagonal, the x_1:y_1 and x_2:y_2 processes will be independent as intended. Here are Equations (2.9) and (2.11) written as a MARSS model (in matrix form):

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \end{bmatrix} = \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \end{bmatrix} + \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} + \begin{bmatrix} w_{1,t} \\ w_{2,t} \end{bmatrix}, \ \mathbf{w}_t \sim \text{MVN}\left(0, \begin{bmatrix} \sigma_1^2 & 0 \\ 0 & \sigma_2^2 \end{bmatrix}\right) \tag{2.12}$$

$$\begin{bmatrix} y_{1,t} \\ y_{2,t} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_{1,t} \\ x_{2,t} \end{bmatrix} + \begin{bmatrix} v_{1,t} \\ v_{2,t} \end{bmatrix}, \ \mathbf{v}_t \sim \text{MVN}\left(0, \begin{bmatrix} \eta_1^2 & 0 \\ 0 & \eta_2^2 \end{bmatrix}\right) \tag{2.13}$$

The variance-covariance matrix for $\mathbf{w}_t$ is a diagonal matrix with unequal variances, σ_1^2 and σ_2^2. The variance-covariance matrix for $\mathbf{v}_t$ is a diagonal matrix with unequal variances, η_1^2 and η_2^2. We can write this succinctly as

$$\mathbf{x}_t = \mathbf{x}_{t-1} + \mathbf{u} + \mathbf{w}_t, \ \ \mathbf{w}_t \sim \text{MVN}(0, \mathbf{Q}) \tag{2.14}$$

$$\mathbf{y}_t = \mathbf{x}_t + \mathbf{v}_t, \ \ \mathbf{v}_t \sim \text{MVN}(0, \mathbf{R}). \tag{2.15}$$

2.9 Problems

1. Write the equations for each of these models: ARIMA(0,0,0), ARIMA(0,1,0), ARIMA(1,0,0), ARIMA(0,0,1), ARIMA(1,0,1). Read the help file for the `Arima()` function (in the **forecast** package) if you are fuzzy on the arima notation.

2. The **MARSS** package includes a data set of sharp-tailed grouse in Washington. Load the data to use as follows:

```
library(MARSS)
dat = log(grouse[, 2])
```

   Consider these two models for the data:

   - Model 1 random walk with no drift observed with no error
   - Model 2 random walk with drift observed with no error

   Written as a univariate state-space model, model 1 is

$$\begin{aligned} x_t &= x_{t-1} + w_t \text{ where } w_t \sim \text{N}(0, q) \\ x_0 &= a \\ y_t &= x_t \end{aligned} \tag{2.16}$$

   Model 2 is almost identical except with u added

$$\begin{aligned} x_t &= x_{t-1} + u + w_t \text{ where } w_t \sim \text{N}(0, q) \\ x_0 &= a \\ y_t &= x_t \end{aligned} \tag{2.17}$$

   y is the log grouse count in year t.

   a. Plot the data. The year is in column 1 of `grouse`.
   b. Fit each model using `MARSS()`.
   c. Which one appears better supported given AICc?

d. Load the **forecast** package. Use `?auto.arima` to learn what it does. Then use `auto.arima(dat)` to fit the data. Next run `auto.arima(dat, trace=TRUE)` to see all the ARIMA models that the function compared. Note, ARIMA(0,1,0) is a random walk with b=1. ARIMA(0,1,0) with drift would be a random walk (b=1) with drift (with u).

e. Is the difference in the AICc values between a random walk with and without drift comparable between MARSS() and auto.arima()?

Note when using `auto.arima()`, an AR(1) model of the following form will be fit (notice the b): $x_t = bx_{t-1} + w_t$. `auto.arima()` refers to this model $x_t = x_{t-1} + w_t$, which is also AR(1) but with $b = 1$, as ARIMA(0,1,0). This says that the first difference of the data (that's the 1 in the middle) is a ARMA(0,0) process (the 0s in the 1st and 3rd spots). So ARIMA(0,1,0) means this: $x_t - x_{t-1} = w_t$.

3. Create a random walk with drift time series using `cumsum()` and `rnorm()`. Look at the `rnorm()` help file (`?rnorm`) to make sure you know what the arguments to the `rnorm()` are.

```
dat <- cumsum(rnorm(100, 0.1, 1))
```

a. What is the order of this random walk written as ARIMA(p, d, q)? "what is the order" means "what is p, d, and q. Model"order" is how `arima()` and `Arima()` specify arima models.

b. Fit that model using `Arima()` in the **forecast** package. You'll need to specify the arguments `order` and `include.drift`. Use `?Arima` to review what that function does if needed.

c. Write out the equation for this random walk as a univariate state-space model. Notice that there is no observation error, but still write this as a state-space model.

d. Fit that model with `MARSS()`.

e. How are the two estimates from `Arima()` and `MARSS()` different?

4. The first-difference of `dat` used in the previous problem is:

```
diff.dat = diff(dat)
```

Use `?diff` to check what the `diff()` function does.

a. If x_t denotes a time series. What is the first difference of x? What is the second difference?

b. What is the `x` model for `diff.dat`? Look at your answer to part (a) and the answer to part (e).

c. Fit `diff.dat` using `Arima()`. You'll need to change the arguments `order` and `include.mean`.

d. Fit with `MARSS()`. You will need to write the model for `diff.dat` as a state-space model. If you've done this right, the estimated parameters using `Arima()` and `MARSS()` will now be the same.

This question should clue you into the fact that `Arima()` is not exactly fitting Equation (2.1). It's very similar, but not quite written that way. By the way, Equation (2.1) is how structural time series observed with error are written (state-space models). To recover the estimates that a function like `arima()` or `Arima()` returns, you need to write your state-space model in a specific way (as seen above).

5. `Arima()` will also fit what it calls an "AR(1) with drift". An AR(1) with drift is NOT this model:

$$x_t = bx_{t-1} + u + w_t \text{ where } w_t \sim \text{N}(0, q) \tag{2.18}$$

In the population dynamics literature, this equation is called the Gompertz model and is a type of density-dependent population model.

a. Write R code to simulate Equation (2.18). Make b less than 1 and greater than 0. Set u and x_0 to whatever you want. You can use a for loop.

b. Plot the trajectories and show that this model does not "drift" upward or downward. It fluctuates about a mean value.

c. Hold b constant and change u. How do the trajectories change?

d. Hold u constant and change b. Make sure to use a b close to 1 and another close to 0. How do the trajectories change?

e. Do 2 simulations each with the same w_t. In one simulation, set $u = 1$ and in the other $u = 2$. For both simulations, set $x_1 = u/(1-b)$. You can set b to whatever you want as long as $0 < b < 1$. Plot the 2 trajectories on the same plot. What is different?

We will fit what `Arima()` calls "AR(1) with drift" models in the chapter on MARSS models with covariates.

6. The **MARSS** package includes a data set of gray whales. Load the data to use as follows:

```
library(MARSS)
dat <- log(graywhales[, 2])
```

Fit a random walk with drift model observed with error to the data:

$$\begin{gathered} x_t = x_{t-1} + u + w_t \text{ where } w_t \sim \text{N}(0, q) \\ y_t = x_t + v_t \text{ where } v_t \sim \text{N}(0, r) \\ x_0 = a \end{gathered} \tag{2.19}$$

y is the whale count in year t. x is interpreted as the 'true' unknown population size that we are trying to estimate.

a. Fit this model with `MARSS()`

b. Plot the estimated x as a line with the actual counts added as points. x is in `fit$states`. It is a matrix. To plot using `plot()`, you will need to change it to a vector using `as.vector()` or `fit$states[1,]`

c. Simulate 1000 sample gray whale populstion trajectories (the x in your model) using the estimated u and q starting at the estimated x in 1997. You can do this with a couple for loops or write something terse with `cumsum()` and `apply()`.

d. Using these simulated trajectories, what is your estimate of the probability that the grey whale population will be above 50,000 graywhales in 2007?

e. What kind(s) of uncertainty does your estimate above NOT include?

7. Fit the following models to the graywhales data using MARSS(). Assume $b = 1$.

   - Model 1 Process error only model with drift
   - Model 2 Process error only model without drift
   - Model 3 Process error with drift and observation error with observation error variance fixed = 0.05.
   - Model 4 Process error with drift and observation error with observation error variance estimated.

   a. Compute the AICc's for each model and likelihood or deviance (-2 * log likelihood). Where to find these? Try `names(fit)`. `logLik()` is the standard R function to return log-likelihood from fits.

   b. Calculate a table of ΔAICc values and AICc weights.

   c. Show the acf of the model and state residuals for the best model. You will need a vector of the residuals to do this. If `fit` is the fit from a fit call like `fit = MARSS(dat)`, you get the residuals using this code:

   ```
   residuals(fit)$state.residuals[1, ]
   residuals(fit)$model.residuals[1, ]
   ```

   Do the acf's suggest any problems?

8. Evaluate the predictive accuracy of forecasts using the **forecast** package using the `airmiles` dataset. Load the data to use as follows:

   ```
   library(forecast)
   dat <- log(airmiles)
   n <- length(dat)
   training.dat <- dat[1:(n - 3)]
   test.dat <- dat[(n - 2):n]
   ```

   This will prepare the training data and set aside the last 3 data points for validation.

a. Fit the following four models using `Arima()`: ARIMA(0,0,0), ARIMA(1,0,0), ARIMA(0,0,1), ARIMA(1,0,1).

b. Use `forecast()` to make 3 step ahead forecasts from each.

c. Calculate the MASE statistic for each using the `accuracy()` function in the **forecast** package. Type `?accuracy` to learn how to use this function.

d. Present the results in a table.

e. Which model is best supported based on the MASE statistic?

9. The WhaleNet Archive of STOP Data has movement data on loggerhead turtles on the east coast of the US from ARGOS tags. The **MARSS** package `loggerheadNoisy` dataset is lat/lot data on eight individuals, however we have corrupted this data severely by adding random errors in order to create a "bad tag" problem (very noisy). Use `head(loggerheadNoisy)` to get an idea of the data. Then load the data on one turtle, MaryLee. MARSS needs time across the columns to you need to use transpose the data (as shown).

```
turtlename <- "MaryLee"
dat <- loggerheadNoisy[which(loggerheadNoisy$turtle == turtlename),
    5:6]
dat <- t(dat)
```

a. Plot MaryLee's locations (as a line not dots). Put the latitude locations on the y-axis and the longitude on the y-axis. You can use `rownames(dat)` to see which is in which row. You can just use `plot()` for the homework. But if you want, you can look at the MARSS Manual chapter on animal movement to see how to plot the turtle locations on a map using the **maps** package.

b. Analyze the data with a state-space model (movement observed with error) using

```
fit0 <- MARSS(dat)
```

Look at the output from the above MARSS call. What is the meaning of the parameters output from MARSS in terms of turtle movement? What exactly is the u estimate for example? Look at the data and think about the model you fit.

c. What assumption did the default MARSS model make about observation error and process error? What does that assumption mean in terms of how steps in the N-S and E-W directions are related? What does that assumption mean in terms of our assumption about the latitudal and longitudinal observation errors?

d. Does MaryLee move faster in the latitude direction versus longitude direction?

e. Add MaryLee's estimated "true" positions to your plot of her locations. You can use `lines(x, y, col="red")` (with x and y replaced with your x and y data). The true position is the "state". This is in the states element of an output from MARSS `fit0$states`.

f. Fit the following models with different assumptions regarding the movement in the lat/lon direction:

- Lat/lon movements are independent but the variance is the same
- Lat/lon movements are correlated and lat/lon variances are different
- Lat/lon movements are correlated and the lat/lon variances are the same.

You only need to change `Q` specification. Your MARSS call will now look like the following with `...` replaced with your `Q` specification.

```
fit1 <- MARSS(dat, list(Q = ...))
```

g. Plot your state residuals (true location residuals). What are the problems? Discuss in reference to your plot of the location data. Here is how to get state residuals from `MARSS()` output:

```
resids <- residuals(fit0)$state.residuals
```

`The lon residuals are in row 1 and lat residuals are in row 2 (same order as`