MARSS Package Manual

E. E. Holmes

2019-11-26

Contents

Preface

The **MARSS** R package allows you to fit **constrained** multivariate autoregressive state-space models.

This manual covers the **MARSS R** package: what it does, how to set up your models, how to structure your input, and how to get different types of output. For vignettes showing how to use MARSS models to analyze data, see the companion book *MARSS Modeling for Environmental Data* by Holmes, Scheurell, and Ward.

Installation

To install and load the **MARSS** package from CRAN:

```
install.packages("MARSS")
library(MARSS)
```

The latest release on GitHub may be ahead of the CRAN release. To install the latest release on GitHub:

```
install.packages("devtools")
library(devtools)
install_github("nwfsc-timeseries/MARSS@*release")
library(MARSS)
```

The master branch on GitHub is not a 'release'. It has work leading up to a GitHub release. The code here may be broken though usually preliminary work is done on a development branch. To install the master branch:

```
install_github("nwfsc-timeseries/MARSS")
```

If you are on a Windows machine and get an error saying 'loading failed for i386' or similar, then try

```
options(devtools.install.args = "--no-multiarch")
```

To install an **R** package from Github, you need to be able to build an **R** package on your machine. If you are on Windows, that means you will need to install Rtools. On a Mac, installation should work fine; you don't need to install anything.

Author

Elizabeth E. Holmes is a research scientist at the Northwest Fisheries Science Center (NWFSC) a US Federal government research center. This work was conducted as part of her job for NOAA Fisheries, as such the work is in the public domain and cannot be copyrighted.

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

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

Citation

Holmes, E. E. 2019. MARSS Manual. NOAA Fisheries, Northwest Fisheries Science Center, 2725 Montlake Blvd E., Seattle, WA 98112. Contact eli.holmes@noaa.gov.

Preface

Chapter 1

Overview

MARSS stands for Multivariate Auto-Regressive(1) State-Space. The **MARSS** package is an ,**R** package for estimating the parameters of linear MARSS models with Gaussian errors. This class of model is extremely important in the study of linear stochastic dynamical systems, and these models are important in many different fields, including economics, engineering, genetics, physics and ecology (Appendix **??**). The model class has different names in different fields, for example in some fields they are termed dynamic linear models (DLMs) or vector autoregressive (VAR) state-space models. The **MARSS** package allows you to easily fit time-varying constrained and unconstrained MARSS models with or without covariates to multivariate time-series data via maximum-likelihood using primarily an EM algorithm. The EM algorithm in the **MARSS** package allows you to apply linear constraints on all the parameters within the model matrices. Fitting via the BFGS algorithm is also provided in the package using ,**R**'s `optim` function, but this is not the focus of the **MARSS** package.

A full MARSS model, with Gaussian errors, takes the form:

$$\mathbf{x}_t = \mathbf{B}_t\mathbf{x}_{t-1} + \mathbf{u}_t + \mathbf{C}_t\mathbf{c}_t + \mathbf{G}_t\mathbf{w}_t, \text{ where } \mathbf{w}_t \sim \text{MVN}(0, \mathbf{Q}_t) \tag{1.1a}$$

$$\mathbf{y}_t = \mathbf{Z}_t\mathbf{x}_t + \mathbf{a}_t + \mathbf{D}_t\mathbf{d}_t + \mathbf{H}_t\mathbf{v}_t, \text{ where } \mathbf{v}_t \sim \text{MVN}(0, \mathbf{R}_t) \tag{1.1b}$$

$$\mathbf{x}_1 \sim \text{MVN}(\boldsymbol{\pi}, \boldsymbol{\Lambda}) \text{ or } \mathbf{x}_0 \sim \text{MVN}(\boldsymbol{\pi}, \boldsymbol{\Lambda}) \tag{1.1c}$$

The $\mathbf{x}$ equation is termed the state process and the $\mathbf{y}$ equation is termed the

observation process. Data enter the model as the $\mathbf{y}$; that is the $\mathbf{y}$ is treated as the data although there may be missing data. The $\mathbf{c}_t$ and $\mathbf{d}_t$ are inputs (aka, exogenous variables, covariates or indicator variables). The $\mathbf{G}_t$ and $\mathbf{H}_t$ are also typically inputs (fixed values with no missing values).

The bolded terms are matrices with the following definitions:

$\mathbf{x}$ is a $m \times T$ matrix of states. Each $\mathbf{x}_t$ is a realization of the random variable $\mathbf{X}_t$ at time t.

$\mathbf{w}$ is a $m \times T$ matrix of the process errors. The process errors at time t are multivariate normal with mean 0 and covariance matrix $\mathbf{Q}_t$.

$\mathbf{y}$ is a $n \times T$ matrix of the observations. Some observations may be missing.

$\mathbf{v}$ is a $n \times T$ column vector of the non-process errors. The observation erros at time t are multivariate normal with mean 0 and covariance matrix $\mathbf{R}_t$.

$\mathbf{B}_t$ **and** $\mathbf{Z}_t$ are parameters and are $m \times m$ and $n \times m$ matrices.

$\mathbf{u}_t$ **and** $\mathbf{a}_t$ are parameters and are $m \times 1$ and $n \times 1$ column vectors.

$\mathbf{Q}_t$ **and** $\mathbf{R}_t$ are parameters and are $g \times g$ (typically $m \times m$) and $h \times h$ (typically $n \times n$) variance-covariance matrices.

$\boldsymbol{\pi}$ is either a parameter or a fixed prior. It is a $m \times 1$ matrix.

$\boldsymbol{\Lambda}$ is either a parameter or a fixed prior. It is a $m \times m$ variance-covariance matrix.

$\mathbf{C}_t$ **and** $\mathbf{D}_t$ are parameters and are $m \times p$ and $n \times q$ matrices.

c and d are inputs (no missing values) and are $p \times T$ and $q \times T$ matrices.

$\mathbf{G}_t$ **and** $\mathbf{H}_t$ are inputs (no missing values) and are $m \times g$ and $n \times h$ matrices.

AR(p) models can be written in the above form by properly defining the $\mathbf{x}$ vector and setting some of the $\mathbf{R}$ variances to zero; see Chapter **??**. Although the model appears to only include i.i.d. errors ($\mathbf{v}_t$ and $\mathbf{w}_t$), in practice, AR(p) errors can be included by moving the error terms into the state model. Similarly, the model appears to have independent process ($\mathbf{v}_t$) and observation ($\mathbf{w}_t$) errors, however, in practice, these can be modeled as identical or correlated by using one of the state processes to model the errors with the $\mathbf{B}$ matrix set appropriately for AR or white noise—although one may have to

fix many of the parameters associated with the errors to have an identifiable model. Study the application chapters and textbooks on MARSS models (Appendix **??**) for examples of how a wide variety of autoregressive models can be written in MARSS form.

1.1 Examples of MARSS models

Written in an unconstrained form, meaning all the elements in a parameter matrices are allowed to be different and none constrained to be equal or related, a MARSS model can be written out as follows. Two state processes ($\mathbf{x}$) and three observation processes ($\mathbf{y}$) are used here as an example.

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}_t = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}_{t-1} + \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}_t, \quad \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}_t \sim \text{MVN}\left(\begin{bmatrix} u_1 \\ u_2 \end{bmatrix}, \begin{bmatrix} q_{11} & q_{12} \\ q_{21} & q_{22} \end{bmatrix} \right)$$

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}_t = \begin{bmatrix} z_{11} & z_{12} \\ z_{21} & z_{22} \\ z_{31} & z_{32} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}_t + \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}_t, \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}_t \sim \text{MVN}\left(\begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}, \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \right)$$

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}_0 \sim \text{MVN}\left(\begin{bmatrix} \pi_1 \\ \pi_2 \end{bmatrix}, \begin{bmatrix} \nu_{11} & \nu_{12} \\ \nu_{21} & \nu_{22} \end{bmatrix} \right) \quad \textit{or} \quad \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}_1 \sim \text{MVN}\left(\begin{bmatrix} \pi_1 \\ \pi_2 \end{bmatrix}, \begin{bmatrix} \nu_{11} & \nu_{12} \\ \nu_{21} & \nu_{22} \end{bmatrix} \right)$$

However not all parameter elements can be estimated simultaneously. Constraints are required in order to specify a model with a unique solution. The MARSS package allows you to specify constraints by fixing elements in a parameter matrix or specifying that some elements are estimated—and have a linear relationship to other elements. Here is an example of a MARSS model

with fixed and estimated parameter elements:

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}_t = \begin{bmatrix} a & 0 \\ 0 & a \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}_{t-1} + \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}_t, \quad \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}_t \sim \text{MVN}\left(\begin{bmatrix} 0.1 \\ u \end{bmatrix}, \begin{bmatrix} q_{11} & q_{12} \\ q_{12} & q_{22} \end{bmatrix} \right)$$

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}_t = \begin{bmatrix} d & d \\ c & c \\ 1+2d+3c & 2+3d \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}_t + \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}_t, \quad \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}_t \sim \text{MVN}\left(\begin{bmatrix} a_1 \\ a_2 \\ 0 \end{bmatrix}, \begin{bmatrix} r & 0 & 0 \\ 0 & r & 0 \\ 0 & 0 & r \end{bmatrix} \right)$$

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}_0 \sim \text{MVN}\left(\begin{bmatrix} \pi \\ \pi \end{bmatrix}, \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \right)$$

Notice that some elements are fixed (in this case to 0, but could be any fixed number), some elements are shared (have the same value), and some elements are linear combinations of other estimated values (c, $1 + 2d + 3c$ and $2 + 3d$ are linear combinations of c and d).

Chapter 2

How to get started (quickly)

If you already work with models in the form of Equation 1.1, you can immediately fit your model with the **MARSS** package. Install the **MARSS** package and then type `library(MARSS)` at the command line to load the package. Look at the Quick Start Guide and then skim through Chapter **??** to, hopefully, find an example similar to your application. Appendix **??** also has many examples of how to specify different forms for your parameter matrices.

2.1 Getting your data in right format

Your data need to be a matrix (not dataframe nor a ts object) with time across the columns ($n \times T$ matrix). The **MARSS** functions assume discrete time steps and you will need a column for each time step. Replace any missing time steps with NA. Write your model down on paper and identify which parameters correspond to which parameter matrices in Equation 1.1. Call the `MARSS()` function (Chapter **??**) using your data and using the `model` argument to specify the structure of each parameter.

A R **ts** object (time series object) stores information about the time steps of the data and often seasonal information (the quarter or month). `MARSS()` needs this information in matrix form. If you have your data in **ts** form, then you are probably using year and season (quarter, month) as covariates

to estimate trend and seasonality. Here is how to get your **ts** into the form that `MARSS()` wants.

Here is how to get your ts into the form that `MARSS()` needs.

2.1.1 Univariate example

This converts a univariate ts object with year and quarter into a matrix with a row for the response (here called `Temp`), year, and quarter.

```
z = ts(rnorm(10), frequency = 4, start = c(1959, 2))
dat = data.frame(Yr = floor(time(z) + .Machine$double.eps),
      Qtr = cycle(z), Temp=z)
dat = t(dat)
```

When you call `MARSS()`, `dat["Temp",]` is the data. `dat[c("Yr","Qtr"),]` are your covariates.

2.1.2 Multivariate example

In this example, we have two temperature readings and a salinity reading. The data are monthly.

```
z <- ts(matrix(rnorm(300), 100, 3), start = c(1961, 1),
     frequency = 12, names=c("Temp1","Temp2","Sal"))
dat = data.frame(Yr = floor(time(z) + .Machine$double.eps),
     Month = cycle(z), z)
```

When you call `MARSS()`, `dat[c("Temp1","Temp2"),]` are the data and `dat[c("Yr","Month","Sal"),]` are your covariates.

See the chapters that discuss seasonality for examples of how to model seasonality. The brute force method of treating month or quarter as a factor requires estimation of more parameters than necessary in many cases.

2.2 Important notes about the algorithms

Specification of a properly constrained model with a unique solution is the responsibility of the user because `MARSS()` has no way to tell if you have specified an insufficiently constrained model—with correspondingly an infinite number of solutions. How do you know if the model is properly constrained? If you are using a MARSS model form that is widely used, then you can probably assume that it is properly constrained. If you go to papers where someone developed the model or method, the issue of constraints necessary to ensure "identifiability" will likely be addressed if it is an issue. Are you fitting novel MARSS models? Then you will need to do some study on identifiability in this class of models using textbooks (Appendix **??**). Often textbooks do not address identifiability explicitly. Rather it is addressed implicitly by only showing a model constructed in such a way that it is identifiable. In our work, if we suspect identification problems, we will often first do a Bayesian analysis with flat priors and look for oddities in the posteriors, such as ridges, plateaus or bimodality.

All the EM code in the **MARSS** package is currently in native ,**R**. Thus the model fitting is slow. The classic Kalman filter/smoother algorithm, as shown in **?**, p. 331-335, is based on the original smoother presented in **?**. This Kalman filter is provided in function `MARSSkfss()`, but the default Kalman filter and smoother used in the **MARSS** package is based on the algorithm in **?** and papers by Koopman et al. This Kalman filter and smoother is provided in the **KFAS** package (Helske 2012). Table 2 in **?** indicates that the classic algorithm is 40-100 times slower than the algorithm given in **?**, **?**, and **?**. The **MARSS** package function `MARSSkfas()` provides a translator between the model objects in **MARSS** and those in **KFAS** so that the **KFAS** functions can be used. `MARSSkfas()` also includes a lag-one covariance smoother algorithm as this is not output by the **KFAS** functions, and it provides proper formulation of the priors so that one can use the **KFAS** functions when the prior on the states is set at $t = 0$ instead of $t = 1$. Simply off-setting your data to start at t=2 and sending that value to $t_{init} = 1$ in the **KFAS** Kalman filter would not be mathematically correct!

EM algorithms will quickly get in the vicinity of the maximum likelihood, but the final approach to the maximum is generally slow relative to quasi-Newton methods. On the flip side, EM algorithms are quite robust to initial

conditions choices and can be extremely fast at getting close to the MLE values for high-dimensional models. The **MARSS** package also allows one to use the BFGS method to fit MARSS models, thus one can use an EM algorithm to "get close" and then the BFGS algorithm to polish off the estimate. Restricted maximum-likelihood algorithms are also available for AR(1) state-space models, both univariate (**?**) and multivariate (**?**). REML can give parameter estimates with lower variance than plain maximum-likelihood algorithms. However, the algorithms for REML when there are missing values are not currently available (although that will probably change in the near future). Another maximum-likelihood method is data-cloning which adapts MCMC algorithms used in Bayesian analysis for maximum-likelihood estimation (**?**).

Missing values are seamlessly accommodated with the **MARSS** package. Simply specify missing data with NAs. The likelihood computations are exact and will deal appropriately with missing values. However, no innovations, referring to the non-parametric bootstrap developed by Stoffer and Wall (1991), bootstrapping can be done if there are missing values. Instead parametric bootstrapping must be used.

You should be aware that maximum-likelihood estimates of variance in MARSS models are fundamentally biased, regardless of the algorithm used. This bias is more severe when one or the other of $\mathbf{R}$ or $\mathbf{Q}$ is very small, and the bias does not go to zero as sample size goes to infinity. The bias arises because variance is constrained to be positive. Thus if $\mathbf{R}$ or $\mathbf{Q}$ is essentially zero, the mean estimate will not be zero and thus the estimate will be biased high while the corresponding bias of the other variance will be biased low. You can generate unbiased variance estimates using a bootstrap estimate of the bias. The function `MARSSparamCIs()` will do this. However be aware that adding an estimated bias to a parameter estimate will lead to an increase in the variance of your parameter estimate. The amount of variance added will depend on sample size.

You should also be aware that mis-specification of the prior on the initial states ($\boldsymbol{\pi}$ and $\boldsymbol{\Lambda}$) can have catastrophic effects on your parameter estimates if your prior conflicts with the distribution of the initial states implied by the MARSS model. These effects can be very difficult to detect because the model will appear to be well-fitted. Unless you have a good idea of what the parameters should be, you might not realize that your prior conflicts.

The most common problems we have found with priors on $\mathbf{x}_0$ are the following. Problem 1) The correlation structure in $\mathbf{\Lambda}$ (whether the prior is diffuse or not) does not match the correlation structure in $\mathbf{x}_0$ implied by your model. For example, you specify a diagonal $\mathbf{\Lambda}$ (independent states), but the implied distribution has correlations. Problem 2) The correlation structure in $\mathbf{\Lambda}$ does not match the structure in $\mathbf{x}_0$ implied by constraints you placed on $\boldsymbol{\pi}$. For example, you specify that all values in $\boldsymbol{\pi}$ are shared, yet you specify that $\mathbf{\Lambda}$ is diagonal (independent).

Unfortunately, using a diffuse prior does not help with these two problems because the diffuse prior still has a correlation structure and can still conflict with the implied correlation in $\mathbf{x}_0$. One way to get around these problems is to set $\mathbf{\Lambda}$=0 (a $m \times m$ matrix of zeros) and estimate $\boldsymbol{\pi} \equiv \mathbf{x}_0$ only. Now $\boldsymbol{\pi}$ is a fixed but unknown (estimated) parameter, not the mean of a distribution. In this case, $\mathbf{\Lambda}$ does not exist in your model and there is no conflict with the model.
Be aware however that estimating $\boldsymbol{\pi}$ as a parameter is not always robust. If you specify that $\mathbf{\Lambda}$=0 and specify that $\boldsymbol{\pi}$ corresponds to $\mathbf{x}_0$, but your model "explodes" when run backwards in time, you cannot estimate $\boldsymbol{\pi}$ because you cannot get a good estimate of $\mathbf{x}_0$. Sometimes this can be avoided by specifying that $\boldsymbol{\pi}$ corresponds to $\mathbf{x}_1$ so that it can be constrained by the data $\mathbf{y}_1$.

In summary, if the implied correlation structure of your initial states is independent (diagonal variance-covariance matrix), you should generally be ok with a diagonal and high variance prior or with treating the initial states as parameters (with $\mathbf{\Lambda} = 0$). But if your initial states have an implied correlation structure that is not independent, then proceed with caution. 'With caution' means that you should assume you have problems and test how your model fits with simulated data.

2.3 State-space form of ARMA(p,q) models

There is a large class of models in the statistical finance literature that have the form

$$\mathbf{x}_{t+1} = \mathbf{B}\mathbf{x}_t + \mathbf{\Gamma}\boldsymbol{\eta}_t$$
$$\mathbf{y}_t = \mathbf{Z}\mathbf{x}_t + \boldsymbol{\eta}_t$$

For example, ARMA(p,q) models can be written in this form. The MARSS model framework in this package will not allow you to write models in that form. You can put the $\boldsymbol{\eta}_t$ into the $\mathbf{x}_t$ vector and set $\mathbf{R} = 0$ to make models of this form using the MARSS form, but the EM algorithm in the **MARSS** package won't let you estimate parameters because the parameters will drop out of the full likelihood being maximized in the algorithm. You can try using BFGS by passing in the `method` argument to the `MARSS()` call.

2.4 Troubleshooting

Numerical errors due to ill-conditioned matrices are not uncommon when fitting MARSS models. The Kalman and EM algorithms need inverses of matrices. If those matrices become ill-conditioned, for example all elements are close to the same value, then the algorithm becomes unstable. Warning messages will be printed if the algorithms are becoming unstable and you can set `control$trace=1`, to see details of where the algorithm is becoming unstable. Whenever possible, you should avoid using shared $\boldsymbol{\pi}$ values in your model[1]. The way our algorithm deals with $\boldsymbol{\Lambda}$ tends to make this case unstable, especially if $\mathbf{R}$ is not diagonal. In general, estimation of a non-diagonal $\mathbf{R}$ is more difficult, more prone to ill-conditioning, and more data-hungry.

You may also see non-convergence warnings, especially if your MLE model turns out to be degenerate. This means that one of the elements on the diagonal of your $\mathbf{Q}$ or $\mathbf{R}$ matrix are going to zero (are degenerate). It will take the EM algorithm forever to get to zero. BFGS will have the same problem, although it will often get a bit closer to the degenerate solution. If you are using `method="kem"`, `MARSS()` will warn you if it looks like the solution is degenerate. If you use `control=list(allow.degen=TRUE)`, the EM algorithm will attempt to set the degenerate variances to zero (instead of trying to get to zero using an infinite number of iterations). However, if one of the variances is going to zero, first think about why this is happening. This is typically caused by one of three problems: 1) you made a mistake in inputting your data, e.g. used -99 as the missing value in your data but did not replace these with NAs before passing to `MARSS()`, 2) your data are not

[1] An example of a $\boldsymbol{\pi}$ with shared values is $\boldsymbol{\pi} = \left[\begin{smallmatrix} a \\ a \\ a \end{smallmatrix}\right]$.

sufficient to estimate multiple variances or 3) your data are inconsistent with the model you are trying fit.

The algorithms in the **MARSS** package are designed for cases where the **Q** and **R** diagonals are all non-minuscule. For example, the EM update equation for **u** will grind to a halt (not update **u**) if **Q** is tiny (like 1E-7). Conversely, the BFGS equations are likely to miss the maximum-likelihood when **R** is tiny because then the likelihood surface becomes hyper-sensitive to $\boldsymbol{\pi}$. The solution is to use the degenerate likelihood function for the likelihood calculation and the EM update equations. `MARSS()` will implement this automatically when **Q** or **R** diagonal elements are set to zero and will try setting **Q** and **R** terms to zero automatically if `control$allow.degen=TRUE`.

One odd case can occur when **R** goes to zero (a matrix of zeros), but you are estimating $\boldsymbol{\pi}$. If `model$tinitx=1`, then $\boldsymbol{\pi} = \mathbf{x}_1^0$ and $\mathbf{y}_1 - \mathbf{Z}\mathbf{x}_1^0$ can go to 0 as well as $\text{var}(\mathbf{y}_1 - \mathbf{Z}\mathbf{x}_1^0)$ by driving **R** to zero. But as this happens, the log-likelihood associated with $\mathbf{y}_1$ will go (correctly) to infinity and thus the log-likelihood goes to infinity. But if you set $\mathbf{R} = 0$, the log-likelihood will be finite. The reason is that $\mathbf{R} \approx 0$ and $\mathbf{R} = 0$ specify different likelihoods associated with $\mathbf{y}_1 - \mathbf{Z}\mathbf{x}_1^0$. With $\mathbf{R} = 0$, $\mathbf{y}_1 - \mathbf{Z}\mathbf{x}_1^0$ does not have a distribution; it is just a fixed value. So there is no likelihood to go to infinity. If some elements of the diagonal of **R** are going to zero, you should be suspect of the parameter estimates. Sometimes the structure of your data, e.g. one data value followed by a long string of missing values, is causing an odd spike in the likelihood at $\mathbf{R} \approx 0$. Try manually setting **R** equal to zero to get the correct log-likelihood[2].

2.5 Other related packages

Packages that will do Kalman filtering and smoothing are many, but packages that estimate the parameters in a MARSS model, especially constrained MARSS models, are much less common. The following are those with which we are familiar, however there are certainly more packages for estimating MARSS models in engineering and economics of which we are unfamiliar.

[2]The likelihood returned when $\mathbf{R} \approx 0$ is not incorrect. It is just not the likelihood that you probably want. You want the likelihood where the **R** term is dropped because it is zero.

The **MARSS** package is unusual in that it uses an EM algorithm for maximizing the likelihood as opposed to a Newton-esque method (e.g. BFGS). The package is also unusual in that it allows you to specify the initial conditions at $t = 0$ or $t = 1$, allows degenerate models (with some of the diagonal elements of **R** or **Q** equal to zero). Lastly, model specification in the **MARSS** package has a one-to-one relationship between the model list in `MARSS()` and the model as you would write it on paper as a matrix equation. This makes the learning curve a bit less steep. However, the **MARSS** package has not been optimized for speed and probably will be really slow if you have time-series data with a lot of time points.

atsar atsar is an **R** package we wrote for fitting MARSS models using STAN. It allows fast and flexible fitting of MARSS models in a Bayesian framework. Our book from our time-series class has example applications Applied Time-Series Analysis for Fisheries and Environmental Sciences.

stats The **stats** package (part of base R) has functions for fitting univariate structural time series models (MARSS models with a univariate y). Read the help file at '?StructTS'. The Kalman filter and smoother functions are described here: '?KalmanLike'.

DLM DLM is an **R** package for fitting MARSS models. Our impression is that it is mainly Bayesian focused but it does allow MLE estimation via the 'optim()' function. It has a book, Dynamic Linear Models with **R** by Petris et al., which has many examples of how to write MARSS models for different applications.

sspir sspir an **R** package for fitting ARSS (univariate) models with Gaussian, Poisson and binomial error distributions.

dse dse (Dynamic Systems Estimation) is an **R** package for multivariate Gaussian state-space models with a focus on ARMA models.

SsfPack SsfPack is a package for Ox/Splus that fits constrained multivariate Gaussian state-space models using mainly (it seems) the BFGS algorithm but the newer versions support other types of maximization. SsfPack is very flexible and written in C to be fast. It has been used extensively on statistical finance problems and is optimized for dealing with large (financial) data sets. It is used and documented in Time Series Analysis by State Space Methods by Durbin and Koopman, An Introduction to State Space Time Series Analysis by Commandeur and

Koopman, and Statistical Algorithms for Models in State Space Form: SsfPack 3.0, by Koopman, Shephard, and Doornik.

Brodgar The Brodgar software was developed by Alain Zuur to do (among many other things) dynamic factor analysis, which involves a special type of MARSS model. The methods and many example analyses are given in Analyzing Ecological Data by Zuur, Ieno and Smith. This is the one package that we are aware of that also uses an EM algorithm for parameter estimation.

eViews eViews is a commercial economics software that will estimate at least some types of MARSS models.

KFAS The KFAS **R** package provides a fast Kalman filter and smoother. Examples in the package show how to estimate MARSS models using the **KFAS** functions and **R**'s 'optim()' function. The **MARSS** package uses the filter and smoother functions from the **KFAS** package.

S+FinMetrics S+FinMetrics is a S-plus module for fitting MAR models, which are called vector autoregressive (VAR) models in the economics and finance literature. It has some support for state-space VAR models, though we haven't used it so are not sure which parameters it allows you to estimate. It was developed by Andrew Bruce, Doug Martin, Jiahui Wang, and Eric Zivot, and it has a book associated with it: Modeling Financial Time Series with S-plus by Eric Zivot and Jiahui Wang.

kftrack The kftrack **R** package provides a suite of functions specialized for fitting MARSS models to animal tracking data.

Chapter 3

Short Examples

In this chapter, we work through a series of short examples to illustrate the MARSS package functions. This chapter is oriented towards those who are already somewhat familiar with MARSS models and want to get started quickly. We provide little explanatory text. Those unfamiliar with MARSS models might want to start with the application chapters.

In these examples, we will use the default `form="marxss"` argument for a `MARSS()` call. This specifies a MARSS model of the form:

(#eq:marss.qe)

$$\mathbf{x}_t = \mathbf{B}_t\mathbf{x}_{t-1} + \mathbf{u}_t + \mathbf{C}_t\mathbf{c}_t + \mathbf{G}_t\mathbf{w}_t, \text{ where } \mathbf{w}_t \sim \text{MVN}(0, \mathbf{Q}_t) \tag{3.1a}$$

$$\mathbf{y}_t = \mathbf{Z}_t\mathbf{x}_t + \mathbf{a}_t + \mathbf{D}_t\mathbf{d}_t + \mathbf{H}_t\mathbf{v}_t, \text{ where } \mathbf{v}_t \sim \text{MVN}(0, \mathbf{R}_t) \tag{3.1b}$$

$$\mathbf{x}_1 \sim \text{MVN}(\boldsymbol{\pi}, \boldsymbol{\Lambda}) \text{ or } \mathbf{x}_0 \sim \text{MVN}(\boldsymbol{\pi}, \boldsymbol{\Lambda}) \tag{3.1c}$$

The $\mathbf{c}$ and $\mathbf{d}$ are inputs (not estimated). In the examples here, we leave off $\mathbf{c}$ and $\mathbf{d}$. We address including inputs only briefly at the end of the chapter. See Chapter **??** for extended examples of including covariates as inputs in a MARSS model. We will also not use $\mathbf{G}_t$ or $\mathbf{H}_t$ in this chapter.

3.1 Fixed and estimated elements in parameter matrices

Suppose one has a MARSS model (Equation @ref(eq:marss.qe)) with the following structure:

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \end{bmatrix} = \begin{bmatrix} b_1 & 0.1 \\ b_2 & 2 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \end{bmatrix} + \begin{bmatrix} u \\ u \end{bmatrix} + \begin{bmatrix} w_{1,t} \\ w_{2,t} \end{bmatrix}, \ \mathbf{w}_t \sim \text{MVN}\left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} q_1 & q_3 \\ q_3 & q_2 \end{bmatrix} \right)$$

$$\begin{bmatrix} y_{1,t} \\ y_{2,t} \\ y_{3,t} \end{bmatrix} = \begin{bmatrix} z_1 & 0 \\ z_2 & z_2 \\ 0 & 3 \end{bmatrix} \begin{bmatrix} x_{1,t} \\ x_{2,t} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} v_{1,t} \\ v_{2,t} \\ v_{3,t} \end{bmatrix}, \ \mathbf{v}_t \sim \text{MVN}\left(\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} r & 0 & 0 \\ 0 & r & 0 \\ 0 & 0 & 1 \end{bmatrix} \right)$$

$$\mathbf{x}_0 \sim \text{MVN}\left(\begin{bmatrix} \pi_1 \\ \pi_2 \end{bmatrix}, \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \right)$$

Notice how this model mixes fixed values, estimated values and shared values.

In MARSS, model structure is specified using a list with the names, `Z`, `A`, `R`, `B`, `U`, `Q`, `x0` and `V0`. Each element is matrix (class matrix) with the same dimensions as the matrix of the same name in the MARSS model. MARSS distinguishes between the estimated and fixed values in a matrix by using a list matrix in which you can have numeric and character elements. Numeric elements are fixed; character elements are names of things to be estimated. The model above would be specified as:

```
Z = matrix(list("z1", "z2", 0, 0, "z2", 3), 3, 2)
A = matrix(0, 3, 1)
R = matrix(list(0), 3, 3)
diag(R) = c("r", "r", 1)
B = matrix(list("b1", 0.1, "b2", 2), 2, 2)
U = matrix(c("u", "u"), 2, 1)
Q = matrix(c("q1", "q3", "q3", "q2"), 2, 2)
x0 = matrix(c("pi1", "pi2"), 2, 1)
V0 = diag(1, 2)
model.gen = list(Z = Z, A = A, R = R, B = B, U = U, Q = Q, x0 = x0,
    V0 = V0, tinitx = 0)
```

Notice that there is a one-to-one correspondence between the model list in ,**R**

and the model on paper. Fitting the model is then just a matter of passing the data and model list to the MARSS function:

```
kemfit = MARSS(dat, model = model.gen)
```

If you work often with MARSS models then you will probably know whether prior sensitivity is a problem for your types of MARSS applications. If so, note that the MARSS package is unusual in that it allows you to set $\mathbf{\Lambda} = 0$ and treat $\mathbf{x}_0$ as an unknown estimated parameter. This eliminates the prior and thus the prior sensitivity problems—at the cost of adding m parameters. Depending on your application, you may need to set the initial conditions at $t = 1$ instead of the default of $t = 0$. If you are unsure, look in the index and read all the sections that talk about troubleshooting priors.

3.2 Different numbers of state processes

Here we show a series of short examples using a dataset on Washington harbor seals (`?harborSealWA`), which has five observation time series. The dataset is a little unusual in that it has four missing years from years 2 to 5. This causes some interesting issues with prior specification. Before starting the harbor seal examples, we set up the data, making time go across the columns and removing the year column:

```
dat = t(harborSealWA)
dat = dat[2:nrow(dat), ]  #remove the year row
```

3.2.1 One hidden state process for each observation time series

This is the default model for the `MARSS()` function. In this case, $n = m$, the observation errors are i.i.d. and the process errors are independent and have different variances. The elements in $\mathbf{u}$ are all different (meaning, they are

not forced to be the same). Mathematically, the MARSS model being fit is:

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \\ x_{3,t} \\ x_{4,t} \\ x_{5,t} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \\ x_{3,t-1} \\ x_{4,t-1} \\ x_{5,t-1} \end{bmatrix} + \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \end{bmatrix} + \begin{bmatrix} w_{1,t} \\ w_{2,t} \\ w_{3,t} \\ w_{4,t} \\ w_{5,t} \end{bmatrix}, \ \mathbf{w}_t \sim \text{MVN}\left(\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} q_1 & 0 & 0 & 0 & 0 \\ 0 & q_2 & 0 & 0 & 0 \\ 0 & 0 & q_3 & 0 & 0 \\ 0 & 0 & 0 & q_4 & 0 \\ 0 & 0 & 0 & 0 & q_5 \end{bmatrix} \right)$$

$$\begin{bmatrix} y_{1,t} \\ y_{2,t} \\ y_{3,t} \\ y_{4,t} \\ y_{5,t} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{1,t} \\ x_{2,t} \\ x_{3,t} \\ x_{4,t} \\ x_{5,t} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} v_{1,t} \\ v_{2,t} \\ v_{3,t} \\ v_{4,t} \\ v_{5,t} \end{bmatrix}, \ \mathbf{v}_t \sim \text{MVN}\left(\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} r & 0 & 0 & 0 & 0 \\ 0 & r & 0 & 0 & 0 \\ 0 & 0 & r & 0 & 0 \\ 0 & 0 & 0 & r & 0 \\ 0 & 0 & 0 & 0 & r \end{bmatrix} \right)$$

This is the default model, so you can fit it by simply passing `dat` to `MARSS()`.

```
kemfit = MARSS(dat)
```

```
Success! abstol and log-log tests passed at 38 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 38 iterations.
Log-likelihood: 19.13428
AIC: -6.268557   AICc: 3.805517

                   Estimate
R.diag              0.00895
U.X.SJF             0.06839
U.X.SJI             0.07163
U.X.EBays           0.04179
U.X.PSnd            0.05226
U.X.HC             -0.00279
Q.(X.SJF,X.SJF)     0.03205
Q.(X.SJI,X.SJI)     0.01098
```

```
Q.(X.EBays,X.EBays)  0.00706
Q.(X.PSnd,X.PSnd)    0.00414
Q.(X.HC,X.HC)        0.05450
x0.X.SJF             5.98647
x0.X.SJI             6.72487
x0.X.EBays           6.66212
x0.X.PSnd            5.83969
x0.X.HC              6.60482
Initial states (x0) defined at t=0

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

The output warns you that the convergence tolerance is high. You can set it lower by passing in `control=list(conv.test.slope.tol=0.1)`. `MARSS()` is automatically creating parameter names since you did not tell it the names. To see exactly where each parameter element appears in its parameter matrix, type `summary(kemfit$model)`.

Though it is not necessary to specify the model for this example since it is the default, here is how you could do so using matrices:

```
B = Z = diag(1, 5)
U = matrix(c("u1", "u2", "u3", "u4", "u5"), 5, 1)
x0 = A = matrix(0, 5, 1)
R = Q = matrix(list(0), 5, 5)
diag(R) = "r"
diag(Q) = c("q1", "q2", "q3", "q4", "q5")
```

Notice that when a matrix has both fixed and estimated elements (like **R** and **Q**), a list matrix is used to allow you to specify the fixed elements as numeric and to give the estimated elements character names.

The default MLE method is the EM algorithm (`method="kem"`). You can also use a quasi-Newton method (BFGS) by setting `method="BFGS"`.

```
kemfit.bfgs = MARSS(dat, method = "BFGS")
```

```
Success! Converged in 99 iterations.
Function MARSSkfas used for likelihood calculation.
```

```
MARSS fit is
Estimation method: BFGS
Estimation converged in 99 iterations.
Log-likelihood: 19.13936
AIC: -6.278712   AICc: 3.795362

                    Estimate
R.diag               0.00849
U.X.SJF              0.06838
U.X.SJI              0.07152
U.X.EBays            0.04188
U.X.PSnd             0.05233
U.X.HC              -0.00271
Q.(X.SJF,X.SJF)      0.03368
Q.(X.SJI,X.SJI)      0.01124
Q.(X.EBays,X.EBays)  0.00722
Q.(X.PSnd,X.PSnd)    0.00437
Q.(X.HC,X.HC)        0.05600
x0.X.SJF             5.98437
x0.X.SJI             6.72169
x0.X.EBays           6.65689
x0.X.PSnd            5.83527
x0.X.HC              6.60425
Initial states (x0) defined at t=0

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

Using the default EM convergence criteria, the EM algorithm stops at a log-likelihood a little lower than the BFGS algorithm does, but the EM algorithm was faster, 6.2 times faster, in this case. If you wanted to use the EM fit as the initial conditions, pass in the `inits` argument using the `$par` element (or `coef(fit,form="marss")`) of the EM fit.

```
kemfit.bfgs2 = MARSS(dat, method = "BFGS", inits = kemfit$par)
```

The BFGS algorithm now converges in 103 iterations. Output not shown.

We mentioned that the missing years from year 2 to 4 creates an interesting issue with the prior specification. The default behavior of MARSS is to treat the initial state as at $t = 0$ instead of $t = 1$. Usually this doesn't make a difference, but for this dataset, if we set the prior at $t = 1$, the MLE estimate of $\mathbf{R}$ becomes 0. If we estimate $\mathbf{x}_1$ as a parameter and let $\mathbf{R}$ go to 0, the likelihood will go to infinity (slowly but surely). This is neither an error nor a pathology, but is probably not what you would like to have happen. Note that the "BFGS" algorithm will not find the maximum in this case; it will stop before $\mathbf{R}$ gets small and the likelihood gets very large. However, the EM algorithm will climb up the peak. You can try it by running the following code. It will report warnings which you can read about in Appendix **??**.

```
kemfit.strange = MARSS(dat, model = list(tinitx = 1))
```

3.2.2 Five correlated hidden state processes

This is the same model except that the five hidden states have correlated process errors. Mathematically, this is the model:

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \\ x_{3,t} \\ x_{4,t} \\ x_{5,t} \end{bmatrix} = \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \\ x_{3,t-1} \\ x_{4,t-1} \\ x_{5,t-1} \end{bmatrix} + \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \end{bmatrix} + \begin{bmatrix} w_{1,t} \\ w_{2,t} \\ w_{3,t} \\ w_{4,t} \\ w_{5,t} \end{bmatrix}, \ \mathbf{w}_t \sim \text{MVN}\left(0, \begin{bmatrix} q_1 & c_{1,2} & c_{1,3} & c_{1,4} & c_{1,5} \\ c_{1,2} & q_2 & c_{2,3} & c_{2,4} & c_{2,5} \\ c_{1,3} & c_{2,3} & q_3 & c_{3,4} & c_{3,5} \\ c_{1,4} & c_{2,4} & c_{3,4} & q_4 & c_{4,5} \\ c_{1,5} & c_{2,5} & c_{3,5} & c_{4,5} & q_5 \end{bmatrix}\right)$$

$$\begin{bmatrix} y_{1,t} \\ y_{2,t} \\ y_{3,t} \\ y_{4,t} \\ y_{5,t} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{1,t} \\ x_{2,t} \\ x_{3,t} \\ x_{4,t} \\ x_{5,t} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} v_{1,t} \\ v_{2,t} \\ v_{3,t} \\ v_{4,t} \\ v_{5,t} \end{bmatrix}, \ \mathbf{v}_t \sim \text{MVN}\left(0, \begin{bmatrix} r & 0 & 0 & 0 & 0 \\ 0 & r & 0 & 0 & 0 \\ 0 & 0 & r & 0 & 0 \\ 0 & 0 & 0 & r & 0 \\ 0 & 0 & 0 & 0 & r \end{bmatrix}\right)$$

$\mathbf{B}$ is not shown in the top equation; it is a $m \times m$ identity matrix. To fit, use `MARSS()` with the `model` argument set (output not shown).

```
kemfit = MARSS(dat, model = list(Q = "unconstrained"))
```

This shows one of the text shortcuts, `"unconstrained"`, which means estimate all elements in the matrix. This shortcut can be used for all parameter matrices.

3.2.3 Five equally correlated hidden state processes

This is the same model except that now there is only one process error variance and one process error covariance. Mathematically, the model is:

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \\ x_{3,t} \\ x_{4,t} \\ x_{5,t} \end{bmatrix} = \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \\ x_{3,t-1} \\ x_{4,t-1} \\ x_{5,t-1} \end{bmatrix} + \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \end{bmatrix} + \begin{bmatrix} w_{1,t} \\ w_{2,t} \\ w_{3,t} \\ w_{4,t} \\ w_{5,t} \end{bmatrix}, \; \mathbf{w}_t \sim \text{MVN}\left(0, \begin{bmatrix} q & c & c & c & c \\ c & q & c & c & c \\ c & c & q & c & c \\ c & c & c & q & c \\ c & c & c & c & q \end{bmatrix}\right)$$

$$\begin{bmatrix} y_{1,t} \\ y_{2,t} \\ y_{3,t} \\ y_{4,t} \\ y_{5,t} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{1,t} \\ x_{2,t} \\ x_{3,t} \\ x_{4,t} \\ x_{5,t} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} v_{1,t} \\ v_{2,t} \\ v_{3,t} \\ v_{4,t} \\ v_{5,t} \end{bmatrix}, \; \mathbf{v}_t \sim \text{MVN}\left(0, \begin{bmatrix} r & 0 & 0 & 0 & 0 \\ 0 & r & 0 & 0 & 0 \\ 0 & 0 & r & 0 & 0 \\ 0 & 0 & 0 & r & 0 \\ 0 & 0 & 0 & 0 & r \end{bmatrix}\right)$$

Again $\mathbf{B}$ is not shown in the top equation; it is a $m \times m$ identity matrix. To fit, use the following code (output not shown):

```
kemfit = MARSS(dat, model = list(Q = "equalvarcov"))
```

The shortcut ‘`"equalvarcov"` means one value on the diagonal and one on the off-diagonal. It can be used for all square matrices ($\mathbf{B}$,$\mathbf{Q}$,$\mathbf{R}$, and $\mathbf{\Lambda}$).

3.2.4 Five hidden state processes with a `north'' and asouth” u and Q elements

Here we fit a model with five independent hidden states where each observation time series is an independent observation of a different hidden trajectory

but the hidden trajectories 1-3 share their $\mathbf{u}$ and $\mathbf{Q}$ elements, while hidden trajectories 4-5 share theirs. This is the model:

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \\ x_{3,t} \\ x_{4,t} \\ x_{5,t} \end{bmatrix} = \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \\ x_{3,t-1} \\ x_{4,t-1} \\ x_{5,t-1} \end{bmatrix} + \begin{bmatrix} u_n \\ u_n \\ u_n \\ u_s \\ u_s \end{bmatrix} + \begin{bmatrix} w_{1,t} \\ w_{2,t} \\ w_{3,t} \\ w_{4,t} \\ w_{5,t} \end{bmatrix}, \ \mathbf{w}_t \sim \text{MVN}\left(0, \begin{bmatrix} q_n & 0 & 0 & 0 & 0 \\ 0 & q_n & 0 & 0 & 0 \\ 0 & 0 & q_n & 0 & 0 \\ 0 & 0 & 0 & q_s & 0 \\ 0 & 0 & 0 & 0 & q_s \end{bmatrix}\right)$$

$$\begin{bmatrix} y_{1,t} \\ y_{2,t} \\ y_{3,t} \\ y_{4,t} \\ y_{5,t} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{1,t} \\ x_{2,t} \\ x_{3,t} \\ x_{4,t} \\ x_{5,t} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} v_{1,t} \\ v_{2,t} \\ v_{3,t} \\ v_{4,t} \\ v_{5,t} \end{bmatrix}, \ \mathbf{v}_t \sim \text{MVN}\left(0, \begin{bmatrix} r & 0 & 0 & 0 & 0 \\ 0 & r & 0 & 0 & 0 \\ 0 & 0 & r & 0 & 0 \\ 0 & 0 & 0 & r & 0 \\ 0 & 0 & 0 & 0 & r \end{bmatrix}\right)$$

To fit use the following code, we specify the `model` argument for $\mathbf{u}$ and $\mathbf{Q}$ using list matrices. List matrices allow us to combine numeric and character values in a matrix. MARSS will interpret the numeric values as fixed, and the character values as parameters to be estimated. Parameters with the same name are constrained to be identical.

```
regions = list("N", "N", "N", "S", "S")
U = matrix(regions, 5, 1)
Q = matrix(list(0), 5, 5)
diag(Q) = regions
kemfit = MARSS(dat, model = list(U = U, Q = Q))
```

Only $\mathbf{u}$ and $\mathbf{Q}$ need to be specified since the other parameters are at their default values.

3.2.5 Fixed observation error variance

Here we fit the same model but with a known observation error variance. This is the model:

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \\ x_{3,t} \\ x_{4,t} \\ x_{5,t} \end{bmatrix} = \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \\ x_{3,t-1} \\ x_{4,t-1} \\ x_{5,t-1} \end{bmatrix} + \begin{bmatrix} u_n \\ u_n \\ u_n \\ u_s \\ u_s \end{bmatrix} + \begin{bmatrix} w_{1,t} \\ w_{2,t} \\ w_{3,t} \\ w_{4,t} \\ w_{5,t} \end{bmatrix}, \; \mathbf{w}_t \sim \text{MVN}\left(0, \begin{bmatrix} q_n & 0 & 0 & 0 & 0 \\ 0 & q_n & 0 & 0 & 0 \\ 0 & 0 & q_n & 0 & 0 \\ 0 & 0 & 0 & q_s & 0 \\ 0 & 0 & 0 & 0 & q_s \end{bmatrix}\right)$$

$$\begin{bmatrix} y_{1,t} \\ y_{2,t} \\ y_{3,t} \\ y_{4,t} \\ y_{5,t} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{1,t} \\ x_{2,t} \\ x_{3,t} \\ x_{4,t} \\ x_{5,t} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} v_{1,t} \\ v_{2,t} \\ v_{3,t} \\ v_{4,t} \\ v_{5,t} \end{bmatrix},$$

$$\mathbf{v}_t \sim \text{MVN}\left(0, \begin{bmatrix} 0.01 & 0 & 0 & 0 & 0 \\ 0 & 0.01 & 0 & 0 & 0 \\ 0 & 0 & 0.01 & 0 & 0 \\ 0 & 0 & 0 & 0.01 & 0 \\ 0 & 0 & 0 & 0 & 0.01 \end{bmatrix}\right)$$

To fit this model, use the following code (output not shown):

```
regions = list("N", "N", "N", "S", "S")
U = matrix(regions, 5, 1)
Q = matrix(list(0), 5, 5)
diag(Q) = regions
R = diag(0.01, 5)
kemfit = MARSS(dat, model = list(U = U, Q = Q, R = R))
```

3.2.6 One hidden state and five i.i.d. observation time series

Instead of five hidden state trajectories, we specify that there is only one and all the observations are of that one trajectory. Mathematically, the model is:

$$x_t = x_{t-1} + u + w_t,\ w_t \sim \mathrm{N}(0, q)$$

$$\begin{bmatrix} y_{1,t} \\ y_{2,t} \\ y_{3,t} \\ y_{4,t} \\ y_{5,t} \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} x_t + \begin{bmatrix} 0 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \end{bmatrix} + \begin{bmatrix} v_{1,t} \\ v_{2,t} \\ v_{3,t} \\ v_{4,t} \\ v_{5,t} \end{bmatrix},\ \mathbf{v}_t \sim \mathrm{MVN}\left(0, \begin{bmatrix} r & 0 & 0 & 0 & 0 \\ 0 & r & 0 & 0 & 0 \\ 0 & 0 & r & 0 & 0 \\ 0 & 0 & 0 & r & 0 \\ 0 & 0 & 0 & 0 & r \end{bmatrix}\right)$$

Note the default model for **R** is `"diagonal and equal"`' so we can leave this off when specifying the `model` argument. To fit, use this code (output not shown):

```
Z = factor(c(1, 1, 1, 1, 1))
kemfit = MARSS(dat, model = list(Z = Z))
```

```
Success! abstol and log-log tests passed at 28 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 28 iterations.
Log-likelihood: 3.593276
AIC: 8.813447   AICc: 11.13603

        Estimate
A.SJI    0.80153
A.EBays  0.28245
A.PSnd  -0.54802
A.HC    -0.62665
```

```
R.diag   0.04523
U.U      0.04759
Q.Q      0.00429
x0.x0    6.39199
Initial states (x0) defined at t=0

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

You can also pass in **Z** exactly as it is in the equation: `Z=matrix(1,5,2)`, but the factor shorthand is handy if you need to assign different observed time series to different underlying state time series (see next examples). The default **a** form is `"scaling"`, which means that the first **y** row associated with a given x has $a = 0$ and the rest are estimated.

3.2.7 One hidden state and five independent observation time series with different variances

Mathematically, this model is:

$$x_t = x_{t-1} + u + w_t, \; w_t \sim \mathrm{N}(0, q)$$

$$\begin{bmatrix} y_{1,t} \\ y_{2,t} \\ y_{3,t} \\ y_{4,t} \\ y_{5,t} \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} x_t + \begin{bmatrix} 0 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \end{bmatrix} + \begin{bmatrix} v_{1,t} \\ v_{2,t} \\ v_{3,t} \\ v_{4,t} \\ v_{5,t} \end{bmatrix}, \; \mathbf{v}_t \sim \mathrm{MVN}\left(0, \begin{bmatrix} r_1 & 0 & 0 & 0 & 0 \\ 0 & r_2 & 0 & 0 & 0 \\ 0 & 0 & r_3 & 0 & 0 \\ 0 & 0 & 0 & r_4 & 0 \\ 0 & 0 & 0 & 0 & r_5 \end{bmatrix}\right)$$

To fit this model:

```
Z = factor(c(1, 1, 1, 1, 1))
R = "diagonal and unequal"
kemfit = MARSS(dat, model = list(Z = Z, R = R))
```

```
Success! abstol and log-log tests passed at 24 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 24 iterations.
Log-likelihood: 16.66199
AIC: -9.323982   AICc: -3.944671

                Estimate
A.SJI            0.79555
A.EBays          0.27540
A.PSnd          -0.53694
A.HC            -0.60874
R.(SJF,SJF)      0.03229
R.(SJI,SJI)      0.03528
R.(EBays,EBays)  0.01352
R.(PSnd,PSnd)    0.01082
R.(HC,HC)        0.19609
U.U              0.05270
Q.Q              0.00604
x0.x0            6.26676
Initial states (x0) defined at t=0

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

3.2.8 Two hidden state processes

Here we fit a model with two hidden states (north and south) where observation time series 1-3 are for the north and 4-5 are for the south. We make the hidden state processes independent (meaning a diagonal $\mathbf{Q}$ matrix) but with the same process variance. We make the observation errors i.i.d. (the

default) and the **u** elements equal. Mathematically, this is the model:

$$\begin{bmatrix} x_{n,t} \\ x_{s,t} \end{bmatrix} = \begin{bmatrix} x_{n,t-1} \\ x_{s,t-1} \end{bmatrix} + \begin{bmatrix} u \\ u \end{bmatrix} + \begin{bmatrix} w_{n,t} \\ w_{s,t} \end{bmatrix}, \ \mathbf{w}_t \sim \text{MVN}\left(0, \begin{bmatrix} q & 0 \\ 0 & q \end{bmatrix}\right)$$

$$\begin{bmatrix} y_{1,t} \\ y_{2,t} \\ y_{3,t} \\ y_{4,t} \\ y_{5,t} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_{n,t} \\ x_{s,t} \end{bmatrix} + \begin{bmatrix} 0 \\ a_2 \\ a_3 \\ 0 \\ a_5 \end{bmatrix} + \begin{bmatrix} v_{1,t} \\ v_{2,t} \\ v_{3,t} \\ v_{4,t} \\ v_{5,t} \end{bmatrix}, \ \mathbf{v}_t \sim \text{MVN}\left(0, \begin{bmatrix} r & 0 & 0 & 0 & 0 \\ 0 & r & 0 & 0 & 0 \\ 0 & 0 & r & 0 & 0 \\ 0 & 0 & 0 & r & 0 \\ 0 & 0 & 0 & 0 & r \end{bmatrix}\right)$$

To fit the model, use the following code (output not shown):

```
Z = factor(c("N", "N", "N", "S", "S"))
Q = "diagonal and equal"
U = "equal"
kemfit = MARSS(dat, model = list(Z = Z, Q = Q, U = U))
```

You can also pass in **Z** exactly as it is in the equation as a numeric matrix `Z=matrix(c(1,1,1,0,0,0,0,0,1,1),5,2)`; the `factor` notation is a shortcut for making a design matrix (as **Z** is in these examples. `"equal"` is a shortcut meaning all elements in a matrix are constrained to be equal. It can be used for all column matrices (**a**, **u** and $\boldsymbol{\pi}$). `"diagonal and equal"` can be used as a shortcut for all square matrices (**B**,**Q**,**R**, and $\boldsymbol{\Lambda}$).

3.3 Time-varying parameters

Time-varying parameters are specified by passing in an array of matrices (list, numeric or character) where the 3rd dimension of the array is time and must be the same value as the 2nd (time) dimension of the data matrix. No text shortcuts are allowed for time-varying parameters; you need to use the matrix form.

For example, let's say we wanted a different **u** for the first half versus second half of the harbor seal time series. We would pass in an array for **u** as follows:

```
U1 = matrix("t1", 5, 1)
U2 = matrix("t2", 5, 1)
Ut = array(U2, dim = c(dim(U1), dim(dat)[2]))
TT = dim(dat)[2]
Ut[, , 1:floor(TT/2)] = U1
kemfit.tv = MARSS(dat, model = list(U = Ut, Q = "diagonal and equal"))
```

You can have some elements in a parameter matrix be time-constant and some be time-varying:

```
U1 = matrix(c(rep("t1", 4), "hc"), 5, 1)
U2 = matrix(c(rep("t2", 4), "hc"), 5, 1)
Ut = array(U2, dim = c(dim(U1), dim(dat)[2]))
Ut[, , 1:floor(TT/2)] = U1
kemfit.tv = MARSS(dat, model = list(U = Ut, Q = "diagonal and equal"))
```

Note that how the time-varying model is specified for MARSS is the same as you would write the time-varying model on paper in matrix math form.

3.4 Including inputs (or covariates)

In MARSS models with covariates, the covariates are often treated as inputs and appear as either the $\mathbf{c}$ or $\mathbf{d}$ in Equation @ref(eq:marss.qe), depending on the application. However, more generally, $\mathbf{c}$ and $\mathbf{d}$ are simply inputs that are fully-known (no missing values). $\mathbf{c}_t$ is the $p \times 1$ vector of inputs at time t which affect the states and $\mathbf{d}_t$ is a $q \times 1$ vector of inputs (potentially the same as $\mathbf{c}_t$), which affect the observations.

$\mathbf{C}_t$ is an $m \times p$ matrix of coefficients relating the effects of $\mathbf{c}_t$ to the $m \times 1$ state vector $\mathbf{x}_t$, and $\mathbf{D}_t$ is an $n \times q$ matrix of coefficients relating the effects of $\mathbf{d}_t$ to the $n \times 1$ observation vector $\mathbf{y}_t$. The elements of $\mathbf{C}$ and $\mathbf{D}$ can be estimated, and their form is specified much like the other matrices.

With the `MARSS()` function, one can fit a model with inputs by simply passing in `model$c` and/or `model$d` in the MARSS() call as a $p \times T$ or $q \times T$ matrix, respectively. The form for $\mathbf{C}_t$ and $\mathbf{D}_t$ is similarly specified by passing in `model$C` and/or `model$D`. If $\mathbf{C}$ and $\mathbf{D}$ are not time-varying, they are passed

in as a 2-dimensional matrix. If they are time-varying, they must be passed in as an 3-dimensional array with the 3rd dimension equal to the number of time steps if they are time-varying.

See Chapter **??** for extended examples of including covariates as inputs in a MARSS model. Also note that it is not necessary to have your covariates appear in **c** and/or **d**. That is a common form, however in some MARSS models, covariates will appear in one of the parameter matrices as fixed values.

3.5 Printing and summarizing models and model fits

The package includes print functions for marssMODEL objects and marssMLE objects (fitted models).

```
print(kemfit)
print(kemfit$model)
```

This will print the basic information on model structure and model fit that you have seen in the previous examples. The package also includes a summary function for models.

```
summary(kemfit$model)
```

Output for the summary function is not shown because it is verbose. It prints each matrix with the fixed elements denoted with their values and the free elements denoted by their names. This is very helpful for confirming exactly what model structure you are fitting to the data.

The print function will also print various other types of output such as a vector of the estimated parameters, the estimated states, the state standard errors, etc. You use the `what` argument in the print call to specify the desired output.

```
print(kemfit, what = "par")
```

will print the `par` element of a marssMLE object. This will only include the estimated elements in a column matrix. To see the entire parameter matrix

with both estimated and fixed values, you can use:

```
print(kemfit, what = "Q")
```

```
Parameter matrix Q
             [,1]        [,2]
[1,] 0.007669608 0.000000000
[2,] 0.000000000 0.007669608
```

Type `?print.MARSS` to see a list of the types of output that can be printed with a `print` call. If you want to use the output from print instead of printing to the console, then assign the print call to a value:

```
x = print(kemfit, what = "states", silent = TRUE)
```

3.6 Tidy output

The `augment`, `tidy` and `glance` functions from the broom package will provide summaries as a data.frame for use in further analyses and for passing to `ggplot()`. See `?augment.marssMLE` for examples.

```
require(broom)
head(augment(kemfit))
```

```
  .rownames t        y  .fitted   .se.fit      .resids .std.resid
1       SJF 1 6.033086 6.215483 0.1729465 -0.182397213 -1.0724758
2       SJF 2       NA 6.329702 0.2079884  0.000000000         NA
3       SJF 3       NA 6.443921 0.2142727  0.000000000         NA
4       SJF 4       NA 6.558140 0.2148820  0.000000000         NA
5       SJF 5       NA 6.672359 0.2098656  0.000000000         NA
6       SJF 6 6.783325 6.786578 0.1691246 -0.003253081  0.4048375
```

```
tidy(kemfit)
```

```
     term     estimate   std.error     conf.low  conf.high
1   A.SJI  0.797864531 0.061519575  0.677288381 0.91844068
2 A.EBays  0.277434738 0.062534476  0.154869417 0.40000006
3    A.HC -0.070350207 0.088815035 -0.244424477 0.10372406
4  R.diag  0.034061922 0.006712493  0.020905678 0.04721817
```

```
5     U.1  0.043176408 0.014380922  0.014990318 0.07136250
6  Q.diag  0.007669608 0.004603993 -0.001354052 0.01669327
7    x0.N  6.172047633 0.137758419  5.902046093 6.44204917
8    x0.S  6.206154697 0.157084082  5.898275554 6.51403384
```

```
glance(kemfit)
```

```
   coef.det      sigma df   logLik       AIC     AICc convergence errors
1 0.9186375 0.03013659  8 7.949236 0.1015284 2.424109           0      0
```

3.7 Confidence intervals on a fitted model

The function `MARSSparamCIs()` is used to compute confidence intervals with a default alpha level of 0.05. The default is to compute approximate confidence intervals using a the Hessian matrix (`method="hessian"`). Confidence intervals can also be computed via parametric (`method="parametric"`) or non-parametric (`method="innovations"`) bootstrapping. Note, if you want confidence intervals on variances, then it is unwise to use the Hessian approximation as it is symmetric and variances are constrained to be positive.

3.7.1 Approximate confidence intervals from a Hessian matrix

The default method for `MARSSparamCIs` computes approximate confidence intervals using an analytically computed Hessian matrix **?**, sec 3.4.5. The call is:

```
kem.with.hess.CIs = MARSSparamCIs(kemfit)
```

See `?MARSShessian` for a discussion of the Hessian calculations. Use `print` or just type the marssMLE object name to see the confidence intervals:

```
print(kem.with.hess.CIs)
```

```
MARSS fit is
Estimation method: kem
```

```
Convergence test: conv.test.slope.tol = 0.5, abstol = 0.001
Estimation converged in 22 iterations.
Log-likelihood: 7.949236
AIC: 0.1015284   AICc: 2.424109

          ML.Est Std.Err   low.CI  up.CI
A.SJI    0.79786 0.06152  0.67729 0.9184
A.EBays  0.27743 0.06253  0.15487 0.4000
A.HC    -0.07035 0.08882 -0.24442 0.1037
R.diag   0.03406 0.00671  0.02091 0.0472
U.1      0.04318 0.01438  0.01499 0.0714
Q.diag   0.00767 0.00460 -0.00135 0.0167
x0.N     6.17205 0.13776  5.90205 6.4420
x0.S     6.20615 0.15708  5.89828 6.5140
Initial states (x0) defined at t=0

CIs calculated at alpha = 0.05 via method=hessian
```

3.7.2 Confidence intervals from a parametric bootstrap

Use `method="parametric"` to use a parametric bootstrap to compute confidence intervals and bias using a parametric bootstrap.

```
kem.w.boot.CIs = MARSSparamCIs(kemfit, method = "parametric",
    nboot = 10)
# nboot should be more like 1000, but set low for example's
# sake
print(kem.w.boot.CIs)
```

```
MARSS fit is
Estimation method: kem
Convergence test: conv.test.slope.tol = 0.5, abstol = 0.001
Estimation converged in 22 iterations.
Log-likelihood: 7.949236
AIC: 0.1015284   AICc: 2.424109
```

```
          ML.Est Std.Err   low.CI  up.CI Est.Bias Unbias.Est
A.SJI    0.79786 0.06791  0.69927 0.9028 -0.00841     0.7895
A.EBays  0.27743 0.02764  0.21968 0.2945  0.02008     0.2975
A.HC    -0.07035 0.10225 -0.21061 0.1082 -0.01040    -0.0808
R.diag   0.03406 0.00678  0.02430 0.0425  0.00182     0.0359
U.1      0.04318 0.02390  0.00441 0.0825 -0.00112     0.0421
Q.diag   0.00767 0.00421  0.00000 0.0131  0.00274     0.0104
x0.N     6.17205 0.24154  5.83794 6.5124 -0.01888     6.1532
x0.S     6.20615 0.36187  5.76468 6.8899 -0.05196     6.1542
Initial states (x0) defined at t=0

CIs calculated at alpha = 0.05 via method=parametric
Bias calculated via  bootstrapping with  bootstraps.
```

3.8 Vectors of just the estimated parameters

Often it is useful to have a vector of the estimated parameters. For example, if you are writing a call to `optim`, you will need a vector of just the estimated parameters. You can use the function `coef`:

```
parvec = coef(kemfit, type = "vector")
parvec
```

```
      A.SJI      A.EBays         A.HC       R.diag          U.1
0.797864531  0.277434738 -0.070350207  0.034061922  0.043176408
     Q.diag         x0.N         x0.S
0.007669608  6.172047633  6.206154697
```

3.9 Kalman filter and smoother output

All the standard Kalman filter and smoother output (along with the lag-one covariance smoother output) is available using the `MARSSkf` function. Read the help file (`?MARSSkf`) for details and meanings of the names in the output list.

```
kf = MARSSkf(kemfit)
names(kf)
```

```
 [1] "xtT"         "VtT"         "Vtt1T"       "x0T"         "V0T"
 [6] "x10T"        "V10T"        "x00T"        "V00T"        "Vtt"
[11] "Vtt1"        "J"           "J0"          "Kt"          "xtt1"
[16] "xtt"         "Innov"       "Sigma"       "kfas.model" "logLik"
[21] "ok"          "errors"
```

```
# if you only need the logLik, this is the fast way to get it
MARSSkf(kemfit, only.logLik = TRUE)
```

```
$logLik
[1] 7.949236
```

3.10 Degenerate variance estimates

If your data are short relative to the number of parameters you are estimating, then you are liable to find that some of the variance elements are degenerate (equal to zero). Try the following:

```
dat.short = dat[1:4, 1:10]
kem.degen = MARSS(dat.short, control = list(allow.degen = FALSE))
```

```
Warning! Abstol convergence only. Maxit (=500) reached before log-log convergence.

MARSS fit is
Estimation method: kem
Convergence test: conv.test.slope.tol = 0.5, abstol = 0.001
WARNING: Abstol convergence only no log-log convergence.
 maxit (=500) reached before log-log convergence.
 The likelihood and params might not be at the ML values.
 Try setting control$maxit higher.
Log-likelihood: 11.67854
AIC: 2.642914   AICc: 63.30958

                    Estimate
```

```
R.diag               1.22e-02
U.X.SJF              9.79e-02
U.X.SJI              1.09e-01
U.X.EBays            9.28e-02
U.X.PSnd             1.11e-01
Q.(X.SJF,X.SJF)      1.89e-02
Q.(X.SJI,X.SJI)      1.03e-05
Q.(X.EBays,X.EBays) 8.24e-06
Q.(X.PSnd,X.PSnd)    3.05e-05
x0.X.SJF             5.96e+00
x0.X.SJI             6.73e+00
x0.X.EBays           6.60e+00
x0.X.PSnd            5.71e+00
Initial states (x0) defined at t=0

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

Convergence warnings
 Warning: the  Q.(X.SJI,X.SJI)  parameter value has not converged.
 Warning: the  Q.(X.EBays,X.EBays)  parameter value has not converged.
 Warning: the  Q.(X.PSnd,X.PSnd)  parameter value has not converged.
 Type MARSSinfo("convergence") for more info on this warning.
```

This will print a warning that the maximum number of iterations was reached before convergence of some of the **Q** parameters. It might be that if you just ran a few more iterations the variances will converge. So first try setting `control$maxit` higher.

```
kem.degen2 = MARSS(dat.short, control = list(maxit = 1000, allow.degen = FALSE
    silent = 2)
```

Output not shown, but if you run the code, you will see that some of the **Q** terms are still not converging. MARSS can detect if a variance is going to zero and it will try zero to see if that has a higher likelihood. Try removing the `allow.degen=FALSE` which was turning off this feature.

```
kem.short = MARSS(dat.short)
```

```
Warning! Abstol convergence only. Maxit (=500) reached before log-log converge
```

```
MARSS fit is
Estimation method: kem
Convergence test: conv.test.slope.tol = 0.5, abstol = 0.001
WARNING: Abstol convergence only no log-log convergence.
 maxit (=500) reached before log-log convergence.
 The likelihood and params might not be at the ML values.
 Try setting control$maxit higher.
Log-likelihood: 11.6907
AIC: 2.6186   AICc: 63.28527

                    Estimate
R.diag              1.22e-02
U.X.SJF             9.79e-02
U.X.SJI             1.09e-01
U.X.EBays           9.24e-02
U.X.PSnd            1.11e-01
Q.(X.SJF,X.SJF)     1.89e-02
Q.(X.SJI,X.SJI)     1.03e-05
Q.(X.EBays,X.EBays) 0.00e+00
Q.(X.PSnd,X.PSnd)   3.04e-05
x0.X.SJF            5.96e+00
x0.X.SJI            6.73e+00
x0.X.EBays          6.60e+00
x0.X.PSnd           5.71e+00
Initial states (x0) defined at t=0

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

Convergence warnings
 Warning: the  Q.(X.SJI,X.SJI)  parameter value has not converged.
 Warning: the  Q.(X.PSnd,X.PSnd)  parameter value has not converged.
 Type MARSSinfo("convergence") for more info on this warning.
```

So three of the four $\mathbf{Q}$ elements are going to zero. This often happens when you do not have enough data to estimate both observation and process variance.

Perhaps we are trying to estimate too many variances. We can try using only one variance value in $\mathbf{Q}$ and one u value in $\mathbf{u}$:

```
kem.small = MARSS(dat.short, model = list(Q = "diagonal and equal",
    U = "equal"))
```

```
Success! abstol and log-log tests passed at 164 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 164 iterations.
Log-likelihood: 11.19
AIC: -8.379994   AICc: 0.9533396

           Estimate
R.diag       0.0191
U.1          0.1027
Q.diag       0.0000
x0.X.SJF     6.0609
x0.X.SJI     6.7698
x0.X.EBays   6.5307
x0.X.PSnd    5.7451
Initial states (x0) defined at t=0

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

No, there are simply not enough data to estimate both process and observation variances.

3.11 Bootstrap parameter estimates

You can easily produce bootstrap parameter estimates from a fitted model using `MARSSboot()`:

```
boot.params = MARSSboot(kemfit, nboot = 20, output = "parameters",
    sim = "parametric")$boot.params
```

```
          |2%       |20%       |40%       |60%       |80%       |100%
Progress: ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
```

Use `silent=TRUE` to stop the progress bar from printing. The function will also produce parameter sets generated using a Hessian matrix (`sim="hessian"`) or a non-parametric bootstrap (`sim="innovations"`).

3.12 Data simulation

3.12.1 Simulated data from a fitted MARSS model

Data can be simulated from marssMLE object using `MARSSsimulate()`.

```
sim.data = MARSSsimulate(kemfit, nsim = 2, tSteps = 100)$sim.data
```

Then you might want to estimate parameters from that simulated data. Above we created two simulated datasets (`nsim=2`). We will fit to the first one. Here the default settings for `MARSS()` are used.

```
kem.sim.1 = MARSS(sim.data[, , 1])
```

Then we might like to see the likelihood of the second set of simulated data under the model fit to the first set of data. We do that with the Kalman filter function. This function takes a marssMLE object (as output by say the MARSS function), and we have to replace the data in kem.sim.1 with the second set of simulated data.

```
kem.sim.2 = kem.sim.1
kem.sim.2$model$data = sim.data[, , 2]
MARSSkf(kem.sim.2)$logLik
```

```
[1] 7.964597
```

3.13 Bootstrap AIC

The function `MARSSaic()` computes a bootstrap AIC for model selection purposes. Use `output="AICbp"` to produce a parameter bootstrap. Use `output="AICbb"` to produce a non-parametric bootstrap AIC. You will need a large number of bootstraps (`nboot`). We use only 10 bootstraps to show you how to compute AICb with the MARSS package, but the AICbp estimate will be terrible with this few bootstraps.

```
kemfit.with.AICb = MARSSaic(kemfit, output = "AICbp", Options = list(nboot = 1
    silent = TRUE))
# nboot should be more like 1000, but set low here for
# example sake

print(kemfit.with.AICb)
```

```
MARSS fit is
Estimation method: kem
Convergence test: conv.test.slope.tol = 0.5, abstol = 0.001
Estimation converged in 22 iterations.
Log-likelihood: 7.949236
AIC: 0.1015284   AICc: 2.424109   AICbp(param): 270.463

        Estimate
A.SJI    0.79786
A.EBays  0.27743
A.HC    -0.07035
R.diag   0.03406
U.1      0.04318
Q.diag   0.00767
x0.N     6.17205
x0.S     6.20615
Initial states (x0) defined at t=0

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

3.14 Convergence

MARSS uses two convergence tests. The first is

$$\text{logLik}_{i+1} - \text{logLik}_i < \text{tol}$$

This is called `abstol` (meaning absolute tolerance) in the output. The second is called the `conv.test.slope`. This looks at the slope of the log parameter value (or likelihood) versus log iteration number and asks whether that is close to zero (not changing).

If you are having trouble getting the model to converge, then start by addressing the following 1) Are you trying to fit a bad model, e.g. a non-stationary model fit to stationary data or the opposite or a model that specifies independence of errors or states to data that clearly violate that or a model that implies a particular stationary distribution (particular mean and variance) to data that strongly violate that? 2) Do you have confounded parameters, e.g. two parameters that have the same effect (like effectively two intercepts)?, 3) Are you trying to fit a model to 1 data point somewhere, e.g. in a big multivariate dataset with lots of missing values? 4) How many parameters are you trying to estimate per data point? 5) Check your residuals (`residuals(kemfit)$model.residuals`) for normality. 6) Did you do any data transformations that would cause one of the variances to go to zero? Replacing 0s with a constant will do that. Try replacing them with NAs (missing). Do you have long strings of constant numbers in your data? Binned data often look like that, and that will drive $\mathbf{Q}$ to 0.

Chapter 4

Getting your data in right format

Your data need to be a matrix (not dataframe nor a ts object) with time across the columns ($n \times T$ matrix). The MARSS functions assume discrete time steps and you will need a column for each time step. Replace any missing time steps with NA. Write your model down on paper and identify which parameters correspond to which parameter matrices in Equation 1.1. Call the `MARSS()` function (Chapter **??**) using your data and using the `model` argument to specify the structure of each parameter.

A R **ts** object (time series object) stores information about the time steps of the data and often seasonal information (the quarter or month). MARSS needs this information in matrix form. If you have your data in **ts** form, then you are probably using year and season (quarter, month) as covariates to estimate trend and seasonality. Here is how to get your **ts** into the form that MARSS wants.

Here is how to get your ts into the form that MARSS wants.

4.1 Univariate example

This converts a univariate ts object with year and quarter into a matrix with a row for the response (here called `Temp`), year, and quarter.

```
z = ts(rnorm(10), frequency = 4, start = c(1959, 2))
dat = data.frame(Yr = floor(time(z) + .Machine$double.eps),
      Qtr = cycle(z), Temp=z)
dat = t(dat)
```

When you call MARSS, `dat["Temp",]` is the data. `dat[c("Yr","Qtr"),]` are your covariates.

4.2 Multivariate example

In this example, we have two temperature readings and a salinity reading. The data are monthly.

```
z <- ts(matrix(rnorm(300), 100, 3), start = c(1961, 1),
     frequency = 12, names=c("Temp1","Temp2","Sal"))
dat = data.frame(Yr = floor(time(z) + .Machine$double.eps),
     Month = cycle(z), z)
```

When you call MARSS, `dat[c("Temp1","Temp2"),]` are the data and `dat[c("Yr","Month","Sal"),]` are your covariates.

See the chapters that discuss seasonality for examples of how to model seasonality. The brute force method of treating month or quarter as a factor requires estimation of more parameters than necessary in many cases.

4.3 Important notes about the algorithms

Specification of a properly constrained model with a unique solution is the responsibility of the user because MARSS has no way to tell if you have specified an insufficiently constrained model—with correspondingly an infinite number of solutions. How do you know if the model is properly constrained? If you are using a MARSS model form that is widely used, then you can probably assume that it is properly constrained. If you go to papers where someone developed the model or method, the issue of constraints necessary to ensure "identifiability" will likely be addressed if it is an issue. Are you fitting

novel MARSS models? Then you will need to do some study on identifiability in this class of models using textbooks (Appendix **??**). Often textbooks do not address identifiability explicitly. Rather it is addressed implicitly by only showing a model constructed in such a way that it is identifiable. In our work, if we suspect identification problems, we will often first do a Bayesian analysis with flat priors and look for oddities in the posteriors, such as ridges, plateaus or bimodality.

All the EM code in the MARSS package is currently in native ,**R**. Thus the model fitting is slow. The classic Kalman filter/smoother algorithm, as shown in **?**, p. 331-335, is based on the original smoother presented in **?**. This Kalman filter is provided in function `MARSSkfss()`, but the default Kalman filter and smoother used in the MARSS package is based on the algorithm in **?** and papers by Koopman et al. This Kalman filter and smoother is provided in the **KFAS** package (Helske 2012). Table 2 in **?** indicates that the classic algorithm is 40-100 times slower than the algorithm given in **?**, **?**, and **?**. The MARSS package function `MARSSkfas()` provides a translator between the model objects in MARSS and those in **KFAS** so that the **KFAS** functions can be used. `MARSSkfas()` also includes a lag-one covariance smoother algorithm as this is not output by the **KFAS** functions, and it provides proper formulation of the priors so that one can use the **KFAS** functions when the prior on the states is set at $t = 0$ instead of $t = 1$. Simply off-setting your data to start at t=2 and sending that value to $t_{init} = 1$ in the **KFAS** Kalman filter would not be mathematically correct!

EM algorithms will quickly get in the vicinity of the maximum likelihood, but the final approach to the maximum is generally slow relative to quasi-Newton methods. On the flip side, EM algorithms are quite robust to initial conditions choices and can be extremely fast at getting close to the MLE values for high-dimensional models. The **MARSS** package also allows one to use the BFGS method to fit MARSS models, thus one can use an EM algorithm to "get close" and then the BFGS algorithm to polish off the estimate. Restricted maximum-likelihood algorithms are also available for AR(1) state-space models, both univariate (**?**) and multivariate (**?**). REML can give parameter estimates with lower variance than plain maximum-likelihood algorithms. However, the algorithms for REML when there are missing values are not currently available (although that will probably change in the near future). Another maximum-likelihood method is data-cloning which adapts MCMC algorithms used in Bayesian analysis for maximum-likelihood estima-

tion (**?**).

Missing values are seamlessly accommodated with the **MARSS** package. Simply specify missing data with NAs. The likelihood computations are exact and will deal appropriately with missing values. However, no innovations, referring to the non-parametric bootstrap developed by Stoffer and Wall (1991), bootstrapping can be done if there are missing values. Instead parametric bootstrapping must be used.

You should be aware that maximum-likelihood estimates of variance in MARSS models are fundamentally biased, regardless of the algorithm used. This bias is more severe when one or the other of $\mathbf{R}$ or $\mathbf{Q}$ is very small, and the bias does not go to zero as sample size goes to infinity. The bias arises because variance is constrained to be positive. Thus if $\mathbf{R}$ or $\mathbf{Q}$ is essentially zero, the mean estimate will not be zero and thus the estimate will be biased high while the corresponding bias of the other variance will be biased low. You can generate unbiased variance estimates using a bootstrap estimate of the bias. The function `MARSSparamCIs()` will do this. However be aware that adding an estimated bias to a parameter estimate will lead to an increase in the variance of your parameter estimate. The amount of variance added will depend on sample size.

You should also be aware that mis-specification of the prior on the initial states ($\boldsymbol{\pi}$ and $\boldsymbol{\Lambda}$) can have catastrophic effects on your parameter estimates if your prior conflicts with the distribution of the initial states implied by the MARSS model. These effects can be very difficult to detect because the model will appear to be well-fitted. Unless you have a good idea of what the parameters should be, you might not realize that your prior conflicts.

The most common problems we have found with priors on $\mathbf{x}_0$ are the following. Problem 1) The correlation structure in $\boldsymbol{\Lambda}$ (whether the prior is diffuse or not) does not match the correlation structure in $\mathbf{x}_0$ implied by your model. For example, you specify a diagonal $\boldsymbol{\Lambda}$ (independent states), but the implied distribution has correlations. Problem 2) The correlation structure in $\boldsymbol{\Lambda}$ does not match the structure in $\mathbf{x}_0$ implied by constraints you placed on $\boldsymbol{\pi}$. For example, you specify that all values in $\boldsymbol{\pi}$ are shared, yet you specify that $\boldsymbol{\Lambda}$ is diagonal (independent).

Unfortunately, using a diffuse prior does not help with these two problems because the diffuse prior still has a correlation structure and can still conflict

with the implied correlation in $\mathbf{x}_0$. One way to get around these problems is to set $\mathbf{\Lambda}$=0 (a $m \times m$ matrix of zeros) and estimate $\boldsymbol{\pi} \equiv \mathbf{x}_0$ only. Now $\boldsymbol{\pi}$ is a fixed but unknown (estimated) parameter, not the mean of a distribution. In this case, $\mathbf{\Lambda}$ does not exist in your model and there is no conflict with the model.
Be aware however that estimating $\boldsymbol{\pi}$ as a parameter is not always robust. If you specify that $\mathbf{\Lambda}$=0 and specify that $\boldsymbol{\pi}$ corresponds to $\mathbf{x}_0$, but your model "explodes" when run backwards in time, you cannot estimate $\boldsymbol{\pi}$ because you cannot get a good estimate of $\mathbf{x}_0$. Sometimes this can be avoided by specifying that $\boldsymbol{\pi}$ corresponds to $\mathbf{x}_1$ so that it can be constrained by the data $\mathbf{y}_1$.

In summary, if the implied correlation structure of your initial states is independent (diagonal variance-covariance matrix), you should generally be ok with a diagonal and high variance prior or with treating the initial states as parameters (with $\mathbf{\Lambda} = 0$). But if your initial states have an implied correlation structure that is not independent, then proceed with caution. 'With caution' means that you should assume you have problems and test how your model fits with simulated data.

4.4 State-space form of ARMA(p,q) models

There is a large class of models in the statistical finance literature that have the form

$$\begin{aligned}\mathbf{x}_{t+1} &= \mathbf{B}\mathbf{x}_t + \mathbf{\Gamma}\boldsymbol{\eta}_t \\ \mathbf{y}_t &= \mathbf{Z}\mathbf{x}_t + \boldsymbol{\eta}_t\end{aligned}$$

For example, ARMA(p,q) models can be written in this form. The MARSS model framework in this package will not allow you to write models in that form. You can put the $\boldsymbol{\eta}_t$ into the $\mathbf{x}_t$ vector and set $\mathbf{R} = 0$ to make models of this form using the MARSS form, but the EM algorithm in the MARSS package won't let you estimate parameters because the parameters will drop out of the full likelihood being maximized in the algorithm. You can try using BFGS by passing in the `method` argument to the MARSS() call.

4.5 Troubleshooting

Numerical errors due to ill-conditioned matrices are not uncommon when fitting MARSS models. The Kalman and EM algorithms need inverses of matrices. If those matrices become ill-conditioned, for example all elements are close to the same value, then the algorithm becomes unstable. Warning messages will be printed if the algorithms are becoming unstable and you can set `control$trace=1`, to see details of where the algorithm is becoming unstable. Whenever possible, you should avoid using shared $\boldsymbol{\pi}$ values in your model[1]. The way our algorithm deals with $\boldsymbol{\Lambda}$ tends to make this case unstable, especially if $\mathbf{R}$ is not diagonal. In general, estimation of a non-diagonal $\mathbf{R}$ is more difficult, more prone to ill-conditioning, and more data-hungry.

You may also see non-convergence warnings, especially if your MLE model turns out to be degenerate. This means that one of the elements on the diagonal of your $\mathbf{Q}$ or $\mathbf{R}$ matrix are going to zero (are degenerate). It will take the EM algorithm forever to get to zero. BFGS will have the same problem, although it will often get a bit closer to the degenerate solution. If you are using `method="kem"`, MARSS will warn you if it looks like the solution is degenerate. If you use `control=list(allow.degen=TRUE)`, the EM algorithm will attempt to set the degenerate variances to zero (instead of trying to get to zero using an infinite number of iterations). However, if one of the variances is going to zero, first think about why this is happening. This is typically caused by one of three problems: 1) you made a mistake in inputting your data, e.g. used -99 as the missing value in your data but did not replace these with NAs before passing to MARSS, 2) your data are not sufficient to estimate multiple variances or 3) your data are inconsistent with the model you are trying fit.

The algorithms in the MARSS package are designed for cases where the $\mathbf{Q}$ and $\mathbf{R}$ diagonals are all non-minuscule. For example, the EM update equation for $\mathbf{u}$ will grind to a halt (not update $\mathbf{u}$) if $\mathbf{Q}$ is tiny (like 1E-7). Conversely, the BFGS equations are likely to miss the maximum-likelihood when $\mathbf{R}$ is tiny because then the likelihood surface becomes hyper-sensitive to $\boldsymbol{\pi}$. The solution is to use the degenerate likelihood function for the likelihood calculation and the EM update equations. MARSS will implement this

[1] An example of a $\boldsymbol{\pi}$ with shared values is $\boldsymbol{\pi} = \left[\begin{smallmatrix} a \\ a \\ a \end{smallmatrix}\right]$.

automatically when $\mathbf{Q}$ or $\mathbf{R}$ diagonal elements are set to zero and will try setting $\mathbf{Q}$ and $\mathbf{R}$ terms to zero automatically if `control$allow.degen=TRUE`.

One odd case can occur when $\mathbf{R}$ goes to zero (a matrix of zeros), but you are estimating $\boldsymbol{\pi}$. If `model$tinitx=1`, then $\boldsymbol{\pi} = \mathbf{x}_1^0$ and $\mathbf{y}_1 - \mathbf{Z}\mathbf{x}_1^0$ can go to 0 as well as $\text{var}(\mathbf{y}_1 - \mathbf{Z}\mathbf{x}_1^0)$ by driving $\mathbf{R}$ to zero. But as this happens, the log-likelihood associated with $\mathbf{y}_1$ will go (correctly) to infinity and thus the log-likelihood goes to infinity. But if you set $\mathbf{R} = 0$, the log-likelihood will be finite. The reason is that $\mathbf{R} \approx 0$ and $\mathbf{R} = 0$ specify different likelihoods associated with $\mathbf{y}_1 - \mathbf{Z}\mathbf{x}_1^0$. With $\mathbf{R} = 0$, $\mathbf{y}_1 - \mathbf{Z}\mathbf{x}_1^0$ does not have a distribution; it is just a fixed value. So there is no likelihood to go to infinity. If some elements of the diagonal of $\mathbf{R}$ are going to zero, you should be suspect of the parameter estimates. Sometimes the structure of your data, e.g. one data value followed by a long string of missing values, is causing an odd spike in the likelihood at $\mathbf{R} \approx 0$. Try manually setting $\mathbf{R}$ equal to zero to get the correct log-likelihood[2].

4.6 Other related packages

Packages that will do Kalman filtering and smoothing are many, but packages that estimate the parameters in a MARSS model, especially constrained MARSS models, are much less common. The following are those with which we are familiar, however there are certainly more packages for estimating MARSS models in engineering and economics of which we are unfamiliar. The MARSS package is unusual in that it uses an EM algorithm for maximizing the likelihood as opposed to a Newton-esque method (e.g. BFGS). The package is also unusual in that it allows you to specify the initial conditions at $t = 0$ or $t = 1$, allows degenerate models (with some of the diagonal elements of $\mathbf{R}$ or $\mathbf{Q}$ equal to zero). Lastly, model specification in the MARSS package has a one-to-one relationship between the model list in MARSS and the model as you would write it on paper as a matrix equation. This makes the learning curve a bit less steep. However, the MARSS package has not been optimized for speed and probably will be really slow if you have time-

[2]The likelihood returned when $\mathbf{R} \approx 0$ is not incorrect. It is just not the likelihood that you probably want. You want the likelihood where the $\mathbf{R}$ term is dropped because it is zero.

series data with a lot of time points.

atsar atsar is an **R** package we wrote for fitting MARSS models using STAN. It allows fast and flexible fitting of MARSS models in a Bayesian framework. Our book from our time-series class has example applications Applied Time-Series Analysis for Fisheries and Environmental Sciences.

stats The **stats** package (part of base R) has functions for fitting univariate structural time series models (MARSS models with a univariate y). Read the help file at '?StructTS'. The Kalman filter and smoother functions are described here: '?KalmanLike'.

DLM DLM is an **R** package for fitting MARSS models. Our impression is that it is mainly Bayesian focused but it does allow MLE estimation via the 'optim()' function. It has a book, Dynamic Linear Models with **R** by Petris et al., which has many examples of how to write MARSS models for different applications.

sspir sspir an **R** package for fitting ARSS (univariate) models with Gaussian, Poisson and binomial error distributions.

dse dse (Dynamic Systems Estimation) is an **R** package for multivariate Gaussian state-space models with a focus on ARMA models.

SsfPack SsfPack is a package for Ox/Splus that fits constrained multivariate Gaussian state-space models using mainly (it seems) the BFGS algorithm but the newer versions support other types of maximization. SsfPack is very flexible and written in C to be fast. It has been used extensively on statistical finance problems and is optimized for dealing with large (financial) data sets. It is used and documented in Time Series Analysis by State Space Methods by Durbin and Koopman, An Introduction to State Space Time Series Analysis by Commandeur and Koopman, and Statistical Algorithms for Models in State Space Form: SsfPack 3.0, by Koopman, Shephard, and Doornik.

Brodgar The Brodgar software was developed by Alain Zuur to do (among many other things) dynamic factor analysis, which involves a special type of MARSS model. The methods and many example analyses are given in Analyzing Ecological Data by Zuur, Ieno and Smith. This is the one package that we are aware of that also uses an EM algorithm for parameter estimation.

eViews eViews is a commercial economics software that will estimate at least some types of MARSS models.

KFAS The KFAS **R** package provides a fast Kalman filter and smoother. Examples in the package show how to estimate MARSS models using the KFAS functions and **R**'s 'optim()' function. The MARSS package uses the filter and smoother functions from the KFAS package.

S+FinMetrics S+FinMetrics is a S-plus module for fitting MAR models, which are called vector autoregressive (VAR) models in the economics and finance literature. It has some support for state-space VAR models, though we haven't used it so are not sure which parameters it allows you to estimate. It was developed by Andrew Bruce, Doug Martin, Jiahui Wang, and Eric Zivot, and it has a book associated with it: Modeling Financial Time Series with S-plus by Eric Zivot and Jiahui Wang.

kftrack The kftrack **R** package provides a suite of functions specialized for fitting MARSS models to animal tracking data.

Chapter 5

MARSS outputs

MARSS models are used in many different ways and different users will want different types of output. Some users will want the parameter estimates while others want the smoothed states and others want to use MARSS models to interpolate missing values and want the expected values of missing data.

The best way to find out how to get output is to type `?print.MARSS` at the command line after installing the **MARSS** package. The print help page discusses how to get parameter estimates in different forms, the smoothed and filtered states, all the Kalman filter and smoother output, all the expectations of y (missing data), confidence intervals and bias estimates for the parameters, and standard errors of the states. If you are looking only for Kalman filter and smoother output, see the relevant section in Chapter **??** and see the help page for the `MARSSkf()` function (type `?MARSSkf` at the ,**R** command line).

You might also want to look at the `augment()`, `tidy()` and `glance()` functions which will summarize commonly needed output from a MARSS model fit. Type `?augment.marssMLE` at the command line to see examples. These functions work as they do in the **broom** R package.

Chapter 6

MARSS Residuals

Chapter 7

Confidence Intervals

Chapter 8

Predictions

Chapter 9

Troubleshooting

Numerical errors due to ill-conditioned matrices are not uncommon when fitting MARSS models. The Kalman and EM algorithms need inverses of matrices. If those matrices become ill-conditioned, for example all elements are close to the same value, then the algorithm becomes unstable. Warning messages will be printed if the algorithms are becoming unstable and you can set `control$trace=1`, to see details of where the algorithm is becoming unstable. Whenever possible, you should avoid using shared $\boldsymbol{\pi}$ values in your model[1]. The way our algorithm deals with $\boldsymbol{\Lambda}$ tends to make this case unstable, especially if $\mathbf{R}$ is not diagonal. In general, estimation of a non-diagonal $\mathbf{R}$ is more difficult, more prone to ill-conditioning, and more data-hungry.

You may also see non-convergence warnings, especially if your MLE model turns out to be degenerate. This means that one of the elements on the diagonal of your $\mathbf{Q}$ or $\mathbf{R}$ matrix are going to zero (are degenerate). It will take the EM algorithm forever to get to zero. BFGS will have the same problem, although it will often get a bit closer to the degenerate solution. If you are using `method="kem"`, MARSS will warn you if it looks like the solution is degenerate. If you use `control=list(allow.degen=TRUE)`, the EM algorithm will attempt to set the degenerate variances to zero (instead of trying to get to zero using an infinite number of iterations). However, if one of the variances is going to zero, first think about why this is happening. This is typically caused by one of three problems: 1) you made a mistake in

[1] An example of a $\boldsymbol{\pi}$ with shared values is $\boldsymbol{\pi} = \left[\begin{smallmatrix} a \\ a \\ a \end{smallmatrix}\right]$.

inputting your data, e.g. used -99 as the missing value in your data but did not replace these with NAs before passing to MARSS, 2) your data are not sufficient to estimate multiple variances or 3) your data are inconsistent with the model you are trying fit.

The algorithms in the **MARSS** package are designed for cases where the **Q** and **R** diagonals are all non-minuscule. For example, the EM update equation for **u** will grind to a halt (not update **u**) if **Q** is tiny (like 1E-7). Conversely, the BFGS equations are likely to miss the maximum-likelihood when **R** is tiny because then the likelihood surface becomes hyper-sensitive to $\boldsymbol{\pi}$. The solution is to use the degenerate likelihood function for the likelihood calculation and the EM update equations. **MARSS** will implement this automatically when **Q** or **R** diagonal elements are set to zero and will try setting **Q** and **R** terms to zero automatically if `control$allow.degen=TRUE`.

One odd case can occur when **R** goes to zero (a matrix of zeros), but you are estimating $\boldsymbol{\pi}$. If `model$tinitx=1`, then $\boldsymbol{\pi} = \mathbf{x}_1^0$ and $\mathbf{y}_1 - \mathbf{Z}\mathbf{x}_1^0$ can go to 0 as well as $\text{var}(\mathbf{y}_1 - \mathbf{Z}\mathbf{x}_1^0)$ by driving **R** to zero. But as this happens, the log-likelihood associated with $\mathbf{y}_1$ will go (correctly) to infinity and thus the log-likelihood goes to infinity. But if you set $\mathbf{R} = 0$, the log-likelihood will be finite. The reason is that $\mathbf{R} \approx 0$ and $\mathbf{R} = 0$ specify different likelihoods associated with $\mathbf{y}_1 - \mathbf{Z}\mathbf{x}_1^0$. With $\mathbf{R} = 0$, $\mathbf{y}_1 - \mathbf{Z}\mathbf{x}_1^0$ does not have a distribution; it is just a fixed value. So there is no likelihood to go to infinity. If some elements of the diagonal of **R** are going to zero, you should be suspect of the parameter estimates. Sometimes the structure of your data, e.g. one data value followed by a long string of missing values, is causing an odd spike in the likelihood at $\mathbf{R} \approx 0$. Try manually setting **R** equal to zero to get the correct log-likelihood[2].

[2]The likelihood returned when $\mathbf{R} \approx 0$ is not incorrect. It is just not the likelihood that you probably want. You want the likelihood where the **R** term is dropped because it is zero.

Chapter 10

EM algorithm

The **MARSS** package fits models via maximum likelihood. The MARSS package is unusual among packages for fitting MARSS models in that fitting is performed via a constrained EM algorithm (**?**) based on a vectorized form of Equation 1.1 (See Chapter **??** for the vectorized form used in the algorithm). Although fitting via the BFGS algorithm is also provided using `method="BFGS"` and the optim function in ,**R**, the examples in this guide use the EM algorithm primarily because it gives robust estimation for datasets replete with missing values and for high-dimensional models with various constraints. However, there are many models/datasets where BFGS is faster and we typically try both for problems. The EM algorithm is also often used to provide initial conditions for the BFGS algorithm (or an MCMC routine) in order to improve the performance of those algorithms. In addition to the main model fitting function, the MARSS package supplies functions for bootstrap and approximate confidence intervals, parametric and non-parametric bootstrapping, model selection (AIC and bootstrap AIC), simulation, and bootstrap bias correction.

10.1 Important notes about the algorithms

Specification of a properly constrained model with a unique solution is the responsibility of the user because MARSS has no way to tell if you have specified an insufficiently constrained model—with correspondingly an infinite

number of solutions. How do you know if the model is properly constrained? If you are using a MARSS model form that is widely used, then you can probably assume that it is properly constrained. If you go to papers where someone developed the model or method, the issue of constraints necessary to ensure "identifiability" will likely be addressed if it is an issue. Are you fitting novel MARSS models? Then you will need to do some study on identifiability in this class of models using textbooks (Appendix **??**). Often textbooks do not address identifiability explicitly. Rather it is addressed implicitly by only showing a model constructed in such a way that it is identifiable. In our work, if we suspect identification problems, we will often first do a Bayesian analysis with flat priors and look for oddities in the posteriors, such as ridges, plateaus or bimodality.

All the EM code in the **MARSS** package is currently in native ,**R**. Thus the model fitting is slow. The classic Kalman filter/smoother algorithm, as shown in **?**, p. 331-335, is based on the original smoother presented in **?**. This Kalman filter is provided in function `MARSSkfss()`, but the default Kalman filter and smoother used in the **MARSS** package is based on the algorithm in **?** and papers by Koopman et al. This Kalman filter and smoother is provided in the **KFAS** package (Helske 2012). Table 2 in **?** indicates that the classic algorithm is 40-100 times slower than the algorithm given in **?**, **?**, and **?**. The MARSS package function `MARSSkfas()` provides a translator between the model objects in MARSS and those in **KFAS** so that the **KFAS** functions can be used. `MARSSkfas()` also includes a lag-one covariance smoother algorithm as this is not output by the **KFAS** functions, and it provides proper formulation of the priors so that one can use the **KFAS** functions when the prior on the states is set at $t = 0$ instead of $t = 1$. Simply off-setting your data to start at t=2 and sending that value to $t_{init} = 1$ in the **KFAS** Kalman filter would not be mathematically correct!

EM algorithms will quickly get in the vicinity of the maximum likelihood, but the final approach to the maximum is generally slow relative to quasi-Newton methods. On the flip side, EM algorithms are quite robust to initial conditions choices and can be extremely fast at getting close to the MLE values for high-dimensional models. The **MARSS** package also allows one to use the BFGS method to fit MARSS models, thus one can use an EM algorithm to "get close" and then the BFGS algorithm to polish off the estimate. Restricted maximum-likelihood algorithms are also available for AR(1) state-space models, both univariate (**?**) and multivariate (**?**). REML can give

parameter estimates with lower variance than plain maximum-likelihood algorithms. However, the algorithms for REML when there are missing values are not currently available (although that will probably change in the near future). Another maximum-likelihood method is data-cloning which adapts MCMC algorithms used in Bayesian analysis for maximum-likelihood estimation (**?**).

Missing values are seamlessly accommodated with the **MARSS** package. Simply specify missing data with NAs. The likelihood computations are exact and will deal appropriately with missing values. However, no innovations, referring to the non-parametric bootstrap developed by Stoffer and Wall (1991), bootstrapping can be done if there are missing values. Instead parametric bootstrapping must be used.

You should be aware that maximum-likelihood estimates of variance in MARSS models are fundamentally biased, regardless of the algorithm used. This bias is more severe when one or the other of $\mathbf{R}$ or $\mathbf{Q}$ is very small, and the bias does not go to zero as sample size goes to infinity. The bias arises because variance is constrained to be positive. Thus if $\mathbf{R}$ or $\mathbf{Q}$ is essentially zero, the mean estimate will not be zero and thus the estimate will be biased high while the corresponding bias of the other variance will be biased low. You can generate unbiased variance estimates using a bootstrap estimate of the bias. The function `MARSSparamCIs()` will do this. However be aware that adding an estimated bias to a parameter estimate will lead to an increase in the variance of your parameter estimate. The amount of variance added will depend on sample size.

You should also be aware that mis-specification of the prior on the initial states ($\boldsymbol{\pi}$ and $\boldsymbol{\Lambda}$) can have catastrophic effects on your parameter estimates if your prior conflicts with the distribution of the initial states implied by the MARSS model. These effects can be very difficult to detect because the model will appear to be well-fitted. Unless you have a good idea of what the parameters should be, you might not realize that your prior conflicts.

The most common problems we have found with priors on $\mathbf{x}_0$ are the following. Problem 1) The correlation structure in $\boldsymbol{\Lambda}$ (whether the prior is diffuse or not) does not match the correlation structure in $\mathbf{x}_0$ implied by your model. For example, you specify a diagonal $\boldsymbol{\Lambda}$ (independent states), but the implied distribution has correlations. Problem 2) The correlation structure in $\boldsymbol{\Lambda}$ does not match the structure in $\mathbf{x}_0$ implied by constraints you placed on $\boldsymbol{\pi}$. For

example, you specify that all values in $\boldsymbol{\pi}$ are shared, yet you specify that $\boldsymbol{\Lambda}$ is diagonal (independent).

Unfortunately, using a diffuse prior does not help with these two problems because the diffuse prior still has a correlation structure and can still conflict with the implied correlation in $\mathbf{x}_0$. One way to get around these problems is to set $\boldsymbol{\Lambda}$=0 (a $m \times m$ matrix of zeros) and estimate $\boldsymbol{\pi} \equiv \mathbf{x}_0$ only. Now $\boldsymbol{\pi}$ is a fixed but unknown (estimated) parameter, not the mean of a distribution. In this case, $\boldsymbol{\Lambda}$ does not exist in your model and there is no conflict with the model.
Be aware however that estimating $\boldsymbol{\pi}$ as a parameter is not always robust. If you specify that $\boldsymbol{\Lambda}$=0 and specify that $\boldsymbol{\pi}$ corresponds to $\mathbf{x}_0$, but your model "explodes" when run backwards in time, you cannot estimate $\boldsymbol{\pi}$ because you cannot get a good estimate of $\mathbf{x}_0$. Sometimes this can be avoided by specifying that $\boldsymbol{\pi}$ corresponds to $\mathbf{x}_1$ so that it can be constrained by the data $\mathbf{y}_1$.

In summary, if the implied correlation structure of your initial states is independent (diagonal variance-covariance matrix), you should generally be ok with a diagonal and high variance prior or with treating the initial states as parameters (with $\boldsymbol{\Lambda} = 0$). But if your initial states have an implied correlation structure that is not independent, then proceed with caution. 'With caution' means that you should assume you have problems and test how your model fits with simulated data.

10.2 State-space form of ARMA(p,q) models

There is a large class of models in the statistical finance literature that have the form

$$\begin{aligned} \mathbf{x}_{t+1} &= \mathbf{B}\mathbf{x}_t + \boldsymbol{\Gamma}\boldsymbol{\eta}_t \\ \mathbf{y}_t &= \mathbf{Z}\mathbf{x}_t + \boldsymbol{\eta}_t \end{aligned}$$

For example, ARMA(p,q) models can be written in this form. The MARSS model framework in this package will not allow you to write models in that form. You can put the $\boldsymbol{\eta}_t$ into the $\mathbf{x}_t$ vector and set $\mathbf{R} = 0$ to make models of this form using the MARSS form, but the EM algorithm in the MARSS package won't let you estimate parameters because the parameters will drop

out of the full likelihood being maximized in the algorithm. You can try using BFGS by passing in the `method` argument to the MARSS() call.

Chapter 11

Other related packages

Packages that will do Kalman filtering and smoothing are many, but packages that estimate the parameters in a MARSS model, especially constrained MARSS models, are much less common. The following are those with which we are familiar, however there are certainly more packages for estimating MARSS models in engineering and economics of which we are unfamiliar. The **MARSS** package is unusual in that it uses an EM algorithm for maximizing the likelihood as opposed to a Newton-esque method (e.g. BFGS). The package is also unusual in that it allows you to specify the initial conditions at $t = 0$ or $t = 1$, allows degenerate models (with some of the diagonal elements of **R** or **Q** equal to zero). Lastly, model specification in the **MARSS** package has a one-to-one relationship between the model list in `MARSS()` and the model as you would write it on paper as a matrix equation. This makes the learning curve a bit less steep. However, the **MARSS** package has not been optimized for speed and probably will be really slow if you have time-series data with a lot of time points.

atsar atsar is an **R** package we wrote for fitting MARSS models using STAN. It allows fast and flexible fitting of MARSS models in a Bayesian framework. Our book from our time-series class has example applications Applied Time-Series Analysis for Fisheries and Environmental Sciences.

stats The **stats** package (part of base R) has functions for fitting univariate structural time series models (MARSS models with a univariate y). Read the help file at '?StructTS'. The Kalman filter and smoother

functions are described here: '?KalmanLike'.

DLM DLM is an **R** package for fitting MARSS models. Our impression is that it is mainly Bayesian focused but it does allow MLE estimation via the 'optim()' function. It has a book, *Dynamic Linear Models* with **R** by Petris et al., which has many examples of how to write MARSS models for different applications.

sspir sspir an **R** package for fitting ARSS (univariate) models with Gaussian, Poisson and binomial error distributions.

dse dse (Dynamic Systems Estimation) is an **R** package for multivariate Gaussian state-space models with a focus on ARMA models.

SsfPack SsfPack is a package for Ox/Splus that fits constrained multivariate Gaussian state-space models using mainly (it seems) the BFGS algorithm but the newer versions support other types of maximization. **SsfPack** is very flexible and written in C to be fast. It has been used extensively on statistical finance problems and is optimized for dealing with large (financial) data sets. It is used and documented in *Time Series Analysis by State Space Methods* by Durbin and Koopman, *An Introduction to State Space Time Series Analysis* by Commandeur and Koopman, and *Statistical Algorithms for Models in State Space Form: SsfPack 3.0*, by Koopman, Shephard, and Doornik.

Brodgar The Brodgar software was developed by Alain Zuur to do (among many other things) dynamic factor analysis, which involves a special type of MARSS model. The methods and many example analyses are given in *Analyzing Ecological Data* by Zuur, Ieno and Smith. This is the one package that we are aware of that also uses an EM algorithm for parameter estimation.

eViews **eViews** is a commercial economics software that will estimate at least some types of MARSS models.

KFAS The KFAS **R** package provides a fast Kalman filter and smoother. Examples in the package show how to estimate MARSS models using the **KFAS** functions and **R**'s 'optim()' function. The **MARSS** package uses the filter and smoother functions from the **KFAS** package.

S+FinMetrics S+FinMetrics is a S-plus module for fitting MAR models,

which are called vector autoregressive (VAR) models in the economics and finance literature. It has some support for state-space VAR models, though we haven't used it so are not sure which parameters it allows you to estimate. It was developed by Andrew Bruce, Doug Martin, Jiahui Wang, and Eric Zivot, and it has a book associated with it: *Modeling Financial Time Series with S-plus* by Eric Zivot and Jiahui Wang.

kftrack The kftrack **R** package provides a suite of functions specialized for fitting MARSS models to animal tracking data.

Bibliography