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

Analysis of multivariate time series using the MARSS package

version 3.11.10

September 9, 2025

NOAA Fisheries and USGS WA Cooperative Fish and Wildlife Research Unit
Seattle, WA, USA

Holmes, E. E., M. D. Scheuerell and E. J. Ward. Analysis of multivariate time-series using the MARSS package. Version 3.11.10. NOAA Fisheries, Northwest Fisheries Science Center, 2725 Montlake Blvd E., Seattle, WA 98112.
DOI: https://doi.org/10.5281/zenodo.5781847

Preface

The initial motivation for our work with MARSS models was a collaboration with Rich Hinrichsen. Rich developed a framework for analysis of multi-site population count data using MARSS models and bootstrap AICb (Hinrichsen and Holmes, 2009). Our work (EEH and EJW) extended Rich's framework, made it more general, and led to the development of a parametric bootstrap AICb for MARSS models, which allows one to do model-selection using datasets with missing values (Ward et al., 2010; Holmes and Ward, 2010). Later, we developed additional algorithms for simulation and confidence intervals. Discussions with Mark Scheuerell led to an extensive revision of the EM algorithm and to the development of a general EM algorithm for constrained MARSS models (Holmes, 2012). Discussions with Mark also led to a complete rewrite of the model specification so that the package could be used for MARSS models in general—rather than simply the form of MARSS model used in our applications. Many collaborators have helped test the package; we thank especially Yasmin Lucero, Kevin See, and Brice Semmens. Development of the code into a R package would not have been possible without Kellie Wills, who wrote much of the original package code outside of the algorithm functions. Finally, we thank the participants of our MARSS workshops and courses and the MARSS users who have contacted us regarding issues that were unclear in the manual, errors, or suggestions regarding new applications. Discussions with these users have helped us improve the manual and go in new directions.

The application chapters were developed originally as part of workshops on analysis of multivariate time-series data given at the Ecological Society of America meetings since 2005 and taught by us along with Yasmin Lucero, Stephanie Hampton, and Brice Semmens. The chapter on extinction estimation and trend estimation was initially developed by Brice Semmens and later extended by us for this user guide. The algorithm behind the TMU figure in Chapter 7 was developed during a collaboration with Steve Ellner (Ellner and Holmes, 2008). Later we further developed the chapters as part of a course we teach on analysis of fisheries and environmental time-series data at the University of Washington. You can find online versions of our time-series analysis course and an eBook from the course on our Applied Time Series Analysis website `http://atsa-es.github.io`.

The authors are federal research scientists; EEH and EJW are with NOAA Fisheries and MDS is with USGS (and University of Washington). This work was conducted as part of our jobs with United States federal government agencies. A CAMEO grant from the National Science Foundation and NOAA Fisheries provided the initial impetus for the development of the package as part of a research project with Stephanie Hampton, Lindsay Scheef, and Steven Katz on analysis of marine plankton time series. During the initial stages of this work, EJW was supported on a post-doctoral fellowship from the National Research Council and MDS was partially supported by a PECASE award from the White House Office of Science and Technology Policy.

commercial applications nor may it be copyrighted. Use of the EM algorithm should cite Holmes (2012). Links to more code and publications on MARSS applications can be found by following the links at our academic websites:

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

Contents

Part I

The MARSS package

1

Overview

MARSS stands for Multivariate Auto-Regressive(1) State-Space. The {MARSS} package is an R package[1] 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 D). 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[2].

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

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

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

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

The **x** equation is termed the state process and the **y** equation is termed the observation process. Data enter the model as the **y**; that is the **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:

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.

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 $\mathbf{Q}_t$.

[1] The curly brackets are used to denote an R package.

[2] Fitting via the BFGS algorithm is also provided using R 's `optim()` function, but this is not the focus of the package.

$\mathbf{y}$ 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 errors 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.
π is either a parameter or a fixed prior. It is a $m \times 1$ matrix.
Λ 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.
$\mathbf{c}$ and $\mathbf{d}$ 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.

In some fields, the $\mathbf{u}$ and $\mathbf{a}$ terms are routinely set to 0 or the model is written in such a way that they are incorporated into $\mathbf{B}$ or $\mathbf{Z}$. However, in other fields, the $\mathbf{u}$ and $\mathbf{a}$ terms are the main objects of interest, and the model is written to explicitly show them. We include them throughout our discussion, but they can be set to zero if desired.

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 18. 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 D) for examples of how a wide variety of autoregressive models can be written in MARSS form.

1.1 What does the {MARSS} package do?

Written in an unconstrained form[3], 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.

[3] meaning all the elements in a parameter matrices are allowed to be different and none constrained to be equal or related.

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

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

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

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).

The {MARSS} package fits models via maximum likelihood. The package is unusual among packages for fitting MARSS models in that fitting is performed via a constrained EM algorithm (Holmes, 2012) based on a vectorized form of Equation 1.1 (See Chapter 3 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.

1.2 Output: fitted values, residuals, predictions, plots etc

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 {MARSS}. 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 3 and see the help page for the `MARSSkf()` function (type `?MARSSkf` at the R command line).

The `tidy()` and `glance()` functions will summarize commonly needed output from a `MARSS()` model fit.

1.3 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 5. Appendix C also has many examples of how to specify different forms for your parameter matrices.

1.4 Getting your data in right format

Your data need to be a matrix, not data frame, with time across the columns ($n \times T$ matrix). Note a univariate or multivariate ts (time-series) object can also be used and this will be converted to a $n \times T$ 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 4) using your data and using the `model` argument to specify the structure of each parameter.

1.4.1 Getting a ts object into the right form

A R ts object (time series object) stores information about the time steps of the data and often seasonal information (the quarter or month). You can pass in your data as a ts object and `MARSS()` will convert this to matrix form. However if you have your data in ts form, then you may be 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 with a matrix of covariates for season.

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.

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 your covariates are `dat[c("Yr","Month","Sal"),]`.

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.

1.5 Important notes about the algorithms

Specification of a properly constrained model with a unique solution is the responsibility of the user. {MARSS} includes a number of checks to catch some cases of unsolvable models, but there are many othere cases where there is 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 D). 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 Shumway and Stoffer (2006, p. 331-335), is based on the original smoother presented in Rauch (1963). 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 Kohn and Ansley (1989) and papers by Koopman et al. This Kalman filter and smoother is provided in the {KFAS} package (Helske 2012). Table 2 in Koopman (1993) indicates that the classic algorithm is 40-100 times slower than the algorithm given in Kohn and Ansley (1989), Koopman (1993), and Koopman et al. (1999). 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 (Staples et al., 2004) and multivariate (Hinrichsen and Holmes, 2009). 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 (Lele et al., 2007).

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[4] 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

[4] referring to the non-parametric bootstrap developed by Stoffer and Wall (1991).

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 (π and Λ) 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 Λ (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 Λ (independent states), but the implied distribution has correlations. Problem 2) The correlation structure in Λ does not match the structure in $\mathbf{x}_0$ implied by constraints you placed on π. For example, you specify that all values in π are shared, yet you specify that Λ 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 $\Lambda = 0$ (a $m \times m$ matrix of zeros) and estimate $\pi \equiv \mathbf{x}_0$ only. Now π is a fixed but unknown (estimated) parameter, not the mean of a distribution. In this case, Λ does not exist in your model and there is no conflict with the model. Be aware however that estimating π as a parameter is not always robust. If you specify that Λ=0 and specify that π corresponds to $\mathbf{x}_0$, but your model "explodes" when run backwards in time, you cannot estimate π because you cannot get a good estimate of $\mathbf{x}_0$. Sometimes this can be avoided by specifying that π 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 $\Lambda = 0$). 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.

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

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

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 η_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.

1.6 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 π values in your model[5]. The way our algorithm deals with Λ 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 to 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 π. The solution is to use the EM update equations with the degenerate likelihood function. `MARSS()` will implement this automatically by trying to set $\mathbf{Q}$ and $\mathbf{R}$ diagonal terms to zero if they are going to zero[6].

One odd case can occur when $\mathbf{R}$ goes to zero (a matrix of zeros), but you are estimating π. If `model$tinitx=1`, then $\pi = \mathbf{x}_1^0$ 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

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

[6] You can turn off this behavior by passing in `control=list(allow.degen=FALSE)`.

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[7].

1.7 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 $t = 1$ qne 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 (in matrix form). However, the {MARSS} package has not been optimized for speed and probably will be very slow if you have time-series data with many 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. It is mainly Bayesian focused but it also allows 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.

[7] 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.

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 package 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. 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.

2

The main package functions

The {MARSS} package is object-based. It has two main types of objects: a model object (class marssMODEL) and a maximum-likelihood fitted model object (class marssMLE). A marssMODEL object specifies the structure of the model to be fitted. It is an R code version of the MARSS equation (Equation 1.1). A marssMLE object specifies both the model and the information necessary for fitting (initial conditions, controls, method). If the model has been fitted, the marssMLE object will also have the parameter estimates and (optionally) confidence intervals and bias.

2.1 The `MARSS()` function: inputs

The function `MARSS()` is an interface to the core fitting functions in the {MARSS} package. It allows a user to fit a MARSS model using a list to describe the model structure. It returns marssMODEL and marssMLE objects which the user can later use in other functions, e.g., simulating or computing bootstrap confidence intervals.

`MLEobj=MARSS(data, model=list(), ..., fit=TRUE)` This function will fit a MARSS model to the data using a model list which is a list describing the structure of the model parameter matrices. In the default model, i.e., if you use `MARSS(dat)` with no `model` argument, **Z** and **B** are the identity matrix, **R** is a diagonal matrix with one variance, **Q** is a diagonal matrix with unique variances, **u** is unique, **a** is scaling, and **C**, **c**, **D**, and **d** are all zero. The output is a marssMLE object where the estimated parameter matrices are in `MLEobj$par`. If `fit=FALSE`, it returns a minimal marssMLE object that is ready for passing to a fitting function (below) but with no `par` element.

2.2 The `MARSS()` function: outputs

The marssMLE object returned by a `MARSS()` call includes the estimated parameters, states, and expected values of any missing data. Derived statistics, such as confidence intervals and standard errors, can be computed using the functions described below.

estimated parameters `coef(MLEobj)` The coef function can output parameters in a variety of formats, such as a list of matrices versus a vector of the estimates. See `?coef.marssMLE`.

residuals `residuals(MLEobj)`. See `?MARSSresiduals` for a discussion of standardized residuals in the context of MARSS models.

Kalman filter and smoother output `tsSmooth(MLEobj)`. The smoothed states are in `MLEobj$states`. `tsSmooth(MLEobj)` provides filter and smoother output as a data frame, but the full Kalman filter and smoother output is available in matrix form from `MARSSkf(MLEobj)`. See `?MARSSkf` for a discussion of the Kalman filter and smoother outputs. If you just want the estimated states conditioned on all the data, use `tsSmooth(MLEobj)`; you can pass in `interval="confidence"`.

expected value of missing y `tsSmooth(MLEobj, type="ytT")` returns these as a data frame. `MARSShatyt(MLEobj)` returns the same (and much more) as matrices. See `?MARSShatyt` for a discussion of the expectations involving $\mathbf{Y}$.

log-likelihood `logLik(MLEobj)` returns the log-likelihood.

AIC `AIC(MLEobj)` ({stats} package) returns the uncorrected AIC. Use `MLEobj$AICc` to return the small sample size corrected AIC.

Note the `print` method for marssMLE objects will print or compute all the frequently needed output using the `what=` argument in the print call. Type `?print.MARSS` at the R command line to see the print help file which will also point you to the more standard functions (like `coef()`).

2.3 Core functions for fitting a MARSS model

The following core functions are designed to work with 'unfitted' marssMLE objects, that is a marssMLE object without the `par` element. Users do not normally need to call the `MARSSkem()` or `MARSSoptim()` functions since `MARSS()` will call those. Note, these functions can be called with a marssMLE object with a `par` element, but these functions will overwrite that element.

`MLEobj=MARSSkem(MLEobj)` This will fit a MARSS model via the EM algorithm to the data using a properly specified marssMLE object, which has data, the marssMODEL object and the necessary initial condition and control elements. See the appendix on the object structures in the {MARSS} package. `MARSSkem()` does no error-checking. See `is.marssMLE()` for error-checking. `MARSSkem()` uses `MARSSkf()` described below.

`MLEobj=MARSSoptim(MLEobj)` This will fit a MARSS model via the BFGS algorithm provided in `optim()`. This requires a properly specified marssMLE object, such as would be passed to `MARSSkem()`.

`is.marssMLE(MLEobj)` This will check that a marssMLE object is properly specified and ready for fitting. This should be called before `MARSSkem()` or `MARSSoptim()` is called. This function is not typically needed if using `MARSS()` since `MARSS()` builds the model object for the user and does error-checking on model structure.

2.4 Functions for a fitted marssMLE object

The following functions use a marssMLE object that has a populated `par` element, i.e., a marssMLE object returned from one of the fitting functions (`MARSS()`, `MARSSkem()`, `MARSSoptim()`). Below `MODELobj` means the argument is a marssMODEL object and `MLEobj` means the argument is a marssMLE object. Type `?function.name` to see information on function usage and examples.

standard functions The standard R functions for fitted objects are provided: `residuals()`, `fitted()`, `logLik()`, `AIC()`, `coef()`, `predict()` and `tsSmooth()`.

summary functions Standard functions for printing output are also available: `summary()` and `print()` along with 'tidyverse' output: `tidy()` and `glance()`.

In addition, the following are special functions for {MARSS} fitted models:

`kf=MARSSkf(MLEobj)` This will compute the expected values of the hidden states given data via the Kalman filter (to produce estimates conditioned on the data from $t = 1$ to $t - 1$) and the Kalman smoother (to produce estimates conditioned on data from $t = 1$ to $t = T$). The function also returns the exact likelihood of the data conditioned on `MLEobj$par`. A variety of other Kalman filter/smoother information is also output (`kf` is a list of output); see `?MARSSkf` for details. `tsSmooth.marssMLE()` returns this information as a data frame that is `ggplot()` friendly.

`MLEobj=MARSSaic(MLEobj)` This adds model selection criteria, AIC, AICc, and AICb, to a marssMLE object. Note, AIC and AICc are added to marssMLE objects by the `MARSS()` function but AICb is not.

`boot=MARSSboot(MLEobj)` This returns a list containing bootstrapped parameters and data via parametric or innovations bootstrapping.

`MLEobj=MARSShessian(MLEobj)` This adds the Hessian matrix for the estimated parameters to a marssMLE object. The default algorithm is the analytical solution for the Hessian. See `?MARSShessian`.

`MLEobj=MARSSparamCIs(MLEobj)` This adds standard errors, confidence intervals, and bootstrap estimated bias for the maximum-likelihood parameters using bootstrapping or the Hessian to the marssMLE object.

`sim.data=MARSSsimulate(MLEobj)` This returns simulated data from a MARSS model specified via a list of parameter matrices in `MLEobj$parList` (this is a list with elements `Q`, `R`, `U`, etc.).

`paramVec=MARSSvectorizeparam(MLEobj)` This returns the estimated (and only the estimated) parameters as a vector. This is useful for storing the results of simulations or for writing functions that fit MARSS models using R 's `optim` function. `coef(MLEobj)` will return the same vector.

`new.MLEobj=MARSSvectorizeparam(MLEobj, paramVec)` This will return a marssMLE object in which the estimated parameters (which are in `MLEobj$par`) are replaced with the values in `paramVec`.

2.5 Functions for marssMODEL objects

`is.marssMODEL(MODELobj)` This will check that the free and fixed matrices in a marssMODEL object are properly specified. This function is not typically needed if using `MARSS()` since `MARSS()` builds the marssMODEL object for the user and does error-checking on model structure.

`summary(MODELobj)` This will print the model parameter matrices showing the fixed values (in parentheses) and the location of the estimated elements. The estimated elements are shown as g1, g2, g3, ... which indicates which elements are shared, i.e., forced to have the same value. For example, an i.i.d. **R** matrix would appear as a diagonal matrix with just g1 on the diagonal.

3

Algorithms used in the {MARSS} package

3.1 The full time-varying MARSS model

In mathematical form, the model that is being fit with the package is

$$\begin{aligned}
\mathbf{x}_t &= (\mathbf{x}_{t-1}^\top \otimes \mathbf{I}_m)\,\mathrm{vec}(\mathbf{B}_t) + (\mathbf{u}_t^\top \otimes \mathbf{I}_m)\,\mathrm{vec}(\mathbf{U}_t) + \mathbf{w}_t, \quad \mathbf{W}_t \sim \mathrm{MVN}(0, \mathbf{Q}_t) \\
\mathbf{y}_t &= (\mathbf{x}_t^\top \otimes \mathbf{I}_n)\,\mathrm{vec}(\mathbf{Z}_t) + (\mathbf{a}_t^\top \otimes \mathbf{I}_n)\,\mathrm{vec}(\mathbf{A}_t) + \mathbf{v}_t, \quad \mathbf{V}_t \sim \mathrm{MVN}(0, \mathbf{R}_t) \\
\mathbf{x}_{t_0} &= \pi + \mathbf{F} \ll, \mathbf{L} \sim \mathrm{MVN}(0, \Lambda)
\end{aligned} \tag{3.1}$$

Each model parameter matrix, $\mathbf{B}_t$, $\mathbf{U}_t$, $\mathbf{Q}_t$, $\mathbf{Z}_t$, $\mathbf{A}_t$, and $\mathbf{R}_t$, is written as a time-varying linear model, $\mathbf{f}_t + \mathbf{D}_t\mathbf{m}$, where $\mathbf{f}$ and $\mathbf{D}$ are fully-known (not estimated and no missing values) and $\mathbf{m}$ is a column vector of the estimates elements of the parameter matrix:

$$\begin{gathered}
\mathrm{vec}(\mathbf{B}_t) = \mathbf{f}_{t,b} + \mathbf{D}_{t,b}\boldsymbol{\beta} \quad \mathrm{vec}(\mathbf{U}_t) = \mathbf{f}_{t,u} + \mathbf{D}_{t,u}\boldsymbol{\upsilon} \quad \mathrm{vec}(\mathbf{Q}_t) = \mathbf{f}_{t,q} + \mathbf{D}_{t,q}\mathbf{q} \\
\mathrm{vec}(\mathbf{Z}_t) = \mathbf{f}_{t,z} + \mathbf{D}_{t,z}\boldsymbol{\zeta} \quad \mathrm{vec}(\mathbf{A}_t) = \mathbf{f}_{t,a} + \mathbf{D}_{t,a}\boldsymbol{\alpha} \quad \mathrm{vec}(\mathbf{R}_t) = \mathbf{f}_{t,r} + \mathbf{D}_{t,r}\mathbf{r} \\
\mathrm{vec}(\Lambda) = \mathbf{f}_\lambda + \mathbf{D}_\lambda\boldsymbol{\lambda} \quad \mathrm{vec}(\pi) = \mathbf{f}_\pi + \mathbf{D}_\pi\mathbf{p}
\end{gathered}$$

The internal model specification (element `$marss` in a fitted marssMLE object output by a `MARSS()` call) is a list with the $\mathbf{f}_t$ ("fixed") and $\mathbf{D}_t$ ("free") matrices for each parameter. The output from fitting are the vectors, $\boldsymbol{\beta}$, $\boldsymbol{\upsilon}$, etc. The trick is to rewrite the user's linear multivariate problem into the general form (Equation 3.1). MARSS does this using functions that take more familiar arguments as input and then constructs the $\mathbf{f}_t$ and $\mathbf{D}_t$ matrices. Because the $\mathbf{f}_t$ and $\mathbf{D}_t$ can be whatever the user wants (assuming they are the right shape), this allows users to include covariates, trends (linear, sinusoidal, etc) or indicator variables in a variety of ways. It also means that terms like $1 + b + 2c$ can appear in the parameter matrices.

Although the above form looks unusual, it is equivalent to the commonly seen form but leads to a log-likelihood function where all terms have form $\mathbf{Mm}$, where $\mathbf{M}$ is a matrix and $\mathbf{m}$ is a column vector of only the different estimated values. This makes it easy to do the partial differentiation with respect to $\mathbf{m}$ necessary for the EM algorithm and as a result, easy to impose linear constraints and structure on the elements in a parameter matrix (Holmes, 2012).

3.2 Maximum-likelihood parameter estimation

3.2.1 EM algorithm

Function `MARSSkem()` in the {MARSS} package provides a maximum-likelihood algorithm for parameter estimation based on an Expectation-Maximization (EM) algorithm (Holmes, 2012). EM algorithms are widely used algorithms that extend maximum-likelihood estimation to cases where there are hidden random variables in a model (Dempster et al., 1977; Harvey, 1989; Harvey and Shephard, 1993; McLachlan and Krishnan, 2008). Expectation-Maximization algorithms for unconstrained MARSS models have been around for many years and algorithms for certain constrained cases have also been published. What makes the EM algorithm in MARSS different is that it is a general constrained algorithm that allows generic linear constraints among matrix elements (thus allows fixed, shared and linear combinations of estimated elements).

The EM algorithm finds the maximum-likelihood estimates of the parameters in a MARSS model using an iterative process. Starting with an initial set of parameters[1], which we will denote $\hat{\Theta}_1$, an updated parameter set $\hat{\Theta}_2$ is obtaining by finding the $\hat{\Theta}_2$ that maximizes the expected value of the likelihood over the distribution of the states ($\mathbf{X}$) conditioned on $\hat{\Theta}_1$. This distributon of states is computed via the Kalman smoother (Section 3.3). Mathematically, each iteration of an EM algorithm does this maximization:

$$\hat{\Theta}_2 = \arg\max_{\Theta} \quad \mathrm{E}_{\mathbf{X}|\hat{\Theta}_1}[\log L(\Theta|\mathbf{Y}=\mathbf{y}_1^T,\mathbf{X})]$$

Then using $\hat{\Theta}_2$, the distibution of $\mathbf{X}$ conditioned on $\hat{\Theta}_2$ is computed. Then that distibution along with $\hat{\Theta}_2$ in place of $\hat{\Theta}_1$ is used in Equation **??** to produce an updated parameter set $\hat{\Theta}_3$. This is repeated until the expected log-likelihood stops increasing (or increases less than some set tolerance level).

Implementing this algorithm is straight-forward, hence its popularity.

1. Set an initial set of parameters, $\hat{\Theta}_1$
2. E step: using the model for the hidden states ($\mathbf{X}$) and $\hat{\Theta}_1$, calculate the expected values of $\mathbf{X}$ conditioned on all the data $\mathbf{y}_1^T$; this is `xtT` output by the Kalman smoother (function `MARSSkf()`). Also calculate expected values of any functions of $\mathbf{X}$ (or $\mathbf{Y}$ if there are missing $\mathbf{Y}$ values) that appear in your expected log-likelihood function.
3. M step: put those $\mathrm{E}[\mathbf{X}|\mathbf{Y}=\mathbf{y}_1^T,\hat{\Theta}_1]$ and $\mathrm{E}[g(\mathbf{X})|\mathbf{Y}=\mathbf{y}_1^T,\hat{\Theta}_1]$ into your expected log-likelihood function in place of $\mathbf{X}$ (and $g(\mathbf{X})$) and maximize with respect to Θ. This gives you $\hat{\Theta}_2$.
4. Repeat the E and M steps until the log likelihood stops increasing.

The EM equations used in the {MARSS} package (function `MARSSkem()`) are described in Holmes (2012) and are extensions of those in Shumway and Stoffer

[1] You can choose these however you wish, however choosing something not too far off from the correct values will make the algorithm go faster.

(1982) and Ghahramani and Hinton (1996). Our EM algorithm is an extended version because our algorithm is for cases where there are constraints within the parameter matrices (shared values, linear combinations, diagonal structure, block-diagonal structure, ...), where there are fixed values within the parameter matrices, or where there may be 0s on the diagonal of $\mathbf{Q}$, $\mathbf{R}$ and Λ.

The EM algorithm is a hill-climbing algorithm and like all hill-climbing algorithms can get stuck on local maxima. See Chapter 6 for a discussion on how to implement a Monte-Carlo initial conditions search based on Biernacki et al. (2003) to minimize this problem. EM algorithms are also known to get close to the maximum very quickly but then creep toward the absolute maximum. Once in the vicinity of the maximum, quasi-Newton methods find the absolute maximum much faster, but they can be sensitive to initial conditions. In practice, we have found the EM algorithm to be much faster for some types of MARSS models while BFGS is faster for others, so often we will try both.

3.3 Kalman filter and smoother

The Kalman filter (Kalman, 1960) is a recursive algorithm that solves for the expected value of the hidden states (the $\mathbf{X}$) in a MARSS model (Equation 1.1) at time t conditioned on the data up to time t: $\mathrm{E}[\mathbf{X}_t|\mathbf{y}_1^t]$. The Kalman filter gives the optimal (lowest mean square error) estimate of the unobserved $\mathbf{x}_t$ based on the observed data up to time t for this class of linear dynamical system. The Kalman smoother (Rauch et al., 1965) solves for the expected value of the hidden state(s) conditioned on all the data: $\mathrm{E}[\mathbf{X}_t|\mathbf{y}_1^T]$. If the errors in the stochastic process are Gaussian, then the estimators from the Kalman filter and smoother are also the maximum-likelihood estimates.

However, even if the the errors are not Gaussian, the estimators are optimal in the sense that they are estimators with the least variability possible. This robustness is one reason the Kalman filter is so powerful—it provides well-behaving estimates of the hidden states for all kinds of multivariate autoregressive processes, not just Gaussian processes. The Kalman filter and smoother are widely used in time-series analysis, and there are many textbooks covering it and its applications. In the interest of giving the reader a single point of reference, we use Shumway and Stoffer (2006) as our primary reference.

The `MARSSkf()` function provides the Kalman filter and smoother output using one of two algorithms (specified by `fun.kf`). The algorithm in `MARSSkfss()` is that shown in Shumway and Stoffer (2006). This algorithm is not computationally efficient; see Koopman et al. (1999, section 4.3) for a more efficient Kalman filter implementation. The Koopman et al. implementation is provided in the functions `MARSSkfas()` using the {KFAS} package (Helske, 2017). `MARSSkfss()` (and `MARSSkfas()` with a few exceptions) has the following outputs:

`xtt1` The expected value of $\mathbf{X}_t$ conditioned on the data up to time $t-1$.
`xtt` The expected value of $\mathbf{X}_t$ conditioned on the data up to time t.

`xtT` The expected value of $\mathbf{X}_t$ conditioned on all the data from time 1 to T. These are called the smoothed state estimates.

`Vtt1` The variance of $\mathbf{X}_t$ conditioned on the data up to time $t-1$. Denoted P_t^{t-1} in section 6.2 in Shumway and Stoffer (2006).

`Vtt` The variance of $\mathbf{X}_t$ conditioned on the data up to time t. Denoted P_t^t in section 6.2 in Shumway and Stoffer (2006).

`VtT` The variance of $\mathbf{X}_t$ conditioned on all the data from time 1 to T.

`Vtt1T` The lag-one covariance of $\mathbf{X}_t$ and $\mathbf{X}_{t-1}$ conditioned on all the data, 1 to T.

`Kt` The Kalman gain. This is part of the update equations and relates to the amount `xtt1` is updated by the data at time t to produce `xtt`. Not output by `MARSSkfas`.

`J` This is similar to the Kalman gain but is part of the Kalman smoother. See Equation 6.49 in Shumway and Stoffer (2006). Not output by `MARSSkfas`.

`Innov` This has the innovations at time t, defined as $\varepsilon_t \equiv \mathbf{y}_t$-E$[\mathbf{Y}_t]$. These are the residuals, the difference between the data and their predicted values. See Equation 6.24 in Shumway and Stoffer (2006). Not output by `MARSSkfas`.

`Sigma` This has the Σ_t, the variance-covariance matrices for the innovations at time t. This is used for the calculation of confidence intervals, the s.e. on the state estimates and the likelihood. See Equation 6.25 in Shumway and Stoffer (2006) for the Σ_t calculation. Not output by `MARSSkfas`.

`logLik` The log-likelihood of the data conditioned on the model parameters.

3.4 The exact likelihood

The likelihood of the data given a set of MARSS parameters is part of the output of the `MARSSkfss()` and `MARSSkfas()` functions. The likelihood computation is based on the innovations form of the likelihood (Schweppe, 1965) and uses the output from the Kalman filter:

$$\log \mathrm{L}(\Theta|data) = -\frac{N}{2\log(2\pi)} - \frac{1}{2}\left(\sum_{t=1}^{T}\log|\Sigma_t| + \sum_{t=1}^{T}(\varepsilon_t)^\top \Sigma_t^{-1}\varepsilon_t\right) \tag{3.2}$$

where N is the total number of data points, ε_t is the innovations at time t and $|\Sigma_t|$ is the determinant of the innovations variance-covariance matrix at time t. This likelihood function is shown in Equation 6.62 in Shumway and Stoffer (2006). However there are a few differences between the log-likelihood output by `MARSSkf()` and that described in Shumway and Stoffer (2006).

The standard likelihood calculation (Equation 6.62 in Shumway and Stoffer (2006)) is biased when there are missing values in the data, and the missing data modifications discussed in Section 6.4 in Shumway and Stoffer (2006) do not correct for this bias. Harvey (1989), Section 3.4.7, discusses at length that the standard missing values correction leads to an inexact likelihood when there are missing values. The bias is minor if there are few missing values, but it becomes severe as the number of missing values increases. Many ecological datasets have high fractions of missing values and this leads to a very biased likelihood if one uses the inexact

formula. Harvey (1989) provides some non-trivial ways to compute the exact likelihood.

The {MARSS} package uses instead the exact likelihood correction for missing values that is presented in Section 12.3 in Brockwell and Davis (1991). This solution is straight-forward to implement. The correction involves the following changes to ε_t and Σ_t in the Equation 3.2. Suppose the value $y_{i,t}$ is missing. First, the corresponding i-th value of ε_t is set to 0. Second, the i-th diagonal value of Σ_t is set to 1 and the off-diagonal elements on the i-th column and i-th row are set to 0.

3.5 Parametric and innovations bootstrapping

Bootstrapping can be used to construct frequentist confidence intervals on the parameter estimates (Stoffer and Wall, 1991) and to compute the small-sample AIC corrector for MARSS models (Cavanaugh and Shumway, 1997); the functions `MARSSparamCIs()` and `MARSSaic()` do these computations.

The `MARSSboot()` function provides both parametric and innovations bootstrapping of MARSS models. The innovations bootstrap algorithm by Stoffer and Wall (1991) bootstraps the model residuals (the innovations). This is a semi-parametric bootstrap since is uses, partially, the maximum-likelihood parameter estimates. This algorithm cannot be used if there are missing values in the data. Also for short time series, it gives biased bootstraps because one cannot resample the first few innovations.

`MARSSboot()` also provides a fully parametric bootstrap. This uses the maximum-likelihood MARSS parameters to simulate data from which bootstrap parameter estimates are obtained. Our research (Holmes and Ward, 2010) indicates that this provides unbiased bootstrap parameter estimates, and it works with datasets with missing values. Lastly, `MARSSboot()` can also output parameters sampled from the Hessian matrix.

3.6 Simulation and forecasting

The `MARSSsimulate()` function simulates from a fitted marssMLE object (e.g., output from a `MARSS()` call). It uses `rmvnorm()` (in package {mvtnorm}) to produce draws of the process and observation errors from multivariate normal distributions for each time step.

3.7 Model selection

The package provides the `MARSSaic()` function (accessed with `AIC()`) for computing AIC, AICc and AICb. The latter is a small-sample corrector for autoregressive state-space models. The bias problem with AIC and AICc for short time-series data has been shown in Cavanaugh and Shumway (1997) and Holmes and Ward (2010).

AIC and AICc tend to select overly complex MARSS models when the time-series data are short. AICb corrects this bias. The algorithm for a non-parametric AICb is given in Cavanaugh and Shumway (1997). Their algorithm uses the innovations bootstrap (Stoffer and Wall, 1991), which means it cannot be used when there are missing data. We added a parametric AICb (Holmes and Ward, 2010), which uses a parametric bootstrap. This algorithm allows one to compute AICb when there are missing data and it provides unbiased AIC even for short time series. See Holmes and Ward (2010) for discussion and testing of parametric AICb for MARSS models.

AICb is comprised of the familiar AIC fit term, $-2\log L$, plus a penalty term that is the mean difference between the log likelihood the data under the bootstrapped maximum-likelihood parameter estimates and the log likelihood of the data under the original maximum-likelihood parameter estimate:

$$AICb = -2\log \mathrm{L}(\hat{\Theta}|\mathbf{y}) + 2\left(\frac{1}{N_b}\sum_{i=1}^{N_b} -\log\frac{\mathrm{L}(\hat{\Theta}^*(i)|\mathbf{y})}{\mathrm{L}(\hat{\Theta}|\mathbf{y})}\right) \tag{3.3}$$

where $\hat{\Theta}$ is the maximum-likelihood parameter set under the original data $\mathbf{y}$, $\hat{\Theta}^*(i)$ is a maximum-likelihood parameter set estimated from the i-th bootstrapped data set $\mathbf{y}^*(i)$, and N_b is the number of bootstrap data sets. It is important to notice that the likelihood in the AICb equation is $\mathrm{L}(\hat{\Theta}^*|\mathbf{y})$ not $\mathrm{L}(\hat{\Theta}^*|\mathbf{y}^*)$. In other words, we are taking the average of the likelihood of the original data given the bootstrapped parameter sets.

Part II

Fitting models with {MARSS}

4

The `MARSS()` function

From the user perspective, the main package function is `MARSS()`. This fits a MARSS model (Equation 1.1) to a matrix of data. The function call takes the form:

```
MARSS(data, model=list(), form="marxss"))
```

The model argument is a list with names `B`, `U`, `C`, `c`, `Q`, `Z`, `A`, `D`, `d`, `R`, `x0`, `V0`. Elements can be left off to use default values. The form argument tells `MARSS()` how to use the model list elements. The default is `form="marxss"` which is the model in Equation 1.1.

The data must be passed in as a $n \times T$ matrix (time goes across columns) or a ts object or vector (which will be converted to a $n \times T$ matrix with time across the columns). A data matrix consisting of three time series ($n = 3$) with six time steps might look like

$$\mathbf{y} = \begin{bmatrix} 1 & 2 & NA & NA & 3.2 & 8 \\ 2 & 5 & 3 & NA & 5.1 & 5 \\ 1 & NA & 2 & 2.2 & NA & 7 \end{bmatrix}$$

where NA denotes a missing value.

The argument `model` specifies the structure of the MARSS model. It is a list, where the list elements for each model parameter specify the form of that parameter.

The most general way to specify model structure is to use a list matrix. The list matrix allows one to combine fixed and estimated elements in one's parameter specification. It allows a one-to-one correspondence between how you write the parameter matrix on paper and how you specify it in R . For example, let's say $\mathbf{Q}$ and $\mathbf{u}$ have the following forms in your model:

$$\mathbf{Q} = \begin{bmatrix} q & 0 & 0 \\ 0 & q & 0 \\ 0 & 0 & 1 \end{bmatrix} \text{ and } \mathbf{u} = \begin{bmatrix} 0.05 \\ u_1 \\ u_2 \end{bmatrix}$$

So $\mathbf{Q}$ is a diagonal matrix with the 3rd variance fixed at 1 and the 1st and 2nd estimated and equal. The 1st element of $\mathbf{u}$ is fixed, and the 2nd and 3rd are estimated and different. You can specify this using a list matrix:

```
Q=matrix(list("q",0,0,0,"q",0,0,0,1),3,3)
U=matrix(list(0.05,"u1","u2"),3,1)
```

If you print out `Q` and `U`, you will see they look exactly like **Q** and **u** written above. MARSS will keep the fixed values fixed and estimate q, $u1$, and $u2$.

List matrices allow the most flexible model structures, but `MARSS()` also has text shortcuts for a number of common model structures. Below, the possible ways to specify each model parameter are shown, using $m = 3$ (the number of hidden state processes) and $n = 3$ (number of observation time series).

4.1 u, a and π model structures

u, **a** and π are all row matrices and the options for specifying their structures are the same. **a** has one special option, `"scaling"` described below. The allowable structures are shown using **u** as an example. Note that you should be careful about specifying shared structure in π because you need to make sure the structure in Λ matches. For example, if you require that all the π values are shared (equal) then Λ cannot be a diagonal matrix since that would be saying that the π values are independent, which they are clearly not if you force them to be equal.

`U=matrix(list(),m,1)`: This is the most general form and allows one to specify fixed and estimated elements in **u**. Each character string in **u** is the name of one of the **u** elements to be estimated. For example if `U=matrix(list(0.01,"u","u"),3,1)`, then **u** in the model has the following structure:

$$\begin{bmatrix} 0.01 \\ u \\ u \end{bmatrix}$$

`U=matrix(c(),m,1)`, where the values in `c()` are all character strings: each character string is the name of an element to be estimated. For example if `U=matrix(c("u1","u1","u2"),3,1)`, then **u** in the model has the following structure:

$$\begin{bmatrix} u_1 \\ u_1 \\ u_2 \end{bmatrix}$$

with two values being estimated. `U=matrix(list("u1","u1","u2"),3,1)` has the same effect.

`U="unequal"` or `U="unconstrained"`: Both of these stings indicate that each element of **u** is estimated. If $m = 3$, then **u** would have the form:

$$\begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix}$$

`U="equal"`: There is only one value in **u**:

$$\begin{bmatrix} u \\ u \\ u \end{bmatrix}$$

`U=matrix(c(),m,1)`, where the values in `c()` all numerical values: **u** is fixed and has no estimated values. If `U=matrix(c(0.01,1,-0.5),3,1)`, then **u** in the model is:

$$\begin{bmatrix} 0.01 \\ 1 \\ -0.5 \end{bmatrix}$$

`U=matrix(list(0.01,1,-0.5),3,1)` would have the same effect.

`U="zero"`: **u** is all zero:

$$\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

The **a** parameter has a special option, `"scaling"`, which is the default behavior. In this case, **a** is treated like a scaling parameter. If there is only one **y** row associated with an **x** row, then the corresponding **a** element is 0. If there are more than one **y** rows associated with an **x** row, then the first **a** element is set to 0 and the others are estimated. For example, say $m = 2$ and $n = 4$ and **Z** looks like the following:

$$\mathbf{Z} = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Then the 1st-3rd rows of **y** are associated with the first row of **x**, and the 4th row of **y** is associated with the last row of **x**. Then if **a** is specified as `"scaling"`, **a** has the following structure:

$$\begin{bmatrix} 0 \\ a_1 \\ a_2 \\ 0 \end{bmatrix}$$

4.2 Q, R, Λ model structures

The possible **Q**, **R**, and Λ model structures are identical, except that **R** is $n \times n$ while **Q** and Λ are $m \times m$. All types of structures can be specified using a list matrix, but there are also text shortcuts for specifying common structures. The structures are shown using **Q** as the example.

`Q=matrix(list(),m,m)`: This is the most general way to specify the parameters and allows there to be fixed and estimated elements. Each character string in the list matrix is the name of one of the **Q** elements to be estimated, and each numerical value is a fixed value. For example if
`Q=matrix(list("s2a",0,0,0,"s2a",0,0,0,"s2b"),3,3)`,
then **Q** has the following structure:

$$\begin{bmatrix} \sigma_a^2 & 0 & 0 \\ 0 & \sigma_a^2 & 0 \\ 0 & 0 & \sigma_b^2 \end{bmatrix}$$

Note that `diag(c("s2a","s2a","s2b"))` will not have the desired effect of producing a matrix with numeric 0s on the off-diagonals. It will have character 0s and MARSS will interpret "0" as the name of an element of **Q** to be estimated. Instead, the following two lines can be used:

```
Q=matrix(list(0),3,3)
diag(Q)=c("s2a","s2a","s2b")
```

`Q="diagonal and equal"`: There is only one process variance value in this case:

$$\begin{bmatrix} \sigma^2 & 0 & 0 \\ 0 & \sigma^2 & 0 \\ 0 & 0 & \sigma^2 \end{bmatrix}$$

`Q="diagonal and unequal"`: There are m process variance values in this case:

$$\begin{bmatrix} \sigma_1^2 & 0 & 0 \\ 0 & \sigma_2^2 & 0 \\ 0 & 0 & \sigma_3^2 \end{bmatrix}$$

`Q="unconstrained"`: There are values on the diagonal and the off-diagonals of **Q** and the variances and covariances are all different:

$$\begin{bmatrix} \sigma_1^2 & \sigma_{1,2} & \sigma_{1,3} \\ \sigma_{1,2} & \sigma_2^2 & \sigma_{2,3} \\ \sigma_{1,3} & \sigma_{2,3} & \sigma_3^2 \end{bmatrix}$$

There are m process variances and $(m^2-m)/2$ covariances in this case, so $(m^2+m)/2$ values to be estimated. Note that variance-covariance matrices are never truly unconstrained since the upper and lower triangles of the matrix must be equal.

`Q="equalvarcov"`: There is one process variance and one covariance:

$$\begin{bmatrix} \sigma^2 & \beta & \beta \\ \beta & \sigma^2 & \beta \\ \beta & \beta & \sigma^2 \end{bmatrix}$$

`Q=matrix(c(), m, m)`, where all values in `c()` are character strings: Each element in **Q** is estimated and each character string is the name of a value to be estimated. Note if $m = 1$, you still need to wrap its value in `matrix()` so that its class is matrix.

`Q=matrix(c(), m, m)`, where all values in `c()` are numeric values: Each element in **Q** is fixed to the values in the matrix.

`Q="identity"`: The **Q** matrix is the identity matrix:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

`Q="zero"`: The **Q** matrix is all zeros:

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Be careful when setting Λ model structures. Mis-specifying the structure of Λ can have catastrophic, but difficult to discern, effects on your estimates. See the comments on priors in Chapter 1.

4.3 B model structures

Like the variance-covariance matrices (**Q**, **R** and Λ), **B** can be specified with a list matrix to allow you to have both fixed and shared elements in the **B** matrix. Character matrices and matrices with fixed values operate the same way as for the variance-covariance matrices. In addition, the same text shortcuts are available: “unconstrained", “identity", “diagonal and equal", “diagonal and unequal", “equalvarcov", and “zero". A fixed **B** can be specified with a numeric matrix, but all eigenvalues must fall within the unit circle; meaning `all(abs(eigen(B)$values)<=1)` must be true.

4.4 Z model

Like **B** and the variance-covariance matrices, **Z** can be specified with a list matrix to allow you to have both fixed and estimated elements in **Z**. If **Z** is a square matrix, many of the same text shortcuts are available: “diagonal and equal", “diagonal and equal", and “equalvarcov". If **Z** is a design matrix[1], then a special shortcut is available using `factor()` which allows you to specify which **y** rows are associated with which **x** rows. See Chapter 5 and the applications chapters for more examples.

[1] a matrix with only 0s and 1s and where the row sums are all equal to 1

`Z=factor(c(1,1,1))`: All **y** time series are observing the same (and only) hidden state trajectory x ($n = 3$ and $m = 1$):

$$\mathbf{Z} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

`Z=factor(c(1,2,3))`: Each time series in **y** corresponds to a different hidden state trajectory. This is the default **Z** model and in this case $n = m$:

$$\mathbf{Z} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

`Z=factor(c(1,1,2))`: The first two time series in **y** corresponds to one hidden state trajectory and the third **y** time series corresponds to a different hidden state trajectory. Here $n = 3$ and $m = 2$:

$$\mathbf{Z} = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$$

The **Z** model can be specified using either numeric or character factor levels. `c(1,1,2)` is the same as `c("north","north","south")`

`Z="identity"`: This is the default behavior. This means **Z** is a $n \times n$ identity matrix and $m = n$. If $n = 3$, it is the same as `Z=factor(c(1,2,3))`.

`Z=matrix(c(), n, m)`, where the elements in `c()` are all strings: Passing in a $n \times m$ character matrix, means that each character string is a value to be estimated. Be careful that you are specifying an identifiable model when using this option.

`Z=matrix(c(), n, m)`, where the elements in `c()` are all numeric: Passing in a $n \times m$ numeric matrix means that **Z** is fixed to the values in the matrix. The matrix must be numeric but it does not need to be a design matrix.

`Z=matrix(list(), n, m)`: Passing in a $n \times m$ list matrix allows you to combine fixed and estimated values in the **Z** matrix. Be careful that you are specifying an identifiable model.

4.5 Default model structures

The defaults for the model arguments in `form="marxss"` are

`Z="identity"` each y in **y** corresponds to one x in **x**
`B="identity"` no interactions among the x's in **x**
`U="unequal"` the u's in **u** are all different
`Q="diagonal and unequal"` process errors are independent but have different variances

`R="diagonal and equal"` the observations are i.i.d.
`A="scaling"` **a** is a set of scaling factors
`C="zero"` and `D="zero"` no inputs.
`c="zero"` and `d="zero"` no inputs.
`x0="unequal"` all initial states are different
`V0="zero"` the initial condition on the states ($\mathbf{x}_0$ or $\mathbf{x}_1$) is fixed but unknown
`tinitx=0` the initial state refers to $t = 0$ instead of $t = 1$.

5

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 multivariate (or vector) autoregressive state-space (MARSS or VARSS) models and want to get started quickly. We provide little explanatory text. Those unfamiliar with MARSS (or VARSS) models might prefer 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:

$$\mathbf{x}_t = \mathbf{B}_t\mathbf{x}_{t-1} + \mathbf{u}_t + \mathbf{C}_t\mathbf{c}_t + \mathbf{G}_t\mathbf{w}_t, \text{ where } \mathbf{w}_t \sim \text{MVN}(0, \mathbf{Q}_t) \tag{5.1a}$$

$$\mathbf{y}_t = \mathbf{Z}_t\mathbf{x}_t + \mathbf{a}_t + \mathbf{D}_t\mathbf{d}_t + \mathbf{H}_t\mathbf{v}_t, \text{ where } \mathbf{v}_t \sim \text{MVN}(0, \mathbf{R}_t) \tag{5.1b}$$

$$\mathbf{x}_1 \sim \text{MVN}(\pi, \Lambda) \text{ or } \mathbf{x}_0 \sim \text{MVN}(\pi, \Lambda) \tag{5.1c}$$

The **c** and **d** are inputs (not estimated). In the examples here, we leave off **c** and **d**. We address including inputs only briefly at the end of the chapter. See Chapter 13 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.

5.0.1 Output from model fits

{MARSS} provides the following functions for output from fitted model objects. These functions output data frames in long form. There are companion functions which return the same information as lists in matrix form.

- `fitted(fit)` Model and state fitted values (predictions). This is the right-side of the **y** and **x** equations without the error terms. Will return confidence and prediction intervals.

Type `RShowDoc("Quick_Examples.R",package="MARSS")` at the R command line to open a file with all the code for the examples in this chapter.

- `tidy(fit)` Parameter estimates and confidence intervals.
- `logLik(fit)`, `AIC(fit)` Log-likelihood and AIC.
- `residuals(fit)` Innovations, smoothations, and contemporaneous model and state residuals.
- `predict(fit)`, `forecast(fit)` Predictions and forecasts. Use `?predict.marssMLE` for information. `ggplot2::autoplot(fr)`, where `fr <- forecast(fit)`, plots the forecasts.
- `plot(fit)`, `ggplot2::autoplot(fit)` A series of informative and diagnostic plots. Individual plots can be selected.
- `stats::tsSmooth(fit, type=...)`, with `...` equal to `"xtT"`, `"xtt"` or `"xtt1"`. Kalman filter and smoother output. Expected value of $\mathbf{X}$ (states) conditioned on all data, data 1 to t or data 1 to $t-1$. `MARSSkf(fit)` returns the same in a list of matrices.
- `stats::tsSmooth(fit, type=...)`, with `...` equal to `"ytT"`, `"ytt"` or `"ytt1"`. These are the expected values of the y (left side of the y equation with the error terms). `MARSShatyt(fit)` returns the same in matrix form. Analogous to `MARSSkf(fit)` but for the $\mathbf{y}$ equation. Most users will likely want `fitted()` which is the model fitted values (expected value of the right side of the $\mathbf{y}$ equation without the error term).

5.1 Fixed and estimated elements in parameter matrices

Suppose one has a MARSS model (Equation 5.1) 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:

```
kem <- 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 $\Lambda = 0$ and treat π (initial $\mathbf{x}$) 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.

5.2 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
```

5.2.1 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()`.

```
kem <- 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(kem$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"`.

```
bfgs <- MARSS(dat, method = "BFGS")
```

```
Success! Converged in 34 iterations.
Function MARSSkfas used for likelihood calculation.

MARSS fit is
Estimation method: BFGS
Estimation converged in 34 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, 7 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.

```
bfgs2 <- MARSS(dat, method = "BFGS", inits = kem$par)
```

The BFGS algorithm now converges in 20 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 A.

```
kem.strange <- MARSS(dat, model = list(tinitx = 1))
```

5.2.2 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:

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

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

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).

```
kem <- 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.

5.2.3 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 **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):

```
kem <- 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 (**B**, **Q**, **R**, and Λ).

5.2.4 Five hidden state processes with a "north" and a "south" 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 **u** and **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 we use the following code:

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

Only **u** and **Q** need to be specified since the other parameters are at their default values.

5.2.5 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)
kem <- MARSS(dat, model = list(U = U, Q = Q, R = R))
```

5.2.6 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 observing that one trajectory. Mathematically, the model is:

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

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

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))
kem <- 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.

5.2.7 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"
kem <- 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.
```

5.2.8 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 **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"
kem <- 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; the `factor` notation is simply a shortcut for making this 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 π). `"diagonal and equal"` can be used as a shortcut for all square matrices (**B**, **Q**, **R**, and Λ).

5.3 Linear constraints

Your model can have simple linear constraints within all the parameters except **Q**, **R** and Λ. For example $1+2a-3b$ is a linear constraint. When entering this value for your matrix, you specify this as `"1+2*a+-3*b"`. NOTE: $+$'s join parts so use `"+-3*b"` to specify $-3b$. Anything after `*` is a parameter. So `1*1` has a parameter called `"1"`. Example, let's specify the following **B**, **Q** and **Z** matrices:

$$\mathbf{U} = \begin{bmatrix} u-0.1 \\ u+0.1 \end{bmatrix} \quad \mathbf{Q} = \begin{bmatrix} q_{11} & 0 \\ 0.01 & 0 \end{bmatrix} \quad \mathbf{Z} = \begin{bmatrix} z_1 - z_2 & 2z_1 \\ 0 & z_1 \\ z_2 & 0 \\ 0 & z_3 \\ 0 & 1+z_3 \end{bmatrix}$$

This would be specified as (notice `"1*z1+-1*z2"` for $z_1 - z_2$):

```
U <- matrix(list("-0.1+1*u","0.1+1*u"),2,1)
Q <- matrix(list("q11",0,0,0.01),2,2)
Z <- matrix(list("1*z1+-1*z2",0,"z2",0,0,"2*z1","z1",0,"z3","1+z3"),5,2)
```

We need to fix **A** if **Z** is estimated.

```
kem <- MARSS(dat, model = list(Z = Z, Q = Q, U = U, A="zero"))
```

5.4 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
Qde <- "diagonal and equal"
kem.tv <- MARSS(dat, model = list(U = Ut, Q = Qde))
```

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
kem.tv <- MARSS(dat, model = list(U = Ut, Q = Qde))
```

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.

5.5 Including inputs (or covariates)

In MARSS models with covariates, the covariates are often treated as inputs and appear as either the **c** or **d** in Equation 5.1, depending on the application. However, more generally, **c** and **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 **C** and **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 **C** and **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.

See Chapter 13 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.

5.6 Printing and summarizing models and model fits

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

```
print(kem)
print(kem$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(kem$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. 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(kem, what = "states", silent = TRUE)
```

The package also includes the common functions for working with the output from fitted models: `residuals(fit)`, `coef(fit)` (the estimated parameters), `fitted(fit)`, `logLik(fit)` and `predict(fit)`.

5.7 Tidy output

The `tidy()` and `glance()` functions will provide summaries as a data.frame for use in further analyses and for passing to `ggplot()`.

```
tidy(kem)
```

```
   term   estimate    std.error     conf.low     conf.up
1 R.diag 0.14015704 0.0247974347  0.091554965 0.188759123
2    U.1 0.04770272 0.0104435216  0.027233790 0.068171643
3 Q.diag 0.00000000 0.0005655539 -0.001108465 0.001108465
4   x0.N 6.92924815 0.1238199246  6.686565560 7.171930745
5   x0.S 5.95499350 0.1710235191  5.619793566 6.290193441
```

```
glance(kem)
```

```
   coef.det     sigma df    logLik      AIC     AICc
1 0.6149352 0.1420352  5 -30.98719 71.97439 72.89747
  convergence errors
1           0      0
```

5.8 Confidence intervals on a fitted model

The function `MARSSparamCIs()` is used to compute confidence intervals with a default α level of 0.05. The default is to compute approximate confidence intervals using 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.

5.8.1 Approximate confidence intervals from the Hessian matrix

The default method for `MARSSparamCIs()` computes approximate confidence intervals using an analytically computed Hessian matrix (Harvey, 1989, section 3.4.5). The call is:

```
kem.with.hess.CIs <- MARSSparamCIs(kem)
```

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 390 iterations.
Log-likelihood: -30.98719
AIC: 71.97439   AICc: 72.89747

      ML.Est  Std.Err   low.CI   up.CI
```

```
R.diag 0.1402 0.024797  0.09155 0.18876
U.1    0.0477 0.010444  0.02723 0.06817
Q.diag 0.0000 0.000566 -0.00111 0.00111
x0.N   6.9292 0.123820  6.68657 7.17193
x0.S   5.9550 0.171024  5.61979 6.29019
Initial states (x0) defined at t=0

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

5.8.2 Confidence intervals from a parametric bootstrap

Use `method="parametric"` to use a parametric bootstrap to compute confidence intervals and bias using a parametric bootstrap. Note, `nboot` should be more like 1000, but it is set low here so the example runs quickly.

```
kem.w.boot.CIs <- MARSSparamCIs(kem, method = "parametric", nboot = 10)
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 390 iterations.
Log-likelihood: -30.98719
AIC: 71.97439   AICc: 72.89747

       ML.Est Std.Err low.CI  up.CI Est.Bias Unbias.Est
R.diag 0.1402 0.01823 0.1021 0.1568  0.00461     0.1448
U.1    0.0477 0.00711 0.0447 0.0658 -0.00491     0.0428
Q.diag 0.0000 0.00000 0.0000 0.0000  0.00000     0.0000
x0.N   6.9292 0.11037 6.6088 6.9122  0.09596     7.0252
x0.S   5.9550 0.11792 5.7593 6.0635  0.06161     6.0166
Initial states (x0) defined at t=0

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

5.9 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(kem, type = "vector")
parvec
```

```
    R.diag        U.1     Q.diag       x0.N       x0.S
0.14015704 0.04770272 0.00000000 6.92924815 5.95499350
```

If you need the parameters as a matrix, use `type = "matrix"`.

5.10 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 `tsSmooth()` and `MARSSkf()` functions. Read the help file (`?MARSSkf`) for details and meanings of the names in the output list. `tsSmooth()` returns a data frame in long form. You need to pass in the type of conditioning you want (on all data, data 1 to t or data 1 to $t-1$).

```
df <- tsSmooth(kem)
head(df)
```

```
  .rownames t .estimate .se
1         N 1  6.976951   0
2         N 2  7.024654   0
3         N 3  7.072356   0
4         N 4  7.120059   0
5         N 5  7.167762   0
6         N 6  7.215464   0
```

`MARSSkf()` returns a list with all the filter and smoother output (including variance matrices) in matrix and array form.

```
kf <- MARSSkf(kem)
names(kf)
```

```
 [1] "xtT"        "VtT"        "Vtt1T"      "x0T"
 [5] "V0T"        "x01T"       "V10T"       "x00T"
 [9] "V00T"       "Vtt"        "Vtt1"       "J"
[13] "J0"         "Kt"         "xtt1"       "xtt"
[17] "Innov"      "Sigma"      "kfas.model" "logLik"
[21] "ok"         "errors"
```

```
# if you only need the logLik,
MARSSkf(kem, only.logLik = TRUE)
```

```
$logLik
[1] -30.98719
```

```
# or
logLik(kem)
```

```
'log Lik.' -30.98719 (df=5)
```

5.11 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 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: 11.6907
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 **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 **Q** and one u value in **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.

5.12 Bootstrap parameter estimates

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

```
boot.params <- MARSSboot(kem,
  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 the Hessian matrix (`sim="hessian"`) or a non-parametric bootstrap (`sim="innovations"`).

5.13 Data simulation

5.13.1 Simulated data from a fitted MARSS model

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

```
sim.data <- MARSSsimulate(kem, nsim = 2, tSteps = 100)$sim.data
```

Then you might want to estimate parameters from the 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] -206.8656
```

5.14 Bootstrap AIC

The function `MARSSaic()` computes a bootstrap AIC for model selection purposes. `output="AICbp"` will 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. `nboot` should be more like 1000.

```
kem.with.AICb <- MARSSaic(kem,
  output = "AICbp",
  Options = list(nboot = 10, silent = TRUE)
)
print(kem.with.AICb)
```

```
MARSS fit is
Estimation method: kem
Convergence test: conv.test.slope.tol = 0.5, abstol = 0.001
Estimation converged in 390 iterations.
Log-likelihood: -30.98719
AIC: 71.97439   AICc: 72.89747   AICbp(param): 69.56102

       Estimate
R.diag   0.1402
U.1      0.0477
Q.diag   0.0000
x0.N     6.9292
x0.S     5.9550
Initial states (x0) defined at t=0

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

5.15 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., fitting a non-stationary model to stationary data or fitting a model that specifies independence of errors or states to data that clearly violate that assumption or fitting a model that implies a particular stationary distribution to data that strongly violate that? 2) Do you have confounded parameters, e.g., two parameters that have the same effect (e.g., 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 (look at the QQplots in `plot(fit)`) 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 **Q** to 0.

6

Setting and searching initial conditions

The EM algorithm is very robust to initial starting conditions however before final results are accepted, they should be tested using other initial conditions. Other times you will want to pass in specific initial conditions because the `MARSS()` function cannot find initial conditions on its own, which is the case for certain models with certain **Z** matrices in particular or you might want to start with initial conditions at the MLEs of another fit. This chapter shows you how to set initial conditions.

The chapter will also cover using a Monte Carlo search over random initial values. This is a brute force method for finding optimal initial conditions. It simply uses random initial conditions and runs the EM algorithm for a number of iterations and selects the initial conditions with the highest log-likelihood after the given number of iterations. In MARSS versions 3.9 and earlier, there was a utility function to perform a Monte Carlo search. However, it is very hard for the function to come up with reasonable random initial conditions for the wide variety of models that MARSS can fit. In MARSS version 3.10, the `MARSSmcinit()` function was removed and replaced with this chapter discussing how to do your own Monte Carlo initial conditions search. The original `MARSSmcinit()` function is included in the R code with this chapter (see footnote).

6.1 Fitting a model with a new set of initial conditions

Fitting a model with a new set of initial conditions is straight-forward with the `MARSS()` function. Simply pass in the argument `inits`. This is illustrated with an example from Chapter 13.

We will fit a model with covariates to phytoplankton data:

```
fulldat <- lakeWAplanktonTrans
years <- fulldat[, "Year"] >= 1965 & fulldat[, "Year"] < 1975
```

Type `RShowDoc("Chapter_inits.R",package="MARSS")` at the R command line to open a file with all the code for this chapter and see a copy of the old `MARSSmcinit()` function.

```
dat <- t(fulldat[years, c("Greens", "Bluegreens")])
the.mean <- apply(dat, 1, mean, na.rm = TRUE)
the.sigma <- sqrt(apply(dat, 1, var, na.rm = TRUE))
dat <- (dat - the.mean) * (1 / the.sigma)
```

The covariates for this example are temperature and total phosphorous.

```
covariates <- rbind(
  Temp = fulldat[years, "Temp"],
  TP = fulldat[years, "TP"]
)
# demean the covariates
the.mean <- apply(covariates, 1, mean, na.rm = TRUE)
covariates <- covariates - the.mean
```

We will fit a model where algal abundance is a random walk without drift and where the observation errors are explained by the covariates plus independent unexplained noise:

```
U <- x0 <- "zero"
Q <- "unconstrained"
d <- covariates
A <- "zero"
D <- "unconstrained"
R <- "diagonal and equal"
model.list <- list(
  U = U, Q = Q, A = A, R = R,
  D = D, d = d, x0 = x0
)
kem <- MARSS(dat, model = model.list)
```

```
Success! abstol and log-log tests passed at 72 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 72 iterations.
Log-likelihood: -236.5911
AIC: 489.1822   AICc: 489.8582

                  Estimate
R.diag              0.0720
Q.(1,1)             0.9946
Q.(2,1)            -0.0290
Q.(2,2)             0.0976
```

```
D.(Greens,Temp)       0.3572
D.(Bluegreens,Temp)   0.2537
D.(Greens,TP)        -0.0215
D.(Bluegreens,TP)     0.0354
Initial states (x0) defined at t=0

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

The `inits` argument can be set either as a list that is the same as that from the following:

```
coef(kem, what = "par")
```

or as a marssMLE object from previous `MARSS()` call for a model with the same structure. 'same structure' means the model list used for the `model` argument is the same and the dimensions of the model (number of states and rows of data) are the same.

6.1.1 Specifying initial conditions as a list

The output from `coef(kem, what="par")` is a list with a column vector of the estimated values for each parameter matrix. Here is the value for **D** and **Q**:

```
out <- coef(kem, what = "par")
out$D
```

```
                        [,1]
(Greens,Temp)      0.35722321
(Bluegreens,Temp)  0.25372912
(Greens,TP)       -0.02151303
(Bluegreens,TP)    0.03544925
```

```
out$Q
```

```
             [,1]
(1,1)  0.99456586
(2,1) -0.02895620
(2,2)  0.09757775
```

`MARSS()` gave names to the **D** and **Q** estimated values. It is important to look at the output from `coef(..., what="par")` before passing in the inits list so that you know where the parameter values fall in the parameter column vector. For example, note that in the **Q** column vector, the variance for the first row in **x** (Greens) is first, the (1,1) element, then the covariance, and last value is the variance of the second row in **x** (Bluegreens) which appears in the (2,2) element of **Q**.

You can pass in `inits` for any of the parameters in `coef(..., what="par")`. You can pass in either a column vector that is the same size as that output by `coef()`,

so for **Q**, the column vectors must be 3×1, or a scalar. If a scalar, then all values in the par column vector will be set to that value—except for **Q**, **R** and Λ which will be set to diagonal matrices with the scalar on the diagonal.

Examples

Pass in an initial value for **Q** that is a diagonal matrix with 1 on the diagonal.

```
inits <- list(Q = 1)
kem <- MARSS(dat, model = model.list, inits = inits)
```

```
Success! abstol and log-log tests passed at 134 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 134 iterations.
Log-likelihood: -236.6064
AIC: 489.2127   AICc: 489.8888

                   Estimate
R.diag               0.0686
Q.(1,1)              1.0079
Q.(2,1)             -0.0296
Q.(2,2)              0.1004
D.(Greens,Temp)      0.3312
D.(Bluegreens,Temp)  0.2531
D.(Greens,TP)       -0.0294
D.(Bluegreens,TP)    0.0345
Initial states (x0) defined at t=0

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

```
# or
inits <- list(Q = matrix(c(1, 0, 1), 3, 1))
kem <- MARSS(dat, model = model.list, inits = inits)
```

```
Success! abstol and log-log tests passed at 134 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 134 iterations.
Log-likelihood: -236.6064
AIC: 489.2127   AICc: 489.8888

                    Estimate
R.diag                0.0686
Q.(1,1)               1.0079
Q.(2,1)              -0.0296
Q.(2,2)               0.1004
D.(Greens,Temp)       0.3312
D.(Bluegreens,Temp)   0.2531
D.(Greens,TP)        -0.0294
D.(Bluegreens,TP)     0.0345
Initial states (x0) defined at t=0

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

Pass in an initial value for **Q** that is this non-diagonal matrix:

$$\begin{bmatrix} 1 & 0.5 \\ 0.5 & 0.7 \end{bmatrix}$$

```
inits <- list(Q = matrix(c(1, 0.5, 0.7), 3, 1))
kem <- MARSS(dat, model = model.list, inits = inits)
```

Pass in an initial value for **Q** and **D**:

```
inits <- list(Q = matrix(c(1, 0.5, 0.7), 3, 1), D = 1)
kem <- MARSS(dat, model = model.list, inits = inits)
```

The initial values for **D** will be all 1s.

Pass in an initial value for **D** set to the value from a previous fit but use default `inits` for everything else:

```
inits <- list(D = coef(kem, what = "par")$D)
kem <- MARSS(dat, model = model.list, inits = inits)
```

6.1.2 Specifying initial conditions using output from another fit

You can also use a `MARSS()` fit as an initial condition. The model must be the same structure. This is typically used when you want to use an EM fit as a start for a BFGS fit if you are using BFGS for the final MLE search.

You can pass in the initial conditions as a list using the `coef()` function.

```
# create the par list from the output
inits <- coef(kem, what = "par")
bfgs <- MARSS(dat, model = model.list, inits = inits, method = "BFGS")
```

Or you can pass in a marssMLE object output from a prior call to `MARSS()`. This is a shortcut for the above call.

```
# create the par list from the output
bfgs <- MARSS(dat, model = model.list, inits = kem, method = "BFGS")
```

6.2 Searching across initial values using a Monte Carlo routine

The EM algorithm is a hill-climbing algorithm and like all hill-climbing algorithms it can get stuck on local maxima and ridges. There are a number approaches to doing a pre-search of the initial conditions space, but a brute force random Monte Carlo search appears to work well (Biernacki et al., 2003). It is slow, but normally sufficient. In our papers on the distributional properties of MARSS parameter estimates, we rarely found that an initial conditions search changed the estimates—except in cases where **Z** and **B** are estimated as unconstrained or when the fraction of missing data in the data set became large. Regardless an initial conditions search should be done before reporting final estimates for an analysis[1].

The idea behind a Monte Carlo search of initial conditions is simple. One simply randomly generates initial conditions, runs the EM algorithm a few iterations (10-20), and saves the log-likelihood at the end of those iterations. The starting initial conditions is selected as the initial conditions that gives the lowest log-likelihood.

The R code included for this chapter includes a function that will do a simple Monte Carlo search using a marssMLE object (output from a `MARSS()` call) and drawing random initial conditions from a uniform distribution or a Wishart distribution for the variance-covariance matrices. The function will not work for all MARSS models but will give you a starting point for setting up your own Monte Carlo search. The function uses a control list to set `numInits`, the number of random initial value draws, `numInitSteps`, the maximum number of EM iterations for each random initial value draw, and `boundsInits`, the bounds for the random distributions. It outputs a list with specifying the initial values that give the lowest log-likelihood.

Here is a simple example of using the function. `numInits` is set low so that the example runs quickly.

```
dat <- t(harborSeal)
dat <- dat[c(2, nrow(dat)), ]
fit1 <- MARSS(dat)
```

```
Success! abstol and log-log tests passed at 59 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
```

[1] It is also a good idea to try `method="BFGS"` to see if this changes the estimates.

```
Estimation converged in 59 iterations.
Log-likelihood: 11.68334
AIC: -9.366688   AICc: -5.366688

                                          Estimate
R.diag                                     0.00653
U.X.CoastalEstuaries                       0.06083
U.X.Georgia.Strait                         0.08278
Q.(X.CoastalEstuaries,X.CoastalEstuaries)  0.02048
Q.(X.Georgia.Strait,X.Georgia.Strait)      0.00889
x0.X.CoastalEstuaries                      7.37351
x0.X.Georgia.Strait                        8.40877
Initial states (x0) defined at t=0

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

```
MCinits <- MARSSmcinit(fit1, control = list(numInits = 10))
```

```
> Starting Monte Carlo Initializations
         |2%      |20%      |40%      |60%      |80%      |100%
Progress: ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
```

```
fit2 <- MARSS(dat, inits = MCinits)
```

```
Success! abstol and log-log tests passed at 50 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 50 iterations.
Log-likelihood: 11.68337
AIC: -9.366739   AICc: -5.366739

                                          Estimate
R.diag                                     0.00652
U.X.CoastalEstuaries                       0.06083
U.X.Georgia.Strait                         0.08278
Q.(X.CoastalEstuaries,X.CoastalEstuaries)  0.02049
Q.(X.Georgia.Strait,X.Georgia.Strait)      0.00889
x0.X.CoastalEstuaries                      7.37351
x0.X.Georgia.Strait                        8.40877
Initial states (x0) defined at t=0
```

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

Part III

Applications

In this part, we walk you through some longer analyses using MARSS models for a variety of different applications. Most of these are analyses of ecological data, but the same models are used in many other fields. These longer examples will take you through both the conceptual steps (with pencil and paper) and a R step which translates the conceptual model into code.

Set-up

- If you haven't already, install the {MARSS} package from CRAN.
- Type in `library(MARSS)` at the R command line to load the package after you install it.

Tips

- `summary(fit$model)`, where `fit = MARSS(...)`, will print detailed information on the structure of the MARSS model. This allows you to double check the model you are fitting.
- `tidy(fit)` will print the parameter estimates with approximate CIs (based on the Hessian).
- `ggplot2::autoplot(fit)` will print a standard set of state-space plots and diagnostic plots.
- When you run `MARSS()`, it will output the number of iterations used. If you reached the maximum, re-run with `control=list(maxit=...)` set higher than the default.
- If you mis-specify the model, `MARSS()` will post an error that should give you an idea of the problem (make sure `silent=FALSE` to see full error reports). Remember, the number of rows in your data is n, time is across the columns, and the length of the vector of factors passed in for `model$Z` must be n while the number of unique factors must be m, the number of x hidden state trajectories in your model.
- The missing value indicator is NA.
- Running `MARSS(data)`, with no arguments except your data, will fit a MARSS model with $m = n$, a diagonal $\mathbf{Q}$ matrix with m variances, and i.i.d. observation errors.
- Try `MARSSinfo()` at the command line if you get errors or warnings you don't understand. You might find insight there. Or look at the warnings and errors notes in Appendix A.

7

Count-based population viability analysis (PVA) using corrupted data

7.1 Background

Estimates of extinction and quasi-extinction risk are an important risk metric used in the management and conservation of endangered and threatened species. By necessity, these estimates are based on data that contain both variability due to real year-to-year changes in the population growth rate (process errors) and variability in the relationship between the true population size and the actual count (observation errors). Classic approaches to extinction risk assume the data have only process error, i.e., no observation error. In reality, observation error is ubiquitous both because of the sampling variability and also because of year-to-year (and day-to-day) variability in sightability.

In this application, we will fit a univariate state-space model to population count data with observation error. We will compute the extinction risk metrics given in Dennis et al. (1991), however instead of using a process-error only model (as is done in the original paper), we use a model with both process and observation error. The risk metrics and their interpretations are the same as in Dennis et al. (1991). The only real difference is how we compute σ^2, the process error variance. However this difference has a large effect on our risk estimates, as you will see.

We use here a density-independent model, a stochastic exponential growth model in log space. This is equivalent to a MARSS model with $\mathbf{B} = 1$. Density-independence is often a reasonable assumption when doing a population viability analysis because we do such calculations for at-risk populations that are either declining or that are well below historical levels (and presumably carrying capacity). In an actual population viability analysis, it is necessary to justify this assumption and if there is reason to doubt the assumption, one tests for density-dependence (Taper and Dennis, 1994) and does sensitivity analyses using state-space models with density-dependence (Dennis et al., 2006).

Type `RShowDoc("Chapter_PVA.R",package="MARSS")` at the R command line to open a file with all the code for the examples in this chapter.

The univariate model is written:

$$x_t = x_{t-1} + u + w_t \qquad \text{where } w_t \sim \mathrm{N}(0, \sigma^2) \tag{7.1}$$

$$y_t = x_t + v_t \qquad \text{where } v_t \sim \mathrm{N}(0, \eta^2) \tag{7.2}$$

where y_t is the logarithm of the observed population size at time t, x_t is the unobserved state at time t, u is the growth rate, and σ^2 and η^2 are the process and observation error variances, respectively. In the R code to follow, σ^2 is denoted `Q` and η^2 is denoted `R` because the functions we are using are also for multivariate state-space models and those models use **Q** and **R** for the respective variance-covariance matrices.

7.2 Simulated data with process and observation error

We will start by using simulated data to see the difference between data and estimates from a model with process error only versus a model that also includes observation error. For our simulated data, we used a decline of 5% per year, process variability of 0.02 (typical for small to medium-sized vertebrates), and a observation variability of 0.05 (which is a bit on the high end). We'll randomly set 10% of the values as missing. Here is the code:

First, set things up:

```
sim.u <- -0.05 # growth rate
sim.Q <- 0.02 # process error variance
sim.R <- 0.05 # non-process error variance
nYr <- 50 # number of years of data to generate
fracmissing <- 0.1 # fraction of years that are missing
init <- 7 # log of initial pop abundance
years <- seq(1:nYr) # sequence 1 to nYr
x <- rep(NA, nYr) # replicate NA nYr times
y <- rep(NA, nYr)
```

Then generate the population sizes using Equation 7.1:

```
x[1] <- init
for (t in 2:nYr) {
  x[t] <- x[t - 1] + sim.u + rnorm(1, mean = 0, sd = sqrt(sim.Q))
}
```

Lastly, add observation error using Equation 7.2 and then add missing values:

```
for (t in 1:nYr) {
  y[t] <- x[t] + rnorm(1, mean = 0, sd = sqrt(sim.R))
}
missYears <- sample(years[2:(nYr - 1)], floor(fracmissing * nYr),
  replace = FALSE
)
y[missYears] <- NA
```

Stochastic population trajectories show much variation, so it is best to look at a few simulated data sets at once. In Figure 7.1, nine simulations from the identical parameters are shown.

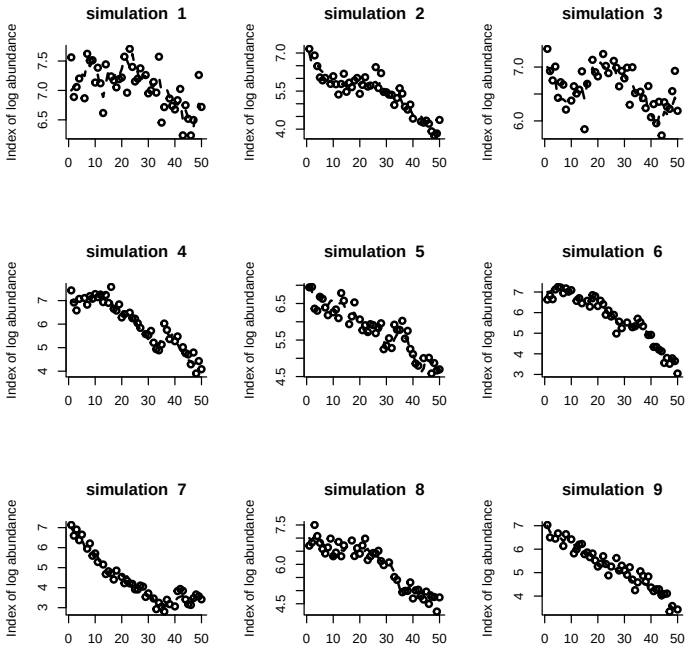

Fig. 7.1. Plot of nine simulated population time series with process and observation error. Circles are observation and the dashed line is the true population size.

Example 7.1 (The effect of parameter values on parameter estimates)

A good way to get a feel for reasonable σ^2 *values is to generate simulated data and look at the time series. A biologist would have a pretty good idea of what kind of year-to-year population changes are reasonable for their study species. For example for many large mammalian species, the maximum population yearly increase would be around 50% (the population could go from 1000 to 1500 in one year), but some fish species could easily double or even triple in a really good year. Observed data may bounce around for many different reasons having to do with sightability, sampling error, age-structure, etc., but the underlying population trajectory is constrained by*

the kinds of year-to-year changes in population size that are biologically possible. σ^2 *describes those true population changes.*

You should run the example code several times using different parameter values to get a feel for how different the time series can look based on identical parameter values. You can cut and paste from the pdf into the R command line. Typical vertebrate σ^2 *values are 0.002 to 0.02, and typical* η^2 *values are 0.005 to 0.1 (Holmes et al., 2007). A u of -0.01 translates to an average 1% per year decline and a u of -0.1 translates to a roughly 10% per year decline.*

Example 7.1 code

```
par(mfrow = c(3, 3))
sim.u <- -0.05
sim.Q <- 0.02
sim.R <- 0.05
nYr <- 50
fracmiss <- 0.1
init <- 7
years <- seq(1:nYr)
for (i in 1:9) {
  x <- rep(NA, nYr) # vector for ts w/o measurement error
  y <- rep(NA, nYr) # vector for ts w/ measurement error
  x[1] <- init
  for (t in 2:nYr) {
    x[t] <- x[t - 1] + sim.u + rnorm(1, mean = 0, sd = sqrt(sim.Q))
  }
  for (t in 1:nYr) {
    y[t] <- x[t] + rnorm(1, mean = 0, sd = sqrt(sim.R))
  }
  missYears <-
    sample(years[2:(nYr - 1)], floor(fracmiss * nYr), replace = FALSE)
  y[missYears] <- NA
  plot(years, y,
    xlab = "", ylab = "Log abundance", lwd = 2, bty = "l"
  )
  lines(years, x, type = "l", lwd = 2, lty = 2)
  title(paste("simulation ", i))
}
legend("topright", c("Observed", "True"),
  lty = c(-1, 2), pch = c(1, -1)
)
```

7.3 Maximum-likelihood parameter estimation

7.3.1 Model with process and observation error

Using the simulated data, we estimate the parameters, u, σ^2, and η^2, and the hidden population sizes. These are the estimates using a model with process and observation variability. The function call is `kem = MARSS(data)`, where `data` is a vector of logged (base e) counts with missing values denoted by NA. After this call, the maximum-likelihood parameter estimates are shown with `coef(kem)`. There are numerous other outputs from the `MARSS()` function. To get a list of the standard model output available type in `?print.MARSS`. Here's code to fit to the simulated time series:

```
kem <- MARSS(y)
```

Let's look at the parameter estimates for the nine simulated time series in Figure 7.1 to get a feel for the variation. The `MARSS()` function was used on each time series to produce parameter estimate for each simulation. The estimates are followed by the mean (over the nine simulations) and the true values:

```
               kem.U        kem.Q      kem.R
sim 1    -0.01372942 0.0006117375 0.08532812
sim 2    -0.05957730 0.0197465734 0.05979038
sim 3    -0.01233111 0.0110145851 0.06824987
sim 4    -0.06190387 0.0215591496 0.05939873
sim 5    -0.03997667 0.0001107659 0.06889955
sim 6    -0.07161604 0.0297689096 0.03361911
sim 7    -0.07266849 0.0322957128 0.03400083
sim 8    -0.04994895 0.0095406636 0.05709867
sim 9    -0.06273875 0.0000000000 0.06161410
mean sim -0.04938785 0.0138497886 0.05866660
true     -0.05000000 0.0200000000 0.05000000
```

As expected, the estimated parameters do not exactly match the true parameters, but the average should be fairly close (although nine simulations is a small sample size). Also note that although we do not get u quite right, our estimates are usually negative. Thus our estimates usually indicate declining dynamics. Some of the `kem.Q` estimates may be 0. This means that the maximum-likelihood estimate that the data are generated by is a process with no environment variation and only observation error.

The MARSS model fit also gives an estimate of the true population size with observation error removed. This is in `kem$states`. Figure 7.2 shows the estimated true states of the population over time as a solid line. Note that the solid line is considerably closer to the actual true states (dashed line) than the observations. On the other hand with certain datasets, the estimates can be quite wrong as well!

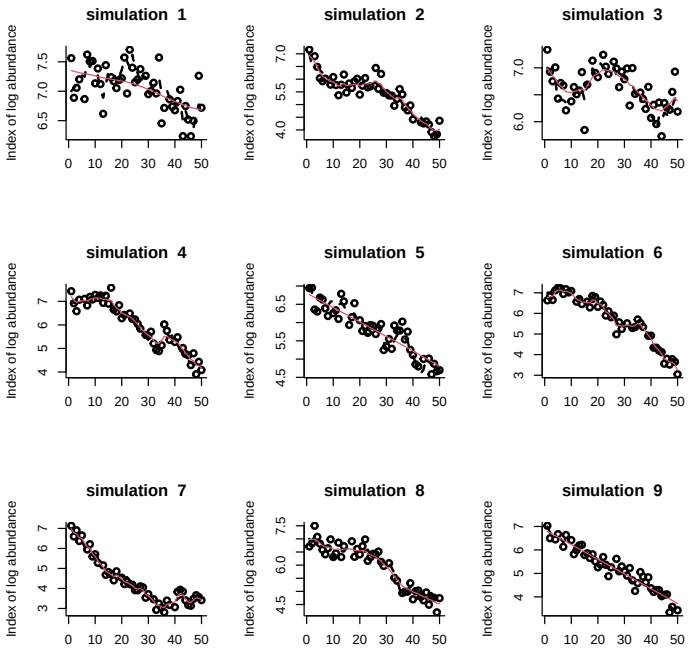

Fig. 7.2. The circles are the observed population sizes with error. The dashed lines are the true population sizes. The solid thin lines are the estimates of the true population size from the MARSS model. When the process error variance is 0, these lines are straight.

7.3.2 Model with no observation error

We used the MARSS model to estimate the mean population rate u and process variability σ^2 under the assumption that the count data have observation error. However, the classic approach to this problem, referred to as the "Dennis model" (Dennis et al., 1991), uses a model that assumes the data have no observation error (a MAR model); all the variability in the data is assumed to result from process error. This approach works well if the observation error in the data is low, but not so well if the observation error is high. We will next fit the data using the classic approach so that we can compare and contrast parameter estimates from the different methods.

Using the estimation method in Dennis et al. (1991), our data need to be re-specified as the observed population changes (`delta.pop`) between censuses along with the time between censuses (`tau`). We re-specify the data as follows:

```
den.years <- years[!is.na(y)] # the non missing years
den.y <- y[!is.na(y)] # the non missing counts
den.n.y <- length(den.years)
delta.pop <- rep(NA, den.n.y - 1) # population transitions
```

```
tau <- rep(NA, den.n.y - 1) # step sizes
for (i in 2:den.n.y) {
  delta.pop[i - 1] <- den.y[i] - den.y[i - 1]
  tau[i - 1] <- den.years[i] - den.years[i - 1]
} # end i loop
```

Next, we regress the changes in population size between censuses (`delta.pop`) on the time between censuses (`tau`) while setting the regression intercept to 0. The slope of the resulting regression line is an estimate of u, while the variance of the residuals around the line is an estimate of σ^2. The regression is shown in Figure 7.3. Here is the code to do that regression:

```
den91 <- lm(delta.pop ~ -1 + tau)
# note: the "-1" specifies no intercept
den91.u <- den91$coefficients
den91.Q <- var(resid(den91))
# type summary(den91) to see other info about our regression fit
```

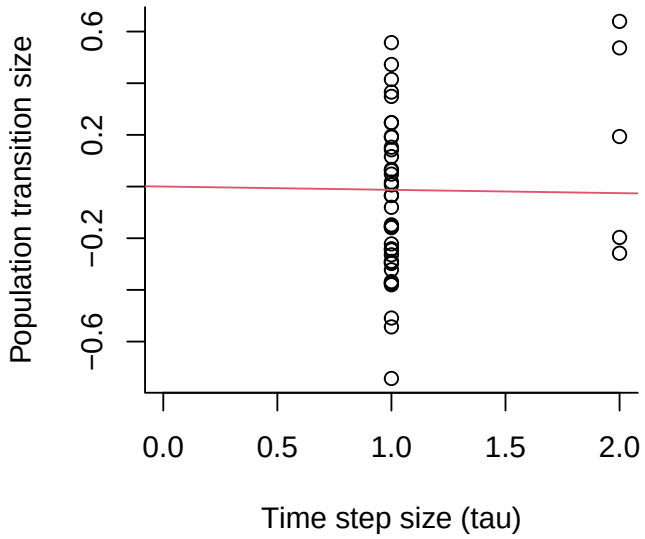

Fig. 7.3. The regression of $log(N_{t+\tau}) - log(N_t)$ against τ. The slope is the estimate of u and the variance of the residuals is the estimate of σ^2. The regression is constrained to go through (0,0).

Here are the parameter values for the data in Figure 7.2 using the process-error only model:

```
              den91.U    den91.Q
sim 1    -0.009908040 0.17834080
sim 2    -0.045790281 0.14306116
sim 3     0.002251094 0.17337902
sim 4    -0.068824821 0.14556544
sim 5    -0.051882180 0.09800453
sim 6    -0.074047428 0.10863073
sim 7    -0.088664176 0.10735425
sim 8    -0.041228813 0.13319684
sim 9    -0.071083959 0.10881767
mean sim -0.049908734 0.13292783
true     -0.050000000 0.02000000
```

Notice that the *u* estimates are similar to those from MARSS model, but the σ^2 estimate (`Q`) is much larger. That is because this approach treats all the variance as process variance, so any observation variance in the data is lumped into process variance. The additional variance added is two times the observation variance.

Example 7.2 (The variability in parameter estimates)

In this example, we will look at how variable the parameter estimates are by generating multiple simulated data sets and then estimating parameter values for each. This example compares the MARSS estimates to the estimates using a process error only model, i.e., ignoring the observation error.

Run the example code a few times to compare the estimates using a state-space model (`kem`*) versus the model with no observation error (*`den91`*). Next, change the observation variance in the code,* `sim.R`*, in the data generation step in order to get a feel for the estimation performance as observations are further corrupted. What happens as observation error is increased? Next, decrease the number of years of data,* `nYr`*, and re-run the parameter estimation. What is the effect of fewer years of data? If you find that the example code takes too long to run, reduce the number of simulations by reducing* `nsim` *in the code.*

Example 7.2 code

```
sim.u <- -0.05 # growth rate
sim.Q <- 0.02 # process error variance
sim.R <- 0.05 # non-process error variance
nYr <- 50 # number of years of data to generate
fracmiss <- 0.1 # fraction of years that are missing
init <- 7 # log of initial pop abundance (~1100 individuals)
nsim <- 9
years <- seq(1:nYr) # col of years
params <- matrix(NA,
  nrow = (nsim + 2), ncol = 5,
  dimnames = list(
    c(paste("sim", 1:nsim), "mean sim", "true"),
    c("kem.U", "den91.U", "kem.Q", "kem.R", "den91.Q")
  )
)
x.ts <- matrix(NA, nrow = nsim, ncol = nYr) # ts w/o measurement error
y.ts <- matrix(NA, nrow = nsim, ncol = nYr) # ts w/ measurement error
for (i in 1:nsim) {
  x.ts[i, 1] <- init
  for (t in 2:nYr) {
    x.ts[i, t] <- x.ts[i, t - 1] + sim.u + rnorm(1, mean = 0, sd = sqrt(sim.Q))
  }
  for (t in 1:nYr) {
    y.ts[i, t] <- x.ts[i, t] + rnorm(1, mean = 0, sd = sqrt(sim.R))
  }
  missYears <- sample(years[2:(nYr - 1)], floor(fracmiss * nYr),
    replace = FALSE
  )
  y.ts[i, missYears] <- NA

  # MARSS estimates
  kem <- MARSS(y.ts[i, ], silent = TRUE)
  # type=vector outputs the estimates as a vector instead of a list
  params[i, c(1, 3, 4)] <- coef(kem, type = "vector")[c(2, 3, 1)]

  # Dennis et al 1991 estimates
  den.years <- years[!is.na(y.ts[i, ])] # the non missing years
  den.yts <- y.ts[i, !is.na(y.ts[i, ])] # the non missing counts
  den.n.yts <- length(den.years)
  delta.pop <- rep(NA, den.n.yts - 1) # transitions
  tau <- rep(NA, den.n.yts - 1) # time step lengths
  for (t in 2:den.n.yts) {
    delta.pop[t - 1] <- den.yts[t] - den.yts[t - 1] # transitions
    tau[t - 1] <- den.years[t] - den.years[t - 1] # time step length
  } # end i loop
  den91 <- lm(delta.pop ~ -1 + tau) # -1 specifies no intercept
  params[i, c(2, 5)] <- c(den91$coefficients, var(resid(den91)))
}
params[nsim + 1, ] <- apply(params[1:nsim, ], 2, mean)
params[nsim + 2, ] <- c(sim.u, sim.u, sim.Q, sim.R, sim.Q)
```

Here is an example of the output from the Example 7.2 code:

```
print(params, digits = 3)
```

```
           kem.U den91.U   kem.Q  kem.R den91.Q
sim 1    -0.0504 -0.0557 0.03760 0.0582  0.1737
sim 2    -0.0385 -0.0266 0.04167 0.0357  0.1223
sim 3    -0.0311 -0.0331 0.02445 0.0380  0.1215
sim 4    -0.0332 -0.0377 0.00935 0.0543  0.1293
sim 5    -0.0129 -0.0102 0.02040 0.0449  0.1072
sim 6    -0.0548 -0.0580 0.01758 0.0491  0.1200
sim 7    -0.0336 -0.0391 0.01575 0.0581  0.1225
sim 8    -0.0265 -0.0106 0.00639 0.0460  0.1191
sim 9    -0.0298 -0.0270 0.01342 0.0436  0.0864
mean sim -0.0345 -0.0331 0.02074 0.0475  0.1225
true     -0.0500 -0.0500 0.02000 0.0500  0.0200
```

7.4 Probability of hitting a threshold $\Pi(x_d, t_e)$

A common extinction risk metric is 'the probability that a population will hit a certain threshold x_d within a certain time frame t_e – *if the observed trends continue*'. In practice, the threshold used is not $N_e = 1$, which would be true extinction. Often a 'functional' extinction threshold will be used ($N_e >> 1$). Other times a threshold representing some fraction of current levels is used. The latter is used because we often have imprecise information about the relationship between the true population size and what we measure in the field; that is, many population counts are index counts. In these cases, one must use 'fractional declines' as the threshold. Also, extinction estimates that use an absolute threshold (like 100 individuals) are quite sensitive to error in the estimate of true population size. Here, we are going to use fractional declines as the threshold, specifically $p_d = 0.1$ which means a 90% decline.

The probability of hitting a threshold, denoted $\Pi(x_d, t_e)$, is typically presented as a curve showing the probabilities of hitting the threshold (y-axis) over different time horizons (t_e) on the x-axis. Extinction probabilities can be computed through Monte Carlo simulations or analytically using Equation 16 in Dennis et al. (1991) (note there is a typo in Equation 16; the last $+$ is supposed to be a $-$). We will use the latter method:

$$\Pi(x_d, t_e) = \pi(u) \times \Phi\left(\frac{-x_d + |u|t_e}{\sqrt{\sigma^2 t_e}}\right) + \exp(2x_d|u|/\sigma^2)\Phi\left(\frac{-x_d - |u|t_e}{\sqrt{\sigma^2 t_e}}\right) \quad (7.3)$$

where x_e is the threshold and is defined as $x_e = \log(N_0/N_e)$. N_0 is the current population estimate and N_e is the threshold. If we are using fractional declines then

$x_e = \log(N_0/(p_d \times N_0)) = -\log(p_d)$. $\pi(u)$ is the probability that the threshold is eventually hit (by $t_e = \infty$). $\pi(u) = 1$ if $u <= 0$ and $\pi(u) = \exp(-2ux_d/\sigma^2)$ if $u > 0$. $\Phi()$ is the cumulative probability distribution of the standard normal (mean = 0, sd = 1).

Here is the R code for that computation:

```
pd <- 0.1 # means a 90 percent decline
tyrs <- 1:100
xd <- -log(pd)
p.ever <- ifelse(u <= 0, 1, exp(-2 * u * xd / Q)) # Q=sigma2
for (i in 1:100) {
  Pi[i] <- p.ever * pnorm((-xd + abs(u)*tyrs[i])/sqrt(Q*tyrs[i])) +
    exp(2*xd*abs(u)/Q) * pnorm((-xd - abs(u)*tyrs[i])/sqrt(Q*tyrs[i]))
}
```

Figure 7.4 shows the estimated probabilities of hitting the 90% decline for the nine 30-year times series simulated with $u = -0.05$, $\sigma^2 = 0.01$ and $\eta^2 = 0.05$. The dashed line shows the estimates using the MARSS parameter estimates and the solid line shows the estimates using a process-error only model (the `den91` estimates). The circles are the true probabilities. The difference between the estimates and the true probabilities is due to errors in $\hat{u}$. Those errors are due largely to process error—not observation error. As we saw earlier, by chance population trajectories with a $u < 0$ will increase, even over a 50-year period. In this case, $\hat{u}$ will be positive when in fact $u < 0$.

Looking at the figure, it is obvious that the probability estimates are highly variable. However, look at the first panel. This is the average estimate (over nine simulations). Note that on average (over nine simulations), the estimates are good. If we had averaged over 1000 simulations instead of nine, the MARSS line would have fallen on the true line. It is an unbiased predictor. While that may seem a small consolation if estimates for individual simulations are all over the map, it is important for correctly specifying our uncertainty about our estimates. Second, rather than focusing on how the estimates and true lines match up, see if there are any types of forecasts that seem better than others. For example, are 20-year predictions better than 50-year and are 100-year forecasts better or worse. In Example 7.3, we will remake this figure with different u. This demonstrates how forecasts are more certain for populations that are declining faster.

Example 7.3 (The effect of parameter values on risk estimates)

In this example, we will recreate Figure 7.4 using different parameter values. This will illustrate how variability in the data and population process affect the risk estimates. The Example 7.2 code needs to be run before the Example 7.3 code.

Begin by changing `sim.R` *and rerunning the Example 7.2 code. Now run the Example 7.3 code and generate parameter estimates. When are the estimates using the*

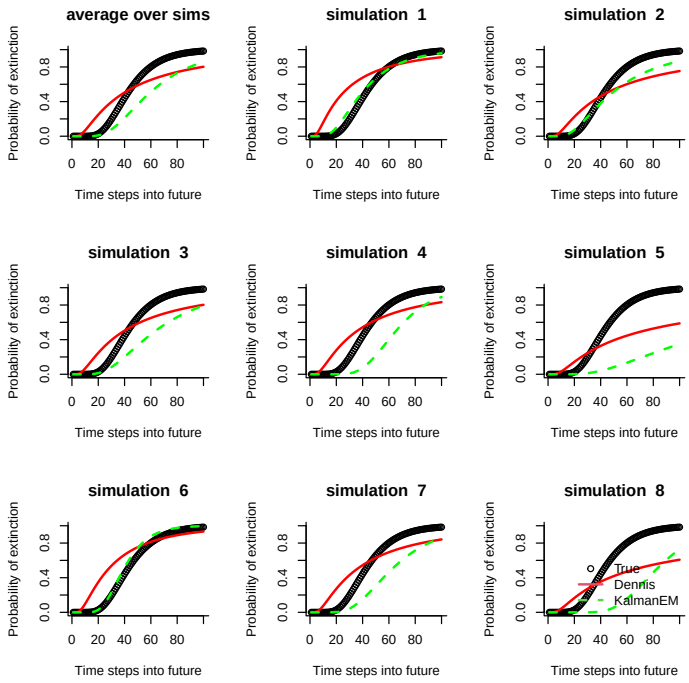

Fig. 7.4. Plot of the true and estimated probability of declining 90% in different time horizons for nine simulated population time series with observation error. The plot may look like a step-function if the σ^2 estimate is very small (<1e-4 or so).

process-error only model (`den91`*) worse and in what way are they worse? You might imagine that you should always use a model that includes observation error, since in practice observations are never perfect. However, there is a cost to estimating that extra variance parameter and the cost is a more variable* σ^2 (`Q`) *estimate. Play with shortening the time series and decreasing the* `sim.R` *values. Are there situations when the 'cost' of the extra parameter is greater than the 'cost' of ignoring observation error?*

Next change the rate of decline in the simulated data. To do this, rerun the Example 7.2 code using a lower `sim.u`*; then run the Example 7.3 code. Do the estimates seem better or worse for rapidly declining populations? Rerun the Example 7.2 code using fewer number of years (*`nYr` *smaller) and increase* `fracmiss`*. Run the Example 7.3 code again. The graphs will start to look peculiar. Why do you think it is doing that? Hint: look at the estimated parameters.*

Last change the extinction threshold (`pd` in the Example 7.3 code). How does changing the extinction threshold change the extinction probability curves? Do not remake the data, i.e., don't rerun the Example 7.2 code.

Example 7.3 code

```
# Needs Example 2 to be run first
par(mfrow = c(3, 3))
pd <- 0.1; xd <- -log(pd) # decline threshold
te <- 100; tyrs <- 1:te # extinction time horizon
for (j in c(10, 1:8)) {
  real.ex <- denn.ex <- kal.ex <- matrix(nrow = te)

  # MARSS parameter estimates
  u <- params[j, 1]; Q <- params[j, 3]
  if (Q == 0) Q <- 1e-4 # just so the extinction calc doesn't choke
  p.ever <- ifelse(u <= 0, 1, exp(-2 * u * xd / Q))
  for (i in 1:100) {
    if (is.finite(exp(2 * xd * abs(u) / Q))) {
      sec.part <- exp(2 * xd * abs(u) / Q) *
        pnorm((-xd - abs(u) * tyrs[i]) / sqrt(Q * tyrs[i]))
    } else { sec.part <- 0 }
    kal.ex[i] <- p.ever * pnorm((-xd + abs(u) * tyrs[i]) / sqrt(Q * tyrs[i])) +
  } # end i loop

  # Dennis et al 1991 parameter estimates
  u <- params[j, 2]; Q <- params[j, 5]
  p.ever <- ifelse(u <= 0, 1, exp(-2 * u * xd / Q))
  for (i in 1:100) {
    denn.ex[i] <- p.ever * pnorm((-xd + abs(u) * tyrs[i]) / sqrt(Q * tyrs[i]))
      exp(2 * xd * abs(u) / Q) *
      pnorm((-xd - abs(u) * tyrs[i]) / sqrt(Q * tyrs[i]))
  } # end i loop

  # True parameter values
  u <- sim.u; Q <- sim.Q
  p.ever <- ifelse(u <= 0, 1, exp(-2 * u * xd / Q))
  for (i in 1:100) {
    real.ex[i] <- p.ever * pnorm((-xd + abs(u) * tyrs[i]) / sqrt(Q * tyrs[i]))
      exp(2 * xd * abs(u) / Q) *
      pnorm((-xd - abs(u) * tyrs[i]) / sqrt(Q * tyrs[i]))
  } # end i loop

  plot(tyrs, real.ex, xlab = "Time steps into future",
    ylab = "Probability of extinction", ylim = c(0, 1), bty = "l")
  if (j <= 8) title(paste("simulation ", j))
  if (j == 10) title("average over sims")
  lines(tyrs, denn.ex, type = "l", col = "red", lwd = 2, lty = 1)
  lines(tyrs, kal.ex, type = "l", col = "green", lwd = 2, lty = 2)
}
legend("bottomright", c("True", "Dennis", "KalmanEM"), pch = c(1, -1, -1),
  col = c(1, 2, 3), lty = c(-1, 1, 2), lwd = c(-1, 2, 2), bty = "n")
```

7.5 Certain and uncertain regions

Example 7.3 illustrates one of the problems with estimates of the probability of hitting thresholds. Looking over the nine simulations, the risk estimates will be on the true line sometimes and other times they are way off. The estimates are highly variable and one should not present only the point estimates of the probability of 90% decline. At the minimum, confidence intervals need to be added (next section), but even with confidence intervals, the probability of hitting declines often does not capture our certainty and uncertainty about extinction risk estimates.

By running Example 7.3, you might have also noticed that there are some time horizons (10, 20 years) for which the estimate are highly certain (the threshold is never hit), while for other time horizons (30, 50 years) the estimates are all over the map. Put another way, you may be able to say with high confidence that a 90% decline will not occur between years 1 to 20 and that by year 100 it most surely will have occurred. However, between the years 20 and 100, you are very uncertain about the risk. The point is that you can be certain about some forecasts while at the same time being uncertain about other forecasts.

One way to show this is to plot the uncertainty as a function of the forecast, where the forecast is defined in terms of the forecast length (number of years) and forecasted decline (percentage). Uncertainty is defined as how much of the 0-1 range your 95% confidence interval covers. Ellner and Holmes (2008) show such a figure (their Figure 1). Figure 7.5 shows a version of this figure that you can produce with the function `CSEGtmufigure(u= val, N= val, s2p= val)`. For the figure, the values $u = -0.05$ which is a 5% per year decline, $N = 25$ so 25 years between the first and last census, and $s_p^2 = 0.01$ are used. The process variability for big mammals is typically in the range of 0.002 to 0.02.

Example 7.4 (Uncertain and certain regions)

Use the Example 7.4 code to re-create Figure 7.5 and get a feel for when risk estimates are more certain and when they are less certain. `N` *are the number of years of data,* `u` *is the mean population growth rate, and* `s2p` *is the process variance.*

Example 7.4 code

```
par(mfrow = c(1, 1))
CSEGtmufigure(N = 50, u = -0.05, s2p = 0.02)
```

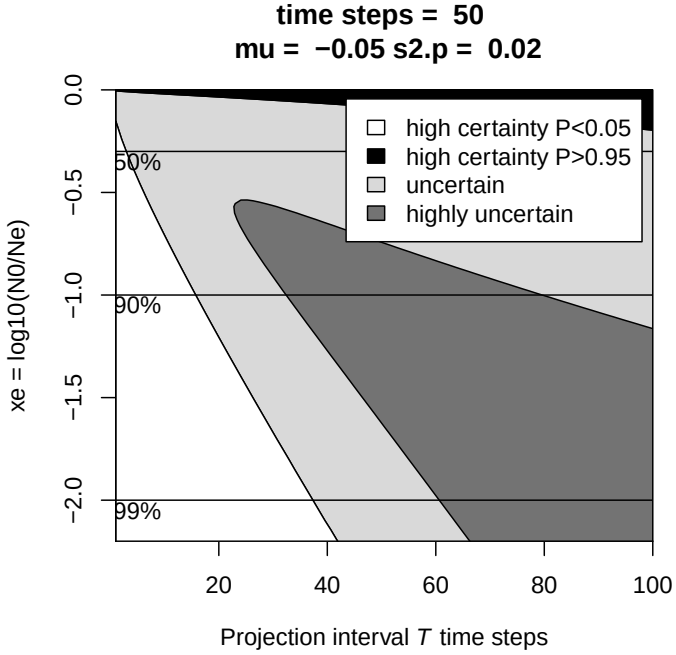

Fig. 7.5. This figure shows the region of high uncertainty (dark gray). In this region, the minimum 95% confidence intervals (meaning no observation error) span 80% of the 0 to 1 probability. That is, we are uncertain if the probability of a specified decline is close to 0 or close to 1. The white area shows where the upper 95% CIs does not exceed P=0.05. In this region, we are quite sure the probability of a specified decline is less than 0.05. The black area shows where the lower 95% confidence interval is above P=0.95. Here we are quite sure the probability is greater than P=0.95. The light gray is between these two certain/uncertain extremes.

7.6 More risk metrics and some real data

The previous sections have focused on the probability of hitting thresholds because this is an important and common risk metric used in population viability analysis and it appears in IUCN Red List criteria. However, there is high uncertainty associated with such estimates. Part of the problem is that probability is constrained to be 0 to 1, and it is easy to get estimates with confidence intervals that span 0 to 1. Other metrics of risk, $\hat{u}$ and the distribution of the time to hit a threshold (Dennis et al., 1991), do not have this problem and may be more informative. Figure 7.6 shows different risk metrics from Dennis et al. (1991) on a single plot. This figure is generated by a call to the function `CSEGriskfigure()`:

```
dat <- read.table(datafile, skip = 1)
dat <- as.matrix(dat)
CSEGriskfigure(dat)
```

The `datafile` is the name of the data file, with years in column 1 and population count (logged) in column 2. `CSEGriskfigure()` has a number of arguments that can be passed in to change the default behavior. The variable `te` is the forecast length (default is 100 years), `threshold` is the extinction threshold either as an absolute number, if `absolutethresh=TRUE`, or as a fraction of current population count, if `absolutethresh=FALSE`. The default is `absolutethresh=FALSE` and `threshold=0.1`. `datalogged=TRUE` means the data are already logged; this is the default.

Example 7.5 (Risk figures for different species)

Use the Example 7.5 code to re-create Figure 7.6. The {MARSS} package includes other data that you can also run: `prairiechicken` *from the endangered Attwater Prairie Chicken,* `graywhales` *from Gerber et al. (1999), and* `grouse` *from the Sharp-tailed Grouse (a species of U.S. federal concern) in Washington State. Note for some of these other datasets, the Hessian matrix cannot be inverted and you will need to use* `CI.method="parametric"`. *The commented lines show how to read in your own data from a tab-delimited text file with a header line.*

Example 7.5 code

```
# If you have your data in a tab delimited file with a header
# This is how you would read it in using file.choose()
# to call up a directory browser.
# However, the package has the datasets for the examples
# dat=read.table(file.choose(), skip=1)
# dat=as.matrix(dat)
dat <- wilddogs
CSEGriskfigure(dat, CI.method = "hessian", silent = TRUE)
```

7.7 Confidence intervals

The figures produced by `CSEGriskfigure()` have confidence intervals (95% and 75%) on the probabilities in the top right panel. A standard way to produce these intervals is via parametric bootstrapping. Here are the steps in a parametric bootstrap:

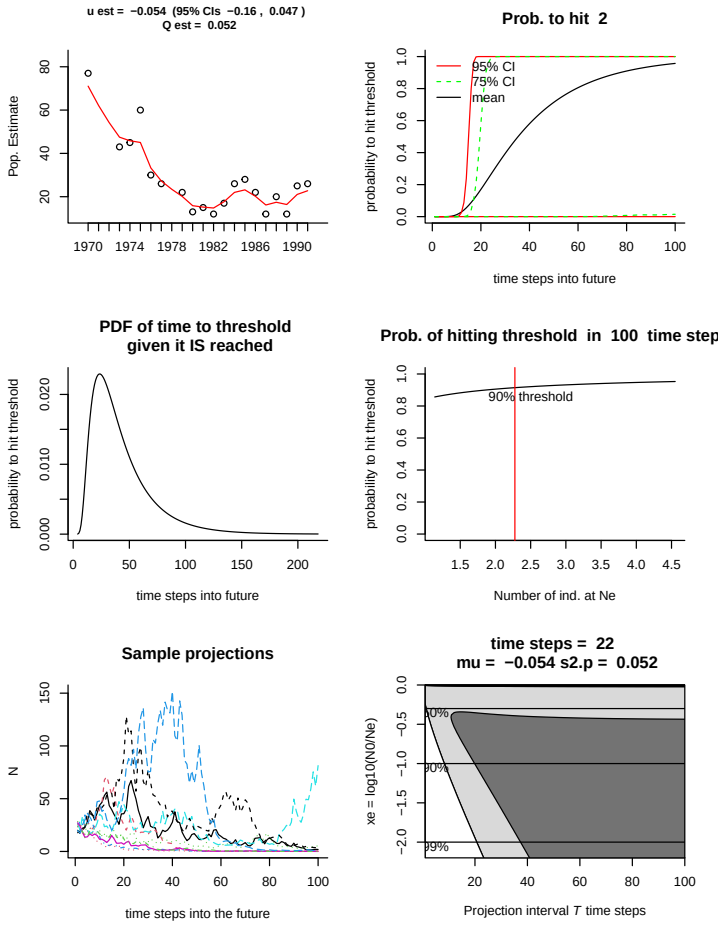

Fig. 7.6. Risk figure using data for the critically endangered African Wild Dog (data from Ginsberg et al. 1995). This population went extinct after 1992.

- Estimate u, σ^2 and η^2
- Simulate time series using those estimates and Equations 7.1 and 7.2
- Re-estimate the model parameters from the simulated data (using say `MARSS(simdata)`)
- Repeat for 1000s of time series simulated using your estimated parameters. This gives a large set of bootstrapped parameter estimates
- For each bootstrapped parameter set, compute a set of extinction estimates (Equation 7.3 and code from Example 7.3)
- The α% ranges on those bootstrapped extinction estimates gives the α confidence intervals on the probabilities of hitting thresholds

The {MARSS} package provides the function `MARSSparamCIs()` to add bootstrapped confidence intervals to fitted models (type `?MARSSparamCIs` to learn about the function).

In the function `CSEGriskfigure()`, you can set `CI.method = c("hessian", "parametric", "innovations", "none")` to tell it how to compute the confidence intervals. The methods 'parametric' and 'innovations' specify parametric and non-parametric bootstrapping respectively. Producing parameter estimates by bootstrapping is quite slow. Approximate confidence intervals on the parameters can be generated rapidly using the inverse of the estimate of the Hessian matrix (method 'hessian'). This uses an estimate of the variance-covariance matrix of the parameters (the inverse of the Hessian matrix). Using an estimated Hessian matrix to compute confidence intervals is a handy trick that can be used for all sorts of maximum-likelihood parameter estimates.

7.8 Discussion

Data with cycles, from age-structure or predator-prey interactions, are difficult to analyze and the EM algorithm used in the {MARSS} package will give poor estimates for this type of data. The slope method (Holmes, 2001) is more robust to those problems. Holmes et al. (2007) used the slope method in a large study of data from endangered and threatened species, and Ellner and Holmes (2008) showed that the slope estimates are close to the theoretical minimum uncertainty. Especially, when doing a population viability analysis using a time series with fewer than 25 years of data, the slope method is often less biased and (much) less variable because that method is less data-hungry (Holmes, 2004). However the slope method is not a true maximum-likelihood method and this constrains the types of further analyses you can do (such as model selection).

8

Combining multi-site data to estimate regional population trends

8.1 Harbor seals in the Puget Sound, WA.

In this application, we will use multivariate state-space models to combine surveys from multiple regions (or sites) into one estimate of the average long-term population growth rate and the year-to-year variability in that growth rate. Note this is not quite the same as estimating the trend; "trend" often means "what population change happened?", whereas the long-term population growth rate refers to the underlying population dynamics. We will use as our example a dataset from harbor seals in Puget Sound, Washington, USA.

We have five regions (or sites) where harbor seals were censused from 1978-1999 while hauled out of land Jeffries et al. (2003). During the period of this dataset, harbor seals were recovering steadily after having been reduced to low levels by hunting prior to protection. The methodologies were consistent throughout the 20 years of the data but we do not know what fraction of the population that each region represents nor do we know the observation-error variance for each region. Given differences between behaviors of animals in different regions and the numbers of haul-outs in each region, the observation errors may be quite different. The regions have had different levels of sampling; the best sampled region has only 4 years missing while the worst has over half the years missing (Figure 8.1).

We will assume that the underlying population process is a stochastic exponential growth process with rates of increase that were not changing through 1978-1999. However, we are not sure if all five regions sample a single "total Puget Sound" population or if there are independent subpopulations. We will estimate the long-term population growth rate using different assumptions about the population structures (one big population versus multiple smaller ones) and observation error structures to see how different assumptions change the trend estimates.

Type `RShowDoc("Chapter_SealTrend.R",package="MARSS")` at the R command line to open a file with all the code for the examples in this chapter.

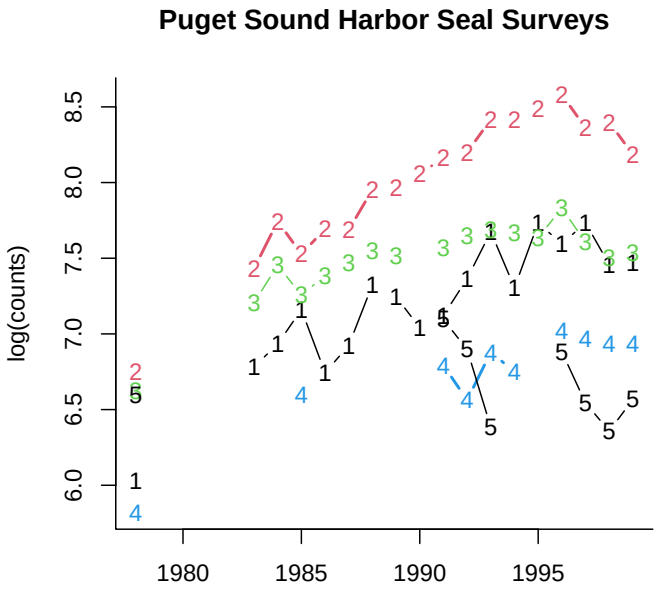

Fig. 8.1. Plot of the of the count data from the five harbor seal regions (Jeffries et al. 2003). The numbers on each line denote the different regions: 1) Strait of Juan de Fuca (SJF), 2) San Juan Islands (SJI), 2) Eastern Bays (EBays), 4) Puget Sound (PSnd), and 5) Hood Canal (HC). Each region is an index of the total harbor seal population, but the bias (the difference between the index and the true population size) for each region is unknown.

The harbor seal data are included in the {MARSS} package. The data have time running down the rows and years in the first column. We need time across the columns for the `MARSS()` function, so we will transpose the data:

```
dat <- t(harborSealWA) # Transpose
years <- dat[1, ] # [1,] means row 1
n <- nrow(dat) - 1
dat <- dat[2:nrow(dat), ] # no years
```

The years are in column 1 of `dat` and the logged data are in the rest of the columns. The number of observation time series (n) is the number of rows in `dat` minus 1 (for years row). Let's look at the first few years of data:

```
print(harborSealWA[1:8, ], digits = 3)
```

```
     Year  SJF  SJI EBays PSnd  HC
[1,] 1978 6.03 6.75  6.63 5.82 6.6
```

```
[2,] 1979   NA   NA    NA   NA  NA
[3,] 1980   NA   NA    NA   NA  NA
[4,] 1981   NA   NA    NA   NA  NA
[5,] 1982   NA   NA    NA   NA  NA
[6,] 1983 6.78 7.43  7.21   NA  NA
[7,] 1984 6.93 7.74  7.45   NA  NA
[8,] 1985 7.16 7.53  7.26 6.60  NA
```

The `NA`'s in the data are missing values.

8.1.1 A MARSS model for Puget Sound harbor seals

The first step is to mathematically specify the population structure and how the regions relate to that structure. The general state-space model is

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

where all the bolded symbols are matrices. To specify the structure of the population and observations, we will specify what those matrices look like.

8.2 A single well-mixed population with i.i.d. errors

When we are looking at data over a large geographic region, we might make the assumption that the different census regions are measuring a single population if we think animals are moving sufficiently such that the whole area (multiple regions together) is "well-mixed". We write a model of the total population abundance for this case as:

$$n_t = \exp(u + w_t) n_{t-1}, \tag{8.1}$$

where n_t is the total count in year t, u is the mean population growth rate, and w_t is the deviation from that average in year t. We then take the log of both sides and write the model in log space:

$$x_t = x_{t-1} + u + w_t, \text{ where } w_t \sim \text{N}(0, q) \tag{8.2}$$

$x_t = \log n_t$. When there is one effective population, there is one x, therefore $\mathbf{x}_t$ is a 1×1 matrix. There is one population growth rate (u) and there is one process variance (q). Thus $\mathbf{u}$ and $\mathbf{Q}$ are 1×1 matrices.

8.2.1 The observation process

We assume that all five regional time series are observations of this one population trajectory but they are scaled up or down relative to that trajectory. In effect, we think that animals are moving around and our regional samples are some fraction

of the population. There is year-to-year variation in the fraction in each region, just by chance. Notice that under this analysis, we do not think the regions represent independent subpopulations but rather independent observations of one population. Our model for the data, $\mathbf{y}_t = \mathbf{Z}\mathbf{x}_t + \mathbf{a} + \mathbf{v}_t$, is written as:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix}_t = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} x_t + \begin{bmatrix} 0 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \end{bmatrix} + \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{bmatrix}_t \tag{8.3}$$

Each y_i is the time series for a different region. The a's are the bias between the regional sample and the total population. The a's are scaling (or intercept-like) parameters[1]. We allow that each region could have a unique observation variance and that the observation errors are independent between regions. Lastly, we assume that the observations errors on log(counts) are normal and thus the errors on (counts) are log-normal.[2]

For our first analysis, we assume that the observation variance is equal across regions but the errors are independent. This means we estimate one observation variance instead of five. This is a fairly standard assumption for data that come from the uniform survey methodology.[3]. We specify independent observation errors with identical variances by specifying that the v's come from a multivariate normal distribution with variance-covariance matrix $\mathbf{R}$ ($\mathbf{v} \sim \text{MVN}(0, \mathbf{R})$), where

$$\mathbf{R} = \begin{bmatrix} r & 0 & 0 & 0 & 0 \\ 0 & r & 0 & 0 & 0 \\ 0 & 0 & r & 0 & 0 \\ 0 & 0 & 0 & r & 0 \\ 0 & 0 & 0 & 0 & r \end{bmatrix} \tag{8.4}$$

$\mathbf{Z}$ specifies which observation time series, $y_{i,1:T}$, is associated with which population trajectory, $x_{j,1:T}$. $\mathbf{Z}$ is like a look up table with 1 row for each of the n observation time series and 1 column for each of the m population trajectories. A 1 in row i column j means that observation time series i is measuring state process j. Otherwise

[1] To get rid of the a's, we scale multiple observation time series against each other; thus one a will be fixed at 0. Estimating the bias between regional indices and the total population is important for getting an estimate of the total population size. The type of time-series analysis that we are doing here (trend analysis) is not useful for estimating a's. Instead to get a's one would need some type of mark-recapture data. However, for trend estimation, the a's are not important. The regional observation variance captures increased variance due to a regional estimate being a smaller sample of the total population.

[2] The assumption of normality is not unreasonable since these regional counts are the sum of counts across multiple haul-outs.

[3] By the way, this is not a good assumption for these data since the number haul-outs in each region varies and the regional counts are the sums across all haul-outs in a region. We will change this assumption in the next fit and see that the AIC values decline.

the value in $\mathbf{Z}_{ij} = 0$. Since we have only 1 population trajectory, all the regions must be measuring that one population trajectory. Thus $\mathbf{Z}$ is $n \times 1$:

$$\mathbf{Z} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \tag{8.5}$$

8.2.2 Fitting the model

We have specified the mathematical form of our state-space model. The next step is to fit this model with `MARSS()`. The function call will now look like:

```
kem1 <- MARSS(dat, model = list(Z = Z.model, R = R.model))
```

The `model` list argument tells the `MARSS()` function the model structure, i.e., the form of **Z**, **u**, **Q**, etc. For our first analysis, we only need to set the model structure for **Z** and **R**. Since there is only one population, there is only one **u** and **Q** (they are scalars), so they have no 'structure'.

First we specify the **Z** matrix. We need to tell the `MARSS` function that **Z** is a 5×1 matrix of 1s (as in Equation 8.3). We can do this two ways. We can pass in `Z.model` as a matrix of ones, `matrix(1,5,1)`, just like in Equation 8.3 or we can pass in a vector of five factors, `factor(c(1,1,1,1,1))`. The *i*-th factor specifies which population trajectory the *i*-th observation time series belongs to. Since there is only one population trajectory in this first analysis, we will have a vector of five 1's: every observation time series is measuring the first, and only, population trajectory.

```
Z.model <- factor(c(1, 1, 1, 1, 1))
```

Note, the vector (the `c()` bit) must be wrapped in `factor()` so that `MARSS` recognizes what it is. You can use either numeric or character vectors: `c(1,1,1,1,1)` is the same as `c("PS","PS","PS","PS","PS")`.

Next we specify that the **R** variance-covariance matrix only has terms on the diagonal (the variances) with the off-diagonal terms (the covariances) equal to zero:

```
R.model <- "diagonal and equal"
```

The 'and equal' part specifies that the variances are the same value. We will relax this assumption later.

Code 8.2 shows you how to fit the single population model (Equations 8.2 and 8.3) to the harbor seal data.

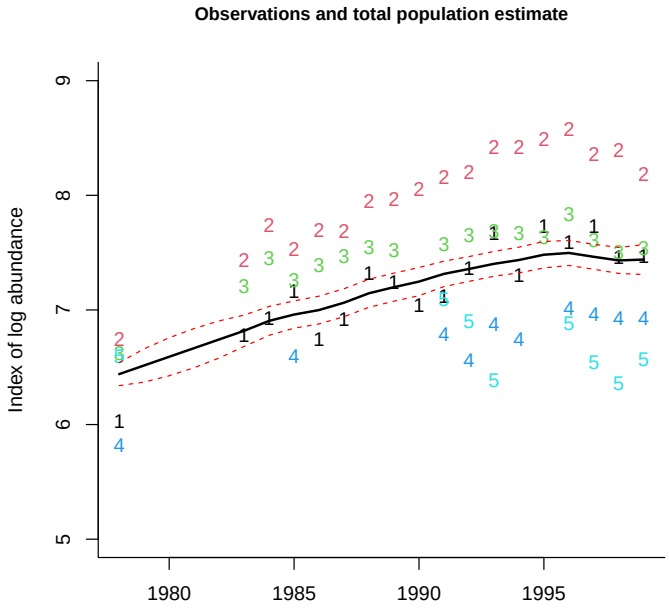

Fig. 8.2. Plot of the estimate of "log total harbor seals in Puget Sound". The estimate of the total count has been scaled relative to the first time series. The 95% confidence intervals on the population estimates are the dashed lines. These are not the confidence intervals on the observations, and the observations (the numbers) will not fall between the confidence interval lines.

Code 8.2

```
# Code to fit the single population model with i.i.d. errors
# Read in data
dat <- t(harborSealWA) # MARSS needs time ACROSS columns
years <- dat[1, ]
n <- nrow(dat) - 1
dat <- dat[2:nrow(dat), ]
legendnames <- (unlist(dimnames(dat)[1]))
# estimate parameters
Z.model <- factor(c(1, 1, 1, 1, 1))
R.model <- "diagonal and equal"
kem1 <- MARSS(dat, model = list(Z = Z.model, R = R.model))
# make figure
graphics::matplot(years, t(dat),
  xlab = "", ylab = "Index of log abundance",
  pch = c("1", "2", "3", "4", "5"), ylim = c(5, 9), bty = "L"
)
lines(years, kem1$states - 1.96 * kem1$states.se,
  type = "l",
  lwd = 1, lty = 2, col = "red"
)
lines(years, kem1$states + 1.96 * kem1$states.se,
  type = "l",
  lwd = 1, lty = 2, col = "red"
)
lines(years, kem1$states, type = "l", lwd = 2)
```

8.2.3 The `MARSS()` output

The output from `MARSS()`, here assigned the name `kem1`, is a list of objects:

```
names(kem1)
```

The maximum-likelihood estimates of "total harbor seal population" scaled to the first observation data series (Figure 8.2) are in `kem1$states`, and `kem1$states.se` are the standard errors on those estimates. To get 95% confidence intervals, use `kem1$states +/- 1.96*kem1$states.se`. Figure 8.2 shows a plot of `kem1$states` with its 95% confidence intervals over the data. Because `kem1$states` has been scaled relative to the first time series, it is on top of that time series. One of the **a** cannot be estimated and arbitrarily our algorithm chooses $a_1 = 0$, so the population estimate is scaled to the first observation time series.

The estimated parameters are output with the function `coef()`: `coef(kem1)`. To get the estimate just for `U`, which is the estimated long-term population growth rate, use `coef(kem1)$U`. Multiply by 100 to get the percent increase per year. The estimated process variance is given by `coef(kem2)$Q`.

The log-likelihood of the fitted model is in `kem1$logLik`. We estimated one initial x ($t = 1$), one process variance, one u, four a's, and five observation variances. So $K = 12$ parameters. The AIC of this model is $-2 \times$ log-like $+ 2K$, which we can show by typing `kem1$AIC`.

8.3 Single population with independent and non-identical errors

Here is the estimated **R** matrix for our first model:

```
coef(kem1, type = "matrix")$R
```

```
             SJF        SJI      EBays       PSnd         HC
SJF   0.04523437 0.00000000 0.00000000 0.00000000 0.00000000
SJI   0.00000000 0.04523437 0.00000000 0.00000000 0.00000000
EBays 0.00000000 0.00000000 0.04523437 0.00000000 0.00000000
PSnd  0.00000000 0.00000000 0.00000000 0.04523437 0.00000000
HC    0.00000000 0.00000000 0.00000000 0.00000000 0.04523437
```

Notice that the variances along the diagonal are all the same—we estimated one observation variance and it applied to all observation time series. We might be able to improve the fit (at the cost of more parameters) by assuming that the observation variance is different across regions while the errors are still independent. This means we estimate five observation variances instead of one. In this case, **R** has the form:

$$\mathbf{R} = \begin{bmatrix} r_1 & 0 & 0 & 0 & 0 \\ 0 & r_2 & 0 & 0 & 0 \\ 0 & 0 & r_3 & 0 & 0 \\ 0 & 0 & 0 & r_4 & 0 \\ 0 & 0 & 0 & 0 & r_5 \end{bmatrix} \tag{8.6}$$

To impose this model, we set the **R** model to

```
R.model <- "diagonal and unequal"
```

This tells `MARSS` that all the r's along the diagonal in $\mathbf{R}$ are different. To fit this model to the data, call `MARSS()` as:

```
Z.model <- factor(c(1, 1, 1, 1, 1))
R.model <- "diagonal and unequal"
kem2 <- MARSS(dat, model = list(Z = Z.model, R = R.model))
```

We estimated one initial x, one process variance, one u, four a's, and five observation variances. So $K = 11$ parameters. The AIC for this new model compared to the old model with one observation variance is:

```
c(kem1$AIC, kem2$AIC)
```

```
[1]  8.813447 -9.323982
```

A smaller AIC means a better model. The difference between the one observation variance versus the unique observation variances is >10, suggesting that the unique observation variances model is better.

One of the key diagnostics when you are comparing fits from multiple models is whether the model is flexible enough to fit the data. This can be checked by looking for temporal trends in the residuals between the fitted data (e.g., the predicted value of the data given the states estimates) and the actual data. These are the smoothations model residuals (as opposed to the innovations model residuals). In Figure 8.3, the residuals for the second analysis are shown. Ideally, these residuals should not have a temporal trend. The fact that the residuals have a strong temporal trend is an indication that our one population model is too restrictive for the data[4]. Code 8.3 shows you how to fit the second model and make the diagnostics plot.

Code 8.3

```
# Fit the single population model with independent and unequal errors
Z.model <- factor(c(1, 1, 1, 1, 1))
R.model <- "diagonal and unequal"
kem2 <- MARSS(dat, model = list(Z = Z.model, R = R.model))
coef(kem2) # the estimated parameter elements
kem2$logLik # log likelihood
c(kem1$AIC, kem2$AIC) # AICs
plot(kem2, plot.type="model.resids.ytT")
```

---

[4] When comparing models via AIC, it is important that you only compare models that are flexible enough to fit the data. Fortunately if you neglect to do this, the inadequate models will usually have very high AICs and fall out of the mix anyhow.

```
plot type = model.resids.ytT
```

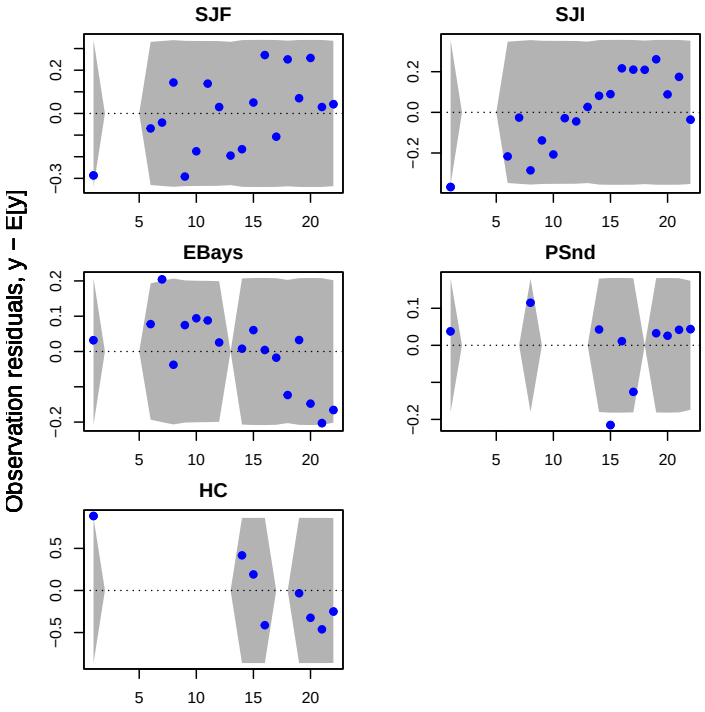

Fig. 8.3. Residuals for the model with a single population. The plots of the residuals should not have trends with time, but they do. This is an indication that the single population model is inconsistent with the data.

8.4 Two subpopulations, north and south

For the third analysis, we will change our assumption about the structure of the population. We will assume that there are two subpopulations, north and south, and that regions 1 and 2 (Strait of Juan de Fuca and San Juan Islands) fall in the north subpopulation and regions 3, 4 and 5 fall in the south subpopulation. For this analysis, we will assume that these two subpopulations share their growth parameter, u, and process variance, q, since they share a similar environment and prey base. However we postulate that because of fidelity to natal rookeries for breeding, animals do not move much year-to-year between the north and south and the two subpopulations are independent.

We need to write down the state-space model to reflect this population structure. There are two subpopulations, x_n and x_s, and they have the same growth rate u:

$$\begin{bmatrix} x_n \\ x_s \end{bmatrix}_t = \begin{bmatrix} x_n \\ x_s \end{bmatrix}_{t-1} + \begin{bmatrix} u \\ u \end{bmatrix} + \begin{bmatrix} w_n \\ w_s \end{bmatrix}_t \tag{8.7}$$

We specify that they are independent by specifying that their year-to-year population fluctuations (their process errors) come from a multivariate normal with no covariance:

$$\begin{bmatrix} w_n \\ w_s \end{bmatrix}_t \sim MVN\left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} q & 0 \\ 0 & q \end{bmatrix}\right) \tag{8.8}$$

For the observation process, we use the **Z** matrix to associate the regions with their respective x_n and x_s values:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix}_t = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_n \\ x_s \end{bmatrix}_t + \begin{bmatrix} 0 \\ a_2 \\ 0 \\ a_4 \\ a_5 \end{bmatrix} + \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{bmatrix}_t \tag{8.9}$$

8.4.1 Specifying the `model` elements

We need to change the **Z** specification to indicate that there are two subpopulations (north and south), and that regions 1 and 2 are in the north subpopulation and regions 3,4 and 5 are in the south subpopulation. There are a few ways, we can specify this **Z** matrix for `MARSS()`:

```
Z.model <- matrix(c(1, 1, 0, 0, 0, 0, 0, 1, 1, 1), 5, 2)
Z.model <- factor(c(1, 1, 2, 2, 2))
Z.model <- factor(c("N", "N", "S", "S", "S"))
```

Which you choose is a matter of preference as they all specify the same form for **Z**.

We also want to specify that the u's are the same for each subpopulation and that **Q** is diagonal with equal q's. To do this, we set

```
U.model <- "equal"
Q.model <- "diagonal and equal"
```

This says that there is one u and one q parameter and both subpopulations share it (if we wanted the u's to be different, we would use `U.model="unequal"` or leave off the **u** model since the default behavior is `U.model="unequal"`).

Code 8.4 puts all the pieces together and shows you how to fit the north and south population model and create the residuals plot (Figure 8.4). The residuals look better (less temporal trend) but the Hood Canal residuals are still have a trend.

Code 8.4

```
# fit the north and south population model
Z.model <- factor(c(1, 1, 2, 2, 2))
U.model <- "equal"
Q.model <- "diagonal and equal"
R.model <- "diagonal and unequal"
kem3 <- MARSS(dat, model = list(
  Z = Z.model,
  R = R.model, U = U.model, Q = Q.model
))
# plot smoothation residuals
plot(kem3, plot.type="model.resids.ytT")
```

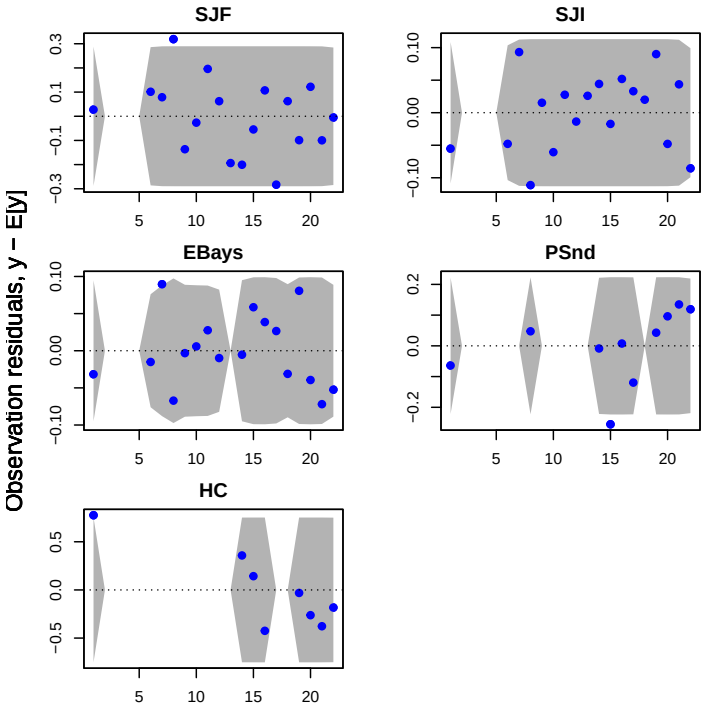

Fig. 8.4. The residuals for the analysis with a north and south subpopulation. The plots of the residuals should not have trends with time. Compare with the residuals for the analysis with one subpopulation.

8.5 Other population structures

Now work through a number of different structures and examine how your estimation of the mean population growth rate varies under different assumptions about the structure of the population and the data. You can compare the model fits using AIC (or AICc). For AIC, lower is better and only the relative differences matter. A difference of 10 between two AICs means substantially more support for the model with lower AIC. A difference of 30 or 40 between two AICs is very large.

8.5.1 Five subpopulations

Analyze the data using a model with five subpopulations, where each of the five census regions is sampling one of the subpopulations. Assume that the subpopulations are independent (diagonal **Q**), however let each subpopulation share the same population parameters, u and q. Code 8.5.1 shows how to set the `MARSS()` arguments for this case. You can use `R.model="diagonal and equal"` to make all the observation variances equal.

Code 8.5.1

```
Z.model <- factor(c(1, 2, 3, 4, 5))
U.model <- "equal"
Q.model <- "diagonal and equal"
R.model <- "diagonal and unequal"
kem <- MARSS(dat, model = list(
  Z = Z.model,
  U = U.model, Q = Q.model, R = R.model
))
```

8.5.2 Two subpopulations with different population parameters

Analyze the data using a model that assumes that the Strait of Juan de Fuca and San Juan Islands census regions represent a northern Puget Sound subpopulation, while the other three regions represent a southern Puget Sound subpopulation. This time assume that each population trajectory (north and south) has different u and q parameters: u_n, u_s and q_n, q_s. Also assume that each of the five census regions has a different observation variance. Try to write your own code. If you get stuck, you can find R code for this model by typing `RShowDoc("Chapter_SealTrend.R",package="MARSS")` at the R command line.

In math form, this model is:

$$\begin{bmatrix} x_n \\ x_s \end{bmatrix}_t = \begin{bmatrix} x_n \\ x_s \end{bmatrix}_{t-1} + \begin{bmatrix} u_n \\ u_s \end{bmatrix} + \begin{bmatrix} w_n \\ w_s \end{bmatrix}_t, \begin{bmatrix} w_n \\ w_s \end{bmatrix}_t \sim \text{MVN}\left(0, \begin{bmatrix} q_n & 0 \\ 0 & q_s \end{bmatrix}\right) \tag{8.10}$$

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix}_t = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_n \\ x_s \end{bmatrix}_t + \begin{bmatrix} 0 \\ a_2 \\ 0 \\ a_4 \\ a_5 \end{bmatrix} + \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{bmatrix}_t \tag{8.11}$$

8.5.3 Hood Canal covaries with the other regions

Analyze the data using a model with two subpopulations with the divisions being Hood Canal versus everywhere else. In math form, this model is:

$$\begin{bmatrix} x_p \\ x_h \end{bmatrix}_t = \begin{bmatrix} x_p \\ x_h \end{bmatrix}_{t-1} + \begin{bmatrix} u_p \\ u_h \end{bmatrix} + \begin{bmatrix} w_p \\ w_h \end{bmatrix}_t, \begin{bmatrix} w_p \\ w_h \end{bmatrix}_t \sim \text{MVN}\left(0, \begin{bmatrix} q & c \\ c & q \end{bmatrix}\right) \tag{8.12}$$

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix}_t = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_p \\ x_h \end{bmatrix}_t + \begin{bmatrix} 0 \\ a_2 \\ a_3 \\ a_4 \\ 0 \end{bmatrix} + \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{bmatrix}_t \tag{8.13}$$

To specify that **Q** has one value on the diagonal (one variance) and one value on the off-diagonal (covariance) you can specify `Q.model` two ways:

```
Q.model <- "equalvarcov"
Q.model <- matrix(c("q", "c", "c", "q"), 2, 2)
```

8.5.4 Three subpopulations with shared parameter values

Analyze the data using a model with three subpopulations as follows: north (regions 1 and 2), south (regions 3 and 4), Hood Canal (region 5). You can specify that some subpopulations share parameters while others do not. First, let's specify that each population is affected by independent environmental variability, but that the variance of that variability is the same for the two interior populations:

```
Q.model <- matrix(list(0), 3, 3)
diag(Q.model) <- c("coastal", "interior", "interior")
print(Q.model)
```

Notice that **Q** is a diagonal matrix (independent year-to-year environmental variability) but the variance of two of the populations is the same. Notice too that the off-diagonal terms are numeric; they do not have quotes. We specified **Q** using a matrix of class list, so that we could have numeric values (fixed) and character values (estimated parameters).

In a similar way, we specify that the observation errors are independent but that estimates from an airplane do not have the same variance as those from a boat:

```
R.model <- matrix(list(0), 5, 5)
diag(R.model) <- c("boat", "boat", "plane", "plane", "plane")
```

MARSS also has a helper function `ldiag()` to make this matrix:

```
R.model <- ldiag(c("boat", "boat", "plane", "plane", "plane"))
```

For the long-term trends, we specify that x_1 and x_2 share a long-term trend ("puget sound") while x_3 is allowed to have a separate trend ("hood canal").

```
U.model <- matrix(c("puget sound", "puget sound", "hood canal"), 3, 1)
```

8.6 Discussion

There are a number of corners that we cut in order to show you code that runs quickly:

- We ran the code starting from one initial condition. For a real analysis, you should start from a large number of random initial conditions and use the one that gives the highest likelihood. Since the EM algorithm is a "hill-climbing" algorithm, this ensures that it did not get stuck on a local maxima. See Chapter 6 for a discussion of initial conditions searchers.
- We assume independent observation and process errors. Depending on your system, observation errors may be driven by large-scale environmental factors (temperature, tides, prey locations) that would cause your observation errors to covary across regions. If your observation errors strongly covary between regions and you treat them as independent, this could be bad for your analysis. Unfortunately, separating covariance across observation versus process errors will require much data (to have any power). In practice, the first step is to think hard about what drives sightability for your species and what are the relative levels of process and observation variance. You may be able to subsample your data in a way that will make the observation errors more independent.
- The `MARSS()` argument `control` specifies the options for the EM algorithm. We left the default tolerance for the convergence test. You would want to set this lower for a real analysis. You will need to up the `maxit` argument correspondingly.
- We used the large-sample approximation for AIC instead of a bootstrap AIC that is designed to correct for small sample size in state-space models. The bootstrap metric, AICb, takes a long time to run. Use the call `MARSSaic(kem, output=c("AICbp"))` to compute AICb. We could have shown AICc, which is the small-sample size corrector for non-state-space models. Type `kem$AICc` to get that.

Finally, in a real (maximum-likelihood) analysis, one needs to be careful not to dredge the data. The temptation is to look at the data and pick a population structure that will fit that data. This can lead to including models in your analysis that have no biological basis. In practice, we spend a great deal of time discussing the population structure with biologists working on the species and review all the biological data that might tell us what are reasonable structures. From that, a set of model structures to use are selected. Other times, a particular model structure needs to be used because the population structure is not in question rather it is a matter of using that prespecified structure and using all the data to get parameter estimates for forecasting.

Some more questions you might ponder

Do different assumptions about whether the observation error variances are all identical versus different affect your estimate of the long-term population growth rate (u)? You may want to rerun Examples 3-7 with the `R.model` changed. `R.model="diagonal and unequal"` means measurement variances all different versus `"diagonal and equal"`.

Do assumptions about the underlying structure of the population affect your estimates of u? Structure here means number of subpopulations and which areas are in which subpopulation.

The confidence intervals for the first two analyses are very tight because the estimated process variance, **Q**, was very small. Why do you think process variance (q) was forced to be so small? Hint: We are forcing there to be one and only one true population trajectory and all the observation time series have to fit that one time series. Look at the AICs too.

9

Identifying spatial population structure and covariance

9.1 Harbor seals on the U.S. west coast

In this application, we use harbor seal abundance estimates along the west coast to examine large-scale spatial structure. Harbor seals are distributed along the west coast of the U.S. from California to Washington. The populations have been surveyed at haul-out sites since the mid-1970s (Figure 9.1) and have been increasing steadily since the 1972 Marine Mammal Protection Act. See `?harborSeal` for the data sources.

For management purposes, three stocks are recognized: the CA stock, the OR/WA coastal stock which consists of four regions (Northern/Southern Oregon, Coastal Estuaries, Olympic Peninsula), and the inland WA stock which consists of the regions in the WA inland waters minus Hood Canal (Figure 9.1). Differences exist in the demographics across regions (e.g., pupping dates), however mtDNA analyses and tagging studies support the larger stock structure. Harbor seals are known for strong site fidelity, but at the same time travel large distances to forage.

Our goal is to address the following questions about spatial structure: 1) Does population abundance data support the existing management boundaries, or are there alternative groupings that receive more support?, 2) Do subpopulations (if they exist) experience independent environmental variability or correlated variability? and 3) Does the Hood Canal site represent a distinct subpopulation? To address these questions, we will mathematically formulate different hypotheses about population structure via different MARSS models. We will then compare the data support for different models using model selection criteria, specifically AICc and AIC weights.

9.1.1 MARSS models for a population with spatial structure

The mathematical form of the model we will use is

Type `RShowDoc("Chapter_SealPopStructure.R",package="MARSS")` at the R command line to open a file with all the code for the examples in this chapter.

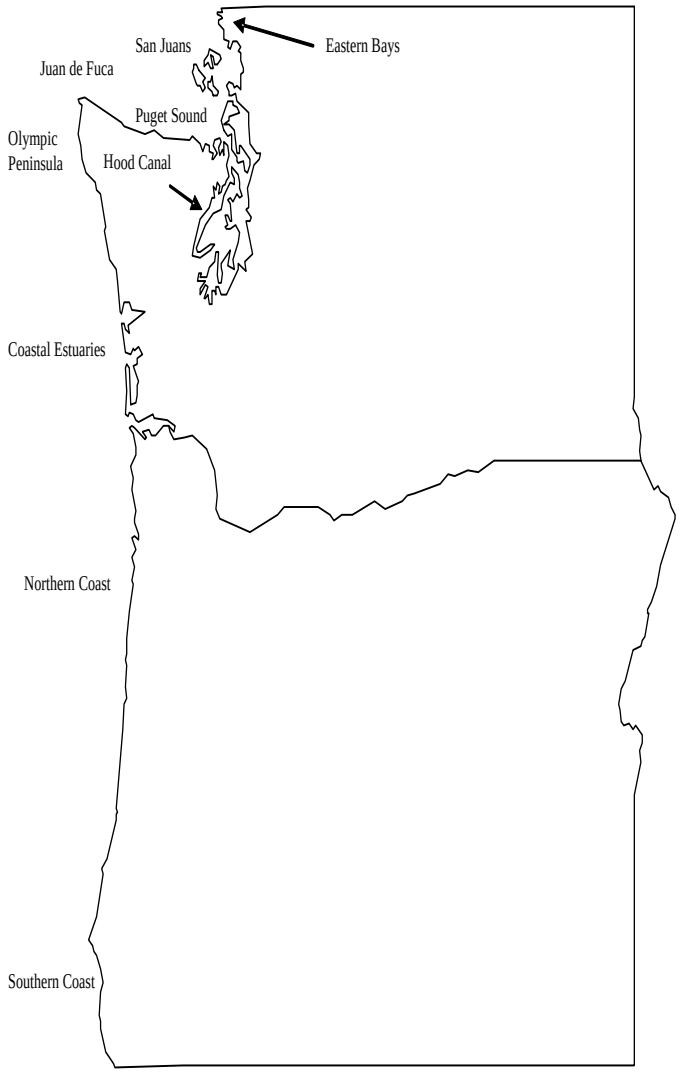

Fig. 9.1. Map of spatial distribution of harbor seal survey regions in Washington and Oregon. In addition to these nine survey regions, we also have data from the Georgia Strait just north of the San Juan Islands, the California coast and the Channels Islands in Southern California.

$$\begin{aligned} \mathbf{x}_t &= \mathbf{x}_{t-1} + \mathbf{u} + \mathbf{w}_t \text{ where } \mathbf{w}_t \sim \text{MVN}(0, \mathbf{Q}) \\ \mathbf{y}_t &= \mathbf{Z}\mathbf{x}_t + \mathbf{a} + \mathbf{v}_t \text{ where } \mathbf{v}_t \sim \text{MVN}(0, \mathbf{R}) \\ \mathbf{x}_0 &\sim \text{MVN}(\pi, \Lambda) \end{aligned} \tag{9.1}$$

B is in front of **x** but is left off above since it is the identity matrix[1]. We will use **Z**, **u**, and **Q** to specify different hypotheses about the population structure. The form of **a** will be "scaling" in all cases. Aerial survey methodology has been relatively constant across time and space, and we will assume that all the time series from each region has identical and independent observation error variance, which means a diagonal **R** matrix with one variance term on the diagonal[2].

Each call to `MARSS()` will look like

```
fit <- MARSS(sealData, model=list(
  Z = Z.model, Q = Q.model, ...))
```

where the `...` are components of the model list that are the same across all models. We will specify different `Z.model` and `Q.model` in order to model different population spatial structures.

9.2 Question 1, How many distinct subpopulations?

We will start by evaluating the data support for the following hypotheses about the population structure:

H1 3 subpopulations defined by stock
H2 2 subpopulations defined by coastal versus WA inland
H3 2 subpopulations defined by north and south split in the middle of Oregon
H4 4 subpopulations defined by N coastal, S coastal, SJF+Georgia Strait, and Puget Sound
H5 All regions are part of the same panmictic population
H6 Each of the 11 regions is a subpopulation

We will analyze each of these under the assumption of independent process errors with each subpopulation having different variances or the same variance.

9.2.1 Specify the Z matrices

The **Z** matrices specify the relationship between the survey regions and the subpopulations and allow us to specify the spatial population structures in the hypotheses. Each column of **Z** corresponds to a different subpopulation and associates regions

[1] a diagonal matrix with 1s on the diagonal

[2] The sampling regions have different number of sites where animals are counted. But we are working with log counts. We assume that the distribution of percent errors is the same (the probability of a 10% over-count is the same) and thus that the variances are similar on the log-scale.

with particular subpopulations. For example for hypothesis 1, column 1 of the **Z** matrix is OR/WA Coastal, column 2 is inland WA (ps for Puget Sound) and column 3 is CA. The **Z** matrix for hypotheses 1, 2, 4, and 5 take the following form:

	H1 **Z**			H2 **Z**		H4 **Z**				H5 **Z**
	wa.or	ps	ca	coast	ps	nc	is	ps	sc	pan
Coastal Estuaries	1	0	0	1	0	1	0	0	0	1
Olympic Peninsula	1	0	0	1	0	1	0	0	0	1
Str. Juan de Fuca	0	1	0	0	1	0	1	0	0	1
San Juan Islands	0	1	0	0	1	0	1	0	0	1
Eastern Bays	0	1	0	0	1	0	0	1	0	1
Puget Sound	0	1	0	0	1	0	0	1	0	1
CA.Mainland	0	0	1	1	0	0	0	0	1	1
CA.ChannelIslands	0	0	1	1	0	0	0	0	1	1
OR North Coast	1	0	0	1	0	1	0	0	0	1
OR South Coast	1	0	0	1	0	0	0	0	1	1
Georgia Strait	0	1	0	0	1	0	1	0	0	1

To tell `MARSS()` the form of **Z**, we construct the same matrix in R. For example, for hypotheses 1, we can write:

```
Z.model <- matrix(0, 11, 3)
Z.model[c(1, 2, 9, 10), 1] <- 1 # which elements in col 1 are 1
Z.model[c(3:6, 11), 2] <- 1 # which elements in col 2 are 1
Z.model[7:8, 3] <- 1 # which elements in col 3 are 1
```

MARSS has a shortcut for making this kind of **Z** matrix using `factor()`. The following code specifies the same **Z** matrix:

```
Z1 <- factor(c("wa.or", "wa.or", rep("ps", 4),
               "ca", "ca", "wa.or", "wa.or", "bc"))
```

Each element in the `c()` vector is for one of the rows of **Z** and indicates which column the "1" appears in or which row of your data belongs to which subpopulation. Notice the vector is 11 elements in length; one element for each row of data (in this case survey region).

9.2.2 Specify the u structure

We will assume that subpopulations can have a unique population growth rate. Mathematically, this means that the **u** matrix in Equation 9.1 looks like this for hypotheses 1 (3 subpopulations):

$$\begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix}$$

To specify this, we construct `U.model` as a character matrix where shared elements have the same character name. For example,

```
U.model <- matrix(c("u1", "u2", "u3"), 3, 1)
```

for a three subpopulation model. Alternatively, we can use the shortcut `U.model="unequal"`.

9.2.3 Specify the Q structures

For our first analysis, we fit a model where the subpopulations experience independent process errors. We will use two different types of independent process errors: independent process errors with different variances and independent process errors with identical variance. Independence is specified with a diagonal variance-covariance matrix with 0s on the off-diagonals.

Independent process errors with different variances is a diagonal matrix with different values on the diagonal:

$$\begin{bmatrix} q_1 & 0 & 0 \\ 0 & q_2 & 0 \\ 0 & 0 & q_3 \end{bmatrix}$$

This matrix has fixed numeric values, the zeros, combined with symbols q_1, q_2 and q_3, representing estimated values. We specified this for `MARSS()` using a list matrix which combines numeric values (the fixed zeros) with character values (names of the estimated elements). The following produces this and printing it shows that it combines numeric values and character strings in quotes.

```
Q.model <- matrix(list(0), 3, 3)
diag(Q.model) <- c("q1", "q2", "q3")
Q.model
```

```
     [,1] [,2] [,3]
[1,] "q1" 0    0
[2,] 0    "q2" 0
[3,] 0    0    "q3"
```

We can also use the shortcut `Q.model="diagonal and unequal"`.

Independent process errors with identical variance is a diagonal matrix with one value on the diagonal:

$$\begin{bmatrix} q & 0 & 0 \\ 0 & q & 0 \\ 0 & 0 & q \end{bmatrix}$$

```
Q.model <- matrix(list(0), 3, 3)
diag(Q.model) <- "q"
Q.model
```

```
     [,1] [,2] [,3]
[1,] "q"  0    0
[2,] 0    "q"  0
[3,] 0    0    "q"
```

The shortcut for this form is `Q.model="diagonal and equal"`.

9.3 Fit the different models

The dataset `harborSeal` is a 29-year dataset of abundance indices for each of 12 regions between 1975-2004 (Figure 9.2). We start by setting up our data matrix. We will leave off Hood Canal (column 8) for now.

```
years <- harborSeal[, 1] # first col is years
# leave off Hood Canal data for now
sealData <- t(harborSeal[, c(2:7, 9:13)])
```

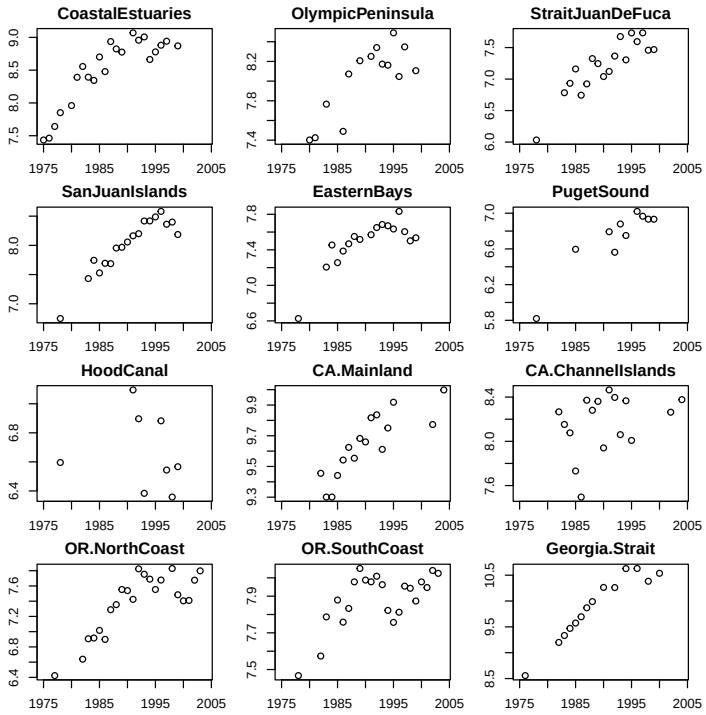

Fig. 9.2. Plot of the of the harbor seal sites in the harborSeal dataset. Each point is an index of the harbor seal abundance in that region.

We will set up our models so we can fit all of them with one loop of code.
First the **Z** models.

```
# H1 stock
Z1 <- factor(c("wa.or", "wa.or", rep("ps", 4),
               "ca", "ca", "wa.or", "wa.or", "bc"))
# H2 coastal+PS
Z2 <- factor(c(rep("coast", 2), rep("ps", 4), rep("coast", 4), "ps"))
# H3 N and S
Z3 <- factor(c(rep("N", 6), "S", "S", "N", "S", "N"))
# H4 North Coast, Inland Strait, Puget Sound, South Coast
Z4 <- factor(c("nc", "nc", "is", "is", "ps", "ps",
               "sc", "sc", "nc", "sc", "is"))
# H5 panmictic
Z5 <- factor(rep("pan", 11))
# H6 Site
Z6 <- factor(1:11) # site
Z.models <- list(Z1, Z2, Z3, Z4, Z5, Z6)
names(Z.models) <-
  c("stock", "coast+PS", "N-S", "NC+Strait+PS+SC", "panmictic", "site")
```

Next we set up the **Q** models.

```
Q.models <- c("diagonal and equal", "diagonal and unequal")
```

The rest of the model matrices have the same form across all models.

```
U.model <- "unequal"
R.model <- "diagonal and equal"
A.model <- "scaling"
B.model <- "identity"
x0.model <- "unequal"
V0.model <- "zero"
model.constant <- list(
  U = U.model, R = R.model, A = A.model,
  x0 = x0.model, V0 = V0.model, tinitx = 0
)
```

We loop through the models, fit and store the results:

```
out.tab <- NULL
fits <- list()
for (i in 1:length(Z.models)) {
  for (Q.model in Q.models) {
    fit.model <- c(list(Z = Z.models[[i]], Q = Q.model), model.constant)
    fit <- MARSS(sealData,
      model = fit.model,
      silent = TRUE, control = list(maxit = 1000)
    )
```

```
    out <- data.frame(
      H = names(Z.models)[i], Q = Q.model, U = U.model,
      logLik = fit$logLik, AICc = fit$AICc, num.param = fit$num.params,
      m = length(unique(Z.models[[i]])),
      num.iter = fit$numIter, converged = !fit$convergence,
      stringsAsFactors = FALSE
    )
    out.tab <- rbind(out.tab, out)
    fits <- c(fits, list(fit))
    if (i == 5) next # one m for panmictic so only run 1 Q
  }
}
```

9.4 Summarize the data support

We will use AICc and AIC weights to summarize the data support for the different hypotheses. First we will sort the fits based on AICc:

```
min.AICc <- order(out.tab$AICc)
out.tab.1 <- out.tab[min.AICc, ]
```

Next we add the ΔAICc values by subtracting the lowest AICc:

```
out.tab.1 <- cbind(out.tab.1,
  delta.AICc = out.tab.1$AICc - out.tab.1$AICc[1]
)
```

Relative likelihood is defined as $\exp(-\Delta\text{AICc}/2)$.

```
out.tab.1 <- cbind(out.tab.1,
  rel.like = exp(-1 * out.tab.1$delta.AICc / 2)
)
```

The AIC weight for a model is its relative likelihood divided by the sum of all the relative likelihoods.

```
out.tab.1 <- cbind(out.tab.1,
  AIC.weight = out.tab.1$rel.like / sum(out.tab.1$rel.like)
)
```

Let's look at the model weights (`out.tab.1`):

```
              H                   Q delta.AICc AIC.weight
NC+Strait+PS+SC   diagonal and equal       0.00      0.886
NC+Strait+PS+SC diagonal and unequal       4.15      0.112
            N-S diagonal and unequal      12.67      0.002
            N-S   diagonal and equal      14.78      0.001
       coast+PS   diagonal and equal      31.23      0.000
```

```
 coast+PS diagonal and unequal      33.36      0.000
    stock   diagonal and equal      34.01      0.000
    stock diagonal and unequal      36.84      0.000
panmictic   diagonal and equal      48.28      0.000
panmictic diagonal and unequal      48.28      0.000
     site   diagonal and equal      56.36      0.000
     site diagonal and unequal      57.95      0.000
```

It appears that a population structure north and south coast subpopulations and two inland subpopulations is more supported than any of the other population structures—under the assumption of independent process errors. The latter means that good and bad years are not correlated across the subpopulations. The stock structure, supported by genetic information, does not appear to correspond to independent subpopulations and the individual survey regions, which are characterized by differential pupping times, does not appear to correspond to independent subpopulations either.

Figure 9.3 shows the the four subpopulation trajectories estimated by the best fit model. The trajectories have been rescaled so that each starts at 0 in 1975 (to facilitate comparison).

9.5 Question 2, Are the subpopulations independent?

The assumption of independent process errors is unrealistic given ocean conditions are correlated across large spatial scales. We will repeat the analysis allowing correlated process errors using two different **Q** models. The first correlated **Q** model is correlated process errors with the same variance and covariance. For a model with three subpopulations, this **Q** would look like:

$$\begin{bmatrix} q & c & c \\ c & q & c \\ c & c & q \end{bmatrix}$$

We can construct this like so

```
#identical variances
Q.model <- matrix("c", 3, 3)
diag(Q.model) <- "q"
```

or use the short-cut `Q.model="equalvarcov"`. The second type of correlated **Q** we will use is allows each subpopulation to have a different process variance and covariances. For a model with three subpopulations, this is the following variance-covariance matrix:

$$\begin{bmatrix} q_1 & c_{1,2} & c_{1,3} \\ c_{1,2} & q_2 & c_{2,3} \\ c_{1,2} & c_{2,3} & q_3 \end{bmatrix}$$

Constructing this is tedious in R, but there is a short-cut: `Q.model="unconstrained"`.

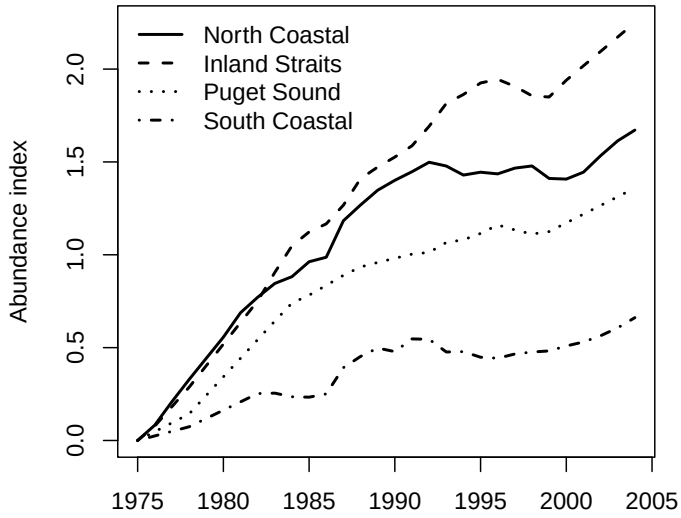

Fig. 9.3. Estimated trajectories for the four subpopulations in the best-fit model. The plots have been rescaled so that each is at 0 at 1975.

We will re-run all the **Z** matrices with these two extra **Q** types and add them to our results table.

```
for (i in 1:length(Z.models)) {
  if (i == 5) next # don't rerun panmictic
  for (Q.model in c("equalvarcov", "unconstrained")) {
    fit.model <- c(list(Z = Z.models[[i]], Q = Q.model), model.constant)
    fit <- MARSS(sealData,
      model = fit.model,
      silent = TRUE, control = list(maxit = 1000)
    )
    out <- data.frame(
      H = names(Z.models)[i], Q = Q.model, U = U.model,
      logLik = fit$logLik, AICc = fit$AICc, num.param = fit$num.params,
      m = length(unique(Z.models[[i]])),
      num.iter = fit$numIter, converged = !fit$convergence,
      stringsAsFactors = FALSE
    )
```

```
    out.tab <- rbind(out.tab, out)
    fits <- c(fits, list(fit))
  }
}
```

Again we sort the models by AICc and compute model weights.

```
min.AICc <- order(out.tab$AICc)
out.tab.2 <- out.tab[min.AICc, ]
fits <- fits[min.AICc]
out.tab.2$delta.AICc <- out.tab.2$AICc - out.tab.2$AICc[1]
out.tab.2$rel.like <- exp(-1 * out.tab.2$delta.AICc / 2)
out.tab.2$AIC.weight <- out.tab.2$rel.like / sum(out.tab.2$rel.like)
```

Examination of the expanded results table (`out.tab.2`) shows there is strong support for correlated process errors; top 10 models shown:

```
              H                    Q delta.AICc AIC.weight
NC+Strait+PS+SC          equalvarcov       0.00      0.976
           site          equalvarcov       7.65      0.021
NC+Strait+PS+SC        unconstrained      11.47      0.003
NC+Strait+PS+SC   diagonal and equal      23.39      0.000
NC+Strait+PS+SC diagonal and unequal      27.53      0.000
            N-S        unconstrained      32.61      0.000
            N-S diagonal and unequal      36.06      0.000
            N-S          equalvarcov      36.97      0.000
          stock          equalvarcov      37.82      0.000
            N-S   diagonal and equal      38.16      0.000
```

The model weight for "equalvarcov", "unconstrained", versus "diagonal and equal" is

```
c(
  sum(out.tab.2$AIC.weight[out.tab.2$Q == "equalvarcov"]),
  sum(out.tab.2$AIC.weight[out.tab.2$Q == "unconstrained"]),
  sum(out.tab.2$AIC.weight[out.tab.2$Q == "diagonal and equal"])
)
```

```
[1] 0.997 0.003 0.000
```

9.5.1 Looking at the correlation structure in the Q matrix

The 3rd model in the output table is a model with all elements of the process error variance-covariance matrix estimated. Estimating a variance-covariance matrix with so many extra parameters is not supported relative to a constrained variance-covariance matrix with two parameters (compare the AICc for the 1st model and 3rd model) but looking at the full variance-covariance matrix shows some interesting and not surprising patterns.

The **Q** matrix is recovered from the model fit using this command

```
Q.unc <- coef(fits[[3]], type = "matrix")$Q
```

The diagonal of this matrix shows that each region appears to experience process variability of a similar magnitude:

```
diag(Q.unc)
```

```
        nc          is          ps          sc
0.009049512 0.007451479 0.004598690 0.005276587
```

We can compute the correlation matrix as follows. Row names are added to make the matrix more readable.

```
h <- diag(1 / sqrt(diag(Q.unc)))
Q.corr <- h %*% Q.unc %*% h
rownames(Q.corr) <- unique(Z4)
colnames(Q.corr) <- unique(Z4)
Q.corr
```

```
          nc        is        ps        sc
nc 1.0000000 0.5970202 0.6421536 0.9163056
is 0.5970202 1.0000000 0.9970869 0.2271385
ps 0.6421536 0.9970869 1.0000000 0.2832502
sc 0.9163056 0.2271385 0.2832502 1.0000000
```

The correlation matrix indicates that the inland strait ('is' in the table) subpopulation experiences process errors (good and bad years) that are almost perfectly correlated with the Puget Sound subpopulation though the two have different population growth rates (Figure 9.3). Similarly the north and south coastal subpopulations ('nc' and 'sc') experience highly correlated process errors, though again population growth rates are much higher to the north. There is much higher correlation between the process errors of the north coastal subpopulation and the nearby inland straits and Puget Sound subpopulations than between the two inland subpopulations and the much farther south coastal subpopulation. These patterns are not ecologically surprising but are not easy to discern looking at the raw count data.

9.6 Question 3, Is the Hood Canal independent?

In the initial analysis, the data from Hood Canal were removed. Hood Canal has experienced a series of hypoxic events which has led to large perturbations to the harbor seal prey. We will add the Hood Canal data back in and look at whether treating Hood Canal as separate is supported compared to treating it as part of the Puget Sound subpopulation in the top model.

```
sealData.hc <- rbind(sealData, harborSeal[, 8])
rownames(sealData.hc)[12] <- "Hood.Canal"
```

Here are the two **Z** matrices for a 'Hood Canal in the Puget Sound' and 'Hood Canal separate' model:

```
ZH1 <- factor(c("nc", "nc", "is", "is", "ps",
  "ps", "sc", "sc", "nc", "sc", "is", "ps"))
ZH2 <- factor(c("nc", "nc", "is", "is", "ps",
  "ps", "sc", "sc", "nc", "sc", "is", "hc"))
Z.models.hc <- list(ZH1, ZH2)
names(Z.models.hc) <- c("hood.in.ps", "hood.separate")
```

We will test three different **Q** matrices: a matrix with one variance and one covariance, an unconstrained variance-covariance matrix and a variance-covariance matrix where the Hood Canal subpopulation has independent process errors.

```
Q3 <- matrix(list("offdiag"), 5, 5)
diag(Q3) <- "q"
Q3[, 5] <- 0
Q3[5, ] <- 0
Q3[5, 5] <- "q.hc"
Q.models <- list("equalvarcov", "unconstrained", Q3)
names(Q.models) <- c("equalvarcov", "unconstrained", "hood.independent")
```

The independent Hood Canal **Q** allow correlation between the other four subpopulations but none between Hood Canal and those four:

```
Q.models$hood.independent
```

```
     [,1]      [,2]      [,3]      [,4]      [,5]
[1,] "q"       "offdiag" "offdiag" "offdiag" 0
[2,] "offdiag" "q"       "offdiag" "offdiag" 0
[3,] "offdiag" "offdiag" "q"       "offdiag" 0
[4,] "offdiag" "offdiag" "offdiag" "q"       0
[5,] 0         0         0         0         "q.hc"
```

As before, we loop through the model and create a results table:

```
out.tab.hc <- NULL
fits.hc <- list()
for (i in 1:length(Z.models.hc)) {
  for (j in 1:length(Q.models)) {
    if (i == 1 & j == 3) next # Q3 is only for Hood Separate model
    Q.model <- Q.models[[j]]
    fit.model <- c(list(Z = Z.models.hc[[i]], Q = Q.model), model.constant)
    fit <- MARSS(sealData.hc,
      model = fit.model,
      silent = TRUE, control = list(maxit = 1000)
    )
    out <- data.frame(
      H = names(Z.models.hc)[i], Q = names(Q.models)[j], U = U.model,
```

```
      logLik = fit$logLik, AICc = fit$AICc, num.param = fit$num.params,
      m = length(unique(Z.models.hc[[i]])),
      num.iter = fit$numIter, converged = !fit$convergence,
      stringsAsFactors = FALSE
    )
    out.tab.hc <- rbind(out.tab.hc, out)
    fits.hc <- c(fits.hc, list(fit))
  }
}
```

We sort the results by AICc and compute the ΔAICc.

```
min.AICc <- order(out.tab.hc$AICc)
out.tab.hc <- out.tab.hc[min.AICc, ]
out.tab.hc$delta.AICc <- out.tab.hc$AICc - out.tab.hc$AICc[1]
out.tab.hc$rel.like <- exp(-1 * out.tab.hc$delta.AICc / 2)
out.tab.hc$AIC.weight <- out.tab.hc$rel.like / sum(out.tab.hc$rel.like)
```

The results table (`out.tab.hc`) indicates strong support for treating Hood Canal as a separate subpopulation but not support for completely independent process errors.

H	Q	delta.AICc	AIC.weight
hood.separate	equalvarcov	0.00	0.988
hood.separate	hood.independent	8.74	0.012
hood.in.ps	equalvarcov	23.53	0.000
hood.separate	unconstrained	30.65	0.000
hood.in.ps	unconstrained	36.66	0.000

9.7 Discussion

In this chapter, we used model selection and AICc model weights to explore the temporal correlation structure in the harbor seal abundance data from the U.S. west coast. We used the term 'subpopulation', however it should be kept in mind that we are actually looking at the data support for different spatial patterns of temporal correlation in the process errors. Treating region A and B as a 'subpopulation' in this context means that we are asking if the counts from A and B can be treated as observations of the same underlying stochastic trajectory.

Metapopulation structure refers to a case where a larger population is composed of a collection of smaller temporally independent subpopulations. Metapopulation structure buffers the variability seen in the larger population and has important consequences for the viability of a population. We tested for temporal independence using diagonal versus non-diagonal **Q** matrices. Although the west coast harbor seal population appears to be divided into 'subpopulations' that experience different population growth rates, there is strong temporal correlation in the year-to-year variability experienced in these subpopulations. This suggests that this harbor seal population does not function as a true metapopulation with independent subpopulations but rather as a collection of subpopulations that are temporally correlated.

10

Dynamic factor analysis (DFA)

10.1 Overview of DFA

In this chapter, we use {MARSS} to do dynamic factor analysis (DFA), which allows us to look for a set of common underlying trends among a relatively large set of time series (Harvey, 1989, section 8.5). See also Zuur et al. (2003) which shows a number of examples of DFA applied to fisheries catch data and densities of zoobenthos. We will walk through some examples to show you the math behind DFA, and then in Section 10.4, we will show a short-cut for doing a DFA with MARSS using `form="dfa"`.

DFA is conceptually different than what we have been doing in the previous applications. Here we are trying to explain temporal variation in a set of n observed time series using linear combinations of a set of m hidden random walks, where $m << n$. A DFA model is a type of MARSS model with the following structure:

$$\begin{aligned} \mathbf{x}_t &= \mathbf{x}_{t-1} + \mathbf{w}_t \text{ where } \mathbf{w}_t \sim \text{MVN}(0, \mathbf{Q}) \\ \mathbf{y}_t &= \mathbf{Z}\mathbf{x}_t + \mathbf{a} + \mathbf{v}_t \text{ where } \mathbf{v}_t \sim \text{MVN}(0, \mathbf{R}) \\ \mathbf{x}_0 &\sim \text{MVN}(\pi, \Lambda) \end{aligned} \tag{10.1}$$

The general idea is that the observations ($\mathbf{y}$) are modeled as a linear combination of hidden trends ($\mathbf{x}$) and factor loadings ($\mathbf{Z}$) plus some offsets ($\mathbf{a}$). The DFA model in Equation 10.1 and the standard MARSS model in Equation 1.1 are equivalent—we have simply set the matrix $\mathbf{B}$ equal to an $m \times m$ identity matrix[1] and the vector $\mathbf{u} = \mathbf{0}$.

10.1.1 Writing out a DFA model as a MARSS model

Imagine a case where we had a data set with six observed time series ($n = 6$) and we want to fit a model with three hidden trends ($m = 3$). If we write out our DFA model

Type `RShowDoc("Chapter_DFA.R",package="MARSS")` at the R command line to open a file with all the code for the examples in this chapter.

[1] a diagonal matrix with 1's on the diagonal

in MARSS matrix form (ignoring the error structures and initial conditions for now), it would look like this:

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

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix}_t = \begin{bmatrix} z_{11} & z_{12} & z_{13} \\ z_{21} & z_{22} & z_{23} \\ z_{31} & z_{32} & z_{33} \\ z_{41} & z_{42} & z_{43} \\ z_{51} & z_{52} & z_{53} \\ z_{61} & z_{62} & z_{63} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}_t + \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \end{bmatrix} + \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \end{bmatrix}_t . \tag{10.2}$$

The process errors of the hidden trends would be

$$\begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}_t \sim \text{MVN}\left(\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} q_{11} & q_{12} & q_{13} \\ q_{12} & q_{22} & q_{23} \\ q_{13} & q_{23} & q_{33} \end{bmatrix} \right), \tag{10.3}$$

and the observation errors would be

$$\begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \end{bmatrix}_t \sim \text{MVN}\left(\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} & r_{15} & r_{16} \\ r_{12} & r_{22} & r_{23} & r_{24} & r_{25} & r_{26} \\ r_{13} & r_{23} & r_{33} & r_{34} & r_{35} & r_{36} \\ r_{14} & r_{24} & r_{34} & r_{44} & r_{45} & r_{46} \\ r_{15} & r_{25} & r_{35} & r_{45} & r_{55} & r_{56} \\ r_{16} & r_{26} & r_{36} & r_{46} & r_{56} & r_{66} \end{bmatrix} \right). \tag{10.4}$$

10.1.2 Constraints to ensure identifiability

If **Z**, **a**, and **Q** in Equation 10.1 are not constrained, then the DFA model above is unidentifiable (Harvey, 1989, sec 4.4). Harvey (1989, section 8.5.1) suggests the following parameter constraints to make the model identifiable:

- in the first $m-1$ rows of **Z**, the z-value in the j-th column and i-th row is set to zero if $j > i$;
- **a** is constrained so that the first m values are set to zero; and
- **Q** is set equal to the identity matrix ($\mathbf{I}_m$).

Zuur et al. (2003), however, found that with Harvey's second constraint, the EM algorithm is not particularly robust, and it takes a long time to converge. Zuur et al. found that the EM estimates are much better behaved if you instead constrain each of the time series in **x** to have a mean of zero across $t = 1$ to T. To do so, they

replaced the estimates of the hidden states, $\mathbf{x}_t^T$, coming out of the Kalman smoother[2] with $\mathbf{x}_t^T - \bar{\mathbf{x}}$ for $t = 1$ to T, where $\bar{\mathbf{x}}$ is the mean of $\mathbf{x}_t$ across t. With this approach, you estimate all of the **a** elements, which represent the average level of $\mathbf{y}_t$ relative to $\mathbf{Z}(\mathbf{x}_t - \bar{\mathbf{x}})$. We found that demeaning the $\mathbf{x}_t^T$ in this way can cause the EM algorithm to have errors (decline in log-likelihood). Instead, we demean our data, and fix all elements of **a** to zero.

Using these constraints, the DFA model in Equation 10.2 becomes

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

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix}_t = \begin{bmatrix} z_{11} & 0 & 0 \\ z_{21} & z_{22} & 0 \\ z_{31} & z_{32} & z_{33} \\ z_{41} & z_{42} & z_{43} \\ z_{51} & z_{52} & z_{53} \\ z_{61} & z_{62} & z_{63} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}_t + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \end{bmatrix}_t . \tag{10.5}$$

The process errors of the hidden trends in Equation 10.3 would then become

$$\begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}_t \sim \text{MVN} \left(\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \right), \tag{10.6}$$

but the observation errors in Equation 10.4 would stay the same, such that

$$\begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \end{bmatrix}_t \sim \text{MVN} \left(\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} & r_{15} & r_{16} \\ r_{12} & r_{22} & r_{23} & r_{24} & r_{25} & r_{26} \\ r_{13} & r_{23} & r_{33} & r_{34} & r_{35} & r_{36} \\ r_{14} & r_{24} & r_{34} & r_{44} & r_{45} & r_{46} \\ r_{15} & r_{25} & r_{35} & r_{45} & r_{55} & r_{56} \\ r_{16} & r_{26} & r_{36} & r_{46} & r_{56} & r_{66} \end{bmatrix} \right). \tag{10.7}$$

To complete our model, we still need the final form for the initial conditions of the state. Following Zuur et al. (2003), we set the initial state vector ($\mathbf{x}_0$) to have zero mean and a diagonal variance-covariance matrix with large variances, such that

$$\mathbf{x}_0 \sim \text{MVN} \left(\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 5 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 5 \end{bmatrix} \right). \tag{10.8}$$

[2] This is the estimate of the states conditioned on all the data, $t = 1$ to $t = T$.

10.2 The data

We will analyze some of the Lake Washington plankton data included in the {MARSS} package. This dataset includes 33 years of monthly counts for 13 plankton species along with data on water temperature, total phosphorous (TP), and pH. First, we load the data and then extract a subset of columns corresponding to the phytoplankton species only. For the purpose of speeding up model fitting times and to limit our analysis to years with no missing covariate data, we will only examine 10 years of data (1980-1989).

```
data(lakeWAplankton)
# we want lakeWAplanktonTrans, which has been log-transformed
# and the 0s replaced with NAs
plankdat <- lakeWAplanktonTrans
years <- plankdat[, "Year"] >= 1980 & plankdat[, "Year"] < 1990
phytos <- c(
  "Cryptomonas", "Diatoms", "Greens",
  "Unicells", "Other.algae"
)
dat.spp.1980 <- plankdat[years, phytos]
```

Next, we transpose the data and calculate the number of time series and their length.

```
# transpose data so time goes across columns
dat.spp.1980 <- t(dat.spp.1980)
N.ts <- nrow(dat.spp.1980)
TT <- ncol(dat.spp.1980)
```

It is normal in this type of analysis to standardize each time series by first subtracting its mean and then dividing by its standard deviation (i.e., create a z-score $\mathbf{y}_t^*$ with mean = 0 and standard deviation = 1), such that

$$\mathbf{y}_t^* = \Sigma^{-1}(\mathbf{y}_t - \bar{\mathbf{y}}),$$

Σ is a diagonal matrix with the standard deviations of each time series along the diagonal, and $\bar{\mathbf{y}}$ is a vector of the means. In R, this can be done as follows

```
Sigma <- sqrt(apply(dat.spp.1980, 1, var, na.rm = TRUE))
y.bar <- apply(dat.spp.1980, 1, mean, na.rm = TRUE)
dat.z <- (dat.spp.1980 - y.bar) * (1 / Sigma)
rownames(dat.z) <- rownames(dat.spp.1980)
```

MARSS also has a helper function to z-score data:

```
dat.z <- zscore(dat.spp.1980)
```

Figure 10.1 shows time series of Lake Washington phytoplankton data following z-score transformation.

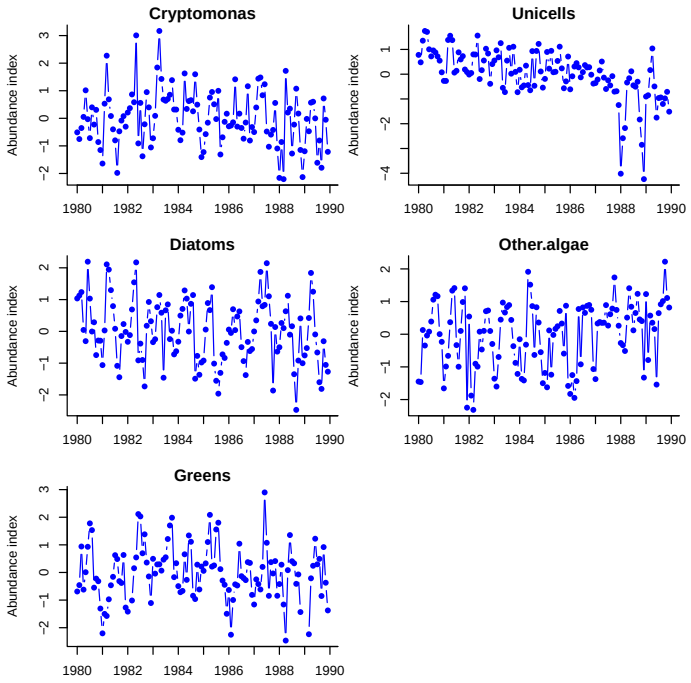

Fig. 10.1. Time series of Lake Washington phytoplankton data following z-score transformation.

10.3 Setting up the model for `MARSS()`

As we have seen in other cases, setting up the model structure for MARSS requires that the parameter matrices have a one-to-one correspondence to the model as you would write it on paper (i.e., Equations 10.5 through 10.8). If a parameter matrix has a combination of fixed and estimated values, then you specify that using `matrix(list(), nrow, ncol)`. This is a matrix of class list and allows you to combine numeric and character values in a single matrix. MARSS recognizes the numeric values as fixed values and the character values as estimated values.

This is how we set up **Z** for MARSS, assuming a model with 5 observed time series and 3 hidden trends:

```
Z.vals <- list(
  "z11", 0, 0,
  "z21", "z22", 0,
  "z31", "z32", "z33",
  "z41", "z42", "z43",
  "z51", "z52", "z53"
```

```
)
Z <- matrix(Z.vals, nrow = N.ts, ncol = 3, byrow = TRUE)
```

When specifying the list values, spacing and carriage returns were added to help show the correspondence with the **Z** matrix in Equation 10.3. If you print **Z** (at the R command line), you will see that it is a matrix with character values (the estimated elements) and numeric values (the fixed 0's).

```
print(Z)
```

```
     [,1]  [,2]  [,3]
[1,] "z11" 0     0
[2,] "z21" "z22" 0
[3,] "z31" "z32" "z33"
[4,] "z41" "z42" "z43"
[5,] "z51" "z52" "z53"
```

Notice that the `0`'s do not have quotes around them. If they did, it would mean the `"0"` is a character value and would be interpreted as the name of a parameter to be estimated rather than a fixed numeric value.

The **Q** and **B** matrices are both set equal to the identity matrix using `diag()`.

```
Q <- B <- diag(1, 3)
```

For our first analysis, we will assume that each time series of phytoplankton has a different observation variance, but that there is no covariance among time series. Thus, **R** should be a diagonal matrix that looks like:

$$\begin{bmatrix} r_{11} & 0 & 0 & 0 & 0 \\ 0 & r_{22} & 0 & 0 & 0 \\ 0 & 0 & r_{33} & 0 & 0 \\ 0 & 0 & 0 & r_{44} & 0 \\ 0 & 0 & 0 & 0 & r_{55} \end{bmatrix}$$

and each of the $r_{i,i}$ elements is a different parameter to be estimated. We can also specify this **R** structure using a list matrix as follows:

```
R.vals <- list(
  "r11", 0, 0, 0, 0,
  0, "r22", 0, 0, 0,
  0, 0, "r33", 0, 0,
  0, 0, 0, "r44", 0,
  0, 0, 0, 0, "r55"
)
R <- matrix(R.vals, nrow = N.ts, ncol = N.ts, byrow = TRUE)
```

You can print **R** at the R command line to see what it looks like:

```
print(R)
```

```
     [,1]  [,2]  [,3]  [,4]  [,5]
[1,] "r11" 0     0     0     0
[2,] 0     "r22" 0     0     0
[3,] 0     0     "r33" 0     0
[4,] 0     0     0     "r44" 0
[5,] 0     0     0     0     "r55"
```

This form of variance-covariance matrix is commonly used, and therefore {MARSS} has a built-in shorthand for this structure.

```
R <- "diagonal and unequal"
```

Type ?MARSS at the R command line to see a list of the shorthand options for each parameter vector/matrix.

The parameter vectors π (termed x0 in MARSS), **a** and **u** are each set to be a column vector of zeros. Any of the following can be used:

```
x0 <- U <- matrix(0, nrow = 3, ncol = 1)
A <- matrix(0, nrow = 6, ncol = 1)
x0 <- U <- A <- "zero"
```

The Λ matrix (termed V0 in MARSS) is a diagonal matrix with 5's along the diagonal:

```
V0 <- diag(5, 3)
```

Finally, we make a list of the model parameters to pass to the MARSS() function and set the control list:

```
dfa.model <- list(
  Z = Z, A = "zero", R = R, B = B, U = U,
  Q = Q, x0 = x0, V0 = V0
)
cntl.list <- list(maxit = 50)
```

For the examples in this chapter, we have set the maximum iterations to 50 to speed up model fitting. Note, however, that the parameter estimates will not have converged to their maximum likelihood values, which would likely take 100s, if not 1000+, iterations.

10.3.1 Fitting the model

We can now pass the DFA model list to MARSS() to estimate the **Z** matrix and underlying hidden states (**x**). The output is not shown because it is voluminous, but the model fits are plotted in Figure 10.2. The warnings regarding non-convergence are due to setting maxit to 50.

```
kemz.3 <- MARSS(dat.z, model = dfa.model, control = cntl.list)
```

```
Warning! Reached maxit before parameters converged. Maxit was 50.
 neither abstol nor log-log convergence tests were passed.

MARSS fit is
Estimation method: kem
Convergence test: conv.test.slope.tol = 0.5, abstol = 0.001
WARNING: maxit reached at  50  iter before convergence.
 Neither abstol nor log-log convergence test were passed.
 The likelihood and params are not at the MLE values.
 Try setting control$maxit higher.
Log-likelihood: -782.202
AIC: 1598.404   AICc: 1599.463

                            Estimate
Z.z11                         0.4163
Z.z21                         0.5364
Z.z31                         0.2780
Z.z41                         0.5179
Z.z51                         0.1611
Z.z22                         0.6757
Z.z32                        -0.2381
Z.z42                        -0.2381
Z.z52                        -0.2230
Z.z33                         0.2305
Z.z43                        -0.1225
Z.z53                         0.3887
R.(Cryptomonas,Cryptomonas)   0.6705
R.(Diatoms,Diatoms)           0.0882
R.(Greens,Greens)             0.7201
R.(Unicells,Unicells)         0.1865
R.(Other.algae,Other.algae)   0.5441
Initial states (x0) defined at t=0

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

Convergence warnings
10 warnings. First 10 shown.  Type cat(object$errors) to see the full list.
 Warning: the  Z.z51  parameter value has not converged.
 Warning: the  Z.z32  parameter value has not converged.
 Warning: the  Z.z52  parameter value has not converged.
 Warning: the  Z.z33  parameter value has not converged.
 Warning: the  Z.z43  parameter value has not converged.
 Warning: the  R.(Diatoms,Diatoms)  parameter value has not converged.
 Warning: the  R.(Greens,Greens)  parameter value has not converged.
```

```
Warning: the  R.(Other.algae,Other.algae)  parameter value has not converged.
Warning: the  logLik  parameter value has not converged.
Type MARSSinfo("convergence") for more info on this warning.
```

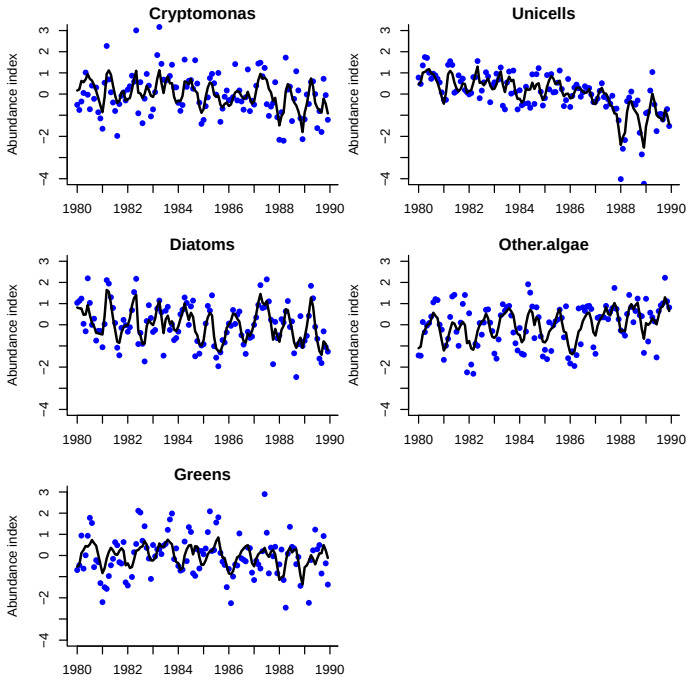

Fig. 10.2. Plots of Lake Washington phytoplankton data with model fits (dark lines) from a model with 3 trends and a diagonal and unequal variance-covariance matrix for the observation errors. This model was run to convergence so is different than that shown in the text which uses `maxit=50`.

10.4 Using model selection to determine the number of trends

Following Zuur et al. (2003), we use model selection criteria (specifically AICc) to determine the number of underlying trends that have the highest data support. Our first model had three underlying trends ($m = 3$). Let's compare this to a model with two underlying trends. The forms for parameter matrix $\mathbf{R}$ and vector $\mathbf{a}$ will stay the same but we need to change the other parameter vectors and matrices because m is different.

After showing you the matrix math behind a DFA model, we will now use the `form` argument for a MARSS call to specify that we want to fit a DFA model. Type `?MARSS.dfa` to learn about the `MARSS()` call with `form="dfa"`. This will set up the **Z** matrix and the other parameters for you. Specify how many trends you want by passing in `model=list(m=x)`. You can also pass in different forms for the **R** matrix in the usual way.

Here is how to fit two trends using `form="dfa"`:

```
model.list <- list(m = 2, R = "diagonal and unequal")
kemz.2 <- MARSS(dat.spp.1980,
  model = model.list,
  z.score = TRUE, form = "dfa", control = cntl.list
)

if (!saved.res) {
  model.list <- list(m = 2, R = "diagonal and unequal")
  kemz.2 <- MARSS(dat.spp.1980,
    model = model.list,
    z.score = TRUE, form = "dfa", control = big.maxit.cntl.list
  )
}
```

and compare its AICc value to that from the 3-trend model.

```
print(cbind(
  model = c("3 trends", "2 trends"),
  AICc = round(c(kemz.3$AICc, kemz.2$AICc))
),
quote = FALSE
)
```

```
    model    AICc
[1,] 3 trends 1589
[2,] 2 trends 1608
```

It looks like a model with 3 trends has much more support from the data because its AICc value is more than 10 units less than that for the 2-trend model.

10.4.1 Comparing many model structures

Now let's examine a larger suite of possible models. We will test from one to four underlying trends (m = 1 to 4) and four different structures for the **R** matrix:

1. same variances & no covariance (`"diagonal and equal"`);
2. different variances & no covariance (`"diagonal and unequal"`);
3. same variances & same covariance (`"equalvarcov"`); and
4. different variances & covariances (`"unconstrained"`).

The following code builds our model matrices; you could also write out each matrix as we did in the first example, but this allows us to build and run all of the models together. *NOTE*: the following piece of code will take a *very long* time to run!

```
# set new control params
cntl.list <- list(minit = 200, maxit = 5000, allow.degen = FALSE)
# set up forms of R matrices
levels.R <- c(
  "diagonal and equal",
  "diagonal and unequal",
  "equalvarcov",
  "unconstrained"
)
model.data <- data.frame(stringsAsFactors = FALSE)
# fit lots of models & store results
# NOTE: this will take a long time to run!
for (R in levels.R) {
  for (m in 1:(N.ts - 1)) {
    dfa.model <- list(A = "zero", R = R, m = m)
    kemz <- MARSS(dat.z,
      model = dfa.model, control = cntl.list,
      form = "dfa", z.score = TRUE
    )
    model.data <- rbind(
      model.data,
      data.frame(
        R = R,
        m = m,
        logLik = kemz$logLik,
        K = kemz$num.params,
        AICc = kemz$AICc,
        stringsAsFactors = FALSE
      )
    )
    assign(paste("kemz", m, R, sep = "."), kemz)
  } # end m loop
} # end R loop
```

Model selection results are shown in Table 10.1. The models with lowest AICc had 2 or 3 trends and an unconstrained **R** matrix. It also appears that, in general, models with an unconstrained **R** matrix fit the data much better than those models with less complex structures for the observation errors (i.e., models with unconstrained forms for **R** had nearly all of the AICc weight).

Table 10.1. Model selection results.

R	m	logLik	delta.AICc	Ak.wt	Ak.wt.cum
unconstrained	3	-762.5	0.0	0.39	0.39
unconstrained	2	-765.9	0.1	0.37	0.76
unconstrained	4	-761.5	2.3	0.12	0.89
unconstrained	1	-772.4	4.4	0.04	0.93
diagonal and unequal	4	-774.2	5.9	0.02	0.95
equalvarcov	2	-782.7	6.1	0.02	0.97
diagonal and unequal	3	-777.1	7.5	0.01	0.98
diagonal and equal	4	-779.3	7.7	0.01	0.99
diagonal and equal	3	-781.8	8.4	0.01	0.99
equalvarcov	4	-779.0	9.1	0.00	1.00
equalvarcov	3	-781.4	9.9	0.00	1.00
diagonal and unequal	2	-786.6	20.2	0.00	1.00
equalvarcov	1	-799.9	32.3	0.00	1.00
diagonal and equal	2	-798.4	35.4	0.00	1.00
diagonal and unequal	1	-798.4	35.4	0.00	1.00
diagonal and equal	1	-813.5	57.4	0.00	1.00

10.5 Using varimax rotation to determine the loadings and trends

As Harvey (1989, p. 450, sec. 8.5.1) discusses, there are multiple equivalent solutions to the dynamic factor loadings. We arbitrarily constrained $\mathbf{Z}$ in such a way to choose only one of these solutions, but fortunately the different solutions are equivalent, and they can be related to each other by a rotation matrix $\mathbf{H}$. Let $\mathbf{H}$ be any $m \times m$ non-singular matrix. The following are then equivalent solutions:

$$\begin{aligned} \mathbf{y}_t &= \mathbf{Z}\mathbf{x}_t + \mathbf{a} + \mathbf{v}_t \\ \mathbf{x}_t &= \mathbf{x}_{t-1} + \mathbf{w}_t \end{aligned} \tag{10.9}$$

and

$$\begin{aligned} \mathbf{y}_t &= \mathbf{Z}\mathbf{H}^{-1}\mathbf{x}_t^{\dagger} + \mathbf{a} + \mathbf{v}_t \\ \mathbf{x}_t^{\dagger} &= \mathbf{x}_{t-1}^{\dagger} + \mathbf{w}_t^{\dagger} \\ \mathbf{x}_t^{\dagger} &= \mathbf{H}\mathbf{x}_t; \mathbf{w}_t^{\dagger} = \mathbf{H}\mathbf{w}_t \end{aligned} \tag{10.10}$$

$\mathbf{x}^{\dagger}$ are the rotated trends.

There are many ways of doing factor rotations, but a common approach is the varimax rotation which seeks a rotation matrix $\mathbf{H}$ that creates the largest difference between loadings. For example, let's say there are three trends in our model. In our estimated $\mathbf{Z}$ matrix, let's say row 3 is $(0.2, 0.2, 0.2)$. That would mean that data series 3 is equally described by trends 1, 2, and 3. If instead row 3 was $(0.8, 0.1, 0.1)$, this would make interpretation easier because we could say that data time series 3 was mostly described by trend 1. The varimax rotation finds the $\mathbf{H}$ matrix that makes the $\mathbf{Z}$ rows more like $(0.8, 0.1, 0.1)$ and less like $(0.2, 0.2, 0.2)$.

The varimax rotation is easy to compute because R has the `varimax()` function[3] that returns $\mathbf{H}^{-1}$. We will illustrate the use of the varimax rotation with the 2-state model with **R** unconstrained. We will fit this with a large maxit.

```
big.maxit.cntl.list <- list(minit = 200, maxit = 5000, allow.degen = FALSE)
model.list <- list(m = 2, R = "unconstrained")
the.fit <- MARSS(dat.z, model = model.list, form = "dfa",
                 control = big.maxit.cntl.list)
```

Next, we retrieve the matrix used for varimax rotation.

```
# get the inverse of the rotation matrix
Z.est <- coef(the.fit, type = "matrix")$Z
H.inv <- 1
if (ncol(Z.est) > 1)
  H.inv <- varimax(coef(the.fit, type = "matrix")$Z)$rotmat
```

The rotation matrix that varimax returns $\mathbf{H}^{-1}$ rather than **H**. If **Z** has one column, there is only one **Z**; there is only a rotation matrix if **Z** has more than one column. We use $\mathbf{H}^{-1}$ to rotate the factor loadings and **H** to rotate the trends as in Equation 10.10.

```
# rotate factor loadings
Z.rot <- Z.est %*% H.inv
# rotate trends
trends.rot <- solve(H.inv) %*% the.fit$states
```

The following will get the confidence intervals on the rotated loadings:

```
# Add CIs to marssMLE object
the.fit <- MARSSparamCIs(the.fit)
# Use coef() to get the upper and lower CIs
Z.low <- coef(the.fit, type = "Z", what = "par.lowCI")
Z.up <- coef(the.fit, type = "Z", what = "par.upCI")
Z.rot.up <- Z.up %*% H.inv
Z.rot.low <- Z.low %*% H.inv
df <- data.frame(
  est = as.vector(Z.rot),
  conf.up = as.vector(Z.rot.up),
  conf.low = as.vector(Z.rot.low)
  )
```

Rotated factor loadings for the model are shown in Figure 10.3. Oddly, some taxa appear to have no loadings on some trends (e.g., diatoms on trend 1). The reason is that, merely for display purposes, we chose to plot only those loadings that are greater than 0.05, and it turns out that after varimax rotation, several loadings are close to 0.

[3] in the {stats} package

Recall that we set $\text{Var}(\mathbf{w}_t) = \mathbf{Q} = \mathbf{I}_m$ in order to make our DFA model identifiable. Does the variance in the process errors also change following varimax rotation? Interestingly, no. Because $\mathbf{H}$ is a non-singular, orthogonal matrix, $\text{Var}(\mathbf{H}\mathbf{w}_t) = \mathbf{H}\text{Var}(\mathbf{w}_t)\mathbf{H}^\top = \mathbf{H}\mathbf{I}_m\mathbf{H}^\top = \mathbf{I}_m$.

10.6 Examining model fits

Now that we have done the appropriate factor and trends rotations, we should examine some plots of model fits. To do so, we will create a function `getDFAfits()` to extract the model fits and estimated $(1-\alpha)\%$ confidence intervals. Note, the function `residuals(..., type="tT")` will also return this information.

```
fit.b <- getDFAfits(the.fit)
```

First, it looks like this model captures some of the high frequency variation (i.e., seasonality) in the time series (see Figure 10.5). Second, some of the time series had much better overall fits than others (e.g., compare Cryptomonas and Unicells). Given the obvious seasonal patterns in the phytoplankton data, it would be worthwhile to first detrend the data and then repeat the model fitting exercise to see (1) how many trends would be favored, and (2) the shape of those trends.

10.7 Adding covariates

It is standard to add covariates to the analysis so that one removes known important drivers. The DFA with covariates is written:

$$\begin{aligned} \mathbf{x}_t &= \mathbf{x}_{t-1} + \mathbf{w}_t \text{ where } \mathbf{w}_t \sim \text{MVN}(0, \mathbf{Q}) \\ \mathbf{y}_t &= \mathbf{Z}\mathbf{x}_t + \mathbf{a} + \mathbf{D}\mathbf{d}_t + \mathbf{v}_t \text{ where } \mathbf{v}_t \sim \text{MVN}(0, \mathbf{R}) \\ \mathbf{x}_0 &\sim \text{MVN}(\pi, \Lambda) \end{aligned} \tag{10.11}$$

where the $q \times 1$ vector $\mathbf{d}_t$ contains the covariate(s) at time t, and the $n \times q$ matrix $\mathbf{D}$ contains the effect(s) of the covariate(s) on the observations. Using `form="dfa"` and `covariates=`*<covariate name(s)>*, we can easily add covariates to our DFA, but this means that the covariates are input, not data, and there can be no missing values. See Chapter 13 for how to include covariates with missing values.

The Lake Washington dataset has two environmental covariates that we might expect to have effects on phytoplankton growth, and hence, abundance: temperature (Temp) and total phosphorous (TP).

```
temp <- t(plankdat[years, "Temp", drop = FALSE])
TP <- t(plankdat[years, "TP", drop = FALSE])
```

Type `RShowDoc("Chapter_DFA.R",package="MARSS")` at the R command line to open a file with all the code for this chapter and search for the function name.

```
# plot the factor loadings
spp <- rownames(dat.z)
minZ <- 0.05
m <- dim(trends.rot)[1]
ylims <- c(-1.1 * max(abs(Z.rot)), 1.1 * max(abs(Z.rot)))
par(mfrow = c(ceiling(m / 2), 2), mar = c(3, 4, 1.5, 0.5), oma = c(0.4, 1, 1, 1))
for (i in 1:m) {
  plot(c(1:N.ts)[abs(Z.rot[, i]) > minZ], as.vector(Z.rot[abs(Z.rot[, i]) > minZ, i]),
    type = "h", lwd = 2, xlab = "", ylab = "", xaxt = "n", ylim = ylims, xlim = c(0, N.t
  )
  for (j in 1:N.ts) {
    if (Z.rot[j, i] > minZ) {
      text(j, -0.05, spp[j], srt = 90, adj = 1, cex = 0.9)
    }
    if (Z.rot[j, i] < -minZ) {
      text(j, 0.05, spp[j], srt = 90, adj = 0, cex = 0.9)
    }
    abline(h = 0, lwd = 1, col = "gray")
  } # end j loop
  mtext(paste("Factor loadings on trend", i, sep = " "), side = 3, line = .5)
} # end i loop
```

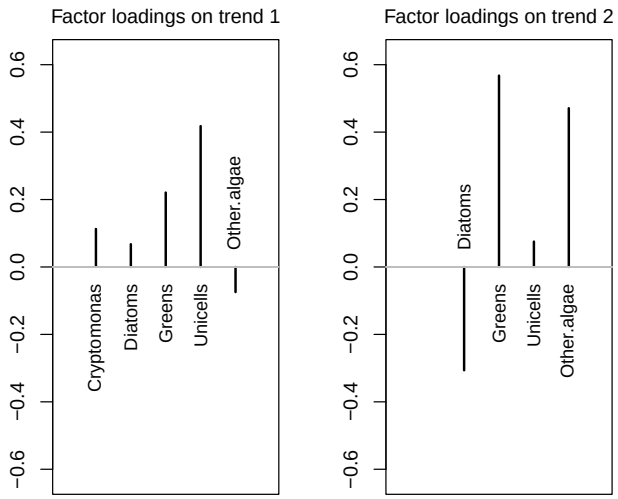

Fig. 10.3. Plot of the factor loadings (following varimax rotation) from the 2-state model fit to the phytoplankton data.

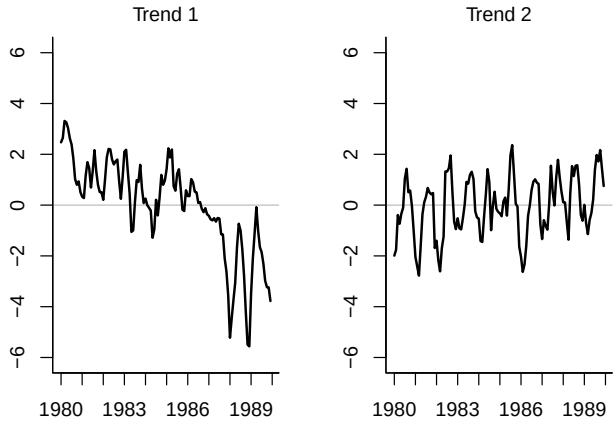

Fig. 10.4. Plot of the unobserved trends (following varimax rotation) from the 2-state model fit to the phytoplankton data.

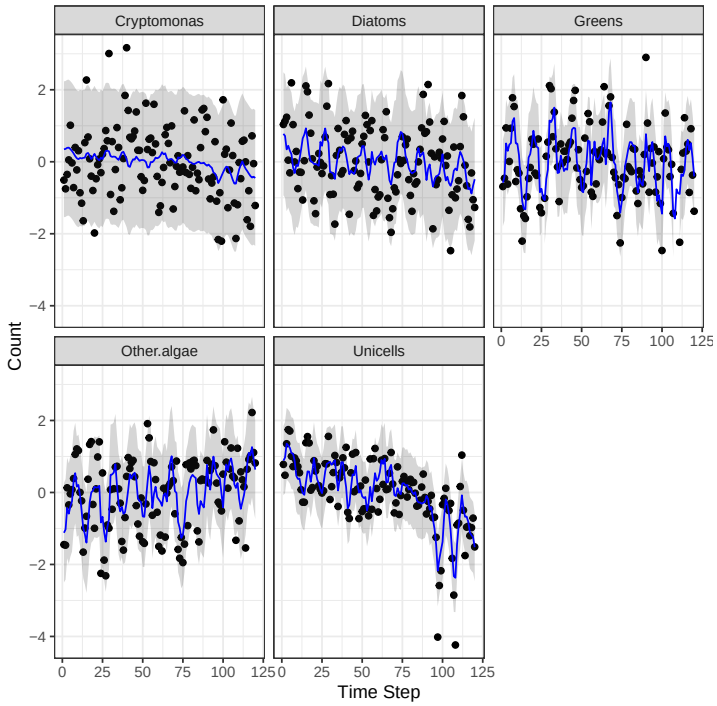

Fig. 10.5. Plot of the 2-state model fits to the phytoplankton data.

We will now fit three different models that each add covariate effects (i.e., Temp, TP, Temp & TP) to our 2-state model with **R** "unconstrained".

```
model.list <- list(m = 2, R = "unconstrained")
kemz.temp <- MARSS(dat.spp.1980,
  model = model.list, z.score = TRUE,
  form = "dfa", control = cntl.list, covariates = temp
)
kemz.TP <- MARSS(dat.spp.1980,
  model = model.list, z.score = TRUE,
  form = "dfa", control = cntl.list, covariates = TP
)
kemz.both <- MARSS(dat.spp.1980,
  model = model.list, z.score = TRUE,
  form = "dfa", control = cntl.list, covariates = rbind(temp, TP)
)
```

Next we can compare whether the addition of the covariates improves the model fit (effectively less residual error while accounting for the additional parameters). *NOTE*: The following results were obtained by letting the EM algorithm run for a *very long* time, so your results may differ.

```
print(cbind(
  model = c("no covars", "Temp", "TP", "Temp & TP"),
  AICc = round(c(
    the.fit$AICc, kemz.temp$AICc, kemz.TP$AICc,
    kemz.both$AICc
  ))
), quote = FALSE)
```

```
     model     AICc
[1,] no covars 1582
[2,] Temp      1518
[3,] TP        1568
[4,] Temp & TP 1522
```

This suggests that adding temperature or phosphorus to the model, either alone or in combination with one another, improves overall model fit. If we were interested in assessing the best model structure that includes covariates, however, we should examine all combinations of trends and structures for **R**. The model fits for the temperature-only model are shown in Fig 10.6 and they appear much better than the model without any covariates.

10.8 Discussion

We analyzed the phytoplankton data alone. You can try analyzing the zooplankton data (type `head(plankdat)` to see the zooplankton names). You can also try ana-

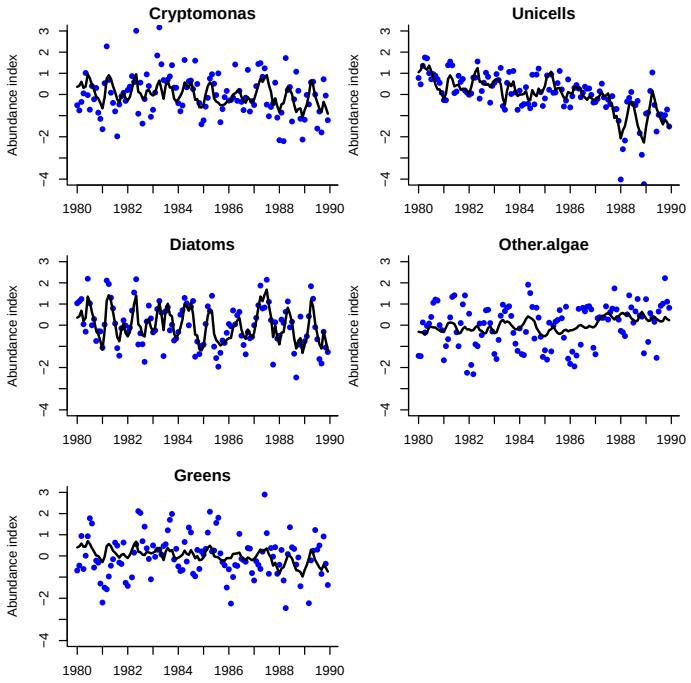

Fig. 10.6. Plot of the fits from the temperature-only model to the phytoplankton data.

lyzing the phytoplankton and zooplankton together. You can also try different assumptions concerning the structure of **R**; we just tried unconstrained, diagonal and unequal, and diagonal and equal. Lastly, notice that there is a seasonal cycle in the data. We did not explicitly include a seasonal cycle and it would be wise to include that as a covariate. A random walk can fit a seasonal cycle, but a random walk is not fundamentally cyclic and thus is not a good way to model a cycle.

DFA models often take an unusually long time to converge. In a real DFA, you will want to make sure to try different initial starting values (see Chapter 6), and force the algorithm to run a long time by using `minit=x` and `maxit=(x+c)`, where `x` and `c` are something like 200 and 5000, respectively. You might also try using `method="BFGS"` in the `MARSS()` call.

11

Analyzing noisy animal tracking data

11.1 A simple random walk model of animal movement

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

$$x_{1,t} = x_{1,t-1} + u_1 + w_{1,t}, \;\; w_{1,t} \sim \mathrm{N}(0, \sigma_1^2) \tag{11.1}$$

$$x_{2,t} = x_{2,t-1} + u_2 + w_{2,t}, \;\; w_{2,t} \sim \mathrm{N}(0, \sigma_2^2) \tag{11.2}$$

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

$$y_{1,t} = x_{1,t} + v_{1,t}, \;\; v_{1,t} \sim \mathrm{N}(0, \eta_1^2) \tag{11.3}$$

$$y_{2,t} = x_{2,t} + v_{2,t}, \;\; v_{2,t} \sim \mathrm{N}(0, \eta_2^2), \tag{11.4}$$

This model is comprised of two separate univariate state-space models. Note that y_1 depends only on x_1 and y_2 depends only on x_2. There are no actual interactions between these two univariate models. However, we can write the model down in the form of a multivariate model using diagonal variance-covariance matrices and a diagonal design ($\mathbf{Z}$) matrix. Because the variance-covariance matrices and $\mathbf{Z}$ are diagonal, the x_1:y_1 and x_2:y_2 processes will be independent as intended. Here are Equations 11.2 and 11.4 written as a MARSS model (in matrix form):

Type `RShowDoc("Chapter_AnimalTracking.R",package="MARSS")` at the R command line to open a file with all the code for the examples in this chapter.

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \end{bmatrix} = \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \end{bmatrix} + \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} + \begin{bmatrix} w_{1,t} \\ w_{2,t} \end{bmatrix}, \ \mathbf{w}_t \sim \text{MVN}\left(0, \begin{bmatrix} \sigma_1^2 & 0 \\ 0 & \sigma_2^2 \end{bmatrix}\right) \tag{11.5}$$

$$\begin{bmatrix} y_{1,t} \\ y_{2,t} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_{1,t} \\ x_{2,t} \end{bmatrix} + \begin{bmatrix} v_{1,t} \\ v_{2,t} \end{bmatrix}, \ \mathbf{v}_t \sim \text{MVN}\left(0, \begin{bmatrix} \eta_1^2 & 0 \\ 0 & \eta_2^2 \end{bmatrix}\right) \tag{11.6}$$

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

$$\mathbf{x}_t = \mathbf{x}_{t-1} + \mathbf{u} + \mathbf{w}_t, \ \ \mathbf{w}_t \sim \text{MVN}(0, \mathbf{Q}) \tag{11.7}$$

$$\mathbf{y}_t = \mathbf{x}_t + \mathbf{v}_t, \ \ \mathbf{v}_t \sim \text{MVN}(0, \mathbf{R}). \tag{11.8}$$

11.2 Loggerhead sea turtle tracking data

Loggerhead sea turtles (*Caretta caretta*) are listed as threatened under the United States Endangered Species Act of 1973. Over the last ten years, a number of state and local agencies have been deploying ARGOS tags on loggerhead turtles on the east coast of the United States. We have data on eight individuals over that period. In this chapter, we use some turtle data from the WhaleNet Archive of STOP Data, however we have corrupted this data severely by adding random errors in order to create a "bad tag" problem (Figure 11.1), and it would appear that our sea turtles are becoming land turtles (at least part of the time). We will use the MARSS model to estimate true positions and speeds from the corrupted data.

Our noisy data are in `loggerheadNoisy`. They consist of daily readings of location (longitude and latitude). If data are missing for a day, then the entries for latitude and longitude for that day should be NA. However, to make the code in this chapter run quickly, we have interpolated all missing values in the original, uncorrupted, dataset (`loggerhead`). The first six lines of the corrupted data are

```
loggerheadNoisy[1:6, ]
```

```
   turtle month day year       lon      lat
1 BigMama     5  28 2001 -81.45989 31.70337
2 BigMama     5  29 2001 -80.88292 32.18865
3 BigMama     5  30 2001 -81.27393 31.67568
4 BigMama     5  31 2001 -81.59317 31.83092
5 BigMama     6   1 2001 -81.35969 32.12685
6 BigMama     6   2 2001 -81.15644 31.89568
```

The file has data for eight turtles:

```
turtles <- levels(loggerheadNoisy$turtle)
turtles
```

```
[1] "BigMama"  "Bruiser"  "Humpty"   "Isabelle" "Johanna"
[6] "MaryLee"  "TBA"      "Yoto"
```

We will analyze the position data for "Big Mama". We put the data for "Big Mama" into matrix `dat`. `dat` is transposed because we need time across the columns.

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

Figure 11.1 shows the corrupted location data for Big Mama. The figure code uses the `maps` R package. You will need to install this R package in order to run the example code.

```
# load the map package; you have to install it first
library(maps)
# Read in our noisy data (no missing values)
pdat <- loggerheadNoisy # for plotting
turtlename <- "BigMama"
theTurtle <- which(loggerheadNoisy$turtle == turtlename)
par(mai = c(0, 0, 0, 0), mfrow = c(1, 1))
map("state",
  region = c(
    "florida", "georgia", "south carolina",
    "north carolina", "virginia", "delaware", "new jersey", "maryland"
  ),
  xlim = c(-85, -70)
)
points(pdat$lon[theTurtle], pdat$lat[theTurtle],
  col = "blue", pch = 21, cex = 0.7
)
```

11.3 Estimate locations from the bad tag data

We will begin by specifying the structure of the MARSS model and then use `MARSS()` to fit that model to the data. There are two state processes (one for latitude and the other for longitude), and there is one observation time series for each state process. As we saw in Equation 11.6, $\mathbf{Z}$ is the an identity matrix (a diagonal matrix with 1s on the diagonal). We could specify this structure as `Z.model="identity"` or `Z.model=factor(c(1,2))`. Although technically, this is unnecessary as this is the default form for $\mathbf{Z}$.

We will assume that the errors are independent and that there are different drift rates (u), process variances (σ^2) and observation variances for latitude and longitude (η^2).

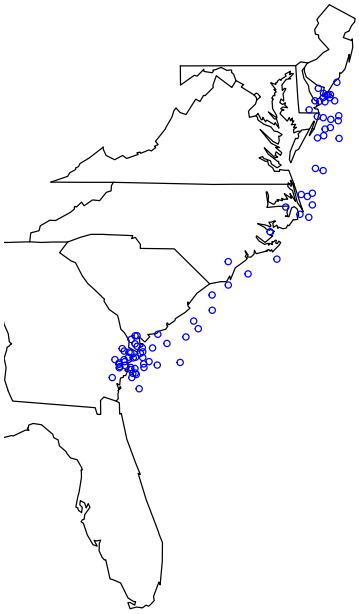

Fig. 11.1. Plot of the tag data from the turtle Big Mama. Errors in the location data make it seem that Big Mama has been moving overland.

```
Z.model <- "identity"
U.model <- "unequal"
Q.model <- "diagonal and unequal"
R.model <- "diagonal and unequal"
```

Fit the model to the data:

```
kem <- MARSS(dat, model = list(
  Z = Z.model,
  Q = Q.model, R = R.model, U = U.model
))
```

We can create a plot comparing the estimated and actual locations (Figure 11.2). The real locations (from which `loggerheadNoisy` was produced by adding noise) are in `loggerhead` and plotted with crosses. There are only a few data points for the real data because in the real tag data, there are many missing days.

```
# Code to plot estimated turtle track against observations
# The estimates
pred.lon <- kem$states[1, ]
```

```
pred.lat <- kem$states[2, ]
par(mai = c(0, 0, 0, 0), mfrow = c(1, 1))
library(maps)
pdat <- loggerheadNoisy
turtlename <- "BigMama"
map("state",
  region = c(
    "florida", "georgia", "south carolina",
    "north carolina", "virginia", "delaware", "new jersey", "maryland"
  ),
  xlim = c(-85, -70)
)
points(pdat$lon[theTurtle], pdat$lat[theTurtle],
  col = "blue", pch = 21, cex = 0.7
)
lines(pred.lon, pred.lat, col = "red", lwd = 2)
goodturtles <- loggerhead
gooddat <- goodturtles[which(goodturtles$turtle == turtlename), 5:6]
points(gooddat[, 1], gooddat[, 2], col = "black", lwd = 2, pch = 3, cex = 1.1)
legend("bottomright", c(
  "bad locations", "estimated true location",
  "good location data"
),
pch = c(1, -1, 3), lty = c(-1, 1, -1),
col = c("blue", "red", "black"), bty = FALSE
)
```

11.4 Estimate speeds for each turtle

For each of the eight turtles, estimate the average miles traveled per day. To calculate the distance traveled by a turtle each day, you use the estimate (from `MARSS()`) of the latitude/longitude location of turtle at day t and at day $t-1$. To calculate distance traveled in miles from latitude/longitude start and finish locations, we will use the function `GCDF`:

```
GCDF <- function(lon1, lon2, lat1, lat2, degrees = TRUE, units = "miles") {
  temp <- ifelse(degrees == FALSE,
    acos(sin(lat1) * sin(lat2) + cos(lat1) * cos(lat2) * cos(lon2 - lon1)),
    acos(sin(lat1 / 57.2958) * sin(lat2 / 57.2958) +
      cos(lat1 / 57.2958) * cos(lat2 / 57.2958) *
        cos(lon2 / 57.2958 - lon1 / 57.2958))
  )
  r <- 3963.0 # (statute miles) , default
  if ("units" == "nm") r <- 3437.74677 # (nautical miles)
```

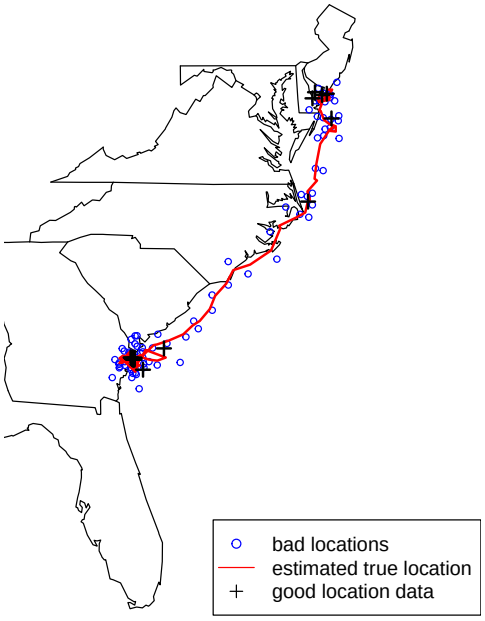

Fig. 11.2. Plot of the estimated track of the turtle Big Mama versus the good location data (before we corrupted it with noise).

```
  if ("units" == "km") r <- 6378.7 # (kilometers)
  return(r * temp)
}
```

We can now compute the distance traveled each day by passing in lat/lon estimates from day $i-1$ and day i:

```
distance[i - 1] <- GCDF(
  pred.lon[i - 1], pred.lon[i],
  pred.lat[i - 1], pred.lat[i]
)
```

`pred.lon` and `pred.lat` are the predicted longitudes and latitudes from `MARSS()`: rows one and two in `kem$states`. To calculate the distances for all days, we put this through a `for` loop:

```
distance <- array(NA, dim = c(dim(dat)[2] - 1, 1))
for (i in 2:dim(dat)[2]) {
  distance[i - 1] <- GCDF(
    pred.lon[i - 1], pred.lon[i],
```

```
    pred.lat[i - 1], pred.lat[i]
  )
}
```

The command `mean(distance)` gives us the average distance per day. We can also make a histogram of the distances traveled per day (Figure 11.3).

```
par(mfrow = c(1, 1))
hist(distance) # make a histogram of distance traveled per day
```

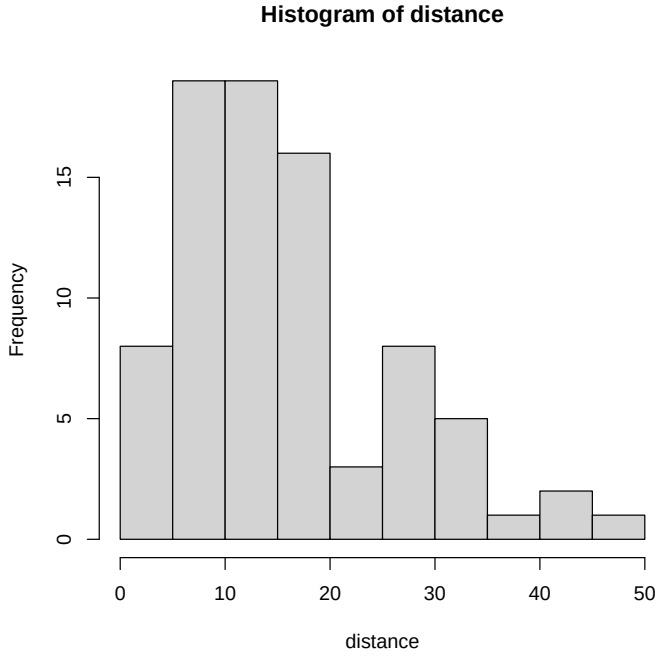

Fig. 11.3. Histogram of the miles traveled per day for Big Mama with estimates that account for measurement error in the data.

We can compare the histogram of daily distances to what we would get if we had not accounted for measurement error (Figure 11.4). We can also compare the mean miles per day:

```
# accounting for observation error
mean(distance)
```

```
[1] 15.53858
```

```
# assuming the data have no observation error
mean(distance.noerr)
```

```
# Compare to the distance traveled per day if you used the raw data
distance.noerr <- array(NA, dim = c(dim(dat)[2] - 1, 1))
for (i in 2:dim(dat)[2]) {
  distance.noerr[i - 1] <- GCDF(dat[1, i - 1], dat[1, i], dat[2, i - 1], dat[2, i])
}
hist(distance.noerr) # make a histogram of distance traveled per day
```

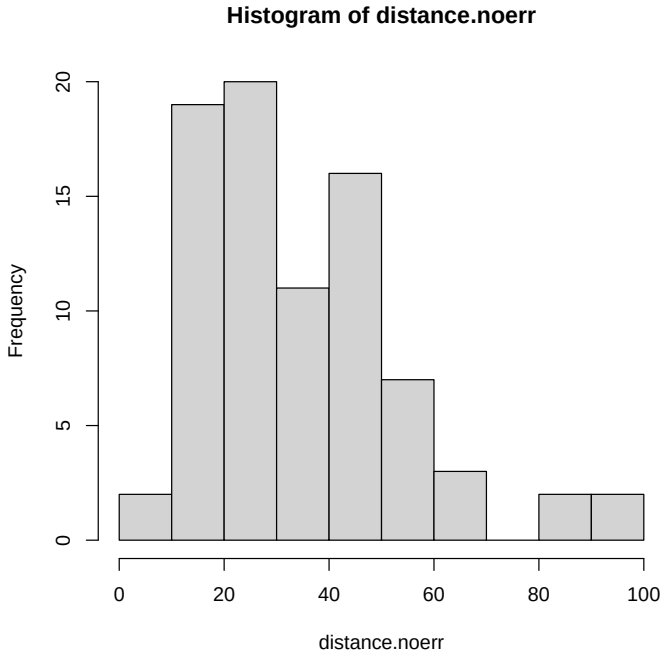

Fig. 11.4. Histogram of the miles traveled per day for Big Mama with estimates that account for measurement error in the data.

```
[1] 34.80579
```

You can repeat the analysis done for "Big Mama" for each of the other turtles and compare the turtle speeds and errors. You will need to replace "Big Mama" in the code with the name of the other turtle:

```
levels(loggerheadNoisy$turtle)
```

```
[1] "BigMama"  "Bruiser"  "Humpty"   "Isabelle" "Johanna"
[6] "MaryLee"  "TBA"      "Yoto"
```

11.5 Using specialized packages to analyze tag data

If you have real tag data to analyze, you should use a state-space modeling package that is customized for fitting MARSS models to tracking data. The {MARSS} package does not have all the bells and whistles that you would want for analyzing tracking data, particularly tracking data in the marine environment. Examples are the {Ukfsst} and {kftrack} R packages:

UKFSST https://github.com/positioning/kalmanfilter/wiki/ArticleUkfsst
KFTRACK https://github.com/positioning/kalmanfilter/wiki/Articlekftrack

`kftrack` is a full-featured toolbox for analyzing tag data with extended Kalman filtering. It incorporates a number of extensions that are important for analyzing track data: barriers to movement such as coastlines and non-Gaussian movement distributions. With `kftrack`, you can use the real tag data which has big gaps, i.e., days with no location. `MARSS()` will struggle with these data because it will estimate states for all the unseen days; `kftrack` only fits to the seen days.

12

Detection of outliers and structural breaks

12.1 Background

This chapter is based on a short example shown on pages 147-148 in Koopman et al. (1999) using a 100-year record of river flow on the Nile River. The methods are based on Harvey et al. (1998) which is in turn based on techniques in Harvey and Koopman (1992) and Koopman (1993). The Nile dataset is included in R . Figure 12.1 shows the data.

12.2 Different models for the Nile flow levels

We begin by fitting different flow models to the data and compare these models with AIC. After that, we will use the model residuals to look for outliers and structural breaks.

12.2.1 Flat level model

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

$$y_t = a + v_t \text{ where } v_t \sim \text{N}(0, r) \tag{12.1}$$

where y_t is the river flow volume at year t and x is some constant average flow level (notice it has no t subscript).

To fit this model with MARSS, we will explicitly show all the MARSS parameters.

Type `RShowDoc("Chapter_StructuralBreaks.R",package="MARSS")` at the R command line to open a file with all the code for the examples in this chapter.

```
# load the datasets package
library(datasets)
data(Nile) # load the data
plot(Nile, ylab = "Flow volume", xlab = "")
```

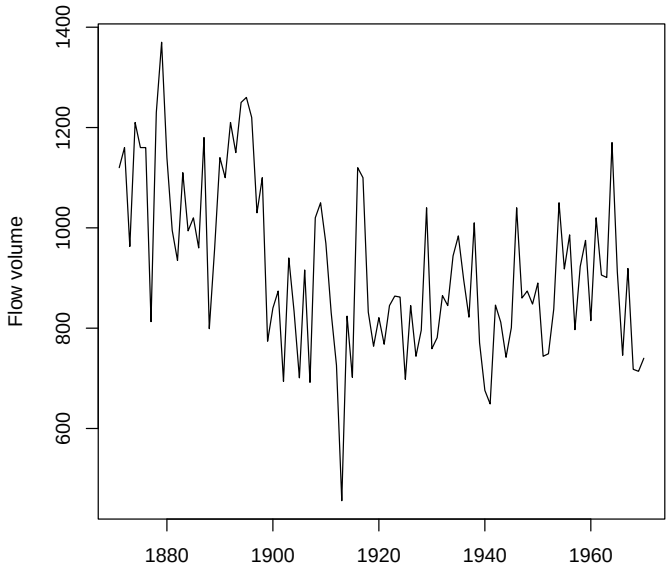

Fig. 12.1. The Nile River flow volume 1871 to 1970 (included dataset in R).

$$\begin{aligned} x_t &= 1 \times x_{t-1} + 0 + w_t \text{ where } w_t \sim \mathrm{N}(0,0) \\ y_t &= 0 \times x_t + a + v_t \text{ where } v_t \sim \mathrm{N}(0,r) \\ x_0 &= 0 \end{aligned} \tag{12.2}$$

MARSS includes the state process x_t but we are setting $\mathbf{Z}$ to zero so that does not appear in our observation model. We need to fix all the state parameters to zero so that the algorithm doesn't "chase its tail" trying to fit x_t to the data.

An equivalent way to write this model is to use x_t as the average flow level and make it be a constant level by setting $q = 0$. The average flow appears as the x_0 parameter. Written as a MARSS model, the model is:

$$\begin{aligned} x_t &= 1 \times x_{t-1} + 0 + w_t \text{ where } w_t \sim \mathrm{N}(0,0) \\ y_t &= 1 \times x_t + 0 + v_t \text{ where } v_t \sim \mathrm{N}(0,r) \\ x_0 &= a \end{aligned} \tag{12.3}$$

We will use this latter format since we will be building on this form. The model is specified as follows:

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

We then fit the model:

```
# The data is in a ts format, and we need a matrix
dat <- t(as.matrix(Nile))
rownames(dat) <- "Nile"
kem.0 <- MARSS(dat, model = mod.nile.0, silent = TRUE)
summary(kem.0)
```

```
m: 1 state process(es) named X.Nile
n: 1 observation time series named Nile

  term estimate
1  R.r 28351.57
2 x0.a   919.35
```

12.2.2 Linear trend in flow model

Figure 12.2 shows the fit for the flat average river flow model. Looking at the data, we might expect that a declining average river flow would be better. In MARSS form, that model would be:

$$\begin{aligned} x_t &= 1 \times x_{t-1} + u + w_t \text{ where } w_t \sim \mathrm{N}(0,0) \\ y_t &= 1 \times x_t + 0 + v_t \text{ where } v_t \sim \mathrm{N}(0,r) \\ x_0 &= a \end{aligned} \tag{12.4}$$

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

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

We then fit the model:

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

```
m: 1 state process(es) named X.Nile
n: 1 observation time series named Nile

  term     estimate
1  R.r 22213.595453
2  U.u    -2.692106
3 x0.a  1054.935067
```

Figure 12.2 shows the fits for the two models with deterministic models (flat and declining) for mean river flow along with their AICc values (smaller AICc is better). The AICc for the model with a declining river flow is lower by over 20 (which is a lot).

12.2.3 Stochastic level model

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

$$\begin{aligned} x_t &= x_{t-1} + w_t \text{ where } w_t \sim \mathrm{N}(0,q) \\ y_t &= x_t + v_t \text{ where } v_t \sim \mathrm{N}(0,r) \\ x_0 &= \pi \end{aligned} \tag{12.5}$$

As before, y_t is the river flow volume at year t. With all the MARSS parameters shown, the model is:

$$\begin{aligned} x_t &= 1 \times x_{t-1} + 0 + w_t \text{ where } w_t \sim \mathrm{N}(0,q) \\ y_t &= 1 \times x_t + 0 + v_t \text{ where } v_t \sim \mathrm{N}(0,r) \\ x_0 &= \pi \end{aligned} \tag{12.6}$$

Thus, $\mathbf{Z} = 1$, $\mathbf{a} = 0$, $\mathbf{R} = r$, $\mathbf{B} = 1$, $\mathbf{u} = 0$, $\mathbf{Q} = q$, and $\mathbf{x}_0 = \pi$. The model is then specified as:

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

We could also use the text shortcuts to specify the model. Because $\mathbf{R}$ and $\mathbf{Q}$ are 1×1 matrices, "unconstrained", "diagonal and unequal", "diagonal and equal" and "equalvarcov" will all lead to a 1×1 matrix with one estimated element.

To fit the model, we use the BFGS algorithm to polish off the estimates, since it will get the maximum faster than the default EM algorithm as long as we start it close to the maximum.

```
kem.2em <- MARSS(dat, model = mod.nile.2, silent = TRUE)
kem.2 <- MARSS(dat,
  model = mod.nile.2,
  inits = kem.2em$par, method = "BFGS", silent = TRUE
)
summary(kem.2)
```

```
m: 1 state process(es) named X.Nile
n: 1 observation time series named Nile

   term  estimate
1   R.r 15336.530
2   Q.q  1218.137
3 x0.pi  1111.591
```

This is the same model fit in Koopman et al. (1999, p. 148) except that we estimate x_1 as parameter rather than specifying x_1 via a diffuse prior. As a result, the log-likelihood value and **R** and **Q** are a little different than in Koopman et al. (1999).

12.3 Observation and state residuals

Figure 12.2 shows the fits to the data. From these model fits, auxiliary residuals can be computed which contain information about whether the data and models fits at time t differ more than you would expect given the model and the model fits at time $t-1$. In this section, we follow the example shown on page 147-148 in Koopman et al. (1999) and use these residuals to look for outliers and sudden flow level changes. Using auxiliary residuals this way follows mainly from Harvey and Koopman (1992), but see also Koopman (1993, section 3), de Jong and Penzer (1998) and Penzer (2001) for discussions of using auxiliary residuals for detection of outliers and structural breaks.

The `MARSS()` function will output the expected values of x_t conditioned on the maximum-likelihood values of q, r, and x_1 and on the data (y from $t=1$ to T). In time-series literature, these are called the smoothed state estimates and they are output by the Kalman filter-smoother. We will denote these smoothed estimates x_t^T (and are `xtT` in the MARSS output). The time value in the superscript indicates the last data time point on which the estimate was conditioned (in this case, the state estimate is conditioned on data from $t=1$ to $t=T$). From these, we can compute the model predicted value of y_t, denoted or $\hat{y}_t^T$. This is the predicted value of y_t conditioned on x_t^T.

$$\begin{aligned} x_t^T &= \mathrm{E}[X_t|\hat{\theta}, y_1^T] \\ \hat{y}_t^T &= \mathrm{E}[Y_t|\hat{\theta}, x_t^T] \\ &= x_t^T + \mathrm{E}[w_t|\hat{\theta}, y_1^T] = x_t^T \end{aligned} \tag{12.7}$$

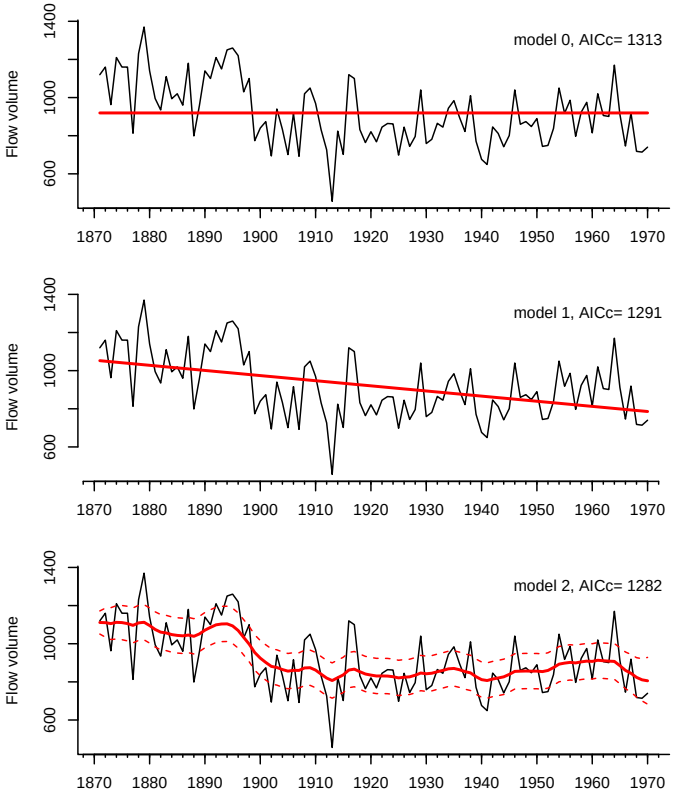

Fig. 12.2. The Nile River flow volume with the model estimated flow rates (solid lines). The bottom model is a stochastic level model, and the 2 standard deviations for the level are also shown. The other two models are deterministic level models so the state is not stochastic and does not have a standard deviation.

where $\hat{\theta}$ are the maximum-likelihood estimates of the parameters. The $\hat{y}_t^T$ equation comes directly from equation 12.5. This expectation is not conditioned on the data y_1^T, directly. It is conditioned on x_t^T, which is conditioned on y_1^T.

12.3.1 Using observation residuals to detect outliers

The standardized smoothed observation (or model) residuals[1] are the difference between the data at time t and the model fit at time t conditioned on all the data and

[1] also called smoothations in the literature to distinguish them from innovations, which are $\mathbf{y}_t - \mathrm{E}[\mathbf{Y}_t|x_t^{t-1}]$. Notice that for innovations the expectation is conditioned on the data up to time $t-1$ while for smoothations, we condition on all the data.

then standardized by the observation variance:

$$\begin{aligned} \hat{v}_t &= y_t - \hat{y}_t^T \\ e_t &= \frac{1}{\sqrt{\text{var}(\hat{v}_t)}} \hat{v}_t \end{aligned} \tag{12.8}$$

These residuals should have (asymptotically) a t-distribution (Kohn and Ansley, 1989, section 3) and by looking at the residuals, we can identify potential outlier data points–or more accurately, we can identify data points that do not fit the model (Equation 12.5). The call `MARSSresiduals(..., type="tT")` will compute the smoothation (or auxilliary) residuals for a marssMLE object (output by a MARSS call). `MARSSresiduals()` returns two types of standardized residuals (also called auxiliary residuals): Cholesky standardized residuals and marginal standardized residuals. We are using the latter here. The residuals are returned as a $n+m \times T$ matrix. The first n rows are the estimated $\mathbf{v}_t$ standardized observation residuals and the next m rows are the estimated $\mathbf{w}_t$ standardized state residuals (discussed below). `residuals(..., type="tT")` will also return the smoothations but in a data frame. Here we use `MARSSresiduals()` which return them in a list of matrices.

```
resids.0 <- MARSSresiduals(kem.0, type = "tT")$mar.residuals
resids.1 <- MARSSresiduals(kem.1, type = "tT")$mar.residuals
resids.2 <- MARSSresiduals(kem.2, type = "tT")$mar.residuals
```

Figure 12.3 shows the observation residuals for the three models developed above. We immediately see that model 0 (flat level) and model 1 (linear declining level) have problems because the residuals are all positive for the first part of the time series and then all negative. The residuals should not be temporally correlated like that. Model 2 with a stochastic level shows well-behaving residuals with low temporal correlation between t and $t-1$. Looking at the residuals for model 2, we see that there are a number of years with flow levels that appear to be outliers (are beyond the dashed level lines).

12.3.2 Detecting sudden level changes

The standardized smoothed state residuals (f_t below) are the difference between the estimated state at time t and the estimated state at time $t-1$ conditioned on all the data and then standardized by the standard deviation:

$$\begin{aligned} \hat{w}_t &= x_t^T - x_{t-1}^T \\ f_t &= \frac{1}{\sqrt{\text{var}(\hat{w}_t)}} \hat{w}_t \end{aligned} \tag{12.9}$$

These state residuals do not show simple changes in the average level; x_t is clearly changing in Figure 12.2, bottom panel. Instead we are looking for “breaks" or sudden changes in the level. The bottom panel of Figure 12.4 shows the standardized state residuals (f_t). This shows, as we can see by eye, the average flow level in the

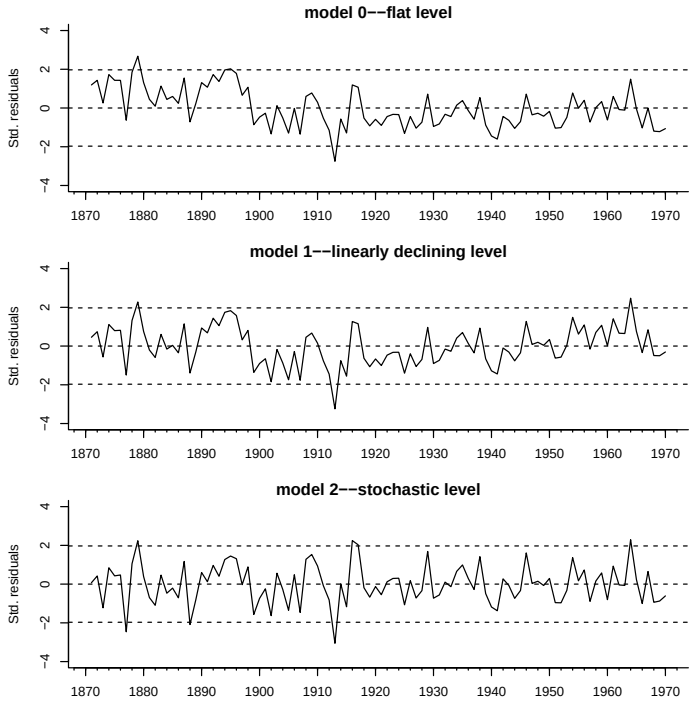

Fig. 12.3. The marginal standardized observation residuals from models 0, 1, and 2. These residuals are the standardized $\hat{v}_t$. The dashed lines are the 95% CIs for a t-distribution.

Nile appears to have suddenly changed around the turn of the century when the first Aswan dam was built. The top panel shows the standardized observation residuals for comparison.

12.3.3 Detecting changes in the drift parameter in a random walk model

The model of a random walk with a fixed u drift term is:

$$\begin{aligned} y_t &= x_t + v_t \text{ where } v_t \sim \mathrm{N}(0, r) \\ x_t &= x_{t-1} + u + w_t \text{ where } w_t \sim \mathrm{N}(0, q) \\ x_0 &= \pi \end{aligned} \tag{12.10}$$

Same as we did for the level, we can model the drift as a random walk and explore whether u_t has changed over time or whether it has experienced sudden breaks. The stochastic level and trend model is

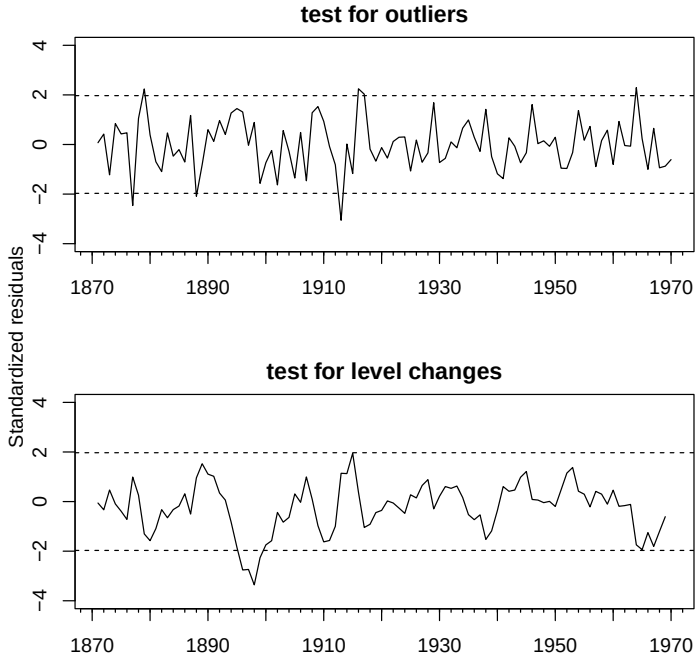

Fig. 12.4. Top panel, the marginal standardized observation residuals. Bottom panel, the standardized state residuals. This replicates Figure 12 in Koopman et al. (1999).

$$\begin{aligned} y_t &= x_t + v_t \text{where } v_t \sim \mathrm{N}(0, r) \\ x_t &= x_{t-1} + u_{t-1} + w_t \text{ where } w_t \sim \mathrm{N}(0, q) \\ u_t &= u_{t-1} + z_t \text{ where } z_t \sim \mathrm{N}(0, p) \\ x_0 &= \pi_x \text{ and } u_0 = \pi_u \end{aligned} \tag{12.11}$$

Write the model in MARSS form:

$$\begin{aligned} y_t &= \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} x_t \\ u_t \end{bmatrix} + v_t \\ \begin{bmatrix} x_t \\ u_t \end{bmatrix} &= \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_{t-1} \\ u_{t-1} \end{bmatrix} + \begin{bmatrix} w_t \\ z_t \end{bmatrix} \text{ where } \begin{bmatrix} v_t \\ z_t \end{bmatrix} \sim \mathrm{MVN}\left(0, \begin{bmatrix} q & 0 \\ 0 & p \end{bmatrix}\right) \end{aligned} \tag{12.12}$$

The model is then:

$$
\begin{aligned}
y_t &= \mathbf{Z}\mathbf{x} + \mathbf{a} + v_t \\
\mathbf{x}_t &= \mathbf{B}\mathbf{x}_{t-1} + \mathbf{u} + \mathbf{w}_t \\
\\
\mathbf{Z} &= \begin{bmatrix} 1 & 0 \end{bmatrix} \; \mathbf{a} = \begin{bmatrix} 0 \end{bmatrix} \; \mathbf{R} = \begin{bmatrix} r \end{bmatrix} \\
\mathbf{B} &= \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \; \mathbf{u} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \; \mathbf{Q} = \begin{bmatrix} q & 0 \\ 0 & p \end{bmatrix} \; \mathbf{x}_0 = \begin{bmatrix} \pi_x \\ \pi_u \end{bmatrix}
\end{aligned}
\tag{12.13}
$$

We then write the model list as:

```
mod.nile.3 <- list(
  Z = matrix(c(1, 0), 1, 2), A = matrix(0), R = matrix("r"),
  B = matrix(c(1, 0, 1, 1), 2, 2), U = matrix(0, 2, 1),
  Q = matrix(list("q", 0, 0, "p"), 2, 2),
  x0 = matrix(c("x", "u"), 2, 1)
)
```

This model takes a long time to fit with the EM algorithm[2]. We could run the EM algorithm a long time, but there is a quicker trick in this case. We will run the EM algorithm for a few iterations and stop before convergence. Then we will use the fit from the EM algorithm as the initial condition for the faster BFGS algorithm for the final approach to the maximum-likelihood:

```
model <- mod.nile.3
kem.3 <- MARSS(dat,
  model = model, inits = list(x0 = matrix(c(1000, -4), 2, 1)),
  control = list(maxit = 20), silent = TRUE
)
kem.3 <- MARSS(dat,
  model = model, inits = kem.3,
  method = "BFGS", silent = TRUE
)
summary(kem.3)
```

```
m: 2 state process(es) named X1 X2
n: 1 observation time series named Nile

  term      estimate
1  R.r  1.611549e+04
2  Q.q  8.427672e+02
3  Q.p  3.693557e-07
4 x0.x  1.118521e+03
5 x0.u -3.108126e+00
```

[2] Normally this type of model is fit with a fixed diffuse initial condition which makes the fitting much faster. See the chapter on Structural time series models

The Nile data is not a good example since the variance for the slope is close to zero so the residual line is just a flat 0 (see value for `Q.p`). Let's run the same model on the `WWWusage` dataset (Figure 12.5).

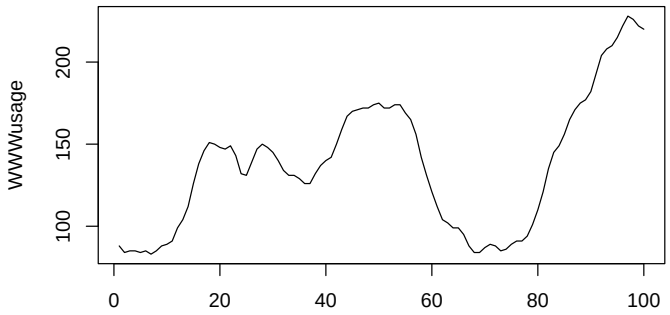

Fig. 12.5. The WWWusage data set.

```
dat <- as.vector(WWWusage)
kem.3 <- MARSS(dat,
  model = model, inits = list(x0 = matrix(0, 2, 1)),
  control = list(maxit = 20), silent = TRUE
)
kem.3 <- MARSS(dat,
  model = model, inits = kem.3,
  method = "BFGS", silent = TRUE
)
summary(kem.3)
```

```
m: 2 state process(es) named X1 X2
n: 1 observation time series named Y1

  term      estimate
1  R.r  1.146401e-20
2  Q.q  6.363461e-24
3  Q.p  1.267741e+01
4 x0.x  9.045973e+01
5 x0.u -2.459726e+00
```

Figure 12.6 shows the standardized residuals for the u, which is the 2nd state in **x**. There appears to be unusually large changes in the trend around year 25. The other changes in the trend are consistent with a random walk; i.e., the trend changes but only the rapid change near year 25 is inconsistent with the estimated trend random walk.

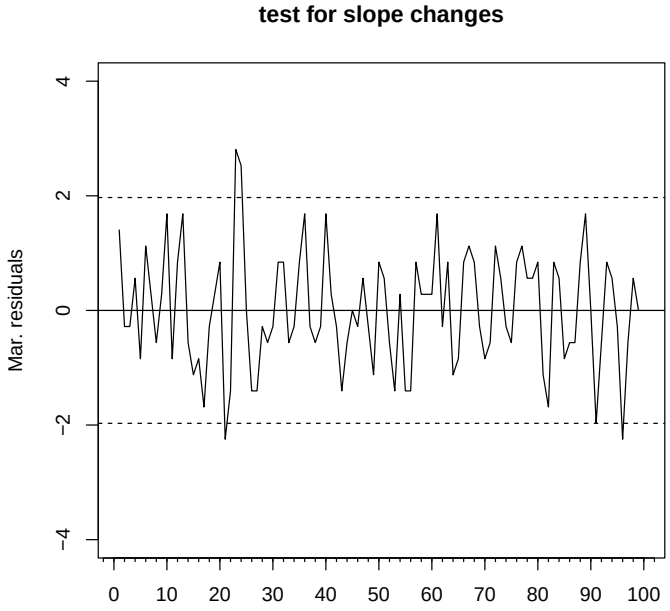

Fig. 12.6. The marginal standardized residuals for slope changes for the WWWusage model.

12.4 Discussion

This chapter shows the basic strategy for doing shock detection sensu Harvey et al. using standardized residuals. This was illustrated with stochastic level and trend models. Stochastic level and trend models are also called Structural Time Series models. You can find more examples of these models in Chapter 19.

13

Incorporating covariates into MARSS models

13.1 Covariates as inputs

A MARSS model with covariate effects in both the process and observation components is written as:

$$\begin{aligned} \mathbf{x}_t &= \mathbf{B}_t\mathbf{x}_{t-1} + \mathbf{u}_t + \mathbf{C}_t\mathbf{c}_t + \mathbf{w}_t, \text{ where } \mathbf{w}_t \sim \text{MVN}(0, \mathbf{Q}_t) \\ \mathbf{y}_t &= \mathbf{Z}_t\mathbf{x}_t + \mathbf{a}_t + \mathbf{D}_t\mathbf{d}_t + \mathbf{v}_t, \text{ where } \mathbf{v}_t \sim \text{MVN}(0, \mathbf{R}_t) \end{aligned} \tag{13.1}$$

where $\mathbf{c}_t$ is the $p \times 1$ vector of covariates (e.g., temperature, rainfall) which affect the states and $\mathbf{d}_t$ is a $q \times 1$ vector of covariates (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 $\mathbf{y}_t$.

With the `MARSS()` function, one can fit this model by 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`. Because $\mathbf{C}$ and $\mathbf{D}$ are matrices, 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. If they are time-constant, then they can be specified as 2-dimensional matrices.

13.2 Examples using plankton data

Here we show some examples using the Lake Washington plankton data set and covariates in that dataset. We use the 10 years of data from 1965-1974 (Figure 13.1),

---

Type `RShowDoc("Chapter_Covariates.R",package="MARSS")` at the R command line to open a file with all the code for the examples in this chapter.

a decade with particularly high green and blue-green algae levels. We use the transformed plankton dataset which has 0s replaced with NAs. Below, we set up the data and z-score the data. The original data were already z-scored, but we changed the mean when we sub-sampled the years so need to z-score again.

```
fulldat <- lakeWAplanktonTrans
years <- fulldat[, "Year"] >= 1965 & fulldat[, "Year"] < 1975
dat <- t(fulldat[years, c("Greens", "Bluegreens")])
the.mean <- apply(dat, 1, mean, na.rm = TRUE)
the.sigma <- sqrt(apply(dat, 1, var, na.rm = TRUE))
dat <- (dat - the.mean) * (1 / the.sigma)
```

Next we set up the covariate data, temperature and total phosphorous. We z-score the covariates to standardize and remove the mean. MARSS has a function to z-score data, so we used that from here out.

```
covariates <- rbind(
  Temp = fulldat[years, "Temp"],
  TP = fulldat[years, "TP"]
)
# z.score the covariates
covariates <- zscore(covariates)
```

13.3 Observation-error only model

We can estimate the effect of the covariates using a process-error only model, an observation-error only model, or a model with both types of error. An observation-error only model is a multivariate regression, and we will start here so you see the relationship of MARSS model to more familiar linear regression models.

13.3.1 Multivariate linear regression

In a standard multivariate linear regression, we only have an observation model with independent errors (*i.e.*, the state process does not appear in the model):

$$\mathbf{y}_t = \mathbf{a} + \mathbf{D}\mathbf{d}_t + \mathbf{v}_t, \text{ where } \mathbf{v}_t \sim \text{MVN}(0, \mathbf{R}) \tag{13.2}$$

The elements in **a** are the intercepts and those in **D** are the slopes (effects). We have dropped the t subscript on **a** and **D** because these will be modeled as time-constant. Writing this out for the two plankton and the two covariates we get:

$$\begin{bmatrix} y_g \\ y_{bg} \end{bmatrix}_t = \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} + \begin{bmatrix} \beta_{\text{g,temp}} & \beta_{\text{g,tp}} \\ \beta_{\text{bg,temp}} & \beta_{\text{bg,tp}} \end{bmatrix} \begin{bmatrix} \text{temp} \\ \text{tp} \end{bmatrix}_{t-1} + \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}_t \tag{13.3}$$

Let's fit this model with MARSS(). The **x** part of the model is irrelevant so we want to fix the parameters in that part of the model. We won't set $\mathbf{B} = 0$ or $\mathbf{Z} = 0$ since that might cause numerical issues for the Kalman filter. Instead we fix them as identity matrices and fix $\mathbf{x}_0 = 0$ so that $\mathbf{x}_t = 0$ for all t.

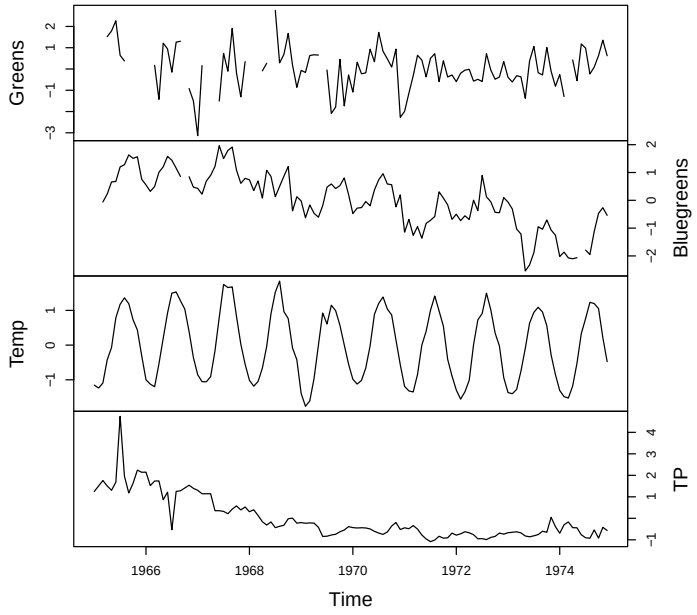

Fig. 13.1. Time series of green andblue-green algae abundances in Lake Washington along with the temperature and total phosphorous covariates.

```
Q <- U <- x0 <- "zero"
B <- Z <- "identity"
d <- covariates
A <- "zero"
D <- "unconstrained"
y <- dat # to show relationship between dat & the equation
model.list <- list(
  B = B, U = U, Q = Q, Z = Z, A = A,
  D = D, d = d, x0 = x0
)
kem <- MARSS(y, model = model.list)
```

```
Success! algorithm run for 15 iterations. abstol and log-log tests passed.
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
Algorithm ran 15 (=minit) iterations and convergence was reached.
Log-likelihood: -276.4287
AIC: 562.8573   AICc: 563.1351

                    Estimate
R.diag                 0.706
D.(Greens,Temp)        0.367
D.(Bluegreens,Temp)    0.392
D.(Greens,TP)          0.058
D.(Bluegreens,TP)      0.535
Initial states (x0) defined at t=0

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

We set `A="zero"` because the data and covariates have been demeaned. Of course, one can do multiple regression in R using, say, `lm()`, and that would be much, much faster. The EM algorithm is over-kill here, but it is shown so that you see how a standard multivariate linear regression model is written as a MARSS model in matrix form.

13.3.2 Multivariate linear regression with autocorrelated errors

We can add a twist to the standard multivariate linear regression model, and instead of having temporally *i.i.d.* errors in the observation process, we'll assume autoregressive errors. There is still no state process in our model, but we will use the state part of a MARSS model to model our errors. Mathematically, this can be written as

$$\begin{gathered}\mathbf{x}_t = \mathbf{B}\mathbf{x}_{t-1} + \mathbf{w}_t, \text{ where } \mathbf{w}_t \sim \text{MVN}(0, \mathbf{Q}) \\ \mathbf{y}_t = \mathbf{D}_t\mathbf{d}_t + \mathbf{x}_t\end{gathered} \tag{13.4}$$

Here, the $\mathbf{x}_t$ are the errors for the observation model; they are modeled as an autoregressive process via the $\mathbf{x}$ equation. We drop the $\mathbf{v}_t$ (set $\mathbf{R} = 0$) because the $\mathbf{x}_t$ in the $\mathbf{y}$ equation are now the observation errors. As usual, we have left the intercepts ($\mathbf{a}$ and $\mathbf{u}$) off since the data and covariates are all demeaned.

Here's how we fit this model in MARSS:

```
Q <- "unconstrained"
B <- "diagonal and unequal"
A <- U <- x0 <- "zero"
R <- "diagonal and equal"
d <- covariates
D <- "unconstrained"
y <- dat
model.list <- list(
```

```
  B = B, U = U, Q = Q, Z = Z, A = A,
  R = R, D = D, d = d, x0 = x0
)
control.list <- list(maxit = 1500)
kem <- MARSS(y, model = model.list, control = control.list)
```

```
Success! abstol and log-log tests passed at 79 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 79 iterations.
Log-likelihood: -209.3408
AIC: 438.6816   AICc: 439.7243

                              Estimate
R.diag                          0.0428
B.(X.Greens,X.Greens)           0.2479
B.(X.Bluegreens,X.Bluegreens)   0.9136
Q.(1,1)                         0.7639
Q.(2,1)                        -0.0285
Q.(2,2)                         0.1265
D.(Greens,Temp)                 0.3777
D.(Bluegreens,Temp)             0.2621
D.(Greens,TP)                   0.0459
D.(Bluegreens,TP)               0.0675
Initial states (x0) defined at t=0

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

You can try setting **B** to identity and MARSS will fit a model with non-mean-reverting autoregressive errors to the data. It is not done here since it turns out that that is not a very good model and it takes a long time to fit. If you try it, you'll see that **Q** gets small meaning that the **x** part is being removed from the model.

13.4 Process-error only model

Now let's model the data as an autoregressive process observed without error, and incorporate the covariates into the process model. Note that this is much different from typical linear regression models. The **x** part represents our model of the data (in this case plankton species). How is this different from the autoregressive observation errors? Well, we are modeling our data as autoregressive so data at $t-1$ affects the

data at t. Population abundances are inherently autoregressive so this model is a bit closer to the underlying mechanism generating the data. Here is our new process model for plankton abundance. $\mathbf{x}$ is the plankton abundance.

$$\mathbf{x}_t = \mathbf{x}_{t-1} + \mathbf{C}\mathbf{c}_t + \mathbf{w}_t, \text{ where } \mathbf{w}_t \sim \text{MVN}(0, \mathbf{Q}) \tag{13.5}$$

We can fit this as follows:

```
R <- A <- U <- "zero"
B <- Z <- "identity"
Q <- "equalvarcov"
C <- "unconstrained"
x <- dat # to show the relation between dat & the equations
model.list <- list(
  B = B, U = U, Q = Q, Z = Z, A = A,
  R = R, C = C, c = covariates
)
kem <- MARSS(x, model = model.list)
```

```
Success! algorithm run for 15 iterations. abstol and log-log tests passed.
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
Algorithm ran 15 (=minit) iterations and convergence was reached.
Log-likelihood: -285.0732
AIC: 586.1465   AICc: 586.8225

                     Estimate
Q.diag                 0.7269
Q.offdiag             -0.0210
x0.X.Greens           -0.5189
x0.X.Bluegreens       -0.2431
C.(X.Greens,Temp)     -0.0434
C.(X.Bluegreens,Temp)  0.0988
C.(X.Greens,TP)       -0.0589
C.(X.Bluegreens,TP)    0.0104
Initial states (x0) defined at t=0

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

Now, it looks like temperature has a strong negative effect on algae, which is odd. Also our log-likelihood dropped a lot. Well, the data do not look at all like a random walk model (*i.e.*, where $\mathbf{B} = 1$), which we can see from the plot of the data (Figure

13.1). The data are fluctuating about some mean so let's switch to a better autoregressive model—a mean-reverting model. To do this, we will allow the diagonal elements of **B** to be something other than 1.

```
model.list$B <- "diagonal and unequal"
kem <- MARSS(dat, model = model.list)
```

```
Success! algorithm run for 15 iterations. abstol and log-log tests passed.
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
Algorithm ran 15 (=minit) iterations and convergence was reached.
Log-likelihood: -236.6106
AIC: 493.2211   AICc: 494.2638

                              Estimate
B.(X.Greens,X.Greens)           0.1981
B.(X.Bluegreens,X.Bluegreens)   0.7672
Q.diag                          0.4899
Q.offdiag                      -0.0221
x0.X.Greens                    -1.2915
x0.X.Bluegreens                -0.4179
C.(X.Greens,Temp)               0.2844
C.(X.Bluegreens,Temp)           0.1655
C.(X.Greens,TP)                 0.0332
C.(X.Bluegreens,TP)             0.1340
Initial states (x0) defined at t=0

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

Notice that the log-likelihood goes up quite a bit, which means that the mean-reverting model fits the data much better.

With this model, we are estimating $\mathbf{x}_0^0$ (the starting value for the Kalman filter). If we set `model$tinitx=1`, to use $\mathbf{x}_1^0$ as the starting value instead, we will get a error message that **R** diagonals are equal to 0 and we need to fix `x0`. This is a restriction of the (default) EM algorithm having to do with the update equation for $\mathbf{x}_1^0$. We cannot use BFGS unless we set **Q** to be either unconstrained or diagonal because the way **Q** is being estimated using the BFGS algorithm to ensure that the matrix stays positive-definite (via a Cholesky transformation) does not allow any constraints on **Q**.

13.5 Both process- & observation-error model

The {MARSS} package is really designed for state-space models where you have errors (**v** and **w**) in both the process and observation models. For example,

$$\begin{aligned} \mathbf{x}_t &= \mathbf{B}\mathbf{x}_{t-1} + \mathbf{C}_t\mathbf{c}_t + \mathbf{w}_t, \text{ where } \mathbf{w}_t \sim \text{MVN}(0, \mathbf{Q}) \\ \mathbf{y}_t &= \mathbf{x}_{t-1} + \mathbf{v}_t, \text{ where } \mathbf{v}_t \sim \text{MVN}(0, \mathbf{R}), \end{aligned} \tag{13.6}$$

x is the true algae abundances and **y** is the observation of the **x**'s.

Let's say we knew that the observation variance on the algae measurements was about 0.16 and we wanted to include that known value in the model. To do that, we can simply add **R** to the model list from the process-error only model in the last example.

```
model.list$R <- diag(0.16, 2)
kem <- MARSS(dat, model = model.list)
```

```
Success! abstol and log-log tests passed at 27 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 27 iterations.
Log-likelihood: -241.5301
AIC: 503.0603   AICc: 504.1029

                              Estimate
B.(X.Greens,X.Greens)          0.31497
B.(X.Bluegreens,X.Bluegreens)  0.76205
Q.diag                         0.33374
Q.offdiag                     -0.00331
x0.X.Greens                   -0.90020
x0.X.Bluegreens               -0.40473
C.(X.Greens,Temp)              0.23448
C.(X.Bluegreens,Temp)          0.16960
C.(X.Greens,TP)                0.02423
C.(X.Bluegreens,TP)            0.14120
Initial states (x0) defined at t=0

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

Note, our estimates of the effect of temperature and total phosphorous are not that different than what you get from a simple multiple regression (our first example). This might be because the autoregressive component is small, meaning the estimated diagonals on the **B** matrix are small.

13.6 Including seasonal effects in MARSS models

Time-series data are often collected at intervals with some implicit seasonality. For example, quarterly earnings for a business, monthly rainfall totals, or hourly air temperatures. In those cases, it is often helpful to extract any recurring seasonal patterns that might otherwise mask some of the other temporal dynamics we are interested in examining.

Here we show a few approaches for including seasonal effects using the Lake Washington plankton data, which were collected monthly. The following examples will use all five phytoplankton species from Lake Washington. First, let's set up the data.

```
years <- fulldat[, "Year"] >= 1965 & fulldat[, "Year"] < 1975
phytos <- c(
  "Diatoms", "Greens", "Bluegreens",
  "Unicells", "Other.algae"
)
dat <- t(fulldat[years, phytos])
# z.score data again because we changed the mean when we subsampled
dat <- zscore(dat)
# number of time periods/samples
TT <- ncol(dat)
```

13.6.1 Seasonal effects as fixed factors

One common approach for estimating seasonal effects is to treat each one as a fixed factor. This adds an estimated parameter for each season (e.g., 24 hours per day, 4 quarters per year). The plankton data are collected monthly, so we will treat each month as a fixed factor. To fit a model with fixed month effects, we create a $12 \times T$ covariate matrix $\mathbf{c}$ with one row for each month (Jan, Feb, ...) and one column for each time point. We put a 1 in the January row for each column corresponding to a January time point, a 1 in the February row for each column corresponding to a February time point, and so on. All other values of $\mathbf{c}$ equal 0. The following code will create such a $\mathbf{c}$ matrix.

```
# number of "seasons" (e.g., 12 months per year)
period <- 12
# first "season" (e.g., Jan = 1, July = 7)
per.1st <- 1
# create factors for seasons
c.in <- diag(period)
for (i in 2:(ceiling(TT / period))) {
  c.in <- cbind(c.in, diag(period))
}
# trim c.in to correct start & length
c.in <- c.in[, (1:TT) + (per.1st - 1)]
```

```
# better row names
rownames(c.in) <- month.abb
```

Next we need to set up the form of the **C** matrix which defines any constraints we want to set on the month effects. **C** is a 5×12 matrix. Five taxon and 12 month effects. If we wanted each taxon to have the same month effect, a common month effect across all taxon, then we have the same value in each **C** column[1]:

```
C <- matrix(month.abb, 5, 12, byrow = TRUE)
C
```

```
     [,1]  [,2]  [,3]  [,4]  [,5]  [,6]  [,7]  [,8]  [,9]
[1,] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep"
[2,] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep"
[3,] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep"
[4,] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep"
[5,] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep"
     [,10] [,11] [,12]
[1,] "Oct" "Nov" "Dec"
[2,] "Oct" "Nov" "Dec"
[3,] "Oct" "Nov" "Dec"
[4,] "Oct" "Nov" "Dec"
[5,] "Oct" "Nov" "Dec"
```

Notice, that **C** only has 12 values in it, the 12 common month effects. However, for this example, we will let each taxon have a different month effect thus allowing different seasonality for each taxon. For this model, we want each value in **C** to be unique:

```
C <- "unconstrained"
```

Now **C** has $5 \times 12 = 60$ separate effects.

Then we set up the form for the rest of the model parameters. We make the following assumptions:

```
# Each taxon has unique density-dependence
B <- "diagonal and unequal"
# Independent process errors
Q <- "diagonal and unequal"
# We have demeaned the data & are fitting a mean-reverting model
# by estimating a diagonal B, thus
U <- "zero"
# Each obs time series is associated with only one process
Z <- "identity"
# The data are demeaned & fluctuate around a mean
A <- "zero"
# Observation errors are independent, but they
```

[1] `month.abb` is a R constant that gives month abbreviations in text.

```
# have similar variance due to similar collection methods
R <- "diagonal and equal"
# No covariate effects in the obs equation
D <- "zero"
d <- "zero"
```

Now we can set up the model list for MARSS and fit the model (results are not shown since they are verbose with 60 different month effects).

```
model.list <- list(
  B = B, U = U, Q = Q, Z = Z, A = A, R = R,
  C = C, c = c.in, D = D, d = d
)
seas.mod.1 <- MARSS(dat, model = model.list, control = list(maxit = 1500))
# Get the estimated seasonal effects
# rows are taxa, cols are seasonal effects
seas.1 <- coef(seas.mod.1, type = "matrix")$C
rownames(seas.1) <- phytos
colnames(seas.1) <- month.abb
```

The top panel in Figure 13.2 shows the estimated seasonal effects for this model. Note that if we had set `U="unequal"`, we would need to set one of the columns of **C** to zero because the model would be under-determined (infinite number of solutions). If we subtracted the mean January abundance off each time series, we could set the January column in **C** to 0 and get rid of 5 estimated effects.

13.6.2 Seasonal effects as a polynomial

The fixed factor approach required estimating 60 effects. Another approach is to model the month effect as a 3^{rd}-order (or higher) polynomial: $a + b \times m + c \times m^2 + d \times m^3$ where m is the month number. This approach has less flexibility but requires only 20 estimated parameters (*i.e.*, 4 regression parameters times 5 taxa). To do so, we create a $4 \times T$ covariate matrix **c** with the rows corresponding to 1, m, m^2, and m^3, and the columns again corresponding to the time points. Here is how to set up this matrix:

```
# number of "seasons" (e.g., 12 months per year)
period <- 12
# first "season" (e.g., Jan = 1, July = 7)
per.1st <- 1
# order of polynomial
poly.order <- 3
# create polynomials of months
month.cov <- matrix(1, 1, period)
for (i in 1:poly.order) {
  month.cov <- rbind(month.cov, (1:12)^i)
}
```

```
# our c matrix is month.cov replicated once for each year
c.m.poly <- matrix(month.cov, poly.order + 1, TT + period, byrow = FALSE)
# trim c.in to correct start & length
c.m.poly <- c.m.poly[, (1:TT) + (per.1st - 1)]
# Everything else remains the same as in the previous example
model.list <- list(
  B = B, U = U, Q = Q, Z = Z, A = A, R = R,
  C = C, c = c.m.poly, D = D, d = d
)
seas.mod.2 <- MARSS(dat, model = model.list, control = list(maxit = 1500))
```

The effect of month m for taxon i is $a_i + b_i \times m + c_i \times m^2 + d_i \times m^3$, where a_i, b_i, c_i and d_i are in the i-th row of $\mathbf{C}$. We can now calculate the matrix of seasonal effects as follows, where each row is a taxon and each column is a month:

```
C.2 <- coef(seas.mod.2, type = "matrix")$C
seas.2 <- C.2 %*% month.cov
rownames(seas.2) <- phytos
colnames(seas.2) <- month.abb
```

The middle panel in Figure 13.2 shows the estimated seasonal effects for this polynomial model.

13.6.3 Seasonal effects as a Fourier series

The factor approach required estimating 60 effects, and the 3rd order polynomial model was an improvement at only 20 parameters. A third option is to use a discrete Fourier series, which is combination of sine and cosine waves; it would require only 10 parameters. Specifically, the effect of month m on taxon i is $a_i \times \cos(2\pi m/p) + b_i \times \sin(2\pi m/p)$, where p is the period (e.g., 12 months, 4 quarters), and a_i and b_i are contained in the i-th row of $\mathbf{C}$.

We begin by defining the $2 \times T$ seasonal covariate matrix $\mathbf{c}$ as a combination of 1 cosine and 1 sine wave:

```
cos.t <- cos(2 * pi * seq(TT) / period)
sin.t <- sin(2 * pi * seq(TT) / period)
c.Four <- rbind(cos.t, sin.t)
```

Everything else remains the same and we can fit this model as follows:

```
model.list <- list(
  B = B, U = U, Q = Q, Z = Z, A = A, R = R,
  C = C, c = c.Four, D = D, d = d
)
seas.mod.3 <- MARSS(dat, model = model.list, control = list(maxit = 1500))
```

We make our seasonal effect matrix as follows:

```
C.3 <- coef(seas.mod.3, type = "matrix")$C
# The time series of net seasonal effects
seas.3 <- C.3 %*% c.Four[, 1:period]
rownames(seas.3) <- phytos
colnames(seas.3) <- month.abb
```

The bottom panel in Figure 13.2 shows the estimated seasonal effects for this seasonal-effects model based on a discrete Fourier series.

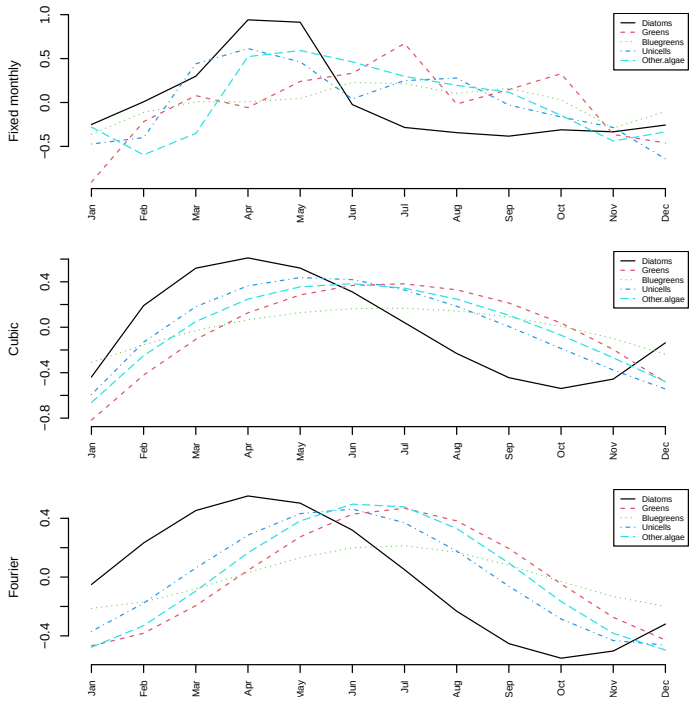

Fig. 13.2. Estimated monthly effects for the three approaches to estimating seasonal effects. Top panel: each month modeled as a separate fixed effect for each taxon (60 parameters); Middle panel: monthly effects modeled as a 3rd order polynomial (20 parameters); Bottom panel: monthly effects modeled as a discrete Fourier series (10 parameters).

Rather than rely on our eyes to judge model fits, we should formally assess which of the three approaches offers the most parsimonious fit to the data. Here is a table of AICc values for the three models:

```
data.frame(
  Model = c("Fixed", "Cubic", "Fourier"),
  AICc = round(c(
```

```
    seas.mod.1$AICc,
    seas.mod.2$AICc,
    seas.mod.3$AICc
  ), 1),
  stringsAsFactors = FALSE
)
```

```
    Model   AICc
1   Fixed 1188.4
2   Cubic 1144.9
3 Fourier 1127.4
```

The model selection results indicate that the model with monthly seasonal effects estimated via the discrete Fourier sequence is the most parsimonious of the three models. Its AICc value is much lower than either the polynomial or fixed-effects models.

13.7 Model diagnostics

We will examine some basic model diagnostics for these three approaches by looking at plots of the model residuals (innovations) and their autocorrelation functions (ACFs) for all five taxa using the following code:

```
for (i in 1:3) {
  dev.new()
  modn <- paste("seas.mod", i, sep = ".")
  for (j in 1:5) {
    plot.ts(MARSSresiduals(modn, type = "tt1")$model.residuals[j, ],
      ylab = "Residual", main = phytos[j]
    )
    abline(h = 0, lty = "dashed")
    acf(MARSSresiduals(modn, type = "tt1")$model.residuals[j, ],
      na.action = na.pass
    )
  }
}
```

Figures 13.3 to 13.5 shows these diagnostics for the three models. The model residuals for all taxa and models appear to show significant negative autocorrelation at lag=1, suggesting that a model with seasonal effects is inadequate to capture all of the systematic variation in phytoplankton abundance.

13.8 Covariates with missing values or observation error

The specific formulation of Equation 13.1 creates restrictions on the assumptions regarding the covariate data. You have to assume that your covariate data has no

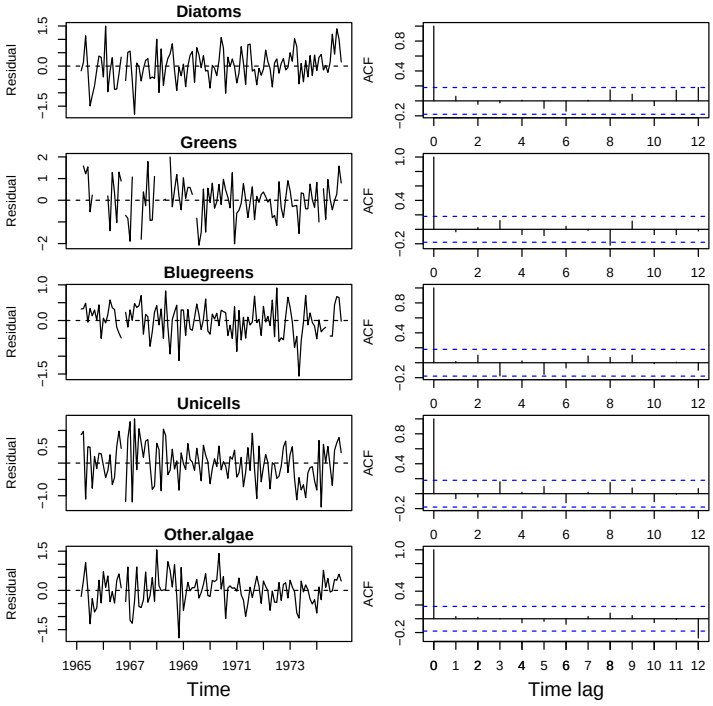

Fig. 13.3. Model residuals and their ACF for the model with fixed monthly effects.

error, which is probably not true. You cannot have missing values in your covariate data, again unlikely. You cannot combine instrument time series; for example, if you have two temperature recorders with different error rates and biases. Also, what if you have one noisy temperature recorder in the first part of your time series and then you switch to a much better recorder in the second half of your time series? All these problems require pre-analysis massaging of the covariate data, leaving out noisy and gappy covariate data, and making what can feel like arbitrary choices about which covariate time series to include.

To circumvent these potential problems and allow more flexibility in how we incorporate covariate data, one can instead treat the covariates as components of an auto-regressive process by including them in both the process and observation models. Beginning with the process equation, we can write

$$\begin{gathered}\begin{bmatrix}\mathbf{x}^{(v)}\\ \mathbf{x}^{(c)}\end{bmatrix}_t = \begin{bmatrix}\mathbf{B}^{(v)} & \mathbf{C}\\ 0 & \mathbf{B}^{(c)}\end{bmatrix}\begin{bmatrix}\mathbf{x}^{(v)}\\ \mathbf{x}^{(c)}\end{bmatrix}_{t-1} + \begin{bmatrix}\mathbf{u}^{(v)}\\ \mathbf{u}^{(c)}\end{bmatrix} + \mathbf{w}_t,\\ \mathbf{w}_t \sim \text{MVN}\left(0, \begin{bmatrix}\mathbf{Q}^{(v)} & 0\\ 0 & \mathbf{Q}^{(c)}\end{bmatrix}\right)\end{gathered} \tag{13.7}$$

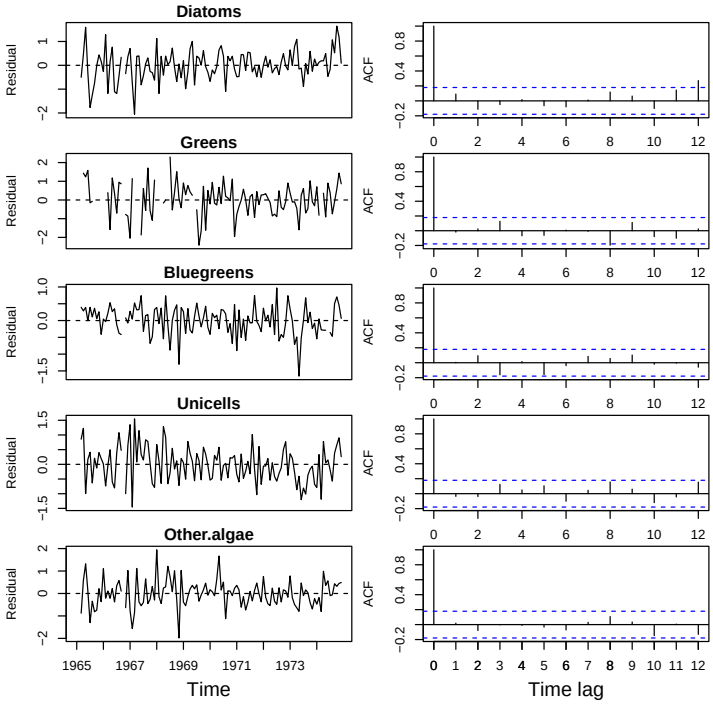

Fig. 13.4. Model residuals and their ACF for the model with monthly effects modeled as a 3^{rd}-rd order polynomial.

The elements with superscript (v) are for the k variate states and those with superscript (c) are for the q covariate states. The dimension of $\mathbf{x}^{(c)}$ is $q \times 1$ and q is not necessarily equal to p, the number of covariate observation time series in your dataset. Imagine, for example, that you have two temperature sensors and you are combining these data. Then you have two covariate observation time series ($p = 2$) but only one underlying covariate state time series ($q = 1$). The matrix $\mathbf{C}$ is dimension $k \times q$, and $\mathbf{B}^{(c)}$ and $\mathbf{Q}^{(c)}$ are dimension $q \times q$. The dimension[2] of $\mathbf{x}^{(v)}$ is $k \times 1$, and $\mathbf{B}^{(v)}$ and $\mathbf{Q}^{(v)}$ are dimension $k \times k$.

Next, we can write the observation equation in an analogous manner, such that

$$\begin{bmatrix}\mathbf{y}^{(v)}\\ \mathbf{y}^{(c)}\end{bmatrix}_t = \begin{bmatrix}\mathbf{Z}^{(v)} & \mathbf{D}\\ 0 & \mathbf{Z}^{(c)}\end{bmatrix}\begin{bmatrix}\mathbf{x}^{(v)}\\ \mathbf{x}^{(c)}\end{bmatrix}_t + \begin{bmatrix}\mathbf{a}^{(v)}\\ \mathbf{a}^{(c)}\end{bmatrix} + \mathbf{v}_t,$$
$$\mathbf{v}_t \sim \text{MVN}\left(0, \begin{bmatrix}\mathbf{R}^{(v)} & 0\\ 0 & \mathbf{R}^{(c)}\end{bmatrix}\right) \tag{13.8}$$

[2] The dimension of $\mathbf{x}$ is always denoted m. If your process model includes only variates, then $k = m$, but now your process model includes k variates and q covariate states so $m = k + q$.

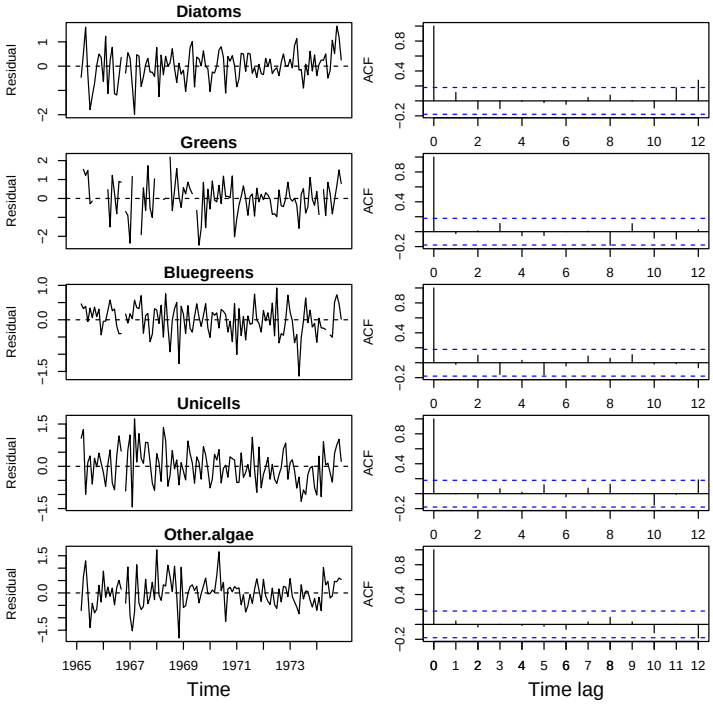

Fig. 13.5. Model residuals and their ACF for the model with monthly effects estimated using a Fourier transform.

The dimension of $\mathbf{y}^{(c)}$ is $p \times 1$, where p is the number of covariate observation time series in your dataset. The dimension of $\mathbf{y}^{(v)}$ is $l \times 1$, where l is the number of variate observation time series in your dataset. The total dimension of $\mathbf{y}$ is $l + p$. The matrix $\mathbf{D}$ is dimension $l \times q$, $\mathbf{Z}^{(c)}$ is dimension $p \times q$, and $\mathbf{R}^{(c)}$ are dimension $p \times p$. The dimension of $\mathbf{Z}^{(v)}$ is dimension $l \times k$, and $\mathbf{R}^{(v)}$ are dimension $l \times l$.

The $\mathbf{D}$ matrix would presumably have a number of all zero rows in it, as would the $\mathbf{C}$ matrix. The covariates that affect the states would often be different than the covariates that affect the observations. For example, mean annual temperature would affect population growth rates for many species while having little or no effect on observability, and turbidity might strongly affect observability in many types of aquatic surveys but have little affect on population growth rate.

Our MARSS model with covariates now looks on the surface like a regular MARSS model:

$$\begin{aligned} \mathbf{x}_t &= \mathbf{B}\mathbf{x}_{t-1} + \mathbf{u} + \mathbf{w}_t, \text{ where } \mathbf{w}_t \sim \text{MVN}(0, \mathbf{Q}) \\ \mathbf{y}_t &= \mathbf{Z}\mathbf{x}_t + \mathbf{a} + \mathbf{v}_t, \text{ where } \mathbf{v}_t \sim \text{MVN}(0, \mathbf{R}) \end{aligned} \tag{13.9}$$

with the $\mathbf{x}_t$, $\mathbf{y}_t$ and parameter matrices redefined as in Equations 13.7 and 13.8:

$$\begin{aligned}
\mathbf{x} &= \begin{bmatrix} \mathbf{x}^{(v)} \\ \mathbf{x}^{(c)} \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} \mathbf{B}^{(v)} & \mathbf{C} \\ 0 & \mathbf{B}^{(c)} \end{bmatrix} \quad \mathbf{u} = \begin{bmatrix} \mathbf{u}^{(v)} \\ \mathbf{u}^{(c)} \end{bmatrix} \quad \mathbf{Q} = \begin{bmatrix} \mathbf{Q}^{(v)} & 0 \\ 0 & \mathbf{Q}^{(c)} \end{bmatrix} \\
\mathbf{y} &= \begin{bmatrix} \mathbf{y}^{(v)} \\ \mathbf{y}^{(c)} \end{bmatrix} \quad \mathbf{Z} = \begin{bmatrix} \mathbf{Z}^{(v)} & \mathbf{D} \\ 0 & \mathbf{Z}^{(c)} \end{bmatrix} \quad \mathbf{a} = \begin{bmatrix} \mathbf{a}^{(v)} \\ \mathbf{a}^{(c)} \end{bmatrix} \quad \mathbf{R} = \begin{bmatrix} \mathbf{R}^{(v)} & 0 \\ 0 & \mathbf{R}^{(c)} \end{bmatrix}
\end{aligned} \tag{13.10}$$

Note **Q** and **R** are written as block diagonal matrices, but you could allow covariances if that made sense. **u** and **a** are column vectors here. We can fit the model (Equation 13.9) as usual using the `MARSS()` function.

The log-likelihood that is returned by `MARSS()` will include the log-likelihood of the covariates under the covariate state model. If you want only the the log-likelihood of the non-covariate data, you will need to subtract off the log-likelihood of the covariate model:

$$\begin{aligned}
\mathbf{x}_t^{(c)} &= \mathbf{B}^{(c)}\mathbf{x}_{t-1}^{(c)} + \mathbf{u}^{(c)} + \mathbf{w}_t, \text{ where } \mathbf{w}_t \sim \text{MVN}(0, \mathbf{Q}^{(c)}) \\
\mathbf{y}_t^{(c)} &= \mathbf{Z}^{(c)}\mathbf{x}_t^{(c)} + \mathbf{a}^{(c)} + \mathbf{v}_t, \text{ where } \mathbf{v}_t \sim \text{MVN}(0, \mathbf{R}^{(c)})
\end{aligned} \tag{13.11}$$

An easy way to get this log-likelihood for the covariate data only is use the augmented model (Equation 13.9 with terms defined as in Equation 13.10) but pass in missing values for the non-covariate data. The following code shows how to do this.

```
y.aug <- rbind(data, covariates)
fit.aug <- MARSS(y.aug, model = model.aug)
```

`fit.aug` is the MLE object that can be passed to `MARSSkf()`. You need to make a version of this MLE object with the non-covariate data filled with NAs so that you can compute the log-likelihood without the covariates. This needs to be done in the `marss` element since that is what is used by `MARSSkf()`. Below is code to do this.

```
fit.cov <- fit.aug
fit.cov$marss$data[1:dim(data)[1], ] <- NA
extra.LL <- MARSSkf(fit.cov)$logLik
```

Note that when you fit the augmented model, the estimates of **C** and $\mathbf{B}^{(c)}$ are affected by the non-covariate data since the model for both the non-covariate and covariate data are estimated simultaneously and are not independent (since the covariate states affect the non-covariates states). If you want the covariate model to be unaffected by the non-covariate data, you can fit the covariate model separately and use the estimates for $\mathbf{B}^{(c)}$ and $\mathbf{Q}^{(c)}$ as fixed values in your augmented model.

14

Estimation of species interaction strengths

14.1 Background

Multivariate autoregressive models (commonly termed MAR models) have been developed as a tool for analyzing community dynamics from time series data (Ives, 1995; Ives et al., 1999, 2003; Hampton et al., 2013). These models are based on a process model for log abundances ($\mathbf{x}$) of the form

$$\mathbf{x}_t = \mathbf{B}\mathbf{x}_{t-1} + \mathbf{u} + \mathbf{w}_t \text{ where } \mathbf{w}_t \sim \text{MVN}(0, \mathbf{Q}) \tag{14.1}$$

$\mathbf{B}$ is the interaction matrix; self interaction strengths (density-dependence) are on the diagonal and inter-specific interaction strengths are on the off-diagonals such that $\mathbf{B}_{i,j}$ is the 'effect' of species j on species i. This model has a stochastic equilibrium—it fluctuates around mean, $(\mathbf{I}-\mathbf{B})^{-1}\mathbf{u}$.

The term $\mathbf{u}$ determines the mean level but once the system is at equilibrium, it does not affect the fluctuations relative to the mean. To see this, compare two models with $b = 0.5$ and $u = 1$ versus $u = 0$. The mean for the first is $1/(1-0.5) = 2$ and for the second is 0. If we start both 1 above the mean, the next x is the same distance from the mean: $x_2 = 0.5(2+1)+1 = 2.5$ and $x_2 = 0.5(0+1)+0 = 0.5$. So both end up at 0.5 above the mean. So once the system is at equilibrium, it is 'scale invariant', where $\mathbf{u}$ is the scaling term. The way that Ives et al. (2003) write their process model (their Equation 10) is $\mathbf{X}_t = \mathbf{A} + \mathbf{B}\mathbf{X}_{t-1} + \mathbf{E}_t$. The $\mathbf{A}$ in Ives's equation is the $\mathbf{u}$ appearing in Equation 14.1 and the $\mathbf{E}_t$ is our $\mathbf{w}_t$.

Often the models include environmental covariates, but we will leave off covariates for the moment and address them at the end of the chapter. If we add a

---

Type `RShowDoc("Chapter_SpeciesInteractions.R",package="MARSS")` at the R command line to open a file with all the code for the examples in this chapter.

measurement process[1], we have a MARSS model:

$$\mathbf{y}_t = \mathbf{Z}\mathbf{x}_t + \mathbf{a} + \mathbf{v}_t \text{ where } \mathbf{v}_t \sim \text{MVN}(0, \mathbf{R}) \tag{14.2}$$

Typically, we have one time series per species and thus we assume that $m = n$ and $\mathbf{Z}$ is an $m \times m$ identity matrix (when $m = n$, $\mathbf{a}$ is set to 0). However, it is certainly possible to have multiple time series per species (for example data taken at multiple sites).

In this chapter, we will estimate the $\mathbf{B}$ matrix of species interactions for a simple wolf-moose system and for a four-species freshwater plankton system.

14.2 Two-species example using wolves and moose

Population dynamics of wolves and moose on Isle Royale, Michigan make an interesting case study of a two-species predator-prey interactions. These populations have been studied intensively since 1958[2]. Unlike other populations of gray wolves, the Isle Royale population has a diet dominated by one prey item, moose. The only predator of moose on Isle Royale is the gray wolf, as this population is not hunted.

We will use the wolf and moose winter census data from Isle Royale to learn how to fit community dynamics models to time-series data. The long-term January (wolf) and February (moose) population estimates are provided at `http://www.isleroyalewolf.org`.

The mathematical form of the process model for the wolf-moose population dynamics is

$$\begin{bmatrix} x_w \\ x_m \end{bmatrix}_t = \begin{bmatrix} b_{w\to w} & b_{m\to w} \\ b_{w\to m} & b_{m\to m} \end{bmatrix} \begin{bmatrix} x_w \\ x_m \end{bmatrix}_{t-1} + \begin{bmatrix} u_w \\ u_m \end{bmatrix} + \begin{bmatrix} w_w \\ w_m \end{bmatrix}_t$$
$$\begin{bmatrix} w_w \\ w_m \end{bmatrix}_t \sim \text{MVN}\left(0, \begin{bmatrix} q_w & 0 \\ 0 & q_m \end{bmatrix}\right) \tag{14.3}$$

where w denotes wolf and m denotes moose. $w \to w$ is the effect of wolf on wolf (density-dependence) and $w \to m$ is the effect of wolf on moose (predation effect on moose).

14.2.1 Load in and plot the data

We will use 1960 to 2011. We will hold out 1959 as we will need that year when we look at the effect of covariates.

```
yr1960to2011 <- isleRoyal[, "Year"] >= 1960 & isleRoyal[, "Year"] <= 2011
royale.dat <- log(t(isleRoyal[yr1960to2011, c("Wolf", "Moose")]))
```

[1] You can fit a MAR model with no observation error by setting $\mathbf{R} = 0$, but a conditional least-squares algorithm is vastly faster than EM or BFGS for the $\mathbf{R} = 0$ case (assuming no missing data).

[2] There are many publications from this long-term study site; see `http://www.isleroyalewolf.org/wolfhome/tech_pubs.html` and the review here `http://www.isleroyalewolf.org/data/data/home.html`.

```
x <- isleRoyal[, "Year"]
y <- log(isleRoyal[, c("Wolf", "Moose")])
graphics::matplot(x, y,
  ylab = "Log count", xlab = "Year", type = "l",
  lwd = 3, bty = "L", col = "black"
)
legend("topright", c("Wolf", "Moose"), lty = c(1, 2), bty = "n")
```

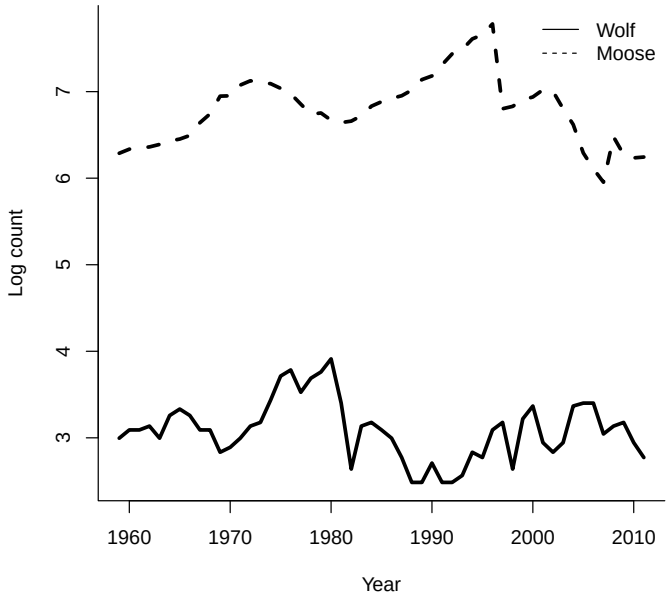

Fig. 14.1. Plot of the Isle Royale wolf and moose data.

14.2.2 Fit the model to the wolf-moose data

The naive way to fit the model is to use Equations 14.2 and 14.1 “as is":

```
royale.model.0 <- list(
  B = "unconstrained", Q = "diagonal and unequal",
  R = "diagonal and unequal", U = "unequal"
)
kem.0 <- MARSS(royale.dat, model = royale.model.0)
```

If you try this, you will notice that it does not converge but stops when it reaches `maxit` and prints a number of warnings about non-convergence. The problem is that when you try to estimate **B** and **u**, they are often confounded. This a well-known

problem, and you will need to find a way to fix **u** at some value. If you are willing to assume that the process is at equilibrium (i.e., not recovering to equilibrium from a big perturbation), then you can simply demean the data and set **u** to 0. It is also common to standardize the variance by dividing by the square root of the variance of the data. This is called z-scoring the data.

```
# if missing values are in the data, they should be NAs
z.royale.dat <- zscore(royale.dat)
```

We can fit the model to the z-scored data, but we still have convergence issues.

```
royale.model.1 <- list(
  Z = "identity", B = "unconstrained",
  Q = "diagonal and unequal", R = "diagonal and unequal",
  U = "zero", tinitx = 1
)
cntl.list <- list(allow.degen = FALSE, maxit = 200)
kem.1 <- MARSS(z.royale.dat, model = royale.model.1, control = cntl.list)
```

```
Warning! Reached maxit before parameters converged. Maxit was 200.
 neither abstol nor log-log convergence tests were passed.

MARSS fit is
Estimation method: kem
Convergence test: conv.test.slope.tol = 0.5, abstol = 0.001
WARNING: maxit reached at  200  iter before convergence.
 Neither abstol nor log-log convergence test were passed.
 The likelihood and params are not at the MLE values.
 Try setting control$maxit higher.
Log-likelihood: -75.56383
AIC: 171.1277   AICc: 173.4933

                    Estimate
R.(Wolf,Wolf)        0.001433
R.(Moose,Moose)      0.000362
B.(1,1)              0.768031
B.(2,1)             -0.178990
B.(1,2)              0.078335
B.(2,2)              0.827922
Q.(X.Wolf,X.Wolf)    0.455214
Q.(X.Moose,X.Moose)  0.178661
x0.X.Wolf            0.002926
x0.X.Moose          -1.192645
Initial states (x0) defined at t=1

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

```
Convergence warnings
 Warning: the  R.(Wolf,Wolf)  parameter value has not converged.
 Warning: the  R.(Moose,Moose)  parameter value has not converged.
 Warning: the  x0.X.Wolf  parameter value has not converged.
 Warning: the  logLik  parameter value has not converged.
 Type MARSSinfo("convergence") for more info on this warning.
```

It looks like **R** is going to zero, meaning that the maximum-likelihood model is a process error only model. That is not too surprising given that the data look more like a random walk than white noise. We will set **R** manually to zero and assume that the census is complete (they count all individuals):

```
royale.model.2 <- list(
  Z = "identity", B = "unconstrained",
  Q = "diagonal and unequal", R = "zero", U = "zero"
)
kem.2 <- MARSS(z.royale.dat, model = royale.model.2)
```

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

MARSS fit is
Estimation method: kem
Convergence test: conv.test.slope.tol = 0.5, abstol = 0.001
Estimation converged in 16 iterations.
Log-likelihood: -81.53121
AIC: 179.0624   AICc: 180.5782

                    Estimate
B.(1,1)               0.7670
B.(2,1)              -0.1788
B.(1,2)               0.0783
B.(2,2)               0.8277
Q.(X.Wolf,X.Wolf)     0.4485
Q.(X.Moose,X.Moose)   0.1758
x0.X.Wolf             0.1471
x0.X.Moose           -1.4089
Initial states (x0) defined at t=0

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

14.2.3 Look at the estimated interactions

The estimated **B** elements are `coef(kem.2)$B`.

```
wolf.B <- coef(kem.2, type = "matrix")$B
rownames(wolf.B) <- colnames(wolf.B) <- rownames(royale.dat)
print(wolf.B, digits = 2)
```

```
      Wolf Moose
Wolf   0.77 0.078
Moose -0.18 0.828
```

The `coef()` function returns the estimated parameters, but in this case we want to see the estimates in matrix form. Thus we use `type="matrix"`. Element row=*i*, col=*j* in **B** is the effect of species *j* on species *i*, so $\mathbf{B}_{2,1}$ is the effect of wolves on moose and $\mathbf{B}_{1,2}$ is the effect of moose on wolves. The **B** matrix suggests that wolves have a negative effect on moose and that moose have a positive effect on wolves—as one would expect. The diagonals are interpreted differently than the off-diagonals since the diagonals are $(b_{i,i}-1)$ so subtract off 1 from the diagonals to get the effect of species i on itself. If the species are density-independent, then $\mathbf{B}_{i,i}$ would equal 1. Smaller $\mathbf{B}_{i,i}$ means more density dependence.

14.2.4 Adding covariates

It is well-known that moose numbers are strongly affected by winter and summer climate. The Isle Royale data set provided with MARSS has climate data from climate stations in Northeastern Minnesota, near Isle Royale[3]. The covariate data include January-February, July-September and April-May average temperature and precipitation. Also included are three-year running means of these data, where the number for year t is the average of years $t-1$, t and $t+1$. We will include these covariates in the analysis to see how they change our interaction estimates. We have to adjust our covariates because the census numbers are from winter in year t and we want the climate data from the previous year to affect this winter's moose count. As usual, we will need to demean our covariate data so that we can set **u** equal to zero. We will standardize the variance also so that we can more easily compare the effects across different covariates.

The mathematical form of our new process model for the wolf-moose population dynamics is

$$\begin{bmatrix} x_w \\ x_m \end{bmatrix}_t = \mathbf{B} \begin{bmatrix} x_w \\ x_m \end{bmatrix}_{t-1} + \begin{bmatrix} 0 & 0 & 0 \\ C_{21} & C_{22} & C_{23} \end{bmatrix} \begin{bmatrix} \text{win temp} \\ \text{win precip} \\ \text{sum temp} \end{bmatrix}_{t-1} + \begin{bmatrix} w_w \\ w_m \end{bmatrix}_t \tag{14.4}$$

The C_{21}, C_{22}, etc. terms are the effect of winter temperature, winter precipitation, previous summer temperature and previous summer precipitation on winter moose numbers. Since climate is known to mainly affect the moose, we set the climate effects to 0 for wolves (top row of **C**).

[3] From the Western Regional Climate Center. See the help file for this dataset for references (`?isleRoyal`).

First we prepare the covariate data and select the winter temperature and precipitation data and the summer temperature data. We need to use the previous year's climate data with this winter's abundance data, so 1959 to 2010.

```
clim.variables <- c(
  "jan.feb.ave.temp", "jan.feb.ave.precip",
  "july.sept.ave.temp"
)
yr1959to2010 <- isleRoyal[, "Year"] >= 1959 & isleRoyal[, "Year"] <= 2010
clim.dat <- t(isleRoyal[yr1959to2010, clim.variables])
z.score.clim.dat <- zscore(clim.dat)
```

A plot of the covariate data against each other indicates that there is not much correlation between winter temperature and precipitation (Figure 14.2, which is good for analysis purposes, but warm winters are somewhat correlated with warm summers. The latter will make it harder to interpret the effect of winter versus summer temperature although the correlation is not too strong fortunately.

Next we prepare the list with the structure of all the model matrices. We give descriptive names to the **C** elements so we can remember what each **C** element means.

```
royale.model.3 <- list(
  Z = "identity", B = "unconstrained",
  Q = "diagonal and unequal", R = "zero", U = "zero",
  C = matrix(list(
    0, "Moose win temp", 0, "Moose win precip",
    0, "Moose sum temp"
  ), 2, 3),
  c = z.score.clim.dat
)
```

Then we fit the model with covariates.

```
kem.3 <- MARSS(z.royale.dat, model = royale.model.3)
```

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

MARSS fit is
Estimation method: kem
Convergence test: conv.test.slope.tol = 0.5, abstol = 0.001
Estimation converged in 16 iterations.
Log-likelihood: -80.65261
AIC: 183.3052   AICc: 186.1748

                  Estimate
B.(1,1)             0.7670
B.(2,1)            -0.1638
```

```
B.(1,2)                0.0783
B.(2,2)                0.8339
Q.(X.Wolf,X.Wolf)      0.4485
Q.(X.Moose,X.Moose)    0.1700
x0.X.Wolf              0.1504
x0.X.Moose            -1.4412
C.Moose win temp       0.0242
C.Moose win precip    -0.0718
C.Moose sum temp      -0.0307
Initial states (x0) defined at t=0

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

The results suggest what is already known about this system: cold winters and heavy snow are bad for moose as are hot summers.

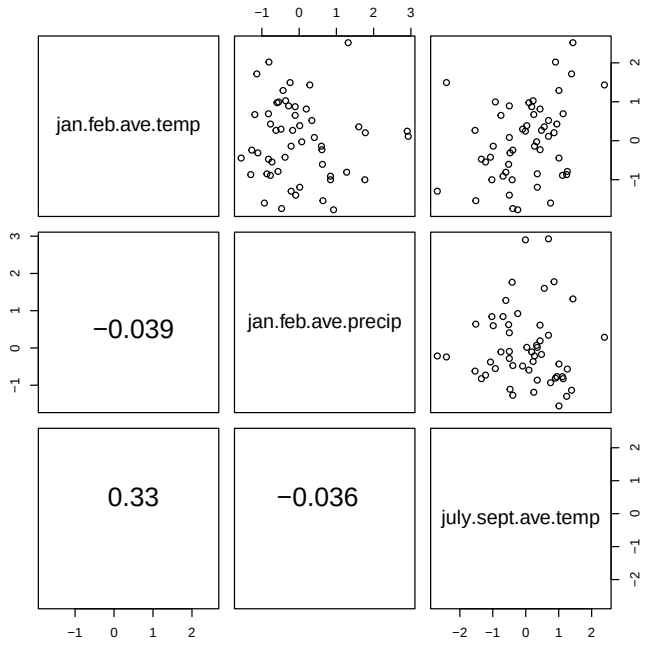

Fig. 14.2. Pairs plot of the covariate data for Isle Royale with correlations in the lower panel. The R code that produced this plot was `cor.fun=function(x, y)text(0.5,0.5,format(cor(x,y),digits=2),cex=2)` `pairs(t(z.score.clim.dat),lower.panel=cor.fun)`.

14.2.5 Change the model and data

You can explore the sensitivity of the **B** estimates when the measurement error is increased by adding white noise to the data:

```
bad.data <- z.royale.dat + matrix(rnorm(100, 0, sqrt(.2)), 2, 50)
kem.bad <- MARSS(bad.data, model = model)
```

You can change the model by changing the constraints on **R** and **Q**.

14.3 Some settings to improve performance when estimating B

In the default MARSS model, the value of $E[\mathbf{X}_0|\mathbf{y}_0]$ (what `x0` denotes when `tinitx=0` in the model list) is estimated. If we are estimating the **B** matrix, it is better to set `tinitx=1` so that we are estimating $E[\mathbf{X}_1|\mathbf{y}_0]$ instead[4]. The model will fit either way, but setting `tinitx=1` in the model list will speed up and stabilize the fitting. It does not make much of a difference for the wolf-moose dataset but can have a large effect for larger models. The reason is that the likelihood surface for $E[\mathbf{X}_1|\mathbf{y}_0]$ is better behaved when **B** is small. For example, if **B** equal to 0, there is little information about $E[\mathbf{X}_0|\mathbf{y}_0]$ so the algorithm goes in circles trying to estimate it while there is good information $E[\mathbf{X}_1|\mathbf{y}_0]$ from $\mathbf{y}_1$. We could use a prior on the initial **x** but this requires its variance-covariance structure, which depends on the unknown **B** and specifying a variance-covariance structure that conflicts with **B** will change your **B** estimates.

For the wolf-moose model, we set $\mathbf{R} = 0$. The EM algorithm (default) cannot estimate `x0` when `tinitx=1` therefore to fit the model with `tinitx=1` we need to use `method="BFGS"`. This is only for the case when $\mathbf{R} = 0$[5]

```
royale.model.4 <- list(
  B = "unconstrained", U = "zero", Q = "diagonal and unequal",
  Z = "identity", R = "zero", tinitx = 1
)
kem.4 <- MARSS(z.royale.dat, model = royale.model.4)
```

The other setting we may want to change is `allow.degen` in the control list. This sets the diagonals of **Q** or **R** to zero if they are heading towards zero. When the initial **x** is at $t = 1$, this can have non-intuitive (not wrong but puzzling; see Appendix A) consequences if **R** is going to zero. So, we will set `control$allow.degen=FALSE` and manually set **R** to 0 if needed.

[4] If there are many missing values at $t = 1$, we might still have problems and have to adjust accordingly.

[5] because the update for $(\mathbf{x}_1^0)_{j+1}$ is $\mathbf{x}_1^T$ but when $\mathbf{R} = 0$ and $\mathbf{V}_1^0 = 0$, $\mathbf{x}_1^T$ will equal $(\mathbf{x}_1^0)_j$, i.e., whatever value you started with. Thus the estimate of $\mathbf{x}_1^0$ never changes.

14.4 Analysis a four-species plankton community

Ives et al. (2003) presented weekly data on the biomass of two species of phytoplankton and two species of zooplankton in two lakes, one with low planktivory and one with high planktivory. They used these data to estimate the interaction terms for the four species. Here we will reanalyze data and compare our results.

Ives et al. (2003) explain the data as: "The data consist of weekly samples of zooplankton and phytoplankton, which for the analyses were divided into two zooplankton groups (Daphnia and non-Daphnia) and two phytoplankton groups (large and small phytoplankton). Daphnia are large, effective herbivores, and small phytoplankton are particularly vulnerable to herbivory, so we anticipate strong interactions between Daphnia and small phytoplankton groups." Figure 14.3 shows the data. What you can see from the figure is that the data are only collected in the summer.

14.4.1 Load in the plankton data

```
# only use the plankton, daphnia, & non-daphnia
plank.spp <- c("Large Phyto", "Small Phyto", "Daphnia", "Non-daphnia")
plank.dat <- ivesDataByWeek[, plank.spp]
# The data are not logged
plank.dat <- log(plank.dat)
# Transpose to get time going across the columns
plank.dat <- t(plank.dat)
# make a demeaned version
d.plank.dat <- (plank.dat - apply(plank.dat, 1, mean, na.rm = TRUE))
```

We will demean the data so we can set **u** to 0. We do not standardize by the variance, however because we are going to fix the **R** variance later as Ives et al. did.

14.4.2 Specify a MARSS model for the plankton data

We will start by fitting a model with the following assumptions:

- All phytoplankton share the same process variance.
- All zooplankton share the same process variance.
- Phytoplankton and zooplankton have different measurement variances
- Measurement errors are independent.
- Process errors are independent.

```
Q <- matrix(list(0), 4, 4)
diag(Q) <- c("Phyto", "Phyto", "Zoo", "Zoo")
R <- matrix(list(0), 4, 4)
diag(R) <- c("Phyto", "Phyto", "Zoo", "Zoo")
plank.model.0 <- list(
  B = "unconstrained", U = "zero", Q = Q,
  Z = "identity", A = "zero", R = R,
```

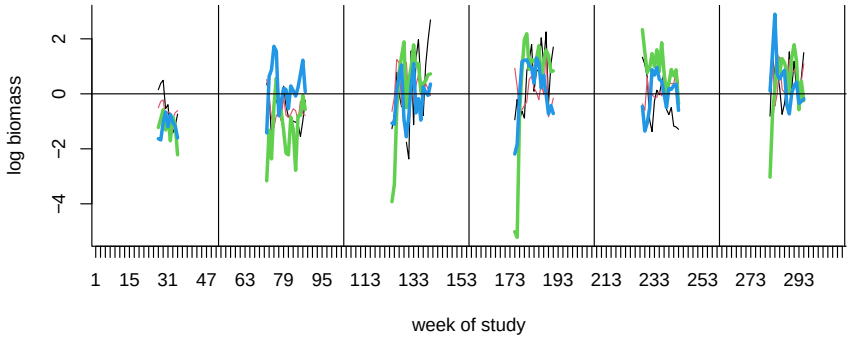

Fig. 14.3. Plot of the demeaned plankton data. Zooplankton are the thicker lines. Phytoplankton are the thinner lines.

```
    x0 = "unequal", tinitx = 1
)
```

Why did we set `U="zero"`? Equation 14.1 is a stationary model; it fluctuates about a mean. The **u** in Equation 14.1 is a scaling term that just affects the mean level—once the system is at equilibrium. If we assume that the mean of **y** (the mean of our data) is a good estimate of the mean of the system (the **x**), then we can set **u** equal to zero (and **a**). The initial states (**x**) are set at $t = 1$ instead of $t = 0$, which improves estimation for large systems.

14.4.3 Fit the plankton model and look at the estimated B matrix

The call to fit the model is:

```
kem.plank.0 <- MARSS(d.plank.dat, model = plank.model.0)
```

Now we can print the **B** matrix, with a little clean up.

```
# Cleaning up the B matrix for printing
B.0 <- coef(kem.plank.0, type = "matrix")$B[1:4, 1:4]
rownames(B.0) <- colnames(B.0) <- c("LP", "SP", "D", "ND")
print(B.0, digits = 2)
```

```
      LP   SP       D     ND
LP  0.77 0.29 -0.0182  0.131
SP  0.19 0.51  0.0052 -0.045
D  -0.43 2.29  0.4916  0.389
ND -0.33 1.35 -0.2180  0.831
```

LP stands for large phytoplankton, SP for small phytoplankton, D for Daphnia and ND for non-Daphnia.

We can compare this to the Ives et al. estimates (in their Table 2, bottom right) and see quite a few differences:

```
     LP    SP     D    ND
LP 0.48 -0.39    --    --
SP   --  0.25 -0.17 -0.11
D    --    --  0.74  0.00
ND   --  0.10  0.00  0.60
```

First, you will notice is that the Ives et al. matrix is missing values. The matrix they show is after a model selection step to determine which interactions had little data support and thus could be set to zero. Also, they fixed the interactions between Daphnia and non-Daphnia at zero because they do not prey on each other. The second thing you will notice is that the estimates are not particularly similar. Next we will try some other ways of fitting the data that are closer to the way that Ives et al. fitted the data.

By the way, if you are curious what would happen if we removed all those NAs, you can run the following code.

```
test.dat <- d.plank.dat[, !is.na(d.plank.dat[1, ])]
test <- MARSS(test.dat, model = plank.model.0)
```

Removing all the NAs would mean that the end of summer 1 is connected to the beginning of summer 2. This adds some steep steps in the Daphnia time series where Daphnia ended the summer high and started the next summer low.

14.4.4 Look at different ways to fit the model

We will try a series of changes to get closer to the way Ives et al. fit the data, and you will see how different assumptions change (or do not change) our species interaction estimates.

First, we change **Q** to be unconstrained. Making **Q** diagonal in model 0 meant that we were assuming that whatever environmental factor is driving variation in phytoplankton numbers is uncorrelated with the environmental factor driving zooplankton variation. That is probably not true since they are all in the same lake. This case takes awhile to run.

```
plank.model.1 <- plank.model.0
plank.model.1$Q <- "unconstrained"
kem.plank.1 <- MARSS(d.plank.dat, model = plank.model.1)
```

Notice that the **Q** specification changed to "unconstrained". Everything else stays the same as in model 0. The code now runs longer, and the **B** estimates are not particularly closer to Ives et al.

```
        LP    SP      D    ND
LP  0.4961 0.061  0.079 0.123
SP -0.1833 0.896  0.067 0.011
D   0.1180 0.350  0.638 0.370
ND  0.0023 0.370 -0.122 0.810
```

Next, we will set some of the interactions to zero as in Table 2 in Ives et al. (2003). In their table, certain interactions were fixed at 0 (denoted with 0s), and some were made 0 after fitting (the blanks). We will fix all to zero. To do this, we need to write out the **B** matrix as a list matrix so that we can have estimated and fixed values (the 0s) in the **B** specification.

```
B.2 <- matrix(list(0), 4, 4) # set up the list matrix
diag(B.2) <- c("B11", "B22", "B33", "B44") # give names to diagonals
# and names to the estimated non-diagonals
B.2[1, 2] <- "B12"
B.2[2, 3] <- "B23"
B.2[2, 4] <- "B24"
B.2[4, 2] <- "B42"
print(B.2)
```

```
     [,1]  [,2]  [,3]  [,4]
[1,] "B11" "B12" 0     0
[2,] 0     "B22" "B23" "B24"
[3,] 0     0     "B33" 0
[4,] 0     "B42" 0     "B44"
```

As you can see, the **B** matrix now has elements that will be estimated (the names in quotes) and fixed values (the numbers with no quotes). When preparing your list matrix, make sure your fixed values do not have quotes around them. If they do, they are strings (class character) not numbers (class numeric), and `MARSS()` will interpret a string as the name of something to be estimated. If you use the same name for an element, then `MARSS()` will force those elements to be shared (have the same value).

```
# model 2
plank.model.2 <- plank.model.1
plank.model.2$B <- B.2
kem.plank.2 <- MARSS(d.plank.dat, model = plank.model.2)
```

Now we are getting closer to the Ives et al. estimates:

```
     LP    SP      D     ND
LP 0.65 -0.33     --     --
SP   --  0.54 0.0016 -0.026
D    --    -- 0.8349     --
ND   --  0.13     --  0.596
```

Ives et al. did not estimate **R**. Instead they used a fixed observation variance of 0.04 for phytoplankton and 0.16 for zooplankton[6]. We fit the model with their fixed **R** as follows:

```
# model 3
plank.model.3 <- plank.model.2
plank.model.3$R <- diag(c(.04, .04, .16, .16))
kem.plank.3 <- MARSS(d.plank.dat, model = plank.model.3)
```

As you can see from Table 14.1, we are getting closer to the Ives et al. estimates, but we are still a bit off. Now we need to add the environmental covariates: phosphorous and fish biomass.

14.4.5 Adding covariates

A standard way that you will see covariate data added to a MARSS model is the following:

$$\begin{aligned} \mathbf{x}_t &= \mathbf{B}\mathbf{x}_{t-1} + \mathbf{u} + \mathbf{C}\mathbf{c}_t + \mathbf{w}_t, \text{ where } \mathbf{w}_t \sim \text{MVN}(0, \mathbf{Q}) \\ \mathbf{y}_t &= \mathbf{Z}\mathbf{x}_t + \mathbf{a} + \mathbf{D}\mathbf{d}_t + \mathbf{v}_t, \text{ where } \mathbf{v}_t \sim \text{MVN}(0, \mathbf{R}) \end{aligned} \tag{14.5}$$

$\mathbf{c}_t$ and $\mathbf{d}_t$ are covariate data, like temperature. At time t and $\mathbf{C}$ is a matrix with the (linear) effects of $\mathbf{c}_t$ on $\mathbf{x}_t$, and $\mathbf{D}$ is a matrix with the (linear) effects of $\mathbf{d}_t$ on $\mathbf{y}_t$.

Ives et al. (2003) only include covariates in their process model, and their process model (their Equation 27) is written $\mathbf{X}_t = \mathbf{A} + \mathbf{B}\mathbf{X}_{t-1} + \mathbf{C}\mathbf{U}_t + \mathbf{E}_t$. In our Equation 14.5, $\mathbf{U}_t = \mathbf{c}_t$, and $\mathbf{C}$ is a $m \times p$ matrix, where p is the number of covariates in $\mathbf{c}_t$. We will set their $\mathbf{A}$ (our $\mathbf{u}$) to zero by demeaning the $\mathbf{y}$ and implicitly assuming that the mean of the $\mathbf{y}$ is a good estimate of the mean of the $\mathbf{x}$'s. Thus the model where covariates only affect the underlying process is

$$\begin{aligned} \mathbf{x}_t &= \mathbf{B}\mathbf{x}_{t-1} + \mathbf{C}\mathbf{c}_t + \mathbf{w}_t, \text{ where } \mathbf{w}_t \sim \text{MVN}(0, \mathbf{Q}) \\ \mathbf{y}_t &= \mathbf{x}_t + \mathbf{v}_t, \text{ where } \mathbf{v}_t \sim \text{MVN}(0, \mathbf{R}) \end{aligned} \tag{14.6}$$

To fit this model, we first need to prepare the covariate data. We will just use the phosphorous data.

```
# transpose to make time go across columns
# drop=FALSE so that R doesn't change our matrix to a vector
phos <- t(log(ivesDataByWeek[, "Phosph", drop = FALSE]))
d.phos <- (phos - apply(phos, 1, mean, na.rm = TRUE))
```

Why log the covariate data? It is what Ives et al. did, so we follow their method. However, in general, you want to think about what relationship you want to assume between the covariates and their effects. For example, log (or square-root) transformations mean that extremes have less impact relative to their untransformed value

[6] You can compare this to the estimated observation variances by looking at `coef(kem.plank.2)$R`

and that a small absolute change, say from 0.01 to 0.0001 in the untransformed value, can mean a large difference in the effect since $\log(0.0001) < \log(0.01)$.

Phosphorous is assumed to only affect phytoplankton so the other terms in **C**, corresponding to the zooplankton, are set to 0. The **C** matrix is defined as follows:

$$\mathbf{C} = \begin{bmatrix} C_{LP,phos} \\ C_{SP,phos} \\ 0 \\ 0 \end{bmatrix} \tag{14.7}$$

To add **C** and **c** to our latest model, we add **C** and **c** to the model list used in the `MARSS()` call:

```
plank.model.4 <- plank.model.3
plank.model.4$C <- matrix(list("C11", "C21", 0, 0), 4, 1)
plank.model.4$c <- d.phos
```

Then we fit the model as usual:

```
kem.plank.4 <- MARSS(d.plank.dat, model = plank.model.4)
```

Here is the **C** matrix. The **C** terms for the zooplankton are shown as `--` since they are not applicable (have been set to 0). Temperature and phosphorous have an estimated positive effect on phytoplankton:

```
   [,1]
LP 0.14
SP 0.16
D    --
ND   --
```

14.4.6 Including a covariate observation model

The difficulty with the standard approach to including covariates (Equation 14.5) is that it limits what kind of covariate data you can use and how you model that covariate data. You have to assume that your covariate data has no error, which is probably not true. Assuming that your covariate has no error reduces the reported uncertainty in your covariate effect because you did not include uncertainty in those values. The standard approach also does not allow missing values in your covariate data, which is why we did not include the fish covariate data in the last model. Also you cannot combine multiple instrument time series; for example, if you have two temperature recorders with different error rates and biases. Perhaps you have one noisy temperature recorder in the first part of your time series and then you switch to a much better recorder in the second half of your time series. All these problems require pre-analysis massaging of the covariate data, leaving out noisy and gappy covariate data, and making what can feel like arbitrary choices about which covariate time series to include. This is especially worrisome when the covariates are then incorporated into the model as if they are known without error.

Instead one can include an observation and process model for the covariates just like for the non-covariate data. Now the covariates are included in $\mathbf{y}_t$ and are modeled with their own state process(es) in $\mathbf{x}_t$. A MARSS model with a covariate observation and process model is shown below. The elements with superscript (v) are for the variates and those with superscript (c) are for the covariates. The superscripts just help us keep straight which of the state processes and parameters correspond to abundances and which correspond to the environmental covariates.

$$\begin{bmatrix}\mathbf{x}^{(v)}\\ \mathbf{x}^{(c)}\end{bmatrix}_t = \begin{bmatrix}\mathbf{B}^{(v)} & \mathbf{C}\\ 0 & \mathbf{B}^{(c)}\end{bmatrix}\begin{bmatrix}\mathbf{x}^{(v)}\\ \mathbf{x}^{(c)}\end{bmatrix}_{t-1} + \begin{bmatrix}\mathbf{u}^{(v)}\\ \mathbf{u}^{(c)}\end{bmatrix} + \mathbf{w}_t,\ \mathbf{w}_t \sim \text{MVN}\left(0, \begin{bmatrix}\mathbf{Q}^{(v)} & 0\\ 0 & \mathbf{Q}^{(c)}\end{bmatrix}\right)$$

$$\begin{bmatrix}\mathbf{y}^{(v)}\\ \mathbf{y}^{(c)}\end{bmatrix}_t = \begin{bmatrix}\mathbf{Z}^{(v)} & 0\\ 0 & \mathbf{Z}^{(c)}\end{bmatrix}\begin{bmatrix}\mathbf{x}^{(v)}\\ \mathbf{x}^{(c)}\end{bmatrix}_t + \begin{bmatrix}\mathbf{a}^{(v)}\\ \mathbf{a}^{(c)}\end{bmatrix} + \mathbf{v}_t,\ \mathbf{v}_t \sim \text{MVN}\left(0, \begin{bmatrix}\mathbf{R}^{(v)} & 0\\ 0 & \mathbf{R}^{(c)}\end{bmatrix}\right) \tag{14.8}$$

Note that when you fit your covariate and non-covariate data jointly as in Equation 14.8, your non-covariate data affect the estimates of the covariate models. When you maximize the likelihood, you do so conditioned on all the data. The likelihood that is output is the likelihood of the non-covariate and covariate data. Depending on your system, you might not want the covariate model affected by the non-covariate data. In this case, you can fit the covariate model separately:

$$\begin{aligned}\mathbf{x}_t^{(c)} &= \mathbf{B}^{(c)}\mathbf{x}_{t-1}^{(c)} + \mathbf{u}^{(c)} + \mathbf{w}_t,\ \mathbf{w}_t \sim \text{MVN}(0, \mathbf{Q}^{(c)})\\ \mathbf{y}_t^{(c)} &= \mathbf{Z}^{(c)}\mathbf{x}_t^{(c)} + \mathbf{a}^{(c)},\ \mathbf{v}_t \sim \text{MVN}(0, \mathbf{R}^{(c)})\end{aligned} \tag{14.9}$$

At this point, you have another choice. Do you want the estimated covariates states, the $\mathbf{x}^{(c)}$, to be affected by the non-covariate data? For example, you have temperature data. You can estimates true temperature for the temperature only from the temperature data or you can decide that the non-covariate data has information about the true temperature, because the non-covariate states are affected by the true temperature. If you want the covariate states to only be affected by the covariate data, then use Equation 14.5 with $\mathbf{c}_t$ set to your estimates of $\mathbf{x}^{(c)}$ from Equation 14.9. Or if you want the non-covariate data to affect the estimates of the covariate states, use Equation 14.8 with the parameters estimated from Equation 14.9.

14.4.7 The MARSS model with covariates following Ives et al.

Ives et al. used Equation 14.5 for phosphorous and Equation 14.8 for fish biomass. Phosphorous was treated as observed with no error since it was experimentally manipulated and there were no missing values. Fish biomass was treated as having observation error and was modeled as a autoregressive process with unknown parameters as in Equation 14.8.

Their MARSS model takes the form:

$$\begin{aligned}\mathbf{x}_t &= \mathbf{B}\mathbf{x}_{t-1} + \mathbf{C}\mathbf{c}_t + \mathbf{w}_t,\ \text{where } \mathbf{w}_t \sim \text{MVN}(0, \mathbf{Q})\\ \mathbf{y}_t &= \mathbf{x}_t + \mathbf{v}_t,\ \text{where } \mathbf{v}_t \sim \text{MVN}(0, \mathbf{R})\end{aligned} \tag{14.10}$$

where **x** and **y** are redefined as

$$\begin{bmatrix} \text{large phyto} \\ \text{small phyto} \\ \text{Daphnia} \\ \text{non-Daphnia zooplank} \\ \text{fish biomass} \end{bmatrix} \tag{14.11}$$

The covariate fish biomass appears in **x** because it will be modeled, and its interaction terms (Ives et al.'s **C** terms) appear in **B**. Phosphorous appears in the $\mathbf{c}_t$ terms because it is treated as a known additive term and its interaction terms appear in **C**. Recall that we set **u** to 0 by demeaning the plankton data, so it does not appear above. The **Z** matrix does not appear in front of the $\mathbf{x}_t$ since there is a one-to-one correspondence the **x**'s and **y**'s, and thus **Z** is the identity matrix.

The **B** matrix is

$$\mathbf{B} = \begin{bmatrix} \mathbf{B}^{(v)} & \mathbf{C} \\ 0 & \mathbf{B}^{(c)} \end{bmatrix} = \begin{bmatrix} b_{LP} & b_{LP,SP} & 0 & 0 & 0 \\ 0 & b_{SP} & b_{SP,D} & b_{SP,ND} & 0 \\ 0 & 0 & b_D & 0 & C_{D,fish} \\ 0 & b_{ND,SP} & 0 & b_{ND,ND} & C_{ND,fish} \\ 0 & 0 & 0 & 0 & b_{fish} \end{bmatrix} \tag{14.12}$$

The **B** elements have some interactions fixed at 0 as in our last model fit. The c's are the interactions between the fish and the species. We will estimate a **B** term for fish since Ives et al. did, but this is an odd thing to do for the fish data since these data were interpolated from two samples per season.

The **Q** matrix is the same as that in our last model fit, with the addition of an element for the variance for the fish biomass:

$$\mathbf{Q} = \begin{bmatrix} \mathbf{Q}^{(v)} & 0 \\ 0 & \mathbf{Q}^{(c)} \end{bmatrix} = \begin{bmatrix} q_{LP} & q_{LP,SP} & q_{LP,D} & q_{LP,ND} & 0 \\ q_{LP,SP} & q_{SP} & q_{SP,D} & q_{SP,ND} & 0 \\ q_{LP,D} & q_{SP,D} & q_D & q_{D,ND} & 0 \\ q_{LP,ND} & q_{SP,ND} & q_{D,ND} & q_{ND} & 0 \\ 0 & 0 & 0 & 0 & q_{fish} \end{bmatrix} \tag{14.13}$$

Again it is odd to estimate a variance term for data interpolated from two points, but we follow Ives et al. here.

Ives et al. set the observation variance for the logged fish biomass data to 0.36 (page 320 in Ives et al. (2003)). The observation variances for the plankton data was set as in our previous model.

$$\mathbf{R} = \begin{bmatrix} 0.04 & 0 & 0 & 0 & 0 \\ 0 & 0.04 & 0 & 0 & 0 \\ 0 & 0 & 0.16 & 0 & 0 \\ 0 & 0 & 0 & 0.16 & 0 \\ 0 & 0 & 0 & 0 & 0.36 \end{bmatrix} \tag{14.14}$$

14.4.8 Setting the model structure for the model with fish covariate data

First we need to add the logged fish biomass to our data matrix.

```
# transpose to make time go across columns
# drop=FALSE so that R doesn't change our matrix to a vector
fish <- t(log(ivesDataByWeek[, "Fish biomass", drop = FALSE]))
d.fish <- (fish - apply(fish, 1, mean, na.rm = TRUE))
# plank.dat.w.fish = rbind(plank.dat,fish)
d.plank.dat.w.fish <- rbind(d.plank.dat, d.fish)
```

Next make the **B** matrix. Some elements are estimated and others are fixed at 0.

```
B <- matrix(list(0), 5, 5)
diag(B) <- list("B11", "B22", "B33", "B44", "Bfish")
B[1, 2] <- "B12"
B[2, 3] <- "B23"
B[2, 4] <- "B24"
B[4, 2] <- "B42"
B[1:4, 5] <- list(0, 0, "C32", "C42")
print(B)
```

```
     [,1]  [,2]  [,3]  [,4]  [,5]
[1,] "B11" "B12" 0     0     0
[2,] 0     "B22" "B23" "B24" 0
[3,] 0     0     "B33" 0     "C32"
[4,] 0     "B42" 0     "B44" "C42"
[5,] 0     0     0     0     "Bfish"
```

Now we have a **B** matrix that looks like that in Equation 14.12.

We need to add an extra row to **C** for the fish biomass row in **x**:

```
C <- matrix(list("C11", "C21", 0, 0, 0), 5, 1)
```

Then we set up the **R** matrix.

```
R <- matrix(list(0), 5, 5)
diag(R) <- list(0.04, 0.04, 0.16, 0.16, 0.36)
```

Last, we need to set up the **Q** matrix:

```
Q <- matrix(list(0), 5, 5)
Q[1:4, 1:4] <- paste(rep(1:4, times = 4), rep(1:4, each = 4), sep = "")
Q[5, 5] <- "fish"
Q[lower.tri(Q)] <- t(Q)[lower.tri(Q)]
print(Q)
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,] "11" "12" "13" "14" 0
[2,] "12" "22" "23" "24" 0
```

```
[3,] "13" "23" "33" "34" 0
[4,] "14" "24" "34" "44" 0
[5,] 0    0    0    0    "fish"
```

14.4.9 Fit the model with covariates

The model is the same as the previous model with updated process parameters and updated **R**. We will pass in the updated data matrix with the fish biomass added:

```
plank.model.5 <- plank.model.4
plank.model.5$B <- B
plank.model.5$C <- C
plank.model.5$Q <- Q
plank.model.5$R <- R
kem.plank.5 <- MARSS(d.plank.dat.w.fish, model = plank.model.5)
```

This is the new **B** matrix using covariates.

```
     LP     SP      D     ND
LP 0.61 -0.465     --     --
SP   --  0.333 -0.019 -0.048
D    --     --  0.896     --
ND   --  0.044     --  0.675
```

Now we are getting are getting close to Ives et al.'s estimates. Compare model 5 in Table 14.1 to the first column.

Table 14.1. The parameter estimates under the different plankton models. Models 0 to 3 do not include covariates, so the C elements are blank. Bij is the effect of species *i* on species *j*. 1=large phytoplankton, 2=small phytoplankton, 3=Daphnia, 4=non-Daphnia zooplankton. The Ives et al. (2003) estimates are from their table 2 for the low planktivory lake with the observation model.

	Ives et al.	Model 0	Model 1	Model 2	Model 3	Model 4	Model 5
B11	0.48	0.77	0.50	0.65	0.62	0.61	0.61
B22	0.25	0.51	0.90	0.54	0.51	0.33	0.33
B33	0.74	0.49	0.64	0.83	0.89	0.89	0.90
B44	0.60	0.83	0.81	0.60	0.67	0.66	0.67
B12	-0.39	0.29	0.06	-0.33	-0.32	-0.46	-0.46
B23	-0.17	0.01	0.07	0.00	-0.02	-0.02	-0.02
B24	-0.11	-0.04	0.01	-0.03	0.02	-0.05	-0.05
B42	0.10	1.35	0.37	0.13	0.09	0.05	0.04
C11	0.25					0.14	0.14
C21	0.25					0.16	0.16
C32	-0.14						-0.04
C42	-0.04						-0.01

NOTE! When you include your covariates in your state model (the **x** part), the reported log-likelihood is for the variate plus the covariate data. If you want just the

log-likelihood for the variates, then you should replace the covariate data with NAs and re-run the Kalman filter with your estimated model:

```
tmp <- kem.plank.5
tmp$marss$data[5, ] <- NA
LL.variates <- MARSSkf(tmp)$logLik
```

`MARSSkf()` is the Kalman filter function and it needs a fitted model as output by a `MARSS()` call. We set up a temporary fitted model, `tmp`, equal to our fitted model and then set the covariate data in that to NAs. Note we need to do this for the marssMODEL object used by `MARSSkf()`, which will be in `MLEobj$marss`. We then pass that temporary fitted model to `MARSSkf()` to get the log-likelihood of just the variates.

14.4.10 Discussion

The estimates for our last model are fairly close to the Ives et al. estimates, but still a bit different. There are two big differences between our last model and the Ives et al. analysis. Ives et al. had data from three lakes and the estimate of $\mathbf{Q}$ used the data from all lakes.

Combining data, whether it be from different areas or years, can be done in a MARSS model as follows. Let $\mathbf{y}_1$ be the first data set (say from site 1) and $\mathbf{y}_2$ be the second data set (say from site 2). Then a MARSS model with shared parameters values across datasets would be

$$\begin{aligned} \mathbf{x}_t^+ &= \mathbf{B}^+\mathbf{x}_{t-1}^+ + \mathbf{u}^+\mathbf{w}_t, \text{ where } \mathbf{w}_t \sim \text{MVN}(0, \mathbf{Q}^+) \\ \mathbf{y}_t^+ &= \mathbf{Z}^+\mathbf{x}_t^+ + \mathbf{a}^+ + \mathbf{v}_t, \text{ where } \mathbf{v}_t \sim \text{MVN}(0, \mathbf{R}^+) \end{aligned} \tag{14.15}$$

where the $+$ matrices are stacked matrices from the different sites (1 and 2):

$$\begin{aligned} \begin{bmatrix} \mathbf{x}_{1,t} \\ \mathbf{x}_{2,t} \end{bmatrix} &= \begin{bmatrix} \mathbf{B} & 0 \\ 0 & \mathbf{B} \end{bmatrix} \begin{bmatrix} \mathbf{x}_{1,t-1} \\ \mathbf{x}_{2,t-1} \end{bmatrix} + \begin{bmatrix} \mathbf{u} \\ \mathbf{u} \end{bmatrix} + \mathbf{w}_t, \ \mathbf{w}_t \sim \text{MVN}\left(0, \begin{bmatrix} \mathbf{Q} & \mathbf{q} \\ \mathbf{q} & \mathbf{Q} \end{bmatrix}\right) \\ \begin{bmatrix} \mathbf{y}_{1,t} \\ \mathbf{y}_{2,t} \end{bmatrix} &= \begin{bmatrix} \mathbf{Z} & 0 \\ 0 & \mathbf{Z} \end{bmatrix} \begin{bmatrix} \mathbf{x}_{1,t} \\ \mathbf{x}_{2,t} \end{bmatrix} + \begin{bmatrix} \mathbf{a} \\ \mathbf{a} \end{bmatrix} + \mathbf{v}_t, \ \mathbf{v}_t \sim \text{MVN}\left(0, \begin{bmatrix} \mathbf{R} & 0 \\ 0 & \mathbf{R} \end{bmatrix}\right) \end{aligned} \tag{14.16}$$

The $\mathbf{q}$ in the process variance allows that the environmental variability might be correlated between datasets, e.g., if they are replicate plots that are nearby. If you did not want all the parameters shared, then you replace the $\mathbf{B}$ in $\mathbf{B}^+$ with $\mathbf{B}_1$ and $\mathbf{B}_2$, say.

The second big difference is that Ives et al. did not demean their data, but estimated $\mathbf{u}$. We could have done that too, but with all the NAs in the data (during winter), estimating $\mathbf{u}$ is not robust and takes a long time. You can try the analysis on the data that has not been demeaned and set `U="unequal"`. The results are not particularly different, but it takes a long, long,...long time to converge.

You can also try using the actual fish data instead of the interpolated data. Fish biomass was estimated at the end and start of the season, so only the values at the

start and finish of strings of fish numbers are the real data. The others are interpolated. You can fill in those interpolated values with NAs (missing values) and rerun. The results are not appreciably different, but the effect of fish drops a bit as you might expect when you have less fish information. You don't see it here, but your estimated confidence in the fish effects would also drop since this estimate is based on less fish data.

14.5 Stability metrics from estimated interaction matrices

The previous sections focused on estimation of the **B** and **C** matrices. The estimated **B** matrix gives a picture of the species interactions, but it also can be used to compute metrics of the intrinsic community stability (Ives et al., 2003). Here we illustrate how to compute these metrics; the reader should see Ives et al. (2003) for details on the meaning of each.

For the examples here, we will use the estimated **B** and **Q** matrices from our model 5:

```
B <- coef(kem.plank.5, type = "matrix")$B[1:4, 1:4]
Q <- coef(kem.plank.5, type = "matrix")$Q[1:4, 1:4]
```

14.5.1 Return rate metrics

Return rate metrics measure how rapidly the system returns to the stationary distribution of species abundances after it is perturbed away from the stationary distribution. With a deterministic ($\mathbf{Q} = 0$) MARSS community model, the equilibrium is a point or stable limit cycle. In a stochastic model ($\mathbf{Q} \neq 0$), the equilibrium is stochastic and is a stationary distribution. Rate of return to the stochastic equilibrium is the rate at which the distribution converges to the stationary distribution after a perturbation away from this stationary distribution. The more rapid the convergence, the more stable the system.

The rate of return of the mean of the stationary distribution is governed by the dominant eigenvalue of **B**. We can compute this as:

```
max(eigen(B)$values)
```

```
[1] 0.8964988
```

The rate of return of the variance of the stationary distribution is governed by the dominant eigenvalue of $\mathbf{B} \otimes \mathbf{B}$:

```
max(eigen(kronecker(B, B))$values)
```

```
[1] 0.8037101
```

14.5.2 Variance metrics

These metrics measure the variance of the stationary distribution of species abundances (with variance due to environmental drivers removed) relative to the process error variance. The system is considered more stable when the stationary distribution variance is low relative to the process error variance.

To compute variance metrics, we need to first compute the variance-covariance matrix for the stationary distribution, $\mathbf{V}_\infty$:

```
m <- nrow(B)
vecV <- solve(diag(m * m) - kronecker(B, B)) %*% as.vector(Q)
V_inf <- matrix(vecV, nrow = m, ncol = m)
```

A measure of the proportion of the "volume" of the stationary distribution due to species interactions is given by the square of the determinant of the **B** matrix (Eqn. 24 in Ives et al. (2003)):

```
abs(det(B))^2
```

```
[1] 0.01559078
```

To compare stability across systems of different sizes, you scale by the number of species:

```
abs(det(B))^(2 / nrow(B))
```

```
[1] 0.3533596
```

14.5.3 Reactivity metrics

Reactivity measure how the system responds to a perturbation. A highly reactive system tends to move farther away from a stable equilibrium immediately after a perturbation, even though the system will eventually return to the equilibrium. High reactivity occurs when species interactions greatly amplify the environmental variance to produce a stationary distribution with high variances in the abundance of individual species.

Both metrics of reactivity of estimates of the average expected change in distance from the mean of the stationary distribution. The first uses estimates of **Q** and $\mathbf{V}_\infty$.

```
-sum(diag(Q)) / sum(diag(V_inf))
```

```
[1] -0.346845
```

Estimation of **Q** is prone to high uncertainty. Another metric that uses only **B** is the worst-case reactivity. This is given by

```
max(eigen(t(B) %*% B)$values) - 1
```

```
[1] -0.1957795
```

14.6 Further information

MAR modeling and models have been used estimate species interaction strengths, stability metrics, and environmental drivers for a variety of freshwater plankton systems (Ives, 1995; Ives et al., 1999, 2003; Hampton et al., 2008, 2006; Hampton and Schindler, 2006; Klug and Cottingham, 2001). They have been used to gain much insight into the dynamics of ecological communities and how environmental drivers affect the system. See Hampton et al. (2013) for a review of the literature using MAR models to understand plankton dynamics.

15

Combining data from multiple time series

15.1 Overview

In this section, we consider the case where multiple time series exist and we want to use all the datasets to estimate a common underlying state process or common underlying parameters. An example where this arises in ecological applications is when 1) there are time series of observations from the same population or location (e.g., aerial and land based surveys of the same site) or 2) there are time series collected in the same survey, but represent observations of multiple underlying state processes (e.g., multiple species or populations or age groups). An example of the latter is data from trawl surveys where multiple species of fish are collected in one trawl.

Why should we consider using other time series? In the first scenario, where methodology differs between time series, observation error may be survey-specific. We would like to use both time series but need to account for the different observation error variance. In the second scenario, we are observing multiple different state processes, but because the survey methodology is the same, it might be reasonable to assume a shared observation error variance. If whatever we are surveying has similar responses to environmental stochasticity, it might be possible to also assume a shared process variance across the state processes.

In both of the above examples, MARSS models offer a way to link multiple time series. If parameters are allowed to be shared among the state processes (trend parameters, process variances) or observation processes (observation variances), parameter estimates will be more precise than if we treated each time series as independent. By improving estimates of variance parameters, we will also be better able to discriminate between process and observation error variances.

Type `RShowDoc("Chapter_CombiningTrendData.R",package="MARSS")` at the R command line to open a file with all the code for the examples in this chapter.

In this chapter, we will show examples of using MARSS models to analyze data on poplations of multiple species but where there are multiple observation time series that come from different survey methods. The state process is written as:

$$\mathbf{x}_t = \mathbf{B}\mathbf{x}_{t-1} + \mathbf{u} + \mathbf{w}_t \text{ where } \mathbf{w}_t \sim \text{MVN}(0, \mathbf{Q}) \tag{15.1}$$

The true population sizes at time t are represented by the state $\mathbf{x}_t$, whose dimensions are equal to the number of state processes (m). The $m \times m$ matrix $\mathbf{B}$ allows interaction between processes (density dependence and competition, for instance), $\mathbf{u}$ is a vector describing the mean trend, and the correlation of the process deviations is determined by the structure of the matrix $\mathbf{Q}$.

The multivariate observation error model is expressed as,

$$\mathbf{y}_t = \mathbf{Z}\mathbf{x}_t + \mathbf{a} + \mathbf{v}_t \text{ where } \mathbf{v}_t \sim \text{MVN}(0, \mathbf{R}) \tag{15.2}$$

where $\mathbf{y}_t$ is a vector of observations at time t, $\mathbf{Z}$ is a design matrix of 0s and 1s, $\mathbf{a}$ is a vector of bias adjustments, and the correlation structure of observation matrices is specified with the matrix $\mathbf{R}$. Including $\mathbf{Z}$ and $\mathbf{a}$ is required when some of the states processes are observed with multiple observation time series.

15.2 Salmon spawner surveys

In our first application, we will analyze a dataset on Chinook salmon (*Oncorhynchus tshawytscha*). This dataset comes from the Okanogan River in Washington state, a major tributary of the Columbia River (with headwaters in British Columbia). As an index of the abundance of spawning adults, biologists have conducted redd surveys during summer months (redds are nests or collection of rocks on stream bottoms where females deposit eggs). Our data are aerial surveys of redds on the Okanogan River conducted 1956-2008 and ground surveys of redds from 1990-2008.

15.2.1 Read in and plot the raw data

We will be using the aerial and ground surveys and logging the counts.

```
head(okanaganRedds)
```

```
     Year aerial ground
[1,] 1956     37     NA
[2,] 1957     53     NA
[3,] 1958     94     NA
[4,] 1959     50     NA
[5,] 1960     29     NA
[6,] 1961     NA     NA
```

```
logRedds <- log(t(okanaganRedds)[c("aerial", "ground"), ])
```

Notice that the ground surveys did not start until 1990.

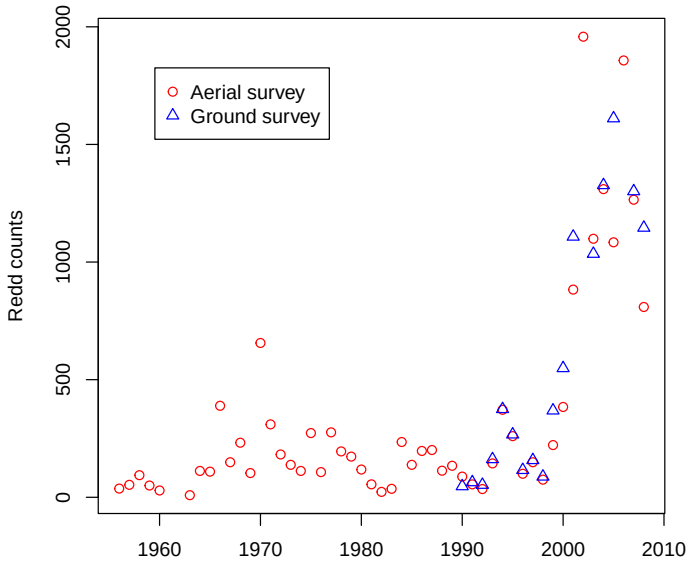

Fig. 15.1. The two time series look to be pretty close to one another in the years where there is overlap.

15.2.2 Test hypotheses about whether the data can be combined

Do these surveys represent observations of the same underlying process? We can evaluate data support for this question by testing a few relatively simple models. Using the logged data, we will start with a simple model that assumes the underlying population process is univariate (there is one underlying population trajectory) and each survey is an independent observation of this population process. Mathematically, the model is:

$$x_t = x_{t-1} + u + w_t, \text{ where } w_t \sim \text{N}(0, q)$$
$$\begin{bmatrix} y_{aer} \\ y_{gnd} \end{bmatrix}_t = \begin{bmatrix} 1 \\ 1 \end{bmatrix} x_t + \begin{bmatrix} 0 \\ a_2 \end{bmatrix} + \begin{bmatrix} v_{aer} \\ v_{gnd} \end{bmatrix}_t, \text{ where } \mathbf{v}_t \sim \text{MVN}\left((0, \begin{bmatrix} r & 0 \\ 0 & r \end{bmatrix}\right) \tag{15.3}$$

The **a** structure means that the a for one of the y's is fixed at 0 and the other a is estimated relative to that fixed a. In MARSS, this is the "scaling" structure for **a**. We specify this model as follows. Since **x** is univariate, **Q** and **u** are just scalars (single numbers), and we can leave them off in our specification.

Fit the single state model, where the two surveys are assumed to be observing the same population.

```
model1 <- list()
model1$R <- "diagonal and equal"
model1$Z <- matrix(1, 2, 1)
model1$A <- "scaling"
kem1 <- MARSS(logRedds, model = model1)
```

The AIC and AICc values for this model are

We can modify the above model to let the observation error variances to be unique:

```
model2 <- model1 # model2 is based on model1
model2$R <- "diagonal and unequal"
kem2 <- MARSS(logRedds, model = model2)
```

It is possible that these surveys are measuring different population processes. They are not done at exactly the same days or locations. For our third model, we will fit a model with two different population process with the same process parameters. For simplicity, we will keep the trend and variance parameters the same. Mathematically, the model we are fitting is:

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}_t = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}_{t-1} + \begin{bmatrix} u \\ u \end{bmatrix} + \mathbf{w}_t, \text{ where } \mathbf{w}_t \sim \text{MVN}\left(0, \begin{bmatrix} q & 0 \\ 0 & q \end{bmatrix}\right)$$
$$\begin{bmatrix} y_{aer} \\ y_{gnd} \end{bmatrix}_t = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}_t + \begin{bmatrix} 0 \\ 0 \end{bmatrix} + \begin{bmatrix} v_{aer} \\ v_{gnd} \end{bmatrix}_t, \text{ where } \mathbf{v}_t \sim \text{MVN}\left(0, \begin{bmatrix} r & 0 \\ 0 & r \end{bmatrix}\right) \tag{15.4}$$

We specify this in MARSS as

```
model3 <- list()
model3$Q <- "diagonal and equal"
model3$R <- "diagonal and equal"
model3$U <- "equal"
model3$Z <- "identity"
model3$A <- "zero"
kem3 <- MARSS(logRedds, model = model3)
```

Based on AICc, it appears that the best model is also the simplest one, with one state vector (model1).

```
c(mod1 = kem1$AICc, mod2 = kem2$AICc, mod3 = kem3$AICc)
```

```
    mod1     mod2     mod3
133.9804 136.2164 174.1392
```

This suggests that the two different surveys are not only measuring the same underlying process, but have the same observation error variance. On the surface, similar observation error variances might seem impossible but it may be that stream turbidity is what drives observation error variance for both types of surveys. Finally, we will make a plot of the model-predicted states (with +/- 2 s.e.s) and the log-transformed data (Figure 15.2).

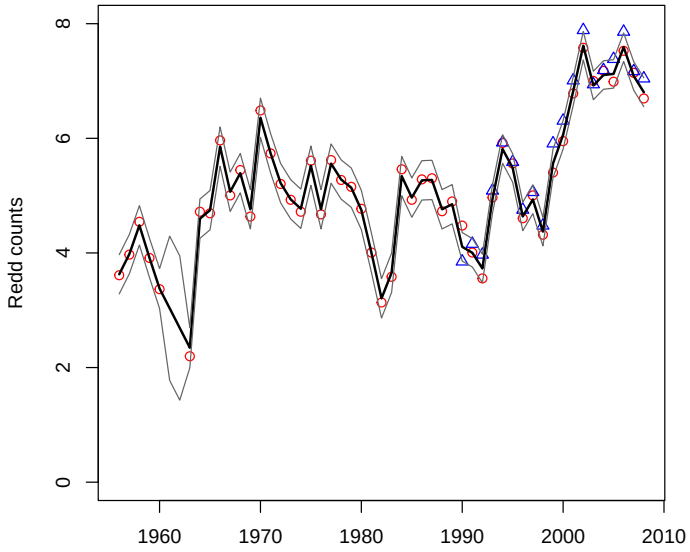

Fig. 15.2. The data support the hypothesis that the two redd-count time series are observations of the same population. The points are the data and the thick black line is the estimated underlying state.

15.3 American kestrel abundance indices

In this example, we evaluate uncertainty in the structure of process variability (environmental stochasticity) using breeding bird surveys data. In this analysis, we use three time series of American kestrel (*Falco sparverius*) abundance from adjacent Canadian provinces along a longitudinal gradient (British Columbia, Alberta, Saskatchewan). The data were collected annually and corrected for changes in observer coverage and detectability.

15.3.1 The data

Figure 15.3 shows the data. The data are already log transformed.

```
birddat <- t(kestrel[, c("British.Columbia", "Alberta", "Saskatchewan")])
head(kestrel)
```

```
     Year British.Columbia Alberta Saskatchewan
[1,] 1969            0.754   0.460        0.000
```

```
[2,] 1970           0.673   0.899       0.192
[3,] 1971           0.734   1.133       0.280
[4,] 1972           0.589   0.528       0.386
[5,] 1973           1.405   0.789       0.451
[6,] 1974           0.624   0.528       0.234
```

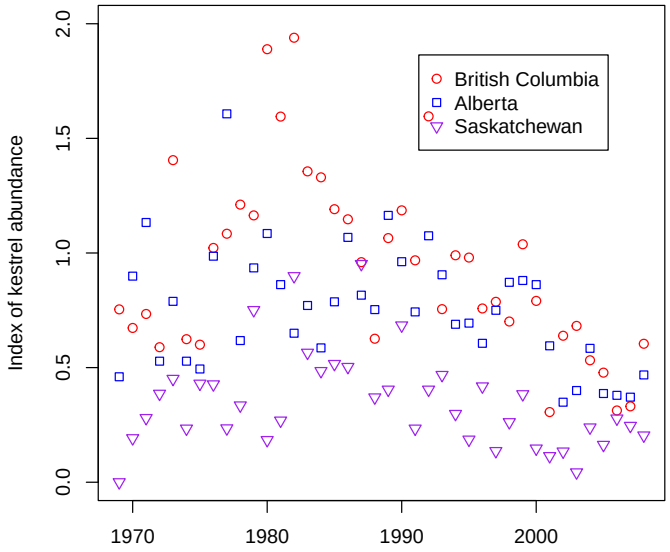

Fig. 15.3. The kestrel data.

We know that the surveys use the same design, so we will force observation error to be shared. Our uncertainty lies in whether these time series are sampling the same population, and how environmental stochasticity varies by subpopulation (if there are subpopulations). Our first model has one population trajectory (meaning there is one panmictic BC/AB/SK population), and each of these three surveys is an observation of this single population with equal observation variances. Mathematically, the model is:

$$x_t = x_{t-1} + u + w_t, \text{ where } w_t \sim \text{N}(0, q)$$
$$\begin{bmatrix} y_{BC} \\ y_{AB} \\ y_{SK} \end{bmatrix}_t = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} x_t + \begin{bmatrix} 0 \\ a_2 \\ a_3 \end{bmatrix} + \begin{bmatrix} v_{BC} \\ v_{AB} \\ v_{SK} \end{bmatrix}_t, \text{ where } \mathbf{v}_t \sim \text{MVN}\left(0, \begin{bmatrix} r & 0 & 0 \\ 0 & r & 0 \\ 0 & 0 & r \end{bmatrix}\right) \quad (15.5)$$

In MARSS, we denote the model:

```
model.b1=list()
model.b1$R="diagonal and equal"
model.b1$Z=matrix(1,3,1)
kem.b1 = MARSS(birddat, model=model.b1, control=list(minit=100) )
```

We do not need to specify the structure of $\mathbf{Q}$ and $\mathbf{u}$ since they are scalar and have no structure.

We will compare this to a model where we assume that there is a separate population for British Columbia, Alberta, and Saskatchewan but they have the same process parameters (trend and process variance). Mathematically, this model is:

$$\begin{bmatrix} x_{BC} \\ x_{AB} \\ x_{SK} \end{bmatrix}_t = \begin{bmatrix} x_{BC} \\ x_{AB} \\ x_{SK} \end{bmatrix}_{t-1} + \begin{bmatrix} u \\ u \\ u \end{bmatrix} + \mathbf{w}_t, \text{ where } \mathbf{w}_t \sim \text{MVN}\left(0, \begin{bmatrix} q & 0 & 0 \\ 0 & q & 0 \\ 0 & 0 & q \end{bmatrix}\right)$$
$$\begin{bmatrix} y_{BC} \\ y_{AB} \\ y_{SK} \end{bmatrix}_t = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{BC} \\ x_{AB} \\ x_{SK} \end{bmatrix}_t + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} v_{BC} \\ v_{AB} \\ v_{SK} \end{bmatrix}_t, \text{ where } \mathbf{v}_t \sim \text{MVN}\left(0, \begin{bmatrix} r & 0 & 0 \\ 0 & r & 0 \\ 0 & 0 & r \end{bmatrix}\right) \tag{15.6}$$

This is specified as:

```
model.b2 <- list()
model.b2$Q <- "diagonal and equal"
model.b2$R <- "diagonal and equal"
model.b2$Z <- "identity"
model.b2$A <- "zero"
model.b2$U <- "equal"
kem.b2 <- MARSS(birddat, model = model.b2)
```

Because these populations are surveyed over a relatively large geographic area, it is reasonable to expect that environmental variation may differ between provinces. For our third model, we will fit a model with separate processes that are allowed to have unequal process parameters.

```
model.b3 <- model.b2 # is is based on model.b2
# all we change is the structure of Q
model.b3$Q <- "diagonal and unequal"
model.b3$U <- "unequal"
kem.b3 <- MARSS(birddat, model = model.b3)
```

For our last model, we will consider a model where the Alberta and Saskatchewan surveys are observing the same population. Mathematically, this model is:

$$\begin{bmatrix} x_{BC} \\ x_{AB-SK} \end{bmatrix}_t = \begin{bmatrix} x_{BC} \\ x_{AB-SK} \end{bmatrix}_{t-1} + \begin{bmatrix} u \\ u \end{bmatrix} + \mathbf{w}_t, \text{ where } \mathbf{w}_t \sim \text{MVN}\left(0, \begin{bmatrix} q & 0 \\ 0 & q \end{bmatrix}\right)$$
$$\begin{bmatrix} y_{BC} \\ y_{AB} \\ y_{SK} \end{bmatrix}_t = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_{BC} \\ x_{AB-SK} \end{bmatrix}_t + \begin{bmatrix} 0 \\ 0 \\ a_3 \end{bmatrix} + \begin{bmatrix} v_{BC} \\ v_{AB} \\ v_{SK} \end{bmatrix}_t, \text{ where } \mathbf{v}_t \sim \text{MVN}\left(0, \begin{bmatrix} r & 0 & 0 \\ 0 & r & 0 \\ 0 & 0 & r \end{bmatrix}\right) \tag{15.7}$$

This model is specified as

```
model.b4 <- list()
model.b4$Q <- "diagonal and unequal"
model.b4$R <- "diagonal and equal"
model.b4$Z <- factor(c("BC", "AB-SK", "AB-SK"))
model.b4$A <- "scaling"
model.b4$U <- "unequal"
kem.b4 <- MARSS(birddat, model = model.b4)
```

The AICc values for the four models are

```
c(mod1 = kem.b1$AICc, mod2 = kem.b2$AICc,
  mod3 = kem.b3$AICc, mod4 = kem.b4$AICc)
```

```
   mod1     mod2     mod3     mod4
20.90670 22.96714 23.75125 14.76889
```

The last model is superior to the others based on AICc. Figure 15.4 shows the fits for this model.

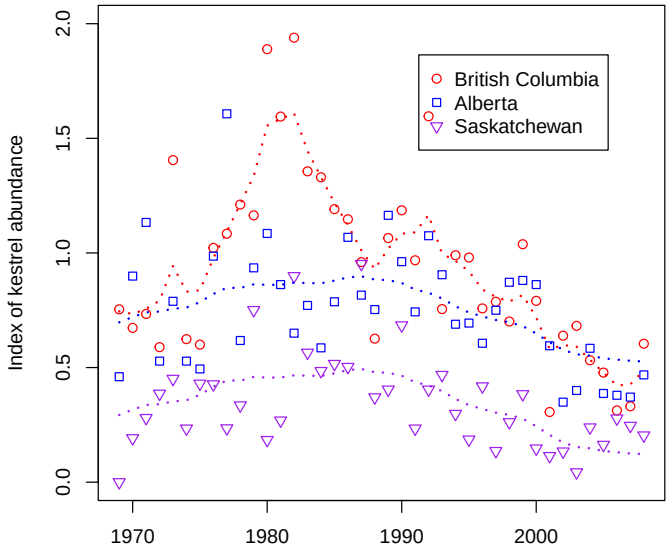

Fig. 15.4. Plot model 4 fits to the kestrel data.

16

Univariate dynamic linear models (DLMs)

16.1 Overview of dynamic linear models

In this chapter, we will use MARSS to analyze dynamic linear models (DLMs), wherein the parameters in a regression model are treated as time-varying. DLMs are used commonly in econometrics, but have received less attention in the ecological literature (c.f. Lamon III et al., 1998; Scheuerell and Williams, 2005). Our treatment of DLMs is rather cursory—we direct the reader to excellent textbooks by Pole et al. (1994) and Petris et al. (2009) for more in-depth treatments of DLMs. The former focuses on Bayesian estimation whereas the latter addresses both likelihood-based and Bayesian estimation methods.

We begin our description of DLMs with a static regression model, wherein the i-th observation is a linear function of an intercept, predictor variable(s), and a random error term. For example, if we had one predictor variable (F), we could write the model as

$$y_i = \alpha + \beta F_i + v_i, \tag{16.1}$$

where the α is the intercept, β is the regression slope, F_i is the predictor variable matched to the i^{th} observation (y_i), and $v_i \sim \mathrm{N}(0, r)$. It is important to note here that there is no implicit ordering of the index i. That is, we could shuffle any/all of the (y_i, F_i) pairs in our dataset with no effect on our ability to estimate the model parameters. We can write the model in Equation 16.1 using vector notation, such that

$$\begin{aligned} y_i &= \begin{bmatrix} 1 & F_i \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} + v_i \\ &= \mathbf{F}_i^\top \theta + v_i, \end{aligned} \tag{16.2}$$

Type `RShowDoc("Chapter_UnivariateDLM.R",package="MARSS")` at the R command line to open a file with all the code for the examples in this chapter.

and $\mathbf{F}_i^\top = \begin{bmatrix}\alpha \\ \beta\end{bmatrix}$ and $\theta = \begin{bmatrix}\alpha\ \beta\end{bmatrix}^\top$.

In a DLM, however, the regression parameters are dynamic in that they evolve over time. For a single observation at time t, we can write

$$y_t = \mathbf{F}_t^\top \theta_t + v_t, \tag{16.3}$$

where $\mathbf{F}_t$ is a column vector of regression variables at time t, θ_t is a column vector of regression parameters at time t and $v_t \sim \mathrm{N}(0, r)$. While seemingly identical, this formulation presents two features that distinguish it from Equation 16.2. First, the observed data are explicitly time ordered (i.e., $\mathbf{y} = \{y_1, y_2, y_3, ..., y_T\}$), which means we expect them to contain implicit information. Second, the relationship between the observed datum and the predictor variables are unique at every time t (i.e., $\theta = \{\theta_1, \theta_2, \theta_3, ..., \theta_T\}$).

However, closer examination of Equation 16.3 reveals an apparent complication for parameter estimation. With only one datum at each time step t, we could, at best, estimate only one regression parameter, and even then, the 1:1 correspondence between data and parameters would preclude any estimation of parameter uncertainty. To address this shortcoming, we return to the time ordering of model parameters. Rather than assume the regression parameters are independent from one time step to another, we instead model them as an autoregressive process where

$$\theta_t = \mathbf{G}_t \theta_{t-1} + \mathbf{w}_t, \tag{16.4}$$

$\mathbf{G}_t$ is the parameter "evolution" matrix, and $\mathbf{w}_t$ is a vector of process errors, such that $\mathbf{w}_t \sim \mathrm{MVN}(\mathbf{0}, \mathbf{Q})$. The elements of $\mathbf{G}_t$ may be known and fixed *a priori*, or unknown and estimated from the data. Although we allow for $\mathbf{G}_t$ to be time-varying, we will typically assume that it is time invariant.

The idea is that the evolution matrix $\mathbf{G}_t$ deterministically maps the parameter space from one time step to the next, so the parameters at time t are temporally related to those before and after. However, the process is stochastic and the mapping includes stochastic error, which leads to a degradation of information over time. If the diagonal elements of $\mathbf{Q}$ are relatively large, then the parameters can vary widely from t to $t+1$. If $\mathbf{Q} = \mathbf{0}$, then $\theta_1 = \theta_2 = \theta_T$ and we are back to the static model in Equation 16.1.

16.2 Example of a univariate DLM

Let's consider an example from the literature. Scheuerell and Williams (2005) used a DLM to examine the relationship between marine survival of Chinook salmon and an index of ocean upwelling strength along the west coast of the USA. Upwelling brings cool, nutrient-rich waters from the deep ocean to shallower coastal areas. Scheuerell and Williams hypothesized that stronger upwelling in April should create better growing conditions for phytoplankton, which would then translate into more zooplankton. In turn, juvenile salmon ("smolts") entering the ocean in May and

June should find better foraging opportunities. Thus, for smolts entering the ocean in year t,

$$survival_t = \alpha_t + \beta_t F_t + v_t \text{ with } v_t \sim \text{N}(0, r), \tag{16.5}$$

and F_t is the coastal upwelling index[1] for the month of April in year t.

Both the intercept and slope are time varying, so

$$\alpha_t = \alpha_{t-1} + w_t^{(1)} \text{ with } w_t^{(1)} \sim \text{N}(0, q_1); \text{ and} \tag{16.6}$$

$$\beta_t = \beta_{t-1} + w_t^{(2)} \text{ with } w_t^{(2)} \sim \text{N}(0, q_2). \tag{16.7}$$

If we define $\theta_t = \begin{bmatrix}\alpha_t & \beta_t\end{bmatrix}^\top$, $\mathbf{G}_t = \mathbf{I}$ for all t, $\mathbf{w}_t = \begin{bmatrix}w_t^{(1)} & w_t^{(2)}\end{bmatrix}^\top$, and $\mathbf{Q} = \begin{bmatrix}q_1 & 0\\ 0 & q_2\end{bmatrix}$, we get Equation 16.4. If we define $y_t = survival_t$ and $\mathbf{F}_t = \begin{bmatrix}1 & F_t\end{bmatrix}^\top$, we can write out the full univariate DLM as a state-space model with the following form:

$$\begin{aligned} \theta_t &= \mathbf{G}_t\theta_{t-1} + \mathbf{w}_t \text{ with } \mathbf{w}_t \sim \text{MVN}(\mathbf{0}, \mathbf{Q});\\ y_t &= \mathbf{F}_t^\top\theta_t + v_t \text{ with } v_t \sim \text{N}(0, r);\\ \theta_0 &\sim \text{MVN}(\pi_0, \Lambda_0). \end{aligned} \tag{16.8}$$

Equation 16.8 is equivalent to our standard MARSS model:

$$\begin{aligned} \mathbf{x}_t &= \mathbf{B}_t\mathbf{x}_{t-1} + \mathbf{u}_t + \mathbf{C}_t\mathbf{c}_t + \mathbf{w}_t \text{ with } \mathbf{w}_t \sim \text{MVN}(0, \mathbf{Q}_t);\\ \mathbf{y}_t &= \mathbf{Z}_t\mathbf{x}_t + \mathbf{a}_t + \mathbf{D}_t\mathbf{d}_t + \mathbf{v}_t \text{ with } \mathbf{v}_t \sim \text{MVN}(0, \mathbf{R}_t);\\ \mathbf{x}_0 &\sim \text{MVN}(\pi, \Lambda); \end{aligned} \tag{16.9}$$

where $\mathbf{x}_t = \theta_t$, $\mathbf{B}_t = \mathbf{G}_t$, $\mathbf{u}_t = \mathbf{C}_t = \mathbf{c}_t = 0$, $\mathbf{y}_t = y_t$ (i.e., $\mathbf{y}_t$ is 1 x 1), $\mathbf{Z}_t = \mathbf{F}_t^\top$, $\mathbf{a}_t = \mathbf{D}_t = \mathbf{d}_t = 0$, and $\mathbf{R}_t = r$ (i.e., $\mathbf{R}_t$ is 1 x 1).

16.2.1 Fitting a univariate DLM with the {MARSS} package

Now let's go ahead and analyze the DLM specified in Equations 16.5–16.8. We begin by getting the data set, which has three columns for 1) the year the salmon smolts migrated to the ocean (*year*), 2) logit-transformed survival[2] (*logit.s*), and 3) the coastal upwelling index for April (*CUI.apr*). There are 42 years of data (1964–2005).

```
data(SalmonSurvCUI)
years <- SalmonSurvCUI[, 1]
TT <- length(years)
# response data: logit(survival)
dat <- matrix(SalmonSurvCUI[, 2], nrow = 1)
```

[1] cubic meters of seawater per second per 100 m of coastline

[2] Survival in the original context was defined as the proportion of juveniles that survive to adulthood. Thus, we use the logit function, defined as $logit(p) = log_e(p/[1-p])$, to map survival from the open interval (0,1) onto the interval $(-\infty, \infty)$, which allows us to meet our assumption of normally distributed observation errors.

As we have seen in other chapters, standardizing our covariate(s) to have zero-mean and unit-variance can be helpful in model fitting and interpretation. In this case, it is a good idea because the variance of *CUI.apr* is orders of magnitude greater than *survival*.

```
CUI <- SalmonSurvCUI[, "CUI.apr"]
CUI.z <- zscore(CUI)
# number of state = # of regression params (slope(s) + intercept)
m <- 1 + 1
```

Plots of logit-transformed survival and the z-scored April upwelling index are shown in Figure 16.1.

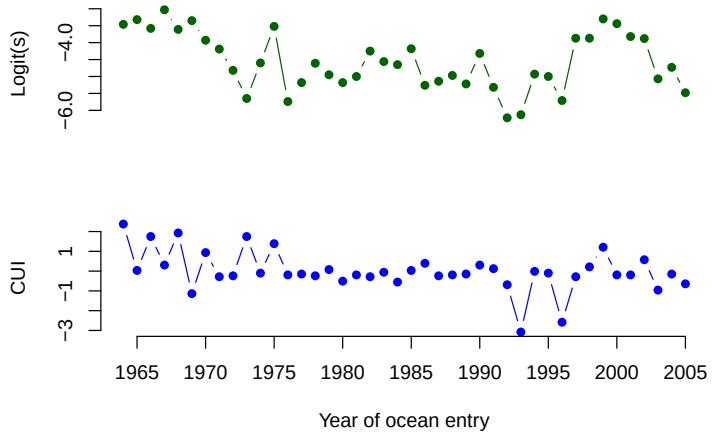

Fig. 16.1. Time series of logit-transformed marine survival estimates for Snake River spring/summer Chinook salmon (top) and z-scores of the coastal upwelling index at 45N 125W (bottom). The *x*-axis indicates the year that the salmon smolts entered the ocean.

Next, we need to set up the appropriate matrices and vectors for the `MARSS()` function. Let's begin with those for the process equation because they are straightforward.

```
# for process eqn
B <- diag(m) # 2x2; Identity
U <- matrix(0, nrow = m, ncol = 1) # 2x1; both elements = 0
Q <- matrix(list(0), m, m) # 2x2; all 0 for now
diag(Q) <- c("q1", "q2") # 2x2; diag = (q1,q2)
```

Defining the correct form for the observation model is a little more tricky, however, because of how we model the effect(s) of explanatory variables. In a DLM, we need to use $\mathbf{Z}_t$ (instead of $\mathbf{d}_t$) as the matrix of known regressors (covariates or drivers)

that affect $\mathbf{y}_t$, and $\mathbf{x}_t$ (instead of $\mathbf{D}_t$) as the regression parameters. Therefore, we need to set $\mathbf{Z}_t$ equal to an $n \times m \times T$ array, where n is the number of response variables (= 1; y_t is univariate), m is the number of regression parameters (= intercept + slope = 2), and T is the length of the time series (= 42).

```
# for observation eqn
Z <- array(NA, c(1, m, TT)) # NxMxT; empty for now
Z[1, 1, ] <- rep(1, TT) # Nx1; 1's for intercept
Z[1, 2, ] <- CUI.z # Nx1; regr variable
A <- matrix(0) # 1x1; scalar = 0
R <- matrix("r") # 1x1; scalar = r
```

Lastly, we need to define our lists of initial starting values and model matrices/vectors.

```
# only need starting values for regr parameters
inits.list <- list(x0 = matrix(c(0, 0), nrow = m))
# list of model matrices & vectors
mod.list <- list(B = B, U = U, Q = Q, Z = Z, A = A, R = R)
```

And now we can fit our DLM with the `MARSS()` function.

```
dlm1 <- MARSS(dat, inits = inits.list, model = mod.list)
```

```
Success! abstol and log-log tests passed at 115 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 115 iterations.
Log-likelihood: -40.03813
AIC: 90.07627   AICc: 91.74293

      Estimate
R.r    0.15708
Q.q1   0.11264
Q.q2   0.00564
x0.X1 -3.34023
x0.X2 -0.05388
Initial states (x0) defined at t=0

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

Notice that the `MARSS()` output does not list any estimates of the regression parameters themselves. Why not? Remember that in a DLM the states ($\mathbf{x}$) are the estimates of the regression parameters ($\boldsymbol{\theta}$). Therefore, we need to look in `dlm1$states`

for the MLEs of the regression parameters, and in `dlm1$states.se` for their standard errors.

Time series of the estimated intercept and slope are shown in Figure 16.2. It appears as though the intercept is much more dynamic than the slope, as indicated by a much larger estimate of process variance for the former (`Q.q1`). In fact, although the effect of April upwelling appears to be increasing over time, it doesn't really become important as an explanatory variable until about 1990 when the approximate 95% confidence interval for the slope no longer overlaps zero.

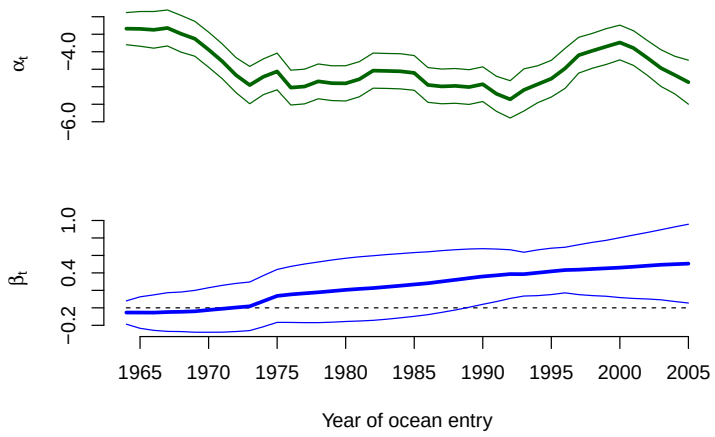

Fig. 16.2. Time series of estimated mean states (thick lines) for the intercept (top) and slope (bottom) parameters from the univariate DLM specified by Equations 16.5–16.8. Thin lines denote the mean $\pm$ 2 standard deviations.

16.3 Forecasting with a univariate DLM

Scheuerell and Williams (2005) were interested in how well upwelling could be used to forecast expected survival of salmon. Let's look at how well our model does in that context. To do so, we need the predictive distributions for the regression parameters and observation.

Beginning with our definition for the distribution of the parameters at time $t = 0$, $\theta_0 \sim \text{MVN}(\pi_0, \Lambda_0)$ in Equation 16.8, we write

$$\theta_{t-1} | y_1^{t-1} \sim \text{MVN}(\pi_{t-1}, \Lambda_{t-1}) \tag{16.10}$$

to indicate the distribution of θ at time $t-1$ conditioned on the observed data through time $t-1$ (i.e., y_1^{t-1}). Then, we can write the one-step ahead predictive distribution for θ_t given y_1^{t-1} as

$$\begin{aligned} \theta_t | y_1^{t-1} &\sim \text{MVN}(\eta_t, \Phi_t), \text{ where} \\ \eta_t &= \mathbf{G}_t \pi_{t-1}, \text{ and} \\ \Phi_t &= \mathbf{G}_t \Lambda_{t-1} \mathbf{G}_t^\top + \mathbf{Q}. \end{aligned} \tag{16.11}$$

Consequently, the one-step ahead predictive distribution for the observation at time t given y_1^{t-1} is

$$\begin{aligned} y_t | y_1^{t-1} &\sim \text{N}(\zeta_t, \Psi_t), \text{ where} \\ \zeta_t &= \mathbf{F}_t \eta_t, \text{ and} \\ \Psi_t &= \mathbf{F}_t \Phi_t \mathbf{F}_t^\top + \mathbf{R}. \end{aligned} \tag{16.12}$$

16.3.1 Forecasting a univariate DLM with the {MARSS} package

Working from Equation 16.12, we can now use the {MARSS} package to compute the expected value of the forecast at time t ($\text{E}[y_t | y_1^{t-1}] = \zeta_t$), and its variance ($\text{var}[y_t | y_1^{t-1}] = \Psi_t$). For the expectation, we need $\mathbf{F}_t \eta_t$. Recall that $\mathbf{F}_t$ is our $1 \times m$ matrix of explanatory variables at time t ($\mathbf{F}_t$ is called $\mathbf{Z}_t$ in {MARSS} notation). The one-step ahead forecasts of the parameters at time t (η_t) are calculated as part of the Kalman filter algorithm—they are termed x_t^{t-1} in {MARSS} notation and stored as `xtt1` in the list produced by the `MARSSkf()` function.

```
# get list of Kalman filter output
kf.out <- MARSSkfss(dlm1)
# forecasts of regr parameters; 2xT matrix
eta <- kf.out$xtt1
# ts of E(forecasts)
fore.mean <- vector()
for (t in 1:TT) {
  fore.mean[t] <- Z[, , t] %*% eta[, t, drop = F]
}
```

For the variance of the forecasts, we need $\mathbf{F}_t \Phi_t \mathbf{F}_t^\top + \mathbf{R}$. As with the mean, $\mathbf{F}_t \equiv \mathbf{Z}_t$. The variances of the one-step ahead forecasts of the parameters at time t (Φ_t) are also calculated as part of the Kalman filter algorithm—they are stored as `Vtt1` in the list produced by the `MARSSkf()` function. Lastly, the observation variance $\mathbf{R}$ is part of the standard MARSS output.

```
# variance of regr parameters; 1x2xT array
Phi <- kf.out$Vtt1
# obs variance; 1x1 matrix
R.est <- coef(dlm1, type = "matrix")$R
# ts of Var(forecasts)
fore.var <- vector()
for (t in 1:TT) {
  tZ <- matrix(Z[, , t], m, 1) # transpose of Z
```

```
  fore.var[t] <- Z[, , t] %*% Phi[, , t] %*% tZ + R.est
}
```

Plots of the model mean forecasts with their estimated uncertainty are shown in Figure 16.3. Nearly all of the observed values fell within the approximate prediction interval. Notice that we have a forecasted value for the first year of the time series (1964), which may seem at odds with our notion of forecasting at time t based on data available only through time $t-1$. In this case, however, `MARSS()` is estimated the states at $t=0$ (θ_0), which allows us to compute a forecast for the first time point.

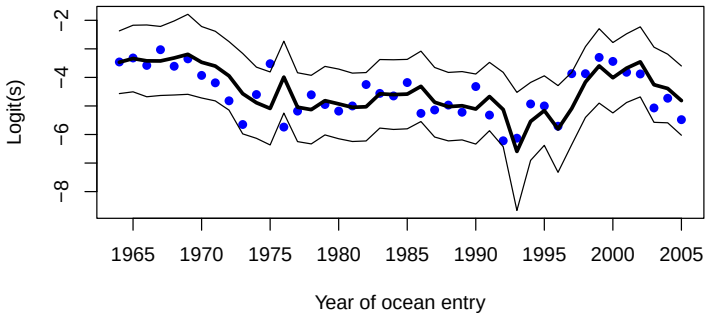

Fig. 16.3. Time series of logit-transformed survival data (blue dots) and model mean one-step ahead forecasts (thick line). Thin lines denote the approximate 95% prediction intervals.

Although our model forecasts look reasonable in logit-space, it is worthwhile to examine how well they look when the survival data and forecasts are back-transformed onto the interval [0,1] (Figure 16.4). In that case, the accuracy does not seem to be affected, but the precision appears much worse, especially during the early and late portions of the time series when survival is changing rapidly.

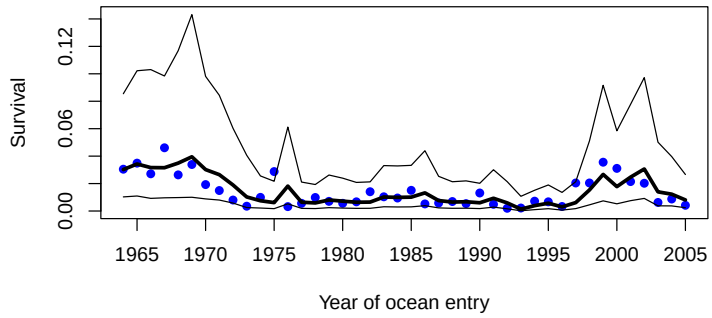

Fig. 16.4. Time series of survival data (blue dots) and model mean forecasts (thick line). Thin lines denote the approximate 95% prediction intervals.

16.3.2 DLM forecast diagnostics

As with other time series models, evaluation of a DLM should include model diagnostics. In a forecasting context, we are often interested in the forecast errors, which are simply the observed data minus the forecasts ($e_t = y_t - \zeta_t$). In particular, the following assumptions should hold true for e_t:

1. $e_t \sim \mathrm{N}(0, \sigma^2)$;
2. $\mathrm{cov}(e_t, e_{t-k}) = 0$.

In the literature on state-space models, the set of e_t are commonly referred to as "innovations". The innovations as part of the Kalman filter algorithm—they are stored as `Innov` in the list produced by the `MARSSkfss()` function[3].

```
# forecast errors
innov <- kf.out$Innov
```

Let's see if our innovations meet the model assumptions. Beginning with (1), we can use a Q-Q plot to see whether the innovations are normally distributed with a mean of zero. We will use the `qqnorm()` function to plot the quantiles of the innovations on the y-axis versus the theoretical quantiles from a Normal distribution on the x-axis. If the two distributions are similar, the points should fall on the line defined by $y = x$.

```
# Q-Q plot of innovations
qqnorm(t(innov), main = "", pch = 16, col = "blue")
# add y=x line for easier interpretation
qqline(t(innov))
```

[3] We need to use the Shumway and Stoffer Kalman filter instead of the {KFAS} Kalman filter.

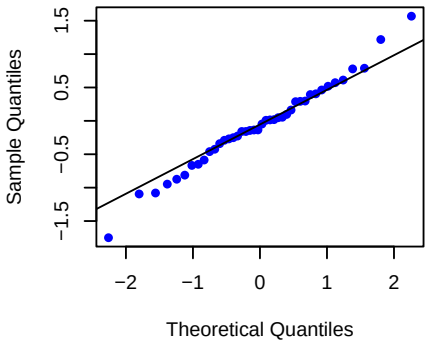

Fig. 16.5. Q-Q plot of the forecast errors (innovations) for the DLM specified in Equations 16.5 to 16.8.

The Q-Q plot (Figure 16.5) indicates that the innovations appear to be more-or-less normally distributed (i.e., most points fall on the line). Furthermore, it looks like the mean of the innovations is about 0, but we should use a more reliable test than simple visual inspection. We can formally test whether the mean of the innovations is significantly different from 0 by using a one-sample t-test. based on a null hypothesis of $\mathrm{E}[e_t] = 0$. To do so, we will use the function `t.test()` and base our inference on a significance value of $\alpha = 0.05$.

```
# p-value for t-test of H0: E(innov) = 0
t.test(t(innov), mu = 0)$p.value
```

```
[1] 0.4840901
```

The p-value $>> 0.05$ so we cannot reject the null hypothesis that $\mathrm{E}[e_t] = 0$.

Moving on to assumption (2), we can use the sample autocorrelation function (ACF) to examine whether the innovations are autocorrelated (they should not be). Using the `acf()` function, we can compute and plot the correlations of e_t and e_{t-k} for various values of k. Assumption (2) will be met if none of the correlation coefficients exceed the 95% confidence intervals defined by $\pm z_{0.975}/\sqrt{n}$.

```
# plot ACF of innovations
acf(t(innov), lag.max = 10)
```

The ACF plot (Figure 16.6) shows no significant autocorrelation in the innovations at lags 1–10, so it appears that both of our model assumptions have been met.

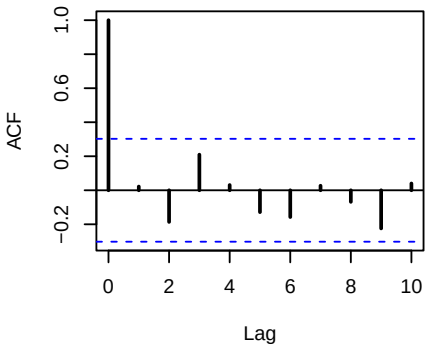

Fig. 16.6. Autocorrelation plot of the forecast errors (innovations) for the DLM specified in Equations 16.5 to 16.8. Horizontal blue lines define the upper and lower 95% confidence intervals.

17

Multivariate linear regression

This chapter shows how to write regression models with multivariate responses and multivariate explanatory variables in MARSS form. R has many excellent functions and packages for multiple linear regression. We will be showing how to use the `MARSS()` function to fit these models, but note that R 's standard linear regression functions would be much better choices in most cases. The purpose of this chapter is to show the relationship between multivariate linear regression and the MARSS equation.

In a classic linear regression, the response variable (y) is univariate and there may be one to multiple explanatory variables (d_1, d_2, ...) plus an optional intercept (α):

$$y_t = \alpha + \sum_k \beta_k d_k + e_t, \text{ where } e_t \sim \mathrm{N}(0, \sigma^2) \tag{17.1}$$

Here the subscript, t is used since we are working with time-series data. Explanatory variables are normally denoted x in linear regression however x is not used here since x is already used in MARSS models to denote the hidden process trajectory. Instead d is used when the explanatory variables appear in the y part of the equation (and c if they appear in the x part).

This chapter will start with classical linear regression where the explanatory variables are treated as inputs that are known without error and where we are trying to explain the variation in y with our explanatory variables. We will extend this to the case of autocorrelated errors.

17.1 Univariate linear regression

A vanilla linear regression where our data are time ordered but we treat them as independent can be written as

---

Type `RShowDoc("Chapter_MLR.R",package="MARSS")` at the R command line to open a file with all the code for the examples in this chapter.

$$y_t = \alpha + \beta_1 d_{1,t} + \beta_2 d_{2,t} + e_t, \tag{17.2}$$

where the d are our explanatory variables. This model can be written in many different ways in as a MARSS equation. Here we use a specific form where the i.i.d. component of the errors is v_t in the y part of the MARSS equation and autocorrelated errors will appear as x_t in the y equation. Specifying the MARSS model this way allows us to use the EM-algorithm to fit the model which will prove to be important.

$$\begin{gathered} y_t = \alpha + \begin{bmatrix}\beta_1 & \beta_2 & \ldots\end{bmatrix} \begin{bmatrix} d_{1,t} \\ d_{2,t} \\ \vdots \end{bmatrix} + v_t + x_t, v_t \sim \mathrm{N}(0, r) \\ x_t = b x_{t-1} + w_t, w_t \ \mathrm{N}(0, q) \\ x_0 = 0 \end{gathered} \tag{17.3}$$

The v_t are the i.i.d. errors and the x_t are the AR(1) errors.

17.1.1 Univariate response using the Longley dataset: example 1

We will start by using an example from Chapter 6 in Linear Models in R (Faraway, 2004). This example uses the built-in R dataset "longley" which has the number of people employed from 1947 to 1962 and a number of predictors. For this example we will regress the number of people employed against gross National product and population size (following Faraway).

Mathematically, the model we are fitting is

$$Employed_t = \alpha + \begin{bmatrix}\beta_{GNP} & \beta_{Pop}\end{bmatrix} \begin{bmatrix} GNP_t \\ Pop_t \end{bmatrix} + v_t, v_t \sim \mathrm{N}(0, r) \tag{17.4}$$

x does not appear in the vanilla linear regression since we do not have autocorrelated errors (yet). We are trying to estimate α (intercept), β_{GNP} and β_{Pop}.

A full multivariate MARSS model looks like

$$\begin{aligned} \mathbf{y}_t &= \mathbf{Z}\mathbf{x}_t + \mathbf{a} + \mathbf{D}\mathbf{d}_t + \mathbf{v}_t, \text{ where } \mathbf{v}_t \sim \mathrm{MVN}(0, \mathbf{R}) \\ \mathbf{x}_t &= \mathbf{B}\mathbf{x}_{t-1} + \mathbf{u} + \mathbf{C}\mathbf{c}_t + \mathbf{w}_t, \text{ where } \mathbf{w}_t \sim \mathrm{MVN}(0, \mathbf{Q}) \end{aligned} \tag{17.5}$$

We need to specify the parameters in Equation 17.5 such that we get Equation 17.4.

First, we load the data and set up y, the response variable number of people employed, as a matrix with time going across the columns.

```
data(longley)
Employed <- matrix(longley$Employed, nrow = 1)
```

Second create a list to hold our model specification.

```
longley.model <- list()
```

Set the $\mathbf{u}$, $\mathbf{Q}$ and $\mathbf{x}_0$ parameters to 0. We will also set $\mathbf{a}$ and $\mathbf{C}$ to 0 and $\mathbf{B}$ and $\mathbf{Z}$ to identity although this is not necessary since these are the defaults.

Fig. 17.1. Employment time series from the Longley dataset.

```
longley.model$U <- longley.model$Q <- "zero"
longley.model$C <- "zero"
longley.model$B <- longley.model$Z <- "identity"
longley.model$x0 <- "zero"
longley.model$tinitx <- 0
```

We will estimate **R**, the variance of the i.i.d. errors (residuals).

```
longley.model$R <- matrix("r")
```

The **D** matrix has the two β (slope) parameters for GNP and Population and *a* has the intercept.[1]

```
longley.model$A <- matrix("intercept")
longley.model$D <- matrix(c("GNP", "Pop"), nrow = 1)
```

[1] A better way to fit the model is to put the intercept into **D** by adding a row of 1s to **d** and putting the intercept parameter on the first row of **D**. This reduces by one the number of matrices being estimated by the EM algorithm. It's not done here just so the equations look more like standard linear regression equations.

Last we set up our explanatory variables. This is the **d** matrix and we need each explanatory variable in a row with time across the columns.

```
longley.model$d <- rbind(longley$GNP, longley$Population)
```

Now we can fit the model:

```
mod1 <- MARSS(Employed, model = longley.model)
```

and look at the estimates.

```
coef(mod1, type = "vector")
```

`method="BFGS"` can also be used and gives similar results.

We can compare the fit to that from lm() and see that we get the same estimates:

```
mod1.lm <- lm(Employed ~ GNP + Population, data = longley)
coef(mod1.lm)
```

```
(Intercept)         GNP  Population
88.93879831  0.06317244 -0.40974292
```

17.1.2 Univariate response using auto-correlated errors: example 1

As Faraway (2004) discusses, the errors in this dataset are temporally correlated. We can model the errors as an AR(1) process to account for this. This changes our model to

$$\begin{gathered} Employed_t = \alpha + \begin{bmatrix}\beta_{GNP} & \beta_{Pop}\end{bmatrix}\begin{bmatrix}GNP_t \\ Pop_t\end{bmatrix} + v_t + x_t, v_t \sim \mathrm{N}(0,r) \\ x_t = bx_{t-1} + w_t, w_t \sim \mathrm{N}(0,q) \\ \mathbf{x}_0 = 0 \end{gathered} \tag{17.6}$$

We assume the AR(1) errors have mean 0 so $u = 0$ in the x_t equation. Setting u to anything else would make the mean of our errors equal to $u/(1-b)$ for $-1 < b < 1$. This would lead to two mean levels in our model, α and $u/(1-b)$, and we would not be able to estimate both. Notice that the model is somewhat confounded since if $b = 0$ then x_t is i.i.d. errors same as v_t. In this case, either q or r would be redundant. It is thus possible that either r or q will go to zero.

To fit the model with autoregressive errors, we add the x parameters to our the model list. We estimate b and q.

```
longley.ar1 <- longley.model
longley.ar1$B <- matrix("b")
longley.ar1$Q <- matrix("q")
```

Now we can fit the model as before

```
mod2 <- MARSS(Employed, model = longley.ar1)
```

however, this is a difficult model to fit and takes a long, long time to converge. The default `maxit` used in the call above is not nearly enough iterations. Using `method="BFGS"` helps a little but not much. We can improve behavior by using the fit of the model with i.i.d. errors as initial conditions for **D** and a.

```
inits <- list(A = coef(mod1)$A, D = coef(mod1)$D)
mod2 <- MARSS(Employed,
  model = longley.ar1, inits = inits,
  control = list(maxit = 1000)
)
ests.marss <- c(
  b = coef(mod2)$B, alpha = coef(mod2)$A,
  GNP = coef(mod2)$D[1], Population = coef(mod2)$D[2],
  logLik = logLik(mod2)
)
```

We can the fit the same model using `gls()` (in the {nlme} package). The b term is called Phi in the `gls()` call and is somewhat difficult to recover although it is printed by `summary()`.

```
library(nlme)
mod2.gls <- gls(Employed ~ GNP + Population,
  correlation = corAR1(), data = longley, method = "ML"
)
mod2.gls.phi <- coef(mod2.gls$modelStruct[[1]], unconstrained = FALSE)
ests.gls <- c(
  b = mod2.gls.phi, alpha = coef(mod2.gls)[1],
  GNP = coef(mod2.gls)[2], Population = coef(mod2.gls)[3],
  logLik = logLik(mod2.gls)
)
```

Note we need to set `method="ML"` to maximize the likelihood because the default is to maximize the restricted maximum-likelihood (`method="REML"`) and that gives a different answer from the `MARSS()` function since `MARSS()` is maximizing the likelihood.

Both functions return similar values though `gls()` is much faster and the EM algorithm has not quite converged even with 1000 iterations.

```
rbind(MARSS = ests.marss, GLS = ests.gls)
```

```
             b    alpha        GNP Population    logLik
MARSS 0.3509377 95.36017 0.06748567 -0.4784003 -10.53742
GLS   0.3651196 96.09369 0.06822305 -0.4871554 -10.47396
```

17.1.3 Univariate response using the Longley dataset: example 2

The full Longley dataset is often used to test the performance of numerical methods for fitting linear regression models because it has severe collinearity problems (Figure 17.2). We can compare the EM and BFGS algorithms for the full dataset and see

how fitting a MARSS model with the BFGS algorithm leads to estimates far from the maximum-likelihood values for this problem.

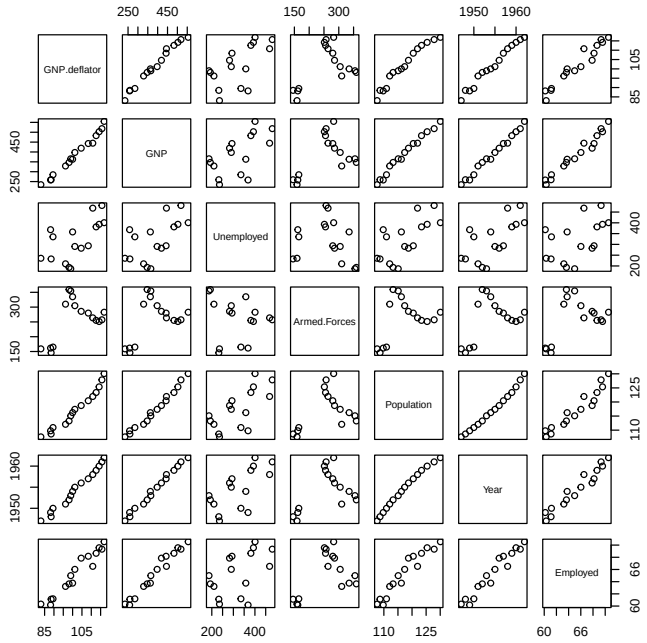

Fig. 17.2. Pairs plot showing collinearity in the Longley explanatory variables.

We can fit a regression of Employed to all the Longley explanatory variables using the following code. The mathematical model is the same as in Equation 17.4 except that instead of two explanatory variables with have all seven shown in Figure 17.2.

```
eVar.names <- colnames(longley)[-7]
eVar <- t(longley[, eVar.names])
longley.model <- list()
longley.model$U <- longley.model$Q <- "zero"
longley.model$C <- "zero"
longley.model$B <- longley.model$Z <- "identity"
longley.model$A <- matrix("intercept")
longley.model$R <- matrix("r")
longley.model$D <- matrix(eVar.names, nrow = 1)
longley.model$d <- eVar
```

```
longley.model$x0 <- "zero"
longley.model$tinitx <- 0
```

Then we fit as usual. We will fit with the EM-algorithm (the default) and compare to BFGS.

```
mod3.em <- MARSS(Employed, model = longley.model)
mod3.bfgs <- MARSS(Employed, model = longley.model, method = "BFGS")
```

Here are the EM estimates with the log-likelihood.

```
par.names <- c("A.intercept", paste("D", eVar.names, sep = "."))
c(coef(mod3.em, type = "vector")[par.names], logLik = mod3.em$logLik)
```

```
  A.intercept D.GNP.deflator          D.GNP   D.Unemployed
-3.482258e+03   1.506187e-02  -3.581917e-02  -2.020230e-02
D.Armed.Forces   D.Population         D.Year         logLik
-1.033227e-02  -5.110413e-02   1.829151e+00   9.066497e-01
```

Compared to the BFGS estimates:

```
c(coef(mod3.bfgs, type = "vector")[par.names], logLik = mod3.bfgs$logLik)
```

```
  A.intercept D.GNP.deflator          D.GNP   D.Unemployed
-14.062098960   -0.052705201    0.070642032   -0.004298481
D.Armed.Forces   D.Population         D.Year         logLik
 -0.005744197   -0.412771923    0.055610012   -6.996818720
```

And compared to the estimates from the `lm()` function:

```
mod3.lm <- lm(Employed ~ 1 + GNP.deflator + GNP + Unemployed
  + Armed.Forces + Population + Year, data = longley)
c(coef(mod3.lm), logLik = logLik(mod3.lm))
```

```
  (Intercept)  GNP.deflator           GNP    Unemployed
-3.482259e+03  1.506187e-02 -3.581918e-02 -2.020230e-02
 Armed.Forces    Population          Year        logLik
-1.033227e-02 -5.110411e-02  1.829151e+00  9.066497e-01
```

As you can see the BFGS algorithm struggles with the ridge-like likelihood caused by the collinearity in the explanatory variables.

We can also compare the performance of the model with AR(1) errors. This is Equation 17.6 but with all seven explanatory variables. We set up the MARSS model[2] for a linear regression with correlated errors as before with the addition of b (called Phi in `gls()`) and q.

[2] Notice that x_0 is set at 0. The model is having a hard time fitting x_0 because the time series is short. Estimating x_0 or using a diffuse prior by setting $\mathbf{V}_0$ big, leads to poor estimates. Since this is just the error term, we set $x_0 = 0$ since the mean of the errors is assumed to be 0.

```
longley.correrr.model <- longley.model
longley.correrr.model$B <- matrix("b")
longley.correrr.model$Q <- matrix("q")
```

We fit as usual and compare the EM-algorithm (the default) to fits using BFGS. We will use the estimate from the model with i.i.d. errors as initial conditions.

```
inits <- list(A = coef(mod3.em)$A, D = coef(mod3.em)$D)
mod4.em <- MARSS(Employed, model = longley.correrr.model, inits = inits)
mod4.bfgs <- MARSS(Employed,
  model = longley.correrr.model,
  inits = inits, method = "BFGS"
)
```

Here are the EM estimates with the log-likelihood. We only show ϕ (the b term in the AR(1) error equation) and the log-likelihood.

```
c(coef(mod4.em, type = "vector")["B.b"], logLik = mod4.em$logLik)
```

```
      B.b    logLik
-0.7737465  4.5374546
```

Compared to the BFGS estimates:

```
c(coef(mod4.bfgs, type = "vector")["B.b"], logLik = mod4.bfgs$logLik)
```

```
     B.b   logLik
0.8368899 0.9066497
```

And compared to the estimates from the `gls()` function:

```
mod4.gls <- gls(Employed ~ 1 + GNP.deflator + GNP + Unemployed
  + Armed.Forces + Population + Year,
correlation = corAR1(), data = longley, method = "ML"
)
mod4.gls.phi <- coef(mod4.gls$modelStruct[[1]], unconstrained = FALSE)
c(mod4.gls.phi, logLik = logLik(mod4.gls))
```

```
      Phi    logLik
-0.7288697  4.3865475
```

Again we see that the BFGS algorithm struggles with the ridge-like likelihood caused by the collinearity in the explanatory variables.

17.2 Multivariate response example using longitudinal data

We will illustrate linear regression with a multivariate response using longitudinal data from a sleep study on 18 subjects from the {lme4} R package. These are data on reaction time of subjects after 0 to 9 days of being restricted to 3 hours of sleep.

We load the data from the {lme4} package:

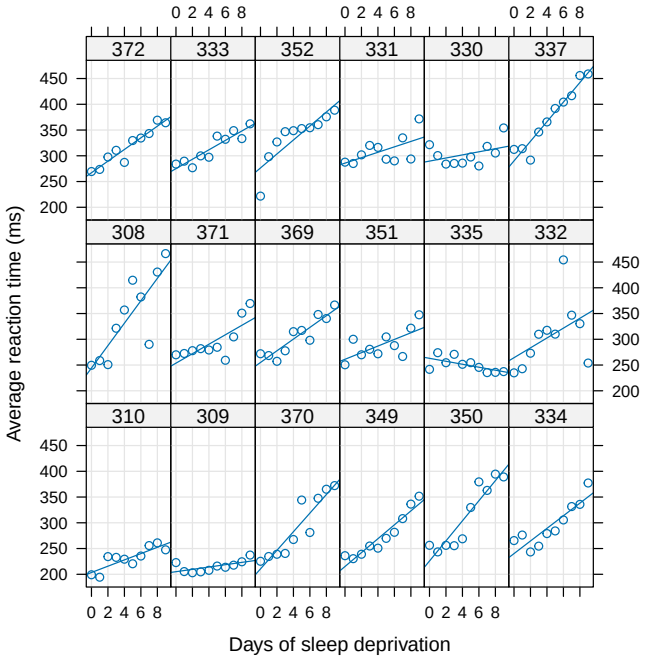

Fig. 17.3. Plot of the sleep study data (package lme4).

```
data(sleepstudy, package = "lme4")
```

We set up the data into a matrix for the `MARSS()` function with each subject as a row with day across the columns. The explanatory variable is the the day number 0 to 9 and we make this into a matrix with one row and day across the columns.

```
# number of subjects
nsub <- length(unique(sleepstudy$Subject))
ndays <- length(sleepstudy$Days) / nsub
dat <- matrix(sleepstudy$Reaction, nsub, ndays, byrow = TRUE)
rownames(dat) <- paste("sub", unique(sleepstudy$Subject), sep = ".")
exp.var <- matrix(sleepstudy$Days, 1, ndays, byrow = TRUE)
```

Let's start with a simple regression where each subject has a separate intercept (reaction time at day 0) but the slope (increase in reaction time with each successive day) is the same across the 18 subjects. Mathematically the model is

$$\begin{bmatrix} resp_1 \\ resp_2 \\ \dots \\ resp_{18} \end{bmatrix}_t = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \dots \\ \alpha_{18} \end{bmatrix} + \begin{bmatrix} \beta \\ \beta \\ \dots \\ \beta \end{bmatrix} day_t + \begin{bmatrix} v_1 \\ v_2 \\ \dots \\ v_{18} \end{bmatrix}_t$$
$$\begin{bmatrix} v_1 \\ v_2 \\ \dots \\ v_{18} \end{bmatrix}_t \sim \mathrm{N}\left(0, \begin{bmatrix} r & 0 & \dots & 0 \\ 0 & r & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & r \end{bmatrix}\right) \tag{17.7}$$

The response time of subject i is a subject specific intercept (α_i) plus an effect of day at time t that doesn't vary by subject and error that is i.i.d. across subject and day.

We specify and fit this model as follows

```
sleep.model <- list(
  A = "unequal", B = "zero", x0 = "zero", U = "zero",
  D = matrix("b1", nsub, 1), d = exp.var, tinitx = 0, Q = "zero"
)
sleep.mod1 <- MARSS(dat, model = sleep.model)
```

This is the same as the following with `lm()`:

```
sleep.lm1 <- lm(Reaction ~ -1 + Subject + Days, data = sleepstudy)
```

Now let's allow each subject to have different slopes (increase in reaction time with each successive day) across subjects. This model is

$$\begin{bmatrix} resp_1 \\ resp_2 \\ \dots \\ resp_{18} \end{bmatrix}_t = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \dots \\ \alpha_{18} \end{bmatrix} + \begin{bmatrix} \beta_1 \\ \beta_2 \\ \dots \\ \beta_{18} \end{bmatrix} day_t + \begin{bmatrix} v_1 \\ v_2 \\ \dots \\ v_{18} \end{bmatrix}_t$$
$$\begin{bmatrix} v_1 \\ v_2 \\ \dots \\ v_{18} \end{bmatrix}_t \sim \mathrm{N}\left(0, \begin{bmatrix} r & 0 & \dots & 0 \\ 0 & r & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & r \end{bmatrix}\right) \tag{17.8}$$

We specify and fit this model as

```
sleep.model <- list(
  A = "unequal", B = "zero", x0 = "zero", U = "zero",
  D = "unequal", d = exp.var, tinitx = 0, Q = "zero"
)
sleep.mod2 <- MARSS(dat, model = sleep.model, silent = TRUE)
```

This is the same as the following with `lm()`:

```
sleep.lm2 <- lm(Reaction ~ 0 + Subject + Days:Subject, data = sleepstudy)
```

We can repeat the above but allow the residual variance to differ across subjects by setting `R="diagonal and unequal"`. This model is

$$\begin{gathered}\begin{bmatrix} resp_1 \\ resp_2 \\ \dots \\ resp_{18} \end{bmatrix}_t = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \dots \\ \alpha_{18} \end{bmatrix} + \begin{bmatrix} \beta_1 \\ \beta_2 \\ \dots \\ \beta_{18} \end{bmatrix} day_t + \begin{bmatrix} v_1 \\ v_2 \\ \dots \\ v_{18} \end{bmatrix}_t \\ \begin{bmatrix} v_1 \\ v_2 \\ \dots \\ v_{18} \end{bmatrix}_t \sim \mathrm{N}\left(0, \begin{bmatrix} r_1 & 0 & \dots & 0 \\ 0 & r_2 & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & r_{18} \end{bmatrix}\right)\end{gathered} \tag{17.9}$$

```
sleep.model <- list(
  A = "unequal", B = "zero", x0 = "zero", U = "zero",
  D = "unequal", d = exp.var, tinitx = 0, Q = "zero",
  R = "diagonal and unequal"
)
sleep.mod3 <- MARSS(dat, model = sleep.model, silent = TRUE)
```

Or we can allow AR(1) errors across subjects and allow each subject to have its own AR(1) parameters for this error. This model is

$$\begin{gathered}\begin{bmatrix} resp_1 \\ resp_2 \\ \dots \\ resp_{18} \end{bmatrix}_t = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \dots \\ \alpha_{18} \end{bmatrix} + \begin{bmatrix} \beta_1 \\ \beta_2 \\ \dots \\ \beta_{18} \end{bmatrix} day_t + \begin{bmatrix} v_1 \\ v_2 \\ \dots \\ v_{18} \end{bmatrix}_t + \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_{18} \end{bmatrix}_t \\ \begin{bmatrix} v_1 \\ v_2 \\ \dots \\ v_{18} \end{bmatrix}_t \sim \mathrm{N}\left(0, \begin{bmatrix} r_1 & 0 & \dots & 0 \\ 0 & r_2 & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & r_{18} \end{bmatrix}\right) \\ \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_{18} \end{bmatrix}_t = \begin{bmatrix} b_1 & 0 & \dots & 0 \\ 0 & b_2 & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & b_{18} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_{18} \end{bmatrix}_{t-1} + \begin{bmatrix} w_1 \\ w_2 \\ \dots \\ w_{18} \end{bmatrix}_t \\ \begin{bmatrix} w_1 \\ w_2 \\ \dots \\ w_{18} \end{bmatrix}_t \sim \mathrm{N}\left(0, \begin{bmatrix} q_1 & 0 & \dots & 0 \\ 0 & q_2 & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & q_{18} \end{bmatrix}\right)\end{gathered} \tag{17.10}$$

We fit this model as

```
inits <- list(A = coef(sleep.mod3)$A, D = coef(sleep.mod3)$D)
# estimate a separate intercept for each but slope is the same
sleep.model <- list(
  A = "unequal", B = "diagonal and unequal", x0 = "zero", U = "zero",
  D = "unequal", d = exp.var, tinitx = 0, Q = "diagonal and unequal",
  R = "diagonal and unequal"
)
sleep.mod4 <- MARSS(dat, model = sleep.model, inits = inits, silent = TRUE)
```

It is not obvious how to specify these last two models using `gls()` or if it is possible.

We can also allow each subject to have his/her own error process but specify that the parameters of these (b, q and r) are the same across subjects. We do this by using `"diagonal and equal"`. Mathematically this model is

$$\begin{gathered}
\begin{bmatrix} resp_1 \\ resp_2 \\ \dots \\ resp_{18} \end{bmatrix}_t = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \dots \\ \alpha_{18} \end{bmatrix} + \begin{bmatrix} \beta_1 \\ \beta_2 \\ \dots \\ \beta_{18} \end{bmatrix} day_t + \begin{bmatrix} v_1 \\ v_2 \\ \dots \\ v_{18} \end{bmatrix}_t + \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_{18} \end{bmatrix}_t \\
\begin{bmatrix} v_1 \\ v_2 \\ \dots \\ v_{18} \end{bmatrix}_t \sim \mathrm{N}\left(0, \begin{bmatrix} r & 0 & \dots & 0 \\ 0 & r & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & r \end{bmatrix}\right) \\
\begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_{18} \end{bmatrix}_t = \begin{bmatrix} b & 0 & \dots & 0 \\ 0 & b & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & b \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_{18} \end{bmatrix}_{t-1} + \begin{bmatrix} w_1 \\ w_2 \\ \dots \\ w_{18} \end{bmatrix}_t \\
\begin{bmatrix} w_1 \\ w_2 \\ \dots \\ w_{18} \end{bmatrix}_t \sim \mathrm{N}\left(0, \begin{bmatrix} q & 0 & \dots & 0 \\ 0 & q & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & q \end{bmatrix}\right)
\end{gathered} \tag{17.11}$$

We specify and fit this model as

```
inits <- list(A = coef(sleep.mod3)$A, D = coef(sleep.mod3)$D)
# estimate a separate intercept for each but slope is the same
sleep.model <- list(
  A = "unequal", B = "diagonal and equal", x0 = "zero", U = "zero",
  D = "unequal", d = exp.var, tinitx = 0, Q = "diagonal and equal",
  R = "diagonal and equal"
)
sleep.mod5 <- MARSS(dat, model = sleep.model, inits = inits, silent = TRUE)
```

This is fairly close to this model fit with `gls()`.

```
sleep.mod5.gls <- gls(Reaction ~ 0 + Subject + Days:Subject,
  data = sleepstudy,
  correlation = corAR1(form = ~ 1 | Subject), method = "ML"
)
```

The way the variance-covariance structure is modeled is a little different but it is the same idea.

17.3 Discussion

The purpose of this chapter is to illustrate how linear regression models with multivariate explanatory variables can be written in MARSS form and fit with the `MARSS()`

Table 17.1. Parameter estimates of different versions of the model where each subject has a separate intercept (response time on normal sleep) and different slope by day (increase in response time with each day of sleep deprivation). The model types are discussed in the text.

	lm	mod2 em	mod3 em	mod4 em	mod5 em	mod5 gls
logLik	-818.94	-818.94	-770.19	-754.97	-818.76	-818.55
slope 308	21.76	21.76	21.76	21.77	21.83	21.87
slope 309	2.26	2.26	2.26	1.43	2.24	2.23
slope 310	6.11	6.11	6.11	6.12	6.10	6.08
slope 330	3.01	3.01	3.01	2.93	3.01	3.04
slope 331	5.27	5.27	5.27	3.59	5.36	5.46
slope 332	9.57	9.57	9.57	8.55	9.39	9.21
slope 333	9.14	9.14	9.14	8.85	9.12	9.12
slope 334	12.25	12.25	12.25	11.73	12.24	12.26
slope 335	-2.88	-2.88	-2.88	-3.19	-2.82	-2.77
slope 337	19.03	19.03	19.03	19.09	18.95	18.90
slope 349	13.49	13.49	13.49	12.14	13.47	13.46
slope 350	19.50	19.50	19.50	18.21	19.38	19.28
slope 351	6.43	6.43	6.43	6.15	6.54	6.64
slope 352	13.57	13.57	13.57	19.20	13.71	13.80
slope 369	11.35	11.35	11.35	11.41	11.32	11.31
slope 370	18.06	18.06	18.06	18.31	18.01	17.97
slope 371	9.19	9.19	9.19	9.56	9.23	9.28
slope 372	11.30	11.30	11.30	11.45	11.28	11.26
phi 308				0.02	0.12	0.08
phi 309				0.63	0.12	0.08
phi 310				-0.01	0.12	0.08
phi 330				0.32	0.12	0.08
phi 331				-1.66	0.12	0.08
phi 332				0.26	0.12	0.08
phi 333				-1.04	0.12	0.08
phi 334				0.51	0.12	0.08
phi 335				-0.40	0.12	0.08
phi 337				-0.08	0.12	0.08
phi 349				0.80	0.12	0.08
phi 350				0.32	0.12	0.08
phi 351				-0.15	0.12	0.08
phi 352				0.80	0.12	0.08
phi 369				-0.25	0.12	0.08
phi 370				-0.44	0.12	0.08
phi 371				0.63	0.12	0.08
phi 372				-0.47	0.12	0.08

function[3]. This is to help one understand the relationship between the MARSS model form and the more familiar multivariate linear model forms. Obviously R has many, many excellent packages for linear regression and generalized linear regression (non-Gaussian errors). While the {MARSS} package can fit a variety of linear regression models with Gaussian errors, that is not what the package is designed to do. The {MARSS} package is designed for fitting models that cannot be fit with typical linear regression: multivariate autoregressive state-space models with inputs (explanatory variables) and linear constraints.

[3] with caveat that one must always be careful when the likelihood surface has prominent ridges which will occur with collinear explanatory variables.

18

Lag-p MARSS models

18.1 Background

Most of the chapters in the User Guide are 'lag-1' in the autoregressive part of the model. This means that $\mathbf{x}_t$ in the process model only depends on $\mathbf{x}_{t-1}$ and not $\mathbf{x}_{t-2}$ (lag-2) or more generally $\mathbf{x}_{t-p}$ (lag-p). A lag-p model can be written in state-space form as a MARSS lag-1 model, aka a MARSS(1) model (see section 11.3.2 in Tsay (2010)). Writing lag-p models in this form allows one to take advantage of the fitting algorithms for MARSS(1) models. There are a number of ways to do the conversion to a MARSS(1) form. We use Hamilton's form (section 1 in Hamilton (1994)) because it can be fit with an EM algorithm while the other forms (Harvey's and Akaike's) cannot.

This chapter shows how to convert and fit the following using the MARSS(1) form:

AR(p) A univariate autoregressive model where x_t is a function of x_{t-p} (and the prior lags usually too). No observation error.

MAR(p) The same as AR(p) but the **x** term is multivariate not univariate.

ARSS(p) The same as AR(p) but with a observation model and observation error. The observations (**y**) may be multivariate but the **x** term is univariate.

MARSS(p) The same as ARSS(p) but the **x** term is multivariate not univariate.

Note that only ARSS(p) and MARSS(p) assume observation error in the data. AR(p) and MAR(p) will be rewritten in the state-space form with a **y** component to facilitate statistical analysis but the data themselves are considered error free.

Note there are many R packages for fitting AR(p) (and ARMA(p,q) for that matter) models. If you are only interested in univariate data with no observation error in

---

Type `RShowDoc("Chapter_MARp.R",package="MARSS")` at the R command line to open a file with all the code for the examples in this chapter.

the data then you probably want to look into the `arima()` function included in base R and into R packages that specialize in fitting ARMA models to univariate data. The {forecast} package in R is a good place to start but others can be found on the CRAN task view: Time Series Analysis.

18.2 MAR(2) models

A MAR(2) model is a lag-2 MAR model, aka a multivariate autoregressive process with no observation process (no SS part). A MAR(2) model is written

$$\mathbf{x}_t' = \mathbf{B}_1\mathbf{x}_{t-1}' + \mathbf{B}_2\mathbf{x}_{t-2}' + \mathbf{u} + \mathbf{w}_t, \text{ where } \mathbf{w}_t \sim \text{MVN}(0, \mathbf{Q}) \tag{18.1}$$

We rewrite this as MARSS(1) by defining $\mathbf{x}_t = \begin{bmatrix} \mathbf{x}_t' \\ \mathbf{x}_{t-1}' \end{bmatrix}$:

$$\begin{bmatrix} \mathbf{x}_t' \\ \mathbf{x}_{t-1}' \end{bmatrix} = \begin{bmatrix} \mathbf{B}_1 & \mathbf{B}_2 \\ \mathbf{I}_m & 0 \end{bmatrix} \begin{bmatrix} \mathbf{x}_{t-1}' \\ \mathbf{x}_{t-2}' \end{bmatrix} + \begin{bmatrix} \mathbf{u} \\ 0 \end{bmatrix} + \begin{bmatrix} w_t \\ 0 \end{bmatrix}, \ \begin{bmatrix} w_t \\ 0 \end{bmatrix} \sim \text{MVN}\left(0, \begin{bmatrix} \mathbf{Q} & 0 \\ 0 & 0 \end{bmatrix}\right)$$
$$\begin{bmatrix} \mathbf{x}_0' \\ \mathbf{x}_{-1}' \end{bmatrix} \sim \text{MVN}(\mu, \Lambda) \tag{18.2}$$

Our observations are of $\mathbf{x}_t$ only, so our observation model is

$$\mathbf{y}_t = \begin{bmatrix} \mathbf{I} & 0 \end{bmatrix} \begin{bmatrix} \mathbf{x}_t' \\ \mathbf{x}_{t-1}' \end{bmatrix} \tag{18.3}$$

18.2.1 Example of AR(2): univariate data

Here is an example of fitting a univariate AR(2) model written in MARSS(1) form. First, let's generate some simulated AR(2) data from this AR(2) process:

$$x_t = -1.5x_{t-1} + -0.75x_{t-2} + w_t, \text{ where } w_t \sim \text{N}(0,1) \tag{18.4}$$

```
TT <- 50
true.2 <- c(r = 0, b1 = -1.5, b2 = -0.75, q = 1)
temp <- arima.sim(n = TT, list(ar = true.2[2:3]), sd = sqrt(true.2[4]))
sim.ar2 <- matrix(temp, nrow = 1)
```

Next, we set up the model list for an AR(2) model written in MARSS(1) form (refer to Equation 18.2 and 18.3):

```
Z <- matrix(c(1, 0), 1, 2)
B <- matrix(list("b1", 1, "b2", 0), 2, 2)
U <- matrix(0, 2, 1)
Q <- matrix(list("q", 0, 0, 0), 2, 2)
A <- matrix(0, 1, 1)
R <- matrix(0, 1, 1)
```

```
mu <- matrix(sim.ar2[2:1], 2, 1)
V <- matrix(0, 2, 2)
model.list.2 <- list(
  Z = Z, B = B, U = U, Q = Q, A = A,
  R = R, x0 = mu, V0 = V, tinitx = 0
)
```

Notice that we do not estimate μ. We will fit our model to the data (y) starting at $t = 3$. Because $\mathbf{R} = 0$, this means $\mathrm{E}[\mathbf{X}_t|\mathbf{y}_t] = \mathbf{x}_t^t = \mathbf{y}_t$ and $\mathbf{x}_0^0 \equiv \begin{bmatrix} y_2 \\ y_1 \end{bmatrix}$. Note $\mathrm{E}[\mathbf{X}_t|\mathbf{y}_{t-1}] = \mathbf{x}_t^{t-1} \neq \mathbf{y}_t$ so we do not use $\mathbf{x}_1^0$ as our initial $\mathbf{x}$.

Then we can then estimate the b_1 and b_2 parameters for the AR(2) process.

```
ar2 <- MARSS(sim.ar2[3:TT], model = model.list.2)
```

```
Success! algorithm run for 15 iterations. abstol and log-log tests passed.
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
Algorithm ran 15 (=minit) iterations and convergence was reached.
Log-likelihood: -63.02523
AIC: 132.0505   AICc: 132.5959

     Estimate
B.b1   -1.582
B.b2   -0.777
Q.q     0.809
Initial states (x0) defined at t=0

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

Comparison to the true values shows the estimates are close:

```
print(cbind(true = true.2[2:4], estimates = coef(ar2, type = "vector")))
```

```
    true  estimates
b1 -1.50 -1.5816137
b2 -0.75 -0.7767462
q   1.00  0.8091055
```

Missing values in the data are fine. Let's make half the data missing being careful that the first data point does not get categorized as missing. `MARSSkfss()` is used as the Kalman filter/smoother function as this is a model where `MARSSkfas()` can return negative values on the states variance-covariance matrix.

```
gappy.data <- sim.ar2[3:TT]
gappy.data[floor(runif(TT / 2, 2, TT))] <- NA
ar2.gappy <- MARSS(gappy.data, model = model.list.2, fun.kf="MARSSkfss")
```

And the estimates are still close:

```
print(cbind(
  true = true.2[2:4],
  estimates.no.miss = coef(ar2, type = "vector"),
  estimates.w.miss = coef(ar2.gappy, type = "vector")
))
```

```
   true estimates.no.miss estimates.w.miss
b1 -1.50       -1.5816137      -1.5549387
b2 -0.75       -0.7767462      -0.7463820
q   1.00        0.8091055       0.8868251
```

By the way, there are much better and faster functions in R for fitting univariate AR models (no observation error). The {MARSS} package is really for fitting to multivariate data with observation error not AR(p) models. For example, here is how you would fit the AR(2) model using the `arima()` function:

```
arima(gappy.data, order = c(2, 0, 0), include.mean = FALSE)
```

```
Call:
arima(x = gappy.data, order = c(2, 0, 0), include.mean = FALSE)

Coefficients:
          ar1      ar2
      -1.5674  -0.7494
s.e.   0.1033   0.1015

sigma^2 estimated as 0.9428:  log likelihood = -51.81,  aic = 109.62
```

The estimates will be different because `arima()` sets $\mathbf{x}_1^0$ as coming from the stationary distribution. That is a non-linear constraint that `MARSS()` cannot handle.

The assumption that $\mathbf{x}_1^0$ comes from the stationary distribution is fine if the initial $\mathbf{x}$ indeed comes from the stationary distribution, but if the initial $\mathbf{x}$ is well outside the stationary distribution the estimates will be incorrect.

```
TT <- 50
true.2 <- c(r = 0, b1 = -1.5, b2 = -0.75, q = 1)
sim.ar2.ns <- rep(NA, TT)
sim.ar2.ns[1] <- -30
sim.ar2.ns[2] <- -10
for (i in 3:TT) {
  sim.ar2.ns[i] <- true.2[2] * sim.ar2.ns[i - 1] +
    true.2[3] * sim.ar2.ns[i - 2] + rnorm(1, 0, sqrt(true.2[4]))
```

```
}
model.list.3 <- model.list.2
model.list.3$x0 <- matrix(sim.ar2.ns[2:1], 2, 1)
ar3.marss <- MARSS(sim.ar2.ns[3:TT], model = model.list.3, silent = TRUE)
ar3.arima <- arima(sim.ar2.ns[3:TT], order = c(2, 0, 0), include.mean = FALSE)
print(cbind(
  true = true.2[2:4],
  estimates.marss = coef(ar3.marss, type = "vector"),
  estimates.arima = c(coef(ar3.arima, type = "vector"), ar3.arima$sigma2)
))
```

```
   true estimates.marss estimates.arima
b1 -1.50      -1.5037048      -1.7490942
b2 -0.75      -0.7464002      -0.9856986
q   1.00       1.3551075       3.0661061
```

18.2.2 Example of MAR(2): multivariate data

Here we show an example of fitting a MAR(2) model. Let's make some simulated data of two realizations of the same AR(2) process:

```
TT <- 50
true.2 <- c(r = 0, b1 = -1.5, b2 = -0.75, q = 1)
temp1 <- arima.sim(n = TT, list(ar = true.2[c("b1", "b2")]),
                   sd = sqrt(true.2["q"]))

temp2 <- arima.sim(n = TT, list(ar = true.2[c("b1", "b2")]),
                   sd = sqrt(true.2["q"]))
sim.mar2 <- rbind(temp1, temp2)
```

Although these are independent time series, we want to fit with a MAR(2) model to allow us to use both datasets together to estimate the AR(2) parameters. We need to set up the model list for the multivariate model (Equation 18.2 and 18.3):

```
Z <- matrix(c(1, 0, 0, 1, 0, 0, 0, 0), 2, 4)
B1 <- matrix(list(0), 2, 2)
diag(B1) <- "b1"
B2 <- matrix(list(0), 2, 2)
diag(B2) <- "b2"
B <- matrix(list(0), 4, 4)
B[1:2, 1:2] <- B1
B[1:2, 3:4] <- B2
B[3:4, 1:2] <- diag(1, 2)
U <- matrix(0, 4, 1)
Q <- matrix(list(0), 4, 4)
Q[1, 1] <- "q"
Q[2, 2] <- "q"
```

```
A <- matrix(0, 2, 1)
R <- matrix(0, 2, 2)
pi <- matrix(c(sim.mar2[, 2], sim.mar2[, 1]), 4, 1)
V <- matrix(0, 4, 4)
model.list.2m <- list(
  Z = Z, B = B, U = U, Q = Q, A = A,
  R = R, x0 = pi, V0 = V, tinitx = 1
)
```

Notice the form of the **Z** matrix:

```
     [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    0    1    0    0
```

It is a 2×2 identity matrix followed by a 2×2 all-zero matrix. The **B** matrix is composed of $\mathbf{B}_1$ and $\mathbf{B}_2$ which are diagonal matrices with b_1 and b_2 respectively on the diagonal.

```
     [,1] [,2] [,3] [,4]
[1,] "b1" 0    "b2" 0
[2,] 0    "b1" 0    "b2"
[3,] 1    0    0    0
[4,] 0    1    0    0
```

We fit the model as usual:

```
mar2 <- MARSS(sim.mar2[, 2:TT], model = model.list.2m)
```

Then we can compare how using two time series improves the fit versus using only one alone:

```
model.list.2$x0 <- matrix(sim.mar2[1, 2:1], 2, 1)
mar2a <- MARSS(sim.mar2[1, 2:TT], model = model.list.2)
model.list.2$x0 <- matrix(sim.mar2[2, 2:1], 2, 1)
mar2b <- MARSS(sim.mar2[2, 2:TT], model = model.list.2)
```

```
    true   est.mar2  est.mar2a  est.mar2b
b1 -1.50 -1.4206301 -0.7209367 -1.3506409
b2 -0.75 -0.7642604 -0.3954671 -0.6953739
q   1.00  0.8986820  3.2098084  1.5552943
```

18.3 MAR(p) models

A MAR(p) model is similar to a MAR(2) except it has lags up to time p:

$$\mathbf{x}_t' = \mathbf{B}_1\mathbf{x}_{t-1}' + \mathbf{B}_2\mathbf{x}_{t-2}' + \cdots + \mathbf{B}_p\mathbf{x}_{t-p}' + \mathbf{u}' + \mathbf{w}_t', \text{ where } \mathbf{w}_t' \sim \text{MVN}(0, \mathbf{Q}')$$

where

$$\mathbf{x}_t = \begin{bmatrix} \mathbf{x}'_t \\ \mathbf{x}'_{t-1} \\ \vdots \\ \mathbf{x}'_{t-p} \end{bmatrix}, \mathbf{B} = \begin{bmatrix} \mathbf{B}_1 & \mathbf{B}_2 & \dots & \mathbf{B}_p \\ \mathbf{I}_m & 0 & \dots & 0 \\ 0 & \mathbf{I}_m & \dots & 0 \\ 0 & 0 & \ddots & \vdots \\ 0 & 0 & \dots & 0 \end{bmatrix}, \mathbf{u} = \begin{bmatrix} \mathbf{u}' \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \mathbf{Q} = \begin{bmatrix} \mathbf{Q}' & 0 & \dots & 0 \\ 0 & 0 & \dots & 0 \\ 0 & 0 & \ddots & \vdots \\ 0 & 0 & \dots & 0 \end{bmatrix} \tag{18.5}$$

Here's an example of fitting a univariate AR(3) in MARSS(1) form. We need more data to estimate an AR(3), so use 100 time steps.

```
TT <- 100
true.3 <- c(r = 0, b1 = -1.5, b2 = -0.75, b3 = .05, q = 1)
temp3 <- arima.sim(
  n = TT, list(ar = true.3[c("b1", "b2", "b3")]),
  sd = sqrt(true.3["q"])
)
sim.ar3 <- matrix(temp3, nrow = 1)
```

We set up the model list for the AR(3) in MARSS(1) form as follows:

```
Z <- matrix(c(1, 0, 0), 1, 3)
B <- matrix(list("b1", 1, 0, "b2", 0, 1, "b3", 0, 0), 3, 3)
U <- matrix(0, 3, 1)
Q <- matrix(list(0), 3, 3)
Q[1, 1] <- "q"
A <- matrix(0, 1, 1)
R <- matrix(0, 1, 1)
pi <- matrix(sim.ar3[3:1], 3, 1)
V <- matrix(0, 3, 3)
model.list.3 <- list(
  Z = Z, B = B, U = U, Q = Q, A = A,
  R = R, x0 = pi, V0 = V, tinitx = 1
)
```

and fit as normal:

```
ar3 <- MARSS(sim.ar3[3:TT], model = model.list.3)
```

The estimates are:

```
print(cbind(
  true = true.3[c("b1", "b2", "b3", "q")],
  estimates.no.miss = coef(ar3, type = "vector")
))
```

```
    true estimates.no.miss
b1 -1.50        -1.5130316
b2 -0.75        -0.6755283
b3  0.05         0.1368458
q   1.00         1.1267684
```

18.4 MARSS(p): models with observation error

We can easily fit MAR(p) processes observed with error using MARSS(p) models, but the difficulty is specifying the initial state condition. $\pi \equiv \mathbf{x}_1$ and thus involves $\mathbf{x}_1$, $\mathbf{x}_0$, ... However, we do not know the variance-covariance structure for these consecutive $\mathbf{x}$. Specifying $\Lambda = 0$ and estimating π often causes the EM algorithm to run into numerical problems. But if we have an abundance of data, fixing π might not overly affect the $\mathbf{B}$ and $\mathbf{Q}$ estimates.

Here is an example where we set π to the mean of the data and set Λ to zero. Why not set Λ equal to a diagonal matrix with large values on the diagonal to approximate a vague prior? The temporally consecutive initial states are definitely not independent. A diagonal matrix would imply independence which will conflict with the process model and means our model would be fundamentally inconsistent with the data (and that usually has bad consequences for estimation).

Create some simulated data:

```
TT <- 1000 # set long
true.2ss <- c(r = .5, b1 = -1.5, b2 = -0.75, q = .1)
temp <- arima.sim(
  n = TT, list(ar = true.2ss[c("b1", "b2")]),
  sd = sqrt(true.2ss["q"])
)
sim.ar <- matrix(temp, nrow = 1)
noise <- rnorm(TT - 1, 0, sqrt(true.2ss["r"]))
noisy.data <- sim.ar[2:TT] + noise
```

Set up the model list for the model in MARSS(1) form:

```
Z <- matrix(c(1, 0), 1, 2)
B <- matrix(list("b1", 1, "b2", 0), 2, 2)
U <- matrix(0, 2, 1)
Q <- matrix(list("q", 0, 0, 0), 2, 2)
A <- matrix(0, 1, 1)
R <- matrix("r")
V <- matrix(0, 2, 2)
pi <- matrix(mean(noisy.data), 2, 1)
model.list.2ss <- list(
  Z = Z, B = B, U = U, Q = Q, A = A,
  R = R, x0 = pi, V0 = V, tinitx = 0
)
```

Fit as usual:

```
ar2ss <- MARSS(noisy.data, model = model.list.2ss)
```

```
Success! abstol and log-log tests passed at 101 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 101 iterations.
Log-likelihood: -1368.796
AIC: 2745.592   AICc: 2745.632

     Estimate
R.r     0.477
B.b1   -1.414
B.b2   -0.685
Q.q     0.140
Initial states (x0) defined at t=0

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

We can compare the results to modeling the data as if there is no observation error, and we see that the assumption of no observation error leads to poor **B** estimates:

```
model.list.2ss.bad <- model.list.2ss
# set R to zero in this model
model.list.2ss.bad$R <- matrix(0)
```

Fit using the model with **R** set to 0:

```
ar2ss2 <- MARSS(noisy.data, model = model.list.2ss.bad)
```

Compare results

```
print(cbind(
  true = true.2ss,
  model.no.error = c(NA, coef(ar2ss2, type = "vector")),
  model.w.error = coef(ar2ss, type = "vector")
))
```

```
    true model.no.error model.w.error
r   0.50             NA     0.4772368
b1 -1.50    -0.52826082    -1.4136279
b2 -0.75     0.03372857    -0.6853180
q   0.10     0.95834464     0.1404334
```

The middle column are the estimates assuming that the data have no observation error and the right column are our estimates with the observation error estimated. Clearly, assuming no observation error when it is present has negative consequences for the **B** and **Q** estimates.

By the way, there is a straight-forward way to deal with the measurement error if you are working with univariate ARMA models and you are only interested in

the AR parameters (the b's). Inclusion of measurement error leads to additional MA components up to lag p (Staudenmayer and Buonaccorsi, 2005). This means that if you are fitting an AR(p) model with measurement error, you can fit a ARMA(p,p) and the measurement error will be absorbed in the p MA components. For the example above, we could estimate the AR parameters for our AR(2) data with measurement error by fitting a ARMA(p,p) model. Here's how we could do that using R 's `arima()` function:

```
arima(noisy.data, order = c(2, 0, 2), include.mean = FALSE)
```

```
Call:
arima(x = noisy.data, order = c(2, 0, 2), include.mean = FALSE)

Coefficients:
          ar1      ar2     ma1     ma2
      -1.4448  -0.6961  0.9504  0.3428
s.e.   0.0593   0.0427  0.0686  0.0482

sigma^2 estimated as 0.9069:  log likelihood = -1368.99,  aic = 2747.99
```

Accounting for the measurement error definitely improves the estimates for the AR component.

18.5 Discussion

Although both MARSS(1) and ARMA(p,p) approaches can be used to deal with AR(p) processes (univariate data) observed with error, our simulations suggest that the MARSS(1) approach is less biased and more precise (Figure 18.1) and that the EM algorithm is working better for this problem. The performance of different approaches depends greatly on the underlying model. We chose AR parameters where both ARMA(p,p) and MARSS(1) approaches work. If we used, for example, $b_1 = 0.8$ and $b_2 = -0.2$, the ARMA(2,2) gives b_1 estimates close to 0 (i.e., wrong) while the MARSS(1) EM approach gives estimates close to the truth (though rather variable). One would want to also check REML approaches for fitting the ARMA(p,p) models since REML has been found to be less biased than ML estimation for this class (Cheang and Reinsel, 2000; Ives et al., 2010). Ives et al. 2010 has R code for REML estimation of ARMA(p,q) models in their appendix.

For multivariate data observed with error, especially multivariate data without a one-to-one relationship to the underlying autoregressive process, an explicit MARSS model will need to be used rather than an ARMA(p,p) model. The time steps required for good parameter estimates are likely to be large; in our simulations, we used 100 for a AR(3) and 1000 for a ARSS(2). Thorough simulation testing should be conducted to determine if the data available are sufficient to allow estimation of the **B** terms at multiple lags.

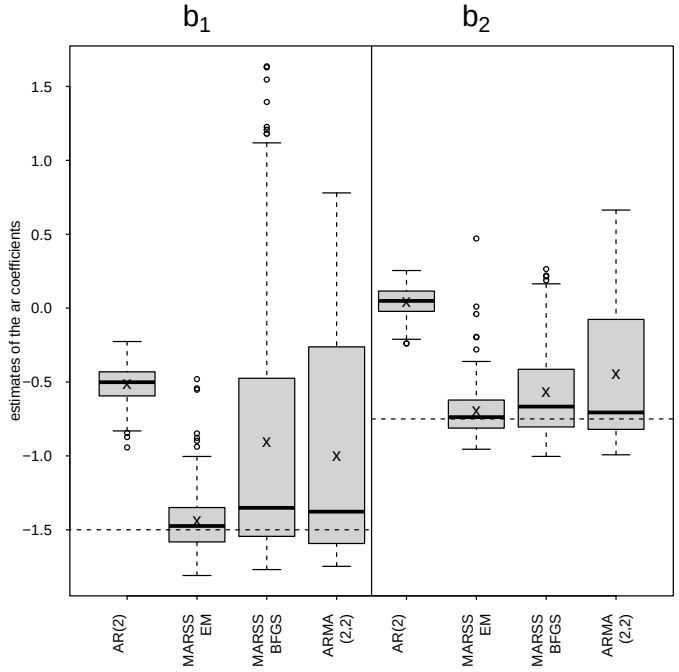

Fig. 18.1. Comparison of the AR parameter estimates using different approaches to model ARSS(2) data (univariate AR(2) data observed with error). Results are from 200 simulations of AR(2) data with 100 time steps. Results are shown for the b_1 and b_2 parameters of the AR process fit with a 1) AR(2) model with no correction for measurement error, 2) MARSS(1) model fit via the EM optimization, 3) MARSS(1) model fit via BFGS optimization (initial conditions not optimized), 4) ARMA(2,2) model fit with the`arima()` function, and 5) AR(2) model fit 2nd differencing with the `arima()` function. The "x" shows the mean of the simulations and the bar in the boxplot is the median. The true values are shown with the dashed horizontal line. The σ^2 for the AR process was 0.1 and the σ^2 for the measurement error was 0.5. The b_1 parameters was -1.5, and b_2 was -0.75.

19

Structural Time Series Models

Structural time series models are linear Gaussian state-space models which decompose the time series into additive random walks for the level, trend and season. These models can be written as a MARSS model. R provides the `StructTS()` function in the {stats} package to fit the level, level plus trend, and level plus trend plus season versions of structural time series models to univariate data.

Here it is shown how to fit structural time series models with `MARSS()` using the same initial conditions assumptions as used in the `StructTS()` function. With `MARSS()`, you are not restricted to univariate time series and you have control over any parameter constraints that you wish to impose. You will see how to fit multivariate structural time series models after the univariate cases are shown.

Required libraries for this chapter:

```
library(MARSS)
library(tidyr)
library(ggplot2)
library(forecast)
```

19.1 Univariate models

19.1.1 Level model

The basic stochastic level model fit by `stats::StructTS()` and using the notation of that function is

$$y_t = m_t + v_t \text{ where } v_t \sim \mathrm{N}(0, \sigma_\varepsilon^2) \tag{19.1}$$

where m is the level and is a random walk:

Type `RShowDoc("Chapter_Structural_TS.R",package="MARSS")` at the R command line to open a file with all the code for the examples in this chapter.

$$m_t = m_{t-1} + w_t \text{ where } w_t \sim \mathrm{N}(0, \sigma_\xi^2) \tag{19.2}$$

The initial conditions assumption used in the `StructTS()` function is the following and this must be used in the `MARSS()` model in order to replicate the `StructTS()` output. The initial condition at $t = 0$ for m is stochastic with fixed mean equal to y_1 and variance equal to 10000 times the variance of the data, denoted s^2.

$$m_0 \sim \mathrm{N}(y_1, 10000s^2) \tag{19.3}$$

Here the model is fit to 20 time steps of tree ring data. `fit1` is the `StructTS()` output, `fit2` is fit with `MARSS()` with parameters fixed at the `StructTS()` estimated values, `fit3` is the model fit with `MARSS()` using BFGS, and `fit4` is the model fit with `MARSS()` using EM. `fit3` and `fit4` are slightly different than `fit1` because the optimization algorithm is a hill-climbing algorithm for all these fits and stops at slightly different points on the likelihood hill.

Fit with `StructTS()`.

```
y <- window(treering, start = 0, end = 20)
fit1 <- StructTS(y, type = "level")
```

Fit with `MARSS()`. We set `control=list(allow.degen=FALSE)` when using the EM algorithm (the default) in order to compare results to the BFGS algorithm used in `StructTS()`. This will not allow variances to go to zero; they may appear that way in the output but that is rounding.

```
vy <- var(y, na.rm = TRUE) / 100
mod.list <- list(
  x0 = matrix(y[1]), U = "zero", tinitx = 0,
  Q = matrix(fit1$coef[1]), R = matrix(fit1$coef[2]),
  V0 = matrix(1e+06 * vy)
)
fit2 <- MARSS(as.vector(y), model = mod.list)
# Now estimate the parameters
mod.list <- list(
  x0 = matrix(y[1]), U = "zero", tinitx = 0, V0 = matrix(1e+06 * vy),
  Q = matrix("s2xi"), R = matrix("s2eps")
)
fit3 <- MARSS(as.vector(y), model = mod.list, method = "BFGS")
fit4 <- MARSS(as.vector(y),
  model = mod.list,
  control = list(allow.degen = FALSE)
)
```

A difference with `StructTS()` is that the reported fitted level (the x state estimate) is the estimate of the state conditioned on the data up to t not T. In the {MARSS} package, the state estimate (in the `states` element of the fitted object) is reported conditioned on all the data (up to T). To compare the outputs, we need to use `MARSSkfss()` to get `xtt` (the estimate of x conditioned on data up to t).

```
fit2$kf <- MARSSkfss(fit2)
fit3$kf <- MARSSkfss(fit3)
fit4$kf <- MARSSkfss(fit4)
df <- data.frame(
  StructTS = fit1$fitted, fit2 = fit2$kf$xtt[1, ],
  fit.bfgs = fit3$kf$xtt[1, ], fit.em = fit4$kf$xtt[1, ]
)
head(df)
```

```
    level     fit2 fit.bfgs   fit.em
1 1.265000 1.265000 1.265000 1.265000
2 1.132475 1.132475 1.132546 1.136460
3 1.200246 1.200246 1.200210 1.198366
4 1.457141 1.457141 1.457002 1.449322
5 1.448101 1.448101 1.448104 1.448054
6 1.123611 1.123611 1.123786 1.133389
```

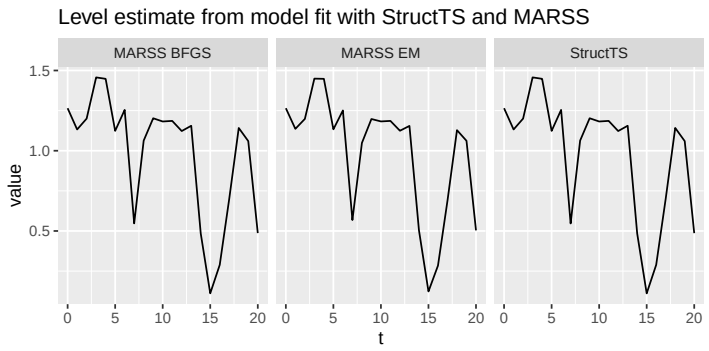

Fig. 19.1. Comparison of the level estimates for the stochastic level model.

19.1.2 Level plus trend model

The basic stochastic level plus trend model fit by `stats::StructTS()` is

$$y_t = m_t + v_t \text{ where } v_t \sim \mathrm{N}(0, \sigma_\varepsilon^2) \tag{19.4}$$

where m and n are the stochastic level and trend. $m_t = m_{t-1} + n_{t-1} + w_t$ and in matrix form this is

$$\begin{bmatrix} m \\ n \end{bmatrix}_t = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} m \\ n \end{bmatrix}_{t-1} + \mathbf{w}_t \text{ where } \mathbf{w}_t \sim \mathrm{MVN}\left(0, \begin{bmatrix} \sigma_\xi^2 & 0 \\ 0 & \sigma_\zeta^2 \end{bmatrix}\right) \tag{19.5}$$

The initial conditions assumption used in `StructTS()` for this model is the following where s^2 is the variance in the data ($\text{var}(y)$):

$$\begin{bmatrix} m \\ n \end{bmatrix}_0 \sim \text{MVN}\left(\begin{bmatrix} y_1 \\ 0 \end{bmatrix}, \begin{bmatrix} 10000s^2 & 10000s^2 \\ 10000s^2 & 10000s^2 \end{bmatrix} \right) \tag{19.6}$$

Because `MARSS()` does an inversion of the initial variance matrix as part of code to force positive definite matrices and deal with degenerate models with 0s on diagonals of **Q** or **R**, the initial conditions variance used in `StructTS()` needs to be made positive definite for `MARSS()`. This is done by adding a small value (1e-10) to the diagonal as shown in the `mod.list` used for `fit3` and `fit4`.

This model will be illustrated with the UKgas data set. For the tree ring data, the trend variance estimate is 0 and that will not illustrate a stochastic trend. The `subset.ts()` function in the {forecast} package is used to subset just the 2nd quarter data.

Fit with `StructTS()`.

```
y <- log10(forecast:::subset.ts(UKgas, quarter = 2))
fit1 <- StructTS(y, type = "trend")
```

Fit with `MARSS()`. First we will create a MARSS model with the same parameters as the `StructTS` fit.

```
vy <- var(y, na.rm = TRUE) / 100
B <- matrix(c(1, 0, 1, 1), 2, 2)
Z <- matrix(c(1, 0), 1, 2)
# fitx parameters at fit1 values
mod.list <- list(
  x0 = matrix(c(y[1], 0), 2, 1), U = "zero", tinitx = 0,
  Q = diag(fit1$coef[1:2]), R = matrix(fit1$coef[3]),
  V0 = matrix(1e+06 * vy, 2, 2), Z = Z, B = B
)
fit2 <- MARSS(as.vector(y),
  model = mod.list, fit = FALSE,
  control = list(trace = -1)
)
fit2$par <- fit2$start # otherwise par is NULL since fit=FALSE
```

Now estimate the parameters with `MARSS()`.

```
mod.list <- list(
  x0 = matrix(c(y[1], 0), 2, 1), U = "zero", tinitx = 0,
  Q = ldiag(c("s2xi", "s2zeta")), R = matrix("s2eps"),
  V0 = matrix(1e+06 * vy, 2, 2) + diag(1e-10, 2), Z = Z, B = B
)
fit3 <- MARSS(as.vector(y), model = mod.list, method = "BFGS")
fit4 <- MARSS(as.vector(y),
  model = mod.list,
```

```
  control = list(allow.degen = FALSE)
)
```

Figure 19.2 shows the comparisons for the full level and trend estimates. The EM algorithm would need a lower tolerance to get closer to the maximum likelihood parameter values.

```
fit2$kf <- MARSSkfss(fit2)
fit3$kf <- MARSSkfss(fit3)
fit4$kf <- MARSSkfss(fit4)
data.frame(
  StructTS = fit1$fitted[, 2], fit2 = fit2$kf$xtt[2, ],
  fit.bfgs = fit3$kf$xtt[2, ], fit.em = fit4$kf$xtt[2, ]
)[1:5, ]
```

```
      StructTS         fit2     fit.bfgs       fit.em
1  0.000000000  0.000000000  0.000000000  0.000000000
2 -0.003519920 -0.003519920 -0.003518509 -0.003551063
3  0.005201072  0.005201072  0.005203550  0.005275918
4  0.006762807  0.006762807  0.006759700  0.006912919
5  0.007290557  0.007290557  0.007286018  0.007453794
```

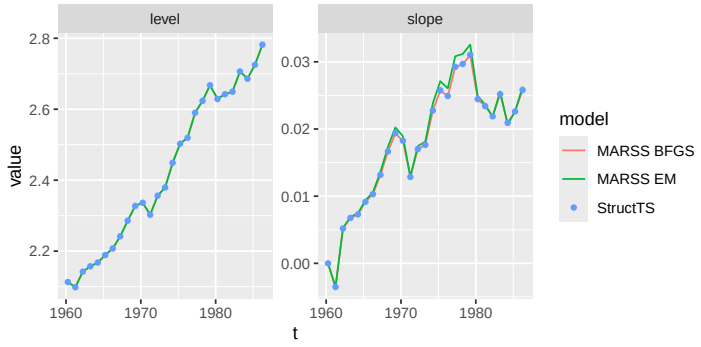

Fig. 19.2. Comparison of the level and trend estimates for the stochastic level plus trend model.

19.1.3 Seasonal or BSM model

The seasonal model fit by `StructTS()` is the level plus trend model with an additional seasonal component s_t. The m_t model is the same as for the level plus trend model.

$$y_t = m_t + s_t + v_t \text{ where } v_t \sim \mathrm{N}(0, \sigma_\varepsilon^2) \tag{19.7}$$

where

$$s_t = -s_{t-1} - \cdots - s_{t-f+1} + v_t \text{ where } v_t \sim \mathrm{N}(0, \sigma_w^2) \tag{19.8}$$

f is the frequency of the seasonality. For quarterly data, $f = 4$ and the s_t model is

$$s_t = -s_{t-1} - s_{t-2} - s_{t-3} + v_t \tag{19.9}$$

Written in MARSS form, the model for a quarterly seasonality is the following. s is the seasonality term while s_1 and s_2 are just keeping track of s_{t-1} and s_{t-2}.

$$y_t = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} m \\ n \\ s \\ s_1 \\ s_2 \end{bmatrix}_t + v_t \tag{19.10}$$

and the x model is

$$\begin{bmatrix} m \\ n \\ s \\ s_1 \\ s_2 \end{bmatrix}_t = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & -1 & -1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} m \\ n \\ s \\ s_1 \\ s_2 \end{bmatrix}_{t-1} + \mathbf{w}_t \tag{19.11}$$

where

$$\mathbf{w}_t \sim \mathrm{MVN}\left(0, \begin{bmatrix} \sigma_\xi^2 & 0 & 0 & 0 & 0 \\ 0 & \sigma_\zeta^2 & 0 & 0 & 0 \\ 0 & 0 & \sigma_w^2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}\right) \tag{19.12}$$

The initial conditions assumption is the following where again s^2 is the variance in the data ($\mathrm{var}(y)$):

$$\begin{bmatrix} m \\ n \\ s \\ s_1 \\ s_2 \end{bmatrix}_0 \sim \mathrm{MVN}\left(\begin{bmatrix} y_1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 10^4 s^2 & 10^4 s^2 & 10^4 s^2 & 10^4 s^2 & 10^4 s^2 \\ 10^4 s^2 & 10^4 s^2 & 10^4 s^2 & 10^4 s^2 & 10^4 s^2 \\ 10^4 s^2 & 10^4 s^2 & 10^4 s^2 & 10^4 s^2 & 10^4 s^2 \\ 10^4 s^2 & 10^4 s^2 & 10^4 s^2 & 10^4 s^2 & 10^4 s^2 \\ 10^4 s^2 & 10^4 s^2 & 10^4 s^2 & 10^4 s^2 & 10^4 s^2 \end{bmatrix}\right) \tag{19.13}$$

Let's see an example with the UK gas data set used in the help page `?StructTS`.

```
y <- log10(UKgas)
fit1 <- StructTS(y, type = "BSM")
```

To make **B** for `MARSS()`, we write a little helper function. `nf` is the frequency.

```
makeB <- function(nf) {
  B <- matrix(0, nf + 1L, nf + 1L)
  B[1L:2L, 1L:2L] <- c(1, 0, 1, 1)
```

```
  B[3L, ] <- c(0, 0, rep(-1, nf - 1L))
  if (nf >= 3L) {
    ind <- 3:nf
    B[cbind(ind + 1L, ind)] <- 1
  }
  return(B)
}
```

Now we can fit with `MARSS()`.

```
nf <- frequency(y)
vy <- var(y) / 100
B <- makeB(nf)
Z <- matrix(c(1, 0, 1, rep(0, nf - 2L)), 1, nf + 1)
Q <- ldiag(list("s2xi", "s2zeta", "s2w", 0, 0))
R <- matrix("s2eps")
V0 <- matrix(1e+06 * vy, nf + 1, nf + 1) + diag(1e-10, nf + 1)
mod.list <- list(
  x0 = matrix(c(y[1], rep(0, nf)), ncol = 1),
  U = "zero", A = "zero", tinitx = 0,
  Q = Q, R = R, V0 = V0, Z = Z, B = B
)
fit3 <- MARSS(as.vector(y), model = mod.list, method = "BFGS")
fit4 <- MARSS(as.vector(y),
  model = mod.list,
  control = list(allow.degen = FALSE)
)
fit4$kf <- MARSSkfss(fit4)
fit3$kf <- MARSSkfss(fit3)
```

Figure 19.3 shows the comparisons.

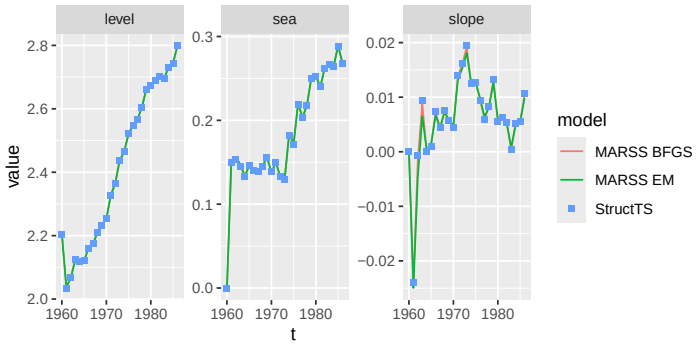

Fig. 19.3. Comparison of the level, trend and season estimates for the BSM model.

19.1.4 Forecasting

Forecasts can be made with the `predict()` function or the `forecast()` function. Here we will use the BSM model fits to illustrate forecasting.

```
y <- log10(UKgas)
fit1 <- StructTS(y, type = "BSM")
nf <- frequency(y)
vy <- var(y) / 100
B <- makeB(nf) # defined in the BSM section above
Z <- matrix(c(1, 0, 1, rep(0, nf - 2L)), 1, nf + 1)
V0 <- matrix(1e+06 * vy, nf + 1, nf + 1) + diag(1e-10, nf + 1)
mod.list <- list(
  x0 = matrix(c(y[1], rep(0, nf)), ncol = 1),
  U = "zero", A = "zero", tinitx = 0,
  Q = diag(c(fit1$coef[1:3], 0, 0)),
  R = matrix(fit1$coef[4]),
  V0 = V0, Z = Z, B = B
)
fit2 <- MARSS(as.vector(y), model = mod.list)
```

`fit1` and `fit2` are exactly the same since `fit2` used the `fit1` estimated parameters.

`stats::predict.StructTS()` is only for forecasting and takes the fit and `n.ahead` as arguments. It returns a list with the forecasts in `pred` and a ts object and their standard errors in `se`.

```
fr1 <- predict(fit1, n.ahead = 5)
fr1
```

```
$pred
        Qtr1     Qtr2     Qtr3     Qtr4
1987 3.130126 2.831481 2.580969 2.947872
1988 3.177549

$se
           Qtr1       Qtr2       Qtr3       Qtr4
1987 0.05450291 0.05465621 0.05773393 0.06022965
1988 0.08818050
```

The `MARSS::predict.marssMLE()` does both predicting within the data (similar to other `predict` methods) and will forecast if `n.ahead` is passed in. It returns a list with the predictions and forecasts in `pred` as a data frame in long form (suitable for `ggplot()` calls). The standard errors and intervals (confidence or prediction) are included in `pred`. The standard error is not printed but is in the `pred` data frame.

```
fr2 <- predict(fit2, n.ahead = 5, interval = "prediction")
fr2
```

```
    .rownames   t estimate    Lo 80    Hi 80    Lo 95
109        Y1 109 3.130126 3.060278 3.199975 3.023303
110        Y1 110 2.831481 2.761436 2.901526 2.724357
111        Y1 111 2.580969 2.506980 2.654958 2.467813
112        Y1 112 2.947872 2.870685 3.025059 2.829824
113        Y1 113 3.177549 3.064541 3.290557 3.004718
       Hi 95
109 3.236950
110 2.938605
111 2.694125
112 3.065920
113 3.350380
```

The estimates are the same. `ft` is the time steps associated with the forecast.

```
rbind(
  pred1 = fr1$pred, pred2 = fr2$pred$estimate[fr2$ft],
  se1 = fr1$se, se2 = fr2$pred$se[fr2$ft]
)
```

```
            [,1]       [,2]       [,3]       [,4]      [,5]
pred1 3.13012626 2.83148104 2.58096900 2.94787202 3.1775490
pred2 3.13012626 2.83148104 2.58096900 2.94787202 3.1775490
se1   0.05450291 0.05465621 0.05773393 0.06022965 0.0881805
se2   0.05450291 0.05465621 0.05773393 0.06022965 0.0881805
```

If we use the `forecast:::forecast.StructTS()` function instead of `predict()`, we get an object that can be plotted since the {forecast} package has a plot method for StructTS objects. The {MARSS} package has a plot method for marssPredict objects returned by `predict()` and `forecast()` functions used with marssMLE objects. The `forecast()` function can be called with `forecast::forecast()` if you have the {forecast} package installed or `forecast.marssMLE()` if not.

The plots from the StructTS and marssMLE objects are similar though they have slightly different formats.

19.1.5 Fitted values

`fitted(x)` applied to a StructTS object will return the expected value of $\mathbf{X}_t$ (but only the first 3 states) conditioned on the data up to time t. It is returned as a ts or mts object depending if there is one state ("level") or multiple ("trend" or "BSM"). For the BSM model, plotting this will show the decomposed time series with the estimated level m_t, slope or trend n_t and season s_t terms. In the {MARSS} package, the estimated states conditioned on the data up to time t is returned with `tsSmooth(x, type="xtt")`. The function `fitted()` in the {MARSS} package has the more typical meaning of *fitted* for a statistical model (model prediction of **y** or **x**).

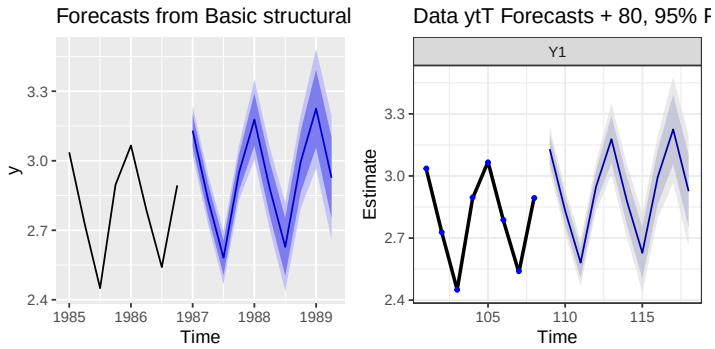

Fig. 19.4. Comparison of the forecast plots.

```
fitted1 <- fitted(fit1)
plot(fitted1)
```

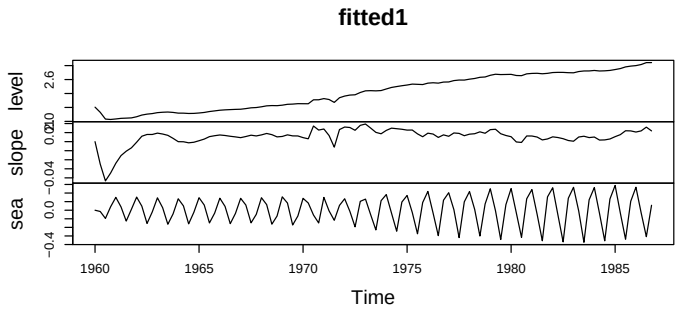

Fig. 19.5. Output from a plot of a fitted StructTS object.

```
fitted2 <- tsSmooth(fit2, type = "xtt")
fitted2 <- subset(fitted2, .rownames %in% c("X1", "X2", "X3"))
```

This is a data frame in long-form which we can plot with `ggplot()`. Alternatively instead of `tsSmooth(x, type="xtt")`, we can use `MARSSkfss(x)$xtt` to return the state estimates as a matrix. Converting the matrix to a ts object makes it easier to plot.

To output the model fitted value for y, we add the level and season states together if using `StructTS()` because for the BSM model, the model for y_t is $m_t + s_t$. With {MARSS}, this would be output with `fitted(x, type="ytt")`. Alternative we could add the m_t and s_t states from the {MARSS} output, but `fitted(x, type="ytt")` allows you to easily compute this for cases that are more complex with a non-identity $\mathbf{Z}$, non-zero $\mathbf{a}$ and covariates $\mathbf{d}$.

```
ggplot(fitted2, aes(x = t, y = .estimate)) +
  geom_line() +
  facet_wrap(~.rownames, ncol = 1, scale = "free_y")
```

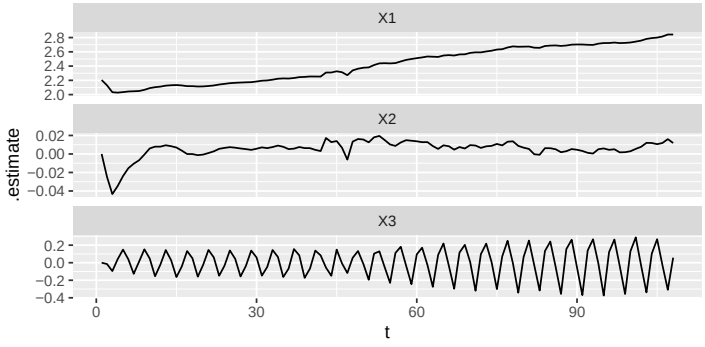

Fig. 19.6. Output from a plot of the states from the marssMLE object using tsSmooth() output.

```
fitted3 <- MARSSkfss(fit2)$xtt
fitted3 <- ts(t(fitted3[1:3, ]))
plot(fitted3)
```

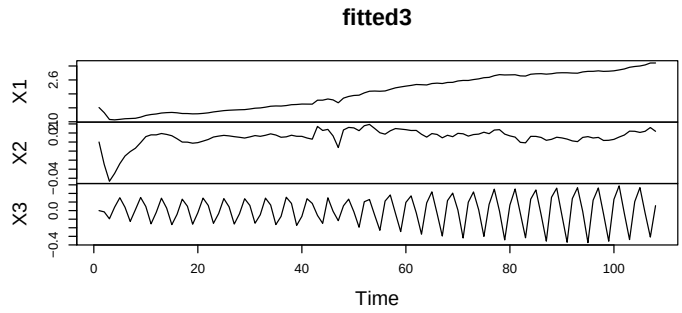

Fig. 19.7. Output from a plot of states from a marssMLE object.

19.1.6 Residuals

`residuals(x)` applied to a StructTS object will return the difference between y_t and the expected value of Y_t conditioned on the data up to time t. The residuals are returned as a ts object. The residuals are standardized, i.e., divided by the square root of the conditional variance of residuals (conditioned on the data). Note the conditional variance of the residuals is not `var(resids)`; see `?MARSSresiduals` for a discussion of how the conditional variance of state-space residuals is computed.

```
resids1 <- residuals(fit1)
```

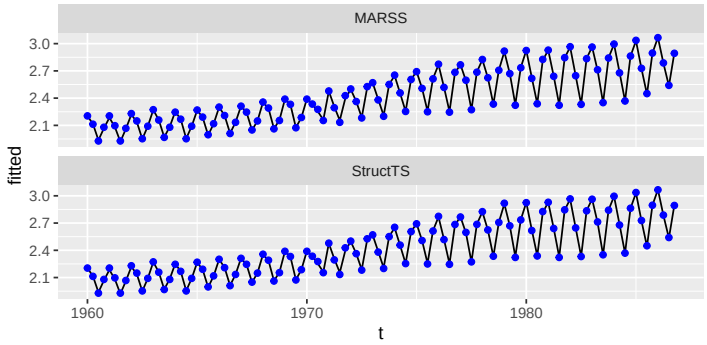

Fig. 19.8. Data and model fitted values.

In the {MARSS} package, the `residuals()` function will return the model residuals conditioned on all the data, data up to time $t-1$ or up to time t. To replicate the behavior for StructTS objects, we need to use conditioning up to time t which is `type="tt"` and we need to specify marginal standardization.

```
resids2 <- residuals(fit2, type = "tt", standardization = "marginal")
```

The output is a data frame in long-form which we can plot with `ggplot()`.

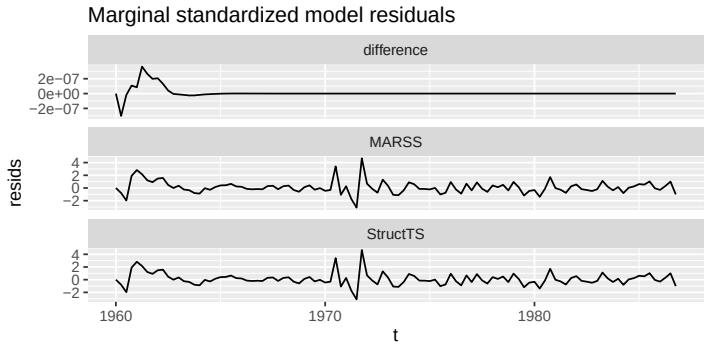

Fig. 19.9. The initial difference in the residuals is due to the small value added to the diagonal of the initial condition variance-covariance matrix to allow `MARSS()` to fit this model.

19.2 Multivariate models

The {MARSS} package allows one to fit multivariate structural equation models. In this section, we will use the level plus trend model as the example, however the

approaches work for all the structural models. The focus for this example will be estimating a changing trend using multiple observation time series.

19.2.1 Multiple observations of same process

The basic stochastic level plus trend model is

$$y_t = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} m \\ n \end{bmatrix}_t + v_t \tag{19.14}$$

$$\begin{bmatrix} m \\ n \end{bmatrix}_t = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} m \\ n \end{bmatrix}_{t-1} + \mathbf{w}_t \text{ where } \mathbf{w}_t \sim \text{MVN}\left(0, \begin{bmatrix} \sigma_\xi^2 & 0 \\ 0 & \sigma_\zeta^2 \end{bmatrix}\right) \tag{19.15}$$

Now imagine that there are three independent y_t observations of the m_t process. The observation model then becomes.

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}_t = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} m \\ n \end{bmatrix}_t + \mathbf{v}_t \text{ where } \mathbf{v}_t \sim \text{MVN}\left(0, \begin{bmatrix} \sigma_\varepsilon^2 & 0 & 0 \\ 0 & \sigma_\varepsilon^2 & 0 \\ 0 & 0 & \sigma_\varepsilon^2 \end{bmatrix}\right) \tag{19.16}$$

The initial conditions assumption used in the `StructTS()` for this model is the following and we will keep that with the addition of a small amount to the diagonal to make the initial condition matrix positive definite.

$$\begin{bmatrix} m \\ n \end{bmatrix}_0 \sim \text{MVN}\left(\begin{bmatrix} y_1 \\ 0 \end{bmatrix}, \begin{bmatrix} 10000s^2 + 1e-10 & 10000s^2 \\ 10000s^2 & 10000s^2 + 1e-10 \end{bmatrix}\right) \tag{19.17}$$

Simulated data

We will model a simulated random walk with level and an abrupt trend change. We will generate observations that are the level + substantial error. We will also insert 50% missing values into the **y** to illustrate how the method will deal with missing values. Figure 19.10 shows the simulated data. We will fit to the points. The line is the true level.

```
set.seed(100)
TT <- 60
t <- 1:TT
q <- 0.01
r <- 0.01
trend <- 0.2 * sin((1:TT) / 4)
level <- cumsum(rnorm(TT, trend, sqrt(q)))
# Simulated data
n <- 5
miss.percent <- 0.5
ym <- matrix(1, n, 1) %*% level + matrix(rnorm(TT * n, 0, sqrt(r * 100)), n, TT
ym[sample(n * TT, miss.percent * n * TT)] <- NA
```

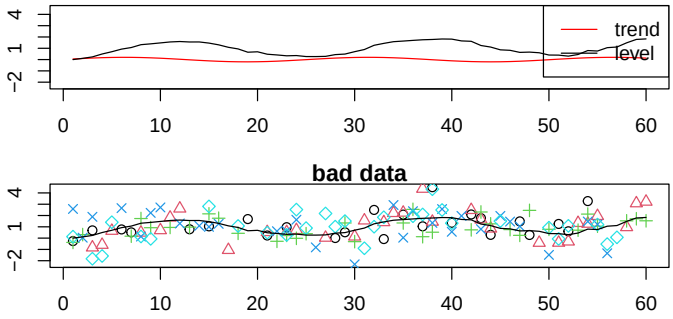

Fig. 19.10. Observations (points) and truth (line).

Model set-up

Set up the **x** part of the equation for `MARSS()`. This part does not change from the univariate case except that we will estimate the initial condition for the level and will treat the variance for the level as known (at the true value). It can be hard to separate the variances with large observation error. We will assume that we know something about the level process and it is the changing trend that we want to estimate.

```
vy <- var(y, na.rm = TRUE) / 100
mod.list.x <- list(
  x0 = matrix(list("x0", 0), nrow = 2), tinitx = 1,
  V0 = matrix(1e+06 * vy, 2, 2) + diag(1e-10, 2),
  Q = ldiag(list(q, "qt")),
  B = matrix(c(1, 0, 1, 1), 2, 2),
  U = "zero"
)
```

Next we set up the **y** part of the equation. This is the part that changes. We will assume that the observations are independent with the same bias (i.e., expected value of each y is the same). We will relax this assumption later.

```
mod.list.y <- list(
  A = "zero",
  R = "diagonal and equal"
)
```

Fit model

Estimate the level and trend from one of the simulated observation time series:

```
Z <- matrix(c(1, 0), 1, 2, byrow = TRUE)
mod.list <- c(mod.list.x, mod.list.y, list(Z = Z))
fitu <- MARSS(ym[1, ], model = mod.list, method = "BFGS", inits = list(x0 = 0))
```

Now estimate the parameters with all the time series.

```
Z <- matrix(c(1, 0), n, 2, byrow = TRUE)
mod.list <- c(mod.list.x, mod.list.y, list(Z = Z))
fitm <- MARSS(ym, model = mod.list, method = "BFGS", inits = list(x0 = 0))
```

Compare trend estimate to truth

Our objective is to estimate the level and trend states (Figure 19.11). In this example, the multivariate model is able to estimate the trend variance unlike when we fit to only one time series (the flat line in the trend plot). But if we increase the error added or the missing values, the ability to estimate the trend variance would disappear.

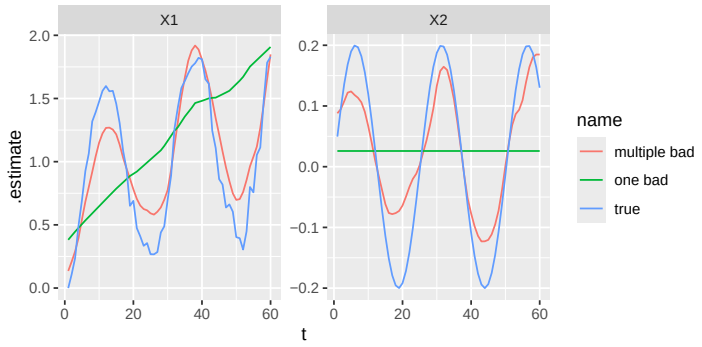

Fig. 19.11. State estimates from the one bad time series versus multivariate bad time series.

19.2.2 Covariate affects observations

We can add a known covariate that affects the observations. In this case, it is a step function representing a before-after effect. We will have it affect only the first few observation time series and the effect will be different for each series (Figure 19.12).

We fit by passing the covariate into `d` (because it affects the observations not the process). The estimated versus true effects are shown in Figure 19.13 and the estimated trend is in Figure 19.14.

```
Z <- matrix(c(1, 0), n, 2, byrow = TRUE)
mod.list <- c(mod.list.x, mod.list.y, list(Z = Z, d = covariate))
fitmc <- MARSS(ymc, model = mod.list, method = "BFGS", inits = list(x0 = 0))
```

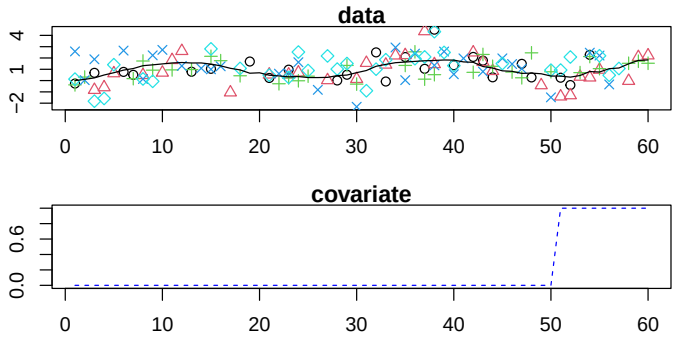

Fig. 19.12. Observations (points) of the true data (line) with covariate effect plus error and missing values added. It affects some time series positively and others negatively.

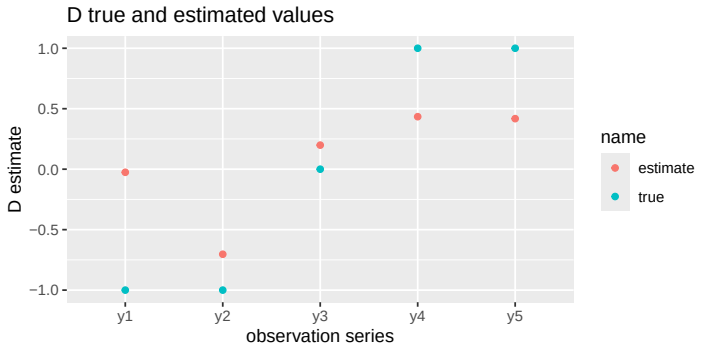

Fig. 19.13. Estimate of the effect of the covariate.

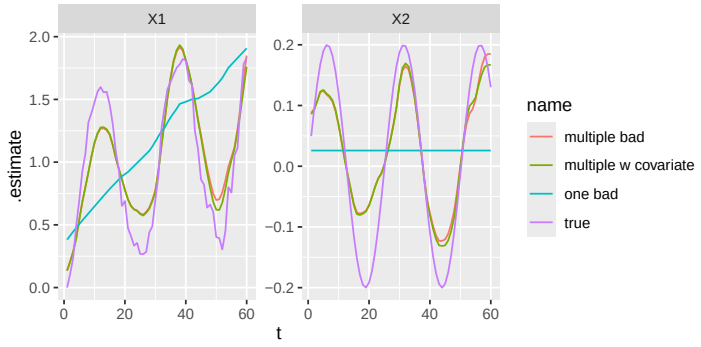

Fig. 19.14. State estimates from the univariate good data, multivariate bad data, and multivariate with a covariate.

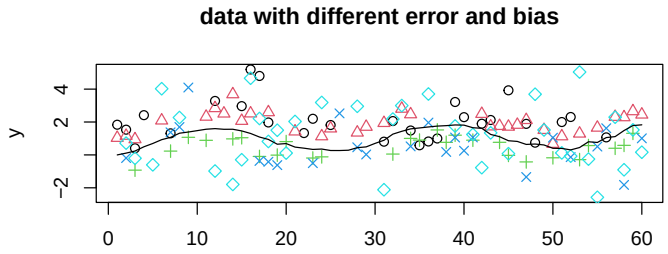

Fig. 19.15. Observations (points) with differing error and bias added.

19.2.3 Observations with bias and different errors

Our observations may have different (unknown) levels of observation error and be biased relative to each other (Figure 19.15).

We fit by changing the **R** and **a** specifications. The **R** estimates are shown in Figure 19.16.

```
Z <- matrix(c(1, 0), n, 2, byrow = TRUE)
mod.list <- c(mod.list.x, list(Z = Z, R = "diagonal and unequal", A = "scaling"
fitm2 <- MARSS(ym2, model = mod.list, method = "BFGS", inits = list(x0 = 0))
```

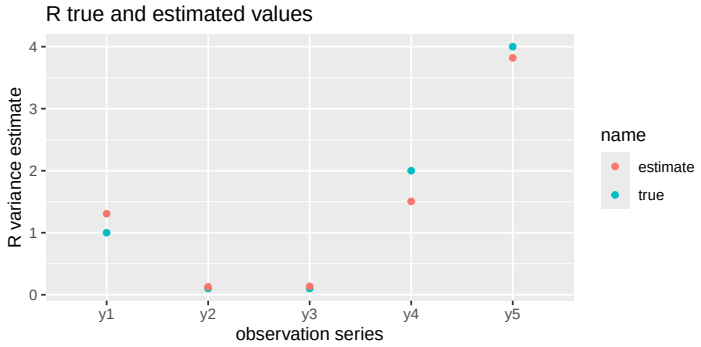

Fig. 19.16. Estimate of the observation variances.

The level will be scaled up or down to fit the first observation time series. We have to set one of the **a** to 0 and by default, `MARSS()` sets the first one to zero. The estimates are in Figure 19.17.

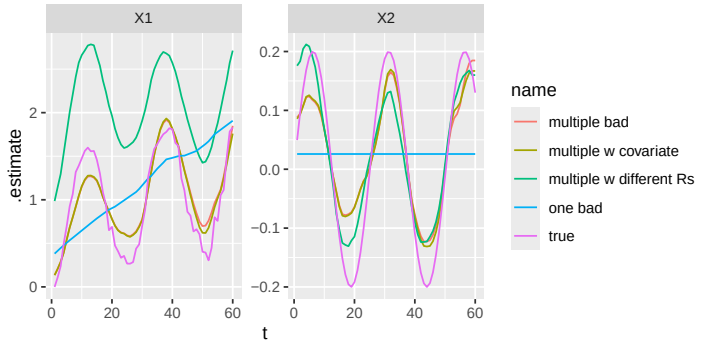

Fig. 19.17. State estimates. The level for the model with bias will be shifted up or down. This is not an error but a feature of having to scale to one of the time series and by default, the first is chosen.

19.2.4 Indepenent realizations of the same process

In the last section, we had multiple observations of the same process. We can also have multiple realizations of independent processes with the same variance values. In this example, we assume that the level is an independent observation of a shared trend.

$$\begin{bmatrix} m_1 \\ m_2 \\ n \end{bmatrix}_t = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} m_1 \\ m_2 \\ n \end{bmatrix}_{t-1} + \mathbf{w}_t \text{ where } \mathbf{w}_t \sim \text{MVN}\left(0, \begin{bmatrix} \sigma_\xi^2 & 0 & 0 \\ 0 & \sigma_\xi^2 & 0 \\ 0 & 0 & \sigma_\zeta^2 \end{bmatrix}\right) \tag{19.18}$$

Each is observed by an independent y_t observation.

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix}_t = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} m_1 \\ m_2 \\ n \end{bmatrix}_t + \mathbf{v}_t \text{ where } \mathbf{v}_t \sim \text{MVN}\left(0, \begin{bmatrix} \sigma_\varepsilon^2 & 0 \\ 0 & \sigma_\varepsilon^2 \end{bmatrix}\right) \tag{19.19}$$

Simulated data

We simulate data as in the last section but simulate two levels and trends. Figure 19.18 shows the simulated data.

```
set.seed(100)
TT <- 60
t <- 1:TT
q <- 0.5
qt <- 0.01
r <- 0.1
```

```
b <- 0.5
trend <- 0.2 * sin((1:TT) / 4)
level1 <- cumsum(rnorm(TT, trend, sqrt(q)))
level2 <- cumsum(rnorm(TT, trend, sqrt(q)))
# Simulated data
ym <- rbind(level1, level2) + matrix(rnorm(TT * 2, 0, sqrt(r)), 2, TT)
```

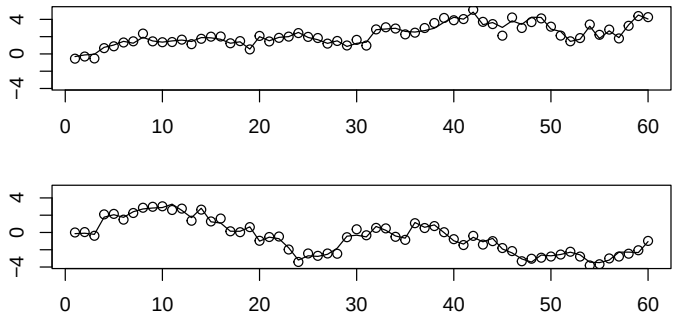

Fig. 19.18. Observations (points) and truth (line). The levels (line) have the same variance but are independent. The level processes share a trend, i.e., $m_{1,t} = m_{1,t-1} + n_{t-1}$ and $m_{2,t} = m_{2,t-1} + n_{t-1}$.

Fit models

Estimate the level and trend from each observation time series alone:

```
vy <- var(y, na.rm = TRUE) / 100
Z <- matrix(c(1, 0), 1, 2)
mod.list.x <- list(
  x0 = matrix(list("x0", 0), nrow = 2), tinitx = 1,
  V0 = matrix(1e+06 * vy, 2, 2) + diag(1e-10, 2),
  Q = ldiag(list(q, "qt")),
  B = matrix(c(1, 0, 1, 1), 2, 2),
  U = "zero"
)
mod.list <- c(mod.list.x, mod.list.y, list(Z = Z))
fitm1 <- MARSS(ym[1, ], model = mod.list, method = "BFGS", inits = list(x0 = 0)
fitm2 <- MARSS(ym[2, ], model = mod.list, method = "BFGS", inits = list(x0 = 0)
```

Estimate the level and trend from the two simulated observation time series together:

```
Z <- matrix(c(1, 0, 0, 0, 1, 0), 2, 3, byrow = TRUE)
m <- 3
mod.list.x <- list(
  x0 = matrix(list("x0.1", "x0.2", 0), nrow = m), tinitx = 1,
  V0 = matrix(1e+06 * vy, m, m) + diag(1e-10, m),
  Q = ldiag(list("q", "q", "qt")),
  B = matrix(c(1, 0, 1, 0, 1, 1, 0, 0, 1), m, m, byrow = TRUE),
  U = "zero"
)
mod.list <- c(mod.list.x, mod.list.y, list(Z = Z))
fitm3 <- MARSS(ym, model = mod.list, method = "BFGS", inits = list(x0 = 0))
```

Compare state estimates

The two time series alone are not able to estimate the trend. Both put all the variation in the data into the level state.

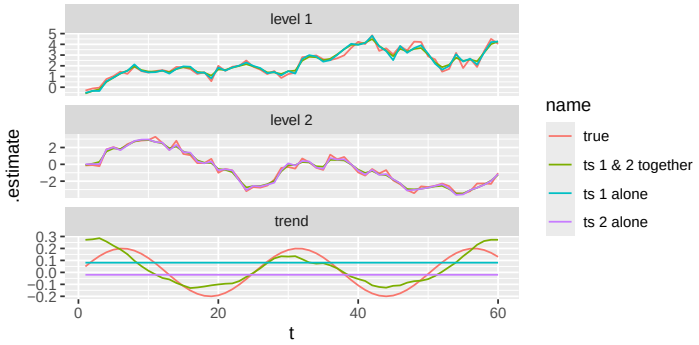

Fig. 19.19. Common trend and separate level estimates.

19.3 Summary

This chapter illustrates how to fit the structural equation models fit by the `StructTS()` function in R . The initial conditions used in that function were used here, however those initial conditions should not be assumed to be the best choice. The default behavior in the {MARSS} package is to treat $\mathbf{x}_0$ as an estimated parameter with initial conditions variance matrix set to all 0. Structural time series models can be a challenge for the EM algorithm and for most examples the BFGS algorithm was used. If using EM, the algorithm will need to be run longer to achieve the maximum likelihood.

The {MARSS} package allows you to fit multivariate structural time series models with a flexible data structure and flexible relationships between the data. It allows you to include covariates and model intervention effects. You can model multiple observations of the same process or different observations of independent processes that share some or all parameter values. You can also model cases where the trend (or seasonality) is shared across processes but not the levels. The ability to use multiple data sets can improve estimation and allow you to estimate an underlying process which might remain hidden if only one data set were used for estimation.

20

Comparison to the {KFAS} Package

The {MARSS} package uses the Kalman filter and smoother in the {KFAS} package (KFAS: Kalman Filter and Smoother for Exponential Family State Space Models) (Helske, 2017) which implements the more stable filter and smoother algorithm by Koopman and Durbin (2000); Durbin and Koopman (2012). The {KFAS} package also provides filtering and smoothing for the general exponential class for the observation errors, e.g., Gaussian, Poisson, binomial, negative binomial, and gamma distributions.

This chapter compares the {KFAS} versus {MARSS} functions for the filter and smoother, fitted values, residuals and predictions for state-space models. Understanding the relationship between the package functions can help understand the state-space outputs. State-space output is complex because there are two processes (state and observation), three possible data conditionings (1 to $t-1$, 1 to t, and 1 to T where T is the last time step), and conditional fitted values versus conditional expected values which the conditional expectation of the right side of the process equation without or with the error term.

This chapter uses the following packages:

```
library(MARSS)
library(KFAS)
library(ggplot2) # plotting
library(tidyr) # data frame manipulation
```

20.1 Nile River example

This is the Nile River example in Durbin and Koopman (2012) and shown in Chapter 12 on structural breaks. This model is

Type `RShowDoc("Chapter_KFAS.R",package="MARSS")` at the R command line to open a file with all the code for the examples in this chapter.

$$\begin{aligned} x_t &= x_{t-1} + w_t \text{ where } w_t \sim \mathrm{N}(0,q) \\ y_t &= x_t + v_t \text{ where } v_t \sim \mathrm{N}(0,r) \end{aligned} \tag{20.1}$$

20.1.1 Fitting models

`KFAS::SSModel()` sets up the {KFAS} model which will be passed to the fitting functions. `KFAS::SSMtrend(degree = 1)` designates a local level model. `KFAS::fitSSM()` fits the model.

```
model_Nile <- SSModel(Nile ~ SSMtrend(
  degree = 1,
  Q = list(matrix(NA))
),
H = matrix(NA)
)
kinits <- c(log(var(Nile)), log(var(Nile)))
fit_kfas_default <- fitSSM(model_Nile, kinits, method = "BFGS")
```

{KFAS} uses a stochastic prior on the initial condition and the fitting function does not estimate x_0. By default, a diffuse prior on x_0 is used. The default behavior for {MARSS}, in contrast, is to estimate x_0 as a parameter and fix V_0^0 (the conditional variance of x_0) to 0. This will lead to small differences between the fits. The EM algorithm in {MARSS} does not implement a true diffuse prior but we can specify a stochastic prior to mimic a {KFAS} fit.

We will set a stochastic prior on x_1 with a mean of 0 and variance of 1000 by changing `P1`, `P1inf`, and `a1` in the {KFAS} model. Setting `P1inf` to 0, turns off the diffuse prior.

```
model_Nile_stoch <- model_Nile
model_Nile_stoch$a1[1, 1] <- 0
model_Nile_stoch$P1[1, 1] <- 1000
model_Nile_stoch$P1inf[1, 1] <- 0
kinits <- c(log(var(Nile)), log(var(Nile)))
fit_kfas_stoch <- fitSSM(model_Nile_stoch, kinits, method = "BFGS")
kfs_kfas_stoch <- KFS(fit_kfas_stoch$model)
```

With MARSS, the model is specified as:

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

The default initial condition in the {MARSS} package is to estimate x_1 as a parameter (and set V_1 to zero). This default behavior prevents prior information about the covariance structure of the states from affecting the estimates, though for some

models, the initial conditions estimation is not well defined (in which case setting a stochastic prior is helpful).

We will fit with the EM and BFGS algorithm in the {MARSS} package. We will start the BFGS algorithm at the same initial conditions used in our {KFAS} fitting call, although this isn't quite the same because {MARSS} and {KFAS} are using different approaches to ensure that the variances stay positive-definite during the BFGS maximization steps.

```
dat <- t(as.matrix(Nile))
rownames(dat) <- "Nile"
fit_em_default <- MARSS(dat, model = mod.nile, silent = TRUE)
inits <- list(Q = matrix(var(Nile)), R = matrix(var(Nile)))
fit_bfgs_default <- MARSS(dat,
  model = mod.nile, inits = inits,
  method = "BFGS", silent = TRUE
)
```

We will also fit a stochastic prior so that we can compare more directly to the same model fit with {KFAS}.

```
mod.nile.stoch <- mod.nile
mod.nile.stoch$x0 <- fit_kfas_stoch$model$a1
mod.nile.stoch$V0 <- fit_kfas_stoch$model$P1
fit_em_stoch <- MARSS(dat, model = mod.nile.stoch, silent = TRUE)
fit_bfgs_stoch <- MARSS(dat,
  model = mod.nile.stoch, inits = inits,
  method = "BFGS", silent = TRUE
)
```

`MARSSkfas()` will return the SSModel object that is passed to `KFAS::KFS()` (internally in the {MARSS} functions). {MARSS} does not use `KFAS::fitSSM()` but it does use `KFAS::KFS()` for the filter, smoother and log-likelihood. The SSModel used inside {MARSS} looks different than `model_Nile` because the **a** term is in `T` and the **u** term is in `T`. We can set `Q` and `H` to NA to estimate those values. The results are the same as for `fit_kfas_stoch`.

```
marss_kfas_model <- MARSSkfas(fit_em_stoch,
  return.kfas.model = TRUE,
  return.lag.one = FALSE
)$kfas.model
marss_kfas_model$Q[1, 1, 1] <- NA
marss_kfas_model$H[1, 1, 1] <- NA
kinits <- c(log(var(Nile)), log(var(Nile)))
fit_marss_kfas <- fitSSM(marss_kfas_model, kinits, method = "BFGS")
```

The {KFAS} parameter estimates are in `$model`. The negative log-likelihood is in `$optim.out$value` (or use `KFS(kfas_temp$model)$logLik` for the log-likelihood). Here is the comparison of all the models. Note that the default {KFAS}

model is fundamentally different than the default {MARSS} model because the former uses a diffuse prior while the later is estimating x_1 as a parameter.

```
                              Q        R    logLik
KFAS default             1469.163 15098.65 -632.5456
MARSS em default         1526.011 14882.34 -637.6218
MARSS bfgs default       1266.874 15282.70 -637.6092
KFAS stoch              15210.195 33874.59 -697.8576
MARSS em stoch          15027.174 33924.66 -697.8586
MARSS bfgs stoch        15102.741 33956.00 -697.8580
KFAS w marss kfas model 15210.195 33874.59 -697.8576
```

20.1.2 State filtering and smoothing

For this section, we will compare filter and smoother output from the two packages. For this we need identical models.

```
fit_kfas <- fit_kfas_stoch
fit_marss <- fit_em_stoch
fit_marss$par$Q[1, 1] <- fit_kfas$model$Q
fit_marss$par$R[1, 1] <- fit_kfas$model$H
```

The Kalman filter and smoother function in {KFAS} is `KFS()`. This returns a variety of output:

```
kf_kfas <- KFS(fit_kfas$model,
  filtering = "state",
  smoothing = "state", simplify = FALSE
)
```

The analogous function in {MARSS} is `MARSSkfas()`. It uses `KFAS::KFS()` for the implementation of the Koopman and Durbin Kalman filter and smoother algorithm (Koopman and Durbin, 2000) but transforms the state-space model passed into that function in order to get a variety of variables needed for the EM algorithm, specifically the lag-1 smoother values.

```
kf_marss <- MARSSkfss(fit_marss)
```

The terminology of the filter/smoother variables is different between `MARSSkfas()` and `KFAS::KFS()`. Note {MARSS} also includes `MARSSkfss()`, which is the classic (less stable) Kalman filter and smoother; see for example the chapter on the Kalman filter in Shumway and Stoffer (2006).

```
names(kf_kfas)
names(kf_marss)
```

The {MARSS} semantics are first letter: x or y process, second letter: time (usually t), and third letter: the time conditioning. So `xtT` means the estimate of the **x** process at time t conditioned on all the data while `xtt1` means the estimate of the **x** process at time t conditioned on the data from time step 1 to time step $t-1$.

- `kf_kfas$a` is `kf_marss$xtt1`. This is the expected value of X_t conditioned on the data up to time step $t-1$. `kf_kfas$att` is `kf_marss$xtt`. This is the expected value of X_t conditioned on the data up to time step t.

```
cbind(
  a = kf_kfas$a[1:n], xtt1 = kf_marss$xtt1[1:n],
  att = kf_kfas$att[1:n], xtt = kf_marss$xtt[1:n]
)
```

```
             a      xtt1        att        xtt
[1,]   0.00000   0.00000   32.11507   32.11507
[2,]  32.11507  32.11507  396.72378  396.72378
[3,] 396.72378 396.72378  643.48197  643.48197
[4,] 643.48197 643.48197  909.42338  909.42338
[5,] 909.42338 909.42338 1029.38565 1029.38565
```

- `kf_kfas$alphahat` is `kf_marss$xtT`. This is the expected value of X_t conditioned on all the data.

```
cbind(kf_kfas$alphahat[1:n], kf_marss$xtT[1:n])
```

```
          [,1]       [,2]
[1,]   64.38081   64.38081
[2,]  569.63686  569.63686
[3,]  809.81105  809.81105
[4,]  981.20113  981.20113
[5,] 1049.85712 1049.85712
```

- `kf_kfas$v` is `kf_marss$Innov`. These are the innovations or one-step-ahead model residuals. `kf_kfas$F` is `kf_marss$Sigma`. This the variance-covariance matrix of the innovations.

```
cbind(
  v = kf_kfas$v[1:n], Innov = kf_marss$Innov[1:n],
  F = kf_kfas$F[1:n], Sigma = kf_marss$Sigma[1:n]
)
```

```
             v     Innov        F    Sigma
[1,] 1120.0000 1120.0000 34874.59 34874.59
[2,] 1127.8849 1127.8849 50056.11 50056.11
[3,]  566.2762  566.2762 60035.34 60035.34
[4,]  566.5180  566.5180 63845.84 63845.84
[5,]  250.5766  250.5766 64986.59 64986.59
```

- `kf_kfas$P` is `kf_marss$Vtt1`. This is the conditional variance of X_t conditioned on the data up to time step $t-1$. `kf_kfas$Ptt` is `kf_marss$Vtt`. This is the conditional variance of X_t conditioned on the data up to time step t.

```
cbind(
  P = kf_kfas$P[1:n], Vtt1 = kf_marss$Vtt1[1:n],
  Ptt = kf_kfas$Ptt[1:n], Vtt = kf_marss$Vtt[1:n]
)
```

```
           P     Vtt1        Ptt        Vtt
[1,]  1000.00  1000.00   971.3258   971.3258
```

```
[2,] 16181.52 16181.52 10950.5590 10950.5590
[3,] 26160.75 26160.75 14761.0518 14761.0518
[4,] 29971.25 29971.25 15901.7997 15901.7997
[5,] 31112.00 31112.00 16217.2867 16217.2867
```

- "r", "r0", "r1", "N", "N0", "N1" and "N2" are specific to the Koopman and Durbin algorithm and are not returned by `MARSSkfss()` though you could get them by using the SSModel object returned by `MARSSkfas()`.

20.1.3 Observation filtering and smoothing

Both {KFAS} and {MARSS} return the smoothed and filtered (one-step ahead) model predictions via `fitted()`. However, for {KFAS} this just returns the smoothed values. The `KFAS::KFS()` function will return the filtered and smoothed model predictions in matrix form along with other filter and smoother output.

```
kf_kfas <- KFS(fit_kfas$model,
  filtering = "signal",
  smoothing = "signal", simplify = FALSE
)
```

The function to obtain these output in {MARSS} is `fitted()`.

```
kf_marss <- MARSSkf(fit_marss)
```

Note, the function `MARSShatyt()` is the statistical counterpart to `MARSSkf()` and returns the equivalent values but for the observation equation. This is very different than what `KFS()` (or `MARSS::fitted()`) returns for the signal. `MARSShatyt()` returns the expected value of $\mathbf{Y}_t$ conditioned on $\mathbf{Y}_t = \mathbf{y}_t$. If there are no missing data, this is simply $\mathbf{y}_t$ and the covariance of $\mathbf{Y}_t$ and $\mathbf{X}_t$ conditioned on $\mathbf{Y}_t = \mathbf{y}_t$ would be 0. These values are not this when there are missing values and these expectations are crucial to the general EM algorithm for missing values.

`ytT` means the estimate of the $\mathbf{y}$ process conditioned on all the data while `ytt1` means the estimate of the $\mathbf{y}$ process conditioned on the data 1 to $t-1$.

- `kf_kfas$m` is the one-step ahead prediction of $\mathbf{y}_t$. In MARSS, this is returned by `fitted(fit_marss, type="ytt1")` in the `.fitted` column. "ytt1" means the expected value of $\mathbf{Y}_t$ conditioned on the data up to time step $t-1$.

```
ytt1_fit <- fitted(fit_marss, type = "ytt1")$.fitted
ytt1_hatyt <- MARSShatyt(fit_marss, only.kem = FALSE)$ytt1
cbind(m = kf_kfas$m[1:n],
      fitted = ytt1_fit[1:n],
      MARSShatyt = ytt1_hatyt[1:n])
```

```
          m     fitted MARSShatyt
[1,]  0.00000   0.00000   0.00000
[2,] 32.11507  32.11507  32.11507
[3,] 396.72378 396.72378 396.72378
[4,] 643.48197 643.48197 643.48197
```

```
 [5,]  909.42338  909.42338  909.42338
 [6,] 1029.38565 1029.38565 1029.38565
 [7,] 1092.24553 1092.24553 1092.24553
 [8,]  957.66596  957.66596  957.66596
 [9,] 1088.96405 1088.96405 1088.96405
[10,] 1224.47121 1224.47121 1224.47121
```

- `kf_kfas$P_mu` is the variance-covariance matrix of the expected value of $\mathbf{Y}_t$ conditioned on the data from time step 1 to time step $t-1$. `MARSShatyt(fit_marss)$var.Eytt1` returns the same values; `var.Eytt1` indicates that it is the variance of the expected value of $\mathbf{Y}_t$ conditioned on data up to time step $t-1$. In {MARSS}, the standard errors of the one-step ahead prediction are also returned by `fitted()` with `type = "ytt1"` and `interval = "confidence"`. Using `fitted()`, you can output the values as matrices instead of a data frame if you need the variance-covariance matrices not just standard errors.

```
var.Eytt1_fit <-
  fitted(fit_marss, type = "ytt1", interval = "confidence")$.se^2
var.Eytt1_hatyt <-
  MARSShatyt(fit_marss, only.kem = FALSE)$var.Eytt1
cbind(
  P_mu = kf_kfas$P_mu[1:n], fitted = var.Eytt1_fit[1:n],
  MARSShatyt = var.Eytt1_hatyt[1:n]
)
```

```
         P_mu   fitted MARSShatyt
 [1,]  1000.00  1000.00    1000.00
 [2,] 16181.52 16181.52   16181.52
 [3,] 26160.75 26160.75   26160.75
 [4,] 29971.25 29971.25   29971.25
 [5,] 31112.00 31112.00   31112.00
 [6,] 31427.48 31427.48   31427.48
 [7,] 31512.79 31512.79   31512.79
 [8,] 31535.71 31535.71   31535.71
 [9,] 31541.86 31541.86   31541.86
[10,] 31543.51 31543.51   31543.51
```

- `kf_kfas$muhat` is the smoothed prediction of $\mathbf{y}_t$. It is the expected value of $\mathbf{Z}\mathbf{X}_t + \mathbf{a}$ conditioned on the data up to time T; notice it is not the expected value of $\mathbf{Y}_t$ rather $\mathbf{Z}\mathbf{X}_t + \mathbf{a}$, which is the model prediction of $\mathbf{y}_t$. In {MARSS}, this is returned by `fitted(fit_marss, type="ytT")$.fitted`. Note, `MARSShatyt(fit_marss)$ytT` does not return this. `MARSShatyt()` returns the expected value of $\mathbf{Y}_t$ conditioned on the data up to time T, i.e., all the data, which if there are no missing data is simply the observed data.

```
ytT_fit <- fitted(fit_marss, type = "ytT")$.fitted
ytT_hatyt <- MARSShatyt(fit_marss)$ytT
cbind(
  a = kf_kfas$muhat[1:n], fitted = ytT_fit[1:n],
```

```
  MARSShatyt = ytT_hatyt[1:n], Nile = Nile[1:n]
)
              a     fitted MARSShatyt Nile
 [1,]   64.38081   64.38081       1120 1120
 [2,]  569.63686  569.63686       1160 1160
 [3,]  809.81105  809.81105        963  963
 [4,]  981.20113  981.20113       1210 1210
 [5,] 1049.85712 1049.85712       1160 1160
 [6,] 1069.05730 1069.05730       1160 1160
 [7,] 1047.42286 1047.42286        813  813
 [8,] 1131.04778 1131.04778       1230 1230
 [9,] 1170.24168 1170.24168       1370 1370
[10,] 1119.74113 1119.74113       1140 1140
```

- `kf_kfas$V_mu` is the variance of the expected value of $\mathbf{Y}_t$ conditioned on all the data. In {MARSS}, this is returned in the standard errors returned by `fitted(..., interval="confidence")` . Again, `var.Eytt1` returned by `MARSShatyt()` is not this because it returns the variance of the expected value of $\mathbf{Y}_t$ conditioned on all the data not the expected value of $\mathbf{ZX}_t + \mathbf{a}$. The latter is the model prediction. For the former, if there are no missing values, $E[\mathbf{Y}_t|\mathbf{Y}_t = \mathbf{y}_t] = \mathbf{y}_t$ and the variance is 0.

```
var.EytT_fit <-
  fitted(fit_marss, type = "ytT", interval = "confidence")$.se^2
var.EytT_hatyt <-
  MARSShatyt(fit_marss, only.kem = FALSE)$var.EytT
cbind(
  V_mu = kf_kfas$V_mu[1:n], fitted = var.EytT_fit[1:n],
  MARSShatyt = var.EytT_hatyt[1:n]
)
           V_mu     fitted MARSShatyt
 [1,]   942.3097   942.3097          0
 [2,]  8128.6821  8128.6821          0
 [3,] 10055.5588 10055.5588          0
 [4,] 10572.2107 10572.2107          0
 [5,] 10710.7402 10710.7402          0
 [6,] 10747.8841 10747.8841          0
 [7,] 10757.8434 10757.8434          0
 [8,] 10760.5138 10760.5138          0
 [9,] 10761.2298 10761.2298          0
[10,] 10761.4218 10761.4218          0
```

20.1.4 Confidence and prediction intervals

Both {KFAS} and {MARSS} use `predict()` for predictions. The inputs and outputs of the `predice()` functions from the two packages have many similarities but also many differences.

Smoothed predictions

With `newdata` and `n.ahead` not passed in, `predict()` returns the model prediction for $\mathbf{Y}_t$ (i.e., fitted values) conditioned on all the data. This is the expected value and standard error of $\mathbf{ZX}_t + \mathbf{a}$ conditioned on all the data (before and after t).

```
conf_kfas <- predict(fit_kfas$model,
  interval = "confidence",
  se.fit = TRUE
)
head(conf_kfas)
```

```
Time Series:
Start = 1871
End = 1876
Frequency = 1
           fit        lwr       upr    se.fit
1871   64.38081   4.215678  124.5460  30.69706
1872  569.63686 392.928063  746.3456  90.15920
1873  809.81105 613.270944 1006.3512 100.27741
1874  981.20113 779.675175 1182.7271 102.82126
1875 1049.85712 847.015139 1252.6991 103.49271
1876 1069.05730 865.863912 1272.2507 103.67200
```

In {MARSS}, the same prediction is returned by `fitted()`. By default `fitted()` returns a data frame, but the output can be changed to return matrices.

```
conf_marss1 <- fitted(fit_marss, type = "ytT", interval = "confidence")
head(conf_marss1)
```

```
  .rownames t    y    .fitted       .se  .conf.low
1      Nile 1 1120   64.38081  30.69706   4.215678
2      Nile 2 1160  569.63686  90.15920 392.928063
3      Nile 3  963  809.81105 100.27741 613.270944
4      Nile 4 1210  981.20113 102.82126 779.675175
5      Nile 5 1160 1049.85712 103.49271 847.015139
6      Nile 6 1160 1069.05730 103.67200 865.863912
   .conf.up
1  124.5460
2  746.3456
3 1006.3512
4 1182.7271
5 1252.6991
6 1272.2507
```

`predict()` can also be used (with type specified). `predict()` returns a list and the data frame is in `pred`.

```
conf_marss2 <- predict(fit_marss,
  type = "ytT",
  interval = "confidence", level = 0.95
)
head(conf_marss2$pred)
```

```
  .rownames t    y   estimate        se      Lo 95
1      Nile 1 1120   64.38081  30.69706   4.215678
2      Nile 2 1160  569.63686  90.15920 392.928063
3      Nile 3  963  809.81105 100.27741 613.270944
4      Nile 4 1210  981.20113 102.82126 779.675175
5      Nile 5 1160 1049.85712 103.49271 847.015139
6      Nile 6 1160 1069.05730 103.67200 865.863912
      Hi 95
1  124.5460
2  746.3456
3 1006.3512
4 1182.7271
5 1252.6991
6 1272.2507
```

Prediction intervals are the intervals for new data. They are the expected value and standard error of $\mathbf{Z}\mathbf{X}_t + \mathbf{a} + \mathbf{v}_t$ conditioned on all the data (before and after t). `predict.SSModel()` returns the upper and lower prediction intervals, but the standard error returned is the standard error for the confidence interval (i.e., for $\mathbf{Z}\mathbf{X}_t + \mathbf{a}$) not the prediction interval.

```
pred_kfas <- predict(fit_kfas$model,
  interval = "prediction", se.fit = TRUE
)
head(pred_kfas)
```

```
Time Series:
Start = 1871
End = 1876
Frequency = 1
            fit       lwr       upr    se.fit
1871   64.38081 -301.3345  430.0961  30.69706
1872  569.63686  167.9481  971.3256  90.15920
1873  809.81105  399.0120 1220.6101 100.27741
1874  981.20113  567.9935 1394.4088 102.82126
1875 1049.85712  636.0060 1463.7082 103.49271
1876 1069.05730  655.0339 1483.0807 103.67200
```

In MARSS, `fitted()` or `predict()` can be used to return the prediction intervals. These functions return the standard deviation of $\mathbf{Z}\mathbf{X}_t + \mathbf{a} + \mathbf{v}_t$ (so standard deviation of the prediction intervals). `.sd` will not be the same as `se.fit` returned by `predict.SSModel()` but the intervals will be the same.

```
pred_marss1 <- fitted(fit_marss, type = "ytT", interval = "prediction")
head(pred_marss1)
```

```
  .rownames t    y    .fitted      .sd      .lwr      .upr
1      Nile 1 1120   64.38081 186.5929 -301.3345  430.0961
2      Nile 2 1160  569.63686 204.9470  167.9481  971.3256
3      Nile 3  963  809.81105 209.5952  399.0120 1220.6101
4      Nile 4 1210  981.20113 210.8241  567.9935 1394.4088
5      Nile 5 1160 1049.85712 211.1524  636.0060 1463.7082
6      Nile 6 1160 1069.05730 211.2403  655.0339 1483.0807
```

This would return the same values but as a marssPredict object instead of a data frame.

```
pred_marss2 <- predict(fit_marss,
  type = "ytT",
  interval = "prediction", level = 0.95
)
```

One step ahead predictions

The default for `predict.SSModel()` in {KFAS} is to return model fitted values conditioned on all the data. For the one-step ahead predictions, set `filtered=TRUE`. This returns the expected value and standard error of $\mathbf{Z}\mathbf{X}_t + \mathbf{a}$ conditioned on the data up to $t-1$ only.

```
conf_kfas_t1 <- predict(fit_kfas$model,
  interval = "confidence",
  se.fit = TRUE, filtered = TRUE
)
head(conf_kfas_t1)
```

```
Time Series:
Start = 1871
End = 1876
Frequency = 1
            fit        lwr       upr    se.fit
1871    0.00000  -61.97950   61.9795  31.62278
1872   32.11507 -217.20530  281.4354 127.20661
1873  396.72378   79.71359  713.7340 161.74287
1874  643.48197  304.16897  982.7950 173.12206
1875  909.42338  563.71332 1255.1334 176.38593
1876 1029.38565  681.92720 1376.8441 177.27798
```

In {MARSS}, this output is returned by setting `type="ytt1"`.

```
conf_marss1_t1 <- fitted(fit_marss, type = "ytt1", interval = "confidence")
head(conf_marss1_t1)
```

```
  .rownames t    y    .fitted       .se  .conf.low
1      Nile 1 1120    0.00000  31.62278  -61.97950
2      Nile 2 1160   32.11507 127.20661 -217.20530
3      Nile 3  963  396.72378 161.74287   79.71359
4      Nile 4 1210  643.48197 173.12206  304.16897
5      Nile 5 1160  909.42338 176.38593  563.71332
6      Nile 6 1160 1029.38565 177.27798  681.92720
   .conf.up
1   61.9795
2  281.4354
3  713.7340
4  982.7950
5 1255.1334
6 1376.8441
```

With `predict()`, the one-step ahead predictions are returned using:

```
conf_marss2_t1 <- predict(fit_marss,
  type = "ytt1",
  interval = "confidence", level = 0.95
)
head(conf_marss2_t1$pred)
```

```
  .rownames t    y   estimate        se      Lo 95
1      Nile 1 1120    0.00000  31.62278  -61.97950
2      Nile 2 1160   32.11507 127.20661 -217.20530
3      Nile 3  963  396.72378 161.74287   79.71359
4      Nile 4 1210  643.48197 173.12206  304.16897
5      Nile 5 1160  909.42338 176.38593  563.71332
6      Nile 6 1160 1029.38565 177.27798  681.92720
      Hi 95
1   61.9795
2  281.4354
3  713.7340
4  982.7950
5 1255.1334
6 1376.8441
```

As before, we can get prediction intervals for the one-step ahead new data also.

```
pred_kfas_t1 <- predict(fit_kfas$model,
  interval = "prediction",
  se.fit = TRUE, filtered = TRUE
)
head(pred_kfas_t1)
```

```
Time Series:
Start = 1871
```

```
End = 1876
Frequency = 1
            fit        lwr       upr    se.fit
1871    0.00000 -366.01817  366.0182  31.62278
1872   32.11507 -406.39204  470.6222 127.20661
1873  396.72378  -83.50877  876.9563 161.74287
1874  643.48197  148.24349 1138.7205 173.12206
1875  909.42338  409.78022 1409.0665 176.38593
1876 1029.38565  528.53116 1530.2401 177.27798
```

In {MARSS}, `fitted()` or `predict()` can be used. Again, these functions return the standard deviation of $\mathbf{ZX}_t + \mathbf{a} + \mathbf{v}_t$ (so standard deviation of predictions) not the standard error of the mean prediction. The {KFAS} returns the latter for prediction intervals.

```
pred_marss1_t1 <- fitted(fit_marss, type = "ytt1", interval = "prediction")
head(pred_marss1_t1)
```

```
  .rownames t    y    .fitted      .sd       .lwr      .upr
1      Nile 1 1120    0.00000 186.7474 -366.01817  366.0182
2      Nile 2 1160   32.11507 223.7322 -406.39204  470.6222
3      Nile 3  963  396.72378 245.0211  -83.50877  876.9563
4      Nile 4 1210  643.48197 252.6773  148.24349 1138.7205
5      Nile 5 1160  909.42338 254.9247  409.78022 1409.0665
6      Nile 6 1160 1029.38565 255.5427  528.53116 1530.2401
```

This would return the same values.

```
pred_marss2_t1 <- predict(fit_marss,
  type = "ytt1",
  interval = "prediction", level = 0.95
)
```

20.1.5 Residuals

Mathematically, the state and model residuals are

$$\begin{aligned} \textit{model} &: \widehat{\mathbf{v}}_t = E[\mathbf{ZX}_t + \mathbf{a} + \mathbf{v}_t | \mathbf{Y} = \mathbf{y}] - E[\mathbf{ZX}_t + \mathbf{a} | \mathbf{Y} = \mathbf{y}] \\ \textit{state} &: \widehat{\mathbf{w}}_t = E[\mathbf{BX}_{t-1} + \mathbf{u} + \mathbf{w}_t | \mathbf{Y} = \mathbf{y}] - E[\mathbf{BX}_{t-1} + \mathbf{u} | \mathbf{Y} = \mathbf{y}] \\ \textit{joint} &: \varepsilon_t \sim \text{MVN}\left(\begin{bmatrix} \widehat{\mathbf{v}}_t \\ \widehat{\mathbf{w}}_{t+1} \end{bmatrix}, \Sigma_t \right) \end{aligned} \tag{20.2}$$

The expectation can be conditioned on all the data (smoothation), data 1 to $t-1$ (one-step ahead), or data 1 to t (contemporaneous). Σ_t is the conditional (on data) variance of the joint residuals (state and observation); note the residuals for the $\widehat{\mathbf{v}}_t$

and $\widehat{\mathbf{w}}_t$ in ε_t have different time indexing[1] Residuals can be standardized by either the full Σ matrix via the inverse of the lower triangle of the Cholesky matrix or via the inverse of the square root of the diagonal of the Σ matrix (aka marginal or Pearson residuals).

The {MARSS} residuals function will return all combinations of state versus observations, three conditioning types, and four standardization types (none, Cholesky, marginal, or Block Cholesky for states only). This amounts to 2 times 3 times 4 = 24 possible residuals (except that state contemporaneous residuals do not exist and Block Cholesky standardization only applies to states so 3*3 + 2*4 = 17 residual types). {KFAS} has two residuals functions: `residuals()` and `rstandard()`. These will return some of the possible residuals types but the names used in {KFAS} versus {MARSS} are different. {MARSS} has two residuals functions, which return the same information in different forms. The normal one for users is `residuals()` and returns a data frame. With `residuals()`, one must specify the conditioning (tT, tt or tt1) and the standardization (none, Cholesky, marginal or Block.Cholesky). `MARSSresiduals()` returns matrices for all 3 standardizations along with the full Σ matrices. With `MARSSresiduals()`, only the conditioning (tT, tt or tt1) needs to be specified. For normal use, `residuals()` is the function to use. For those needing to develop new functions or doing research on the properties of state-space residuals, the full matrices will be helpful.

Here is a table of the correspondence between the {KFAS} and {MARSS} residual functions. The header is the {MARSS} naming scheme for state versus observation (x versus y) and conditioning (all data = tT, 1 to t = tt, and 1 to t-1 = tt1). This shows the corresponding {KFAS} function for a call to
`MARSS::residuals(marss_fit, type=..., conditioning=...)`
`marss_fit` is output from `MARSS()`. In the {KFAS} functions, `kfas_fit` is output from `fitSSM()`.

Case 1. Recursive residuals

```
kfs <- KFS(fit_kfas$model)
resid_kfas <- residuals(kfs, type = "recursive")
resid_marss <- residuals(fit_marss,
  type = "tt1",
  standardization = "marginal"
)
resid_marss <- subset(resid_marss, name == "model")
df <- cbind(
  MARSS = resid_marss$.resids,
  KFAS = as.vector(resid_kfas)
)
head(df)
```

[1] The joint residuals for MARSS models are traditionally written this way but you can certainly write them with the same time indexing if you wanted.

	type			name		standardization			
	tT	tt	tt1	model	state	none	chol	mar	bchol
residuals(kfas_obj, type = "recursive")		X		X		X			
residuals(kfas_obj, type = "pearson")		X		X				X	
residuals(kfas_obj, type = "response")	X			X				X	
residuals(kfas_obj, type = "state")	X				X				X
rstandard(kfas_obj, type = "recursive", standardization_type = "marginal")		X		X				X	
rstandard(kfas_fit$model, type = "recursive", standardization_type = "cholesky")		X		X			X		X
rstandard(kfas_obj, type = "pearson", standardization_type = "marginal")	X			X				X	
rstandard(kfas_fit$model, type = "pearson", standardization_type = "cholesky")	X			X			X		X
rstandard(kfas_obj, type = "state", standardization_type = "marginal")	X				X			X	
rstandard(kfas_fit$model, type = "state", standardization_type = "cholesky")	X				X				X

```
         MARSS      KFAS
[1,] 1120.0000 1120.0000
[2,] 1127.8849 1127.8849
[3,]  566.2762  566.2762
[4,]  566.5180  566.5180
[5,]  250.5766  250.5766
[6,]  130.6143  130.6143
```

```
kfs <- KFS(fit_kfas$model)
resid_kfas <- rstandard(kfs,
  type = "recursive",
  standardization_type = "marginal"
)
resid_marss <- residuals(fit_marss,
  type = "tt1",
  standardization = "marginal"
)
resid_marss <- subset(resid_marss, name == "model")
df <- cbind(
  MARSS = resid_marss$.std.resids,
  KFAS = as.vector(resid_kfas)
)
head(df)
```

```
         MARSS      KFAS
[1,] 5.9974062 5.9974062
```

```
[2,] 5.0412268 5.0412268
[3,] 2.3111324 2.3111324
[4,] 2.2420611 2.2420611
[5,] 0.9829438 0.9829438
[6,] 0.5111253 0.5111253
```

In the univariate case, the Cholesky standardization is not different.

```
kfs <- KFS(fit_kfas$model)
resid_kfas <- rstandard(kfs,
  type = "recursive",
  standardization_type = "cholesky"
)
resid_marss <- residuals(fit_marss,
  type = "tt1",
  standardization = "Cholesky"
)
resid_marss <- subset(resid_marss, name == "model")
df <- cbind(
  MARSS = resid_marss$.std.resids,
  KFAS = as.vector(resid_kfas)
)
head(df)
```

```
        MARSS      KFAS
[1,] 5.9974062 5.9974062
[2,] 5.0412268 5.0412268
[3,] 2.3111324 2.3111324
[4,] 2.2420611 2.2420611
[5,] 0.9829438 0.9829438
[6,] 0.5111253 0.5111253
```

Case 2. Pearson residuals

No standardization is done for `residuals(kfs, type = "pearson")`.

```
kfs <- KFS(fit_kfas$model)
resid_kfas <- residuals(kfs, type = "pearson")
resid_marss <- residuals(fit_marss, type = "tT")
resid_marss <- subset(resid_marss, name == "model")
df <- cbind(
  MARSS = resid_marss$.resids,
  KFAS = as.vector(resid_kfas)
)
head(df)
```

```
        MARSS      KFAS
[1,] 1055.6192 1055.6192
[2,]  590.3631  590.3631
[3,]  153.1889  153.1889
[4,]  228.7989  228.7989
[5,]  110.1429  110.1429
[6,]   90.9427   90.9427
```

```
kfs <- KFS(fit_kfas$model)
resid_kfas <- rstandard(kfs,
  type = "pearson",
  standardization_type = "marginal"
)
resid_marss <- residuals(fit_marss,
  type = "tT",
  standardization = "marginal"
)
resid_marss <- subset(resid_marss, name == "model")
df <- cbind(
  MARSS = resid_marss$.std.resids,
  KFAS = as.vector(resid_kfas)
)
head(df)
```

```
        MARSS      KFAS
[1,] 5.8169639 5.8169639
[2,] 3.6792994 3.6792994
[3,] 0.9925797 0.9925797
[4,] 1.4988347 1.4988347
[5,] 0.7236875 0.7236875
[6,] 0.5980134 0.5980134
```

In the univariate case, the Cholesky standardization is not different.

```
kfs <- KFS(fit_kfas$model)
resid_kfas <- rstandard(kfs,
  type = "pearson",
  standardization_type = "cholesky"
)
resid_marss <- residuals(fit_marss,
  type = "tT",
  standardization = "Cholesky"
)
resid_marss <- subset(resid_marss, name == "model")
df <- cbind(
  MARSS = resid_marss$.std.resids,
  KFAS = as.vector(resid_kfas)
```

```
)
head(df)
```

```
        MARSS      KFAS
[1,] 5.8169639 5.8169639
[2,] 3.6792994 3.6792994
[3,] 0.9925797 0.9925797
[4,] 1.4988347 1.4988347
[5,] 0.7236875 0.7236875
[6,] 0.5980134 0.5980134
```

Case 3. Response residuals

```
kfs <- KFS(fit_kfas$model)
resid_kfas <- residuals(kfs, type = "response")
resid_marss <- residuals(fit_marss, type = "tT")
resid_marss <- subset(resid_marss, name == "model")
df <- cbind(
  MARSS = resid_marss$.resids,
  KFAS = as.vector(resid_kfas)
)
head(df)
```

```
        MARSS      KFAS
[1,] 1055.6192 1055.6192
[2,]  590.3631  590.3631
[3,]  153.1889  153.1889
[4,]  228.7989  228.7989
[5,]  110.1429  110.1429
[6,]   90.9427   90.9427
```

Case 4. State residuals

No standardization.

```
kfs <- KFS(fit_kfas$model, smoothing = "disturbance")
resid_kfas <- residuals(kfs, type = "state")
resid_marss <- residuals(fit_marss, type = "tT")
resid_marss <- subset(resid_marss, name == "state")
df <- cbind(
  MARSS = resid_marss$.resids,
  KFAS = as.vector(resid_kfas)
)
head(df)
```

```
        MARSS      KFAS
[1,] 505.25604 505.25604
[2,] 240.17420 240.17420
[3,] 171.39008 171.39008
[4,]  68.65598  68.65598
[5,]  19.20019  19.20019
[6,] -21.63444 -21.63444
```

Marginal standardization.

```
kfs <- KFS(fit_kfas$model, smoothing = "disturbance")
resid_kfas <- rstandard(kfs,
  type = "state",
  standardization_type = "marginal"
)
resid_marss <- residuals(fit_marss,
  type = "tT",
  standardization = "marginal"
)
resid_marss <- subset(resid_marss, name == "state")
df <- cbind(
  MARSS = resid_marss$.std.resids,
  KFAS = as.vector(resid_kfas)
)
head(df)
```

```
         MARSS       KFAS
[1,]  5.9899274  5.9899274
[2,]  3.2550586  3.2550586
[3,]  2.4247425  2.4247425
[4,]  0.9832026  0.9832026
[5,]  0.2758730  0.2758730
[6,] -0.3111264 -0.3111264
```

```
kfs <- KFS(fit_kfas$model, smoothing = "disturbance")
resid_kfas <- rstandard(kfs,
  type = "state",
  standardization_type = "cholesky"
)
resid_marss <- residuals(fit_marss,
  type = "tT",
  standardization = "Block.Cholesky"
)
resid_marss <- subset(resid_marss, name == "state")
df <- cbind(
  MARSS = resid_marss$.std.resids,
  KFAS = as.vector(resid_kfas)
```

```
)
head(df)
```

```
          MARSS       KFAS
[1,]  5.9899274  5.9899274
[2,]  3.2550586  3.2550586
[3,]  2.4247425  2.4247425
[4,]  0.9832026  0.9832026
[5,]  0.2758730  0.2758730
[6,] -0.3111264 -0.3111264
```

The Cholesky standardization is "block" style in {KFAS} and treats the model and state smoothed residuals as independent (they are not).

```
kfs <- KFS(fit_kfas$model, smoothing = "disturbance")
resid_kfas <- rstandard(kfs,
  type = "state",
  standardization_type = "marginal"
)
resid_marss <- residuals(fit_marss,
  type = "tT",
  standardization = "marginal"
)
resid_marss <- subset(resid_marss, name == "state")
df <- cbind(
  MARSS = resid_marss$.std.resids,
  KFAS = as.vector(resid_kfas)
)
head(df)
```

```
          MARSS       KFAS
[1,]  5.9899274  5.9899274
[2,]  3.2550586  3.2550586
[3,]  2.4247425  2.4247425
[4,]  0.9832026  0.9832026
[5,]  0.2758730  0.2758730
[6,] -0.3111264 -0.3111264
```

```
kfs <- KFS(fit_kfas$model, smoothing = "disturbance")
resid_kfas <- rstandard(kfs,
  type = "state",
  standardization_type = "cholesky"
)
resid_marss <- residuals(fit_marss,
  type = "tT",
  standardization = "Block.Cholesky"
)
```

```
resid_marss <- subset(resid_marss, name == "state")
df <- cbind(
  MARSS = resid_marss$.std.resids,
  KFAS = as.vector(resid_kfas)
)
head(df)
```

```
         MARSS       KFAS
[1,]  5.9899274  5.9899274
[2,]  3.2550586  3.2550586
[3,]  2.4247425  2.4247425
[4,]  0.9832026  0.9832026
[5,]  0.2758730  0.2758730
[6,] -0.3111264 -0.3111264
```

```
kfs <- KFS(fit_kfas$model, smoothing = "disturbance")
test <- cbind(
  b = fit_kfas$model$Q[1, 1, 1] - kfs$V_eta[1, 1, ],
  a = MARSSresiduals(fit_marss, type = "tT")$var.residuals[2, 2, ]
)
test <- as.data.frame(test)
test$diff <- test$b - test$a
head(test)
```

```
        b        a          diff
1 7115.082 7115.082  0.000000e+00
2 5444.212 5444.212  2.728484e-12
3 4996.203 4996.203  1.818989e-12
4 4876.079 4876.079 -5.456968e-12
5 4843.870 4843.870  1.818989e-12
6 4835.234 4835.234  0.000000e+00
```

```
tail(test)
```

```
          b        a         diff
95  4825.374 4825.374 1.818989e-12
96  4807.095 4807.095 1.818989e-12
97  4738.924 4738.924 1.818989e-12
98  4484.677 4484.677 0.000000e+00
99  3536.451 3536.451 5.456968e-12
100    0.000       NA           NA
```

Plotting

We can plot the confidence intervals and predictions (Figure 20.1).

With {MARSS}, there is a plot method (and `ggplot2::autoplot()` method) for marssMLE objects which will make the smoothed model predictions with CIs and PIs (Figure 20.2). Alternatively you could used the fitted output (Figure 20.3).

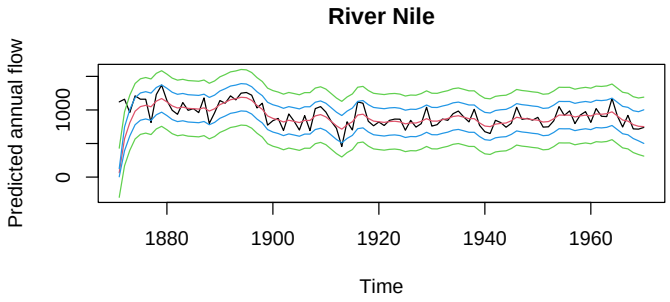

Fig. 20.1. KFAS smooth model fit (expected value of $\mathbf{Z}\mathbf{X}_t + \mathbf{a}$) confidence intervals and predictions.

```
plot.type <- ifelse(packageVersion("MARSS") < '3.11.4', "model.ytT", "fitted.ytT")
plot(fit_marss, plot.type = plot.type, pi.int = TRUE)
```

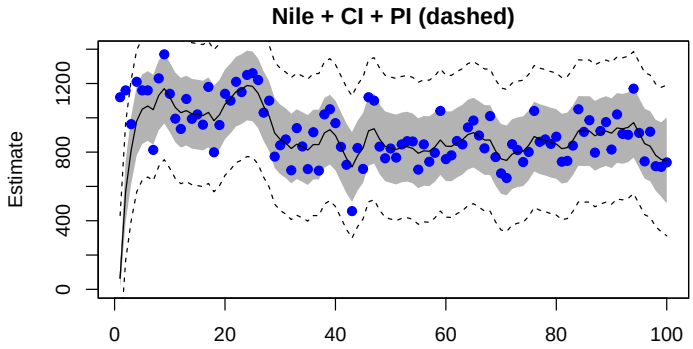

Fig. 20.2. MARSS smooth model fit (expected value of $\mathbf{Z}\mathbf{X}_t + \mathbf{a}$) confidence intervals and predictions. Although `plot()` is used here, `ggplot2::autoplot()` is the recommended plotting function for marssMLE objects.

Missing observations

Missing values are handled seamlessly in both {KFAS} and {MARSS}. We will use a model with a stochastic x_1 again so we can compare directly to {MARSS} output.

```
NileNA <- Nile
NileNA[c(21:40, 61:80)] <- NA
model_NileNA_stoch <-
  SSModel(NileNA ~ SSMtrend(
    degree = 1,
    Q = list(matrix(NA))
```

```
require(ggplot2)
df <- cbind(conf_marss1, pred_marss1[, c(".lwr", ".upr")])
ggplot(df, aes(x = t, y = .fitted)) +
  geom_ribbon(aes(ymin = .lwr, ymax = .upr), fill = "grey") +
  geom_ribbon(aes(ymin = .conf.low, ymax = .conf.up), fill = "blue", alpha = 0.25) +
  geom_line(linetype = 2) +
  ylab("Predicted Annual Flow") +
  xlab("") +
  ggtitle("River Nile")
```

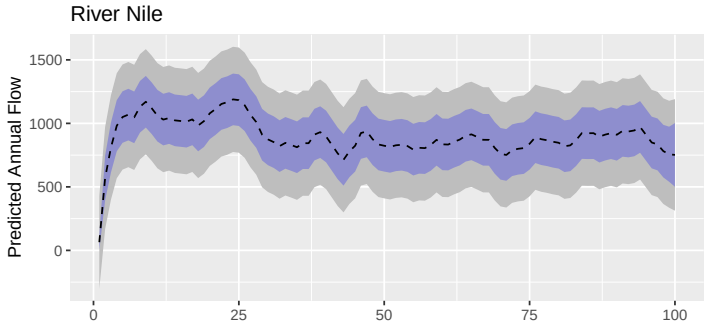

Fig. 20.3. MARSS smooth model fit with confidence intervals and predictions using ggplot.

```
  ),
  H = matrix(NA)
  )
model_NileNA_stoch$a1[1, 1] <- 0
model_NileNA_stoch$P1[1, 1] <- model_Nile_stoch$P1[1, 1]
model_NileNA_stoch$P1inf[1, 1] <- 0
kinits <- c(log(var(Nile)), log(var(Nile)))
fit_kfas_NA <- fitSSM(model_NileNA_stoch, kinits, method = "BFGS")
fit_marss_NA <- MARSS(as.vector(NileNA),
  model = mod.nile.stoch,
  inits = inits, method = "BFGS", silent = TRUE
)
```

The fits are close. The difference is due to the maximization stopping at different places.

```
rbind(
  MARSS = c(
    Q = coef(fit_marss_NA, type = "matrix")$Q,
    R = coef(fit_marss_NA, type = "matrix")$R,
    logLik = logLik(fit_marss_NA)
  ),
```

```
  KFAS = c(
    Q = fit_kfas_NA$model$Q,
    R = fit_kfas_NA$model$H,
    logLik = -1 * fit_kfas_NA$optim.out$value
  )
)
```

```
             Q       R    logLik
MARSS 22133.85 52955.30 -433.0319
KFAS  22084.75 53400.57 -433.0312
```

Plot the confidence intervals on the estimate of the river flow (Figure 20.4). This is the model fit conditioned on all the data.

```
conf_kfas_NA <-
  predict(fit_kfas_NA$model, interval = "confidence", filtered = FALSE)
conf_marss_NA <-
  predict(fit_marss_NA, interval = "confidence", type = "ytT", level = 0.95)$pr
```

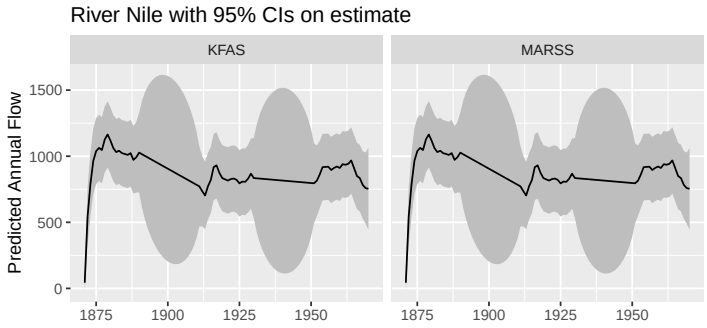

Fig. 20.4. Estimates of the river flow. When there are NAs, the estimate is less certain.

Compare model fitted values using all the data (smoothed) to one-step-ahead estimates (Figure 20.5).

```
fitted_kfas_NA <- data.frame(
  smooth = as.vector(fitted(fit_kfas_NA$model)),
  one.step.ahead = as.vector(fitted(fit_kfas_NA$model, filtered = TRUE)),
  name = "KFAS"
)
fitted_marss_NA <- data.frame(
  smooth = fitted(fit_marss_NA, type = "ytT")$.fitted,
  one.step.ahead = fitted(fit_marss_NA, type = "ytt1")$.fitted,
```

```
  name = "MARSS"
)
```

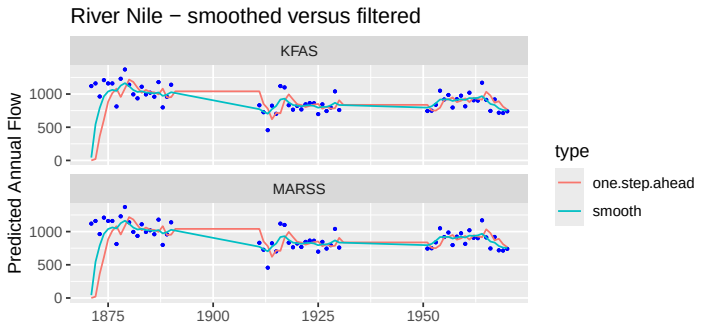

Fig. 20.5. Smoothed (all data) or filtered (one-step ahead) estimates of the river flow.

20.2 Global temperature example

This example uses two series of average global temperature deviations for years 1880-1987 (Figure 20.6) using two observation time series (Shumway and Stoffer, 2006, p. 327). This is a multivariate local level model with only one state process but two observation processes.

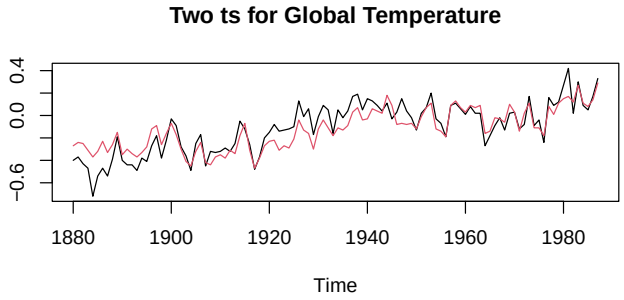

Fig. 20.6. GlobalTemp data set

$$x_t = x_{t-1} + w_t \text{ where } w_t \sim \mathrm{N}(0, q)$$
$$y_t = \begin{bmatrix}1\\1\end{bmatrix} x_t + \mathbf{v}_t \text{ where } \mathbf{v}_t \sim \mathrm{MVN}\left(\begin{bmatrix}0\\0\end{bmatrix}, \begin{bmatrix}r & c\\ c & r\end{bmatrix}\right) \tag{20.3}$$

Fit with {KFAS} (following code in `?KFAS`).

```
data("GlobalTemp")
model_temp <- SSModel(GlobalTemp ~ SSMtrend(1, Q = NA, type = "common"),
  H = matrix(NA, 2, 2)
)
kinits <- chol(cov(GlobalTemp))[c(1, 4, 3)]
kinits <- c(0.5 * log(0.1), log(kinits[1:2]), kinits[3])
kfas_temp_default <- fitSSM(model_temp, kinits, method = "BFGS")
model_temp_stoch <- model_temp
model_temp_stoch$a1[1, 1] <- 0
model_temp_stoch$P1[1, 1] <- 1000 * max(diag(var(GlobalTemp)))
model_temp_stoch$P1inf[1, 1] <- 0
kfas_temp_stoch <- fitSSM(model_temp_stoch, kinits, method = "BFGS")
```

Fit with {MARSS}. We specify the equation matrices. **Q** is univariate so we don't need to specify that. **B** is not used so default is fine.

```
mod.list <- list(
  Z = matrix(1, 2, 1),
  R = matrix(c("r1", "c", "c", "r2"), 2, 2),
  U = matrix(0),
  A = matrix(0, 2, 1),
  tinitx = 1
)
marss_temp_default <- MARSS(t(GlobalTemp), model = mod.list)
mod.list$x0 <- kfas_temp_stoch$model$a1
mod.list$V0 <- kfas_temp_stoch$model$P1
marss_temp_stoch_em <- MARSS(t(GlobalTemp), model = mod.list)
# use inits from a short run of EM algorithm
inits <- MARSS(t(GlobalTemp),
  model = mod.list, control = list(maxit = 20),
  silent = TRUE
)
marss_temp_stoch_bfgs <- MARSS(t(GlobalTemp),
  model = mod.list,
  inits = inits, method = "BFGS"
)
```

Compare estimates. The first two are the default models fit by {KFAS} and {MARSS} respectively. {KFAS} uses a diffuse prior while {MARSS} estimates x_1 as a parameters (with the variance of x_0 equal to 0). These are not the same models and their log-likelihoods will not be comparable. The last two are the same

model (with a stochastic prior on x_0) but fit with {KFAS} versus {MARSS} EM or {MARSS} BFGS.

```
                       Q      R1    Rcov      R2   logLik
KFAS default     0.00263 0.01950 0.00651 0.00539 177.7361
MARSS em default 0.00301 0.01928 0.00620 0.00498 179.7697
KFAS stoch       0.00263 0.01950 0.00651 0.00539 174.8593
MARSS em stoch   0.00299 0.01935 0.00628 0.00508 174.8430
MARSS bfgs stoch 0.00262 0.01951 0.00652 0.00539 174.8592
```

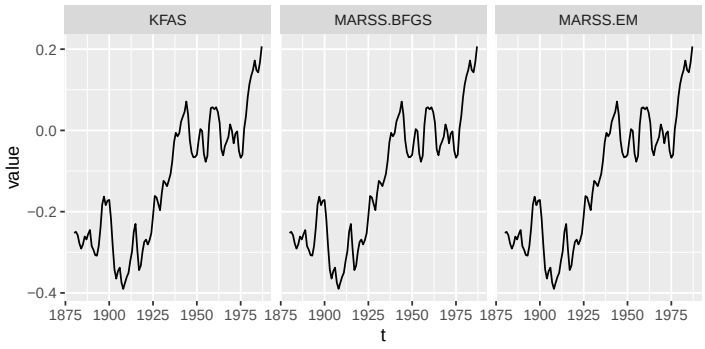

Fig. 20.7. GlobalTemp estimates

Comparison of state and model (Pearson) residuals for the estimated models and a {MARSS} model that has the same parameters as the {KFAS} estimated model.

```
mod.list <- list(
  Z = matrix(kfas_temp_stoch$model$Z, ncol = 1),
  R = kfas_temp_stoch$model$H[, , 1],
  U = matrix(0),
  A = matrix(0, 2, 1),
  Q = matrix(kfas_temp_stoch$model$Q[, , 1]),
  x0 = kfas_temp_stoch$model$a1,
  V0 = kfas_temp_stoch$model$P1,
  tinitx = 1
)
marss_test <- MARSS(t(GlobalTemp), model = mod.list)
```

20.3 Summary

{KFAS} fits state-space models in the general exponential family, of which MARSS models with Gaussian errors are a part. The {KFAS} package relies, largely, on the

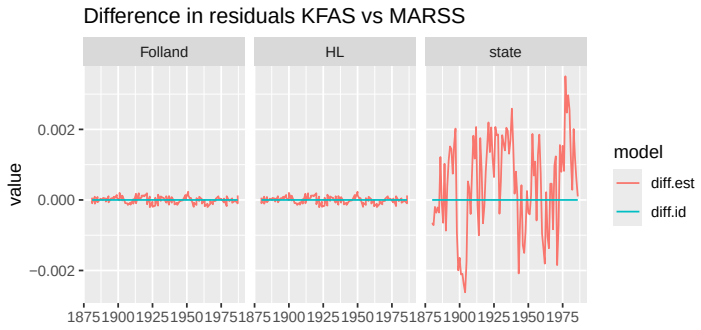

Fig. 20.8. Comparison of residuals. `diff.est` are the difference of the same models estimated with BFGS with the {KFAS} versus {MARSS} package. `diff.id` are identical models (same parameter values) but the residuals are computed with different algorithms.

Durbin and Koopman algorithms which avoid the inversions in the classical Kalman filter/smoother algorithms. These inversions lead to numerical instability and are slow, and avoiding them greatly improves the stability of the fitting of state-space models. The {KFAS} package also includes an exact algorithm for including diffuse priors. The {KFAS} package has a number of functions to create a variety of structural time-series models.

The {MARSS} package implements a general EM algorithm which allows seamless incorporation of linear constraints within matrices, including importantly the **Q** and **R** matrices. It normally treats initial conditions as an estimated parameter to avoid adding any information regarding the covariance structure of the initial conditions, specifically to avoid a diagonal initial conditions variance matrix.

The syntax of the {KFAS} and {MARSS} packages are different and the output functions and semantics are different. This chapter illustrates how to fit the same models with each package and obtain the same output.

Part IV

Appendices

A

Package MARSS: Warnings and errors

The following are brief descriptions of the warning and error messages you may see and what they mean (or might mean). More warning information can be found by typing `MARSSinfo()` at the commmand line.

Over the years of helping people fit MARSS models, we have found that the most common problems arise when the MARSS model is inconsistent with the data. The following are common scenarios.

- The data do not remotely follow a Gaussian distribution. For example, they are binned data with long strings of one value.
- The MARSS model being fit is stationary but the data are clearly non stationary.
- The MARSS model being fit is non-stationary but the data are clearly stationary.
- The initial conditions are impossible given the model or the data. For example, the initial conditions are fixed at 0 but data at $t = 1$ is far from 0. Or the model implies that the initial state are correlated but a diagonal (= i.i.d.) initial condition variance-covariance matrix was used.
- The MARSS model has a equilibrium mean level but the data are nowhere near that level and **a** was set to zero so there is no way for the model to fit the data.
- The data just do not look anything like an autoregressive process.
- There isn't enough data to estimate both process and observation variances.
- The user has designed a MARSS model with confounding parameters. Models with multiple confounded intercepts easy to design by accident.

B update is outside the unit circle

If you are estimating **B**, then if the absolute value of all the eigenvalues of **B** are less than 1, the system is stationary (meaning the **X**'s have some multivariate distribution that does not change over time). In this case, we say that **B** is within the unit circle. A pure univariate random walk for example would have $\mathbf{B} = 1$ and it is not stationary. The distribution of **X** for the pure random walk has a variance that increases with time. If on the other hand $|\mathbf{B}| < 1$, you have an Ornstein-Uhlenbeck process and is

stationary, with a stationary variance of $\mathbf{Q}/(1-\mathbf{B}^2)$ (note $\mathbf{B}$ is a scalar here because in this example $\mathbf{X}$ is univariate). If any of the eigenvalues (real part) are greater than 1, then the system will "explode"—it rapidly diverges.

In the EM algorithm, there is nothing to force $\mathbf{B}$ to be on or within the unit circle (real part of the eigenvalues less than or equal to 1). It is possible at one of the EM iterations the $\mathbf{B}$ update will be outside the unit circle. The problem is that if you get too far outside the unit circle, the algorithm becomes numerically unstable since small errors are magnified by the "explosive" $\mathbf{B}$ term. If you see the 'B outside the unit circle' warning, it is fine as long as it is temporary and the log-likelihood does not start decreasing (you will see a separate warning if that happens).

If you do see $\mathbf{B}$ outside the unit circle and the log-likelihood decreases, then it probably means that you have poorly specified the model somehow. An easy way to do this is to poorly specify the initial conditions, π and Λ. If, say, you try to specify a vague prior on $\mathbf{x}_0$ (or $\mathbf{x}_1$) with π equal to zero and Λ equal to a diagonal matrix with a large variance on the diagonal, you will likely run into trouble if $\mathbf{B}$ has off-diagonal terms. The reason is that by specifying that Λ is diagonal, you specified that the individual X's in $\mathbf{X}_0$ are independent, yet if $\mathbf{B}$ has off-diagonal terms, the stationary distribution of $\mathbf{X}_1$ is NOT independent. If you force the diagonal terms on Λ to be big enough, you can force the maximum-likelihood estimate of $\mathbf{B}$ to be outside the unit circle since this is the only way to account for $\mathbf{X}_0$ independent and $\mathbf{X}_1$ highly correlated.

The problem is that you will not know the stationary distribution of the $\mathbf{X}$'s (from which $\mathbf{X}_0$ was presumably drawn) without knowing the parameters you are trying to estimate. One approach is the estimate both π and Λ by setting `x0="unconstrained"` and `V0="unconstrained"` in the model specification. Estimating both π and Λ cannot be done robustly for all MARSS models, and in general, one probably wants to specify the model in such a way as to fix one or both of these. Another, more robust approach, is to treat $\mathbf{x}_1$ as fixed but unknown (instead of $\mathbf{x}_0$). You do this by setting `model$tinitx=1`, so that π refers to $t=1$ not $t=0$. Then estimate π and fix $\Lambda=0$. This eliminates Λ from the model and often eliminates the problems with prior specification—as the expense of m more parameters. Note, when you set $\Lambda=0$, Λ is truly eliminated from the model; the likelihood function is different, so do not expect $\Lambda=0$ and $\Lambda\sim 0$ to have the same likelihood under all conditions.

Warning! Reached maxit before parameters converged

The maximum number of EM iterations is set by `control$maxit`. If you get this warning, it means that one of the parameters or log-likelihood had not yet reached the convergence stopping criteria before `maxit` was reached. There are many situations where you might want to set `control$maxit` lower than the value needed to reach convergence. For example, if you are using the EM algorithm to produce initial values for a different algorithm (like a Bayesian MCMC algorithm or a Newton method) then you can set `maxit` low, say 20 or 50.

Stopped at iter=xx in MARSSkem() because numerical errors were generated in MARSSkf

This means the Kalman filter/smoother algorithm became unstable and most likely one of the variances became ill-conditioned. When that happens the inverses of those matrices are poor, and you will start to get negative values on the diagonals of your variance-covariance matrices. Once that happens, the inverse of that variance-covariance matrix produces an error. If you get this error, turn on tracing with `control$trace=1`. This will store the error messages so you can see what is going on. It may be that you have specified the model in such a way that some of the variances are being forced very close to 0, which makes the variance-covariance matrix ill-conditioned. The output from the MARSS call will be the parameter values just before the error occurred.

Warning: the xyz parameter value has not converged

The algorithm checks whether the log-likelihood and each individual parameter has converged. If a parameter has not converged, you can try upping `control$maxit` and see if it converges. If you set, `maxit` high, but the parameter is still not converging, then it suggests that one of the variance parameters is so small that the EM update steps for that parameter are tiny. For example, as $\mathbf{Q}$ goes to zero, the update steps for $\mathbf{u}$ go to zero. As Λ goes to zero, the update steps for π go to zero. The first thing to do is to reflect on whether you are inadvertently specifying the model in such a way that one of the variances is forced to zero. For example, if the total variance in $\mathbf{X}$ is 0.1 and you fix $\mathbf{R} = 0.2$ then $\mathbf{Q}$ must go to zero. The second thing to do is to try using a Newton algorithm, using your last EM values as the initial conditions for the Newton algorithm. The initial values are set using the `inits` argument for the `MARSS()` function.

MARSSkem: The solution became unstable and logLik DROPPED

This is a more serious error as in the EM algorithm, the log-likelihood should never drop. The first thing to do is check if you have specified a bizarre model or data, inadvertently. Plot the data you are trying to fit. Often, this error arises when a user has inadvertently scrambled their data order during a demeaning or variance-standardization step. Second, check the model you are trying to fit. Use `test=MARSS(data, model=xyz, fit=FALSE)` and then `summary(test$model)`. This shows you what `MARSS()` thinks your model is. You may be trying to fit an illogical model.

If those checks looks good, then pass `control$trace=1` into the `MARSS()` call. This will report a fuller set of warnings. Look if the error "B is outside the unit circle"

appears. If so, you are probably specifying a strange **B** matrix. Are you forcing the **B** matrix to be outside the unit circle (eigenvalues > 1)? If so, you need to rethink your **B** matrix constraints. If you do not see that error, look at `test$iter.record$logLik`. If the log-likelihood is steadily dropping (at each iteration) or drops by large amounts (much larger than the machine precision), that is bad and means that the EM algorithm did not work. If however the log-likelihood is just fluctuating by small amounts about some steady value, that is ok as it means that the values converged but the parameters are such that there are slight numerical fluctuations. Try passing `control$safe=TRUE` in the `MARSS()` call. This can sometimes help as it inserts a call to the Kalman filter after each individual parameter update.

Stopped at iter=xx in MARSSkem: solution became unstable. R (or Q) update is not positive definite

First check if you have specified an illegally constrained variance-covariance matrix. For example, if the variances (diagonal) are constrained to be equal, you cannot specify the covariances (off-diagonals) as unequal. Or if you specify that some of the covariances are equal, you cannot specify the variances as all unequal. These are illegal constraints on a variance-covariance matrix from a statistical perspective (nothing to do with {MARSS} package functions specifically).

This could also be due to numerical instability as **B** leaves the unit circle or one of the variance matrix becomes ill-conditioned. Try turning on tracing with `control$trace=1` and turn on safe with `control$safe=TRUE`. This will print out the error warnings at each parameter update step. Then consider whether you have inadvertently specified the model in such a way as to force this behavior in the **B** parameter.

You might also get this error if you inadvertantly specified an improper structure for **R** or **Q**. For example, if you used `R=diag(c(1,1,"r"))` with the intent of specifying a diagonal matrix with fixed variance 1 at $\mathbf{R}[1,1]$ and $\mathbf{R}[2,2]$ and an estimated $\mathbf{R}[3,3]$, you would have actually specified a character matrix with `"0"` on the off-diagonals and `c("1","1","r")` on the diagonal. `MARSS()` interprets all elements in quotes as names of parameters to be estimated. Thus it will estimate one off-diagonal covariance and two diagonal variances. That happens to put illegal constraints on estimation of a variance-covariance matrix having nothing to do with the `MARSS()` function but with estimation of variance-covariance matrices in general.

iter=xx MARSSkf: logLik computation is becoming unstable. Condition num. of Sigma[t=1] = Inf and of R = Inf.

This means, generally, that `V0` (Λ) is very small, say 0, and **R** diagonal elements are very small and very close to zero.

Warning: setting diagonal to 0 blocked at iter=X. logLik was lower in attempt to set 0 diagonals on X

This is a warning not an error. What is happening is that one of the variances (in **Q** or **R**) is getting small and the EM algorithm is attempting to set the value to 0 (because `control$degen.allow=TRUE`). But when it tried to do this, the new likelihood with the variance equal to 0 was lower and the variance was not set to 0.

A model with a variance minuscule and a model with the same variance equal to 0 are not the same model. In the first, a stochastic process with small variance exists but in the second, the analogous process is deterministic. And in the first case, you can get a situation where the likelihood term L(x|mean=mu,sigma=0) appears. That term will be infinite when x=mu. So in the model with variance minuscule, you will get very large likelihood values as the variance term gets smaller and smaller. In the analogous model with that variance set to 0, that likelihood term does not appear so the likelihood does not go to infinity.

This is not an error nor pathological behavior; the models are fundamentally different. Nonetheless, this will pose a dilemma when you want to chose the best model based on maximum likelihood. The model with minuscule variance will have infinite likelihood but the same behavior as the one with variance 0. In our experience, this dilemma arises when one has a lot of missing data near the beginning of the time series and is affected by how you specify the prior on the initial state. Try setting the prior at $t = 0$ versus $t = 1$. Try using a diffuse prior. You absolutely want to compare estimates using the BFGS and EM algorithms in this case, because the different algorithms differ in their ability to find the maximum in this strange case. Neither is uniformly better or worse. It seems to depend on which variance (**Q** or **R**) is going to zero.

Warning: kf returned error at iter=X in attempt to set 0 diagonals for X

This is a warning that the EM algorithm tried to set one of the diagonals of element X to 0 because `allow.degen` is TRUE and element X is going to zero. However when this was tried, the Kalman filter returned an error. Typically, this happens when both **R** and **Q** elements are both trying to be set to 0. If the maximum-likelihood estimate is that both **R** and **Q** are zero, it probably means that your MARSS model is not a very good description of the data.

Warning: At iter=X attempt to set 0 diagonals for R blocked for elements where corresponding rows of A or Z are not fixed.

You have `control$degen.allow=TRUE` and one of the **R** diagonal elements is getting very small. {MARSS} attempts to set these **R** elements to 0, but if row i of **R**

is 0, then the corresponding i rows of **a** and **Z** must be fixed. This is for the EM algorithm. It might work with the BFGS algorithm, or might spit out garbage without warning you. Always be suspicious when the EM and BFGS behavior is different. That is a good sign that something is wrong with how your model describes the data. It's not a problem with the algorithms per se; rather for certain pathological models, the algorithms behave differently from each other.

Stopped at iter=X in MARSSkem. XYZ is not invertible.

There are a series of checks in {MARSS} that check if matrix inversions are possible before doing the inversion. These errors crop up most often when **Q** or **R** are getting very small. At some point, they can get so small that inversions become unstable. If this error is given, then the output will be the last parameter estimates before the error. Try setting `control$allow.degen=FALSE`. Sometimes the error occurs when a diagonal element of **Q** or **R** is being set to 0. You will also have to set `control$maxit` to something smaller because the EM algorithm will not stop since the problematic diagonal element will walk slowly and inexorably to 0.

B

Package MARSS: Object structures

Model objects: class marssMODEL

Objects of class 'marssMODEL' specify Multivariate Autoregressive State Space (MARSS) models. The `model` component of an ML estimation object (class marssMLE; see below) is a marssMODEL object. These objects have the following components:

data An optional matrix (not data frame), in which each row is a time series (time across columns).

fixed A list with 8 matrices Z, A, R, B, U, Q, x0, V0, specifying which elements of each parameter are fixed.

free A list with 8 matrices Z, A, R, B, U, Q, x0, V0, specifying which elements of each parameter are to be estimated.

M An array of dim $n \times n \times T$ (an $n \times n$ missing values matrix for each time point). Each matrix is diagonal with 0 at the i,i value if the i-th value of **y** is missing, and 1 otherwise.

miss.value Deprecated. Replace missing values with NAs before passing to MARSS.

The matrices in `fixed` and `free` work as pairs to specify the fixed and free elements for each model parameter. The dimensions for `fixed` and `free` matrices are as follows, where n is the number of observation time series and m is the number of state processes:

Z n x m
B m x m
U m x 1
Q m x m
A n x 1
R n x n
x0 m x 1
V0 m x m

MARSSinputs

All the user inputs to a `MARSS()` call are put into a list and then passed to a function called `MARSS.form()` where `form` is the text specified by the form argument, e.g., `MARSS.marss()`. This function is used to create the marssMODEL object and then `MARSScheckinputs()` is called to error check the other arguments.

data A matrix (not data frame) of observations (rows) $\times$ time (columns).
model The specification is form dependent. For the default marxss form, the inputs are a list with up to 14 elements Z, A, R, B, U, Q, x0, V0, C, c, D, d, tinitx, diffuse
inits A list with up to 10 matrices Z, A, R, B, U, Q, x0, V0, C, D specifying initial values for parameters. Dimensions are given in the class 'marssMODEL' section.
miss.value Deprecated. Specifies missing value representation (default is NA).
method The method used for estimation: 'kem' for EM, 'BFGS' for quasi-Newton.
form The form to use to interpret the 'model' argument and create the marssMODEL object.
control List of estimation options. These are method dependent.

ML estimation objects: class marssMLE

Objects of class marssMLE specify maximum-likelihood estimation for a MARSS model, both before and after fitting. A minimal marssMLE object contains components `model, start` and `control`, which must be present for estimation by functions like `MARSSkem()`.

model MARSS model specification (an object of class 'marssMODEL').
start List with 7 matrices A, R, B, U, Q, x0, V0, specifying initial values for parameters. Dimensions are given in the class marssMODEL section.
control A list specifying estimation options. For `method="kem"`, these are
minit Minimum number of iterations in the maximization algorithm.
maxit Maximum number of iterations in the maximization algorithm.
abstol Optional tolerance for log-likelihood change. If log-likelihood decreases less than this amount relative to the previous iteration, the EM algorithm exits.
trace A positive integer. If not zero, a record will be created of each variable the maximization iterations. The information recorded depends on the maximization method.
safe If TRUE, `MARSSkem()` will rerun `MARSSkf()` after each individual parameter update rather than only after all parameters are updated.
silent Suppresses printing of progress bar and convergence information.

`MARSSkem()` appends the following components to the marssMLE' object:

method A string specifying the estimation method ('kem' for estimation by `MARSSkem()`).
par A list with 8 matrices of estimated parameter values Z, A, R, B, U, Q, x0, V0. If there are fixed elements in the matrices, the corresponding elements in `$par` are set to the fixed values.
kf A list containing Kalman filter/smoother output. See Chapter 2
numIter Number of iterations required for convergence.
convergence Convergence status.
logLik the exact Log-likelihood. See Section 3.4.
errors any error messages
iter.record record of the parameter values at each iteration (if `control$trace=1`)

Several functions append additional components to the 'marssMLE' object passed in. These include:

`MARSSaic()` Appends `AIC, AICc, AICbb, AICbp`, depending on the AIC flavors requested.
`MARSShessian()` Appends `Hessian, gradient, parMean` and `parSigma`.
`MARSSparamCIs()` Appends `par.se, par.bias, par.upCI` and `par.lowCI`.

C

Model specification in the core functions

Most users will not directly work with the core functions nor build marssMODEL objects from scratch. Instead, they will interact with the core functions via the function `MARSS()` described in Chapter 4. With the `MARSS()` function, the user specifies the model structure with matrices or text strings ("diagonal", "unconstrained", etc.) and `MARSS()` builds the marssMODEL object. However, a basic understanding of the structure of marssMODEL objects is useful if one wants to interact directly with the core functions.

C.1 The fixed and free components of the model parameters

In a marssMODEL object, each parameter is written in vec form and specificed by the equation of the form $\mathbf{f} + \mathbf{D}\boldsymbol{\beta}$ as in Equation 77 in Holmes (2012). $\mathbf{f}$ is the fixed matrix, $\mathbf{D}$ is the free matrix and $\boldsymbol{\beta}$ is the column vector of parameters. In a marssMODEL object, the `fixed` list has the $\mathbf{f}$ for each parameter matrix, the `free` list has the $\mathbf{D}$ matrix for each parameter matrix, and `par` in the marssMLE object has the $\boldsymbol{\beta}$ column vector of estimated parameters.

C.2 Examples

C.2.1 Q is a diagonal matrix with one variance value

In this case, there is only one value on the diagonal and the off-diagonals are 0. Thus there is only one estimated parameter and the fixed values are all 0.

$$\mathbf{Q} = \begin{bmatrix} \alpha & 0 & 0 \\ 0 & \alpha & 0 \\ 0 & 0 & \alpha \end{bmatrix}$$

`fixed$Q` is

$$\mathbf{f} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

`par$Q` is

$$\boldsymbol{\beta} = \begin{bmatrix} "alpha" \end{bmatrix}$$

and `free$Q` is

$$\mathbf{D} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

Notice that $\mathbf{f} + \mathbf{D}\boldsymbol{\beta}$ is the vec of $\mathbf{Q}$.

C.2.2 Q is a diagonal matrix with unique variance values

$$\mathbf{Q} = \begin{bmatrix} \alpha_1 & 0 & 0 \\ 0 & \alpha_2 & 0 \\ 0 & 0 & \alpha_3 \end{bmatrix}$$

The fixed matrix is the same with all 0s, but the par and free matrices change. `par$Q` is

$$\boldsymbol{\beta} = \begin{bmatrix} "alpha1" \\ "alpha2" \\ "alpha3" \end{bmatrix}$$

and `free$Q` is

$$\mathbf{D} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

C.2.3 Q has one variance and one covariance

$$\mathbf{Q} = \begin{bmatrix} \alpha & \beta & \beta \\ \beta & \alpha & \beta \\ \beta & \beta & \alpha \end{bmatrix}$$

The fixed vector is still the same, all zero. `par$Q` is

$$\boldsymbol{\beta} = \begin{bmatrix} "alpha" \\ "beta" \end{bmatrix}$$

and `free$Q` is

$$\mathbf{D} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \\ 1 & 0 \end{bmatrix}$$

C.2.4 Q is unconstrained

Since **Q** is a variance-covariance matrix, it must be symmetric across the diagonal.

$$\mathbf{Q} = \begin{bmatrix} \alpha_1 & \beta_1 & \beta_2 \\ \beta_1 & \alpha_2 & \beta_3 \\ \beta_2 & \beta_3 & \alpha_3 \end{bmatrix}$$

There are no fixed values in **Q** so **f** is still all zero. `par$Q` is

$$\boldsymbol{\beta} = \begin{bmatrix} "alpha1" \\ "beta1" \\ "beta2" \\ "alpha2" \\ "beta3" \\ "alpha3" \end{bmatrix}$$

and `free$Q` is

$$\mathbf{D} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

C.2.5 Q is fixed

For example,

$$\mathbf{Q} = \begin{bmatrix} 0.1 & 0 & 0 \\ 0 & 0.1 & 0 \\ 0 & 0 & 0.1 \end{bmatrix}$$

The `fixed$Q` matrix is

$$\begin{bmatrix} 0.1 \\ 0 \\ 0 \\ 0 \\ 0.1 \\ 0 \\ 0 \\ 0 \\ 0.1 \end{bmatrix}$$

There are no estimated parameters so the free matrix is 9×0 and the par matrix is 0×1.

C.3 Limits on the model forms that can be fit

The main limitation is that one must specify a model that has only one solution. The core functions will allow you to attempt to fit an improper model (one with multiple solutions). If you do this accidentally, it may or may not be obvious that you have a problem. The estimation functions may chug along happily and return some solution. Careful thought about your model structure and the structure of the estimated parameter matrices will help you determine if your model is under-constrained and unsolvable.

D

Textbooks and articles that use MARSS modeling for population modeling

Textbooks Describing the Estimation of Process and Non-process Variance

There are many textbooks on Kalman filtering and estimation of state-space models. The following are a sample of books on state-space modeling that we have found especially helpful.

Shumway, R. H., and D. S. Stoffer. 2006. Time series analysis and its applications. Springer-Verlag.

Harvey, A. C. 1989. Forecasting, structural time series models and the Kalman filter. Cambridge University Press.

Durbin, J., and S. J. Koopman. 2001. Time series analysis by state space methods. Oxford University Press.

Kim, C. J. and Nelson, C. R. 1999. State space models with regime switching. MIT Press.

King, R., G. Olivier, B. Morgan, and S. Brooks. 2009. Bayesian analysis for population ecology. CRC Press.

Giovanni, P., S. Petrone, and P. Campagnoli. 2009. Dynamic linear models in R. Springer-Verlag.

Pole, A., M. West, and J. Harrison. 1994. Applied Bayesian forecasting and time series analysis. Chapman and Hall.

Bolker, B. 2008. Ecological models and data in R. Princeton University Press.

West, M. and Harrison, J. 1997. Bayesian forecasting and dynamic models. Springer-Verlag.

Tsay, R. S. 2010. Analysis of financial time series. Wiley.

Maximum-likelihood papers

This is just a sample of the papers from the population modeling literature.

de Valpine, P. 2002. Review of methods for fitting time-series models with process and observation error and likelihood calculations for nonlinear, non-Gaussian state-space models. Bulletin of Marine Science 70:455-471.

de Valpine, P. and A. Hastings. 2002. Fitting population models incorporating process noise and observation error. Ecological Monographs 72:57-76.

de Valpine, P. 2003. Better inferences from population-dynamics experiments using Monte Carlo state-space likelihood methods. Ecology 84:3064-3077.

de Valpine, P. and R. Hilborn. 2005. State-space likelihoods for nonlinear fisheries time series. Canadian Journal of Fisheries and Aquatic Sciences 62:1937-1952.

Dennis, B., J.M. Ponciano, S.R. Lele, M.L. Taper, and D.F. Staples. 2006. Estimating density dependence, process noise, and observation error. Ecological Monographs 76:323-341.

Ellner, S.P. and E.E. Holmes. 2008. Resolving the debate on when extinction risk is predictable. Ecology Letters 11:E1-E5.

Erzini, K. 2005. Trends in NE Atlantic landings (southern Portugal): identifying the relative importance of fisheries and environmental variables. Fisheries Oceanography 14:195-209.

Erzini, K., Inejih, C. A. O., and K. A. Stobberup. 2005. An application of two techniques for the analysis of short, multivariate non-stationary time-series of Mauritanian trawl survey data ICES Journal of Marine Science 62:353-359.

Hinrichsen, R.A. and E.E. Holmes. 2009. Using multivariate state-space models to study spatial structure and dynamics. In Spatial Ecology (editors Robert Stephen Cantrell, Chris Cosner, Shigui Ruan). CRC/Chapman Hall.

Hinrichsen, R.A. 2009. Population viability analysis for several populations using multivariate state-space models. Ecological Modelling 220:1197-1202.

Holmes, E.E. 2001. Estimating risks in declining populations with poor data. Proceedings of the National Academy of Sciences of the United States of America 98:5072-5077.

Holmes, E.E. and W.F. Fagan. 2002. Validating population viability analysis for corrupted data sets. Ecology 83:2379-2386.

Holmes, E.E. 2004. Beyond theory to application and evaluation: diffusion approximations for population viability analysis. Ecological Applications 14:1272-1293.

Holmes, E.E., W.F. Fagan, J.J. Rango, A. Folarin, S.J.A., J.E. Lippe, and N.E. McIntyre. 2005. Cross validation of quasi-extinction risks from real time series: An examination of diffusion approximation methods. U.S. Department of Commerce, NOAA Tech. Memo. NMFS-NWFSC-67, Washington, DC.

Holmes, E.E., J.L. Sabo, S.V. Viscido, and W.F. Fagan. 2007. A statistical approach to quasi-extinction forecasting. Ecology Letters 10:1182-1198.

Kalman, R.E. 1960. A new approach to linear filtering and prediction problems. Journal of Basic Engineering 82:35-45.

Lele, S.R. 2006. Sampling variability and estimates of density dependence: a composite likelihood approach. Ecology 87:189-202.

Lele, S.R., B. Dennis, and F. Lutscher. 2007. Data cloning: easy maximum likelihood estimation for complex ecological models using Bayesian Markov chain Monte Carlo methods. Ecology Letters 10:551-563.

Lindley, S.T. 2003. Estimation of population growth and extinction parameters from noisy data. Ecological Applications 13:806-813.

Ponciano, J.M., M.L. Taper, B. Dennis, S.R. Lele. 2009. Hierarchical models in ecology: confidence intervals, hypothesis testing, and model selection using data cloning. Ecology 90:356-362.

Staples, D.F., M.L. Taper, and B. Dennis. 2004. Estimating population trend and process variation for PVA in the presence of sampling error. Ecology 85:923-929.

Zuur, A. F., and G. J. Pierce. 2004. Common trends in Northeast Atlantic squid time series. Journal of Sea Research 52:57-72.

Zuur, A. F., I. D. Tuck, and N. Bailey. 2003. Dynamic factor analysis to estimate common trends in fisheries time series. Canadian Journal of Fisheries and Aquatic Sciences 60:542-552.

Zuur, A. F., R. J. Fryer, I. T. Jolliffe, R. Dekker, and J. J. Beukema. 2003. Estimating common trends in multivariate time series using dynamic factor analysis. Environmetrics 14:665-685.

Bayesian papers

This is a sample of the papers from the population modeling and animal tracking literature.

Buckland, S.T., K.B. Newman, L. Thomas and N.B. Koestersa. 2004. State-space models for the dynamics of wild animal populations. Ecological modeling 171:157-175.

Calder, C., M. Lavine, P. Müller, J.S. Clark. 2003. Incorporating multiple sources of stochasticity into dynamic population models. Ecology 84:1395-1402.

Chaloupka, M. and G. Balazs. 2007. Using Bayesian state-space modelling to assess the recovery and harvest potential of the Hawaiian green sea turtle stock. Ecological Modelling 205:93-109.

Clark, J.S. and O.N. Bjørnstad. 2004. Population time series: process variability, observation errors, missing values, lags, and hidden states. Ecology 85:3140-3150.

Jonsen, I.D., R.A. Myers, and J.M. Flemming. 2003. Meta-analysis of animal movement using state space models. Ecology 84:3055-3063.

Jonsen, I.D, J.M. Flemming, and R.A. Myers. 2005. Robust state-space modeling of animal movement data. Ecology 86:2874-2880.

Meyer, R. and R.B. Millar. 1999. BUGS in Bayesian stock assessments. Can. J. Fish. Aquat. Sci. 56:1078-1087.

Meyer, R. and R.B. Millar. 1999. Bayesian stock assessment using a state-space implementation of the delay difference model. Can. J. Fish. Aquat. Sci. 56:37-52.

Meyer, R. and R.B. Millar. 2000. Bayesian state-space modeling of age-structured data: fitting a model is just the beginning. Can. J. Fish. Aquat. Sci. 57:43-50.

Newman, K.B., S.T. Buckland, S.T. Lindley, L. Thomas, and C. Fernández. 2006. Hidden process models for animal population dynamics. Ecological Applications 16:74-86.

Newman, K.B., C. Fernández, L. Thomas, and S.T. Buckland. 2009. Monte Carlo inference for state-space models of wild animal populations. Biometrics 65:572-583

Rivot, E., E. Prévost, E. Parent, and J.L. Baglinière. 2004. A Bayesian state-space modelling framework for fitting a salmon stage-structured population dynamic model to multiple time series of field data. Ecological Modeling 179:463-485.

Schnute, J.T. 1994. A general framework for developing sequential fisheries models. Canadian J. Fisheries and Aquatic Sciences 51:1676-1688.

Swain, D.P., I.D. Jonsen, J.E. Simon, and R.A. Myers. 2009. Assessing threats to species at risk using stage-structured state-space models: mortality trends in skate populations. Ecological Applications 19:1347-1364.

Thogmartin, W.E., J.R. Sauer, and M.G. Knutson. 2004. A hierarchical spatial model of avian abundance with application to cerulean warblers. Ecological Applications 14:1766-1779.

Trenkel, V.M., D.A. Elston, and S.T. Buckland. 2000. Fitting population dynamics models to count and cull data using sequential importance sampling. J. Am. Stat. Assoc. 95:363-374.

Viljugrein, H., N.C. Stenseth, G.W. Smith, and G.H. Steinbakk. 2005. Density dependence in North American ducks. Ecology 86:245-254.

Ward, E.J., R. Hilborn, R.G. Towell, and L. Gerber. 2007. A state-space mixture approach for estimating catastrophic events in time series data. Can. J. Fish. Aquat. Sci., Can. J. Fish. Aquat. Sci. 644:899-910.

Wikle, C.K., L.M. Berliner, and N. Cressie. 1998. Hierarchical Bayesian space-time models. Journal of Environmental and Ecological Statistics 5:117-154

Wikle, C.K. 2003. Hierarchical Bayesian models for predicting the spread of ecological processes. Ecology 84:1382-1394.

References

BIERNACKI, C., CELEUX, G., AND GOVAERT, G. 2003. Choosing starting values for the EM algorithm for getting the highest likelihood in multivariate gaussian mixture models. *Computational Statistics and Data Analysis* 41:561–575.

BROCKWELL, P. J. AND DAVIS, R. A. 1991. Time series: theory and methods. Springer-Verlag, New York, NY.

CAVANAUGH, J. E. AND SHUMWAY, R. H. 1997. A bootstrap variant of AIC for state-space model selection. *Statistica Sinica* 7:473–496.

CHEANG, W. K. AND REINSEL, G. C. 2000. Bias reduction of autoregressive estimates in time series regression model through restricted maximum likelihood. *Journal of the American Statistical Association* 95:1173–1184.

DE JONG, P. AND PENZER, J. 1998. Diagnosing shocks in time series. *Journal of the American Statistical Association* 93:796–806.

DEMPSTER, A., LAIRD, N., AND RUBIN, D. 1977. Likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society, Series B* 39:1–38.

DENNIS, B., MUNHOLLAND, P. L., AND SCOTT, J. M. 1991. Estimation of growth and extinction parameters for endangered species. *Ecological Monographs* 61:115–143.

DENNIS, B., PONCIANO, J. M., LELE, S. R., TAPER, M. L., AND STAPLES, D. F. 2006. Estimating density dependence, process noise, and observation error. *Ecological Monographs* 76:323–341.

DURBIN, J. AND KOOPMAN, S. J. 2012. Time series analysis by state space methods. Oxford University Press, Oxford, 2 edition.

ELLNER, S. P. AND HOLMES, E. E. 2008. Resolving the debate on when extinction risk is predictable. *Ecology Letters* 11:E1–E5.

FARAWAY, J. 2004. Linear models with R. CRC Press.

GERBER, L. R., DEMASTER, D. P., AND KAREIVA, P. M. 1999. Grey whales and the value of monitoring data in implementing the u.s. endangered species act. *Conservation Biology* 13:1215–1219.

GHAHRAMANI, Z. AND HINTON, G. E. 1996. Parameter estimation for linear dynamical systems. Technical Report CRG-TR-96-2, University of Toronto, Dept. of Computer Science.

HAMILTON, J. D. 1994. State-space models, pp. 3039–3080. *In* R. F. Engle and D. L. McFadden (eds.), Handbook of Econometrics, volume IV, chapter 50. Elsevier Science.

HAMPTON, S. E., HOLMES, E. E., PENDLETON, D. E., SCHEEF, L., SCHEUERELL, M. D., AND WARD, E. 2013. Quantifying effects of abiotic and biotic drivers on community dynamics with multivariate autoregressive (MAR) models. *Ecology* 94:2663–2669.

HAMPTON, S. E., IZMEST'EVA, L. R., MOORE, M. V., KATZ, S. L., DENNIS, B., AND SILOW, E. A. 2008. Sixty years of environmental change in the world's largest freshwater lake – Lake Baikal, Siberia. *Global Change Biology* 14:1947–1958.

HAMPTON, S. E., SCHEUERELL, M. D., AND SCHINDLER, D. E. 2006. Coalescence in the Lake Washington story: Interaction strengths in a planktonic food web. *Limnology and Oceanography* 51:2042–2051.

HAMPTON, S. E. AND SCHINDLER, D. E. 2006. Empirical evaluation of observation scale effects in community time series. *Oikos* 113:424–439.

HARVEY, A., KOOPMAN, S. J., AND PENZER, J. 1998. Messy time series: a unified approach. *Advances in Econometrics* 13:103–143.

HARVEY, A. C. 1989. Forecasting, structural time series models and the Kalman filter. Cambridge University Press, Cambridge, UK.

HARVEY, A. C. AND KOOPMAN, S. J. 1992. Diagnostic checking of unobserved components time series models. *Journal of Business and Economic Statistics* 10:377–389.

HARVEY, A. C. AND SHEPHARD, N. 1993. Structural time series models. *In* G. Maddala, C. Rao, and H. Vinod (eds.), Handbook of Statistics, volume 11, chapter 10. Elsevier Science Publishers, Amsterdam.

HELSKE, J. 2017. Kfas: Exponential family state space models in r. *Journal of Statistical Software* 78:1–39.

HINRICHSEN, R. AND HOLMES, E. E. 2009. Using multivariate state-space models to study spatial structure and dynamics, pp. 145–166. *In* R. S. Cantrell, C. Cosner, and S. Ruan (eds.), Spatial Ecology, chapter 8. CRC and Chapman Hall.

HOLMES, E. E. 2001. Estimating risks in declining populations with poor data. *Proceedings of the National Academy of Sciences of the United States of America* 98:5072–5077.

HOLMES, E. E. 2004. Beyond theory to application and evaluation: diffusion approximations for population viability analysis. *Ecological Applications* 14:1272–1293.

HOLMES, E. E. 2012. Derivation of the EM algorithm for constrained and unconstrained MARSS models. Technical report, arXiv:1302.3919 [stat.ME].

HOLMES, E. E., SABO, J. L., VISCIDO, S. V., AND FAGAN, W. F. 2007. A statistical approach to quasi-extinction forecasting. *Ecology Letters* 10:1182–1198.

HOLMES, E. E., WARD, E. J., AND WILLS, K. 2012. MARSS: Multivariate autoregressive state-space models for analyzing time-series data. *The R Journal* 4:11–19.

HOLMES, E. E. AND WARD, E. W. 2010. Analyzing noisy, gappy, and multivariate population abundance data: modeling, estimation, and model selection in a maximum-likelihood framework. Technical report, Northwest Fisheries Science Center, Mathematical Biology Program.

IVES, A. R. 1995. Measuring resilience in stochastic systems. *Ecological Monographs* 65:217–233.

IVES, A. R., ABBOTT, K. C., AND ZIEBARTH, N. L. 2010. Analysis of ecological time series with ARMA(p,q) models. *Ecology* 91:858–871.

IVES, A. R., CARPENTER, S. R., AND DENNIS, B. 1999. Community interaction webs and zooplankton responses to planktivory manipulations. *Ecology* 80:1405–1421.

IVES, A. R., DENNIS, B., COTTINGHAM, K. L., AND CARPENTER, S. R. 2003. Estimating community stability and ecological interactions from time-series data. *Ecological Monographs* 73:301–330.

JEFFRIES, S., HUBER, H., CALAMBOKIDIS, J., AND LAAKE, J. 2003. Trends and status of harbor seals in Washington State 1978-1999. *Journal of Wildlife Management* 67:208–219.

KALMAN, R. E. 1960. A new approach to linear filtering and prediction problems. *Journal of Basic Engineering* 82:35–45.

KLUG, J. L. AND COTTINGHAM, K. L. 2001. Interactions among environmental drivers: Community responses to changing nutrients and dissolved organic carbon. *Ecology* 82:3390–3403.

KOHN, R. AND ANSLEY, C. F. 1989. A fast algorithm for signal extraction, influence and cross-validation in state-space models. *Biometrika* 76:65–79.

KOOPMAN, S. J. 1993. Distrubance smoother for state space models. *Biometrika* 80:117–126.

KOOPMAN, S. J. AND DURBIN, J. 2000. Fast filtering and smoothing for non-stationary time series models. *Journal of American Statistical Assosiation* 92:1630–1638.

KOOPMAN, S. J., SHEPHARD, N., AND DOORNIK, J. A. 1999. Statistical algorithms for models in state space using SsfPack 2.2. *Econometrics Journal* 2:113–166.

LAMON III, E., CARPENTER, S. R., AND STOW, C. A. 1998. Forecasting PCB concentrations in Lake Michigan salmonids: a dynamic linear model approach. *Ecological Applications* 8:659–668.

LELE, S. R., DENNIS, B., AND LUTSCHER, F. 2007. Data cloning: easy maximum likelihood estimation for complex ecological models using Bayesian Markov Chain Monte Carlo methods. *Ecology Letters* 10:551–563.

MCLACHLAN, G. J. AND KRISHNAN, T. 2008. The EM algorithm and extensions. John Wiley and Sons, Inc., Hoboken, NJ, 2nd edition.

PENZER, J. 2001. Critical values for time series diagnostics. Technical report, Department of Statistics, London School of Economics.

PETRIS, G., PETRONE, S., AND CAMPAGNOLI, P. 2009. Dynamic linear Models with R. Use R! Springer, London.

POLE, A., WEST, M., AND HARRISON, J. 1994. Applied Bayesian forecasting and time series analysis. Chapman and Hall, New York.

RAUCH, H. E. 1963. Solutions to the linear smoothing problem. *IEEE Transactions on Automatic Control* 8:371–372.

RAUCH, H. E., TUNG, F., AND STRIEBEL, C. T. 1965. Maximum likelihood estimation of linear dynamical systems. *Journal of AIAA* 3:1445–1450.

SCHEUERELL, M. D. AND WILLIAMS, J. G. 2005. Forecasting climate induced changes in the survival of Snake River spring/summer Chinook salmon (Oncorhynchus tshawytscha). *Fisheries Oceanography* 14:448–457.

SCHWEPPE, F. C. 1965. Evaluation of likelihood functions for Gaussian signals. *IEEE Transactions on Information Theory* IT-r:294–305.

SHUMWAY, R. AND STOFFER, D. 2006. Time series analysis and its applications. Springer-Science+Business Media, LLC, New York, New York, 2nd edition.

SHUMWAY, R. H. AND STOFFER, D. S. 1982. An approach to time series smoothing and forecasting using the EM algorithm. *Journal of Time Series Analysis* 3:253–264.

STAPLES, D. F., TAPER, M. L., AND DENNIS, B. 2004. Estimating population trend and process variation for PVA in the presence of sampling error. *Ecology* 85:923–929.

STAUDENMAYER, J. AND BUONACCORSI, J. R. 2005. Measurement error in linear autoregressive models. *Journal of the American Statistical Association* 10:841–852.

STOFFER, D. S. AND WALL, K. D. 1991. Bootstrapping state-space models: Gaussian maximum likelihood estimation and the Kalman filter. *Journal of the American Statistical Association* 86:1024–1033.

TAPER, M. L. AND DENNIS, B. 1994. Density dependence in time series observations of natural populations: estimation and testing. *Ecological Monographs* 64:205–224.

TSAY, R. S. 2010. Analysis of financial time series. Wiley Series in Probability and Statistics. John Wiley and Sons, Inc., Hoboken, New Jersey, 3rd edition.

WARD, E. J., CHIRAKKAL, H., GONZÁLEZ-SUÁREZ, M., AURIOLES-GAMBOA, D., HOLMES, E. E., AND GERBER, L. 2010. Inferring spatial structure from time-series data: using multivariate state-space models to detect metapopulation structure of California sea lions in the Gulf of California, Mexico. *Journal of Applied Ecology* 1:47–56.

ZUUR, A. F., FRYER, R. J., JOLLIFFE, I. T., DEKKER, R., AND BEUKEMA, J. J. 2003. Estimating common trends in multivariate time series using dynamic factor analysis. *Environmetrics* 14:665–685.

Index