

MARSS R Package

Elizabeth E. Holmes, Eric J. Ward, Mark D. Scheuerell, and K. Wills

Version 3.11.3

Contents

1	Overview	13
1.1	MARSS model form	14
1.2	Example model	15
1.3	Notes	16
2	Quick Start	17
2.1	Model specification	18
2.2	Linear constraints	19
2.3	Time-varying parameters	20
2.4	Inputs <code>c</code> and <code>d</code>	21
2.5	Tips and Tricks	21
3	Data format	23
3.1	<code>ts</code> objects	24
3.2	<code>tsibble</code> objects	25
4	Model specification	27
4.1	General matrix specification	28
4.2	Linear constraints	29
4.3	Time-varying parameters	29
4.4	Examples for each parameter	30

5	Covariates format	37
6	Common output for fits	41
7	Univariate Models	45
7.1	Random walk with drift	45
7.2	Random walk with time-varying parameters	47
7.3	AR(1) observed with error	49
7.4	Linear regression (LR)	56
7.5	LR with AR(1) errors	58
7.6	LR with AR(1) errors and independent errors	63
7.7	LR with AR(1) errors driven by covariate	64
7.8	Flat level model	66
7.9	Linear trend model	69
7.10	Stochastic level model	71
7.11	Stochastic slope model	74
8	Multivariate Models	77
8.1	RW observed with multiple time series	78
8.2	Multiple time series each observing a RW	80
8.3	Different time series observing a RW	81
8.4	Trend observed with AR(1) error	84
8.5	Dynamic Factor Model with 3 trends	86
8.6	Linear constraints	87
9	Parameter estimates	93
10	States and smoothed estimates	95
11	Model fits	97

<i>CONTENTS</i>	5
12 Residuals	99
13 Confidence Intervals	101
14 Predictions and forecasts	103
15 Troubleshooting	107
16 Algorithm notes and cautions	109
16.1 Properly constrained models	109
16.2 Notes on the Kalman filter	110
16.3 Notes on the EM algorithm	110
16.4 Bias in variance estimates	111
16.5 Careful if specifying a prior on initial conditions	112
16.6 State-space form of ARMA(p,q) models	113
17 Other related packages	115

Preface

The **MARSS** R package allows you to fit a general class linearly constrained multivariate autoregressive state-space models using an EM algorithm (Holmes, 2013). Fitting using the BFGS algorithm (via `optim`) is also provided.

This manual covers the **MARSS** R package: what it does, how to set up your models, how to structure your input, and how to get different types of output. For vignettes showing how to use MARSS models to analyze data, see our companion book *Applied Time Series Analysis for Fisheries and Environmental Sciences* by Holmes, Scheuerell and Ward.

Installation

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

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

Installation of the development release

Users do not need to install the package from GitHub. Critical updates are published to CRAN. However, if you want/need to install the latest development release on GitHub, use:

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

```
install_github("nwfsc-timeseries/MARSS@*release")  
library(MARSS)
```

The master branch on GitHub is not a ‘release’. It has work leading up to a GitHub release. The code here may be broken though usually preliminary work is done on a development branch.

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

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

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

With R version 4+, Windows users may also need to run the following if installation fails due to warnings converted to errors:

```
Sys.setenv("R_REMOTES_NO_ERRORS_FROM_WARNINGS" = "true")
```

Authors

The authors are research scientists with the US Federal government. This work was conducted as part of their jobs for NOAA Fisheries and USGS, as such the work is in the public domain and cannot be copyrighted.

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

- Elizabeth E. Holmes <http://faculty.washington.edu/eeholmes>
- Mark D. Scheuerell <https://faculty.washington.edu/scheuerl>
- Eric J. Ward <http://faculty.washington.edu/warde>

Citation

Holmes, E.E., M.D. Scheuerell, E.J. Ward. Online MARSS Package Manual. Version 3.11.3. Accessed 2021-02-24. <https://nwfsc-timeseries.github.io/MARSS-Manual/>

Holmes, E.E., E.J. Ward, M.D. Scheuerell, and K. Wills. 2020. MARSS: Multivariate Autoregressive State-Space Modeling. R package version 3.11.3. <https://CRAN.R-project.org/package=MARSS>

Holmes, E.E., E.J. Ward, and K. Wills. 2012. MARSS: multivariate autoregressive state-space models for analyzing time-series data. *R Journal* 4(1): 11-19. <https://doi.org/10.32614/RJ-2012-002>

Acknowledgments

Early development of the MARSS functions into an R package was done by Kellie Wills, a contractor at NWFSC. Many students in our workshops on MAR and MARSS modeling and our winter course at the University of Washington have helped us refine the package. The Kalman filter and smoother functions used by the MARSS package are now those written by Jouni Helske for the **KFAS** package. Use of the **KFAS** Kalman functions has made the **MARSS** package faster and more stable.

The **MARSS** R package began as code developed to study noisy and gappy multivariate time-series data that are ubiquitous in fisheries and ecological research. Our initial forays into model estimation for these data were inspired by Chapter 6 in Shumway and Stoffer (2006) and Harvey (1989). Due to difficulties fitting these models, we gravitated towards the EM algorithm as outlined by Shumway and Stoffer (1982). The first version of the EM algorithm that E. E. Holmes wrote however was based on a 1996 technical report by Z. Ghahramani and G. E. Hinton (Ghahramani and Hinton, 1996). The Kalman filter and smoother was that given in Shumway and Stoffer (2006). This initial algorithm allowed only shared terms within the **u** matrix and generic constraints on the **Q** and **R** matrices, as this is what our work on population structure required.

During this time, the authors were working together at the Northwest Fisheries Science Center. M. D. Scheuerell provided the impetus and inspiration

to derive a general EM algorithm for MARSS models with linear constraints. He needed these models for research problems and in the course of deriving the EM algorithm for constrained \mathbf{Q} matrices, it became apparent how to derive the fully general case. Thus began a five-year project resulting in Holmes (2013). Subsequently, we also needed to work with partially deterministic systems, and thus the EM algorithm was extended to the case of partially deterministic MARSS models.

Later algorithms for residuals and inference from MARSS models were added (Holmes, 2014). This work was founded mainly on work by Andrew Harvey, Siem Jan Koopman, and Jeremy Penzer on inference and diagnostic checking of MARSS models (Harvey, 1989; Harvey and Koopman, 1992; Harvey et al., 1998; de Jong and Penzer, 1998). Extending the algorithm for the Fisher Information matrix in section 3.4.5 of Harvey (1989) to the general MARSS model in the **MARSS** package allowed rapid calculation of approximate confidence intervals. Algorithms for bootstrap AIC (Cavanaugh and Shumway, 1997) and innovations bootstrapping (Stoffer and Wall, 1991) were also added.

Part 1. Overview

In Part 1, the **MARSS** package and MARSS models are introduced. the format for inputs (data) and model specification are shown. Part 2 shows a series of short examples and Part 3 goes into output from **MARSS** fitted objects (class `marssMLE`) and MARSS models in general.

Chapter 1

Overview

MARSS stands for Multivariate Auto-Regressive(1) State-Space. The **MARSS** package (Holmes et al., 2012) is an R package for estimating the parameters of linear MARSS models with Gaussian errors. This class of model is extremely important in the study of linear stochastic dynamical systems, and these models are important in many different fields, including economics, engineering, genetics, physics and ecology. 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 (Holmes, 2013). The EM algorithm in the **MARSS** package allows you to apply linear constraints on all the parameters within the model matrices. Fitting via the BFGS algorithm is also provided in the package using R's `optim` function, but this is not the focus of the **MARSS** package.

MARSS, `MARSS()` and MARSS. MARSS model refers to the class of models which the **MARSS** package fits using, primarily, an EM algorithm. In the text, **MARSS** refers to the R package. Within the package, the main fitting function is `MARSS()`. When the class of model is being discussed, rather than the package or the function, MARSS (not bolded) is used. MAR model refers to a multivariate auto-regressive model while MARSS model refers to a MAR model with the observation (SS) component.

1.1 MARSS model form

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

$$\begin{aligned}\mathbf{x}_t &= \mathbf{B}_t \mathbf{x}_{t-1} + \mathbf{u}_t + \mathbf{C}_t \mathbf{c}_t + \mathbf{G}_t \mathbf{w}_t, \quad \mathbf{w}_t \sim \text{MVN}(0, \mathbf{Q}_t) \\ \mathbf{y}_t &= \mathbf{Z}_t \mathbf{x}_t + \mathbf{a}_t + \mathbf{D}_t \mathbf{d}_t + \mathbf{H}_t \mathbf{v}_t, \quad \mathbf{v}_t \sim \text{MVN}(0, \mathbf{R}_t) \\ \mathbf{x}_1 &\sim \text{MVN}(\boldsymbol{\pi}, \boldsymbol{\Lambda}) \text{ or } \mathbf{x}_0 \sim \text{MVN}(\boldsymbol{\pi}, \boldsymbol{\Lambda})\end{aligned}\tag{1.1}$$

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

The bolded terms are matrices with the following definitions:

- \mathbf{x} is a $m \times T$ matrix of states. Each \mathbf{x}_t is a realization of the random variable \mathbf{X}_t at time t .
- \mathbf{w} is a $m \times T$ matrix of the process errors. The process errors at time t are multivariate normal with mean 0 and covariance matrix \mathbf{Q}_t .
- \mathbf{y} is a $n \times T$ matrix of the observations. Some observations may be missing.
- \mathbf{v} is a $n \times T$ column vector of the non-process errors. The observation 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.
- $\boldsymbol{\pi}$ is either a parameter or a fixed prior. It is a $m \times 1$ matrix.
- $\boldsymbol{\Lambda}$ is either a parameter or a fixed prior. It is a $m \times m$ variance-covariance matrix.
- \mathbf{C}_t and \mathbf{D}_t are parameters and are $m \times p$ and $n \times q$ matrices.
- \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.

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. 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.

1.2 Example model

An unconstrained MARSS model, meaning all the elements in a parameter matrices are allowed to be different and none constrained to be equal or related.

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

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

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

A constrained MARSS model. The MARSS package allows you to specify constraints by fixing elements in a parameter matrix or specifying that some elements are estimated—or have a linear relationship to other elements. Here is an example:

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

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

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

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 .

1.3 Notes

Specification of a properly constrained model with a unique solution is the responsibility of the user because `MARSS()` has no way to tell if you have specified an insufficiently constrained model.

The code in the **MARSS** package is not particularly fast and EM algorithms are famously slow. You can try `method="BFGS"` and see if that is faster. For some models, it will be much faster and for others, much slower. BFGS can be quite sensitive to initial conditions. You can fit a model with EM a few iterations and then pass to the fit to BFGS, and it may do better.

```
fit1 <- MARSS(data, model=model.list, control=list(minit=10, maxit=10))
fit2 <- MARSS(data, model=model.list, method="BFGS", inits=fit1)
```


Chapter 2

Quick Start

The default MARSS model is:

$$\begin{aligned} \mathbf{x}_t &= \mathbf{B}_t \mathbf{x}_{t-1} + \mathbf{u}_t + \mathbf{C}_t \mathbf{c}_t + \mathbf{G}_t \mathbf{w}_t, \quad \mathbf{w}_t \sim \text{MVN}(0, \mathbf{Q}_t) \\ \mathbf{y}_t &= \mathbf{Z}_t \mathbf{x}_t + \mathbf{a}_t + \mathbf{D}_t \mathbf{d}_t + \mathbf{H}_t \mathbf{v}_t, \quad \mathbf{v}_t \sim \text{MVN}(0, \mathbf{R}_t) \\ \mathbf{x}_1 &\sim \text{MVN}(\boldsymbol{\pi}, \boldsymbol{\Lambda}) \text{ or } \mathbf{x}_0 \sim \text{MVN}(\boldsymbol{\pi}, \boldsymbol{\Lambda}) \end{aligned} \quad (2.1)$$

\mathbf{c} and \mathbf{d} are inputs (aka, exogenous variables or covariates or indicator variables) and must have no missing values. In most cases, \mathbf{G} and \mathbf{H} are fixed (not estimated) and must have no missing values. \mathbf{w}_t and \mathbf{v}_t are uncorrelated. The \mathbf{R} , \mathbf{Q} and $\boldsymbol{\Lambda}$ variance-covariance matrices can be set to zero (or partially to zero) to specify partially deterministic systems.

If you already work with models in the form of Equation (2.1), 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.

The call to fit a model in **MARSS** is

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

The data must be entered as a $n \times T$ matrix, a `ts` object (which will be converted to a $n \times T$ matrix) or a vector (which will be converted to a $1 \times T$ matrix). No missing time steps are allowed. Data missing for a time step are

specified with NA. NAs may appear anywhere in the data and do not need to line up across the rows in `data`.

`model.list` is a list that specifies the form of each of the parameter matrices in Equation (2.1). See next section and Chapter 4.

There are `plot` (and `autoplot`), `print`, `summary`, `coef`, `fitted`, `tsSmooth`, `residuals` and `predict` (and `forecast`) functions for `marssMLE` objects. See the man files, e.g. `?plot`, and Part 8.6 for examples.

2.1 Model specification

Model specification for `MARSS()` is one-to-one to your MARSS model written in matrix form. You can mix fixed and estimated parameters, linear constraints, and time-varying parameters. Zeros are allowed on the diagonals of \mathbf{Q} , \mathbf{R} and $\mathbf{\Lambda}$ to specify partially deterministic systems. There are text shortcuts for common forms of the parameter matrices and there are default forms. See Chapter 4.

Example: a mean-reverting random walk model with three observation time series:

$$\begin{aligned} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}_t &= \begin{bmatrix} b & 0 \\ 0 & b \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}_{t-1} + \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}_t, & \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}_t &\sim \text{MVN} \left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} q_{11} & q_{12} \\ q_{12} & q_{22} \end{bmatrix} \right) \\ \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}_t &= \begin{bmatrix} 1 & 1 \\ 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}_t + \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}_t, & \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}_t &\sim \text{MVN} \left(\begin{bmatrix} a_1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} r_{11} & 0 & 0 \\ 0 & r & 0 \\ 0 & 0 & r \end{bmatrix} \right) \\ & & \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}_0 &\sim \text{MVN} \left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \right) \end{aligned}$$

To fit with `MARSS()`, we translate this model into equivalent matrices (or arrays if time-varying). Matrices that combine fixed and estimated values are specified using a list matrix with numerical values for fixed values and character names for the estimated values.

```

B1 <- matrix(list("b", 0, 0, "b"), 2, 2)
U1 <- matrix(0, 2, 1)
Q1 <- matrix(c("q11", "q12", "q12", "q22"), 2, 2)
Z1 <- matrix(c(1, 0, 1, 1, 1, 0), 3, 2)
A1 <- matrix(list("a1", 0, 0), 3, 1)
R1 <- matrix(list("r11", 0, 0, 0, "r", 0, 0, 0, "r"), 3, 3)
pi1 <- matrix(0, 2, 1)
V1 <- diag(1, 2)
model.list <- list(B = B1, U = U1, Q = Q1, Z = Z1, A = A1, R = R1,
  x0 = pi1, V0 = V1, tinitx = 0)

```

If you print these out, you will see the one-to-one correspondence between the model in R and the math version of the model. Matrix names in the model list must be B, U, C, c, Q, Z, A, D, d, R, x0, and V0. In many cases you will use the default values and do not need to specify the parameter for the model list. The `tinitx` specifies whether the initial state for x is at $t = 1$ (`tinitx=1`) or $t = 0$ (`tinitx=0`).

The **MARSS** package is designed to handle linear constraints within the parameter matrices. Linear constraint means you can write the elements of the matrix as a linear equation of all the other elements. Time-varying parameters can also be handled.

2.2 Linear constraints

Your model can have simple linear constraints within all the parameters except **Q**, **R** and **A**. For example $1 + 2a - 3b$ is a linear constraint. When entering this value for you 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 change the matrices in the previous model to:

$$\mathbf{B} = \begin{bmatrix} b - 0.1 & 0 \\ 0 & b + 0.1 \end{bmatrix} \quad \mathbf{Q} = \begin{bmatrix} q_{11} & 0 \\ 1 & 0 \end{bmatrix} \quad \mathbf{Z} = \begin{bmatrix} z_1 - z_2 & 2z_1 \\ 0 & z_1 \\ z_2 & 0 \end{bmatrix}$$

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

```

B1 <- matrix(list("-0.1+1*b", 0, 0, "0.1+1*b"), 2, 2)
Q1 <- matrix(list("q11", 0, 0, 1), 2, 2)
Z1 <- matrix(list("1*z1+-1*z2", 0, "z2", "2*z1", "z1", 0), 3,
  2)
model.list <- list(B = B1, U = U1, Q = Q1, Z = Z1, A = A1, R = R1,
  x0 = pi1, V0 = V1, tinitx = 0)

```

You can call `toLatex()` on your model to make sure you and `MARSS()` agree on what model you are trying to fit:

```

fit <- MARSS(data, model=model.list)
toLatex(fit$model)

```

2.3 Time-varying parameters

You can pass in an array of T matrices for a time-varying parameter. T is the number of time-steps in your data and is the 3rd dimension in the parameter array.

$$\begin{aligned}
 \mathbf{x}_t &= \mathbf{B}_t \mathbf{x}_{t-1} + \mathbf{u}_t + \mathbf{C}_t \mathbf{c}_t + \mathbf{G}_t \mathbf{w}_t, & \mathbf{W}_t &\sim \text{MVN}(0, \mathbf{Q}_t) \\
 \mathbf{y}_t &= \mathbf{Z}_t \mathbf{x}_t + \mathbf{a}_t + \mathbf{D}_t \mathbf{d}_t + \mathbf{H}_t \mathbf{v}_t, & \mathbf{V}_t &\sim \text{MVN}(0, \mathbf{R}_t) \\
 \mathbf{x}_{t_0} &\sim \text{MVN}(\boldsymbol{\pi}, \boldsymbol{\Lambda})
 \end{aligned} \tag{2.2}$$

Note the time indexing. Make sure you enter your arrays such that the right parameter (or input) at time t lines up with \mathbf{x}_t , e.g. it is common for state equations to have \mathbf{B}_{t-1} lined up with \mathbf{x}_t so you might need to enter the \mathbf{B} array such that your \mathbf{B}_{t-1} is entered at \mathbf{B}_t in the R code.

The length of the 3rd dimension must be the same as your data. For example, say in your mean-reverting random walk model (the example on the first page) you wanted $\mathbf{B}(2,2)$ to be one value before $t = 20$ and another value after but $\mathbf{B}(1,1)$ to be time constant. You can pass in the following:

```

TT <- dim(data)[2]
B1 <- array(list(), dim=c(2,2,TT))
B1[, , 1:20] <- matrix(list("b", 0, 0, "b_1"), 2, 2)
B1[, , 21:TT] <- matrix(list("b", 0, 0, "b_2"), 2, 2)

```

Notice the specification is one-to-one to your \mathbf{B}_t matrices on paper.

2.4 Inputs \mathbf{c} and \mathbf{d}

In the MARSS model, \mathbf{C} and \mathbf{D} are the estimated parameters and \mathbf{c} and \mathbf{d} are the inputs. In the model list, \mathbf{c} and \mathbf{d} are entered as a matrix with T columns. The number of rows depends on the number of inputs.

Let's say you have temperature data and you want to include a linear effect of temperature that is different for each \mathbf{x} time series:

```
C1 <- matrix(c("temp1","temp2"),2,1)
model.list <- list(B=B1, U=U1, C=C1, c=temp, Q=Q1,
                  Z=Z1, A=A1, R=R1, x0=pi1, V=V1, tinitx=0)
```

If you want a factor effect, then you will need to recode your factor as a matrix with T columns and a row for each factor. See Chapter 5 for an example.

2.5 Tips and Tricks

- Use `plot(fit)` (or `autoplot(fit)`) to see a series of plots and diagnostics for your model.
- Try `MARSSinfo()` if you get errors you don't understand or fitting is taking a long time to converge.
- When fitting a model with `MARSS()`, pass in `silent=2` to see what `MARSS()` is doing. This puts it in verbose mode.
- Use `fit=FALSE` to set up a model without fitting. Let's say you do `fit <- MARSS(..., fit=FALSE)`. Now you can do `summary(fit$model)` to see what `MARSS()` thinks you are trying to fit. You can also try `toLatex(fit$model)` to make a LaTeX file and pdf version of your model (saved in the working directory). This loads the **Hmisc** package (and all its dependencies) and requires that you are able to process LaTeX files (e.g. you have the **tinytex** package).

- Let's say you specified your model with some text shortcuts, like `Q="unconstrained"`, but you want the list matrix form for a next step. `a <- summary(fit$model)` returns that list (invisibly). Because the model argument of `MARSS()` will understand a list of list matrices, you can pass in `model=a` to specify the model.
- `MARSSkfas(fit, return.kfas.model=TRUE)` will return your model in **KFAS** form (class `SSModel`), thus you can use all the functions available in the **KFAS** package on your model.

Chapter 3

Data format

The first argument to `MARSS()` is your data, aka your response variables or the **y** in your MARSS equation.

`MARSS(data, ...)`

Your data need to be a matrix with time across the columns ($n \times T$ matrix). Note, you can pass in a `ts` object or a vector (not data frame) and `MARSS()` will convert this to a matrix. Here is an example of a data matrix with three observation time series and six time steps. Note that NAs are fine and it is not necessary for all observation time series to have observations at time step t .

Tip: put rownames on your data matrix and those will be used in the output.

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

where NA denotes a missing value. However, the **MARSS** functions assume discrete time steps and you will need a value for each time step. Replace any missing time steps with NA.

Why does **MARSS** require your data in matrix form? Because **MARSS** will not make any guesses about your intentions. You must be 100% explicit in terms of what model you trying to fit and what you consider to be **data**.

MARSS models are used in many different fields in different ways. There is no *guess* that would work for all models. Instead **MARSS** requires that you write your model in matrix form, and then pass everything in in a format that is one-to-one with that mathematical model. That way **MARSS** knows exactly what you are trying to do.

3.1 ts objects

In base R, time-series data can be stored as a **ts** object. A **ts** object (time series object) stores information about the time steps of the data and often seasonal information (the quarter or month). If you pass in a **ts** object as data into **MARSS()**, it will convert this to a matrix but it will ignore any information about the seasonality. It will store this information, but it does not use it as part of model specification.

If you have your data stored as a **ts** object, then you may be using year and season (quarter, month) as covariates to estimate trend and seasonality. The next sections give examples of converting your data to matrix form with the season information.

3.1.1 Univariate ts object

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)
class(dat)
```

Notice that the class of **dat** is matrix, which is what we want. There are three rows, first is the response and the second and third are the covariates, Year and Quarter. When you call **MARSS()**, **dat["Temp",]** is the data. **dat[c("Yr", "Qtr"),]** are your covariates.

3.1.2 Multivariate **ts** object

In this example, we have two temperature readings, our responses, and a salinity reading, a covariate. 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)
dat <- t(dat)
```

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

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

3.2 **tsibble** objects

The **tsibble** package extends the tidyverse to temporal data and **tsibble** (or **tbl_ts**) objects are the way time-series data are stored in that package. There are many ways that you can transform a **tsibble** object into a matrix with each row being an observed time series.

```
library(tidyverse)
library(tsibble)
dat <- tourism %>% as_tibble %>% tidyr::spread(Quarter, Trips)
dat.matrix <- as.matrix(dat[, -1 * (1:3)])
```


Chapter 4

Model specification

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 matrix.

```
MARSS(data, model=list(...))
```

There are text shortcuts for common matrix structures:

- "identity", "diagonal and equal", "diagonal and unequal", "unconstrained", "equalvarcov", "zero", "unequal", "equal"
- Then there are some special shortcuts for **Z** and **a**.

The default model structures 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 $t = 1$.

The shortcuts and general specifications for each parameter are discussed below.

4.1 General matrix specification

`MARSS()` will fit a general class of constrained MARSS models with linear constraints.

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 the 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 \mathbf{Q} and \mathbf{U} , you will see they look exactly like \mathbf{Q} and \mathbf{u} written above. `MARSS()` will keep the fixed values fixed and estimate q , u_1 , and u_2 .

Note, shared values are not allowed across parameter matrices. If you have "a" in your \mathbf{Q} matrix and "a" in your \mathbf{R} matrix they will be estimated as different values: $Q.a$ and $R.a$. Sometimes you can get around this by rewriting your model. For example, incorporating \mathbf{u} in \mathbf{C} or \mathbf{u} into \mathbf{B} .

4.2 Linear constraints

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

$$\mathbf{B} = \begin{bmatrix} b - 0.1 & 0 \\ 0 & b + 0.1 \end{bmatrix} \quad \mathbf{Q} = \begin{bmatrix} q_{11} & 0 \\ 1 & 0 \end{bmatrix} \quad \mathbf{Z} = \begin{bmatrix} z_1 - z_2 & 2z_1 \\ 0 & z_1 \\ z_2 & 0 \end{bmatrix}$$

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

```
B1 <- matrix(list("-0.1+1*b", 0, 0, "0.1+1*b"), 2, 2)
Q1 <- matrix(list("q11", 0, 0, 1), 2, 2)
Z1 <- matrix(list("1*z1+-1*z2", 0, "z2", "2*z1", "z1", 0), 3,
  2)
model.list <- list(B = B1, U = U1, Q = Q1, Z = Z1, A = A1, R = R1,
  x0 = pi1, V0 = V1, tinitx = 0)
```

4.3 Time-varying parameters

All parameters can be time-varying. Specify these with 3-dimensional array where time is the 3rd dimension. If the data have T time steps, the 3rd dimension of your array must be equal to T . No text short-cuts allowed.

Example, for $t = 1$ to $t = 3$, **u** is

$$\begin{bmatrix} 0.01 \\ u_1 \\ u \end{bmatrix}$$

and for $t = 4$ to $t = 10$, **u** is

$$\begin{bmatrix} 0.02 \\ u_2 \\ u \end{bmatrix}$$

Specify this as

```
U <- array(list(0), dim = c(3, 1, 10))
U[, , 1:3] <- list(0.01, "u1", "u")
U[, , 4:10] <- list(0.02, "u2", "u")
```

The **MARSS** algorithms will become rather slow when you use time-varying parameters. Note, dynamic linear models are a way to have stochastic time-varying parameters. See the chapter in the ATSA book on this class of model.

4.4 Examples for each parameter

4.4.1 \mathbf{u} , \mathbf{a} and $\boldsymbol{\pi}$

\mathbf{u} , \mathbf{a} and $\boldsymbol{\pi}$ are matrices with one column and the options for specifying their structures are the same. \mathbf{a} has one special option, "**scaling**" described below. The allowable structures are shown using \mathbf{u} as an example. Note that you should be careful about specifying shared structure in $\boldsymbol{\pi}$ because you need to make sure the structure in $\boldsymbol{\Lambda}$ matches. For example, if you require that all the $\boldsymbol{\pi}$ values are shared (equal) then $\boldsymbol{\Lambda}$ cannot be a diagonal matrix since that would be saying that the $\boldsymbol{\pi}$ 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 \mathbf{u} . Each character string in \mathbf{u} is the name of one of the \mathbf{u} elements to be estimated. For example if `U=matrix(list(0.01,"u","u"),3,1)`, then \mathbf{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 \mathbf{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 strings indicate that each element of \mathbf{u} is estimated. If $m = 3$, then \mathbf{u} would have the form:

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

- `U="equal"`: There is only one value in \mathbf{u} :

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

- `U=matrix(c(),m,1)`, where the values in `c()` are all numerical values: \mathbf{u} is fixed and has no estimated values. If `U=matrix(c(0.01,1,-0.5),3,1)`, then \mathbf{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"`: \mathbf{u} is all zero:

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

The \mathbf{a} parameter has a special option, `"scaling"`, which is the default behavior. In this case, \mathbf{a} is treated like a scaling parameter. If there is only one \mathbf{y} row associated with an \mathbf{x} row, then the corresponding \mathbf{a} element is 0. If there are more than one \mathbf{y} rows associated with an \mathbf{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:

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

4.4.2 **Q**, **R**, **Λ**

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:

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

- $\mathbf{Q}=\text{matrix}(\mathbf{c()}, \mathbf{m}, \mathbf{m})$, where all values in $\mathbf{c}()$ are character strings: Each element in \mathbf{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 $\text{matrix}()$ so that its class is matrix. You must be careful that your \mathbf{Q} specifies a statistically valid variance-covariance matrix. $\text{MARSS}()$ will throw an error/warning telling you if it is not valid.
- $\mathbf{Q}=\text{matrix}(\mathbf{c()}, \mathbf{m}, \mathbf{m})$, where all values in $\mathbf{c}()$ are numeric values: Each element in \mathbf{Q} is fixed to the values in the matrix.
- $\mathbf{Q}=\text{"identity"}$: The \mathbf{Q} matrix is the identity matrix:

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

- $\mathbf{Q}=\text{"zero"}$: The \mathbf{Q} matrix is all zeros:

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

Be careful when setting $\mathbf{\Lambda}$ model structures. Mis-specifying the structure of $\mathbf{\Lambda}$ can have catastrophic, but difficult to discern, effects on your estimates. The default behavior for $\mathbf{\Lambda}$ is safe because it eliminates $\mathbf{\Lambda}$ from the model. However, MARSS models in the literature often use a diffuse diagonal $\mathbf{\Lambda}$, and you must be cautious when using that for general MARSS models. It can lead to mis-specification. What does “mis-specifying” mean? It means your $\mathbf{\Lambda}$ matrix which specifies the variance-covariance structure of the initial \mathbf{x} is fundamentally inconsistent with your model. For example if \mathbf{B} has off-diagonal elements, then $\mathbf{\Lambda}$ diagonal (diffuse or otherwise) is inconsistent with your model.

4.4.3 B

Like the variance-covariance matrices (\mathbf{Q} , \mathbf{R} and $\mathbf{\Lambda}$), \mathbf{B} can be specified with a list matrix to allow you to have both fixed and shared elements in the \mathbf{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 \mathbf{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.

4.4.4 \mathbf{Z}

Like \mathbf{B} and the variance-covariance matrices, \mathbf{Z} can be specified with a list matrix to allow you to have both fixed and estimated elements in \mathbf{Z} . If \mathbf{Z} is a square matrix, many of the same text shortcuts are available: "diagonal and equal", "diagonal and unequal", and "equalvarcov". If \mathbf{Z} is a design matrix (a matrix with only 0s and 1s and where the row sums are all equal to 1), then a special shortcut is available using `factor()` which allows you to specify which \mathbf{y} rows are associated with which \mathbf{x} rows.

- `Z=factor(c(1,1,1))`: All \mathbf{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 \mathbf{y} corresponds to a different hidden state trajectory. This is the default \mathbf{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 \mathbf{y} corresponds to one hidden state trajectory and the third \mathbf{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 \mathbf{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 \mathbf{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 character 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 \mathbf{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 \mathbf{Z} matrix. Be careful that you are specifying an identifiable model.
- \mathbf{Z} for dynamic linear models (DLM). In this case, you covariates will appear in the \mathbf{Z} matrix and \mathbf{Z} will be time-varying. Your regression coefficients appear in the \mathbf{x} part of the model. For example, for a DLM with a time-varying intercept and regression coefficient, your \mathbf{Z} would look like. `Temp <- rnorm(10) Z <- array(1, dim=c(1, 2, 10)) Z[,2,] <- Temp`

Chapter 5

Covariates format

Covariates (aka inputs or exogenous variables) appear as \mathbf{c} and \mathbf{d} in the MARSS model and their effects are in the \mathbf{C} and \mathbf{D} matrices. As usual, constraints (such as shared or fixed values) can be applied within the matrix, though not across the \mathbf{C} and \mathbf{D} matrices. No missing values are allowed in \mathbf{c} and \mathbf{d} .

$$\begin{aligned}\mathbf{x}_t &= \mathbf{B}_t \mathbf{x}_{t-1} + \mathbf{u}_t + \mathbf{C}_t \mathbf{c}_t + \mathbf{G}_t \mathbf{w}_t, \quad \mathbf{w}_t \sim \text{MVN}(0, \mathbf{Q}_t) \\ \mathbf{y}_t &= \mathbf{Z}_t \mathbf{x}_t + \mathbf{a}_t + \mathbf{D}_t \mathbf{d}_t + \mathbf{H}_t \mathbf{v}_t, \quad \mathbf{v}_t \sim \text{MVN}(0, \mathbf{R}_t) \\ \mathbf{x}_1 &\sim \text{MVN}(\boldsymbol{\pi}, \boldsymbol{\Lambda}) \text{ or } \mathbf{x}_0 \sim \text{MVN}(\boldsymbol{\pi}, \boldsymbol{\Lambda})\end{aligned}\tag{5.1}$$

Covariates are passed in the model list.

```
MARSS(data, model=list(c=..., d=...))
```

Your covariates must a matrix with time across the columns ($p \times T$ matrix) where p is the number of covariates. No NAs are allowed in covariates. See the chapters in the ATSA book dealing with missing values in your covariates.

Here is an example of a covariate matrix for 2 covariates.

$$\mathbf{y} = \begin{bmatrix} 1 & 2 & 5 & 7 & 3.2 & 8 \\ 2 & 5 & 3 & 8 & 5.1 & 10.2 \end{bmatrix}$$

Factor covariates

If your covariates are factors, like site number or month, and you are estimating a a value, i.e. level or intercept, for each then you will use a matrix with 0s and 1s. This is identical to how say `lm()` would translate your model with factors.

Say your covariate is quarters and you have 3 years of data:

```
covariate <- rep(paste0("q", 1:4), 3)
```

You translate this to a matrix with four rows and 10 columns. Each row is for a different quarter.

```
vals <- unique(covariate)
TT <- length(covariate)
p <- length(vals)
c <- matrix(0, p, TT)
for (i in 1:p) c[i, ] <- covariate == vals[i]
rownames(c) <- vals
c
```

	[,1]	[,2]	[,3]	[,4]	[,5]	[,6]	[,7]	[,8]	[,9]	[,10]	[,11]	[,12]
q1	1	0	0	0	1	0	0	0	1	0	0	0
q2	0	1	0	0	0	1	0	0	0	1	0	0
q3	0	0	1	0	0	0	1	0	0	0	1	0
q4	0	0	0	1	0	0	0	1	0	0	0	1

Part 2. Short Examples

The section shows a series of short examples for different types of models specified in MARSS structure. See the ATSA book for a wide variety of detailed examples and explanations.

The MARSS model has the form:

$$\begin{aligned}\mathbf{x}_t &= \mathbf{B}_t \mathbf{x}_{t-1} + \mathbf{u}_t + \mathbf{C}_t \mathbf{c}_t + \mathbf{G}_t \mathbf{w}_t, \text{ where } \mathbf{w}_t \sim \text{MVN}(0, \mathbf{Q}_t) \\ \mathbf{y}_t &= \mathbf{Z}_t \mathbf{x}_t + \mathbf{a}_t + \mathbf{D}_t \mathbf{d}_t + \mathbf{H}_t \mathbf{v}_t, \text{ where } \mathbf{v}_t \sim \text{MVN}(0, \mathbf{R}_t) \\ \mathbf{x}_1 &\sim \text{MVN}(\boldsymbol{\pi}, \boldsymbol{\Lambda}) \text{ or } \mathbf{x}_0 \sim \text{MVN}(\boldsymbol{\pi}, \boldsymbol{\Lambda})\end{aligned}$$

The \mathbf{x} on the left are the hidden states. The \mathbf{y} on the left are the observed data. Missing values are allowed in \mathbf{y} .

Chapter 6

Common output for fits

This is a brief summary of common outputs for model fits. The output are shown with the following simple fit:

```
dat <- cumsum(rnorm(100, 0, 0.5)) + rnorm(100, 0, 0.5)
fit <- MARSS(dat)
```

Use the **broom** package to get the model output in **tidy** form. The confidence intervals shown are approximate and based on the estimated Hessian matrix. See `?tidy.marssMLE` for information on how to change to a different type of confidence interval.

```
broom::tidy(fit)
```

	term	estimate	std.error	conf.low	conf.up
1	R.R	0.12727996	0.06250867	0.004765227	0.2497947
2	U.U	-0.09669412	0.06244193	-0.219078047	0.0256898
3	Q.Q	0.38394414	0.10885937	0.170583701	0.5973046
4	x0.x0	0.01028019	0.70069260	-1.363052072	1.3836125

The get the estimated states use:

```
head(stats::tsSmooth(fit, type = "xtT"))
```

	.rownames	t	.estimate	.se
1	X.Y1	1	-0.08649281	0.2825700
2	X.Y1	2	0.20588708	0.2886156
3	X.Y1	3	0.09379051	0.2888742
4	X.Y1	4	-0.46718796	0.2888854
5	X.Y1	5	-1.30803192	0.2888859
6	X.Y1	6	-1.16070018	0.2888859

Note that generic rownames were given since none were specified for `dat`. You can also get all the Kalman filter and smoother estimates for \mathbf{x} from `MARSSkf()`. See `?MARSSkf`.

There are two types of fitted values that are used in the state-space literature: the one-step-ahead which uses on the data up to $t - 1$ and the smoothed fitted values which uses all the data. Read up on fitted values for MARSS models at `?fitted.marssMLE`.

To get the fitted values, the estimated \mathbf{y} , using all the data:

```
head(fitted(fit, type = "ytT"))
```

	.rownames	t	y	.fitted
1	Y1	1	-0.2154995	-0.08649281
2	Y1	2	0.3399737	0.20588708
3	Y1	3	0.2425978	0.09379051
4	Y1	4	-0.3744107	-0.46718796
5	Y1	5	-1.6356185	-1.30803192
6	Y1	6	-1.1682209	-1.16070018

To get the fitted values, the estimated \mathbf{y} , using the data up to time $t - 1$:

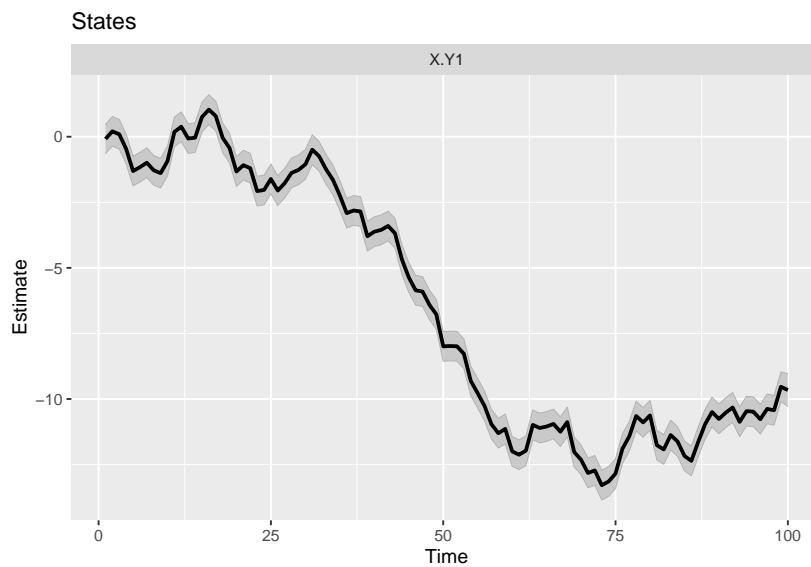
```
head(fitted(fit, type = "ytt1"))
```

	.rownames	t	y	.fitted
1	Y1	1	-0.2154995	-0.08641393
2	Y1	2	0.3399737	-0.28005507
3	Y1	3	0.2425978	0.11322799
4	Y1	4	-0.3744107	0.11898963
5	Y1	5	-1.6356185	-0.36849468
6	Y1	6	-1.1682209	-1.46879885

Note, this is the default for `fitted(fit)`.

To make some generic plots and diagnostic plots, you can use `autoplot()` in the **ggplot2** package:

```
ggplot2::autoplot(fit, plot.type = "xtT")
```



To get residuals, use:

```
resids <- residuals(fit)
head(format(resids, digits = 2))
```

	type	.rownames	name	t	value	.fitted	.resids	.sigma	.std.resids
1	ytt1		Y1 model	1	-0.22	-0.086	-0.1291	0.71	-0.1805
2	ytt1		Y1 model	2	0.34	-0.280	0.6200	0.78	0.7959
3	ytt1		Y1 model	3	0.24	0.113	0.1294	0.78	0.1654
4	ytt1		Y1 model	4	-0.37	0.119	-0.4934	0.78	-0.6307
5	ytt1		Y1 model	5	-1.64	-0.368	-1.2671	0.78	-1.6197
6	ytt1		Y1 model	6	-1.17	-1.469	0.3006	0.78	0.3842

There are many types of residuals possible for MARSS models. The default `residuals(fit)` will return the innovations residuals, which are what are typically used for residuals diagnostics for state-space models. However `residuals()` and the companion `MARSSresiduals()` will return all possible residuals for your model. Read about MARSS residuals in `?residuals.marssMLE`.

To get your parameter estimates, use:

```
coef(fit)
```

Type `?coef.marssMLE` to see the different forms that you can get your estimated parameters in. A common form is as a matrix. To get that use `coef(fit, type="matrix")`.

```
coef(fit, type = "matrix")$Q
```

```
      [,1]
[1,] 0.3839441
```

Chapter 7

Univariate Models

These models can be written in the form:

$$\begin{aligned}x_t &= b_t x_{t-1} + u_t + \beta_t c_t + w_t, \quad w_t \sim N(0, q_t) \\y_t &= z_t x_t + a_t + \psi_t d_t + v_t, \quad v_t \sim N(0, r_t)\end{aligned}\tag{7.1}$$

where c_t and d_t are inputs (no missing values) and y_t are observations (missing values allowed). x_t are the states. Everything else is an estimated parameter. Parameters can be time-varying. If time-varying, then specified as a 3D array with time in the 3rd dimension.

7.1 Random walk with drift

A univariate random walk with drift observed with error is:

$$\begin{aligned}x_t &= x_{t-1} + u + w_t, \quad w_t \sim N(0, q) \\y_t &= x_t + v_t, \quad v_t \sim N(0, r)\end{aligned}\tag{7.2}$$

Create some data from this equation:

```
u <- 0.01
r <- 0.01
q <- 0.1
TT <- 100
yt <- cumsum(rnorm(TT, u, sqrt(q))) + rnorm(TT, 0, sqrt(r))
```

Fit with MARSS():

```
fit <- MARSS(yt)
```

Success! abstol and log-log tests passed at 35 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 35 iterations.
Log-likelihood: -24.85696
AIC: 57.71391 AICc: 58.13496

	Estimate
R.R	0.0184
U.U	0.0408
Q.Q	0.0633
x0.x0	-0.2653

Initial states (x0) defined at t=0

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

Get confidence intervals with tidy():

```
broom::tidy(fit)
```

	term	estimate	std.error	conf.low	conf.up
1	R.R	0.01843982	0.009894122	-0.0009523011	0.03783194
2	U.U	0.04083656	0.025345531	-0.0088397685	0.09051289
3	Q.Q	0.06328006	0.017784821	0.0284224487	0.09813767
4	x0.x0	-0.26534896	0.281394026	-0.8168711121	0.28617320

The get the estimated states:

```
head(stats::tsSmooth(fit, type = "xtT"))
```

	.rownames	t	.estimate	.se
1	X.Y1	1	-0.2246253	0.1098830
2	X.Y1	2	-0.1370475	0.1118654
3	X.Y1	3	0.1814517	0.1119369
4	X.Y1	4	0.2708522	0.1119395
5	X.Y1	5	0.3617262	0.1119396
6	X.Y1	6	0.7986076	0.1119396

The get the fitted values (note these are the smoothed fitted values conditioned on all the data):

```
head(fitted(fit, type = "ytT"))
```

	.rownames	t	y	.fitted
1	Y1	1	-0.2382786	-0.2246253
2	Y1	2	-0.2043381	-0.1370475
3	Y1	3	0.2482112	0.1814517
4	Y1	4	0.2704228	0.2708522
5	Y1	5	0.2608996	0.3617262
6	Y1	6	0.9039099	0.7986076

Read up on fitted values for MARSS models at `?fitted.marssMLE`.

7.2 Random walk with time-varying parameters

Let's fit a random walk where the first 50 time steps have one process variance and the rest of the time series has a different process variance. The model is

$$\begin{aligned}x_t &= x_{t-1} + u + w_t, \quad w_t \sim N(0, q_t) \\ y_t &= x_t + v_t, \quad v_t \sim N(0, r)\end{aligned}\tag{7.3}$$

Create some data:

```

set.seed(123)
TT <- 100
sT <- 50
u <- 0.01
r <- 0.01
q <- c(rep(0.1, sT), rep(0.2, TT - sT))
yt <- cumsum(rnorm(TT, u, sqrt(q))) + rnorm(TT, 0, sqrt(r))

```

Fit with MARSS:

```

Q <- array(0, dim = c(1, 1, TT))
Q[1, 1, ] <- c(rep("q1", sT), rep("q2", TT - sT))
fit <- MARSS(yt, model = list(Q = Q), silent = TRUE)
broom::tidy(fit)

```

	term	estimate	std.error	conf.low	conf.up
1	R.R	0.02103312	0.01287598	-0.004203342	0.04626958
2	U.U	0.04315283	0.02999130	-0.015629036	0.10193470
3	Q.q1	0.06849550	0.02514959	0.019203209	0.11778779
4	Q.q2	0.12424866	0.03710242	0.051529256	0.19696806
5	x0.x0	-0.26716186	0.29456436	-0.844497393	0.31017367

We can do the same with the drift term, u . We have one u for the first 50 time steps and another for the last 50. Let's fit a random walk where the first 50 time steps have one process variance and the rest of the time series has a different process variance. The model is

$$\begin{aligned}
 x_t &= x_{t-1} + u + w_t, \quad w_t \sim N(0, q_t) \\
 y_t &= x_t + v_t, \quad v_t \sim N(0, r)
 \end{aligned}
 \tag{7.4}$$

Create some data:

```

set.seed(123)
q <- 0.1
u <- c(rep(0.1, sT), rep(-0.1, TT - sT))
yt <- cumsum(rnorm(TT, u, sqrt(q))) + rnorm(TT, 0, sqrt(r))

```


Fit with MARSS:

```
U <- array(0, dim = c(1, 1, TT))
U[1, 1, ] <- c(rep("u1", sT), rep("u2", TT - sT))
fit <- MARSS(yt, model = list(U = U), silent = TRUE)
broom::tidy(fit)
```

	term	estimate	std.error	conf.low	conf.up
1	R.R	0.01878371	0.009873106	-0.0005672203	0.03813464
2	U.u1	0.11495619	0.035842744	0.0447057067	0.18520668
3	U.u2	-0.05358255	0.035480829	-0.1231236961	0.01595860
4	Q.Q	0.06237204	0.017600922	0.0278748647	0.09686921
5	x0.x0	-0.24493471	0.281911468	-0.7974710376	0.30760161

7.3 AR(1) observed with error

With the addition of b in front of x_{t-1} we have an AR(1) process.

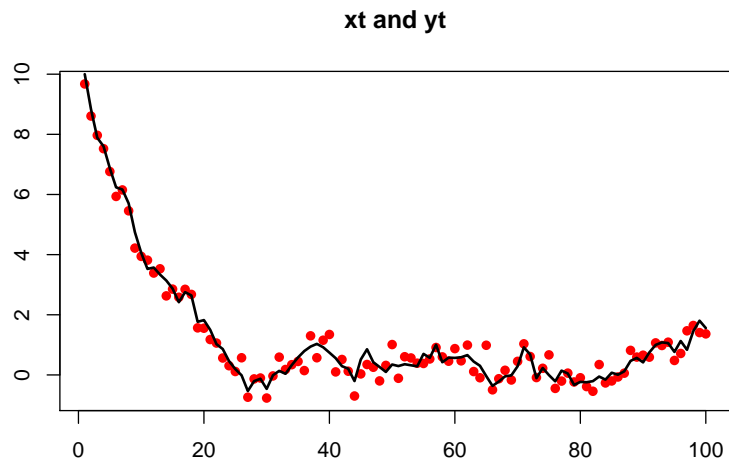
$$\begin{aligned} x_t &= bx_{t-1} + u + w_t, \quad w_t \sim N(0, q) \\ y_t &= x_t + v_t, \quad v_t \sim N(0, r) \end{aligned} \tag{7.5}$$

Create some simulated, non-stationary, AR(1) data:

```
set.seed(123)
u <- 0.01
r <- 0.1
q <- 0.1
b <- 0.9
TT <- 100
x0 <- 10
xt.ns <- rep(x0, TT)
for (i in 2:TT) xt.ns[i] <- b * xt.ns[i - 1] + u + rnorm(1, 0,
  sqrt(q))
yt.ns <- xt.ns + rnorm(TT, 0, sqrt(r))
```

The data were purposefully made to be non-stationary by setting x_0 well outside the stationary distribution of x . The EM algorithm in **MARSS** does not require that the underlying state processes be stationary and it is not necessary to remove the initial non-stationary part of the time-series.

```
plot(yt.ns, xlab = "", ylab = "", main = "xt and yt", pch = 16,
     col = "red")
lines(xt.ns, lwd = 2)
```



```
fit <- MARSS(yt.ns, model = list(B = matrix("b")))
```

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: -61.67475

AIC: 133.3495 AICc: 133.9878

```

      Estimate
R.R      0.1075
B.b      0.9042
U.U      0.0346
Q.Q      0.0535
x0.x0    10.6409
Initial states (x0) defined at t=0

```

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

We could also simulate AR(1) data with `stats::arima.sim()` however this will produce stationary data:

```

xt.s <- arima.sim(n = TT, model = list(ar = b), sd = sqrt(q))
yt.s <- xt.s + rnorm(TT, 0, sqrt(r))
yt.s <- as.vector(yt.s)
xt.s <- as.vector(xt.s)

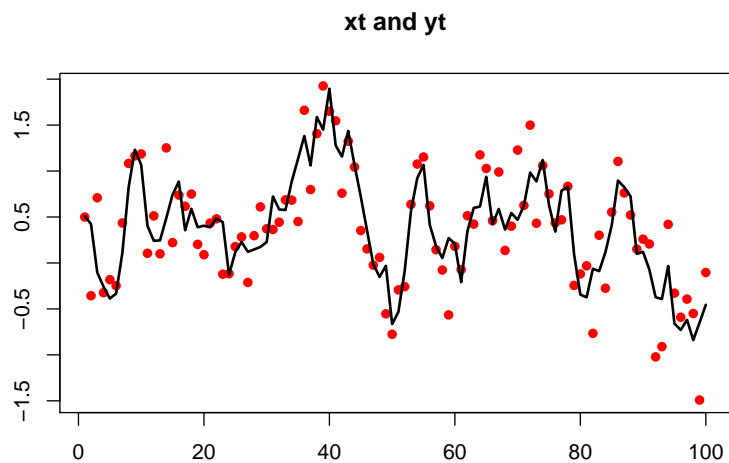
```

These stationary data can be fit as before but the data must be a matrix with time across the columns not a `ts` object. If you pass in a vector, `MARSS()` will convert that to a matrix with one row.

```

plot(yt.s, xlab = "", ylab = "", main = "xt and yt", pch = 16,
     col = "red", type = "p")
lines(xt.s, lwd = 2)

```



```
fit <- MARSS(yt.s, model = list(B = matrix("b")))
```

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: -74.40797

AIC: 158.8159 AICc: 159.4542

Estimate

R.R 0.1064

B.b 0.7884

U.U 0.0735

Q.Q 0.1142

x0.x0 0.2825

Initial states (x0) defined at t=0

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

Note that u is estimated however `arima.sim()` does not include a u . We can set u to zero if we happened to know that it was zero.

```
fit <- MARSS(yt.s, model = list(B = matrix("b"), U = matrix(0)))
```

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: -75.80277
 AIC: 159.6055 AICc: 160.0266

	Estimate
R.R	0.117
B.b	0.874
Q.Q	0.100
x0.x0	0.339

Initial states (x0) defined at t=0

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

If we know r (or q), we could set those too:

```
fit <- MARSS(yt.s, model = list(B = matrix("b"), U = matrix(0),  
                                R = matrix(r)))
```

Success! `abstol` and `log-log` tests passed at 18 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 18 iterations.
Log-likelihood: -75.88583
AIC: 157.7717   AICc: 158.0217
```

```
      Estimate
B.b      0.859
Q.Q      0.114
x0.x0    0.370
Initial states (x0) defined at t=0
```

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

We can fit to just the x data, an AR(1) with no error, by setting r to zero:
If we know r (or q), we could set those too:

```
fit <- MARSS(xt.s, model = list(B = matrix("b"), U = matrix(0),
  R = matrix(0)))
```

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: -26.9401
AIC: 59.8802   AICc: 60.1302
```

```
      Estimate
B.b      0.883
```

```
Q.Q      0.100
x0.x0    0.578
Initial states (x0) defined at t=0
```

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

We can fit `xt.s` with `arima()` also. The results will be similar but not identical because `arima()`'s algorithm assumes the data come from a stationary process and the initial conditions are treated differently.

```
arima(xt.s, order = c(1, 0, 0), include.mean = FALSE, method = "ML")
```

Call:

```
arima(x = xt.s, order = c(1, 0, 0), include.mean = FALSE, method = "ML")
```

Coefficients:

```
      ar1
      0.8793
s.e.    0.0454
```

```
sigma^2 estimated as 0.1009:  log likelihood = -27.98,  aic = 59.96
```

If we try fitting the non-stationary data with `arima()`, the estimates will be poor since `arima()` assumes stationary data:

```
arima(xt.ns, order = c(1, 0, 0), include.mean = FALSE, method = "ML")
```

Call:

```
arima(x = xt.ns, order = c(1, 0, 0), include.mean = FALSE, method = "ML")
```

Coefficients:

```
      ar1
      0.9985
s.e.    0.0021
```

```
sigma^2 estimated as 0.1348:  log likelihood = -44.59,  aic = 93.19
```

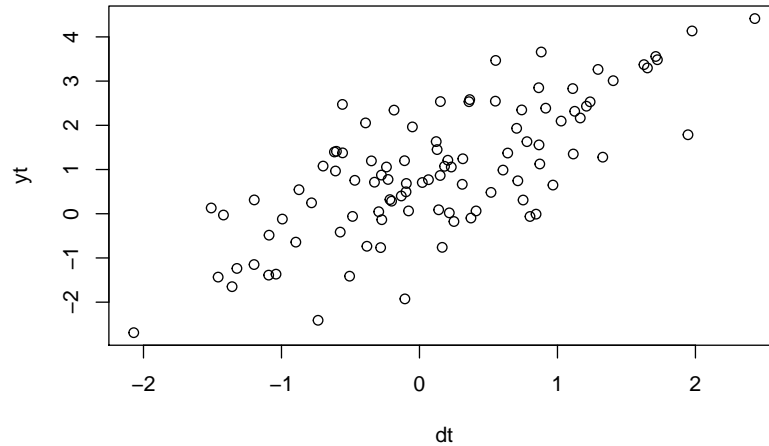
7.4 Linear regression (LR)

A simple linear regression of one covariate is written:

$$y_t = \alpha + \beta d_t + v_t, \quad w_t \sim N(0, q) \quad (7.6)$$

Let's create some simulated data with this structure:

```
beta <- 1.1
alpha <- 1
r <- 1
dt <- rnorm(TT, 0, 1) #our covariate
vt <- rnorm(TT, 0, r)
yt <- alpha + beta * dt + vt
plot(dt, yt)
```



To fit this model, we need to write it in MARSS form. Here's the parts we need with the parameters, we don't need removed.

$$\begin{aligned} \mathbf{x}_t &= \mathbf{x}_{t-1} + \mathbf{w}_t, \quad \mathbf{w}_t \sim \text{MVN}(0, 0) \\ \mathbf{y}_t &= 0 \times \mathbf{x}_t + \mathbf{D}\mathbf{d}_t + \mathbf{v}_t, \quad \mathbf{v}_t \sim \text{MVN}(0, \mathbf{R}) \end{aligned} \quad (7.7)$$

We write out our parameters in matrix form. We want to set most of the \mathbf{x} parameters to 0 so the algorithm doesn't try to estimate them.

```
R <- matrix("r") # no v_t
D <- matrix("beta")
d <- matrix(dt, nrow = 1)
Z <- matrix(0)
A <- matrix("alpha")
Q <- U <- x0 <- matrix(0)
```

MARSS() requires \mathbf{d} be a matrix also. Each row is a covariate and each column is a time step. No missing values allowed as this is an input.

```
mod.list <- list(U = U, Q = Q, x0 = x0, Z = Z, A = A, D = D,
  d = d, R = R)
fit <- MARSS(yt, model = mod.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: -140.495

AIC: 286.9899 AICc: 287.2399

	Estimate
A.alpha	0.810
R.r	0.972
D.beta	1.227

Initial states (x0) defined at t=0

Standard errors have not been calculated.

Use MARSSparamCIs to compute CIs and bias estimates.

The estimates are the same as with `lm()`:

```
lm(yt ~ dt)
```

Call:

```
lm(formula = yt ~ dt)
```

Coefficients:

(Intercept)	dt
0.8096	1.2273

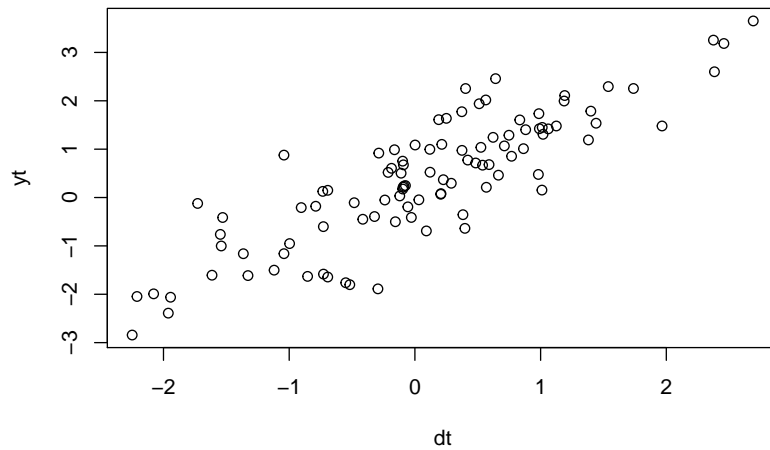
7.5 LR with AR(1) errors

A simple linear regression of one covariate with AR(1) errors is written:

$$\begin{aligned} x_t &= bx_{t-1} + w_t, \quad w_t \sim N(0, q) \\ y_t &= \beta d_t + x_t \end{aligned} \tag{7.8}$$

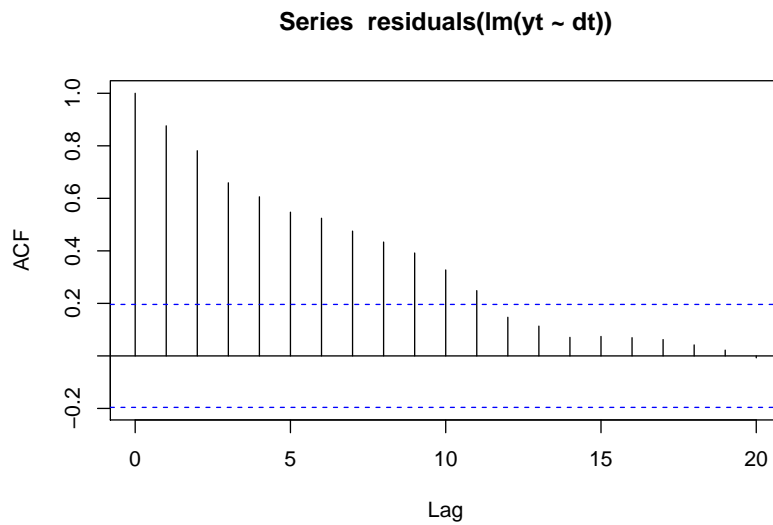
Let's create some simulated data with this structure:

```
beta <- 1.1
dt <- rnorm(TT, 0, 1) #our covariate
wt <- arima.sim(n = TT, model = list(ar = b), sd = sqrt(q))
yt <- beta * dt + wt
yt <- as.vector(yt) # not ts object
plot(dt, yt)
```



If we looked at an ACF of the residuals of a linear regression, we'd see that the residuals are highly autocorrelated:

```
acf(residuals(lm(yt ~ dt)))
```



We can fit this model (Equation (7.8)) with `MARSS()`. Please note that there are many better R packages specifically designed for linear regression models with correlated errors. This simple example is to help you understand model specification with the **MARSS** package.

To fit this model, we need match our Equation (7.8) with the full MARSS model written in matrix form (Equation (5)). Here it is with the parameters that are zero dropped. \mathbf{Z}_t is identity and is also dropped. The \mathbf{B} and \mathbf{D} are time-constant so the t subscript is dropped. The \mathbf{x}_t are the AR(1) errors and the \mathbf{y}_t is the linear regression with \mathbf{D} being the effect sizes and the \mathbf{d} being the covariate.

$$\begin{aligned}\mathbf{x}_t &= \mathbf{B}\mathbf{x}_{t-1} + \mathbf{w}_t, \text{ where } \mathbf{w}_t \sim \text{MVN}(0, \mathbf{Q}_t) \\ \mathbf{y}_t &= \mathbf{x}_t + \mathbf{D}\mathbf{d}_t\end{aligned}\tag{7.9}$$

Here is what the model looks like if we write the parameters explicitly in matrix form. The matrices are 1×1 .

$$\begin{aligned}\begin{bmatrix} x \end{bmatrix}_t &= \begin{bmatrix} b \end{bmatrix} \begin{bmatrix} x \end{bmatrix}_{t-1} + \begin{bmatrix} 0 \end{bmatrix} + \begin{bmatrix} w \end{bmatrix}_t \\ \begin{bmatrix} y \end{bmatrix}_t &= \begin{bmatrix} x \end{bmatrix}_t + \begin{bmatrix} \beta \end{bmatrix} \begin{bmatrix} d \end{bmatrix}_t\end{aligned}\tag{7.10}$$

To create the model list for `MARSS()`, we specify the parameter matrices one-to-one like they look in Equation @ref(eq:short.lr.ar1.mat).

```
R <- matrix("r") # no v_t
D <- matrix("beta")
U <- matrix(0) # since arima.sim was used, no u
B <- matrix("b")
d <- matrix(dt, nrow = 1)
A <- matrix(0)
```

`MARSS()` requires \mathbf{d} be a matrix also. Each row is a covariate and each column is a time step. No missing values allowed as this is an input.

How should we treat the \mathbf{R} matrix? It is zero, and we could set \mathbf{R} to zero:

```
R <- matrix(0)
```

However, the EM algorithm in the **MARSS** package will not perform well at all with **R** set to zero and it has to do with how $\mathbf{R} = 0$ affects the update equations. You can use the BFGS algorithm or estimate **R**.

```
R <- matrix("r")
mod.list <- list(B = B, U = U, R = R, D = D, d = d, A = A)
fit <- MARSS(yt, model = mod.list)
```

Success! abstol and log-log tests passed at 75 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 75 iterations.
Log-likelihood: -31.93036
AIC: 73.86072 AICc: 74.49902

	Estimate
R.r	0.0116
B.b	0.9150
Q.Q	0.0911
x0.x0	0.0916
D.beta	1.0817

Initial states (x0) defined at t=0

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

Or use the BFGS algorithm for fitting:

```
R <- matrix(0)
mod.list <- list(B = B, U = U, R = R, D = D, d = d, A = A)
fit <- MARSS(yt, model = mod.list, method = "BFGS")
```

Success! Converged in 169 iterations.

Function MARSSkfas used for likelihood calculation.

MARSS fit is

Estimation method: BFGS

Estimation converged in 169 iterations.

Log-likelihood: -32.16264

AIC: 72.32527 AICc: 72.74632

	Estimate
B.b	0.8970
Q.Q	0.1114
x0.x0	0.0936
D.beta	1.0858

Initial states (x0) defined at t=0

Standard errors have not been calculated.

Use MARSSparamCIs to compute CIs and bias estimates.

This is the same model you are fitting when you pass in xreg with the arima() function:

```
stats::arima(yt, order = c(1, 0, 0), xreg = dt, include.mean = FALSE,
             method = "ML")
```

Call:

```
stats::arima(x = yt, order = c(1, 0, 0), xreg = dt, include.mean = FALSE, meth
```

Coefficients:

	ar1	dt
	0.8887	1.0859
s.e.	0.0428	0.0251

sigma^2 estimated as 0.1115: log likelihood = -32.97, aic = 71.93

Again the estimates are slightly different due to different treatment of the initial conditons.

7.6 LR with AR(1) errors and independent errors

We can add some independent error to our model:

$$\begin{aligned}x_t &= bx_{t-1} + w_t, \quad w_t \sim N(0, q) \\ y_t &= \beta d_t + x_t + v_t, \quad v_t \sim N(0, r)\end{aligned}\tag{7.11}$$

We'll generate this data by adding independent error to `yt` from the previous example.

```
yt.r <- yt + rnorm(TT, 0, sqrt(r))
```

We can fit as:

```
R <- matrix("r")
mod.list <- list(B = B, U = U, R = R, D = D, d = d, A = A)
fit <- MARSS(yt.r, model = mod.list)
```

Success! `abstol` and `log-log` tests passed at 55 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 55 iterations.
Log-likelihood: -161.9318
AIC: 333.8635   AICc: 334.5018
```

	Estimate
R.r	1.034
B.b	0.810
Q.Q	0.255
x0.x0	1.229
D.beta	0.973

Initial states (x0) defined at t=0

Standard errors have not been calculated.

Use MARSSparamCIs to compute CIs and bias estimates.

This is not a model that can be fit with arima().

7.7 LR with AR(1) errors driven by covariate

We can model a situation where the regression errors are autocorrelated but some of the variance is driven by a covariate. For example, good and bad ‘years’ are driven partially by, say, temperature, which we will model by `ct`. We will use an autocorrelated `ct` in the example, but it could be anything. How are autocorrelated errors different? There is memory in the errors. The `ct` in the past still affects the current error (w_t in this model).

$$\begin{aligned}x_t &= bx_{t-1} + \beta c_t + w_t, \quad w_t \sim N(0, q) \\y_t &= x_t + v_t, \quad v_t \sim N(0, r)\end{aligned}\tag{7.12}$$

Let’s create some simulated data with this structure:

```
beta <- 1.1
x0 <- 0
ct <- arima.sim(n = TT, model = list(ar = 0.8), sd = sqrt(1)) # our covariate
ct <- as.vector(ct)
xt <- rep(x0, TT)
for (i in 2:TT) xt[i] <- b * xt[i - 1] + beta * ct[i] + rnorm(1,
  0, sqrt(q))
yt <- xt + rnorm(TT, 0, sqrt(r))
```

To fit this with `MARSS()`, we match up the model to the full MARSS model form:

$$\begin{aligned}\mathbf{x}_t &= \mathbf{B}\mathbf{x}_{t-1} + \mathbf{C}c_t + \mathbf{w}_t, \quad \mathbf{w}_t \sim \text{MVN}(0, \mathbf{Q}_t) \\ \mathbf{y}_t &= \mathbf{x}_t + \mathbf{v}_t, \quad \mathbf{v}_t \sim \text{MVN}(0, \mathbf{R}_t)\end{aligned}\tag{7.13}$$

The model list for `MARSS()` is:


```

R <- matrix("r") # no v_t
C <- matrix("beta")
U <- matrix(0) # no u
B <- matrix("b")
c <- matrix(ct, nrow = 1)
A <- matrix(0)

```

Now fit:

```

mod.list <- list(B = B, U = U, R = R, C = C, c = c, A = A)
fit <- MARSS(yt, model = mod.list)

```

Success! abstol and log-log tests passed at 56 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 56 iterations.
 Log-likelihood: -142.9528
 AIC: 295.9055 AICc: 296.5438

	Estimate
R.r	0.764
B.b	0.893
Q.Q	0.107
x0.x0	0.043
C.beta	1.045

Initial states (x0) defined at t=0

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

7.8 Flat level model

For the next examples, we will use the Nile river flow from 1871 to 1970, a data set in the **datasets** package.

```
nile <- as.vector(datasets::Nile)
year <- as.vector(time(Nile))
```

The first model we will fit is a flat level model:

$$y_t = a + v_t, \quad v_t \sim N(0, r) \quad (7.14)$$

where y_t is the river flow volume at year t and a is some constant average flow level (notice it has no t subscript).

To fit this model with MARSS, we explicitly show all the MARSS parameters.

$$\begin{aligned} x_t &= 1 \times x_{t-1} + 0 + w_t, \quad w_t \sim N(0, 0) \\ y_t &= 0 \times x_t + a + v_t, \quad v_t \sim N(0, r) \\ x_0 &= 0 \end{aligned} \quad (7.15)$$

The model list and fit for this equation is

```
mod.list1 <- list(Z = matrix(0), A = matrix("a"), R = matrix("r"),
  B = matrix(1), U = matrix(0), Q = matrix(0), x0 = matrix(0))
fit1 <- MARSS(nile, model = mod.list1)
```

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: -654.5157

AIC: 1313.031 AICc: 1313.155

```

      Estimate
A.a      919
R.r     28352
Initial states (x0) defined at t=0

```

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

MARSS includes the state process x_t but we are setting \mathbf{Z} to zero so that x_t 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. In MARSS form, this model is:

$$\begin{aligned}
 x_t &= 1 \times x_{t-1} + 0 + w_t \text{ where } w_t \sim N(0, 0) \\
 y_t &= 1 \times x_t + 0 + v_t \text{ where } v_t \sim N(0, r) \\
 x_0 &= a
 \end{aligned}
 \tag{7.16}$$

The model list and fit for this equation is

```

mod.list2 <- list(Z = matrix(1), A = matrix(0), R = matrix("r"),
  B = matrix(1), U = matrix(0), Q = matrix(0), x0 = matrix("a"))
fit2 <- MARSS(nile, model = mod.list2)

```

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: -654.5157

AIC: 1313.031 AICc: 1313.155

```

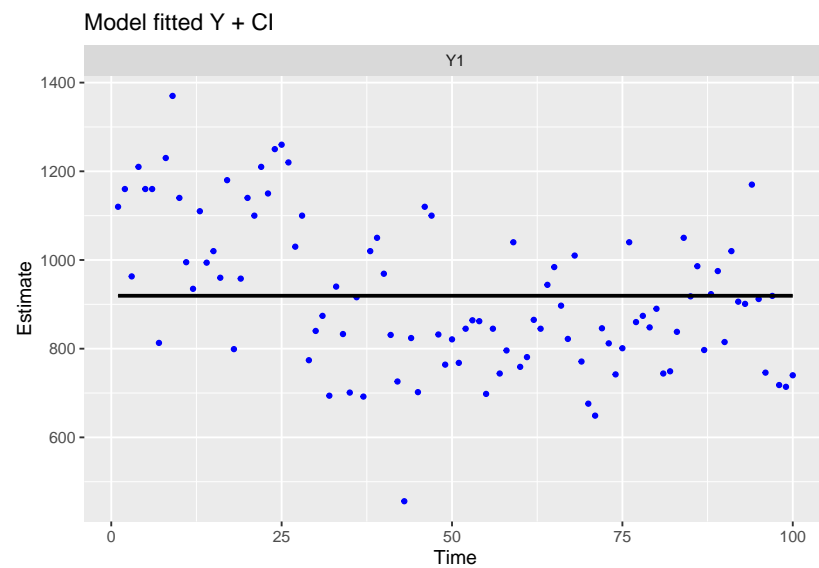
      Estimate
R.r      28352
x0.a      919
Initial states (x0) defined at t=0

```

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

The results are the same. We just formatted the model in different ways.
 We can plot the fitted model against the Nile river flow (blue dots) using `autoplot()`.

```
ggplot2::autoplot(fit2, plot.type = "model.ytT")
```



7.9 Linear trend model

Looking at the data, we might expect that a declining average river flow would be better. In MARSS form, that model would be:

$$\begin{aligned}x_t &= 1 \times x_{t-1} + 0 + w_t, \quad w_t \sim N(0, 0) \\y_t &= 0 \times x_t + a + \beta t + v_t, \quad v_t \sim N(0, r) \\x_0 &= 0\end{aligned}\tag{7.17}$$

where t is the year and u is the average per-year decline in river flow volume.

The model list and fit for this equation is

```
mod.list1 <- list(Z = matrix(0), A = matrix("a"), R = matrix("r"),
  D = matrix("beta"), d = matrix(1:100, nrow = 1), B = matrix(1),
  U = matrix(0), Q = matrix(0), x0 = matrix(0))
fit1 <- MARSS(nile, model = mod.list1)
```

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: -642.3147

AIC: 1290.629 AICc: 1290.879

	Estimate
A.a	1056.42
R.r	22212.64
D.beta	-2.71

Initial states (x0) defined at t=0

Standard errors have not been calculated.

Use MARSSparamCIs to compute CIs and bias estimates.

We can also write this model as follows by modeling the trend with x_t :

$$\begin{aligned}x_t &= 1 \times x_{t-1} + u + w_t, \quad w_t \sim N(0, 0) \\y_t &= 1 \times x_t + 0 + v_t, \quad v_t \sim N(0, r) \\x_0 &= a\end{aligned}\tag{7.18}$$

The model is specified as a list as follows. To fit, we need to force the algorithm to run a bit longer as it is showing convergence a bit early.

```
mod.list2 = list(Z = matrix(1), A = matrix(0), R = matrix("r"),
  B = matrix(1), U = matrix("u"), Q = matrix(0), x0 = matrix("a"))
fit2 <- MARSS(nile, model = mod.list2, control = list(minit = 30))
```

Success! algorithm run for 30 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 30 (=minit) iterations and convergence was reached.

Log-likelihood: -642.3147

AIC: 1290.629 AICc: 1290.879

Estimate

R.r 22212.64

U.u -2.71

x0.a 1056.37

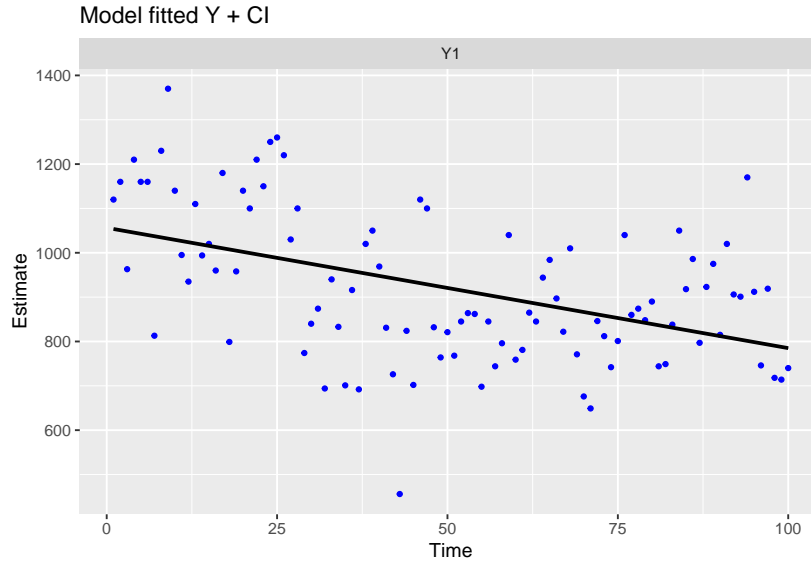
Initial states (x0) defined at t=0

Standard errors have not been calculated.

Use MARSSparamCIs to compute CIs and bias estimates.

The fits are the same with either formulation of the model as long as we force the algorithm to run longer for the second form.

```
ggplot2::autoplot(fit2, plot.type = "model.ytT")
```



7.10 Stochastic level model

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 at year t is a function of average river flow in year $t - 1$.

$$\begin{aligned}x_t &= x_{t-1} + w_t, \quad w_t \sim N(0, q) \\y_t &= x_t + v_t, \quad v_t \sim N(0, r) \\x_0 &= \pi\end{aligned}\tag{7.19}$$

With all the MARSS parameters shown, the model is:

$$\begin{aligned}x_t &= 1 \times x_{t-1} + 0 + w_t, \quad w_t \sim N(0, q) \\y_t &= 1 \times x_t + 0 + v_t, \quad v_t \sim N(0, r) \\x_0 &= \pi\end{aligned}\tag{7.20}$$

The model is specified as a list as follows. We can use the BFGS algorithm to ‘polish’ off the fit and get closer to the MLE. Why not just start with BFGS?

First, it happens to take a long long time to fit and more importantly, the BFGS algorithm is sensitive to starting conditions and can catastrophically fail. In this case, it is slow but works fine. For some models, it does work better (faster and stable), but using the EM algorithm to get decent starting conditions for the BFGS algorithm is a common fitting strategy.

```
mod.list = list(Z = matrix(1), A = matrix(0), R = matrix("r"),
               B = matrix(1), U = matrix(0), Q = matrix("q"), x0 = matrix("pi"))
fit1 <- MARSS(nile, model = mod.list, silent = TRUE)
fit2 <- MARSS(nile, model = mod.list, inits = fit1, method = "BFGS")
```

Success! Converged in 12 iterations.

Function MARSSkfas used for likelihood calculation.

MARSS fit is

Estimation method: BFGS

Estimation converged in 12 iterations.

Log-likelihood: -637.7451

AIC: 1281.49 AICc: 1281.74

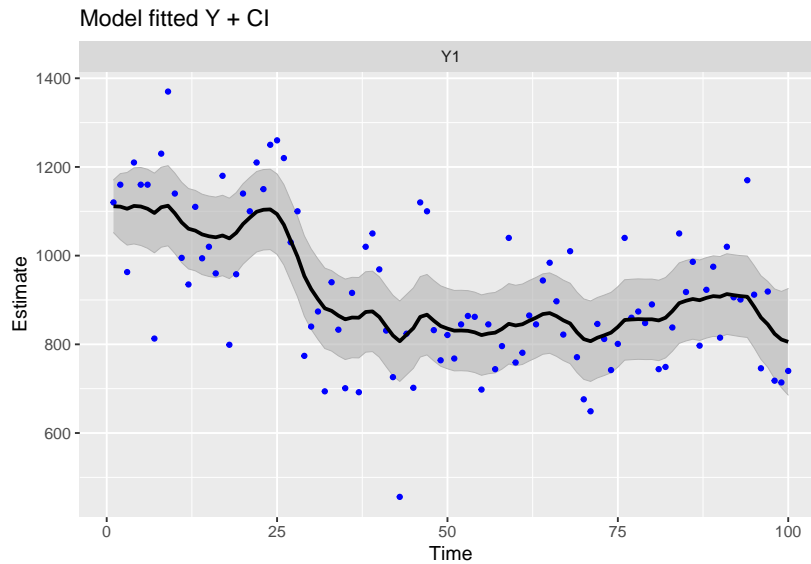
	Estimate
R.r	15337
Q.q	1218
x0.pi	1112

Initial states (x0) defined at t=0

Standard errors have not been calculated.

Use MARSSparamCIs to compute CIs and bias estimates.

```
ggplot2::autoplot(fit2, plot.type = "model.ytT")
```

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 \mathbf{R} and \mathbf{Q} are a little different than in Koopman et al. (1999).

We can fit the Koopman model with `stats::StructTS()`. The estimates are slightly different since the initial conditions are treated differently.

```
fit.ts <- stats::StructTS(nile, type = "level")
fit.ts
```

Call:

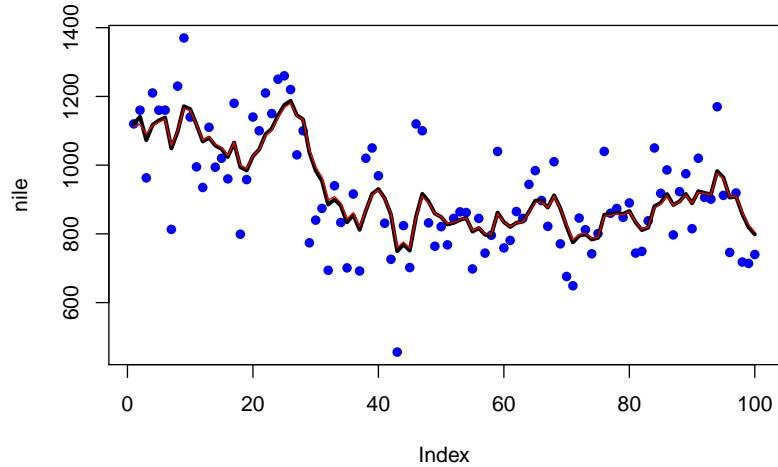
```
stats::StructTS(x = nile, type = "level")
```

Variances:

level	epsilon
1469	15099

The fitted values returned by `fitted()` applied to a `StructTS` object are different than that returned by `fitted()` applied to a `marssMLE` object. The former returns \hat{y} conditioned on the data up to time t , while the latter returns the \hat{y} conditioned on all the data. If you want to compare use:

```
plot(nile, type = "p", pch = 16, col = "blue")
lines(fitted(fit.ts), col = "black", lwd = 3)
lines(MARSSkfss(fit2)$xtt[1, ], col = "red", lwd = 1)
```



The black line is the `StructTS()` fit and the red line is the equivalent `MARSS()` fit.

7.11 Stochastic slope model

We can also model the β as a random walk:

$$\begin{aligned}\beta_t &= \beta_{t-1} + w_t, \quad w_t \sim N(0, q) \\ y_t &= a + \beta_t t + v_t, \quad v_t \sim N(0, r) \\ x_0 &= \pi\end{aligned}\tag{7.21}$$

The β_t is model with x_t . With all the MARSS parameters shown, the model is:

$$\begin{aligned}x_t &= 1 \times x_{t-1} + 0 + w_t, \quad w_t \sim N(0, q) \\ y_t &= t \times x_t + a + v_t, \quad v_t \sim N(0, r) \\ x_0 &= \pi\end{aligned}\tag{7.22}$$

The trick here is to recognize that \mathbf{Z}_t , the matrix in front of \mathbf{x}_t in the \mathbf{y}_t equation, can be time-varying and can be fixed. In a time-varying matrix in **MARSS**, the time element is in the 3rd dimension. We are going to fix $\mathbf{Z}[1,1,t] = t$, where t is `year-mean(year)`. \mathbf{Z} is a $1 \times 1 \times 100$ array. Demeaning the covariate stabilizes the fitting. Try without demeaning to see the difference.

The model is specified as a list as follows.

```
Z <- array(0, dim = c(1, 1, length(nile)))
Z[1, 1, ] <- year - mean(year)
mod.list = list(Z = Z, A = matrix("a"), R = matrix("r"), B = matrix(1),
  U = matrix(0), Q = matrix("q"), x0 = matrix("pi"))
fit1 <- MARSS(nile, model = mod.list, silent = TRUE)
fit2 <- MARSS(nile, model = mod.list, inits = fit1, method = "BFGS")
```

Success! Converged in 13 iterations.

Function MARSSkfas used for likelihood calculation.

MARSS fit is

Estimation method: BFGS

Estimation converged in 13 iterations.

Log-likelihood: -636.6226

AIC: 1281.245 AICc: 1281.666

Estimate

A.a 836.164

R.r 16835.484

Q.q 0.749

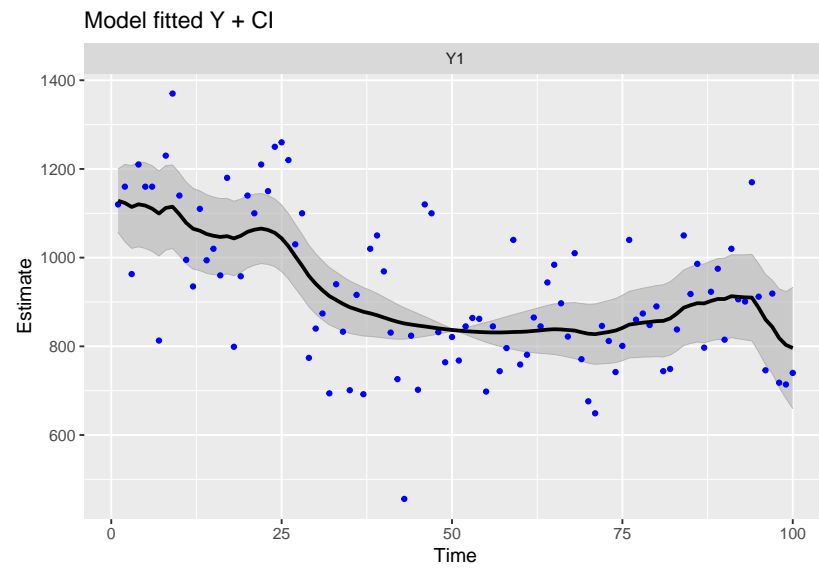
x0.pi -5.905

Initial states (x0) defined at t=0

Standard errors have not been calculated.

Use MARSSparamCIs to compute CIs and bias estimates.

```
ggplot2::autoplot(fit2, plot.type = "model.ytT")
```



Chapter 8

Multivariate Models

These models can be written in the form:

$$\begin{aligned} \mathbf{x}_t &= \mathbf{B}_t \mathbf{x}_{t-1} + \mathbf{u}_t + \mathbf{C}_t \mathbf{c}_t + \mathbf{G}_t \mathbf{w}_t, \quad \mathbf{w}_t \sim \text{MVN}(0, \mathbf{Q}_t) \\ \mathbf{y}_t &= \mathbf{Z}_t \mathbf{x}_t + \mathbf{a}_t + \mathbf{D}_t \mathbf{d}_t + \mathbf{H}_t \mathbf{v}_t, \quad \mathbf{v}_t \sim \text{MVN}(0, \mathbf{R}_t) \\ \mathbf{x}_1 &\sim \text{MVN}(\boldsymbol{\pi}, \boldsymbol{\Lambda}) \text{ or } \mathbf{x}_0 \sim \text{MVN}(\boldsymbol{\pi}, \boldsymbol{\Lambda}) \end{aligned} \quad (8.1)$$

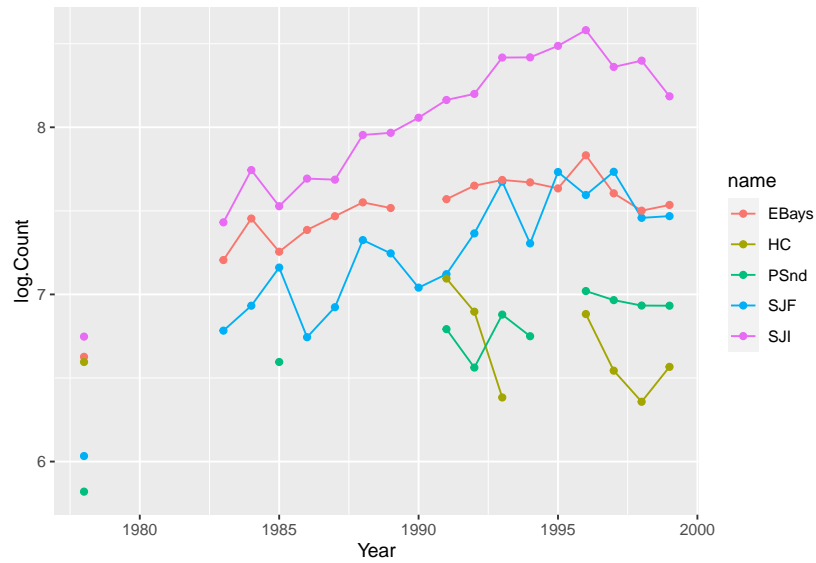
Set up a plotting function for the parameters:

```
plot.pars <- function(fit) {  
  df <- broom::tidy(fit)  
  df$parameter <- sapply(df$term, function(x) {  
    stringr::str_split(x, "[.]")[[1]][1]  
  })  
  df$parameter[df$parameter %in% c("R", "Q")] <- "R and Q"  
  ggplot2::ggplot(df, ggplot2::aes(x = term, y = estimate)) +  
    ggplot2::geom_point() + ggplot2::geom_errorbar(ggplot2::aes(ymin = conf.low,  
      ymax = conf.up), width = 0.2) + geom_hline(yintercept = 0,  
      col = "blue") + facet_wrap(~parameter, scales = "free")  
}
```

Harbor seal data

For these examples, we will use the harbor seal data set.

```
data(harborSealWA, package = "MARSS")
harbordf <- pivot_longer(as.data.frame(harborSealWA), cols = -Year,
  values_to = "log.Count")
ggplot2::ggplot(harbordf, aes(x = Year, y = log.Count, col = name)) +
  geom_line() + geom_point()
```



8.1 RW observed with multiple time series

A univariate random walk observed by 3 observation time series.

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

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}_t = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} x_t + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}, \quad \mathbf{v}_t \sim \text{MVN} \left(\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} r & 0 & 0 \\ 0 & r & 0 \\ 0 & 0 & r \end{bmatrix} \right) \quad (8.2)$$

Create set up the model:

```
Z <- matrix(1, 3, 1)
R <- "diagonal and equal"
```

Fit with MARSS():

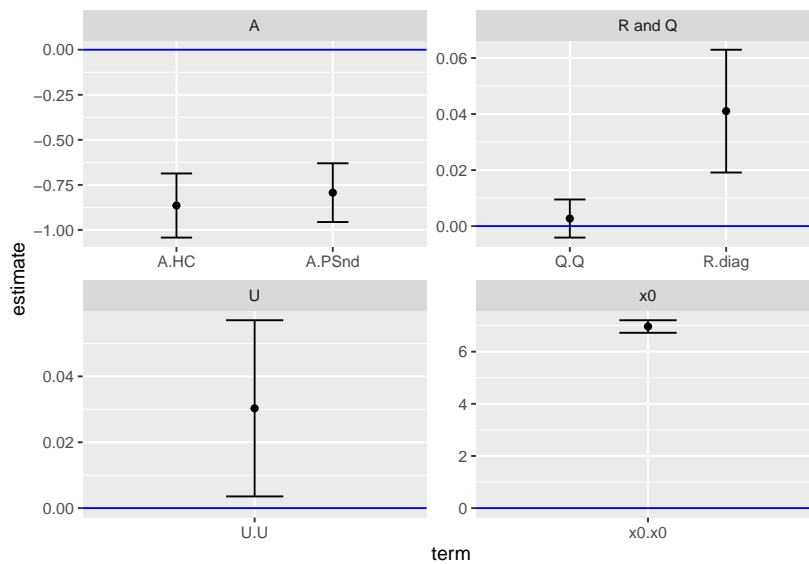
```
yt <- t(harborSealWA)[c("EBays", "PSnd", "HC"), ]
fit <- MARSS(yt, model = list(R = R, Z = Z))
```

Get confidence intervals with tidy():

```
broom::tidy(fit)
```

	term	estimate	std.error	conf.low	conf.up
1	A.PSnd	-0.793079609	0.083137017	-0.956025168	-0.630134050
2	A.HC	-0.864545855	0.090775537	-1.042462638	-0.686629073
3	R.diag	0.041048591	0.011183264	0.019129797	0.062967385
4	U.U	0.030295215	0.013645831	0.003549878	0.057040552
5	Q.Q	0.002704196	0.003471737	-0.004100284	0.009508676
6	x0.x0	6.964254445	0.122862306	6.723448750	7.205060139

```
plot.pars(fit)
```



8.2 Multiple time series each observing a RW

Each time series is observing its own random walk but the random walks have correlated process error.

$$\begin{aligned} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}_t &= \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}_{t-1} + \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} + \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}, \mathbf{w}_t \sim \text{MVN} \left(\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} q_1 & c_{12} & c_{13} \\ c_{12} & q_2 & c_{23} \\ c_{13} & c_{23} & q_3 \end{bmatrix} \right) \\ \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}_t &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}_t + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}, \mathbf{v}_t \sim \text{MVN} \left(\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} r & 0 & 0 \\ 0 & r & 0 \\ 0 & 0 & r \end{bmatrix} \right) \end{aligned} \quad (8.3)$$

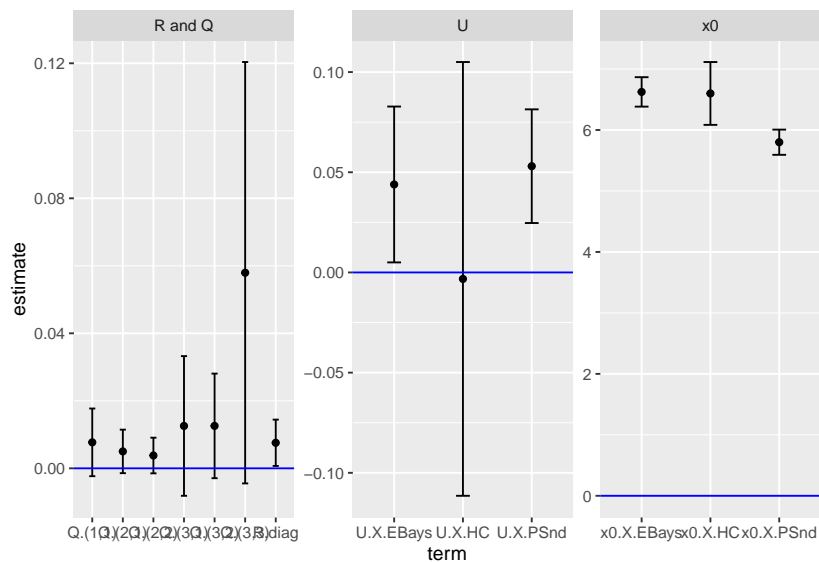
Create set up the model:

```
Q <- "unconstrained"
R <- "diagonal and equal"
```

Fit with MARSS():

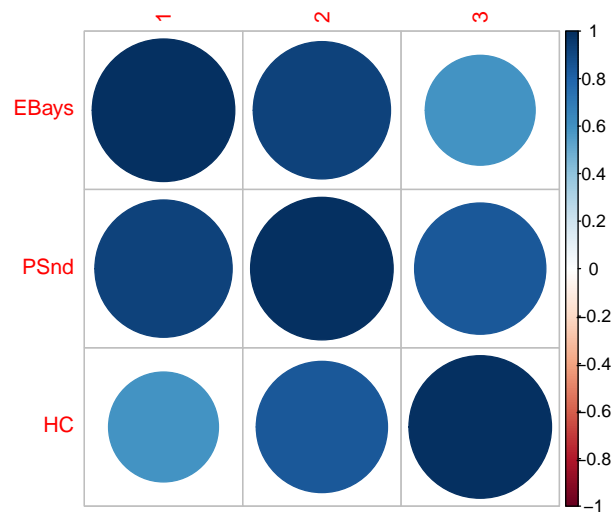
```
fit <- MARSS(yt, model = list(R = R, Q = Q))
```

```
plot.pars(fit)
```



Look at the \mathbf{Q} . Hood Canal (HC) appears to be behaving independently.

```
Q.est <- coef(fit, type = "matrix")$Q
rownames(Q.est) <- rownames(yt)
corrplot::corrplot(cov2cor(Q.est))
```



8.3 Different time series observing a RW

The three outer regions (EBays, SJI, and SJF) are observing one random walk and PSnd and HC (more interior) are each observing their own. The regions are correlated. The observations error variances are shared within

regions but unique across regions.

$$\begin{aligned} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}_t &= \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}_{t-1} + \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} + \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}, \mathbf{w}_t \sim \text{MVN} \left(\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} q_1 & c_{12} & c_{13} \\ c_{12} & q_2 & c_{23} \\ c_{13} & c_{23} & q_3 \end{bmatrix} \right) \\ \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix}_t &= \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}_t + \begin{bmatrix} 0 \\ a_2 \\ a_3 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{bmatrix}, \mathbf{v}_t \sim \text{MVN} \left(\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} r_1 & 0 & 0 & 0 & 0 \\ 0 & r_1 & 0 & 0 & 0 \\ 0 & 0 & r_1 & 0 & 0 \\ 0 & 0 & 0 & r_2 & 0 \\ 0 & 0 & 0 & 0 & r_3 \end{bmatrix} \right) \end{aligned} \quad (8.4)$$

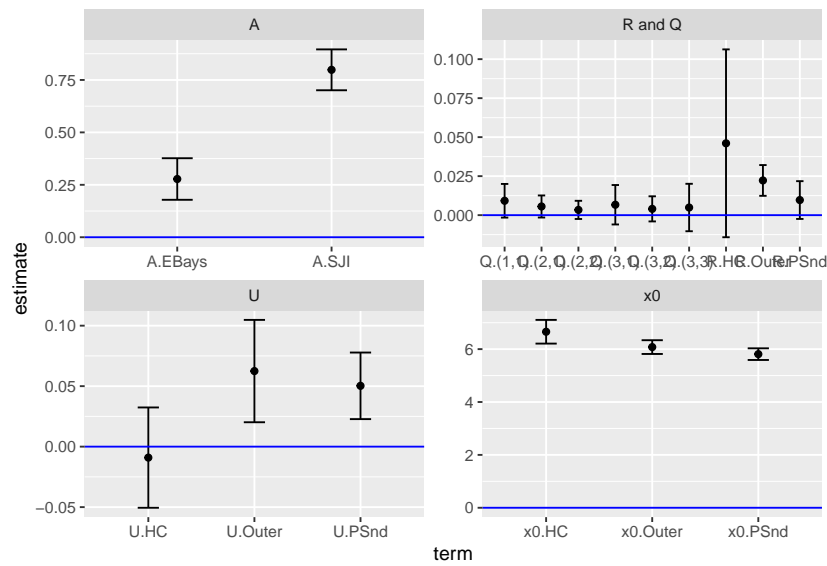
Create set up the model:

```
Q <- "unconstrained"
R <- matrix(list(0), 5, 5)
diag(R) <- c("Outer", "Outer", "Outer", "PSnd", "HC")
Z <- matrix(0, 5, 3)
colnames(Z) <- c("Outer", "PSnd", "HC")
Z[1:3, 1] <- 1
Z[4, 2] <- Z[5, 3] <- 1
```

Fit with MARSS():

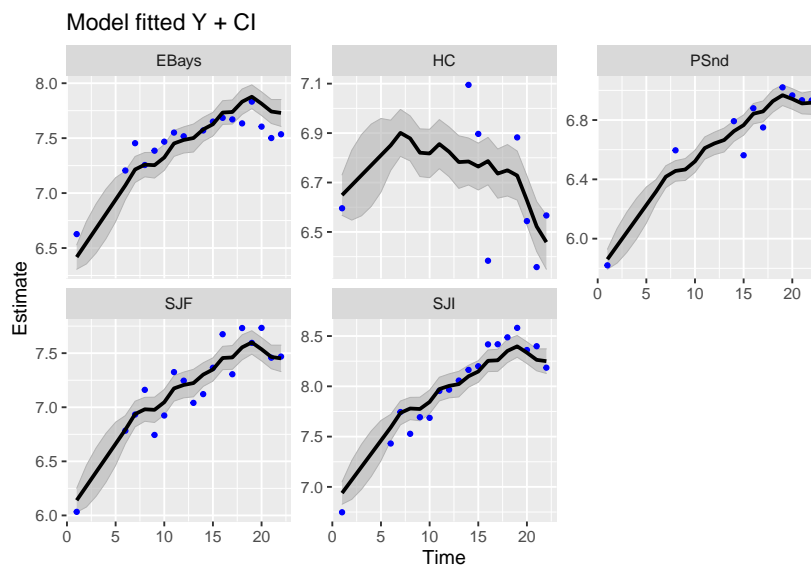
```
yt <- t(harborSealWA)[2:6, ]
fit <- MARSS(yt, model = list(R = R, Q = Q, Z = Z))
```

```
plot.pars(fit)
```



Model fits.

```
ggplot2::autoplot(fit, plot.type = "model.ytT")
```



8.4 Trend observed with AR(1) error

So far we have been assuming that the observations errors, \mathbf{v}_t , are temporally independent. Let's assume 3 observations of one underlying trend but assume the observations have AR(1) errors. We'll assume that the variance of those error is correlated across the first 2 time series and they have the same level of autocorrelation (b). We'll treat the 3rd time series as independent.

To do this, we need to move the observation errors into the \mathbf{x} part of the MARSS model and we need to add the \mathbf{B} matrix for the AR(1) feature. Notice that the \mathbf{y} equation does not have \mathbf{v}_t since the observation errors are specified in the \mathbf{x} part of the equation and are flowing from \mathbf{x} to \mathbf{y} via \mathbf{Z} .

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}_t = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & b_1 & 0 & 0 \\ 0 & 0 & b_1 & 0 \\ 0 & 0 & 0 & b_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}_{t-1} + \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \end{bmatrix}, \quad \mathbf{w}_t \sim \text{MVN} \left(\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} q_1 & 0 & 0 & 0 \\ 0 & q_2 & c & 0 \\ 0 & c & q_3 & 0 \\ 0 & 0 & 0 & q_4 \end{bmatrix} \right)$$

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}_t = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}_t + \begin{bmatrix} 0 \\ a_2 \\ a_3 \end{bmatrix}$$
(8.5)

Create set up the model.

```
Z <- matrix(0, 3, 4)
Z[, 1] <- 1
Z[, 2:4] <- diag(1, 3)
R <- U <- A <- "zero"
Q <- matrix(list(0), 4, 4)
diag(Q) <- paste0("q", 1:4)
Q[2, 3] <- Q[3, 2] <- "c"
```

Print out the Q to make sure it matches our equation.

```
Q
```

```
 [,1] [,2] [,3] [,4]
```

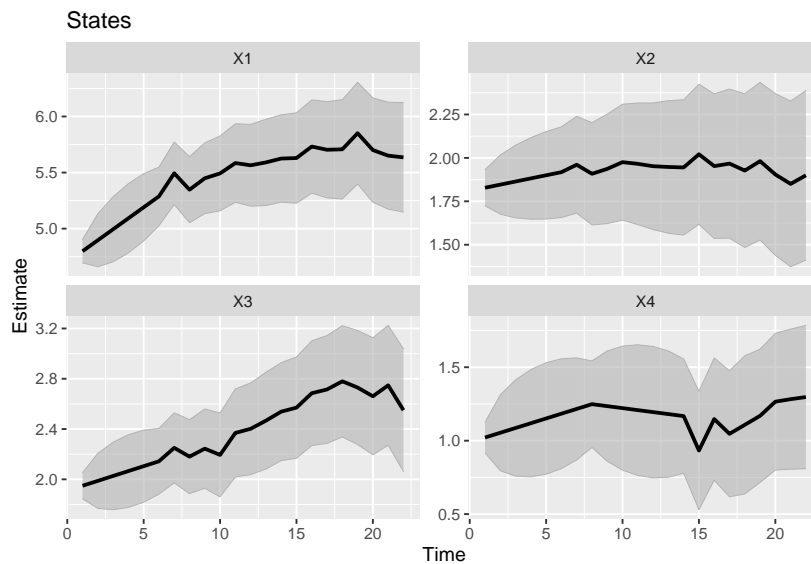
```
[1,] "q1" 0    0    0
[2,] 0    "q2" "c"  0
[3,] 0    "c"  "q3" 0
[4,] 0    0    0    "q4"
```

Fit with MARSS():

```
yt <- t(harborSealWA)[c("EBays", "SJI", "PSnd"), ]
fit <- MARSS(yt, model = list(R = R, Z = Z, Q = Q, U = U, A = A),
  inits = list(x0 = matrix(0, 4, 1)))
```

Estimated states. The first one is the trend and the next 3 are the AR(1) errors. If your goal was to estimate the underlying *signal*, then the plot for X1 is what you want and has the confidence intervals that you want.

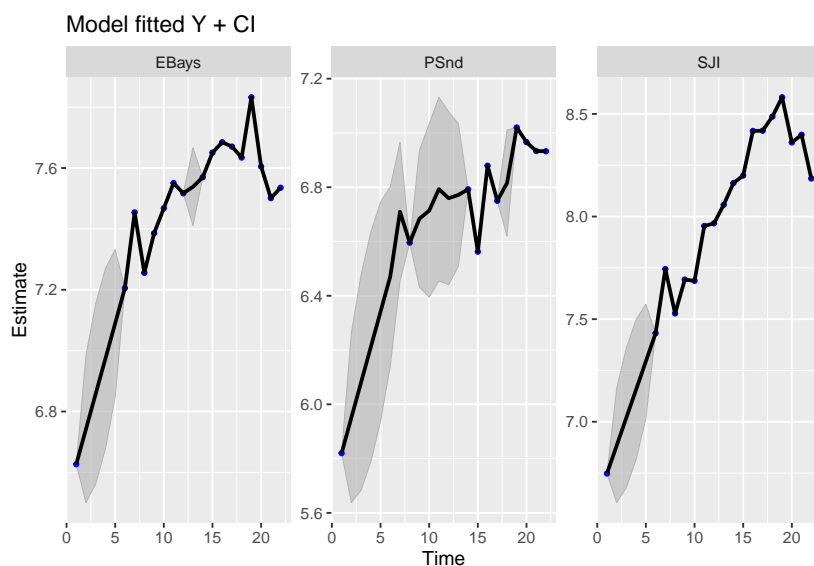
```
ggplot2::autoplot(fit, plot.type = "xtT")
```



If on the otherhand, you wanted estimates of the numbers in each region then sadly the model fit plots are not what you want. Since **R** was set to 0, the model fits will exactly fit the data. You would need to write a custom

function for the expected value of \mathbf{y} and its variance. The expected value is just the estimated first state (x_1) plus the corresponding a term. But the variance would need to be computed using the variances of states 2 to 4 (the AR(1) errors).

```
ggplot2::autoplot(fit, plot.type = "model.ytT")
```



8.5 Dynamic Factor Model with 3 trends

Dynamic Factor Modeling is akin to PCA (Principal Components Analysis) for time series data. This is a large topic. See the chapter on DFAs in the ATSA book. **MARSS** has a short-cut for fitting DFAs. See `?MARSS.dfa` but we will fit manually by specifying all the matrices.

We will fit a DFA with 2 trends.

$$\begin{aligned} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}_t &= \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}_{t-1} + \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}, \mathbf{w}_t \sim \text{MVN} \left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \right) \\ \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix}_t &= \begin{bmatrix} z_1 & 0 \\ z_2 & z_6 \\ z_3 & z_7 \\ z_4 & z_8 \\ z_5 & z_9 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}_t + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{bmatrix}, \mathbf{v}_t \sim \text{MVN} \left(\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} r & 0 & 0 & 0 & 0 \\ 0 & r & 0 & 0 & 0 \\ 0 & 0 & r & 0 & 0 \\ 0 & 0 & 0 & r & 0 \\ 0 & 0 & 0 & 0 & r \end{bmatrix} \right) \end{aligned} \quad (8.6)$$

Create set up the model:

```
Q <- diag(1, 2)
R <- "diagonal and equal"
Z <- matrix(list(0), 5, 2)
Z[, ] <- paste0("z", 1:10)
Z[1, 2] <- 0
A <- "zero"
U <- "zero"
```

Fit with MARSS(). We need to demean the data for a DFA. These models take a long time to converge. This one works better with the BFGS algorithm.

```
yt <- t(harborSealWA)[2:6, ]
fit <- MARSS(zscore(yt, mean.only = TRUE), model = list(R = R,
  Q = Q, Z = Z, A = A, U = U), method = "BFGS")
```

8.6 Linear constraints

Your model can have simple linear constraints within all the parameters except **Q**, **R** and **A**. For example $1 + 2a - 3b$ is a linear constraint. When entering this value for you matrix, you specify this as ‘‘1+2 \emph{a} -3 \emph{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 \mathbf{B} , \mathbf{Q} and \mathbf{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 \mathbf{A} if \mathbf{Z} is estimated.

```
yt <- t(harborSealWA)[2:6, ]
fit <- MARSS(yt, model = list(Z = Z, Q = Q, U = U, A = "zero"))
```

Success! abstol and log-log tests passed at 374 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 374 iterations.

Log-likelihood: -129.771

AIC: 275.5421 AICc: 277.8647

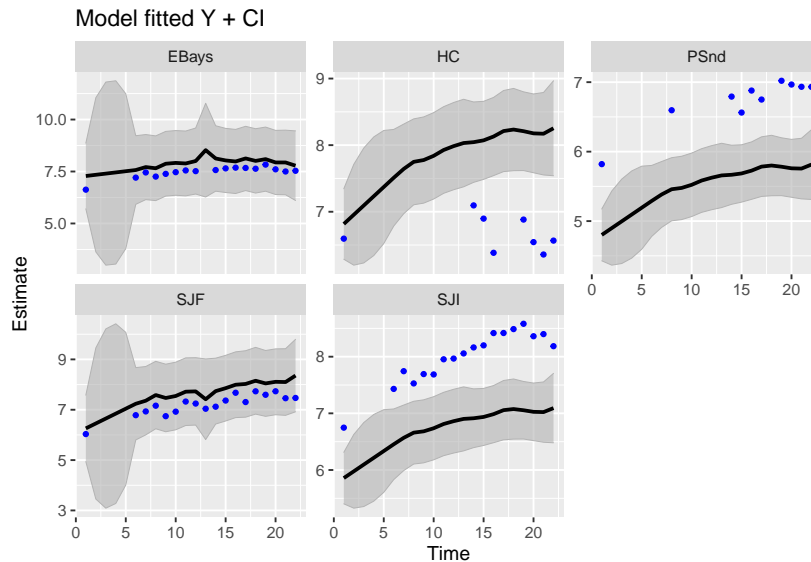
	Estimate
Z.z1	2.9071
Z.z2	11.5822
Z.z3	2.3829
R.diag	1.1864
U.u	-0.0333


```
Q.q11      0.0291
x0.X1      0.7627
x0.X2      1.9489
Initial states (x0) defined at t=0
```

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

This is a toy example and cannot fit the data.

```
ggplot2::autoplot(fit, plot.type = "model.ytT")
```



Part 3. Outputs

Part 3 discusses how to get outputs from `marssMLE` objects which are the class of object resulting from a call to `fit <- MARSS()`. Specifically:

- Estimated states
- Model fits
- Parameter estimates
- Kalman filter and smoother output
- Residuals
- Confidence intervals
- Predictions and forecasts
- Bootstrap resamples: parametric and innovations
- Simulated data

MARSS models are used in many many different ways and different users will want different types of output. **MARSS** functions will give you every type of output but the trick is to figure out what it is you want. Because MARSS models are used in so many different ways, what is *standard* output in one field or application might never be used in another field or application.

The notation in the **MARSS** package follows these conventions:

- `tT` expectation of something at t conditioned on all the data.
- `tt1` expectation of something at t conditioned on the data up to $t - 1$.
- `tt` expectation of something at t conditioned on the data up to t .
- *fitted* means the expected value of the right side of a MARSS \mathbf{x} or \mathbf{y} equation without the error term (\mathbf{w} or \mathbf{v}). Probably not what you want if you are trying to get *states* or \mathbf{x} estimates. Fitted values can be computed using any of the time conditionals listed above. Probably

what you want if you are trying to get expected values of \mathbf{y} or *model* estimates.

- *smoothed* and *filtered* means the expected values including the error terms (\mathbf{w} or \mathbf{v}). Probably what you want if you are trying to get *states* or \mathbf{x} estimates and not what you want if you are trying to get *model* or \mathbf{y} estimates.

Chapter 9

Parameter estimates

Chapter 10

States and smoothed estimates

Chapter 11

Model fits

Chapter 12

Residuals

Chapter 13

Confidence Intervals

Chapter 14

Predictions and forecasts

Part 4. Tips and Tricks

Part 4 discusses troubleshooting, error messages, and tips to get MARSS models to fit quicker and better.

Chapter 15

Troubleshooting

Tip: Use `MARSSinfo()` for information on various common error and warning messages.

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

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

This is typically caused by one of three problems: 1) you made a mistake in inputting your data, e.g. used -99 as the missing value in your data but did not replace these with NAs before passing to MARSS, 2) your data are not sufficient to estimate multiple variances or 3) your data are inconsistent with the model you are trying fit.

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

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

Chapter 16

Algorithm notes and cautions

The **MARSS** package fits models via maximum likelihood. The **MARSS** package is unusual among packages for fitting MARSS models in that fitting is performed via a constrained EM algorithm (Holmes, 2013) based on a vectorized form of Equation (1.1).

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. All the EM code in the **MARSS** package is currently in native R. Thus the model fitting with the EM algorithm is rather slow.

16.1 Properly constrained models

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

How do you know if the model is properly constrained? If you are using a MARSS model form that is widely used, then you can probably assume that it is properly constrained. If you go to papers where someone developed the

model or method, the issue of constraints necessary to ensure “identifiability” will likely be addressed if it is an issue. Are you fitting novel MARSS models? Then you will need to do some study on identifiability in this class of models using textbooks. 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.

16.2 Notes on the Kalman filter

The classic Kalman filter/smoothing algorithm, as shown in page 331-335 in Shumway and Stoffer (2006), 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, 2021). 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 offsetting 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!

16.3 Notes on the EM algorithm

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

16.4 Bias in variance estimates

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

16.5 Careful if specifying a prior on initial conditions

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

The default behavior for `MARSS()` is to set $\boldsymbol{\Lambda}$ to zero and estimate $\boldsymbol{\pi}$. This does not put any constraints on $\boldsymbol{\Lambda}$ (there is no $\boldsymbol{\Lambda}$ to put constraints on) and circumvents this problem. However if you plan to put constraints on $\boldsymbol{\pi}$ or $\boldsymbol{\Lambda}$, you should verse yourself in the most common problems. The common problems we have found with priors on \mathbf{x}_0 are the following. Problem 1) The correlation structure in $\boldsymbol{\Lambda}$ (whether the prior is diffuse or not) does not match the correlation structure in \mathbf{x}_0 implied by your model. For example, you specify a diagonal $\boldsymbol{\Lambda}$ (independent states), but the implied distribution has correlations. Problem 2) The correlation structure in $\boldsymbol{\Lambda}$ does not match the structure in \mathbf{x}_0 implied by constraints you placed on $\boldsymbol{\pi}$. For example, you specify that all values in $\boldsymbol{\pi}$ are shared, yet you specify that $\boldsymbol{\Lambda}$ is diagonal (independent).

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

In summary, if the implied correlation structure of your initial states is independent (diagonal variance-covariance matrix), you should generally be ok with a diagonal and high variance prior or with treating the initial states as

parameters (with $\mathbf{\Lambda} = 0$). But if your initial states have an implied correlation structure that is not independent, then proceed with caution. *With caution* means that you should assume you have problems and test how your model fits with simulated data.

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

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

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

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

Chapter 17

Other related packages

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

- `atsar` 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.
- 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 is an R package for fitting MARSS models. It is mainly Bayesian focused but it does allow MLE estimation via the `optim()` function. It has a book, *Dynamic Linear Models with R* by Petris et al., which has many examples of how to write MARSS models for different applications.
- `sspir` an R package for fitting ARSS (univariate) models with Gaussian, Poisson and binomial error distributions.
- `dse` (Dynamic Systems Estimation) is an R package for multivariate Gaussian state-space models with a focus on ARMA models.
- `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.
- The Brodgar software was developed by Alain Zuur to do (among many other things) dynamic factor analysis, which involves a special type of MARSS model. The methods and many example analyses are given in *Analyzing Ecological Data* by Zuur, Ieno and Smith. This is the one package that we are aware of that also uses an EM algorithm for parameter estimation.
- **eViews** is commercial economics software that will estimate at least some types of MARSS models.
- **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** is a S-plus module for fitting MAR models, which are called vector autoregressive (VAR) models in the economics and finance literature. It has some support for state-space VAR models, though we haven't used it so are not sure which parameters it allows you to estimate. It was developed by Andrew Bruce, Doug Martin, Jiahui Wang, and Eric Zivot, and it has a book associated with it: *Modeling*

Financial Time Series with S-plus by Eric Zivot and Jiahui Wang.

- kftrack is an R package provides a suite of functions specialized for fitting MARSS models to animal tracking data.
- crawl is an R package provides for fitting MARSS models to animal tracking data with covariates.

Bibliography

- Cavanaugh, J. and Shumway, R. (1997). A bootstrap variant of AIC for state-space model selection. *Statistica Sinica*, 7:473–496.
- de Jong, P. and Penzer, J. (1998). Diagnosing shocks in time series. *Journal of the American Statistical Association*, 93(442):796–806.
- Ghahramani, Z. and Hinton, G. E. (1996). Parameter estimation for linear dynamical systems. Technical Report CRG-TR-96-2, University of Totronto, Dept. of Computer Science.
- 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(4):377–389.
- Harvey, A. C., Koopman, S. J., and Penzer, J. (1998). Messy time series: a unified approach. *Advances in Econometrics*, 13:103–143.
- Helske, J. (2021). *KFAS: Kalman Filter and Smoother for Exponential Family State Space Models*. R package version 1.4.4.
- Hinrichsen, R. and Holmes, E. E. (2009). *Using multivariate state-space models to study spatial structure and dynamics*. CRC/Chapman Hall.
- Holmes, E. E. (2013). Derivation of an em algorithm for constrained and unconstrained multivariate autoregressive state-space (marss) models.
- Holmes, E. E. (2014). Computation of standardized residuals for MARSS models.

- 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(1):11–19.
- Kohn, R. and Ansley, C. F. (1989). A fast algorithm for signal extraction, influence and cross-validation in state-space models. *Biometrika*, 76(1):65–79.
- Koopman, S. J. (1993). Disturbance smoother for state space models. *Biometrika*, 80(1):117–126.
- 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.
- 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(7):551–563.
- Rauch, H. E. (1963). Solutions to the linear smoothing problem. *IEEE Transactions on Automatic Control*, 8:371–372.
- 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(4):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(4):923–929.
- 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(416):1024–1033.