Applied Time Series Analysis for Fisheries and Environmental Sciences

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

2021-07-28

Contents

Preface

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

Book package

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

```
library(devtools)
# Windows users will likely need to set this
# Sys.setenv('R_REMOTES_NO_ERRORS_FROM_WARNINGS' = 'true')
devtools::install_github("nwfsc-timeseries/atsalibrary")
```

Authors

Links to more code and publications can be found on our academic websites at the University of Washington:

- Elizabeth Eli Holmes http://faculty.washington.edu/eeholmes
- Mark D. Scheuerell http://faculty.washington.edu/scheuerl
- Eric J. Ward http://faculty.washington.edu/warde

Citation

Holmes, E. E., M. D. Scheuerell, and E. J. Ward. Applied time series analysis for fisheries and environmental data. Edition 2021. Contacts eeholmes@uw.edu, warde@uw.edu, and scheuerl@uw.edu

License

Chapter 1

Basic matrix math in R

This chapter reviews the basic matrix math operations that you will need to understand the course material and shows how to do these operations in R.

A script with all the R code in the chapter can be downloaded here.

After reviewing the material, you can check your knowledge via an online quiz (with solutions) or run the quiz from R using the atsalibrary package:

```
learnr::run_tutorial("matrix", package="atsalibrary")
```

1.1 Creating matrices in R

Create a 3×4 matrix, meaning 3 row and 4 columns, that is all 1s:

```r
matrix(1, 3, 4)
```

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

Create a 3×4 matrix filled in with the numbers 1 to 12 by column (default) and by row:

```r
matrix(1:12, 3, 4)
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

```r
matrix(1:12, 3, 4, byrow = TRUE)
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
```

Create a matrix with one column:

```r
matrix(1:4, ncol = 1)
```

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

Create a matrix with one row:

```r
matrix(1:4, nrow = 1)
```

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

Check the dimensions of a matrix

```r
A = matrix(1:6, 2, 3)
A
```

```
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

```r
dim(A)
```

```
[1] 2 3
```

Get the number of rows in a matrix:

```r
dim(A)[1]
```

```
[1] 2
```

```r
nrow(A)
```

```
[1] 2
```

Create a 3D matrix (called array):

```r
A = array(1:6, dim = c(2, 3, 2))
A
```

```
, , 1

     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

, , 2

     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

```
dim(A)
```

```
[1] 2 3 2
```

Check if an object is a matrix. A data frame is not a matrix. A vector is not a matrix.

```
A = matrix(1:4, 1, 4)
A
```

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

```
class(A)
```

```
[1] "matrix" "array"
```

```
B = data.frame(A)
B
```

```
  X1 X2 X3 X4
1  1  2  3  4
```

```
class(B)
```

```
[1] "data.frame"
```

```
C = 1:4
C
```

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

```
class(C)
```

```
[1] "integer"
```

1.2 Matrix multiplication, addition and transpose

You will need to be very solid in matrix multiplication for the course. If you haven't done it in awhile, google 'matrix multiplication youtube' and you find lots of 5min videos to remind you.

In R, you use the `%*%` operation to do matrix multiplication. When you do matrix multiplication, the columns of the matrix on the left must equal the rows of the matrix on the right. The result is a matrix that has the number of rows of the matrix on the left and number of columns of the matrix on the right.

$$(n \times m)(m \times p) = (n \times p)$$

```
A=matrix(1:6, 2, 3) #2 rows, 3 columns
B=matrix(1:6, 3, 2) #3 rows, 2 columns
A%*%B #this works
```

```
     [,1] [,2]
[1,]   22   49
[2,]   28   64
```

```
B%*%A #this works
```

```
     [,1] [,2] [,3]
[1,]    9   19   29
[2,]   12   26   40
[3,]   15   33   51
```

```
try(B%*%B) #this doesn't
```

```
Error in B %*% B : non-conformable arguments
```

To add two matrices use `+`. The matrices have to have the same dimensions.

```
A+A #works
```

```
     [,1] [,2] [,3]
[1,]    2    6   10
[2,]    4    8   12
```

```
A+t(B) #works
```

```
     [,1] [,2] [,3]
[1,]    2    5    8
[2,]    6    9   12
```

```
try(A+B) #does not work since A has 2 rows and B has 3
```

```
Error in A + B : non-conformable arrays
```

The transpose of a matrix is denoted $\mathbf{A}^\top$ or $\mathbf{A}'$. To transpose a matrix in R, you use `t()`.

```
A=matrix(1:6, 2, 3) #2 rows, 3 columns
t(A) #is the transpose of A
```

```
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
```

```
try(A%*%A) #this won't work
```

```
Error in A %*% A : non-conformable arguments
```

```
A%*%t(A) #this will
```

```
     [,1] [,2]
[1,]   35   44
[2,]   44   56
```

1.3 Subsetting a matrix

To subset a matrix, we use `[ ]`:

```
A=matrix(1:9, 3, 3) #3 rows, 3 columns
#get the first and second rows of A
#it's a 2x3 matrix
A[1:2,]
```

```
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
```

```
#get the top 2 rows and left 2 columns
A[1:2,1:2]
```

```
     [,1] [,2]
[1,]    1    4
[2,]    2    5
```

```
#What does this do?
A[c(1,3),c(1,3)]
```

```
     [,1] [,2]
[1,]    1    7
[2,]    3    9
```

```
#This?
A[c(1,2,1),c(2,3)]
```

```
     [,1] [,2]
[1,]    4    7
[2,]    5    8
[3,]    4    7
```

If you have used matlab, you know you can say something like `A[1,end]` to denote the element of a matrix in row 1 and the last column. R does not have 'end'. To do, the same in R you do something like:

```
A=matrix(1:9, 3, 3)
A[1,ncol(A)]
```

```
[1] 7
```

```
#or
A[1,dim(A)[2]]
```

```
[1] 7
```

Warning R will create vectors from subsetting matrices!

One of the really bad things that R does with matrices is create a vector if you happen to subset a matrix to create a matrix with 1 row or 1 column. Look at this:

```
A=matrix(1:9, 3, 3)
#take the first 2 rows
B=A[1:2,]
#everything is ok
dim(B)
```

```
[1] 2 3
```

```
class(B)
```

```
[1] "matrix" "array"
```

```
#take the first row
B=A[1,]
#oh no! It should be a 1x3 matrix but it is not.
dim(B)
```

```
NULL
```

```
#It is not even a matrix any more
class(B)
```

```
[1] "integer"
```

```
#and what happens if we take the transpose?
#Oh no, it's a 1x3 matrix not a 3x1 (transpose of 1x3)
t(B)
```

```
     [,1] [,2] [,3]
[1,]    1    4    7
```

```
#A%*%B should fail because A is (3x3) and B is (1x3)
A%*%B
```

```
     [,1]
[1,]   66
[2,]   78
[3,]   90
```

```
#It works? That is horrible!
```

This will create hard to find bugs in your code because you will look at `B=A[1,]` and everything looks fine. Why is R saying it is not a matrix! To stop R from doing this use `drop=FALSE`.

```
B=A[1,,drop=FALSE]
#Now it is a matrix as it should be
dim(B)
```

```
[1] 1 3
```

```
class(B)
```

```
[1] "matrix" "array"
```

```
#this fails as it should (alerting you to a problem!)
try(A%*%B)
```

```
Error in A %*% B : non-conformable arguments
```

1.4 Replacing elements in a matrix

Replace 1 element.

```
A=matrix(1, 3, 3)
A[1,1]=2
A
```

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

Replace a row with all 1s or a string of values

```
A=matrix(1, 3, 3)
A[1,]=2
A
```

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

```
A[1,]=1:3
A
```

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

Replace group of elements. This often does not work as one expects so be sure look at your matrix after trying something like this. Here I want to replace elements (1,3) and (3,1) with 2, but it didn't work as I wanted.

```
A=matrix(1, 3, 3)
A[c(1,3),c(3,1)]=2
A
```

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

How do I replace elements (1,1) and (3,3) with 2 then? It's tedious. If you have a lot of elements to replace, you might want to use a for loop.

```
A=matrix(1, 3, 3)
A[1,3]=2
A[3,1]=2
A
```

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

1.5 Diagonal matrices and identity matrices

A diagonal matrix is one that is square, meaning number of rows equals number of columns, and it has 0s on the off-diagonal and non-zeros on the diagonal. In R, you form a diagonal matrix with the `diag()` function:

```
diag(1,3) #put 1 on diagonal of 3x3 matrix
```

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

```
diag(2, 3) #put 2 on diagonal of 3x3 matrix
```

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

```
diag(1:4) #put 1 to 4 on diagonal of 4x4 matrix
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    0    2    0    0
[3,]    0    0    3    0
[4,]    0    0    0    4
```

The `diag()` function can also be used to replace elements on the diagonal of a matrix:

```
A = matrix(3, 3, 3)
diag(A) = 1
A
```

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

```
A = matrix(3, 3, 3)
diag(A) = 1:3
A
```

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

```
A = matrix(3, 3, 4)
diag(A[1:3, 2:4]) = 1
A
```

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

The `diag()` function is also used to get the diagonal of a matrix.

```
A = matrix(1:9, 3, 3)
diag(A)
```

```
[1] 1 5 9
```

The identity matrix is a special kind of diagonal matrix with 1s on the diagonal. It is denoted $\mathbf{I}$. $\mathbf{I}_3$ would mean a 3×3 diagonal matrix. A identity matrix has the property that $\mathbf{AI} = \mathbf{A}$ and $\mathbf{IA} = \mathbf{A}$ so it is like a 1.

```
A = matrix(1:9, 3, 3)
I = diag(3)  #shortcut for 3x3 identity matrix
A %*% I
```

```
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

1.6 Taking the inverse of a square matrix

The inverse of a matrix is denoted $\mathbf{A}^{-1}$. You can think of the inverse of a matrix like $1/a$. $1/a \times a = 1$. $\mathbf{A}^{-1}\mathbf{A} = \mathbf{A}\mathbf{A}^{-1} = \mathbf{I}$. The inverse of a matrix does not always exist; for one it has to be square. We'll be using inverses for variance-covariance matrices and by definition (of a variance-covariance matrix), the inverse of those exist. In R, there are a couple way common ways to take the inverse of a variance-covariance matrix (or something with the same properties). `solve()` is the most common probably:

```
A = diag(3, 3) + matrix(1, 3, 3)
invA = solve(A)
invA %*% A
```

```
             [,1]          [,2] [,3]
[1,] 1.000000e+00 -6.938894e-18    0
[2,] 2.081668e-17  1.000000e+00    0
[3,] 0.000000e+00  0.000000e+00    1
```

```
A %*% invA
```

```
             [,1]          [,2] [,3]
[1,] 1.000000e+00 -6.938894e-18    0
[2,] 2.081668e-17  1.000000e+00    0
[3,] 0.000000e+00  0.000000e+00    1
```

Another option is to use `chol2inv()` which uses a Cholesky decomposition[1]:

```
A = diag(3, 3) + matrix(1, 3, 3)
invA = chol2inv(chol(A))
invA %*% A
```

```
              [,1]         [,2]          [,3]
[1,]  1.000000e+00 6.938894e-17  0.000000e+00
[2,]  2.081668e-17 1.000000e+00 -2.775558e-17
[3,] -5.551115e-17 0.000000e+00  1.000000e+00
```

```
A %*% invA
```

```
             [,1]          [,2]          [,3]
[1,] 1.000000e+00  2.081668e-17 -5.551115e-17
[2,] 6.938894e-17  1.000000e+00  0.000000e+00
[3,] 0.000000e+00 -2.775558e-17  1.000000e+00
```

For the purpose of this course, `solve()` is fine.

[1]The Cholesky decomposition is a handy way to keep your variance-covariance matrices valid when doing a parameter search. Don't search over the raw variance-covariance matrix. Search over a matrix where the lower triangle is 0, that is what a Cholesky decomposition looks like. Let's call it `B`. Your variance-covariance matrix is `t(B)%*%B`.

1.7 Problems

1. Build a 4×3 matrix with the numbers 1 through 3 in each column. Try the same with the numbers 1 through 4 in each row.
2. Extract the elements in the 1st and 2nd rows and 1st and 2nd columns (you'll have a 2×2 matrix). Show the R code that will do this.
3. Build a 4×3 matrix with the numbers 1 through 12 by row (meaning the first row will have the numbers 1 through 3 in it).
4. Extract the 3rd row of the above. Show R code to do this where you end up with a vector and how to do this where you end up with a 1×3 matrix.
5. Build a 4×3 matrix that is all 1s except a 2 in the (2,3) element (2nd row, 3rd column).
6. Take the transpose of the above.
7. Build a 4×4 diagonal matrix with 1 through 4 on the diagonal.
8. Build a 5×5 identity matrix.
9. Replace the diagonal in the above matrix with 2 (the number 2).
10. Build a matrix with 2 on the diagonal and 1s on the offdiagonals.
11. Take the inverse of the above.
12. Build a 3×3 matrix with the first 9 letters of the alphabet. First column should be "a", "b", "c". `letters[1:9]` gives you these letters.
13. Replace the diagonal of this matrix with the word "cat".
14. Build a 4×3 matrix with all 1s. Multiply by a 3×4 matrix with all 2s.
15. If $\mathbf{A}$ is a 4×3 matrix, is $\mathbf{AA}$ possible? Is $\mathbf{AA}^\top$ possible? Show how to write $\mathbf{AA}^\top$ in R.

16. In the equation, $\mathbf{AB} = \mathbf{C}$, let $\mathbf{A} = \left[\begin{smallmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{smallmatrix}\right]$. Build a 3×3 $\mathbf{B}$ matrix with only 1s and 0s such that the values on the diagonal of $\mathbf{C}$ are 1, 8, 6 (in that order). Show your R code for $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{AB}$.

17. Same $\mathbf{A}$ matrix as above and same equation $\mathbf{AB} = \mathbf{C}$. Build a 3×3 $\mathbf{B}$ matrix such that $\mathbf{C} = 2\mathbf{A}$. So $\mathbf{C} = \left[\begin{smallmatrix} 2 & 8 & 14 \\ 4 & 10 & 16 \\ 6 & 12 & 18 \end{smallmatrix}\right]$. Hint, $\mathbf{B}$ is diagonal.

18. Same $\mathbf{A}$ and $\mathbf{AB} = \mathbf{C}$ equation. Build a $\mathbf{B}$ matrix to compute the row sums of $\mathbf{A}$. So the first 'row sum' would be $1 + 4 + 7$, the sum of all elements in row 1 of $\mathbf{A}$. $\mathbf{C}$ will be $\left[\begin{smallmatrix} 12 \\ 15 \\ 18 \end{smallmatrix}\right]$, the row sums of $\mathbf{A}$. Hint, $\mathbf{B}$ is a column matrix (1 column).

19. Same $\mathbf{A}$ matrix as above but now equation $\mathbf{BA} = \mathbf{C}$. Build a $\mathbf{B}$ matrix to compute the column sums of $\mathbf{A}$. So the first 'column sum' would be $1 + 2 + 3$. $\mathbf{C}$ will be a 1×3 matrix.

20. Let $\mathbf{AB} = \mathbf{C}$ equation but $\mathbf{A} = \left[\begin{smallmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{smallmatrix}\right]$ (so A=`diag(3)+1`). Build a $\mathbf{B}$ matrix such that $\mathbf{C} = \left[\begin{smallmatrix} 3 \\ 3 \\ 3 \end{smallmatrix}\right]$. Hint, you need to use the inverse of $\mathbf{A}$.

Chapter 2

Linear regression in matrix form

This chapter shows how to write linear regression models in matrix form. The purpose is to get you comfortable writing multivariate linear models in different matrix forms before we start working with time series versions of these models. Each matrix form is an equivalent model for the data, but written in different forms. You do not need to worry which form is better or worse at this point. Simply get comfortable writing multivariate linear models in different matrix forms.

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

Data and packages

This chapter uses the **stats**, **MARSS** and **datasets** packages. Install those packages, if needed, and load:

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

We will work with the `stackloss` dataset available in the **datasets** package. The dataset consists of 21 observations on the efficiency of a plant that produces nitric acid as a function of three explanatory variables: air flow, water temperature and acid concentration. We are going to use just the first 4 datapoints so that it is easier to write the matrices, but the concepts extend to as many datapoints as you have.

```
data(stackloss, package = "datasets")
dat = stackloss[1:4, ]  #subsetted first 4 rows
dat
```

```
  Air.Flow Water.Temp Acid.Conc. stack.loss
1       80         27         89         42
2       80         27         88         37
3       75         25         90         37
4       62         24         87         28
```

2.1 A simple regression: one explanatory variable

We will start by regressing stack loss against air flow. In R using the `lm()` function this is

```
# the dat data.frame is defined on the first page of the
# chapter
lm(stack.loss ~ Air.Flow, data = dat)
```

This fits the following model for the i-th measurment:

$$stack.loss_i = \alpha + \beta air_i + e_i, \text{ where } e_i \sim \text{N}(0, \sigma^2) \tag{2.1}$$

We will write the model for all the measurements together in two different ways, Form 1 and Form 2.

2.2 Matrix Form 1

In this form, we have the explanatory variables in a matrix on the left of our parameter matrix:

$$\begin{bmatrix} stack.loss_1 \\ stack.loss_2 \\ stack.loss_3 \\ stack.loss_4 \end{bmatrix} = \begin{bmatrix} 1 & air_1 \\ 1 & air_2 \\ 1 & air_3 \\ 1 & air_4 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} + \begin{bmatrix} e_1 \\ e_2 \\ e_3 \\ e_4 \end{bmatrix} \tag{2.2}$$

You should work through the matrix algebra to make sure you understand why Equation (2.2) is Equation (2.1) for all the i data points together.

We can write the first line of Equation (2.2) succinctly as

$$\mathbf{y} = \mathbf{Z}\mathbf{x} + \mathbf{e} \tag{2.3}$$

where $\mathbf{x}$ are our parameters, $\mathbf{y}$ are our response variables, and $\mathbf{Z}$ are our explanatory variables (with a 1 column for the intercept). The `lm()` function uses Form 1, and we can recover the $\mathbf{Z}$ matrix for Form 1 by using the `model.matrix()` function on the output from a `lm()` call:

```
fit = lm(stack.loss ~ Air.Flow, data = dat)
Z = model.matrix(fit)
Z[1:4, ]
```

```
  (Intercept) Air.Flow
1           1       80
2           1       80
3           1       75
4           1       62
```

2.2.1 Solving for the parameters

Note: You will not need to know how to solve linear matrix equations for this course. This section just shows you what the `lm()` function is doing to estimate the parameters.

Notice that $\mathbf{Z}$ is not a square matrix and its inverse does not exist but the inverse of $\mathbf{Z}^\top\mathbf{Z}$ exists—if this is a solveable problem. We can go through the following steps to solve for $\mathbf{x}$, our parameters α and β.

Start with $\mathbf{y} = \mathbf{Z}\mathbf{x} + \mathbf{e}$ and multiply by $\mathbf{Z}^\top$ on the left to get

$$\mathbf{Z}^\top\mathbf{y} = \mathbf{Z}^\top\mathbf{Z}\mathbf{x} + \mathbf{Z}^\top\mathbf{e}$$

Multiply that by $(\mathbf{Z}^\top\mathbf{Z})^{-1}$ on the left to get

$$(\mathbf{Z}^\top\mathbf{Z})^{-1}\mathbf{Z}^\top\mathbf{y} = (\mathbf{Z}^\top\mathbf{Z})^{-1}\mathbf{Z}^\top\mathbf{Z}\mathbf{x} + (\mathbf{Z}^\top\mathbf{Z})^{-1}\mathbf{Z}^\top\mathbf{e}$$

$(\mathbf{Z}^\top\mathbf{Z})^{-1}\mathbf{Z}^\top\mathbf{Z}$ equals the identity matrix, thus

$$(\mathbf{Z}^\top\mathbf{Z})^{-1}\mathbf{Z}^\top\mathbf{y} = \mathbf{x} + (\mathbf{Z}^\top\mathbf{Z})^{-1}\mathbf{Z}^\top\mathbf{e}$$

Move $\mathbf{x}$ to the right by itself, to get

$$(\mathbf{Z}^\top\mathbf{Z})^{-1}\mathbf{Z}^\top\mathbf{y} - (\mathbf{Z}^\top\mathbf{Z})^{-1}\mathbf{Z}^\top\mathbf{e} = \mathbf{x}$$

Let's assume our errors, the $\mathbf{e}$, are i.i.d. which means that

$$\mathbf{e} \sim \text{MVN}\left(0, \begin{bmatrix} \sigma^2 & 0 & 0 & 0 \\ 0 & \sigma^2 & 0 & 0 \\ 0 & 0 & \sigma^2 & 0 \\ 0 & 0 & 0 & \sigma^2 \end{bmatrix}\right)$$

This equation means $\mathbf{e}$ is drawn from a multivariate normal distribution with a variance-covariance matrix that is diagonal with equal variances. Under that assumption, the expected value of $(\mathbf{Z}^\top\mathbf{Z})^{-1}\mathbf{Z}^\top\mathbf{e}$ is zero. So we can solve for $\mathbf{x}$ as

$$\mathbf{x} = (\mathbf{Z}^\top\mathbf{Z})^{-1}\mathbf{Z}^\top\mathbf{y}$$

Let's try that with R and compare to what you get with `lm()`:

```r
y = matrix(dat$stack.loss, ncol = 1)
Z = cbind(1, dat$Air.Flow)  #or use model.matrix() to get Z
solve(t(Z) %*% Z) %*% t(Z) %*% y
```

```
           [,1]
[1,] -11.6159170
[2,]   0.6412918
```

```
coef(lm(stack.loss ~ Air.Flow, data = dat))
```

```
(Intercept)    Air.Flow
-11.6159170   0.6412918
```

As you see, you get the same values.

2.2.2 Form 1 with multiple explanatory variables

We can easily extend Form 1 to multiple explanatory variables. Let's say we wanted to fit this model:

$$stack.loss_i = \alpha + \beta_1 air_i + \beta_2 water_i + \beta_3 acid_i + e_i \tag{2.4}$$

With `lm()`, we can fit this with

```
fit1.mult = lm(stack.loss ~ Air.Flow + Water.Temp + Acid.Conc.,
    data = dat)
```

Written in matrix form (Form 1), this is

$$\begin{bmatrix} stack.loss_1 \\ stack.loss_2 \\ stack.loss_3 \\ stack.loss_4 \end{bmatrix} = \begin{bmatrix} 1 & air_1 & water_1 & acid_1 \\ 1 & air_2 & water_2 & acid_2 \\ 1 & air_3 & water_3 & acid_3 \\ 1 & air_4 & water_4 & acid_4 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix} + \begin{bmatrix} e_1 \\ e_2 \\ e_3 \\ e_4 \end{bmatrix} \tag{2.5}$$

Now **Z** is a matrix with 4 columns and **x** is a column vector with 4 rows. We can show the **Z** matrix again directly from our `lm()` fit:

```
Z = model.matrix(fit1.mult)
Z
```

```
  (Intercept) Air.Flow Water.Temp Acid.Conc.
1           1       80         27         89
2           1       80         27         88
3           1       75         25         90
4           1       62         24         87
attr(,"assign")
[1] 0 1 2 3
```

We can solve for $\mathbf{x}$ just like before and compare to what we get with `lm()`:

```
y = matrix(dat$stack.loss, ncol = 1)
Z = cbind(1, dat$Air.Flow, dat$Water.Temp, dat$Acid.Conc)
# or Z=model.matrix(fit2)
solve(t(Z) %*% Z) %*% t(Z) %*% y
```

```
            [,1]
[1,] -524.904762
[2,]   -1.047619
[3,]    7.619048
[4,]    5.000000
```

```
coef(fit1.mult)
```

```
(Intercept)    Air.Flow  Water.Temp  Acid.Conc.
-524.904762   -1.047619    7.619048    5.000000
```

Take a look at the $\mathbf{Z}$ we made in R. It looks exactly like what is in our model written in matrix form (Equation (2.5)).

2.2.3 When does Form 1 arise?

This form of writing a regression model will come up when you work with dynamic linear models (DLMs). With DLMs, you will be fitting models of the form $\mathbf{y}_t = \mathbf{Z}_t\mathbf{x}_t + \mathbf{e}_t$. In these models you have multiple $\mathbf{y}$ at regular time points and you allow your regression parameters, the $\mathbf{x}$, to evolve through time as a random walk.

2.2.4 Matrix Form 1b: The transpose of Form 1

We could also write Form 1 as follows:

$$\begin{bmatrix} stack.loss_1 & stack.loss_2 & stack.loss_3 & stack.loss_4 \end{bmatrix} = \\ \begin{bmatrix} \alpha & \beta_1 & \beta_2 & \beta_3 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 \\ air_1 & air_2 & air_3 & air_4 \\ wind_1 & wind_2 & wind_3 & wind_4 \\ acid_1 & acid_2 & acid_3 & acid_4 \end{bmatrix} + \begin{bmatrix} e_1 & e_2 & e_3 & e_4 \end{bmatrix} \tag{2.6}$$

This is just the transpose of Form 1. Work through the matrix algebra to make sure you understand why Equation (2.6) is Equation (2.1) for all the i data points together and why it is equal to the transpose of Equation (2.2). You'll need the relationship $(\mathbf{AB})^\top = \mathbf{B}^\top\mathbf{A}^\top$.

Let's write Equation (2.6) as $\mathbf{y} = \mathbf{Dd}$, where $\mathbf{D}$ contains our parameters. Then we can solve for $\mathbf{D}$ following the steps in Section 2.2.1 but multiplying from the right instead of from the left. Work through the steps to show that $\mathbf{d} = \mathbf{y}\mathbf{d}^\top(\mathbf{d}\mathbf{d}^\top)^{-1}$.

```
y = matrix(dat$stack.loss, nrow = 1)
d = rbind(1, dat$Air.Flow, dat$Water.Temp, dat$Acid.Conc)
y %*% t(d) %*% solve(d %*% t(d))
```

```
          [,1]      [,2]     [,3] [,4]
[1,] -524.9048 -1.047619 7.619048    5
```

```
coef(fit1.mult)
```

```
(Intercept)    Air.Flow  Water.Temp  Acid.Conc.
-524.904762   -1.047619    7.619048    5.000000
```

2.3 Matrix Form 2

In this form, we have the explanatory variables in a matrix on the right of our parameter matrix as in Form 1b but we arrange everything a little differently:

$$\begin{bmatrix} stack.loss_1 \\ stack.loss_2 \\ stack.loss_3 \\ stack.loss_4 \end{bmatrix} = \begin{bmatrix} \beta & 0 & 0 & 0 \\ 0 & \beta & 0 & 0 \\ 0 & 0 & \beta & 0 \\ 0 & 0 & 0 & \beta \end{bmatrix} \begin{bmatrix} air_1 \\ air_2 \\ air_3 \\ air_4 \end{bmatrix} + \begin{bmatrix} \alpha \\ \alpha \\ \alpha \\ \alpha \end{bmatrix} + \begin{bmatrix} e_1 \\ e_2 \\ e_3 \\ e_4 \end{bmatrix} \tag{2.7}$$

Work through the matrix algebra to make sure you understand why Equation (2.7) is the same as Equation (2.1) for all the i data points together.

We will write Form 2 succinctly as

$$\mathbf{y} = \mathbf{Zx} + \mathbf{a} + \mathbf{e} \tag{2.8}$$

2.3.1 Form 2 with multiple explanatory variables

The **x** is a column vector of the explanatory variables. If we have more explanatory variables, we add them to the column vector at the bottom. So if we had air flow, water temperature and acid concentration as explanatory variables, **x** looks like

$$\begin{bmatrix} air_1 \\ air_2 \\ air_3 \\ air_4 \\ water_1 \\ water_2 \\ water_3 \\ water_4 \\ acid_1 \\ acid_2 \\ acid_3 \\ acid_4 \end{bmatrix} \tag{2.9}$$

Add columns to the **Z** matrix for each new variable.

$$\begin{bmatrix} \beta_1 & 0 & 0 & 0 & \beta_2 & 0 & 0 & 0 & \beta_3 & 0 & 0 & 0 \\ 0 & \beta_1 & 0 & 0 & 0 & \beta_2 & 0 & 0 & 0 & \beta_3 & 0 & 0 \\ 0 & 0 & \beta_1 & 0 & 0 & 0 & \beta_2 & 0 & 0 & 0 & \beta_3 & 0 \\ 0 & 0 & 0 & \beta_1 & 0 & 0 & 0 & \beta_2 & 0 & 0 & 0 & \beta_3 \end{bmatrix} \tag{2.10}$$

The number of rows of **Z** is always n, the number of rows of **y**, because the number of rows on the left and right of the equal sign must match. The number of columns in **Z** is determined by the size of **x**. Each explanatory variable (like air flow and wind) appears n times (air_1, air_2, $\dots$, air_n, etc). So if the number of explanatory variables is k, the number of columns in **Z** is $k \times n$. The **a** column matrix holds the intercept terms.

2.3.2 When does Form 2 arise?

Form 2 is similar to how multivariate time series models are typically written for reading by humans (on a whiteboard or paper). In these models, we see equations like this:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix}_t = \begin{bmatrix} \beta_a & \beta_b \\ \beta_a & 0.1 \\ \beta_b & \beta_a \\ 0 & \beta_a \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}_t + \begin{bmatrix} a \\ a \\ a \\ a \end{bmatrix} + \begin{bmatrix} e_1 \\ e_2 \\ e_3 \\ e_4 \end{bmatrix}_t \tag{2.11}$$

In this case, $\mathbf{y}_t$ is the set of four observations at time t and $\mathbf{x}_t$ is the set of two explanatory variables at time t. The $\mathbf{Z}$ is showing how we are modeling the effects of x_1 and x_2 on the ys. Notice that the effects are not consistent across the x and y. This model would not be possible to fit with `lm()` but will be easy to fit with `MARSS()`.

2.4 Groups of intercepts

Let's say that the odd numbered plants are in the north and the even numbered are in the south. We want to include this as a factor in our model that affects the intercept. Let's go back to just having air flow be our explanatory variable. Now if the plant is in the north our model is

$$stack.loss_i = \alpha_n + \beta air_i + e_i, \text{ where } e_i \sim \mathrm{N}(0, \sigma^2) \tag{2.12}$$

If the plant is in the south, our model is

$$stack.loss_i = \alpha_s + \beta air_i + e_i, \text{ where } e_i \sim \mathrm{N}(0, \sigma^2) \tag{2.13}$$

We'll add north/south as a factor called 'reg' (region) to our dataframe:

```
dat = cbind(dat, reg = rep(c("n", "s"), 4)[1:4])
dat
```

```
  Air.Flow Water.Temp Acid.Conc. stack.loss reg
1       80         27         89         42   n
2       80         27         88         37   s
3       75         25         90         37   n
4       62         24         87         28   s
```

And we can easily fit this model with `lm()`.

```
fit2 = lm(stack.loss ~ -1 + Air.Flow + reg, data = dat)
coef(fit2)
```

```
  Air.Flow       regn       regs 
 0.5358166 -2.0257880 -5.5429799 
```

The -1 is added to the `lm()` call to get rid of α. We just want the α_n and α_s intercepts coming from our regions.

2.4.1 North/South intercepts in Form 1

Written in matrix form, Form 1 for this model is

$$\begin{bmatrix} stack.loss_1 \\ stack.loss_2 \\ stack.loss_3 \\ stack.loss_4 \end{bmatrix} = \begin{bmatrix} air_1 & 1 & 0 \\ air_2 & 0 & 1 \\ air_3 & 1 & 0 \\ air_4 & 0 & 1 \end{bmatrix} \begin{bmatrix} \beta \\ \alpha_n \\ \alpha_s \end{bmatrix} + \begin{bmatrix} e_1 \\ e_2 \\ e_3 \\ e_4 \end{bmatrix} \tag{2.14}$$

Notice that odd plants get α_n and even plants get α_s. Use `model.matrix()` to see that this is the $\mathbf{Z}$ matrix that `lm()` formed. Notice the matrix output by `model.matrix()` looks exactly like $\mathbf{Z}$ in Equation (2.14).

```
Z = model.matrix(fit2)
Z[1:4, ]
```

```
  Air.Flow regn regs
1       80    1    0
2       80    0    1
3       75    1    0
4       62    0    1
```

We can solve for the parameters using $\mathbf{x} = (\mathbf{Z}^\top\mathbf{Z})^{-1}\mathbf{Z}^\top\mathbf{y}$ as we did for Form 1 before by adding on the 1s and 0s columns we see in the $\mathbf{Z}$ matrix in Equation (2.14). We could build this $\mathbf{Z}$ using the following R code:

```
Z = cbind(dat$Air.Flow, c(1, 0, 1, 0), c(0, 1, 0, 1))
colnames(Z) = c("beta", "regn", "regs")
```

Or just use `model.matrix()`. This will save time when models are more complex.

```
Z = model.matrix(fit2)
Z[1:4, ]
```

```
  Air.Flow regn regs
1       80    1    0
2       80    0    1
3       75    1    0
4       62    0    1
```

Now we can solve for the parameters:

```
y = matrix(dat$stack.loss, ncol = 1)
solve(t(Z) %*% Z) %*% t(Z) %*% y
```

```
               [,1]
Air.Flow  0.5358166
regn     -2.0257880
regs     -5.5429799
```

Compare to the output from `lm()` and you will see it is the same.

```
coef(fit2)
```

```
  Air.Flow       regn       regs
 0.5358166 -2.0257880 -5.5429799
```

2.4.2 North/South intercepts in Form 2

We would write this model in Form 2 as

$$\begin{bmatrix} stack.loss_1 \\ stack.loss_2 \\ stack.loss_3 \\ stack.loss_4 \end{bmatrix} = \begin{bmatrix} \beta & 0 & 0 & 0 \\ 0 & \beta & 0 & 0 \\ 0 & 0 & \beta & 0 \\ 0 & 0 & 0 & \beta \end{bmatrix} \begin{bmatrix} air_1 \\ air_2 \\ air_3 \\ air_4 \end{bmatrix} + \begin{bmatrix} \alpha_n \\ \alpha_s \\ \alpha_n \\ \alpha_s \end{bmatrix} + \begin{bmatrix} e_1 \\ e_2 \\ e_3 \\ e_4 \end{bmatrix} = \mathbf{Zx} + \mathbf{a} + \mathbf{e} \quad (2.15)$$

2.5 Groups of β's

Now let's say that the plants have different owners, Sue and Aneesh, and we want to have β for the air flow effect vary by owner. If the plant is in the north and owned by Sue, the model is

$$stack.loss_i = \alpha_n + \beta_s air_i + e_i, \text{ where } e_i \sim \mathrm{N}(0, \sigma^2) \quad (2.16)$$

If it is in the south and owned by Aneesh, the model is

$$stack.loss_i = \alpha_s + \beta_a air_i + e_i, \text{ where } e_i \sim \mathrm{N}(0, \sigma^2) \quad (2.17)$$

You get the idea.

Now we need to add an operator variable as a factor in our stackloss dataframe. Plants 1,3 are run by Sue and plants 2,4 are run by Aneesh.

```
dat = cbind(dat, owner = c("s", "a"))
dat
```

```
  Air.Flow Water.Temp Acid.Conc. stack.loss reg owner
1       80         27         89         42   n     s
2       80         27         88         37   s     a
3       75         25         90         37   n     s
4       62         24         87         28   s     a
```

Since the operator names can be replicated the length of our data set, R fills in the operator colmun by replicating our string of operator names to the right length, conveniently (or alarmingly).

We can easily fit this model with `lm()` using the ":" notation.

```
coef(lm(stack.loss ~ -1 + Air.Flow:owner + reg, data = dat))
```

```
            regn             regs Air.Flow:ownera Air.Flow:owners
           -38.0             -3.0             0.5             1.0
```

Notice that we have 4 datapoints and are estimating 4 parameters. We are not going to be able to estimate any more parameters than data points. If we want to estimate any more, we'll need to use the fuller stackflow dataset (which has 21 data points).

2.5.1 Owner β's in Form 1

Written in Form 1, this model is

$$\begin{bmatrix} stack.loss_1 \\ stack.loss_2 \\ stack.loss_3 \\ stack.loss_4 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & air_1 \\ 0 & 1 & air_2 & 0 \\ 1 & 0 & 0 & air_3 \\ 0 & 1 & air_4 & 0 \end{bmatrix} \begin{bmatrix} \alpha_n \\ \alpha_s \\ \beta_a \\ \beta_s \end{bmatrix} + \begin{bmatrix} e_1 \\ e_2 \\ e_3 \\ e_4 \end{bmatrix} = \mathbf{Zx} + \mathbf{e} \tag{2.18}$$

The air data have been written to the right of the 1s and 0s for north/south intercepts because that is how `lm()` writes this model in Form 1 and I want to duplicate that (for teaching purposes). Also the β's are ordered to be alphabetical because `lm()` writes the **Z** matrix like that.

Now our model is more complicated and using `model.matrix()` to get our **Z** saves us a lot tedious matrix building.

```
fit3 = lm(stack.loss ~ -1 + Air.Flow:owner + reg, data = dat)
Z = model.matrix(fit3)
Z[1:4, ]
```

```
  regn regs Air.Flow:ownera Air.Flow:owners
1    1    0               0              80
2    0    1              80               0
3    1    0               0              75
4    0    1              62               0
```

Notice the matrix output by `model.matrix()` looks exactly like $\mathbf{Z}$ in Equation (2.18) (ignore the attributes info). Now we can solve for the parameters:

```
y = matrix(dat$stack.loss, ncol = 1)
solve(t(Z) %*% Z) %*% t(Z) %*% y
```

```
                  [,1]
regn             -38.0
regs              -3.0
Air.Flow:ownera    0.5
Air.Flow:owners    1.0
```

Compare to the output from `lm()` and you will see it is the same.

2.5.2 Owner β's in Form 2

To write this model in Form 2, we just add subscripts to the β's in our Form 2 $\mathbf{Z}$ matrix:

$$\begin{bmatrix} stack.loss_1 \\ stack.loss_2 \\ stack.loss_3 \\ stack.loss_4 \end{bmatrix} = \begin{bmatrix} \beta_s & 0 & 0 & 0 \\ 0 & \beta_a & 0 & 0 \\ 0 & 0 & \beta_s & 0 \\ 0 & 0 & 0 & \beta_a \end{bmatrix} \begin{bmatrix} air_1 \\ air_2 \\ air_3 \\ air_4 \end{bmatrix} + \begin{bmatrix} \alpha_n \\ \alpha_s \\ \alpha_n \\ \alpha_s \end{bmatrix} + \begin{bmatrix} e_1 \\ e_2 \\ e_3 \\ e_4 \end{bmatrix} = \mathbf{Zx}+\mathbf{a}+\mathbf{e} \quad (2.19)$$

2.6 Seasonal effect as a factor

Let's imagine that the data were taken consecutively in time by quarter. We want to model the seasonal effect as an intercept change. We will drop all other effects for now. If the data were collected in quarter 1, the model is

$$stack.loss_i = \alpha_1 + e_i, \text{ where } e_i \sim \mathrm{N}(0, \sigma^2) \quad (2.20)$$

If collected in quarter 2, the model is

$$stack.loss_i = \alpha_2 + e_i, \text{ where } e_i \sim \mathrm{N}(0, \sigma^2) \quad (2.21)$$

etc.

We add a column to our dataframe to account for season:

```
dat = cbind(dat, qtr = paste(rep("qtr", 4), 1:4, sep = ""))
dat
```

```
  Air.Flow Water.Temp Acid.Conc. stack.loss reg owner  qtr
1       80         27         89         42   n     s qtr1
2       80         27         88         37   s     a qtr2
3       75         25         90         37   n     s qtr3
4       62         24         87         28   s     a qtr4
```

And we can easily fit this model with `lm()`.

```
coef(lm(stack.loss ~ -1 + qtr, data = dat))
```

```
qtrqtr1 qtrqtr2 qtrqtr3 qtrqtr4
     42      37      37      28
```

The -1 is added to the `lm()` call to get rid of α. We just want the α_1, α_2, etc. intercepts coming from our quarters.

For comparison look at

```
coef(lm(stack.loss ~ qtr, data = dat))
```

```
(Intercept)     qtrqtr2     qtrqtr3     qtrqtr4
         42          -5          -5         -14
```

Why does it look like that when -1 is missing from the `lm()` call? Where did the intercept for quarter 1 go and why are the other intercepts so much smaller?

2.6.1 Seasonal intercepts written in Form 1

Remembering that `lm()` puts models in Form 1, look at the $\mathbf{Z}$ matrix for Form 1:

```
fit4 = lm(stack.loss ~ -1 + qtr, data = dat)
Z = model.matrix(fit4)
Z[1:4, ]
```

```
  qtrqtr1 qtrqtr2 qtrqtr3 qtrqtr4
1       1       0       0       0
2       0       1       0       0
3       0       0       1       0
4       0       0       0       1
```

Written in Form 1, this model is

$$\begin{bmatrix} stack.loss_1 \\ stack.loss_2 \\ stack.loss_3 \\ stack.loss_4 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ \alpha_4 \end{bmatrix} + \begin{bmatrix} e_1 \\ e_2 \\ e_3 \\ e_4 \end{bmatrix} = \mathbf{Zx} + \mathbf{e} \tag{2.22}$$

Compare to the model that `lm()` is using when the intercept included. What does this model look like written in matrix form?

```
fit5 = lm(stack.loss ~ qtr, data = dat)
Z = model.matrix(fit5)
Z[1:4, ]
```

```
  (Intercept) qtrqtr2 qtrqtr3 qtrqtr4
1           1       0       0       0
2           1       1       0       0
3           1       0       1       0
4           1       0       0       1
```

2.6.2 Seasonal intercepts written in Form 2

We do not need to add 1s and 0s to our $\mathbf{Z}$ matrix in Form 2; we just add subscripts to our intercepts matrix like we did when we had north-south intercepts. In this model, we do not have any explanatory variables so **Zx** does not appear.

$$\begin{bmatrix} stack.loss_1 \\ stack.loss_2 \\ stack.loss_3 \\ stack.loss_4 \end{bmatrix} = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ \alpha_4 \end{bmatrix} + \begin{bmatrix} e_1 \\ e_2 \\ e_3 \\ e_4 \end{bmatrix} = \mathbf{a} + \mathbf{e} \tag{2.23}$$

2.7 Seasonal effect plus other explanatory variables*

With our four data points, we are limited to estimating four parameters. Let's use the full 21 data points so we can estimate some more complex models. We'll add an owner variable and a quarter variable to the stackloss dataset.

```
data(stackloss, package = "datasets")
fulldat = stackloss
n = nrow(fulldat)
fulldat = cbind(fulldat, owner = rep(c("sue", "aneesh", "joe"),
    n)[1:n], qtr = paste("qtr", rep(1:4, n)[1:n], sep = ""),
    reg = rep(c("n", "s"), n)[1:n])
```

Let's fit a model where there is only an effect of air flow, but that effect varies by owner and by quarter. We also want a different intercept for each quarter. So if datapoint i is from quarter j on a plant owned by owner k, the model is

$$stack.loss_i = \alpha_j + \beta_{j,k} air_i + e_i \tag{2.24}$$

So there there are 4×3 β's (4 quarters and 3 owners) and 4 α's (4 quarters).

With `lm()`, we fit the model as:

```
fit7 = lm(stack.loss ~ -1 + qtr + Air.Flow:qtr:owner, data = fulldat)
```

Take a look at **Z** for Form 1 using `model.matrix(Z)`. It's not shown since it is large:

```
model.matrix(fit7)
```

The **x** will be

$$\begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ \alpha_4 \\ \beta_{1,a} \\ \beta_{2,a} \\ \beta_{3,a} \\ \dots \end{bmatrix} \tag{2.25}$$

Take a look at the model matrix that `lm()` is using and make sure you understand how **Zx** produces Equation (2.24).

```
Z = model.matrix(fit7)
```

2.8 Models with confounded parameters*

Try adding region as another factor in your model along with quarter and fit with `lm()`:

```
coef(lm(stack.loss ~ -1 + Air.Flow + reg + qtr, data = fulldat))
```

```
  Air.Flow       regn       regs    qtrqtr2    qtrqtr3    qtrqtr4 
  1.066524 -49.024320 -44.831760  -3.066094   3.499428         NA 
```

The estimate for quarter 1 is gone (actually it was set to 0) and the estimate for quarter 4 is NA. Look at the **Z** matrix for Form 1 and see if you can figure out the problem. Try also writing out the model for the 1st plant and you'll see what part of the problem is and why the estimate for quarter 1 is fixed at 0.

```
fit = lm(stack.loss ~ -1 + Air.Flow + reg + qtr, data = fulldat)
Z = model.matrix(fit)
```

But why is the estimate for quarter 4 equal to NA? What if the ordering of north and south regions was different, say 1 through 4 north, 5 through 8 south, 9 through 12 north, etc?

```
fulldat2 = fulldat
fulldat2$reg2 = rep(c("n", "n", "n", "n", "s", "s", "s", "s"),
    3)[1:21]
fit = lm(stack.loss ~ Air.Flow + reg2 + qtr, data = fulldat2)
coef(fit)
```

```
(Intercept)    Air.Flow       reg2s     qtrqtr2     qtrqtr3     qtrqtr4
-45.6158421   1.0407975  -3.5754722   0.7329027   3.0389763   3.6960928
```

Now an estimate for quarter 4 appears.

The problem is two-fold. First by having both region and quarter intercepts, we created models where 2 intercepts appear for one i model and we cannot estimate both. `lm()` helps us out by setting one of the factor effects to 0. It will chose the first alphabetically. But as we saw with the model where odd numbered plants were north and even numbered were south, we can still have a situation where one of the intercepts is non-identifiable. `lm()` helps us out by alerting us to the problem by setting one to NA.

Once you start developing your own models, you will need to make sure that all your parameters are identifiable. If they are not, your code will simply 'chase its tail'. The code will generally take forever to converge or if you did not try different starting conditions, it may look like it converged but actually the estimates for the confounded parameters are meaningless. So you will need to think carefully about the model you are fitting and consider if there are multiple parameters measuring the same thing (for example 2 intercept parameters).

2.9 Solving for the parameters for Form 2*

Solving for the parameters when the model is written in Form 2 is not straight-forward. We could re-write the model in Form 1, or another approach is to use Kronecker products and permutation matrices.

To solve for α and β, we need our parameters in a column matrix like so $\left[\begin{smallmatrix}\alpha\\ \beta\end{smallmatrix}\right]$. We start by moving the intercept matrix, $\mathbf{a}$ into $\mathbf{Z}$.

$$\begin{bmatrix} stack.loss_1 \\ stack.loss_2 \\ stack.loss_3 \\ stack.loss_4 \end{bmatrix} = \begin{bmatrix} \alpha & \beta & 0 & 0 & 0 \\ \alpha & 0 & \beta & 0 & 0 \\ \alpha & 0 & 0 & \beta & 0 \\ \alpha & 0 & 0 & 0 & \beta \end{bmatrix} \begin{bmatrix} 1 \\ air_1 \\ air_2 \\ air_3 \\ air_4 \end{bmatrix} + \begin{bmatrix} e_1 \\ e_2 \\ e_3 \\ e_4 \end{bmatrix} = \mathbf{Zx} + \mathbf{e}. \tag{2.26}$$

Then we rewrite $\mathbf{Zx}$ in Equation (2.26) in 'vec' form: if $\mathbf{Z}$ is a $n \times m$ matrix and $\mathbf{x}$ is a matrix with 1 column and m rows, then $\mathbf{Zx} = (\mathbf{x}^\top \otimes \mathbf{I}_n)\,\mathrm{vec}(\mathbf{Z})$. The symbol $\otimes$ means Kronecker product and just ignore it since you'll never see it again in our course (or google 'kronecker product' if you are curious). The "vec" of a matrix is that matrix rearranged as a single column:

$$\mathrm{vec}\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \\ 2 \\ 4 \end{bmatrix}$$

Notice how you just take each column one by one and stack them under each other. In R, the vec is

```
A = matrix(1:6, nrow = 2, byrow = TRUE)
vecA = matrix(A, ncol = 1)
```

$\mathbf{I}_n$ is a $n \times n$ identity matrix, a diagonal matrix with all 0s on the off-diagonals and all 1s on the diagonal. In R, this is simply `diag(n)`.

To show how we solve for α and β, let's use an example with only 3 data points so Equation (2.26) becomes:

$$\begin{bmatrix} stack.loss_1 \\ stack.loss_2 \\ stack.loss_3 \end{bmatrix} = \begin{bmatrix} \alpha & \beta & 0 & 0 \\ \alpha & 0 & \beta & 0 \\ \alpha & 0 & 0 & \beta \end{bmatrix} \begin{bmatrix} 1 \\ air_1 \\ air_2 \\ air_3 \end{bmatrix} + \begin{bmatrix} e_1 \\ e_2 \\ e_3 \end{bmatrix} \tag{2.27}$$

Using $\mathbf{Z}\mathbf{x} = (\mathbf{x}^\top \otimes \mathbf{I}_n)\,\mathrm{vec}(\mathbf{Z})$, this means

$$\begin{bmatrix}\alpha & \beta & 0 & 0\\ \alpha & 0 & \beta & 0\\ \alpha & 0 & 0 & \beta\end{bmatrix}\begin{bmatrix}1\\ air_1\\ air_2\\ air_3\end{bmatrix} = \left(\begin{bmatrix}1 & air_1 & air_2 & air_3\end{bmatrix} \otimes \begin{bmatrix}1&0&0\\0&1&0\\0&0&1\end{bmatrix}\right)\begin{bmatrix}\alpha\\ \alpha\\ \alpha\\ \beta\\ 0\\ 0\\ 0\\ \beta\\ 0\\ 0\\ 0\\ \beta\end{bmatrix} \tag{2.28}$$

We need to rewrite the vec($\mathbf{Z}$) as a 'permutation' matrix times $\left[\begin{smallmatrix}\alpha\\ \beta\end{smallmatrix}\right]$:

$$\begin{bmatrix}\alpha\\ \alpha\\ \alpha\\ \beta\\ 0\\ 0\\ 0\\ \beta\\ 0\\ 0\\ 0\\ \beta\end{bmatrix} = \begin{bmatrix}1&0\\1&0\\1&0\\0&1\\0&0\\0&0\\0&0\\0&1\\0&0\\0&0\\0&0\\0&1\end{bmatrix}\begin{bmatrix}\alpha\\ \beta\end{bmatrix} = \mathbf{P}\mathbf{p} \tag{2.29}$$

where $\mathbf{P}$ is the permutation matrix and $\mathbf{p} = \left[\begin{smallmatrix}\alpha\\ \beta\end{smallmatrix}\right]$. Thus,

$$\mathbf{y} = \mathbf{Z}\mathbf{x} + \mathbf{e} = (\mathbf{x}^\top \otimes \mathbf{I}_n)\mathbf{P}\begin{bmatrix}\alpha\\ \beta\end{bmatrix} = \mathbf{M}\mathbf{p} + \mathbf{e} \tag{2.30}$$

where $\mathbf{M} = (\mathbf{x}^\top \otimes \mathbf{I}_n)\mathbf{P}$. We can solve for $\mathbf{p}$, the parameters, using

$$(\mathbf{M}^\top\mathbf{M})^{-1}\mathbf{M}^\top\mathbf{y}$$

as before.

2.9.0.1 Code to solve for parameters in Form 2

In the homework, you will use the R code in this section to solve for the parameters in Form 2.

```
#make your y and x matrices
y=matrix(dat$stack.loss, ncol=1)
x=matrix(c(1,dat$Air.Flow),ncol=1)
#make the Z matrix
n=nrow(dat) #number of rows in our data file
k=1
#Z has n rows and 1 col for intercept, and n cols for the n air data points
#a list matrix allows us to combine "characters" and numbers
Z=matrix(list(0),n,k*n+1)
Z[,1]="alpha"
diag(Z[1:n,1+1:n])="beta"
#this function creates that permutation matrix for you
P=MARSS:::convert.model.mat(Z)$free[,,1]
M=kronecker(t(x),diag(n))%*%P
solve(t(M)%*%M)%*%t(M)%*%y
```

```
             [,1]
alpha -11.6159170
beta    0.6412918
```

```
coef(lm(dat$stack.loss ~ dat$Air.Flow))
```

```
 (Intercept) dat$Air.Flow
 -11.6159170    0.6412918
```

Go through this code line by line at the R command line. Look at `Z`. It is a list matrix that allows you to combine numbers (the 0s) with character string (names of parameters). Notice that `class(Z[1,3])="numeric"` while `class(Z[1,2])="character"`. This is important. `0` in R is a number while `"0"` would be a character (the name of a parameter). Look at the permutation matrix `P`. Try `MARSS:::convert.model.mat(Z)$free` and see that it returns a 3D matrix, which is why the `[,,1]` appears (to get us a 2D matrix). To use more data points, you can redefine `dat` to say `dat=stackloss` to use all 21 data points.

Here's another example. Rewrite the model with multiple intercepts (Equation (2.15)) as

$$\begin{bmatrix} stack.loss_1 \\ stack.loss_2 \\ stack.loss_3 \\ stack.loss_4 \end{bmatrix} = \begin{bmatrix} \alpha_n & \beta & 0 & 0 & 0 \\ \alpha_s & 0 & \beta & 0 & 0 \\ \alpha_n & 0 & 0 & \beta & 0 \\ \alpha_s & 0 & 0 & 0 & \beta \end{bmatrix} \begin{bmatrix} 1 \\ air_1 \\ air_2 \\ air_3 \\ air_4 \end{bmatrix} + \begin{bmatrix} e_1 \\ e_2 \\ e_3 \\ e_4 \end{bmatrix} = \mathbf{Z}\mathbf{x} + \mathbf{a} + \mathbf{e} \quad (2.31)$$

To estimate the parameters, we need to be able to write a list matrix that looks like **Z** in Equation (2.31). We can use the same code as above with **Z** changed to look like that in Equation (2.31).

```
y = matrix(dat$stack.loss, ncol = 1)
x = matrix(c(1, dat$Air.Flow), ncol = 1)
n = nrow(dat)
k = 1
# list matrix allows us to combine numbers and character
# strings
Z = matrix(list(0), n, k * n + 1)
Z[seq(1, n, 2), 1] = "alphanorth"
Z[seq(2, n, 2), 1] = "alphasouth"
diag(Z[1:n, 1 + 1:n]) = "beta"
P = MARSS:::convert.model.mat(Z)$free[, , 1]
M = kronecker(t(x), diag(n)) %*% P
solve(t(M) %*% M) %*% t(M) %*% y
```

```
                 [,1]
alphanorth -2.0257880
alphasouth -5.5429799
beta        0.5358166
```

Similarly to estimate the parameters for Equation (2.19), we change the β's in our **Z** list matrix to have owner designations:

```
Z = matrix(list(0), n, k * n + 1)
Z[seq(1, n, 2), 1] = "alphanorth"
Z[seq(2, n, 2), 1] = "alphasouth"
diag(Z[1:n, 1 + 1:n]) = rep(c("beta.s", "beta.a"), n)[1:n]
P = MARSS:::convert.model.mat(Z)$free[, , 1]
M = kronecker(t(x), diag(n)) %*% P
solve(t(M) %*% M) %*% t(M) %*% y
```

```
             [,1]
alphanorth -38.0
alphasouth  -3.0
beta.s       1.0
beta.a       0.5
```

The parameters estimates are the same as with the model in Form 1, though β's are given in reversed order simply due to the way `convert.model.mat()` is ordering the columns in Form 2's $\mathbf{Z}$.

2.10 Problems

For the homework questions, we will using part of the `airquality` data set in R. Load that as

```
data(airquality, package="datasets")
#remove any rows with NAs omitted.
airquality=na.omit(airquality)
#make Month a factor (i.e., the Month number is a name rather than a number)
airquality$Month=as.factor(airquality$Month)
#add a region factor
airquality$region = rep(c("north","south"),60)[1:111]
#Only use 5 data points for the homework so you can show the matrices easily
homeworkdat = airquality[1:5,]
```

1. Using Form 1 $\mathbf{y} = \mathbf{Z}\mathbf{x} + \mathbf{e}$, write out the model, showing the $\mathbf{Z}$ and $\mathbf{x}$ matrices, being fit by this command

```
fit = lm(Ozone ~ Wind + Temp, data = homeworkdat)
```

2. For the above model, write out the following R code.

   a. Create the $\mathbf{y}$ and $\mathbf{Z}$ matrices in R.
   b. Solve for $\mathbf{x}$ (the parameters). Show that they match what you get from the first `lm()` call.

3. Add -1 to your `lm()` call in question 1:

```
fit = lm(Ozone ~ -1 + Wind + Temp, data = homeworkdat)
```

   a. What changes in your model?
   b. Write out the in Form 1 as an equation. Show the new $\mathbf{Z}$ and $\mathbf{x}$ matrices.
   c. Solve for the parameters ($\mathbf{x}$) and show they match what is returned by `lm()`.

4. For the model for question 1,

   a. Write in Form 2 as an equation.
   b. Adapt the code from subsection 2.9.0.1 and construct new `Z`, `y` and `x` in R code.
   c. Solve for the parameters using the code from subsection 2.9.0.1.

5. A model of the ozone data with only a region (north/south) effect can be written:

```r
fit = lm(Ozone ~ -1 + region, data = homeworkdat)
```

   a. Write this model in Form 1 as an equation.
   b. Solve for the parameter values and show that they match what you get from the `lm()` call.

6. Using the same model from question 5,

   a. Write the model in Form 2 as an equation.
   b. Write out the `Z` and `x` in R code.
   c. Solve for the parameter values and show that they match what you get from the `lm()` call. To do this, you adapt the code from subsection 2.9.0.1.

7. Write the model below in Form 2 as an equation. Show the $\mathbf{Z}$, $\mathbf{y}$ and $\mathbf{x}$ matrices.

```r
fit = lm(Ozone ~ Temp:region, data = homeworkdat)
```

8. Using the airquality dataset with 111 data points

   a. Write the model below in Form 2.

```r
fit = lm(Ozone ~ -1 + Temp:region + Month, data = airquality)
```

   b. Solve for the parameters by adapting code from subsection 2.9.0.1.

Chapter 3

Introduction to time series

At a very basic level, a time series is a set of observations taken sequentially in time. It is different than non-temporal data because each data point has an order and is, typically, related to the data points before and after by some process.

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

3.1 Examples of time series

```
data(WWWusage, package = "datasets")
par(mai = c(0.9, 0.9, 0.1, 0.1), omi = c(0, 0, 0, 0))
plot.ts(WWWusage, ylab = "", las = 1, col = "blue", lwd = 2)
```

```
data(lynx, package = "datasets")
par(mai = c(0.9, 0.9, 0.1, 0.1), omi = c(0, 0, 0, 0))
plot.ts(lynx, ylab = "", las = 1, col = "blue", lwd = 2)
```

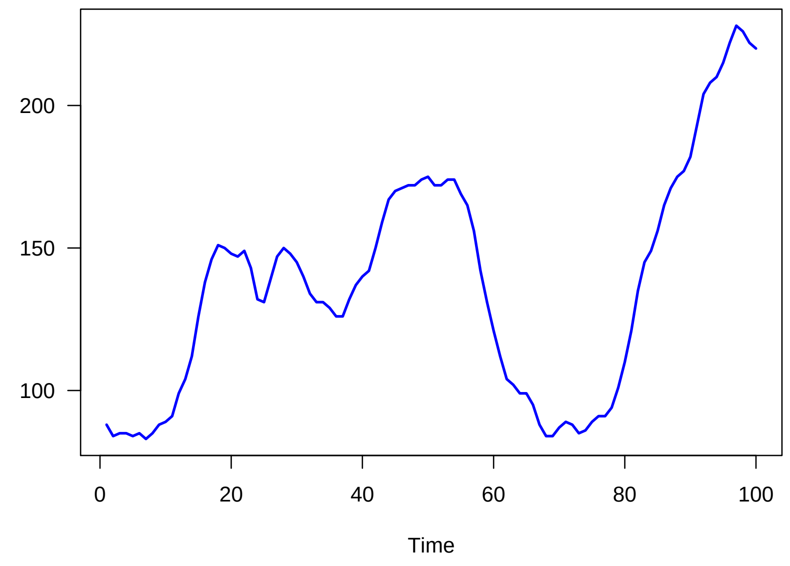


Figure 3.1: Number of users connected to the internet

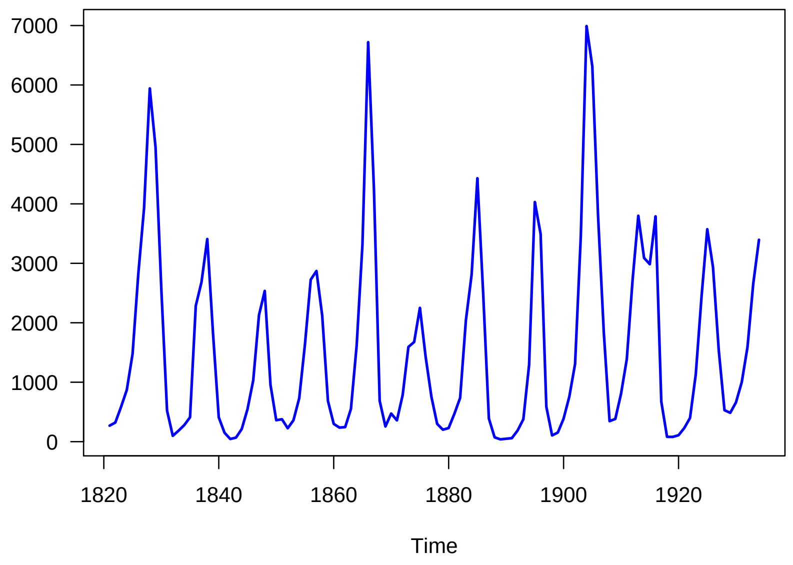


Figure 3.2: Number of lynx trapped in Canada from 1821-1934

3.2 Classification of time series

A ts can be represented as a set

$$\{x_1, x_2, x_3, \dots, x_n\}$$

For example,

$$\{10, 31, 27, 42, 53, 15\}$$

It can be further classified.

3.2.1 By some *index set*

Interval across real time; $x(t)$

- begin/end: $t \in [1.1, 2.5]$

Discrete time; x_t

- Equally spaced: $t = \{1, 2, 3, 4, 5\}$
- Equally spaced w/ missing value: $t = \{1, 2, 4, 5, 6\}$
- Unequally spaced: $t = \{2, 3, 4, 6, 9\}$

3.2.2 By the *underlying process*

Discrete (eg, total # of fish caught per trawl)

Continuous (eg, salinity, temperature)

3.2.3 By the *number of values recorded*

Univariate/scalar (eg, total # of fish caught)

Multivariate/vector (eg, # of each spp of fish caught)

3.2.4 By the *type of values recorded*

Integer (eg, # of fish in 5 min trawl = 2413)

Rational (eg, fraction of unclipped fish = 47/951)

Real (eg, fish mass = 10.2 g)

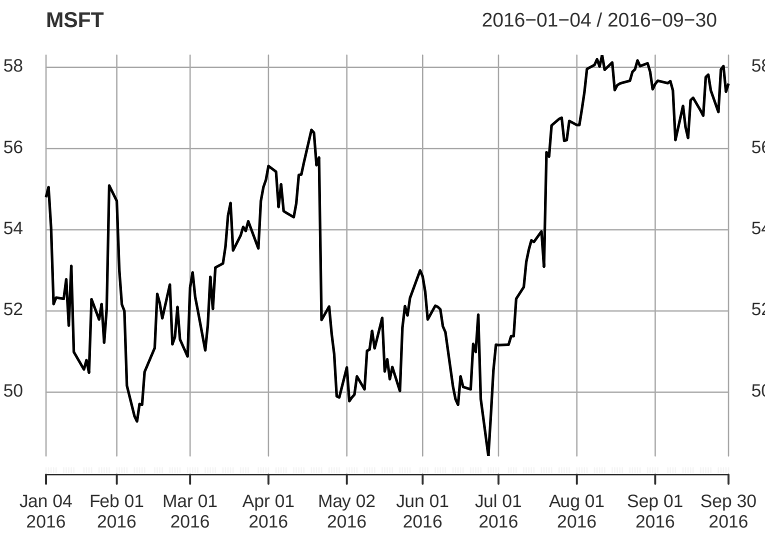
Complex (eg, cos(2 π 2.43) + i sin(2 π 2.43))

3.3 Statistical analyses of time series

Most statistical analyses are concerned with estimating properties of a population from a sample. For example, we use fish caught in a seine to infer the mean size of fish in a lake. Time series analysis, however, presents a different situation:

- Although we could vary the *length* of an observed time series, it is often impossible to make multiple observations at a *given* point in time

For example, one can't observe today's closing price of Microsoft stock more than once. Thus, conventional statistical procedures, based on large sample estimates, are inappropriate.



3.4 What is a time series model?

We use a time series model to analyze time series data. A *time series model* for $\{x_t\}$ is a specification of the joint distributions of a sequence of random variables $\{X_t\}$, of which $\{x_t\}$ is thought to be a realization.

Here is a plot of many realizations from a time series model.

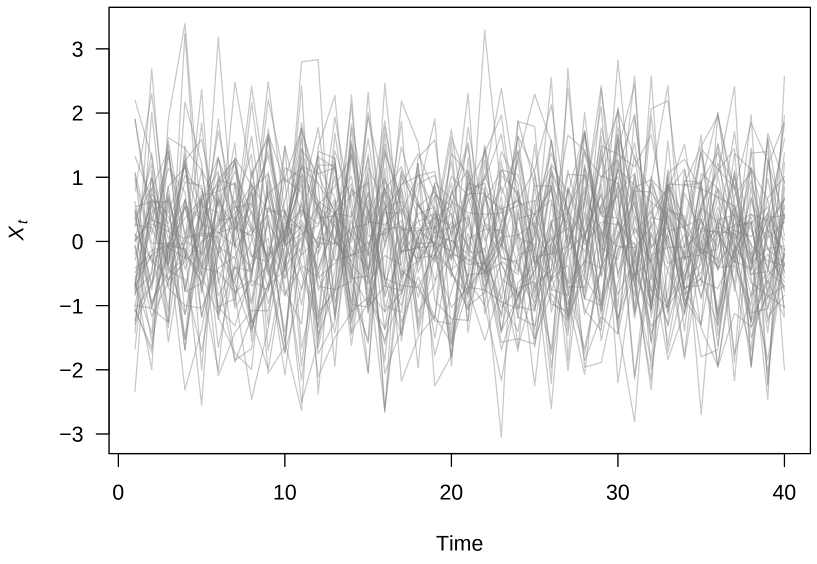


Figure 3.3: Distribution of realizations

These lines represent the distribution of possible realizations. However, we have only one realization. The time series model allows us to use the one realization we have to make inferences about the underlying joint distribution from whence our realization came.

3.5 Two simple and classic time series models

White noise: $x_t \sim N(0, 1)$

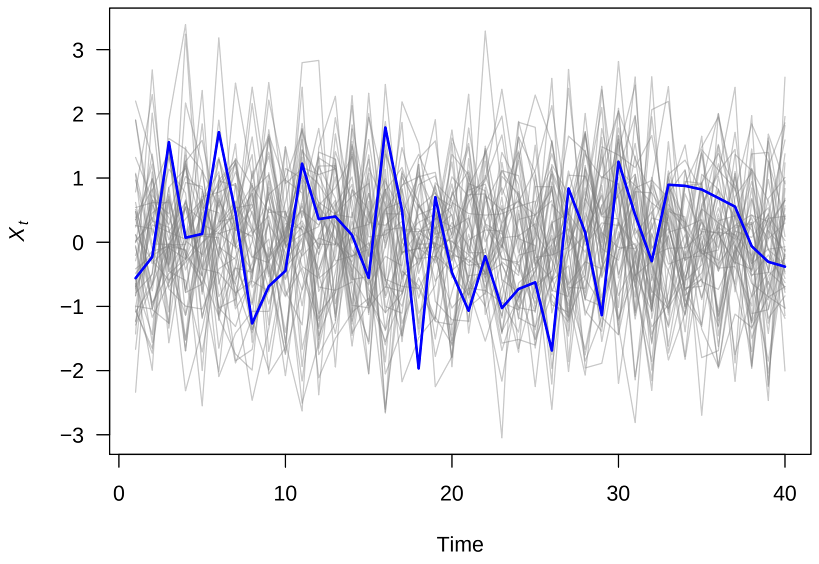
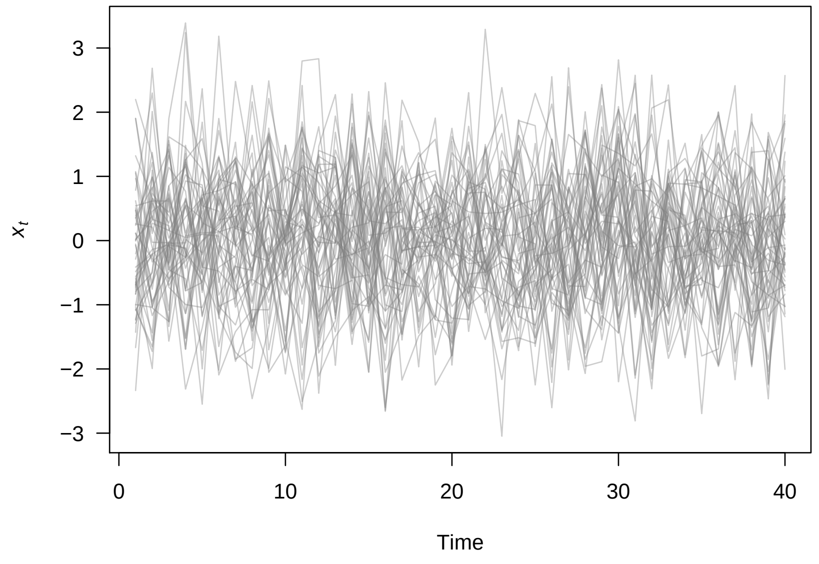


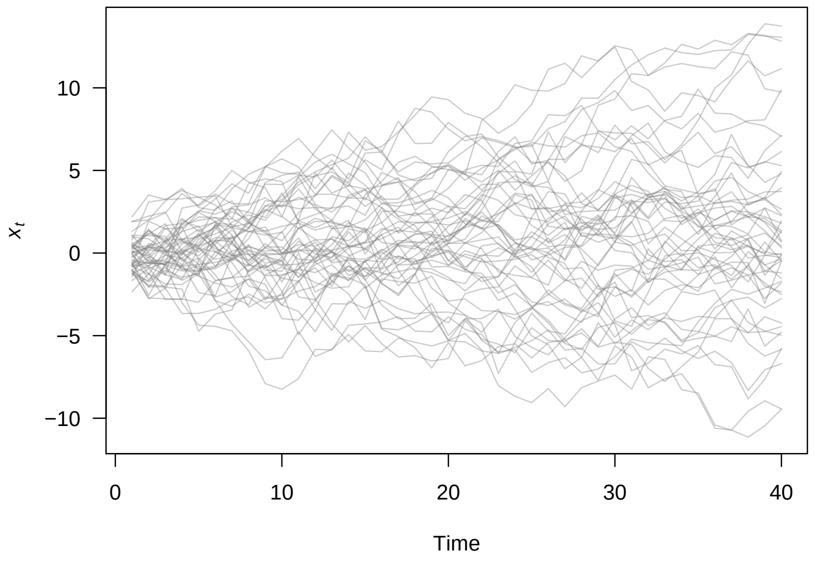
Figure 3.4: Blue line is our one realization.

```r
par(mai = c(0.9, 0.9, 0.1, 0.1), omi = c(0, 0, 0, 0))
matplot(ww, type = "l", lty = "solid", las = 1, ylab = expression(italic(x[t])
    xlab = "Time", col = gray(0.5, 0.4))
```



Random walk: $x_t = x_{t-1} + w_t,$ with $w_t \sim N(0,1)$

```r
par(mai = c(0.9, 0.9, 0.1, 0.1), omi = c(0, 0, 0, 0))
matplot(apply(ww, 2, cumsum), type = "l", lty = "solid", las = 1,
    ylab = expression(italic(x[t])), xlab = "Time", col = gray(0.5,
        0.4))
```



3.6 Classical decomposition

Model time series $\{x_t\}$ as a combination of

1. trend (m_t)
2. seasonal component (s_t)
3. remainder (e_t)

$x_t = m_t + s_t + e_t$

3.6.1 1. The trend (m_t)

We need a way to extract the so-called *signal*. One common method is via "linear filters"

$$m_t = \sum_{i=-\infty}^{\infty} \lambda_i x_{t+1}$$

For example, a moving average

$$m_t = \sum_{i=-a}^{a} \frac{1}{2a+1} x_{t+i}$$

If $a = 1$, then

$$m_t = \frac{1}{3}(x_{t-1} + x_t + x_{t+1})$$

3.6.2 Example of linear filtering

Here is a time series.

A linear filter with $a = 3$ closely tracks the data.

As we increase the length of data that is averaged from 1 on each side ($a = 3$) to 4 on each side ($a = 9$), the trend line is smoother.

When we increase up to 13 points on each side ($a = 27$), the trend line is very smooth.

3.6.3 2. Seasonal effect (s_t)

Once we have an estimate of the trend m_t, we can estimate s_t simply by subtraction:

$$s_t = x_t - m_t$$

This is the seasonal effect (s_t), assuming $\lambda = 1/9$, but, s_t includes the remainder e_t as well. Instead we can estimate the mean seasonal effect (s_t).

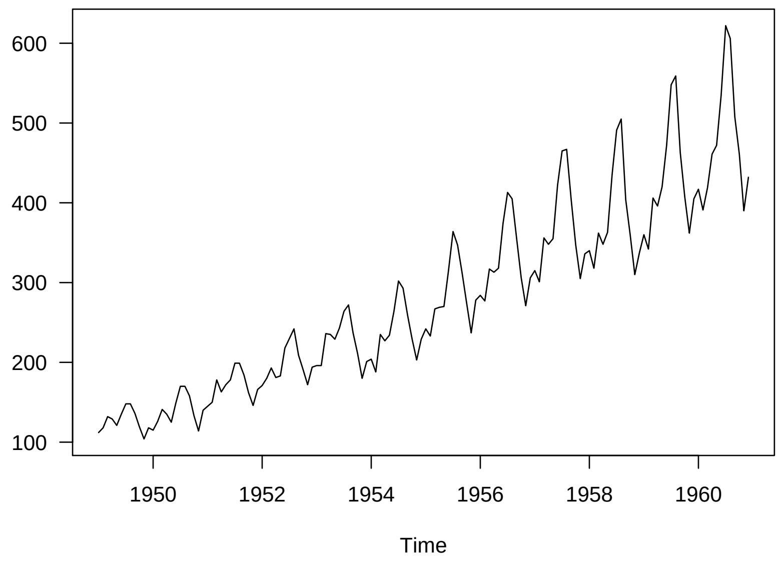


Figure 3.5: Monthly airline passengers from 1949-1960

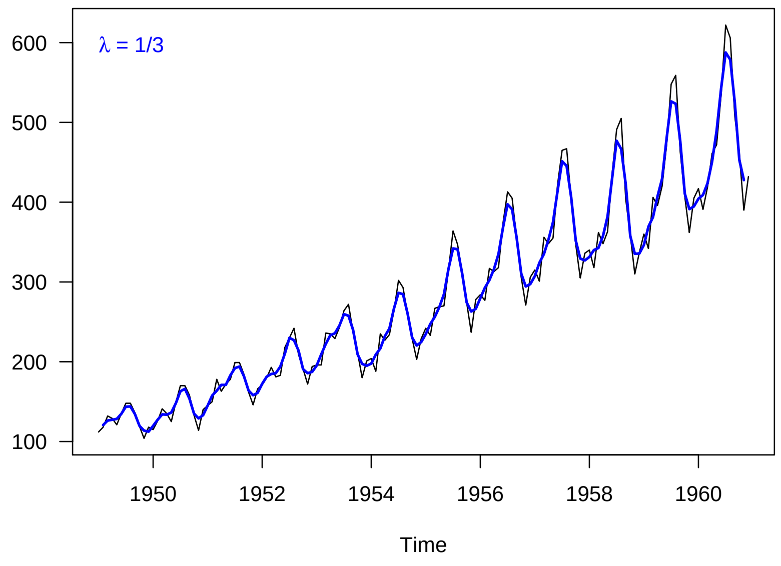


Figure 3.6: Monthly airline passengers from 1949-1960 with a low filter.

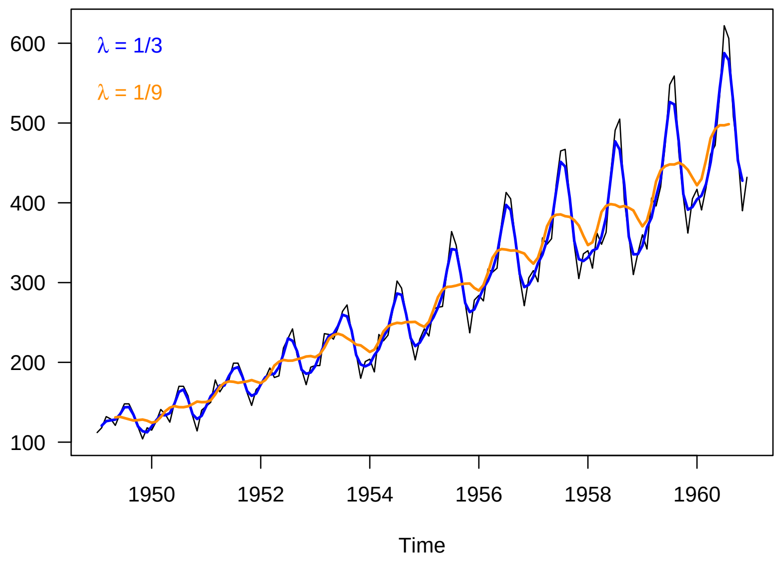


Figure 3.7: Monthly airline passengers from 1949-1960 with a medium filter.

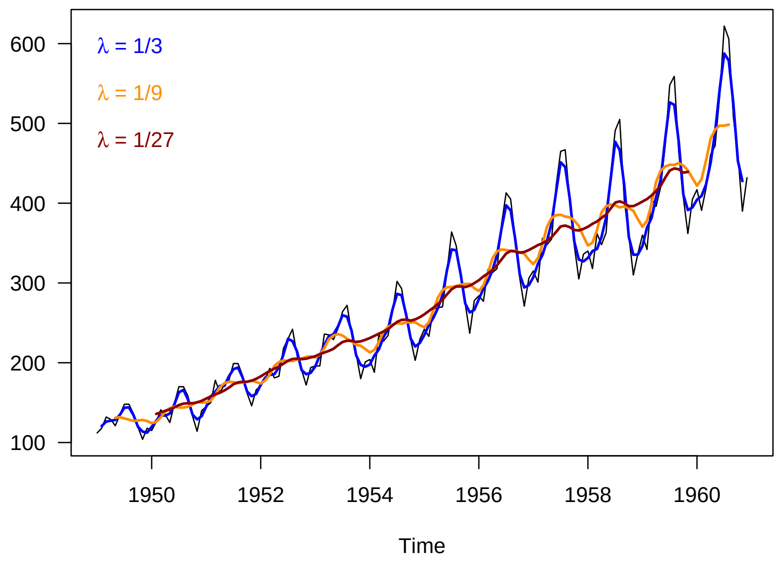
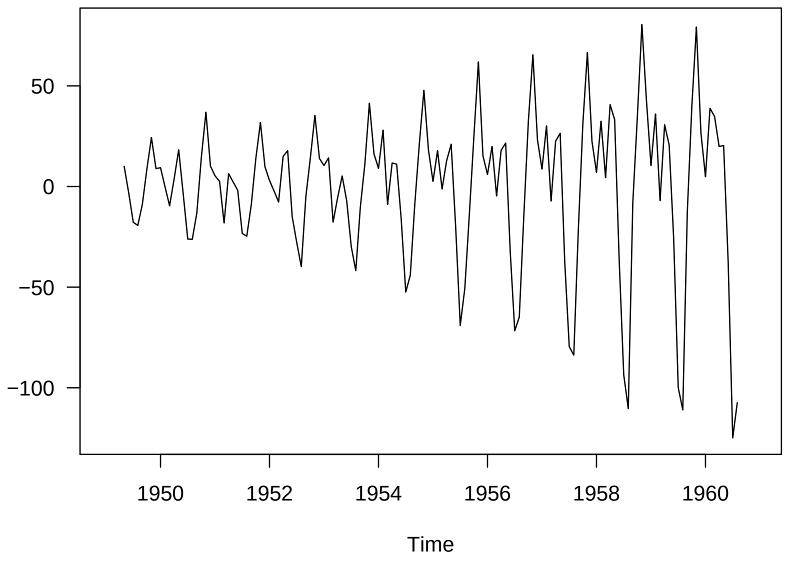


Figure 3.8: Monthly airline passengers from 1949-1960 with a high filter.



```r
seas_2 <- decompose(xx)$seasonal
par(mai = c(0.9, 0.9, 0.1, 0.1), omi = c(0, 0, 0, 0))
plot.ts(seas_2, las = 1, ylab = "")
```

3.6.4 3. Remainder (e_t)

Now we can estimate e_t via subtraction:

$$e_t = x_t - m_t - s_t$$

```r
ee <- decompose(xx)$random
par(mai = c(0.9, 0.9, 0.1, 0.1), omi = c(0, 0, 0, 0))
plot.ts(ee, las = 1, ylab = "")
```

3.7 Decomposition on log-transformed data

Let's repeat the decomposition with the log of the airline data.

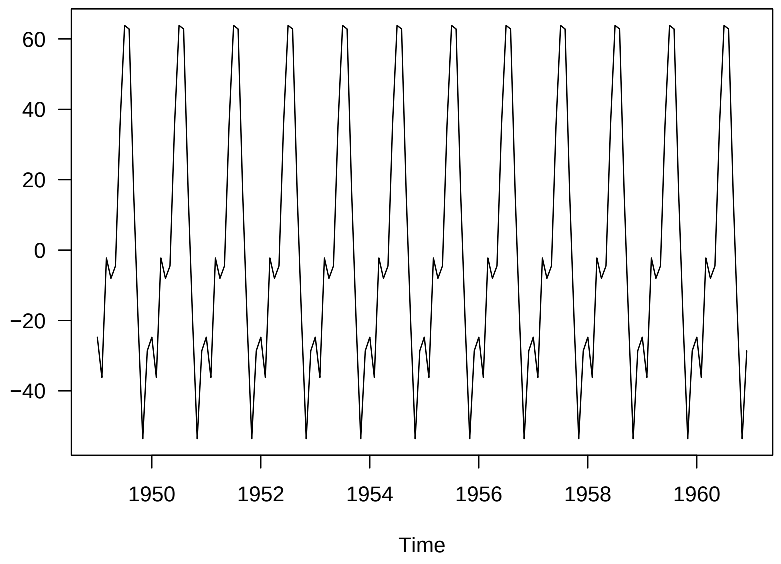


Figure 3.9: Mean seasonal effect.

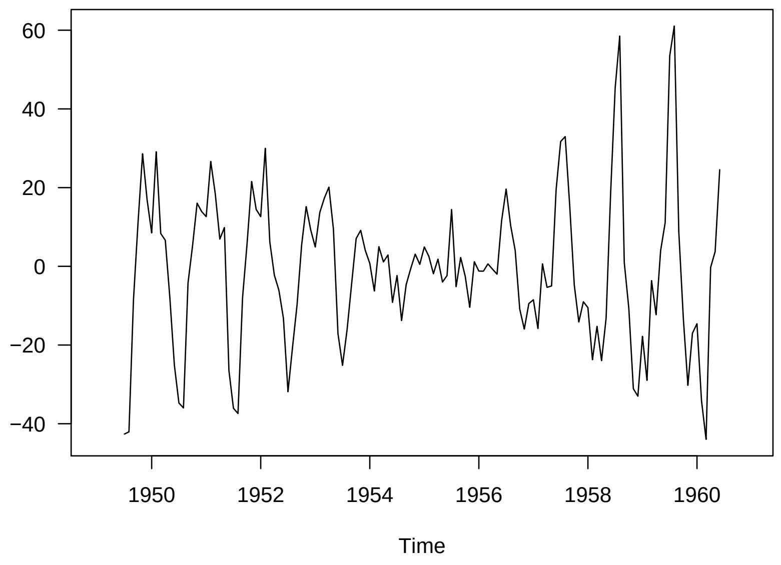


Figure 3.10: Errors.

```
lx <- log(AirPassengers)
par(mai = c(0.9, 0.9, 0.1, 0.1), omi = c(0, 0, 0, 0))
plot.ts(lx, las = 1, ylab = "")
```

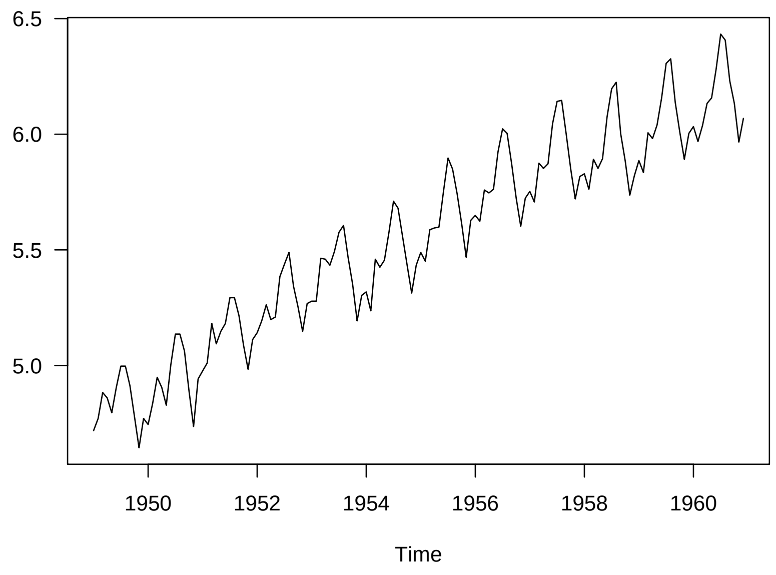
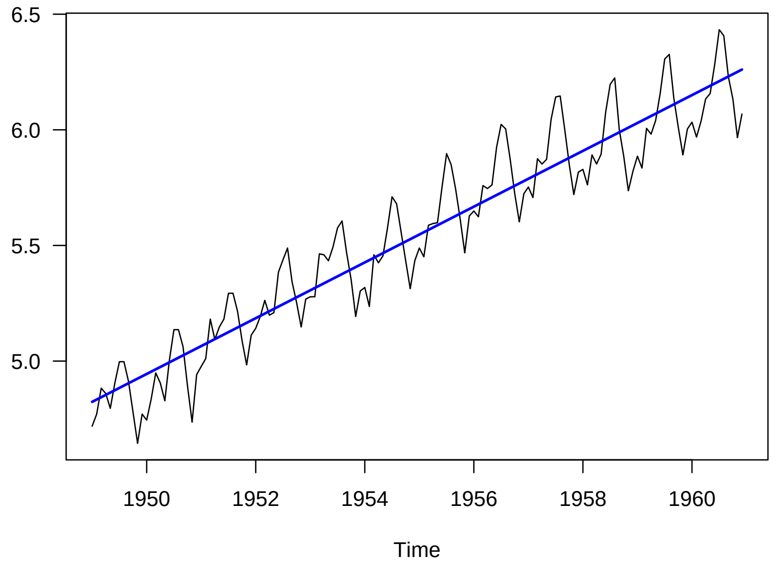
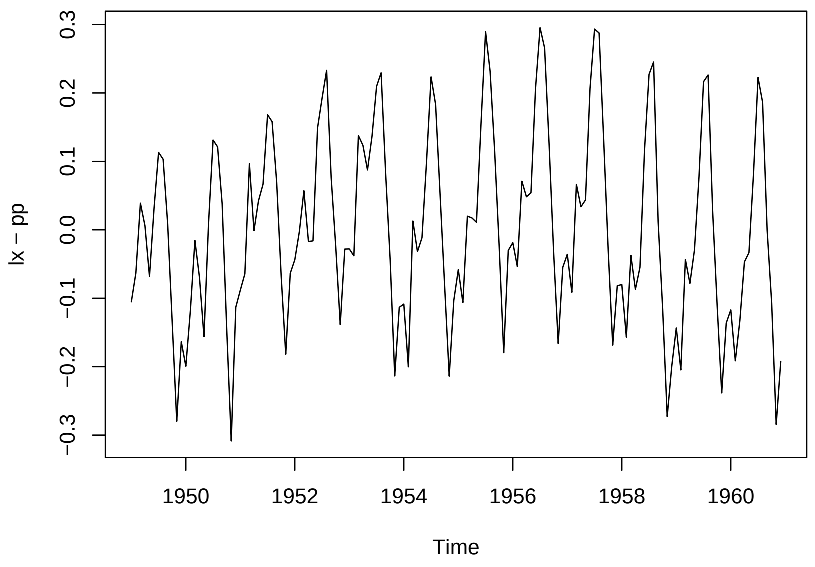


Figure 3.11: Log monthly airline passengers from 1949-1960

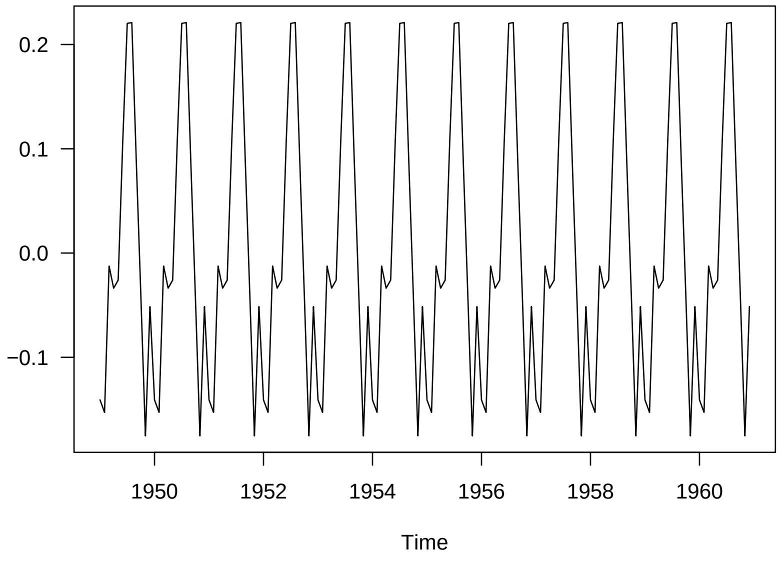
3.7.1 The trend (m_t)



3.7.2 Seasonal effect (s_t) with error (e_t)

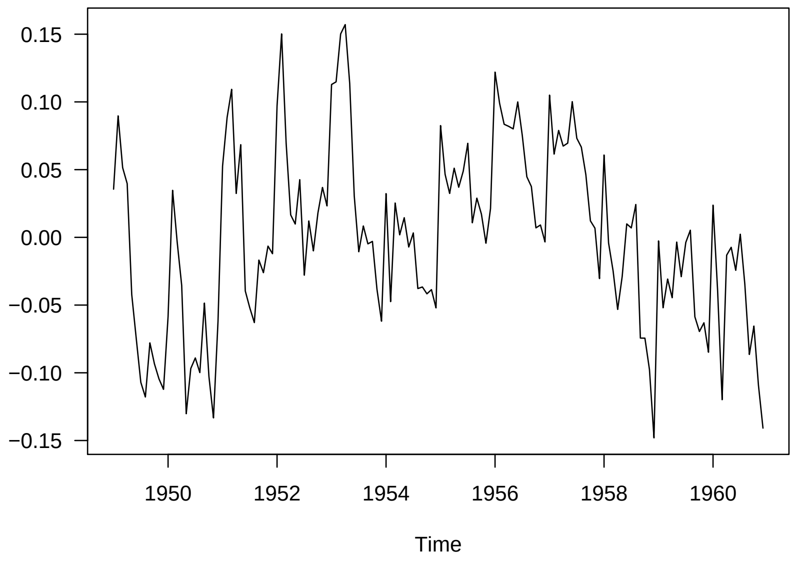


3.7.3 Mean seasonal effect (s_t)



3.7.4 Remainder (e_t)

```
le <- lx - pp - seas_2
par(mai = c(0.9, 0.9, 0.1, 0.1), omi = c(0, 0, 0, 0))
plot.ts(le, las = 1, ylab = "")
```


0.15
0.10
0.05
0.00
−0.05
−0.10
−0.15
1950
1952
1954
1956
1958
1960
Time

Chapter 4

Basic time series functions in R

This chapter introduces you to some of the basic functions in R for plotting and analyzing univariate time series data. Many of the things you learn here will be relevant when we start examining multivariate time series as well. We will begin with the creation and plotting of time series objects in R, and then moves on to decomposition, differencing, and correlation (*e.g.*, ACF, PACF) before ending with fitting and simulation of ARMA models.

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

Data and packages

This chapter uses the **stats** package, which is often loaded by default when you start R, the **MARSS** package and the **forecast** package. The problems use a dataset in the **datasets** package. After installing the packages, if needed, load:

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

The chapter uses data sets which are in the **atsalibrary** package. If needed, install using the **devtools** package.

```
library(devtools)
# Windows users will likely need to set this
# Sys.setenv('R_REMOTES_NO_ERRORS_FROM_WARNINGS' = 'true')
devtools::install_github("nwfsc-timeseries/atsalibrary")
```

The main one is a time series of the atmospheric concentration of CO_2 collected at the Mauna Loa Observatory in Hawai'i (`MLCO2`). The second is Northern Hemisphere land and ocean temperature anomalies from NOAA. (`NHTemp`). The problems use a data set on hourly phytoplankton counts (`hourlyphyto`). Use `?MLCO2`, `?NHTemp` and `?hourlyphyto` for information on these datasets.

Load the data.

```
data(NHTemp, package = "atsalibrary")
Temp <- NHTemp
data(MLCO2, package = "atsalibrary")
CO2 <- MLCO2
data(hourlyphyto, package = "atsalibrary")
phyto_dat <- hourlyphyto
```

4.1 Time series plots

Time series plots are an excellent way to begin the process of understanding what sort of process might have generated the data of interest. Traditionally, time series have been plotted with the observed data on the y-axis and time on the x-axis. Sequential time points are usually connected with some form of line, but sometimes other plot forms can be a useful way of conveying important information in the time series (*e.g.*, bAR_p_coeflots of sea-surface temperature anomolies show nicely the contrasting El Niño and La Niña phenomena).

4.1.1 ts objects and `plot.ts()`

The CO_2 data are stored in R as a `data.frame` object, but we would like to transform the class to a more user-friendly format for dealing with time series. Fortunately, the `ts()` function will do just that, and return an object of class `ts` as well. In addition to the data themselves, we need to provide `ts()` with 2 pieces of information about the time index for the data.

The first, `frequency`, is a bit of a misnomer because it does not really refer to the number of cycles per unit time, but rather the number of observations/samples per cycle. So, for example, if the data were collected each hour of a day then `frequency = 24`.

The second, `start`, specifies the first sample in terms of (*day*, *hour*), (*year*, *month*), etc. So, for example, if the data were collected monthly beginning in November of 1969, then `frequency = 12` and `start = c(1969, 11)`. If the data were collected annually, then you simply specify `start` as a scalar (*e.g.*, `start = 1991`) and omit `frequency` (*i.e.*, R will set `frequency = 1` by default).

The Mauna Loa time series is collected monthly and begins in March of 1958, which we can get from the data themselves, and then pass to `ts()`.

```
## create a time series (ts) object from the CO2 data
co2 <- ts(data = CO2$ppm, frequency = 12, start = c(CO2[1, "year"],
    CO2[1, "month"]))
```

Now let's plot the data using `plot.ts()`, which is designed specifically for `ts` objects like the one we just created above. It's nice because we don't need to specify any x-values as they are taken directly from the `ts` object.

```
## plot the ts
plot.ts(co2, ylab = expression(paste("CO"[2], " (ppm)")))
```

Examination of the plotted time series (Figure 4.1) shows 2 obvious features that would violate any assumption of stationarity: 1) an increasing (and perhaps non-linear) trend over time, and 2) strong seasonal patterns. (*Aside*: Do you know the causes of these 2 phenomena?)

4.1.2 Combining and plotting multiple ts objects

Before we examine the CO_2 data further, however, let's see a quick example of how you can combine and plot multiple time series together. We'll use the

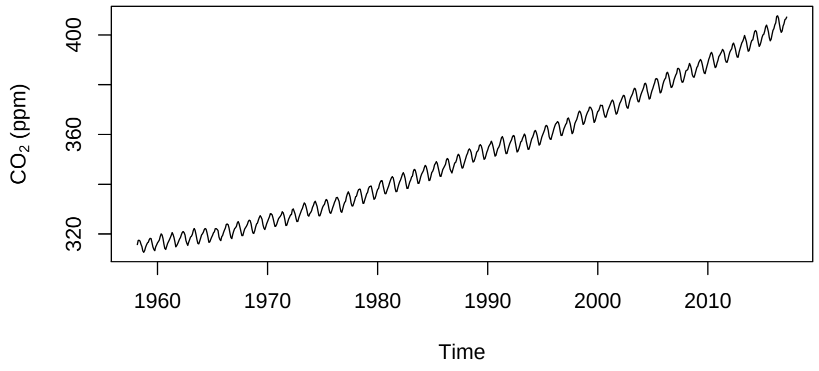


Figure 4.1: Time series of the atmospheric CO_2 concentration at Mauna Loa, Hawai'i measured monthly from March 1958 to present.

data on monthly mean temperature anomolies for the Northern Hemisphere (`Temp`). First convert `Temp` to a `ts` object.

```
temp_ts <- ts(data = Temp$Value, frequency = 12, start = c(1880,
    1))
```

Before we can plot the two time series together, however, we need to line up their time indices because the temperature data start in January of 1880, but the CO_2 data start in March of 1958. Fortunately, the `ts.intersect()` function makes this really easy once the data have been transformed to `ts` objects by trimming the data to a common time frame. Also, `ts.union()` works in a similar fashion, but it pads one or both series with the appropriate number of NA's. Let's try both.

```
## intersection (only overlapping times)
dat_int <- ts.intersect(co2, temp_ts)
## dimensions of common-time data
dim(dat_int)
```

```
[1] 682   2
```

```
## union (all times)
dat_unn <- ts.union(co2, temp_ts)
## dimensions of all-time data
dim(dat_unn)
```

```
[1] 1647    2
```

As you can see, the intersection of the two data sets is much smaller than the union. If you compare them, you will see that the first 938 rows of `dat_unn` contains `NA` in the `co2` column.

It turns out that the regular `plot()` function in R is smart enough to recognize a `ts` object and use the information contained therein appropriately. Here's how to plot the intersection of the two time series together with the y-axes on alternate sides (results are shown in Figure 4.2):

```
## plot the ts
plot(dat_int, main = "", yax.flip = TRUE)
```

4.2 Decomposition of time series

Plotting time series data is an important first step in analyzing their various components. Beyond that, however, we need a more formal means for identifying and removing characteristics such as a trend or seasonal variation. As discussed in lecture, the decomposition model reduces a time series into 3 components: trend, seasonal effects, and random errors. In turn, we aim to model the random errors as some form of stationary process.

Let's begin with a simple, additive decomposition model for a time series x_t

$$x_t = m_t + s_t + e_t, \tag{4.1}$$

where, at time t, m_t is the trend, s_t is the seasonal effect, and e_t is a random error that we generally assume to have zero-mean and to be correlated over time. Thus, by estimating and subtracting both $\{m_t\}$ and $\{s_t\}$ from $\{x_t\}$, we hope to have a time series of stationary residuals $\{e_t\}$.

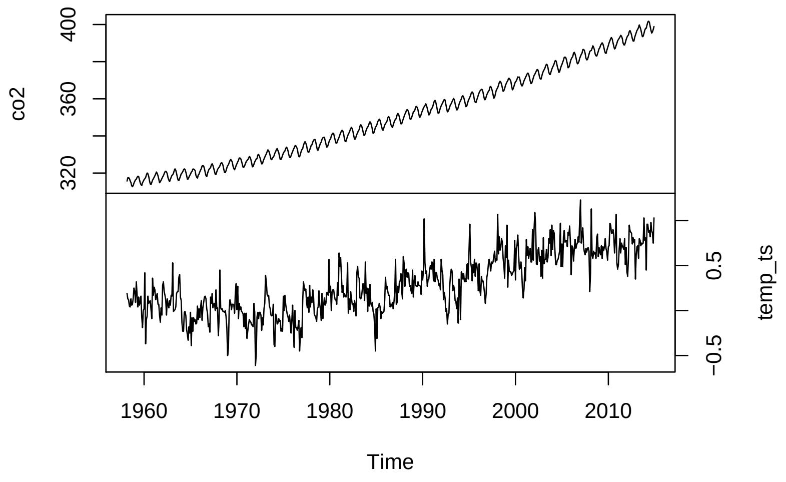


Figure 4.2: Time series of the atmospheric CO_2 concentration at Mauna Loa, Hawai'i (top) and the mean temperature index for the Northern Hemisphere (bottom) measured monthly from March 1958 to present.

4.2.1 Estimating trends

In lecture we discussed how linear filters are a common way to estimate trends in time series. One of the most common linear filters is the moving average, which for time lags from $-a$ to a is defined as

$$\hat{m}_t = \sum_{k=-a}^{a} \left(\frac{1}{1+2a} \right) x_{t+k}. \tag{4.2}$$

This model works well for moving windows of odd-numbered lengths, but should be adjusted for even-numbered lengths by adding only $\frac{1}{2}$ of the 2 most extreme lags so that the filtered value at time t lines up with the original observation at time t. So, for example, in a case with monthly data such as the atmospheric CO_2 concentration where a 12-point moving average would be an obvious choice, the linear filter would be

$$\hat{m}_t = \frac{\frac{1}{2}x_{t-6} + x_{t-5} + \dots + x_{t-1} + x_t + x_{t+1} + \dots + x_{t+5} + \frac{1}{2}x_{t+6}}{12} \tag{4.3}$$

It is important to note here that our time series of the estimated trend $\{\hat{m}_t\}$ is actually shorter than the observed time series by $2a$ units.

Conveniently, R has the built-in function `filter()` in the **stats** package for estimating moving-average (and other) linear filters. In addition to specifying the time series to be filtered, we need to pass in the filter weights (and 2 other arguments we won't worry about here–type `?filter` to get more information). The easiest way to create the filter is with the `rep()` function:

```r
## weights for moving avg
fltr <- c(1/2, rep(1, times = 11), 1/2)/12
```

Now let's get our estimate of the trend $\{\hat{m}\}$ with `filter()`} and plot it:

```r
## estimate of trend
co2_trend <- stats::filter(co2, filter = fltr, method = "convo",
    sides = 2)
## plot the trend
plot.ts(co2_trend, ylab = "Trend", cex = 1)
```

The trend is a more-or-less smoothly increasing function over time, the average slope of which does indeed appear to be increasing over time as well (Figure 4.3).

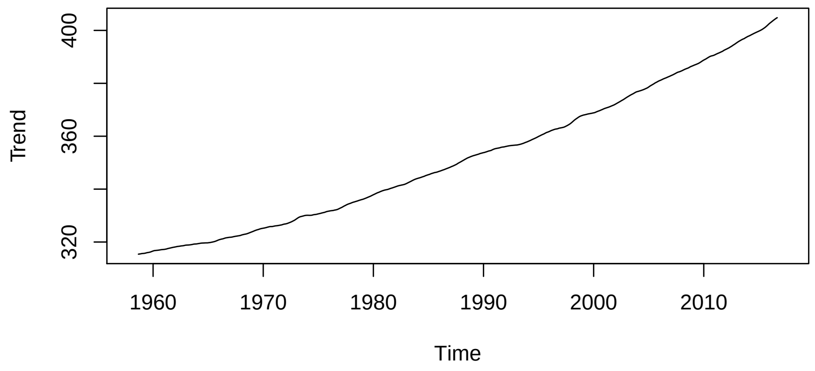


Figure 4.3: Time series of the estimated trend $\{\hat{m}_t\}$ for the atmospheric CO_2 concentration at Mauna Loa, Hawai'i.

4.2.2 Estimating seasonal effects

Once we have an estimate of the trend for time t ($\hat{m}_t$) we can easily obtain an estimate of the seasonal effect at time t ($\hat{s}_t$) by subtraction

$$\hat{s}_t = x_t - \hat{m}_t, \tag{4.4}$$

which is really easy to do in R:

```r
## seasonal effect over time
co2_seas <- co2 - co2_trend
```

This estimate of the seasonal effect for each time t also contains the random error e_t, however, which can be seen by plotting the time series and careful comparison of Equations (4.1) and (4.4).

```r
## plot the monthly seasonal effects
plot.ts(co2_seas, ylab = "Seasonal effect", xlab = "Month", cex = 1)
```

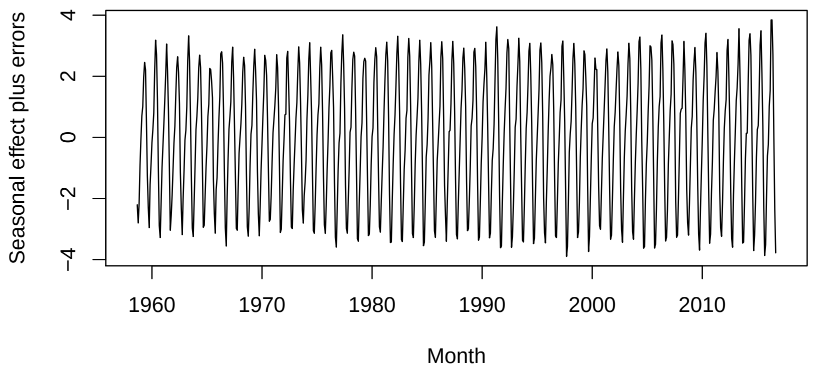


Figure 4.4: Time series of seasonal effects plus random errors for the atmospheric CO_2 concentration at Mauna Loa, Hawai'i, measured monthly from March 1958 to present.

We can obtain the overall seasonal effect by averaging the estimates of $\{\hat{s}_t\}$ for each month and repeating this sequence over all years.

```r
## length of ts
ll <- length(co2_seas)
## frequency (ie, 12)
ff <- frequency(co2_seas)
## number of periods (years); %/% is integer division
periods <- ll%/%ff
## index of cumulative month
index <- seq(1, ll, by = ff) - 1
## get mean by month
mm <- numeric(ff)
for (i in 1:ff) {
    mm[i] <- mean(co2_seas[index + i], na.rm = TRUE)
}
## subtract mean to make overall mean = 0
mm <- mm - mean(mm)
```

Before we create the entire time series of seasonal effects, let's plot them for each month to see what is happening within a year:

```r
## plot the monthly seasonal effects
plot.ts(mm, ylab = "Seasonal effect", xlab = "Month", cex = 1)
```

It looks like, on average, that the CO_2 concentration is highest in spring (March) and lowest in summer (August) (Figure 4.5). (*Aside*: Do you know why this is?)

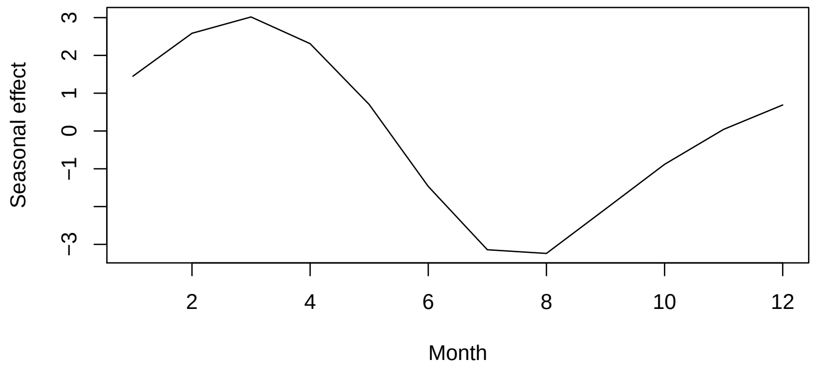


Figure 4.5: Estimated monthly seasonal effects for the atmospheric CO_2 concentration at Mauna Loa, Hawai'i.

Finally, let's create the entire time series of seasonal effects $\{\hat{s}_t\}$:

```r
## create ts object for season
co2_seas_ts <- ts(rep(mm, periods + 1)[seq(ll)], start = start(co2_seas),
    frequency = ff)
```

4.2.3 Completing the model

The last step in completing our full decomposition model is obtaining the random errors $\{\hat{e}_t\}$, which we can get via simple subtraction

$$\hat{e}_t = x_t - \hat{m}_t - \hat{s}_t. \tag{4.5}$$

Again, this is really easy in R:

```
## random errors over time
co2_err <- co2 - co2_trend - co2_seas_ts
```

Now that we have all 3 of our model components, let's plot them together with the observed data $\{x_t\}$. The results are shown in Figure 4.6.

```
## plot the obs ts, trend & seasonal effect
plot(cbind(co2, co2_trend, co2_seas_ts, co2_err), main = "",
    yax.flip = TRUE)
```

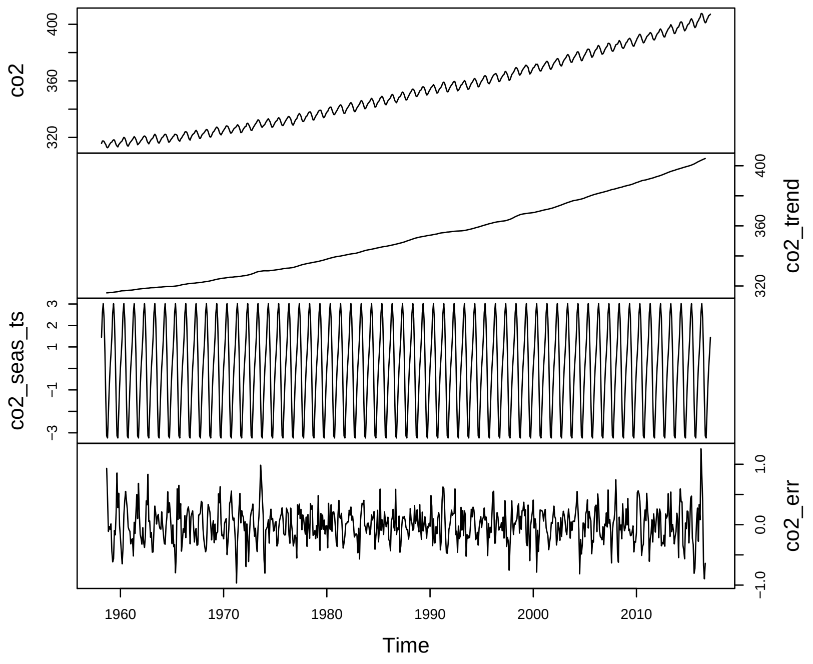


Figure 4.6: Time series of the observed atmospheric CO_2 concentration at Mauna Loa, Hawai'i (top) along with the estimated trend, seasonal effects, and random errors.

4.2.4 Using `decompose()` for decomposition

Now that we have seen how to estimate and plot the various components of a classical decomposition model in a piecewise manner, let's see how to do this in one step in R with the function `decompose()`, which accepts a `ts` object as input and returns an object of class **decomposed.ts**.

```
## decomposition of CO2 data
co2_decomp <- decompose(co2)
```

`co2_decomp` is a list with the following elements, which should be familiar by now:

- `x`: the observed time series $\{x_t\}$
- `seasonal`: time series of estimated seasonal component $\{\hat{s}_t\}$
- `figure`: mean seasonal effect (`length(figure) == frequency(x)`)
- `trend`: time series of estimated trend $\{\hat{m}_t\}$
- `random`: time series of random errors $\{\hat{e}_t\}$
- `type`: type of error (`"additive"` or `"multiplicative"`)

We can easily make plots of the output and compare them to those in Figure 4.6:

```
## plot the obs ts, trend & seasonal effect
plot(co2_decomp, yax.flip = TRUE)
```

The results obtained with `decompose()` (Figure 4.7) are identical to those we estimated previously.

Another nice feature of the `decompose()` function is that it can be used for decomposition models with multiplicative (*i.e.*, non-additive) errors (*e.g.*, if the original time series had a seasonal amplitude that increased with time). To do, so pass in the argument `type = "multiplicative"`, which is set to `type = "additive"` by default.

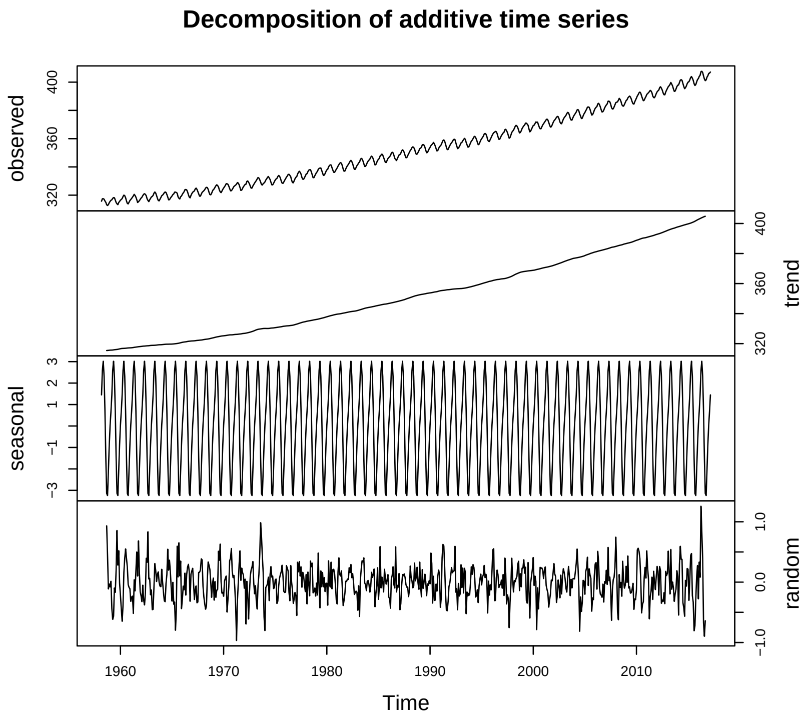


Figure 4.7: Time series of the observed atmospheric CO_2 concentration at Mauna Loa, Hawai'i (top) along with the estimated trend, seasonal effects, and random errors obtained with the function `decompose()`.

4.3 Differencing to remove a trend or seasonal effects

An alternative to decomposition for removing trends is differencing. We saw in lecture how the difference operator works and how it can be used to remove linear and nonlinear trends as well as various seasonal features that might be evident in the data. As a reminder, we define the difference operator as

$$\nabla x_t = x_t - x_{t-1}, \tag{4.6}$$

and, more generally, for order d

$$\nabla^d x_t = (1 - \mathbf{B})^d x_t, \tag{4.7}$$

where $\mathbf{B}$ is the backshift operator (*i.e.*, $\mathbf{B}^k x_t = x_{t-k}$ for $k \geq 1$).

So, for example, a random walk is one of the most simple and widely used time series models, but it is not stationary. We can write a random walk model as

$$x_t = x_{t-1} + w_t, \text{ with } w_t \sim \text{N}(0, q). \tag{4.8}$$

Applying the difference operator to Equation (4.8) will yield a time series of Gaussian white noise errors $\{w_t\}$:

$$\begin{aligned} \nabla(x_t &= x_{t-1} + w_t) \\ x_t - x_{t-1} &= x_{t-1} - x_{t-1} + w_t \\ x_t - x_{t-1} &= w_t \end{aligned} \tag{4.9}$$

4.3.1 Using the `diff()` function

In R we can use the `diff()` function for differencing a time series, which requires 3 arguments: `x` (the data), `lag` (the lag at which to difference), and `differences` (the order of differencing; d in Equation (4.7)). For example, first-differencing a time series will remove a linear trend (*i.e.*, `differences = 1`); twice-differencing will remove a quadratic trend (*i.e.*, `differences =`

2). In addition, first-differencing a time series at a lag equal to the period will remove a seasonal trend (*e.g.*, set `lag = 12` for monthly data).

Let's use `diff()` to remove the trend and seasonal signal from the CO_2 time series, beginning with the trend. Close inspection of Figure 4.1 would suggest that there is a nonlinear increase in CO_2 concentration over time, so we'll set `differences = 2`):

```
## twice-difference the CO2 data
co2_d2 <- diff(co2, differences = 2)
## plot the differenced data
plot(co2_d2, ylab = expression(paste(nabla^2, "CO"[2])))
```

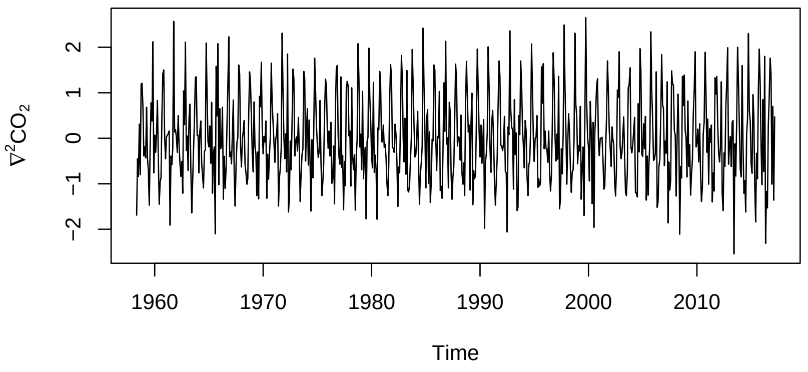


Figure 4.8: Time series of the twice-differenced atmospheric CO_2 concentration at Mauna Loa, Hawai'i.

We were apparently successful in removing the trend, but the seasonal effect still appears obvious (Figure 4.8). Therefore, let's go ahead and difference that series at lag-12 because our data were collected monthly.

```
## difference the differenced CO2 data
co2_d2d12 <- diff(co2_d2, lag = 12)
## plot the newly differenced data
plot(co2_d2d12, ylab = expression(paste(nabla, "(", nabla^2,
    "CO"[2], ")")))
```

Now we have a time series that appears to be random errors without any obvious trend or seasonal components (Figure 4.9).

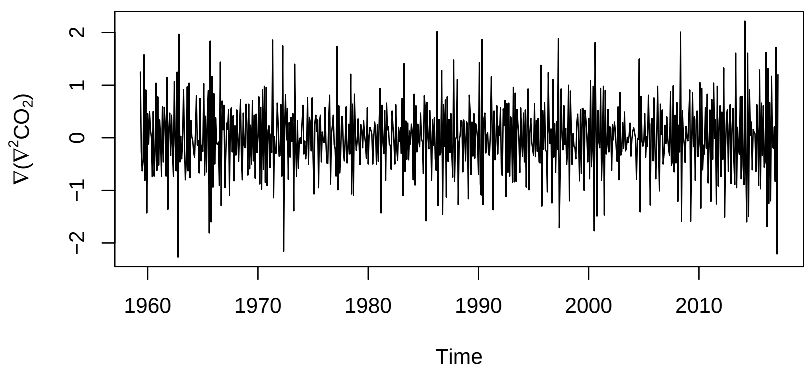


Figure 4.9: Time series of the lag-12 difference of the twice-differenced atmospheric CO_2 concentration at Mauna Loa, Hawai'i.

4.4 Correlation within and among time series

The concepts of covariance and correlation are very important in time series analysis. In particular, we can examine the correlation structure of the original data or random errors from a decomposition model to help us identify possible form(s) of (non)stationary model(s) for the stochastic process.

4.4.1 Autocorrelation function (ACF)

Autocorrelation is the correlation of a variable with itself at differing time lags. Recall from lecture that we defined the sample autocovariance function (ACVF), c_k, for some lag k as

$$c_k = \frac{1}{n} \sum_{t=1}^{n-k} \left(x_t - \bar{x}\right)\left(x_{t+k} - \bar{x}\right) \tag{4.10}$$

Note that the sample autocovariance of $\{x_t\}$ at lag 0, c_0, equals the sample variance of $\{x_t\}$ calculated with a denominator of n. The sample autocorrelation function (ACF) is defined as

$$r_k = \frac{c_k}{c_0} = \mathrm{Cor}(x_t, x_{t+k}) \tag{4.11}$$

Recall also that an approximate 95% confidence interval on the ACF can be estimated by

$$-\frac{1}{n} \pm \frac{2}{\sqrt{n}} \tag{4.12}$$

where n is the number of data points used in the calculation of the ACF.

It is important to remember two things here. First, although the confidence interval is commonly plotted and interpreted as a horizontal line over all time lags, the interval itself actually grows as the lag increases because the number of data points n used to estimate the correlation decreases by 1 for every integer increase in lag. Second, care must be exercised when interpreting the "significance" of the correlation at various lags because we should expect, *a priori*, that approximately 1 out of every 20 correlations will be significant based on chance alone.

We can use the `acf()` function in R to compute the sample ACF (note that adding the option `type = "covariance"` will return the sample auto-covariance (ACVF) instead of the ACF–type `?acf` for details). Calling the function by itself will will automatically produce a correlogram (*i.e.*, a plot of the autocorrelation versus time lag). The argument `lag.max` allows you to set the number of positive and negative lags. Let's try it for the CO_2 data.

```
## correlogram of the CO2 data
acf(co2, lag.max = 36)
```

There are 4 things about Figure 4.10 that are noteworthy:

1. the ACF at lag 0, r_0, equals 1 by default (*i.e.*, the correlation of a time series with itself)–it's plotted as a reference point;
2. the x-axis has decimal values for lags, which is caused by R using the year index as the lag rather than the month;
3. the horizontal blue lines are the approximate 95% CI's; and
4. there is very high autocorrelation even out to lags of 36 months.

As an alternative to the default plots for **acf** objects, let's define a new plot function for **acf** objects with some better features:

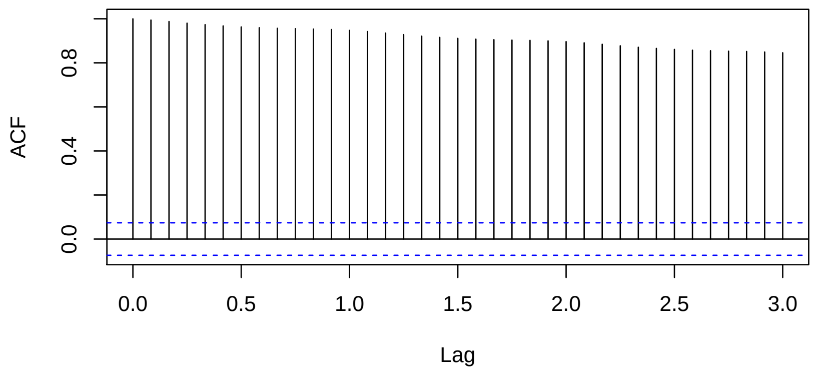
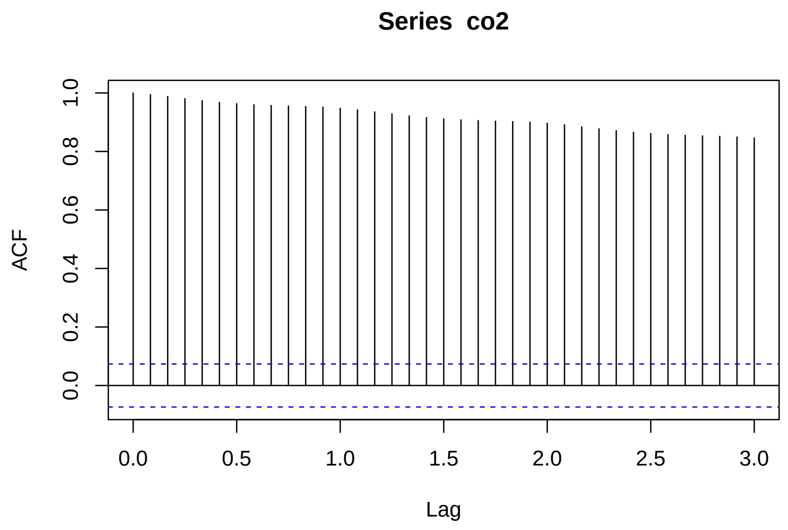


Figure 4.10: Correlogram of the observed atmospheric CO_2 concentration at Mauna Loa, Hawai'i obtained with the function `acf()`.

```
## better ACF plot
plot.acf <- function(ACFobj) {
    rr <- ACFobj$acf[-1]
    kk <- length(rr)
    nn <- ACFobj$n.used
    plot(seq(kk), rr, type = "h", lwd = 2, yaxs = "i", xaxs = "i",
        ylim = c(floor(min(rr)), 1), xlim = c(0, kk + 1), xlab = "Lag",
        ylab = "Correlation", las = 1)
    abline(h = -1/nn + c(-2, 2)/sqrt(nn), lty = "dashed", col = "blue")
    abline(h = 0)
}
```

Now we can assign the result of `acf()` to a variable and then use the information contained therein to plot the correlogram with our new plot function.

```
## acf of the CO2 data
co2_acf <- acf(co2, lag.max = 36)
## correlogram of the CO2 data
plot.acf(co2_acf)
```



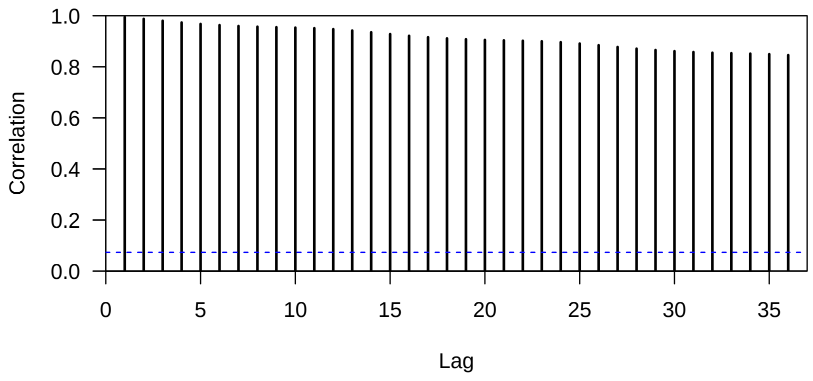
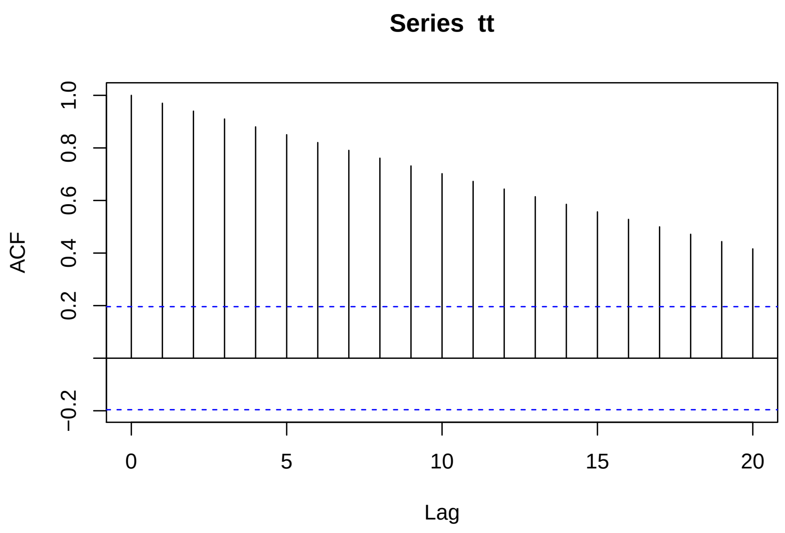


Figure 4.11: Correlogram of the observed atmospheric CO_2 concentration at Mauna Loa, Hawai'i obtained with the function `plot.acf()`.

Notice that all of the relevant information is still there (Figure 4.11), but now $r_0 = 1$ is not plotted at lag-0 and the lags on the x-axis are displayed correctly as integers.

Before we move on to the PACF, let's look at the ACF for some deterministic time series, which will help you identify interesting properties (*e.g.*, trends, seasonal effects) in a stochastic time series, and account for them in time series models–an important topic in this course. First, let's look at a straight line.

```
## length of ts
nn <- 100
## create straight line
tt <- seq(nn)
## set up plot area
par(mfrow = c(1, 2))
## plot line
plot.ts(tt, ylab = expression(italic(x[t])))
## get ACF
line.acf <- acf(tt, plot = FALSE)
## plot ACF
plot.acf(line.acf)
```



The correlogram for a straight line is itself a linearly decreasing function over time (Figure 4.12).

Now let's examine the ACF for a sine wave and see what sort of pattern arises.

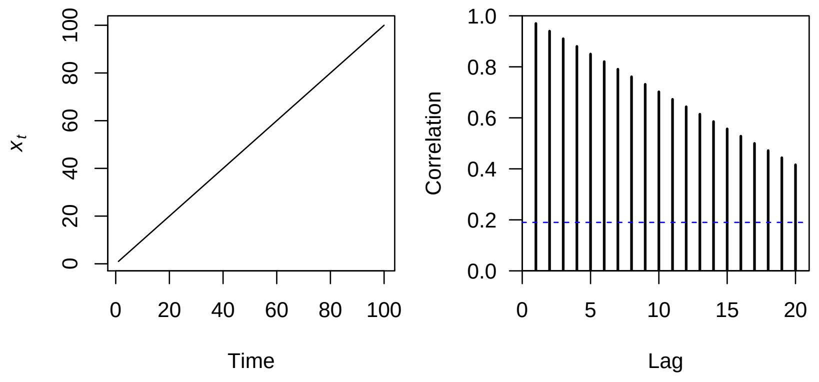


Figure 4.12: Time series plot of a straight line (left) and the correlogram of its ACF (right).

```r
## create sine wave
tt <- sin(2 * pi * seq(nn)/12)
## set up plot area
par(mfrow = c(1, 2))
## plot line
plot.ts(tt, ylab = expression(italic(x[t])))
## get ACF
sine_acf <- acf(tt, plot = FALSE)
## plot ACF
plot.acf(sine_acf)
```

Perhaps not surprisingly, the correlogram for a sine wave is itself a sine wave whose amplitude decreases linearly over time (Figure 4.13).

Now let's examine the ACF for a sine wave with a linear downward trend and see what sort of patterns arise.

```r
## create sine wave with trend
tt <- sin(2 * pi * seq(nn)/12) - seq(nn)/50
## set up plot area
par(mfrow = c(1, 2))
## plot line
plot.ts(tt, ylab = expression(italic(x[t])))
```

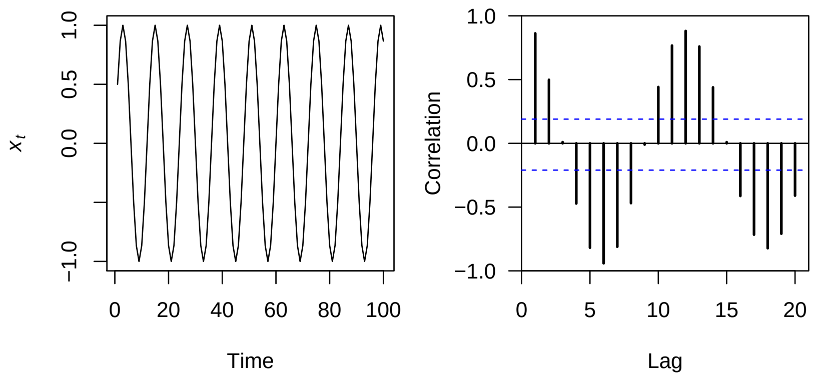


Figure 4.13: Time series plot of a discrete sine wave (left) and the correlogram of its ACF (right).

```
## get ACF
sili_acf <- acf(tt, plot = FALSE)
## plot ACF
plot.acf(sili_acf)
```

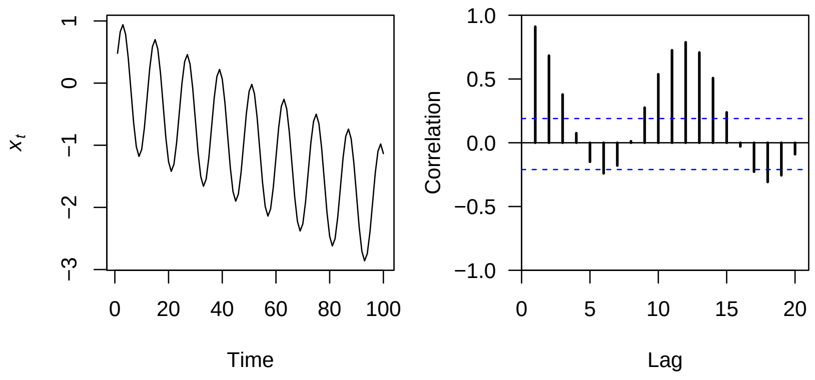


Figure 4.14: Time series plot of a discrete sine wave (left) and the correlogram of its ACF (right).

The correlogram for a sine wave with a trend is itself a nonsymmetrical sine wave whose amplitude and center decrease over time (Figure 4.14).

As we have seen, the ACF is a powerful tool in time series analysis for identifying important features in the data. As we will see later, the ACF is also an important diagnostic tool for helping to select the proper order of p and q in ARMA(p,q) models.

4.4.2 Partial autocorrelation function (PACF)

The partial autocorrelation function (PACF) measures the linear correlation of a series $\{x_t\}$ and a lagged version of itself $\{x_{t+k}\}$ with the linear dependence of $\{x_{t-1}, x_{t-2}, \dots, x_{t-(k-1)}\}$ removed. Recall from lecture that we define the PACF as

$$f_k = \begin{cases} \text{Cor}(x_1, x_0) = r_1 & \text{if } k = 1; \\ \text{Cor}(x_k - x_k^{k-1}, x_0 - x_0^{k-1}) & \text{if } k \geq 2; \end{cases} \tag{4.13}$$

with

$$x_k^{k-1} = \beta_1 x_{k-1} + \beta_2 x_{k-2} + \cdots + \beta_{k-1} x_1; \tag{4.14a}$$
$$x_0^{k-1} = \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_{k-1} x_{k-1}. \tag{4.14b}$$

It's easy to compute the PACF for a variable in R using the `pacf()` function, which will automatically plot a correlogram when called by itself (similar to `acf()`). Let's look at the PACF for the CO$_2$ data.

```r
## PACF of the CO2 data
pacf(co2, lag.max = 36)
```

The default plot for PACF is a bit better than for ACF, but here is another plotting function that might be useful.

```r
## better PACF plot
plot.pacf <- function(PACFobj) {
    rr <- PACFobj$acf
    kk <- length(rr)
    nn <- PACFobj$n.used
    plot(seq(kk), rr, type = "h", lwd = 2, yaxs = "i", xaxs = "i",
         ylim = c(floor(min(rr)), 1), xlim = c(0, kk + 1), xlab = "Lag",
         ylab = "PACF", las = 1)
    abline(h = -1/nn + c(-2, 2)/sqrt(nn), lty = "dashed", col = "blue")
    abline(h = 0)
}
```

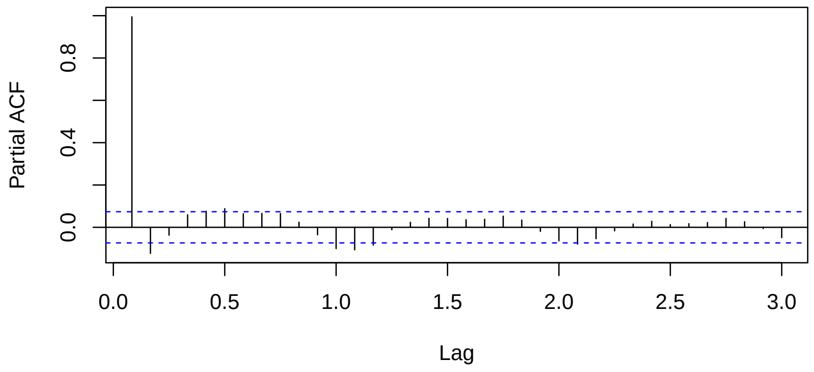


Figure 4.15: Correlogram of the PACF for the observed atmospheric CO_2 concentration at Mauna Loa, Hawai'i obtained with the function `pacf()`.

Notice in Figure 4.15 that the partial autocorrelation at lag-1 is very high (it equals the ACF at lag-1), but the other values at lags > 1 are relatively small, unlike what we saw for the ACF. We will discuss this in more detail later on in this lab.

Notice also that the PACF plot again has real-valued indices for the time lag, but it does not include any value for lag-0 because it is impossible to remove any intermediate autocorrelation between t and $t-k$ when $k = 0$, and therefore the PACF does not exist at lag-0. If you would like, you can use the `plot.acf()` function we defined above to plot the PACF estimates because `acf()` and `pacf()` produce identical list structures (results not shown here).

```
## PACF of the CO2 data
co2_pacf <- pacf(co2)
## correlogram of the CO2 data
plot.acf(co2_pacf)
```

As with the ACF, we will see later on how the PACF can also be used to help identify the appropriate order of p and q in ARMA(p,q) models.

4.4.3 Cross-correlation function (CCF)

Often we are interested in looking for relationships between 2 different time series. There are many ways to do this, but a simple method is via examination of their cross-covariance and cross-correlation.

We begin by defining the sample cross-covariance function (CCVF) in a manner similar to the ACVF, in that

$$g_k^{xy} = \frac{1}{n} \sum_{t=1}^{n-k} \left(y_t - \bar{y}\right) \left(x_{t+k} - \bar{x}\right), \tag{4.15}$$

but now we are estimating the correlation between a variable y and a *different* time-shifted variable x_{t+k}. The sample cross-correlation function (CCF) is then defined analogously to the ACF, such that

$$r_k^{xy} = \frac{g_k^{xy}}{\sqrt{\text{SD}_x \text{SD}_y}}; \tag{4.16}$$

SD_x and SD_y are the sample standard deviations of $\{x_t\}$ and $\{y_t\}$, respectively. It is important to re-iterate here that $r_k^{xy} \neq r_{-k}^{xy}$, but $r_k^{xy} = r_{-k}^{yx}$. Therefore, it is very important to pay particular attention to which variable you call y (*i.e.*, the "response") and which you call x (*i.e.*, the "predictor").

As with the ACF, an approximate 95% confidence interval on the CCF can be estimated by

$$-\frac{1}{n} \pm \frac{2}{\sqrt{n}} \tag{4.17}$$

where n is the number of data points used in the calculation of the CCF, and the same assumptions apply to its interpretation.

Computing the CCF in R is easy with the function `ccf()` and it works just like `acf()`. In fact, `ccf()` is just a "wrapper" function that calls `acf()`. As an example, let's examine the CCF between sunspot activity and number of lynx trapped in Canada as in the classic paper by Moran[1].

To begin, let's get the data, which are conveniently included in the **datasets** package included as part of the base installation of R. Before calculating the CCF, however, we need to find the matching years of data. Again, we'll use the `ts.intersect()` function.

```r
## get the matching years of sunspot data
suns <- ts.intersect(lynx, sunspot.year)[, "sunspot.year"]
## get the matching lynx data
lynx <- ts.intersect(lynx, sunspot.year)[, "lynx"]
```

Here are plots of the time series.

```r
## plot time series
plot(cbind(suns, lynx), yax.flip = TRUE)
```

It is important to remember which of the 2 variables you call y and x when calling `ccf(x, y, ...)`. In this case, it seems most relevant to treat lynx as the y and sunspots as the x, in which case we are mostly interested in the CCF at negative lags (*i.e.*, when sunspot activity predates inferred lynx abundance). Furthermore, we'll use log-transformed lynx trappings.

```r
## CCF of sunspots and lynx
ccf(suns, log(lynx), ylab = "Cross-correlation")
```

From Figures 4.16 and 4.17 it looks like lynx numbers are relatively low 3-5 years after high sunspot activity (*i.e.*, significant correlation at lags of -3 to -5).

4.5 White noise (WN)

A time series $\{w_t\}$ is a discrete white noise series (DWN) if the $w_1, w_1, \dots, w_t$ are independent and identically distributed (IID) with a mean of zero. For

[1]Moran, P.A.P. 1949. The statistical analysis of the sunspot and lynx cycles. *J. Anim. Ecol.* 18:115-116

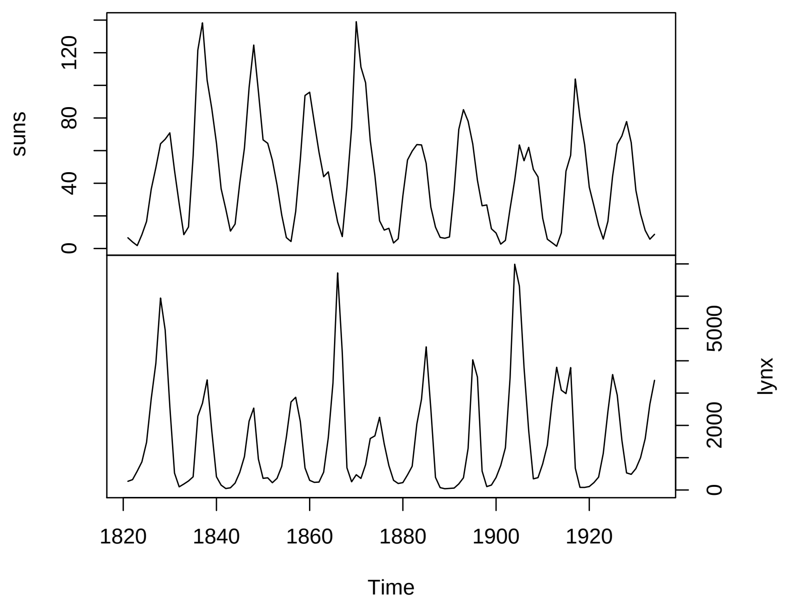


Figure 4.16: Time series of sunspot activity (top) and lynx trappings in Canada (bottom) from 1821-1934.

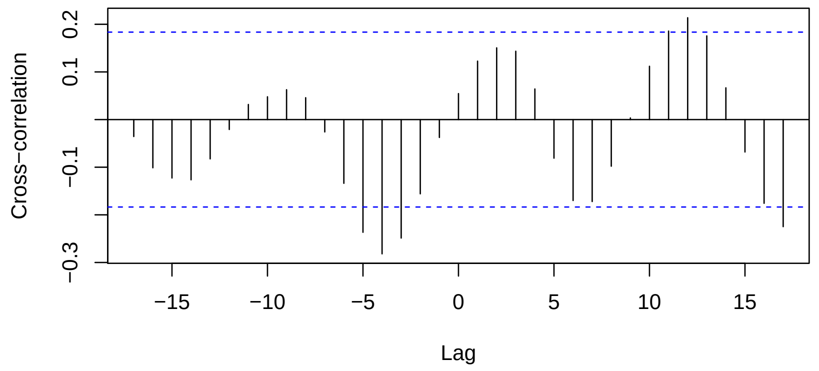


Figure 4.17: CCF for annual sunspot activity and the log of the number of lynx trappings in Canada from 1821-1934.

most of the examples in this course we will assume that the $w_t \sim \text{N}(0, q)$, and therefore we refer to the time series $\{w_t\}$ as Gaussian white noise. If our time series model has done an adequate job of removing all of the serial autocorrelation in the time series with trends, seasonal effects, etc., then the model residuals ($e_t = y_t - \hat{y}_t$) will be a WN sequence with the following properties for its mean ($\bar{e}$), covariance (c_k), and autocorrelation (r_k):

$$\begin{aligned}
\bar{x} &= 0 \\
c_k &= \text{Cov}(e_t, e_{t+k}) = \begin{cases} q & \text{if } k = 0 \\ 0 & \text{if } k \neq 1 \end{cases} \\
r_k &= \text{Cor}(e_t, e_{t+k}) = \begin{cases} 1 & \text{if } k = 0 \\ 0 & \text{if } k \neq 1. \end{cases}
\end{aligned} \tag{4.18}$$

4.5.1 Simulating white noise

Simulating WN in R is straightforward with a variety of built-in random number generators for continuous and discrete distributions. Once you know R's abbreviation for the distribution of interest, you add an `r` to the beginning to get the function's name. For example, a Gaussian (or normal) distribution is abbreviated `norm` and so the function is `rnorm()`. All of the random number functions require two things: the number of samples from the distribution and the parameters for the distribution itself (*e.g.*, mean & SD of a normal). Check the help file for the distribution of interest to find out what parameters you must specify (*e.g.*, type `?rnorm` to see the help for a normal distribution).

Here's how to generate 100 samples from a normal distribution with mean of 5 and standard deviation of 0.2, and 50 samples from a Poisson distribution with a rate (λ) of 20.

```r
set.seed(123)
## random normal variates
GWN <- rnorm(n = 100, mean = 5, sd = 0.2)
## random Poisson variates
PWN <- rpois(n = 50, lambda = 20)
```

Here are plots of the time series. Notice that on one occasion the same number was drawn twice in a row from the Poisson distribution, which is discrete. That is virtually guaranteed to never happen with a continuous distribution.

```r
## set up plot region
par(mfrow = c(1, 2))
## plot normal variates with mean
plot.ts(GWN)
abline(h = 5, col = "blue", lty = "dashed")
## plot Poisson variates with mean
plot.ts(PWN)
abline(h = 20, col = "blue", lty = "dashed")
```

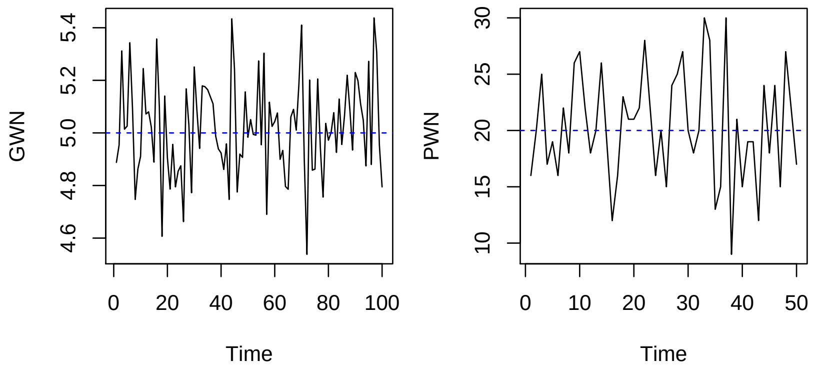


Figure 4.18: Time series plots of simulated Gaussian (left) and Poisson (right) white noise.

Now let's examine the ACF for the 2 white noise series and see if there is, in fact, zero autocorrelation for lags ≥ 1.

```r
## set up plot region
par(mfrow = c(1, 2))
## plot normal variates with mean
acf(GWN, main = "", lag.max = 20)
## plot Poisson variates with mean
acf(PWN, main = "", lag.max = 20)
```

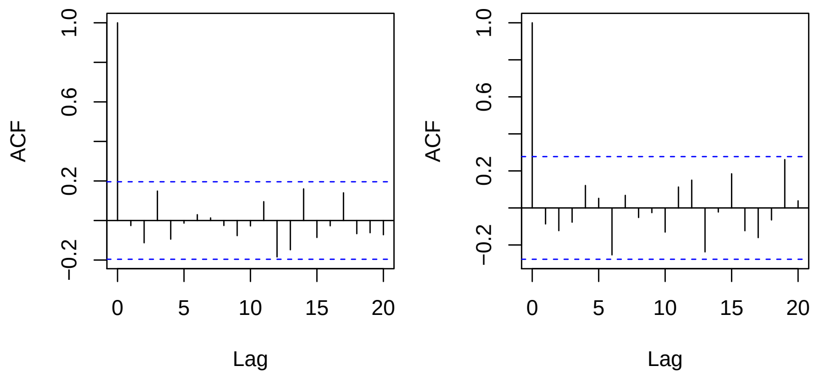


Figure 4.19: ACF's for the simulated Gaussian (left) and Poisson (right) white noise shown in Figure 4.18.

Interestingly, the r_k are all greater than zero in absolute value although they are not statistically different from zero for lags 1-20. This is because we are dealing with a *sample* of the distributions rather than the entire population of all random variates. As an exercise, try setting `n = 1e6` instead of `n = 100` or `n = 50` in the calls calls above to generate the WN sequences and see what effect it has on the estimation of r_k. It is also important to remember, as we discussed earlier, that we should expect that approximately 1 in 20 of the r_k will be statistically greater than zero based on chance alone, especially for relatively small sample sizes, so don't get too excited if you ever come across a case like then when inspecting model residuals.

4.6 Random walks (RW)

Random walks receive considerable attention in time series analyses because of their ability to fit a wide range of data despite their surprising simplicity. In fact, random walks are the most simple non-stationary time series model. A random walk is a time series $\{x_t\}$ where

$$x_t = x_{t-1} + w_t, \tag{4.19}$$

and w_t is a discrete white noise series where all values are independent and identically distributed (IID) with a mean of zero. In practice, we will almost always assume that the w_t are Gaussian white noise, such that $w_t \sim \text{N}(0, q)$. We will see later that a random walk is a special case of an autoregressive model.

4.6.1 Simulating a random walk

Simulating a RW model in R is straightforward with a for loop and the use of `rnorm()` to generate Gaussian errors (type `?rnorm` to see details on the function and its useful relatives `dnorm()` and `pnorm()`). Let's create 100 obs (we'll also set the random number seed so everyone gets the same results).

```
## set random number seed
set.seed(123)
## length of time series
TT <- 100
## initialize {x_t} and {w_t}
xx <- ww <- rnorm(n = TT, mean = 0, sd = 1)
## compute values 2 thru TT
for (t in 2:TT) {
    xx[t] <- xx[t - 1] + ww[t]
}
```

Now let's plot the simulated time series and its ACF.

```
## setup plot area
par(mfrow = c(1, 2))
## plot line
plot.ts(xx, ylab = expression(italic(x[t])))
## plot ACF
plot.acf(acf(xx, plot = FALSE))
```

Perhaps not surprisingly based on their names, autoregressive models such as RW's have a high degree of autocorrelation out to long lags (Figure 4.20).

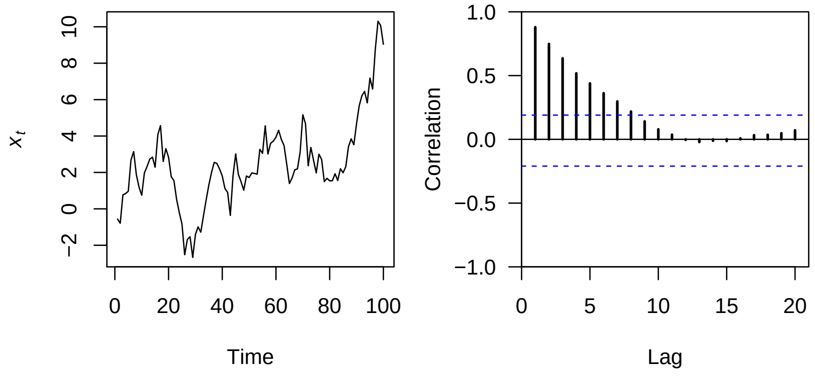


Figure 4.20: Simulated time series of a random walk model (left) and its associated ACF (right).

4.6.2 Alternative formulation of a random walk

As an aside, let's use an alternative formulation of a random walk model to see an even shorter way to simulate an RW in R. Based on our definition of a random walk in Equation (4.19), it is easy to see that

$$\begin{aligned}
x_t &= x_{t-1} + w_t \\
x_{t-1} &= x_{t-2} + w_{t-1} \\
x_{t-2} &= x_{t-3} + w_{t-2} \\
&\vdots
\end{aligned} \tag{4.20}$$

Therefore, if we substitute $x_{t-2} + w_{t-1}$ for x_{t-1} in the first equation, and then $x_{t-3} + w_{t-2}$ for x_{t-2}, and so on in a recursive manner, we get

$$x_t = w_t + w_{t-1} + w_{t-2} + \cdots + w_{t-\infty} + x_{t-\infty}. \tag{4.21}$$

In practice, however, the time series will not start an infinite time ago, but rather at some $t = 1$, in which case we can write

$$\begin{aligned}
x_t &= w_1 + w_2 + \cdots + w_t \\
&= \sum_{t=1}^{T} w_t.
\end{aligned} \tag{4.22}$$

From Equation (4.22) it is easy to see that the value of an RW process at time step t is the sum of all the random errors up through time t. Therefore, in R we can easily simulate a realization from an RW process using the `cumsum(x)` function, which does cumulative summation of the vector `x` over its entire length. If we use the same errors as before, we should get the same results.

```
## simulate RW
x2 <- cumsum(ww)
```

Let's plot both time series to see if it worked.

```
## setup plot area
par(mfrow = c(1, 2))
## plot 1st RW
plot.ts(xx, ylab = expression(italic(x[t])))
## plot 2nd RW
plot.ts(x2, ylab = expression(italic(x[t])))
```

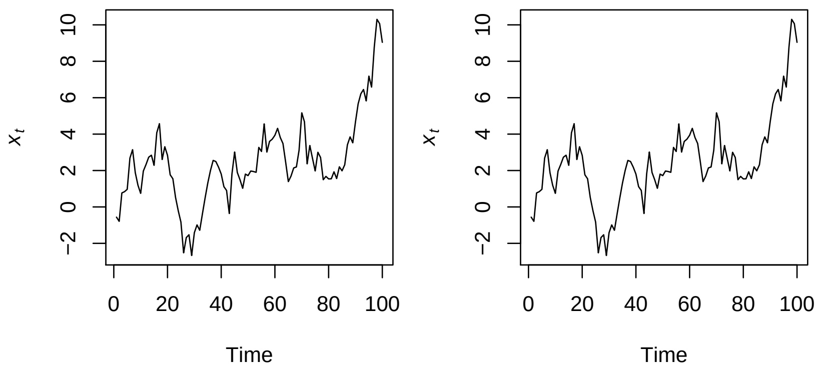


Figure 4.21: Time series of the same random walk model formulated as Equation (4.19) and simulated via a for loop (left), and as Equation (4.22) and simulated via `cumsum()` (right).

Indeed, both methods of generating a RW time series appear to be equivalent.

4.7 Autoregressive (AR) models

Autoregressive models of order p, abbreviated AR(p), are commonly used in time series analyses. In particular, AR(1) models (and their multivariate extensions) see considerable use in ecology as we will see later in the course. Recall from lecture that an AR(p) model is written as

$$(\#eq: defnAR.p.coef) x_t = \phi_1 x_{t-1} + \phi_2 x_{t-2} + \cdots + \phi_p x_{t-p} + w_t, \tag{4.23}$$

where $\{w_t\}$ is a white noise sequence with zero mean and some variance σ^2. For our purposes we usually assume that $w_t \sim \mathrm{N}(0, q)$. Note that the random walk in Equation (4.19) is a special case of an AR(1) model where $\phi_1 = 1$ and $\phi_k = 0$ for $k \geq 2$.

4.7.1 Simulating an AR(p) process

Although we could simulate an AR(p) process in R using a for loop just as we did for a random walk, it's much easier with the function `arima.sim()`, which works for all forms and subsets of ARIMA models. To do so, remember that the AR in ARIMA stands for "autoregressive", the I for "integrated", and the MA for "moving-average"; we specify the order of ARIMA models as p, d, q. So, for example, we would specify an AR(2) model as ARIMA(2,0,0), or an MA(1) model as ARIMA(0,0,1). If we had an ARMA(3,1) model that we applied to data that had been twice-differenced, then we would have an ARIMA(3,2,1) model.

`arima.sim()` will accept many arguments, but we are interested primarily in three of them (type `?arima.sim` to learn more):

1. `n`: the length of desired time series
2. `model`: a list with the following elements:

- `order`: a vector of length 3 containing the ARIMA(p, d, q) order
- `ar`: a vector of length p containing the AR(p) coefficients
- `ma`: a vector of length q containing the MA(q) coefficients

3. `sd`: the standard deviation of the Gaussian errors

Note that you can omit the `ma` element entirely if you have an AR(p) model, or omit the `ar` element if you have an MA(q) model. If you omit the `sd` element, `arima.sim()` will assume you want normally distributed errors with SD = 1. Also note that you can pass `arima.sim()` your own time series of random errors or the name of a function that will generate the errors (*e.g.*, you could use `rpois()` if you wanted a model with Poisson errors). Type `?arima.sim` for more details.

Let's begin by simulating some AR(1) models and comparing their behavior. First, let's choose models with contrasting AR coefficients. Recall that in order for an AR(1) model to be stationary, $\phi < |1|$, so we'll try 0.1 and 0.9. We'll again set the random number seed so we will get the same answers.

```
set.seed(456)
## list description for AR(1) model with small coef
AR_sm <- list(order = c(1, 0, 0), ar = 0.1)
## list description for AR(1) model with large coef
AR_lg <- list(order = c(1, 0, 0), ar = 0.9)
## simulate AR(1)
AR1_sm <- arima.sim(n = 50, model = AR_sm, sd = 0.1)
AR1_lg <- arima.sim(n = 50, model = AR_lg, sd = 0.1)
```

Now let's plot the 2 simulated series.

```
## setup plot region
par(mfrow = c(1, 2))
## get y-limits for common plots
ylm <- c(min(AR1_sm, AR1_lg), max(AR1_sm, AR1_lg))
## plot the ts
plot.ts(AR1_sm, ylim = ylm, ylab = expression(italic(x)[italic(t)]),
    main = expression(paste(phi, " = 0.1")))
plot.ts(AR1_lg, ylim = ylm, ylab = expression(italic(x)[italic(t)]),
    main = expression(paste(phi, " = 0.9")))
```

What do you notice about the two plots in Figure 4.22? It looks like the time series with the smaller AR coefficient is more "choppy" and seems to stay

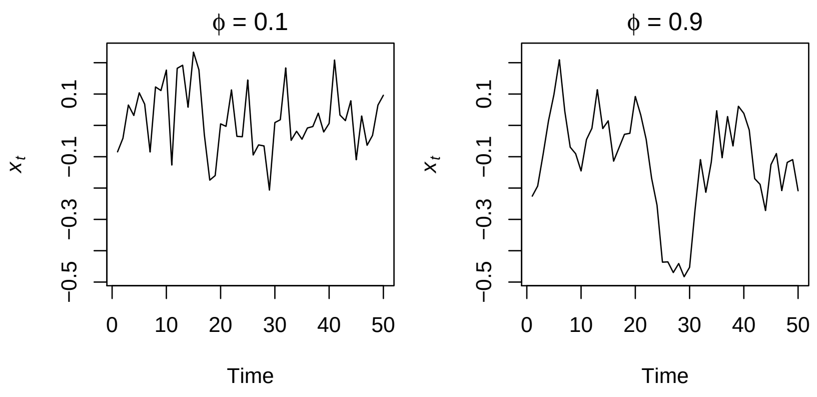


Figure 4.22: Time series of simulated AR(1) processes with $\phi = 0.1$ (left) and $\phi = 0.9$ (right).

closer to 0 whereas the time series with the larger AR coefficient appears to wander around more. Remember that as the coefficient in an AR(1) model goes to 0, the model approaches a WN sequence, which is stationary in both the mean and variance. As the coefficient goes to 1, however, the model approaches a random walk, which is not stationary in either the mean or variance.

Next, let's generate two AR(1) models that have the same magnitude coeficient, but opposite signs, and compare their behavior.

```
set.seed(123)
## list description for AR(1) model with small coef
AR_pos <- list(order = c(1, 0, 0), ar = 0.5)
## list description for AR(1) model with large coef
AR_neg <- list(order = c(1, 0, 0), ar = -0.5)
## simulate AR(1)
AR1_pos <- arima.sim(n = 50, model = AR_pos, sd = 0.1)
AR1_neg <- arima.sim(n = 50, model = AR_neg, sd = 0.1)
```

OK, let's plot the 2 simulated series.

```
## setup plot region
par(mfrow = c(1, 2))
## get y-limits for common plots
```

```r
ylm <- c(min(AR1_pos, AR1_neg), max(AR1_pos, AR1_neg))
## plot the ts
plot.ts(AR1_pos, ylim = ylm, ylab = expression(italic(x)[italic(t)]),
    main = expression(paste(phi[1], " = 0.5")))
plot.ts(AR1_neg, ylab = expression(italic(x)[italic(t)]), main = expression(paste(phi
    " = -0.5")))
```

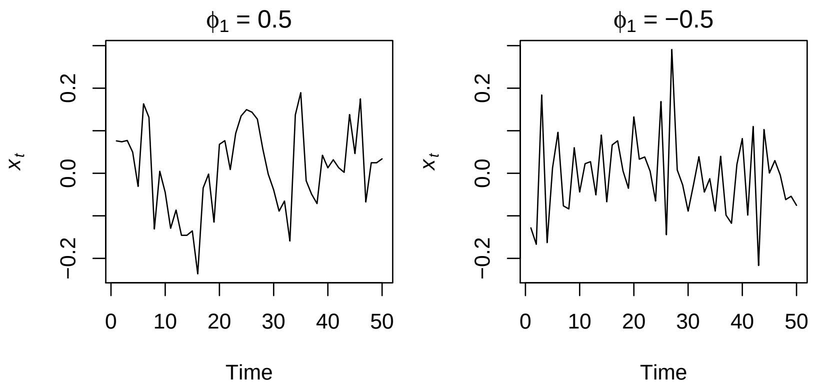


Figure 4.23: Time series of simulated AR(1) processes with $\phi_1 = 0.5$ (left) and $\phi_1 = -0.5$ (right).

Now it appears like both time series vary around the mean by about the same amount, but the model with the negative coefficient produces a much more "sawtooth" time series. It turns out that any AR(1) model with $-1 < \phi < 0$ will exhibit the 2-point oscillation you see here.

We can simulate higher order AR(p) models in the same manner, but care must be exercised when choosing a set of coefficients that result in a stationary model or else `arima.sim()` will fail and report an error. For example, an AR(2) model with both coefficients equal to 0.5 is not stationary, and therefore this function call will not work:

```r
arima.sim(n = 100, model = list(order(2, 0, 0), ar = c(0.5, 0.5)))
```

If you try, R will respond that the "`'ar' part of model is not stationary`".

4.7.2 Correlation structure of AR(p) processes

Let's review what we learned in lecture about the general behavior of the ACF and PACF for AR(p) models. To do so, we'll simulate four stationary AR(p) models of increasing order p and then examine their ACF's and PACF's. Let's use a really big n so as to make them "pure", which will provide a much better estimate of the correlation structure.

```r
set.seed(123)
## the 4 AR coefficients
AR_p_coef <- c(0.7, 0.2, -0.1, -0.3)
## empty list for storing models
AR_mods <- list()
## loop over orders of p
for (p in 1:4) {
    ## assume sd = 1, so not specified
    AR_mods[[p]] <- arima.sim(n = 10000, list(ar = AR_p_coef[1:p]))
}
```

Now that we have our four AR(p) models, lets look at plots of the time series, ACF's, and PACF's.

```r
## set up plot region
par(mfrow = c(4, 3))
## loop over orders of p
for (p in 1:4) {
    plot.ts(AR_mods[[p]][1:50], ylab = paste("AR(", p, ")", sep = ""))
    acf(AR_mods[[p]], lag.max = 12)
    pacf(AR_mods[[p]], lag.max = 12, ylab = "PACF")
}
```

As we saw in lecture and is evident from our examples shown in Figure 4.24, the ACF for an AR(p) process tails off toward zero very slowly, but the PACF goes to zero for lags $> p$. This is an important diagnostic tool when trying to identify the order of p in ARMA(p, q) models.

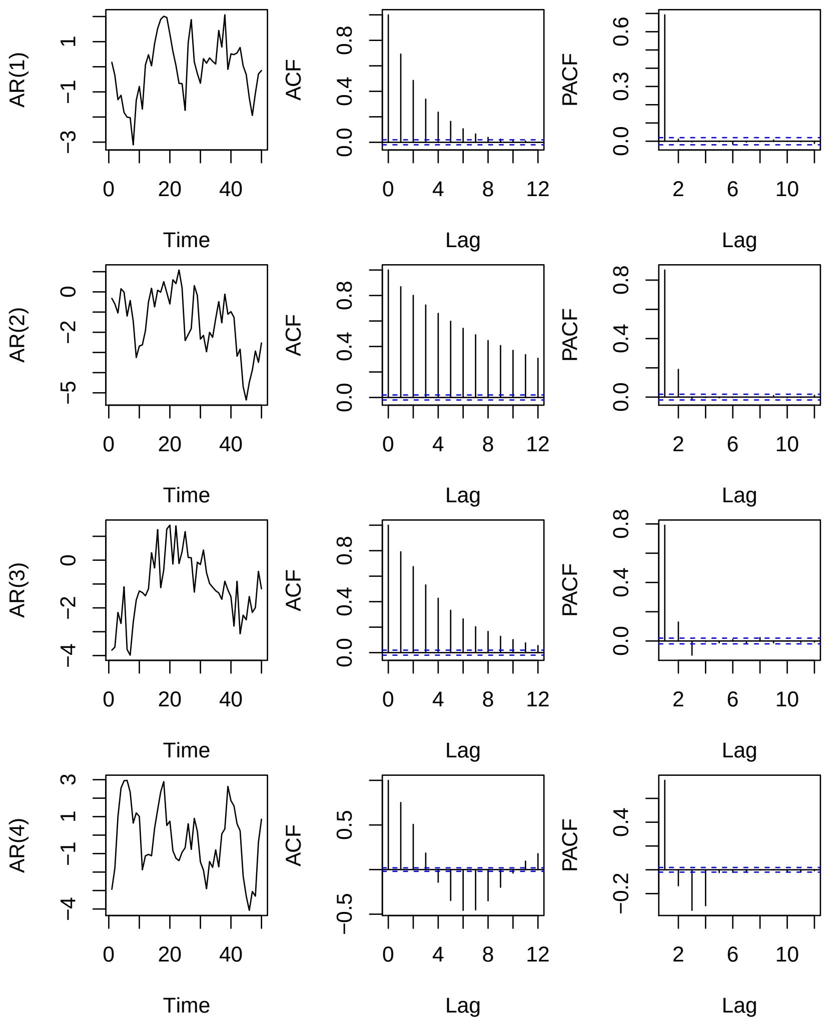


Figure 4.24: Time series of simulated AR(p) processes (left column) of increasing orders from 1-4 (rows) with their associated ACF's (center column) and PACF's (right column). Note that only the first 50 values of x_t are plotted.

4.8 Moving-average (MA) models

A moving-averge process of order q, or MA(q), is a weighted sum of the current random error plus the q most recent errors, and can be written as

$$x_t = w_t + \theta_1 w_{t-1} + \theta_2 w_{t-2} + \dots + \theta_q w_{t-q}, \tag{4.24}$$

where $\{w_t\}$ is a white noise sequence with zero mean and some variance σ^2; for our purposes we usually assume that $w_t \sim \text{N}(0, q)$. Of particular note is that because MA processes are finite sums of stationary errors, they themselves are stationary.

Of interest to us are so-called "invertible" MA processes that can be expressed as an infinite AR process with no error term. The term invertible comes from the inversion of the backshift operator ($\mathbf{B}$) that we discussed in class (*i.e.*, $\mathbf{B}x_t = x_{t-1}$). So, for example, an MA(1) process with $\theta < |1|$ is invertible because it can be written using the backshift operator as

$$\begin{aligned}
x_t &= w_t - \theta w_{t-1} \\
x_t &= w_t - \theta \mathbf{B} w_t \\
x_t &= (1 - \theta \mathbf{B}) w_t, \\
&\Downarrow \\
w_t &= \frac{1}{(1 - \theta \mathbf{B})} x_t \\
w_t &= (1 + \theta \mathbf{B} + \theta^2 \mathbf{B}^2 + \theta^3 \mathbf{B}^3 + \dots) x_t \\
w_t &= x_t + \theta x_{t-1} + \theta^2 x_{t-2} + \theta^3 x_{t-3} + \dots
\end{aligned} \tag{4.25}$$

4.8.1 Simulating an MA(q) process

We can simulate MA(q) processes just as we did for AR(p) processes using `arima.sim()`. Here are 3 different ones with contrasting θ's:

```
set.seed(123)
## list description for MA(1) model with small coef
MA_sm <- list(order = c(0, 0, 1), ma = 0.2)
```

```
## list description for MA(1) model with large coef
MA_lg <- list(order = c(0, 0, 1), ma = 0.8)
## list description for MA(1) model with large coef
MA_neg <- list(order = c(0, 0, 1), ma = -0.5)
## simulate MA(1)
MA1_sm <- arima.sim(n = 50, model = MA_sm, sd = 0.1)
MA1_lg <- arima.sim(n = 50, model = MA_lg, sd = 0.1)
MA1_neg <- arima.sim(n = 50, model = MA_neg, sd = 0.1)
```

with their associated plots.

```
## setup plot region
par(mfrow = c(1, 3))
## plot the ts
plot.ts(MA1_sm, ylab = expression(italic(x)[italic(t)]), main = expression(paste(thet
    " = 0.2")))
plot.ts(MA1_lg, ylab = expression(italic(x)[italic(t)]), main = expression(paste(thet
    " = 0.8")))
plot.ts(MA1_neg, ylab = expression(italic(x)[italic(t)]), main = expression(paste(the
    " = -0.5")))
```

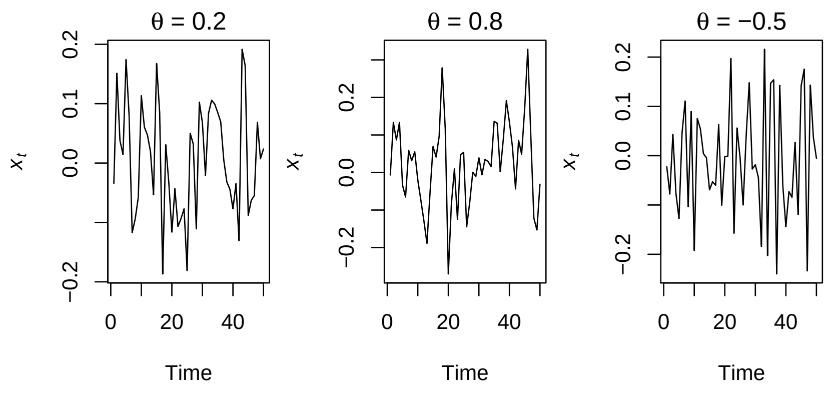


Figure 4.25: Time series of simulated MA(1) processes with $\theta = 0.2$ (left), $\theta = 0.8$ (middle), and $\theta = -0.5$ (right).

In contrast to AR(1) processes, MA(1) models do not exhibit radically different behavior with changing θ. This should not be too surprising given that they are simply linear combinations of white noise.

4.8.2 Correlation structure of MA(q) processes

We saw in lecture and above how the ACF and PACF have distinctive features for AR(p) models, and they do for MA(q) models as well. Here are examples of four MA(q) processes. As before, we'll use a really big n so as to make them "pure", which will provide a much better estimate of the correlation structure.

```r
set.seed(123)
## the 4 MA coefficients
MA_q_coef <- c(0.7, 0.2, -0.1, -0.3)
## empty list for storing models
MA_mods <- list()
## loop over orders of q
for (q in 1:4) {
    ## assume sd = 1, so not specified
    MA_mods[[q]] <- arima.sim(n = 1000, list(ma = MA_q_coef[1:q]))
}
```

Now that we have our four MA(q) models, lets look at plots of the time series, ACF's, and PACF's.

```r
## set up plot region
par(mfrow = c(4, 3))
## loop over orders of q
for (q in 1:4) {
    plot.ts(MA_mods[[q]][1:50], ylab = paste("MA(", q, ")", sep = ""))
    acf(MA_mods[[q]], lag.max = 12)
    pacf(MA_mods[[q]], lag.max = 12, ylab = "PACF")
}
```

Note very little qualitative difference in the realizations of the four MA(q) processes (Figure 4.26). As we saw in lecture and is evident from our examples here, however, the ACF for an MA(q) process goes to zero for lags $> q$, but the PACF tails off toward zero very slowly. This is an important diagnostic tool when trying to identify the order of q in ARMA(p, q) models.

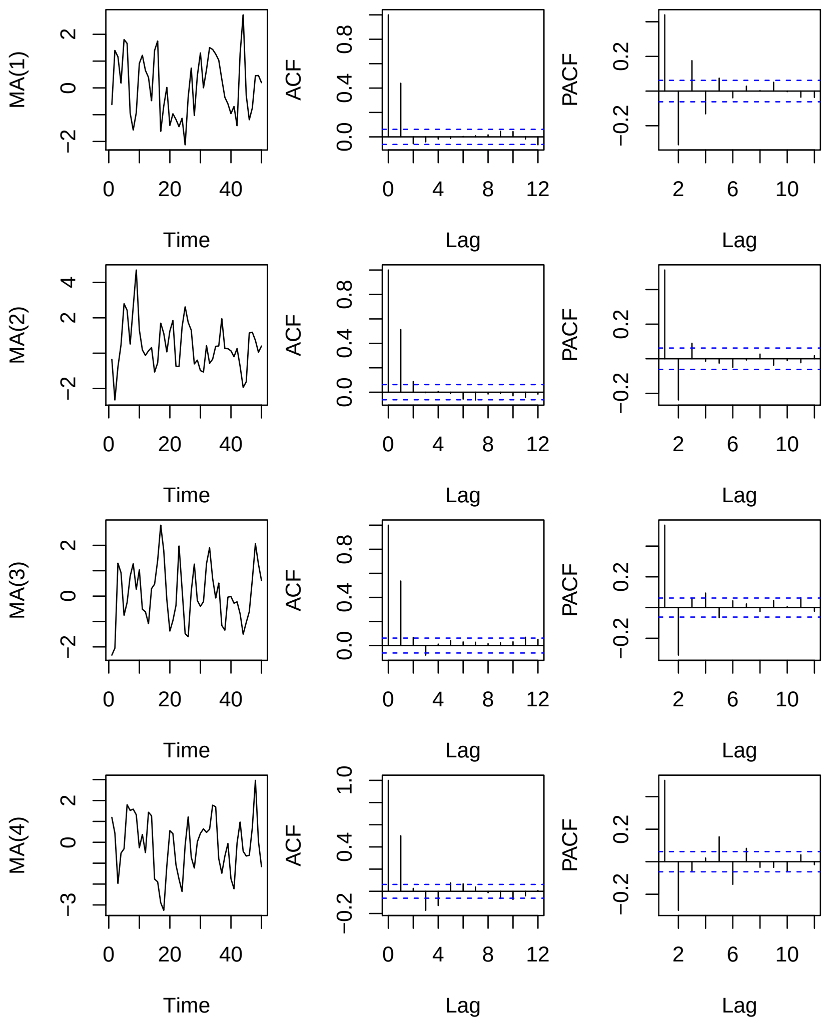


Figure 4.26: Time series of simulated MA(q) processes (left column) of increasing orders from 1-4 (rows) with their associated ACF's (center column) and PACF's (right column). Note that only the first 50 values of x_t are plotted.

4.9 Autoregressive moving-average (ARMA) models

ARMA(p, q) models have a rich history in the time series literature, but they are not nearly as common in ecology as plain AR(p) models. As we discussed in lecture, both the ACF and PACF are important tools when trying to identify the appropriate order of p and q. Here we will see how to simulate time series from AR(p), MA(q), and ARMA(p, q) processes, as well as fit time series models to data based on insights gathered from the ACF and PACF.

We can write an ARMA(p, q) as a mixture of AR(p) and MA(q) models, such that

$$x_t = \phi_1 x_{t-1} + \phi_2 x_{t-2} + \cdots + \phi_p x_{t-p} + w_t + \theta_1 w_{t-1} + \theta_2 w_{t-2} + \cdots + \theta_q w_{t-q}, \quad (4.26)$$

and the w_t are white noise.

4.9.1 Fitting ARMA(p, q) models with `arima()`

We have already seen how to simulate AR(p) and MA(q) models with `arima.sim()`; the same concepts apply to ARMA(p, q) models and therefore we will not do that here. Instead, we will move on to fitting ARMA(p, q) models when we only have a realization of the process (*i.e.*, data) and do not know the underlying parameters that generated it.

The function `arima()` accepts a number of arguments, but two of them are most important:

- `x`: a univariate time series
- `order`: a vector of length 3 specifying the order of ARIMA(p,d,q) model

In addition, note that by default `arima()` will estimate an underlying mean of the time series unless $d > 0$. For example, an AR(1) process with mean μ would be written

$$x_t = \mu + \phi(x_{t-1} - \mu) + w_t. \tag{4.27}$$

If you know for a fact that the time series data have a mean of zero (*e.g.*, you already subtracted the mean from them), you should include the argument `include.mean = FALSE`, which is set to `TRUE` by default. Note that ignoring and not estimating a mean in ARMA(p, q) models when one exists will bias the estimates of all other parameters.

Let's see an example of how `arima()` works. First we'll simulate an ARMA(2,2) model and then estimate the parameters to see how well we can recover them. In addition, we'll add in a constant to create a non-zero mean, which `arima()` reports as `intercept` in its output.

```r
set.seed(123)
## ARMA(2,2) description for arim.sim()
ARMA22 <- list(order = c(2, 0, 2), ar = c(-0.7, 0.2), ma = c(0.7,
    0.2))
## mean of process
mu <- 5
## simulated process (+ mean)
ARMA_sim <- arima.sim(n = 10000, model = ARMA22) + mu
## estimate parameters
arima(x = ARMA_sim, order = c(2, 0, 2))
```

```

Call:
arima(x = ARMA_sim, order = c(2, 0, 2))

Coefficients:
          ar1     ar2     ma1     ma2  intercept
      -0.7079  0.1924  0.6912  0.2001     4.9975
s.e.   0.0291  0.0284  0.0289  0.0236     0.0125

sigma^2 estimated as 0.9972:  log likelihood = -14175.92,  aic = 28363.84
```

It looks like we were pretty good at estimating the true parameters, but our sample size was admittedly quite large; the estimate of the variance of the process errors is reported as `sigma^2` below the other coefficients. As an exercise, try decreasing the length of time series in the `arima.sim()` call above from 10,000 to something like 100 and see what effect it has on the parameter estimates.

4.9.2 Searching over model orders

In an ideal situation, you could examine the ACF and PACF of the time series of interest and immediately decipher what orders of p and q must have generated the data, but that doesn't always work in practice. Instead, we are often left with the task of searching over several possible model forms and seeing which of them provides the most parsimonious fit to the data. There are two easy ways to do this for ARIMA models in R. The first is to write a little script that loops ove the possible dimensions of p and q. Let's try that for the process we simulated above and search over orders of p and q from 0-3 (it will take a few moments to run and will likely report an error about a "`possible convergence problem`", which you can ignore).

```
## empty list to store model fits
ARMA_res <- list()
## set counter
cc <- 1
## loop over AR
for (p in 0:3) {
    ## loop over MA
    for (q in 0:3) {
        ARMA_res[[cc]] <- arima(x = ARMA_sim, order = c(p, 0,
            q))
        cc <- cc + 1
    }
}
```

```
Warning in arima(x = ARMA_sim, order = c(p, 0, q)): possible convergence probl
= 1
```

```
## get AIC values for model evaluation
ARMA_AIC <- sapply(ARMA_res, function(x) x$aic)
```

```
## model with lowest AIC is the best
ARMA_res[[which(ARMA_AIC == min(ARMA_AIC))]]
```

```

Call:
arima(x = ARMA_sim, order = c(p, 0, q))

Coefficients:
          ar1     ar2     ma1     ma2  intercept
      -0.7079  0.1924  0.6912  0.2001     4.9975
s.e.   0.0291  0.0284  0.0289  0.0236     0.0125

sigma^2 estimated as 0.9972:  log likelihood = -14175.92,  aic = 28363.84
```

It looks like our search worked, so let's look at the other method for fitting ARIMA models. The `auto.arima()` function in the **forecast** package will conduct an automatic search over all possible orders of ARIMA models that you specify. For details, type `?auto.arima` after loading the package. Let's repeat our search using the same criteria.

```
## find best ARMA(p,q) model
auto.arima(ARMA_sim, start.p = 0, max.p = 3, start.q = 0, max.q = 3)
```

```

Series: ARMA_sim
ARIMA(2,0,2) with non-zero mean

Coefficients:
          ar1     ar2     ma1     ma2    mean
      -0.7079  0.1924  0.6912  0.2001  4.9975
s.e.   0.0291  0.0284  0.0289  0.0236  0.0125

sigma^2 estimated as 0.9977:  log likelihood=-14175.92
AIC=28363.84   AICc=28363.84   BIC=28407.1
```

We get the same results with an increase in speed and less coding, which is nice. If you want to see the form for each of the models checked by `auto.arima()` and their associated AIC values, include the argument `trace = 1`.

4.10 Problems

We have seen how to do a variety of introductory time series analyses with R. Now it is your turn to apply the information you learned here and in lecture to complete some analyses. You have been asked by a colleague to help analyze some time series data she collected as part of an experiment on the effects of light and nutrients on the population dynamics of phytoplankton. Specifically, after controlling for differences in light and temperature, she wants to know if the natural log of population density can be modeled with some form of ARMA(p, q) model.

The data are expressed as the number of cells per milliliter recorded every hour for one week beginning at 8:00 AM on December 1, 2014. You can load the data using

```
data(hourlyphyto, package = "atsalibrary")
phyto_dat <- hourlyphyto
```

Use the information above to do the following:

1. Convert `phyto_dat`, which is a **data.frame** object, into a **ts** object. This bit of code might be useful to get you started:

```
## what day of 2014 is Dec 1st?
date_begin <- as.Date("2014-12-01")
day_of_year <- (date_begin - as.Date("2014-01-01") + 1)
```

2. Plot the time series of phytoplankton density and provide a brief description of any notable features.

3. Although you do not have the actual measurements for the specific temperature and light regimes used in the experiment, you have been informed that they follow a regular light/dark period with accompanying warm/cool temperatures. Thus, estimating a fixed seasonal effect is justifiable. Also, the instrumentation is precise enough to preclude any systematic change in measurements over time (*i.e.*, you can assume $m_t = 0$ for all t). Obtain the time series of the estimated log-density of phytoplankton absent any hourly effects caused by variation in temperature or light. (*Hint*: You will need to do some decomposition.)

4. Use diagnostic tools to identify the possible order(s) of ARMA model(s) that most likely describes the log of population density for this particular experiment. Note that at this point you should be focusing your analysis on the results obtained in Question 3.

5. Use some form of search to identify what form of ARMA(p, q) model best describes the log of population density for this particular experiment. Use what you learned in Question 4 to inform possible orders of p and q. (*Hint*: if you use `auto.arima()`, include the additional argument `seasonal = FALSE`)

6. Write out the best model in the form of Equation (4.26) using the underscore notation to refer to subscripts (*e.g.*, write `x_t` for x_t). You can round any parameters/coefficients to the nearest hundreth. (*Hint*: if the mean of the time series is not zero, refer to Eqn 1.27 in the lab handout).

Chapter 5

Box-Jenkins method

In this chapter, you will practice selecting and fitting an ARIMA model to catch data using the Box-Jenkins method. After fitting a model, you will prepare simple forecasts using the **forecast** package.

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

Data and packages

We will use the catch landings from Greek waters (`greeklandings`) and the Chinook landings (`chinook`) in Washington data sets for this chapter. These datasets are in the **atsalibrary** package on GitHub. Install using the **devtools** package.

```r
library(devtools)
# Windows users will likely need to set this
# Sys.setenv('R_REMOTES_NO_ERRORS_FROM_WARNINGS' = 'true')
devtools::install_github("nwfsc-timeseries/atsalibrary")
```

Load the data.

```r
data(greeklandings, package = "atsalibrary")
landings <- greeklandings
```

```r
# Use the monthly data
data(chinook, package = "atsalibrary")
chinook <- chinook.month
```

Ensure you have the necessary packages.

```r
library(ggplot2)
library(gridExtra)
library(reshape2)
library(tseries)
```

```

    'tseries' version: 0.10-48

    'tseries' is a package for time series analysis and computational finance.

    See 'library(help="tseries")' for details.
```

```r
library(urca)
library(forecast)
```

5.1 Box-Jenkins method

A. Model form selection

1. Evaluate stationarity
2. Selection of the differencing level (d) – to fix stationarity problems
3. Selection of the AR level (p)
4. Selection of the MA level (q)

B. Parameter estimation

C. Model checking

5.2 Stationarity

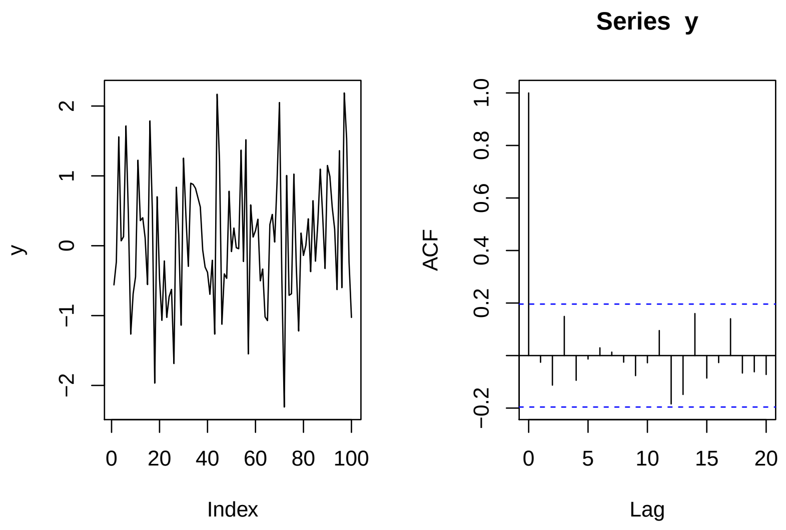
It is important to test and transform (via differencing) your data to ensure stationarity when fitting an ARMA model using standard algorithms. The standard algorithms for ARIMA models assume stationarity and we will be using those algorithms. It possible to fit ARMA models without transforming the data. We will cover that in later chapters. However, that is not commonly done in the literature on forecasting with ARMA models, certainly not in the literature on catch forecasting.

Keep in mind also that many ARMA models are stationary and you do not want to get in the situation of trying to fit an incompatible process model to your data. We will see examples of this when we start fitting models to non-stationary data and random walks.

5.2.1 Look at stationarity in simulated data

We will start by looking at white noise and a stationary AR(1) process from simulated data. White noise is simply a string of random numbers drawn from a Normal distribution. `rnorm()` with return random numbers drawn from a Normal distribution. Use `?rnorm` to understand what the function requires.

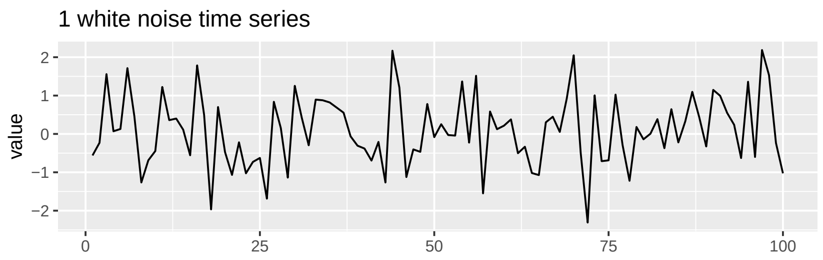
```r
TT <- 100
y <- rnorm(TT, mean = 0, sd = 1)  # 100 random numbers
op <- par(mfrow = c(1, 2))
plot(y, type = "l")
acf(y)
```

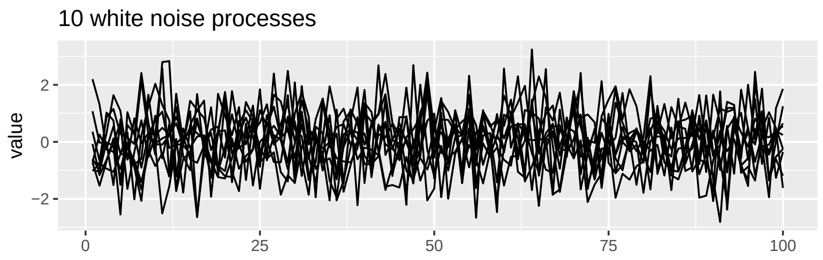


```
par(op)
```

Here we use `ggplot()` to plot 10 white noise time series.

```
dat <- data.frame(t = 1:TT, y = y)
p1 <- ggplot(dat, aes(x = t, y = y)) + geom_line() + ggtitle("1 white noise ti
    xlab("") + ylab("value")
ys <- matrix(rnorm(TT * 10), TT, 10)
ys <- data.frame(ys)
ys$id = 1:TT

ys2 <- melt(ys, id.var = "id")
p2 <- ggplot(ys2, aes(x = id, y = value, group = variable)) +
    geom_line() + xlab("") + ylab("value") + ggtitle("10 white noise processes
grid.arrange(p1, p2, ncol = 1)
```

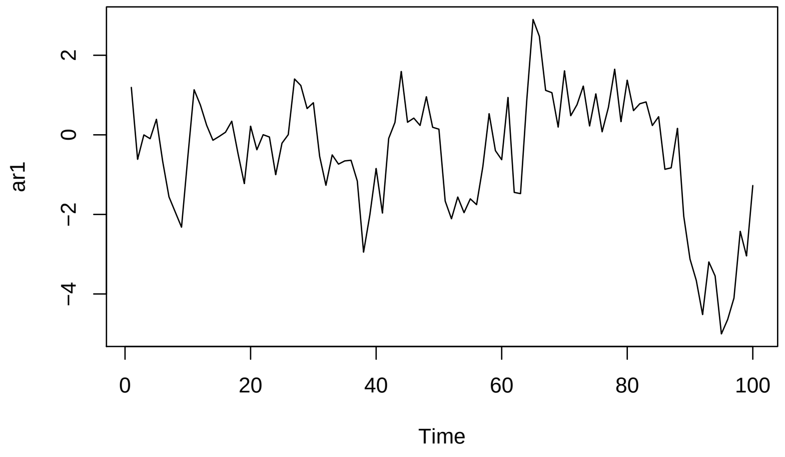




These are stationary because the variance and mean (level) does not change with time.

An AR(1) process is also stationary.

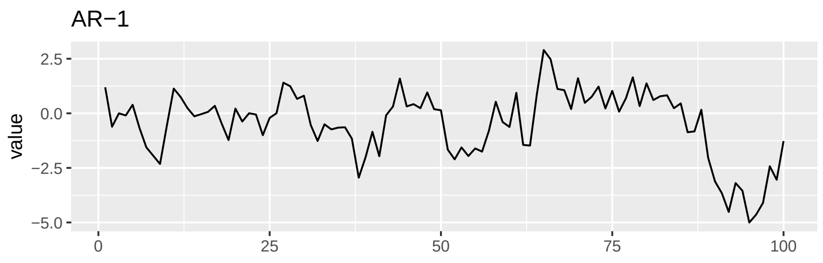
```r
theta <- 0.8
nsim <- 10
ar1 <- arima.sim(TT, model = list(ar = theta))
plot(ar1)
```

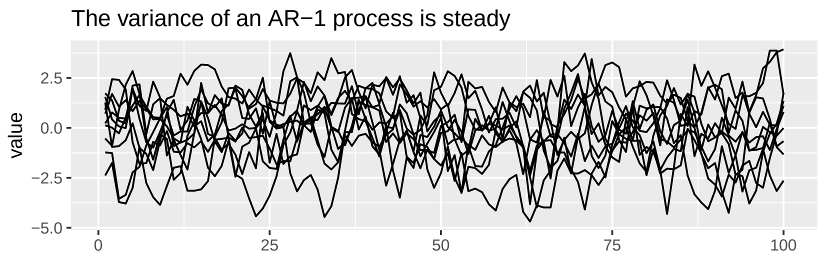


We can use ggplot to plot 10 AR(1) time series, but we need to change the data to a data frame.

```
dat <- data.frame(t = 1:TT, y = ar1)
p1 <- ggplot(dat, aes(x = t, y = y)) + geom_line() + ggtitle("AR-1") +
    xlab("") + ylab("value")
ys <- matrix(0, TT, nsim)
for (i in 1:nsim) ys[, i] <- as.vector(arima.sim(TT, model = list(ar = theta))
ys <- data.frame(ys)
ys$id <- 1:TT

ys2 <- melt(ys, id.var = "id")
p2 <- ggplot(ys2, aes(x = id, y = value, group = variable)) +
    geom_line() + xlab("") + ylab("value") + ggtitle("The variance of an AR-1
grid.arrange(p1, p2, ncol = 1)
```

```
Don't know how to automatically pick scale for object of type ts. Defaulting t
```





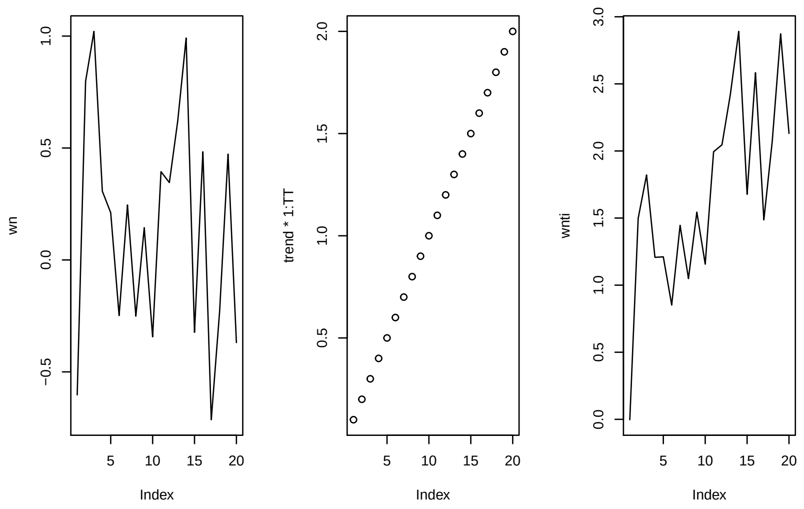
5.2.2 Stationary around a linear trend

Fluctuating around a linear trend is a very common type of stationarity used in ARMA modeling and forecasting. This is just a stationary process, like white noise or AR(1), around an linear trend up or down.

```
intercept <- 0.5
trend <- 0.1
sd <- 0.5
TT <- 20
wn <- rnorm(TT, sd = sd)  #white noise
wni <- wn + intercept  #white noise witn interept
wnti <- wn + trend * (1:TT) + intercept
```

See how the white noise with trend is just the white noise overlaid on a linear trend.

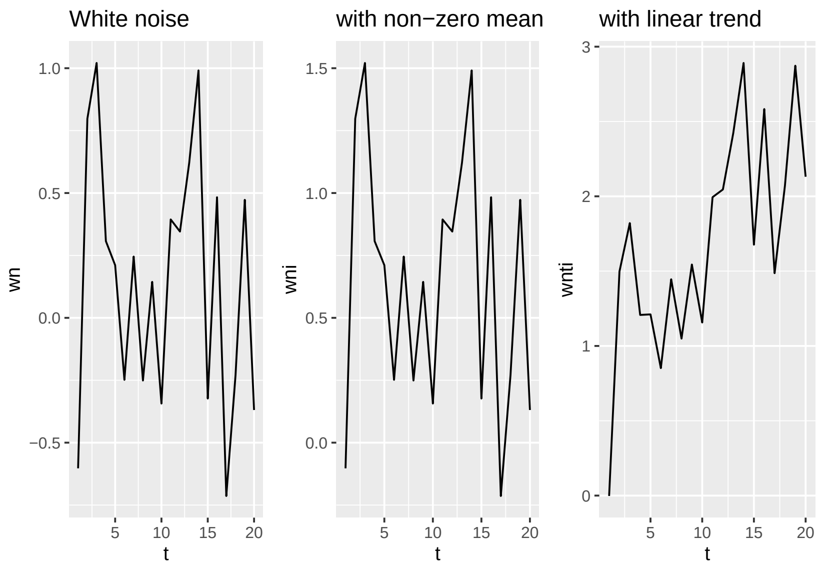
```
op <- par(mfrow = c(1, 3))
plot(wn, type = "l")
plot(trend * 1:TT)
plot(wnti, type = "l")
```



```
par(op)
```

We can make a similar plot with ggplot.

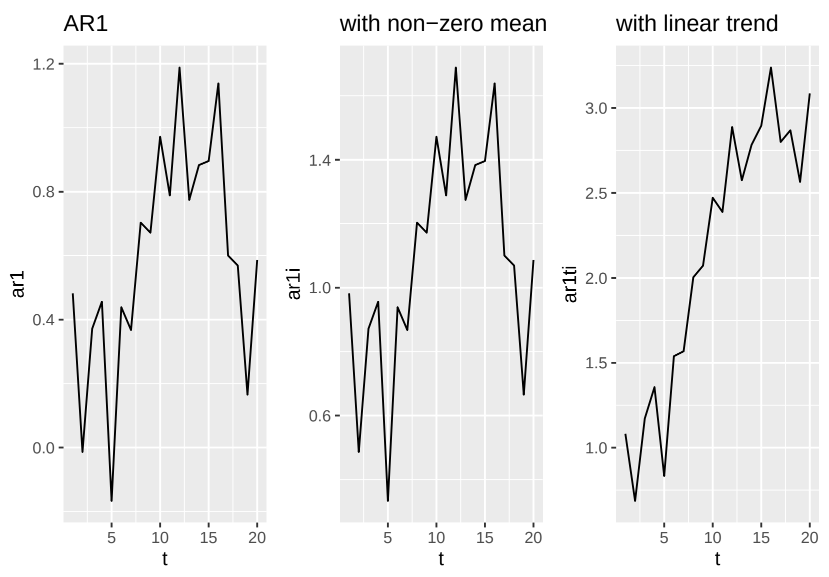
```
dat <- data.frame(t = 1:TT, wn = wn, wni = wni, wnti = wnti)
p1 <- ggplot(dat, aes(x = t, y = wn)) + geom_line() + ggtitle("White noise")
p2 <- ggplot(dat, aes(x = t, y = wni)) + geom_line() + ggtitle("with non-zero
p3 <- ggplot(dat, aes(x = t, y = wnti)) + geom_line() + ggtitle("with linear t
grid.arrange(p1, p2, p3, ncol = 3)
```



We can make a similar plot with AR(1) data. Ignore the warnings about not knowing how to pick the scale.

```
beta1 <- 0.8
ar1 <- arima.sim(TT, model = list(ar = beta1), sd = sd)
ar1i <- ar1 + intercept
ar1ti <- ar1 + trend * (1:TT) + intercept
dat <- data.frame(t = 1:TT, ar1 = ar1, ar1i = ar1i, ar1ti = ar1ti)
p4 <- ggplot(dat, aes(x = t, y = ar1)) + geom_line() + ggtitle("AR1")
p5 <- ggplot(dat, aes(x = t, y = ar1i)) + geom_line() + ggtitle("with non-zero mean")
p6 <- ggplot(dat, aes(x = t, y = ar1ti)) + geom_line() + ggtitle("with linear trend")

grid.arrange(p4, p5, p6, ncol = 3)
```

```
Don't know how to automatically pick scale for object of type ts. Defaulting to conti
Don't know how to automatically pick scale for object of type ts. Defaulting to conti
Don't know how to automatically pick scale for object of type ts. Defaulting to conti
```



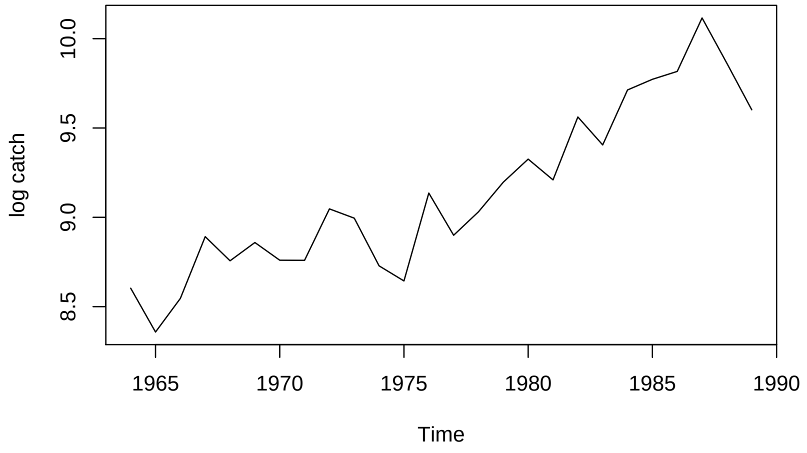
5.2.3 Greek landing data

We will look at the anchovy data. Notice the two `==` in the subset call not one `=`. We will use the Greek data before 1989 for the lab.

```
anchovy <- subset(landings, Species == "Anchovy" & Year <= 1989)$log.metric.to
anchovyts <- ts(anchovy, start = 1964)
```

Plot the data.

```
plot(anchovyts, ylab = "log catch")
```



Questions to ask.

- Does it have a trend (goes up or down)? Yes, definitely
- Does it have a non-zero mean? Yes
- Does it look like it might be stationary around a trend? Maybe

5.3 Dickey-Fuller and Augmented Dickey-Fuller tests

5.3.1 Dickey-Fuller test

The Dickey-Fuller test is testing if $\phi = 0$ in this model of the data:

$$y_t = \alpha + \beta t + \phi y_{t-1} + e_t$$

which is written as

$$\Delta y_t = y_t - y_{t-1} = \alpha + \beta t + \gamma y_{t-1} + e_t$$

where y_t is your data. It is written this way so we can do a linear regression of Δy_t against t and y_{t-1} and test if γ is different from 0. If $\gamma = 0$, then we have a random walk process. If not and $-1 < 1 + \gamma < 1$, then we have a stationary process.

5.3.2 Augmented Dickey-Fuller test

The Augmented Dickey-Fuller test allows for higher-order autoregressive processes by including Δy_{t-p} in the model. But our test is still if $\gamma = 0$.

$$\Delta y_t = \alpha + \beta t + \gamma y_{t-1} + \delta_1 \Delta y_{t-1} + \delta_2 \Delta y_{t-2} + \dots$$

The null hypothesis for both tests is that the data are non-stationary. We want to REJECT the null hypothesis for this test, so we want a p-value of less that 0.05 (or smaller).

5.3.3 ADF test using `adf.test()`

The `adf.test()` from the **tseries** package will do a Augmented Dickey-Fuller test (Dickey-Fuller if we set lags equal to 0) with a trend and an intercept. Use `?adf.test` to read about this function. The function is

```
adf.test(x, alternative = c("stationary", "explosive"),
         k = trunc((length(x)-1)^(1/3)))
```

`x` are your data. `alternative="stationary"` means that $-2 < \gamma < 0$ $(-1 < \phi < 1)$ and `alternative="explosive"` means that is outside these bounds. `k` is the number of δ lags. For a Dickey-Fuller test, so only up to AR(1) time dependency in our stationary process, we set `k=0` so we have no δ's in our test. Being able to control the lags in our test, allows us to avoid a stationarity test that is too complex to be supported by our data.

5.3.3.1 Test on white noise

Let's start by doing the test on data that we know are stationary, white noise. We will use an Augmented Dickey-Fuller test where we use the default number of lags (amount of time-dependency) in our test. For a time-series of 100, this is 4.

```r
TT <- 100
wn <- rnorm(TT)  # white noise
tseries::adf.test(wn)
```

```
Warning in tseries::adf.test(wn): p-value smaller than printed p-value

    Augmented Dickey-Fuller Test

data:  wn
Dickey-Fuller = -4.8309, Lag order = 4, p-value = 0.01
alternative hypothesis: stationary
```

The null hypothesis is rejected.

Try a Dickey-Fuller test. This is testing with a null hypothesis of AR(1) stationarity versus a null hypothesis with AR(4) stationarity when we used the default `k`.

```r
tseries::adf.test(wn, k = 0)
```

```
Warning in tseries::adf.test(wn, k = 0): p-value smaller than printed p-value

    Augmented Dickey-Fuller Test

data:  wn
Dickey-Fuller = -10.122, Lag order = 0, p-value = 0.01
alternative hypothesis: stationary
```

Notice that the test-statistic is smaller. This is a more restrictive test and we can reject the null with a higher significance level.

5.3.3.2 Test on white noise with trend

Try the test on white noise with a trend and intercept.

```
intercept <- 1
wnt <- wn + 1:TT + intercept
tseries::adf.test(wnt)
```

```
Warning in tseries::adf.test(wnt): p-value smaller than printed p-value


    Augmented Dickey-Fuller Test

data:  wnt
Dickey-Fuller = -4.8309, Lag order = 4, p-value = 0.01
alternative hypothesis: stationary
```

The null hypothesis is still rejected. `adf.test()` uses a model that allows an intercept and trend.

5.3.3.3 Test on random walk

Let's try the test on a random walk (nonstationary).

```
rw <- cumsum(rnorm(TT))
tseries::adf.test(rw)
```

```

    Augmented Dickey-Fuller Test

data:  rw
Dickey-Fuller = -2.3038, Lag order = 4, p-value = 0.4508
alternative hypothesis: stationary
```

The null hypothesis is NOT rejected as the p-value is greater than 0.05.

Try a Dickey-Fuller test.

```
tseries::adf.test(rw, k = 0)
```

```
    Augmented Dickey-Fuller Test

data:  rw
Dickey-Fuller = -1.7921, Lag order = 0, p-value = 0.6627
alternative hypothesis: stationary
```

Notice that the test-statistic is larger.

5.3.3.4 Test the anchovy data

```
tseries::adf.test(anchovyts)
```

```
    Augmented Dickey-Fuller Test

data:  anchovyts
Dickey-Fuller = -1.6851, Lag order = 2, p-value = 0.6923
alternative hypothesis: stationary
```

The p-value is greater than 0.05. We cannot reject the null hypothesis. The null hypothesis is that the data are non-stationary.

5.3.4 ADF test using `ur.df()`

The `ur.df()` Augmented Dickey-Fuller test in the **urca** package gives us a bit more information on and control over the test.

```
ur.df(y, type = c("none", "drift", "trend"), lags = 1,
      selectlags = c("Fixed", "AIC", "BIC"))
```

The `ur.df()` function allows us to specify whether to test stationarity around a zero-mean with no trend, around a non-zero mean with no trend, or around a trend with an intercept. This can be useful when we know that our data have no trend, for example if you have removed the trend already. `ur.df()` allows us to specify the lags or select them using model selection.

5.3.4.1 Test on white noise

Let's first do the test on data we know is stationary, white noise. We have to choose the `type` and `lags`. If you have no particular reason to not include an intercept and trend, then use `type="trend"`. This allows both intercept and trend. When you might you have a particular reason not to use `"trend"`? When you have removed the trend and/or intercept.

Next you need to chose the `lags`. We will use `lags=0` to do the Dickey-Fuller test. Note the number of lags you can test will depend on the amount of data that you have. `adf.test()` used a default of `trunc((length(x)-1)^(1/3))` for the lags, but `ur.df()` requires that you pass in a value or use a fixed default of 1.

`lags=0` is fitting the following model to the data:

`z.diff = gamma * z.lag.1 + intercept + trend * tt`

`z.diff` means Δy_t and `z.lag.1` is y_{t-1}. You are testing if the effect for `z.lag.1` is 0.

When you use `summary()` for the output from `ur.df()`, you will see the estimated values for γ (denoted `z.lag.1`), intercept and trend. If you see `***` or `**` on the coefficients list for `z.lag.1`, it suggest that the effect of `z.lag.1` is significantly different than 0 and this supports the assumption of stationarity. However, the test level shown is for independent data not time series data. The correct test levels (critical values) are shown at the bottom of the summary output.

```r
wn <- rnorm(TT)
test <- urca::ur.df(wn, type = "trend", lags = 0)
urca::summary(test)
```

```
###############################################
# Augmented Dickey-Fuller Test Unit Root Test #
###############################################

Test regression trend
```

```
Call:
lm(formula = z.diff ~ z.lag.1 + 1 + tt)

Residuals:
    Min      1Q  Median      3Q     Max
-2.2170 -0.6654 -0.1210  0.5311  2.6277

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept)  0.0776865  0.2037709   0.381    0.704
z.lag.1     -1.0797598  0.1014244 -10.646   <2e-16 ***
tt           0.0004891  0.0035321   0.138    0.890
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.004 on 96 degrees of freedom
Multiple R-squared:  0.5416,    Adjusted R-squared:  0.532
F-statistic: 56.71 on 2 and 96 DF,  p-value: < 2.2e-16


Value of test-statistic is: -10.646 37.806 56.7083

Critical values for test statistics:
      1pct  5pct 10pct
tau3 -4.04 -3.45 -3.15
phi2  6.50  4.88  4.16
phi3  8.73  6.49  5.47
```

Note `urca::` in front of `summary()` is needed if you have not loaded the urca package with `library(urca)`.

We need to look at information at the bottom of the summary output for the test statistics and critical values. The part that looks like this

```
Value of test-statistic is: #1 #2 #3

Critical values for test statistics:
      1pct  5pct 10pct
```

```
tau3   xxx  xxx   xxx
...
```

The first test statistic number is for $\gamma = 0$ and will be labeled `tau`, `tau2` or `tau3`.

In our example with white noise, notice that the test statistic is LESS than the critical value for `tau3` at 5 percent. This means the null hypothesis is rejected at $\alpha = 0.05$, a standard level for significance testing.

5.3.4.2 When you might want to use `ur.df()`

If you remove the trend (and/or level) from your data, the `ur.df()` test allows you to increase the power of the test by removing the trend and/or level from the model.

5.4 KPSS test

The null hypothesis for the KPSS test is that the data are stationary. For this test, we do NOT want to reject the null hypothesis. In other words, we want the p-value to be greater than 0.05 not less than 0.05.

5.4.1 Test on simulated data

Let's try the KPSS test on white noise with a trend. The default is a null hypothesis with no trend. We will change this to `null="Trend"`.

```
tseries::kpss.test(wnt, null = "Trend")
```

```
Warning in tseries::kpss.test(wnt, null = "Trend"): p-value greater than print

    KPSS Test for Trend Stationarity

data:  wnt
KPSS Trend = 0.045579, Truncation lag parameter = 4, p-value = 0.1
```

The p-value is greater than 0.05. The null hypothesis of stationarity around a trend is not rejected.

Let's try the KPSS test on white noise with a trend but let's use the default of stationary with no trend.

```
tseries::kpss.test(wnt, null = "Level")
```

```
Warning in tseries::kpss.test(wnt, null = "Level"): p-value smaller than printed p-va

    KPSS Test for Level Stationarity

data:  wnt
KPSS Level = 2.1029, Truncation lag parameter = 4, p-value = 0.01
```

The p-value is less than 0.05. The null hypothesis of stationarity around a level is rejected. This is white noise around a trend so it is definitely a stationary process but has a trend. This illustrates that you need to be thoughtful when applying stationarity tests.

5.4.2 Test the anchovy data

Let's try the anchovy data.

```
kpss.test(anchovyts, null = "Trend")
```

```
    KPSS Test for Trend Stationarity

data:  anchovyts
KPSS Trend = 0.14779, Truncation lag parameter = 2, p-value = 0.04851
```

The null is rejected (p-value less than 0.05). Again stationarity is not supported.

5.5 Dealing with non-stationarity

The anchovy data have failed both tests for the stationarity, the Augmented Dickey-Fuller and the KPSS test. How do we fix this? The approach in the Box-Jenkins method is to use differencing.

Let's see how this works with random walk data. A random walk is non-stationary but the difference is white noise so is stationary:

$$x_t - x_{t-1} = e_t, e_t \sim N(0, \sigma)$$

```
adf.test(diff(rw))
```

```

    Augmented Dickey-Fuller Test

data:  diff(rw)
Dickey-Fuller = -3.8711, Lag order = 4, p-value = 0.01834
alternative hypothesis: stationary
```

```
kpss.test(diff(rw))
```

```
Warning in kpss.test(diff(rw)): p-value greater than printed p-value

    KPSS Test for Level Stationarity

data:  diff(rw)
KPSS Level = 0.30489, Truncation lag parameter = 3, p-value = 0.1
```

If we difference random walk data, the null is rejected for the ADF test and not rejected for the KPSS test. This is what we want.

Let's try a single difference with the anchovy data. A single difference means `dat(t)-dat(t-1)`. We get this using `diff(anchovyts)`.

```
diff1dat <- diff(anchovyts)
adf.test(diff1dat)
```

```
    Augmented Dickey-Fuller Test

data:  diff1dat
Dickey-Fuller = -3.2718, Lag order = 2, p-value = 0.09558
alternative hypothesis: stationary
```

```
kpss.test(diff1dat)
```

```
Warning in kpss.test(diff1dat): p-value greater than printed p-value

    KPSS Test for Level Stationarity

data:  diff1dat
KPSS Level = 0.089671, Truncation lag parameter = 2, p-value = 0.1
```

If a first difference were not enough, we would try a second difference which is the difference of a first difference.

```
diff2dat <- diff(diff1dat)
adf.test(diff2dat)
```

```
Warning in adf.test(diff2dat): p-value smaller than printed p-value

    Augmented Dickey-Fuller Test

data:  diff2dat
Dickey-Fuller = -4.8234, Lag order = 2, p-value = 0.01
alternative hypothesis: stationary
```

The null hypothesis of a random walk is now rejected so you might think that a 2nd difference is needed for the anchovy data. However the actual problem is that the default for `adf.test()` includes a trend but we removed the trend with our first difference. Thus we included an unneeded trend parameter in our test. Our data are not that long and this affects the result.

Let's repeat without the trend and we'll see that the null hypothesis is rejected. The number of lags is set to be what would be used by `adf.test()`. See `?adf.test`.

```r
k <- trunc((length(diff1dat) - 1)^(1/3))
test <- urca::ur.df(diff1dat, type = "drift", lags = k)
summary(test)
```

```
###############################################
# Augmented Dickey-Fuller Test Unit Root Test #
###############################################

Test regression drift


Call:
lm(formula = z.diff ~ z.lag.1 + 1 + z.diff.lag)

Residuals:
     Min       1Q   Median       3Q      Max
-0.37551 -0.13887  0.04753  0.13277  0.28223

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  0.11062    0.06165   1.794  0.08959 .
z.lag.1     -2.16711    0.64900  -3.339  0.00365 **
z.diff.lag1  0.58837    0.47474   1.239  0.23113
z.diff.lag2  0.13273    0.25299   0.525  0.60623
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.207 on 18 degrees of freedom
Multiple R-squared:  0.7231,    Adjusted R-squared:  0.677
F-statistic: 15.67 on 3 and 18 DF,  p-value: 2.918e-05


Value of test-statistic is: -3.3391 5.848

Critical values for test statistics:
      1pct  5pct 10pct
tau2 -3.75 -3.00 -2.63
```

```
phi1  7.88  5.18  4.12
```

5.5.1 `ndiffs()`

As an alternative to trying many different differences and remembering to include or not include the trend or level, you can use the `ndiffs()` function in the **forecast** package. This automates finding the number of differences needed.

```
forecast::ndiffs(anchovyts, test = "kpss")
```

```
[1] 1
```

```
forecast::ndiffs(anchovyts, test = "adf")
```

```
[1] 1
```

One difference is required to pass both the ADF and KPSS stationarity tests.

5.6 Summary: stationarity testing

The basic stationarity diagnostics are the following

- Plot your data. Look for
  - An increasing trend
  - A non-zero level (if no trend)
  - Strange shocks or steps in your data (indicating something dramatic changed like the data collection methodology)
- Apply stationarity tests
  - `adf.test()` p-value should be less than 0.05 (reject null)
  - `kpss.test()` p-value should be greater than 0.05 (do not reject null)
- If stationarity tests are failed, then try differencing to correct
  - Try `ndiffs()` in the **forecast** package or manually try different differences.

5.7 Estimating ARMA parameters

Let's start with fitting to simulated data.

5.7.1 AR(2) data

Simulate AR(2) data and add a mean level so that the data are not mean 0.

$$\begin{aligned} x_t &= 0.8x_{t-1} + 0.1x_{t-2} + e_t \\ y_t &= x_t + m \end{aligned} \tag{5.1}$$

```
m <- 1
ar2 <- arima.sim(n = 1000, model = list(ar = c(0.8, 0.1))) +
    m
```

To see info on `arima.sim()`, type `?arima.sim`.

5.7.2 Fit with `Arima()`

Fit an ARMA(2) with level to the data.

```
forecast::Arima(ar2, order = c(2, 0, 0), include.constant = TRUE)
```

```
Series: ar2
ARIMA(2,0,0) with non-zero mean

Coefficients:
         ar1     ar2    mean
      0.7684  0.1387  0.9561
s.e.  0.0314  0.0314  0.3332

sigma^2 estimated as 0.9832:  log likelihood=-1409.77
AIC=2827.54   AICc=2827.58   BIC=2847.17
```

Note, the model being fit by `Arima()` is not this model

$$y_t = m + 0.8y_{t-1} + 0.1y_{t-2} + e_t \tag{5.2}$$

It is this model:

$$(y_t - m) = 0.8(y_{t-1} - m) + 0.1(y_{t-2} - m) + e_t \tag{5.3}$$

or as written above:

$$\begin{gathered} x_t = 0.8x_{t-1} + 0.1x_{t-2} + e_t \\ y_t = x_t + m \end{gathered} \tag{5.4}$$

We could also use `arima()` to fit to the data.

```
arima(ar2, order = c(2, 0, 0), include.mean = TRUE)
```

```
Warning in arima(ar2, order = c(2, 0, 0), include.mean = TRUE): possible convergence
optim gave code = 1


Call:
arima(x = ar2, order = c(2, 0, 0), include.mean = TRUE)

Coefficients:
         ar1     ar2  intercept
      0.7684  0.1387     0.9561
s.e.  0.0314  0.0314     0.3332

sigma^2 estimated as 0.9802:  log likelihood = -1409.77,  aic = 2827.54
```

However we will not be using `arima()` directly because for if we have differenced data, it will not allow us to include and estimated mean level. Unless we have transformed our differenced data in a way that ensures it is mean zero, then we want to include a mean.

Try increasing the length of the simulated data (from 100 to 1000 say) and see how that affects your parameter estimates. Run the simulation a few times.

5.7.3 AR(1) simulated data

```
ar1 <- arima.sim(n = 100, model = list(ar = c(0.8))) + m
forecast::Arima(ar1, order = c(1, 0, 0), include.constant = TRUE)
```

```
Series: ar1
ARIMA(1,0,0) with non-zero mean

Coefficients:
         ar1    mean
      0.7091  0.4827
s.e.  0.0705  0.3847

sigma^2 estimated as 1.34:  log likelihood=-155.85
AIC=317.7   AICc=317.95   BIC=325.51
```

5.7.4 ARMA(1,2) simulated data

Simulate ARMA(1,2)

$$x_t = 0.8x_{t-1} + e_t + 0.8e_{t-1} + 0.2e_{t-2}$$

```
arma12 = arima.sim(n = 100, model = list(ar = c(0.8), ma = c(0.8,
    0.2))) + m
forecast::Arima(arma12, order = c(1, 0, 2), include.constant = TRUE)
```

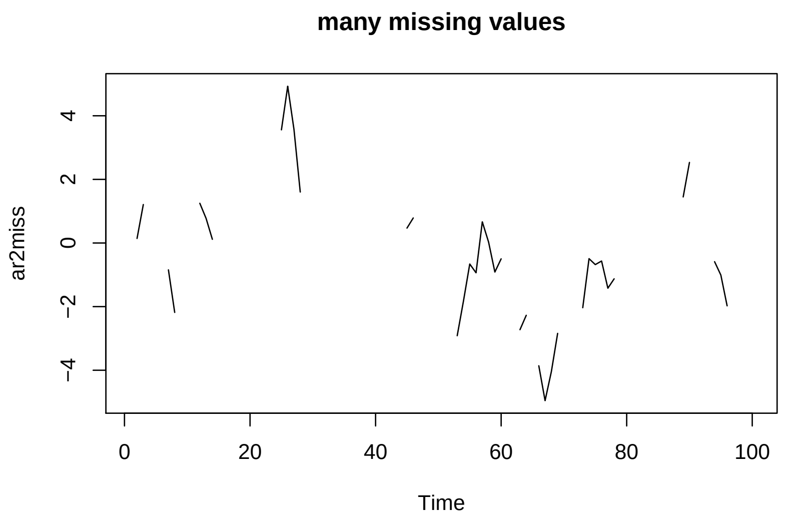
```
Series: arma12
ARIMA(1,0,2) with non-zero mean

Coefficients:
         ar1     ma1     ma2    mean
      0.8138  0.8599  0.1861  0.3350
s.e.  0.0646  0.1099  0.1050  0.8145

sigma^2 estimated as 0.6264:  log likelihood=-118.02
AIC=246.03   AICc=246.67   BIC=259.06
```

We will up the number of data points to 1000 because models with a MA component take a lot of data to estimate. Models with MA(>1) are not very practical for fisheries data for that reason.

5.7.5 These functions work for data with missing values

Create some AR(2) data and then add missing values (NA).

```
ar2miss <- arima.sim(n = 100, model = list(ar = c(0.8, 0.1)))
ar2miss[sample(100, 50)] <- NA
plot(ar2miss, type = "l")
title("many missing values")
```



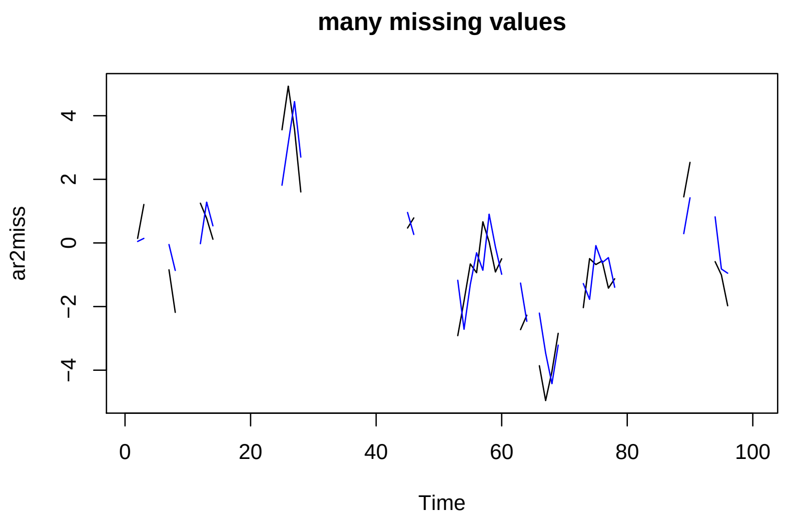
Fit

```
fit <- forecast::Arima(ar2miss, order = c(2, 0, 0))
fit
```

```
Series: ar2miss
ARIMA(2,0,0) with non-zero mean

Coefficients:
         ar1      ar2     mean
      1.0625  -0.2203  -0.0586
s.e.  0.1555   0.1618   0.6061

sigma^2 estimated as 0.9679:  log likelihood=-79.86
AIC=167.72   AICc=168.15   BIC=178.06
```

Note `fitted()` does not return the expected value at time t. It is the expected value of y_t given the data up to time $t-1$.

```r
plot(ar2miss, type = "l")
title("many missing values")
lines(fitted(fit), col = "blue")
```



It is easy enough to get the expected value of y_t for all the missing values but we'll learn to do that when we learn the **MARSS** package and can apply the Kalman Smoother in that package.

5.8 Estimating the ARMA orders

We will use the `auto.arima()` function in **forecast**. This function will estimate the level of differencing needed to make our data stationary and estimate the AR and MA orders using AICc (or BIC if we choose).

5.8.1 Example: model selection for AR(2) data

```
forecast::auto.arima(ar2)
```

```
Series: ar2
ARIMA(2,0,2) with non-zero mean

Coefficients:
         ar1     ar2     ma1      ma2    mean
      0.2795  0.5938  0.4861  -0.0943  0.9553
s.e.  1.1261  1.0413  1.1284   0.1887  0.3398

sigma^2 estimated as 0.9848:  log likelihood=-1409.57
AIC=2831.15   AICc=2831.23   BIC=2860.59
```

Works with missing data too though might not estimate very close to the true model form.

```
forecast::auto.arima(ar2miss)
```

```
Series: ar2miss
ARIMA(0,1,0)

sigma^2 estimated as 1.066:  log likelihood=-82.07
AIC=166.15   AICc=166.19   BIC=168.72
```

5.8.2 Fitting to 100 simulated data sets

Let's fit to 100 simulated data sets and see how often the true (generating) model form is selected.

```
save.fits <- rep(NA, 100)
for (i in 1:100) {
    a2 <- arima.sim(n = 100, model = list(ar = c(0.8, 0.1)))
    fit <- auto.arima(a2, seasonal = FALSE, max.d = 0, max.q = 0)
    save.fits[i] <- paste0(fit$arma[1], "-", fit$arma[2])
}
table(save.fits)
```

```
save.fits
1-0 2-0 3-0 
 71  22   7 
```

`auto.arima()` uses AICc for selection by default. You can change that to AIC or BIC using `ic="aic"` or `ic="bic"`.

Repeat the simulation using AIC and BIC to see how the choice of the information criteria affects the model that is selected.

5.8.3 Trace=TRUE

We can set `Trace=TRUE` to see what models `auto.arima()` fit.

```
forecast::auto.arima(ar2, trace = TRUE)
```

```

 Fitting models using approximations to speed things up...

 ARIMA(2,0,2) with non-zero mean : 2824.88
 ARIMA(0,0,0) with non-zero mean : 4430.868
 ARIMA(1,0,0) with non-zero mean : 2842.785
 ARIMA(0,0,1) with non-zero mean : 3690.512
 ARIMA(0,0,0) with zero mean     : 4602.31
 ARIMA(1,0,2) with non-zero mean : 2827.422
 ARIMA(2,0,1) with non-zero mean : 2825.235
 ARIMA(3,0,2) with non-zero mean : 2830.176
 ARIMA(2,0,3) with non-zero mean : 2826.503
 ARIMA(1,0,1) with non-zero mean : 2825.438
 ARIMA(1,0,3) with non-zero mean : 2829.358
 ARIMA(3,0,1) with non-zero mean : Inf
 ARIMA(3,0,3) with non-zero mean : 2825.766
 ARIMA(2,0,2) with zero mean     : 2829.536

 Now re-fitting the best model(s) without approximations...

 ARIMA(2,0,2) with non-zero mean : 2831.232

 Best model: ARIMA(2,0,2) with non-zero mean

Series: ar2
ARIMA(2,0,2) with non-zero mean

Coefficients:
         ar1     ar2     ma1      ma2    mean
      0.2795  0.5938  0.4861  -0.0943  0.9553
s.e.  1.1261  1.0413  1.1284   0.1887  0.3398

sigma^2 estimated as 0.9848:  log likelihood=-1409.57
AIC=2831.15   AICc=2831.23   BIC=2860.59
```

5.8.4 stepwise=FALSE

We can set `stepwise=FALSE` to use an exhaustive search. The model may be different than the result from the non-exhaustive search.

```
forecast::auto.arima(ar2, trace = TRUE, stepwise = FALSE)
```

```
 Fitting models using approximations to speed things up...
```

```
ARIMA(0,0,0) with zero mean     : 4602.31
ARIMA(0,0,0) with non-zero mean : 4430.868
ARIMA(0,0,1) with zero mean     : 3815.931
ARIMA(0,0,1) with non-zero mean : 3690.512
ARIMA(0,0,2) with zero mean     : 3425.037
ARIMA(0,0,2) with non-zero mean : 3334.754
ARIMA(0,0,3) with zero mean     : 3239.347
ARIMA(0,0,3) with non-zero mean : 3170.541
ARIMA(0,0,4) with zero mean     : 3114.265
ARIMA(0,0,4) with non-zero mean : 3059.938
ARIMA(0,0,5) with zero mean     : 3042.136
ARIMA(0,0,5) with non-zero mean : 2998.531
ARIMA(1,0,0) with zero mean     : 2850.655
ARIMA(1,0,0) with non-zero mean : 2842.785
ARIMA(1,0,1) with zero mean     : 2830.652
ARIMA(1,0,1) with non-zero mean : 2825.438
ARIMA(1,0,2) with zero mean     : 2832.668
ARIMA(1,0,2) with non-zero mean : 2827.422
ARIMA(1,0,3) with zero mean     : 2834.675
ARIMA(1,0,3) with non-zero mean : 2829.358
ARIMA(1,0,4) with zero mean     : 2835.539
ARIMA(1,0,4) with non-zero mean : 2829.825
ARIMA(2,0,0) with zero mean     : 2828.987
ARIMA(2,0,0) with non-zero mean : 2823.774
ARIMA(2,0,1) with zero mean     : 2829.952
ARIMA(2,0,1) with non-zero mean : 2825.235
ARIMA(2,0,2) with zero mean     : 2829.536
ARIMA(2,0,2) with non-zero mean : 2824.88
ARIMA(2,0,3) with zero mean     : 2831.461
ARIMA(2,0,3) with non-zero mean : 2826.503
ARIMA(3,0,0) with zero mean     : 2831.057
ARIMA(3,0,0) with non-zero mean : 2826.236
ARIMA(3,0,1) with zero mean     : Inf
ARIMA(3,0,1) with non-zero mean : Inf
ARIMA(3,0,2) with zero mean     : 2834.788
ARIMA(3,0,2) with non-zero mean : 2830.176
ARIMA(4,0,0) with zero mean     : 2833.323
ARIMA(4,0,0) with non-zero mean : 2828.759
```

```
 ARIMA(4,0,1) with zero mean     : 2827.798
 ARIMA(4,0,1) with non-zero mean : 2823.853
 ARIMA(5,0,0) with zero mean     : 2835.315
 ARIMA(5,0,0) with non-zero mean : 2830.501

 Now re-fitting the best model(s) without approximations...




 Best model: ARIMA(2,0,0) with non-zero mean
```

```
Series: ar2
ARIMA(2,0,0) with non-zero mean

Coefficients:
         ar1     ar2    mean
      0.7684  0.1387  0.9561
s.e.  0.0314  0.0314  0.3332

sigma^2 estimated as 0.9832:  log likelihood=-1409.77
AIC=2827.54   AICc=2827.58   BIC=2847.17
```

5.8.5 Fit to the anchovy data

```
fit <- auto.arima(anchovyts)
fit
```

```
Series: anchovyts
ARIMA(0,1,1) with drift

Coefficients:
          ma1   drift
      -0.6685  0.0542
s.e.   0.1977  0.0142
```

```
sigma^2 estimated as 0.04037:  log likelihood=5.39
AIC=-4.79   AICc=-3.65   BIC=-1.13
```

Note `arima()` writes a MA model like:

$$x_t = e_t + b_1 e_{t-1} + b_2 e_{t-2}$$

while many authors use this notation:

$$x_t = e_t - \theta_1 e_{t-1} - \theta_2 e_{t-2}$$

so the MA parameters reported by `auto.arima()` will be NEGATIVE of that reported in Stergiou and Christou (1996) who analyze these same data. *Note, in Stergiou and Christou, the model is written in backshift notation on page 112. To see the model as the equation above, I translated from backshift to non-backshift notation.*

5.9 Check residuals

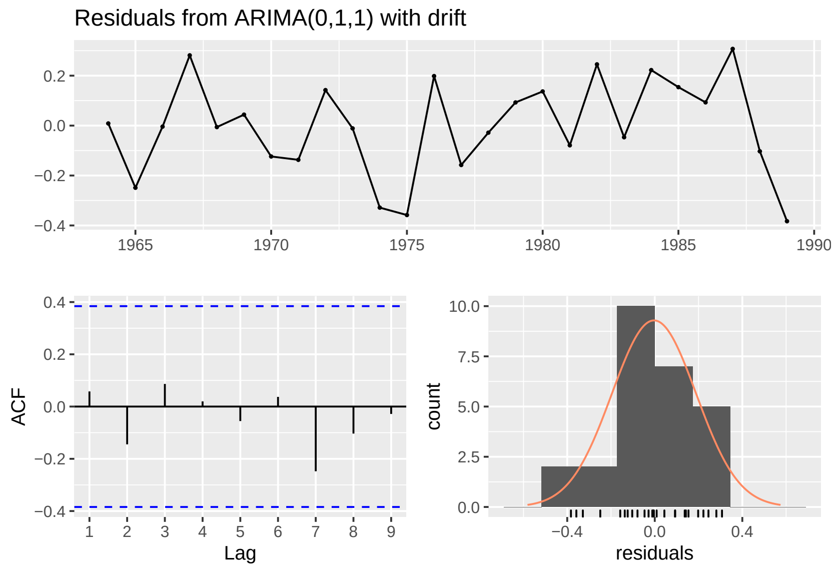
We can do a test of autocorrelation of the residuals with `Box.test()` with `fitdf` adjusted for the number of parameters estimated in the fit. In our case, MA(1) and drift parameters.

```
res <- resid(fit)
Box.test(res, type = "Ljung-Box", lag = 12, fitdf = 2)
```

```

	Box-Ljung test

data:  res
X-squared = 5.1609, df = 10, p-value = 0.8802
```

`checkresiduals()` in the **forecast** package will automate this test and show some standard diagnostics plots.

```r
forecast::checkresiduals(fit)
```

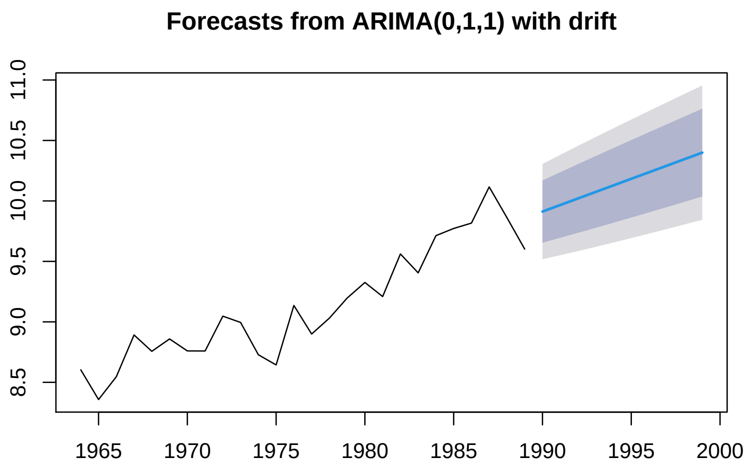


```
    Ljung-Box test

data:  Residuals from ARIMA(0,1,1) with drift
Q* = 1.0902, df = 3, p-value = 0.7794

Model df: 2.   Total lags used: 5
```

5.10 Forecast from a fitted ARIMA model

We can create a forecast from our anchovy ARIMA model using `forecast()`. The shading is the 80% and 95% prediction intervals.

```r
fr <- forecast::forecast(fit, h = 10)
plot(fr)
```



5.11 Seasonal ARIMA model

The Chinook data are monthly and start in January 1990. To make this into a ts object do

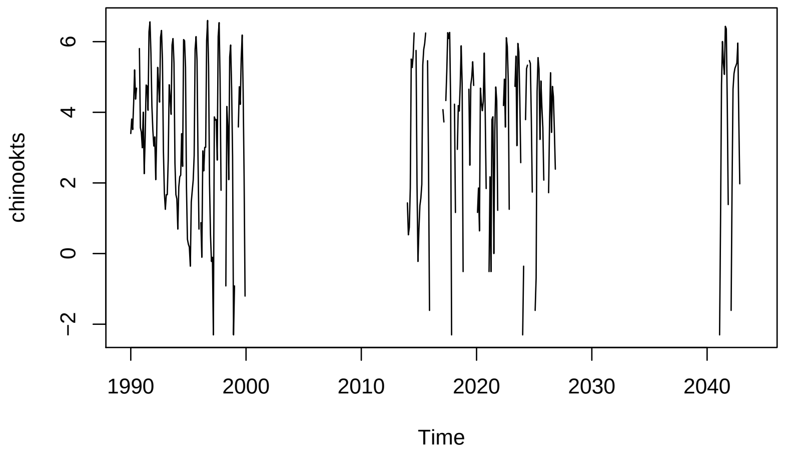
```
chinookts <- ts(chinook$log.metric.tons, start = c(1990, 1),
    frequency = 12)
```

`start` is the year and month and frequency is the number of months in the year.

Use `?ts` to see more examples of how to set up ts objects.

5.11.1 Plot seasonal data

```
plot(chinookts)
```



5.11.2 `auto.arima()` for seasonal ts

`auto.arima()` will recognize that our data has season and fit a seasonal ARIMA model to our data by default. Let's define the training data up to 1998 and use 1999 as the test data.

```
traindat <- window(chinookts, c(1990, 10), c(1998, 12))
testdat <- window(chinookts, c(1999, 1), c(1999, 12))
fit <- forecast::auto.arima(traindat)
fit
```

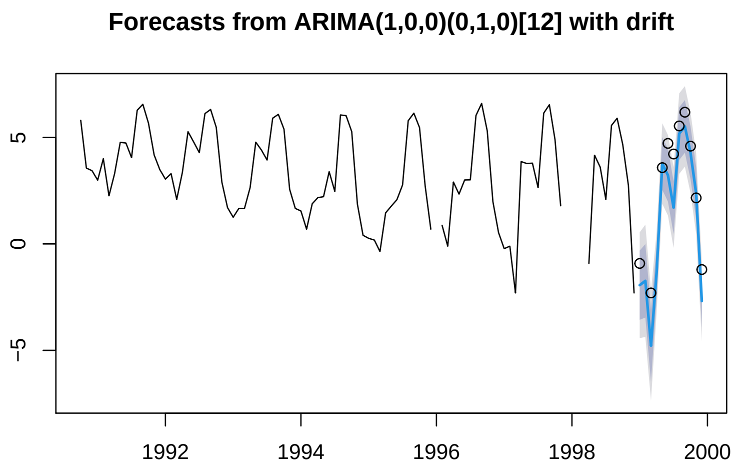
```
Series: traindat
ARIMA(1,0,0)(0,1,0)[12] with drift

Coefficients:
         ar1    drift
      0.3676  -0.0320
s.e.  0.1335   0.0127

sigma^2 estimated as 0.8053:  log likelihood=-107.37
AIC=220.73   AICc=221.02   BIC=228.13
```

Use `?window` to understand how subsetting a ts object works.

5.12 Forecast using a seasonal model

Forecasting works the same using the `forecast()` function.

```
fr <- forecast::forecast(fit, h = 12)
plot(fr)
points(testdat)
```



5.13 Problems

For these problems, use the catch landings from Greek waters (`greeklandings`) and the Chinook landings (`chinook`) in Washington data. Load the data as follows:

```
data(greeklandings, package = "atsalibrary")
landings <- greeklandings
data(chinook, package = "atsalibrary")
chinook <- chinook.month
```

1. Augmented Dickey-Fuller tests in R.

   a. What is the null hypothesis for the Dickey-Fuller and Augmented Dickey-Fuller tests?
   b. How do the Dickey-Fuller and Augmented Dickey-Fuller tests differ?
   c. For `adf.test()`, does the test allow the data to have a non-zero level? Does the test allow the data to be stationarity around a trend (a linear slope)?
   d. For `ur.df()`, what does type = "none", "drift", and "trend" mean? Which one gives you the same result as `adf.test()`? What do you have to set the lags equal to get the default lags in `adf.test()`?
   e. For `ur.df()`, how do you determine if the null hypothesis is rejected?
   f. For `ur.df()`, how do you determine if there is a significant trend in the data? How do you determine if the intercept is different than zero?

2. KPSS tests in R.

   a. What is the null hypothesis for the KPSS test?
   b. For `kpss.test()`, what does setting null equal to "Level" versus "Trend" change?

3. Repeat the stationarity tests for sardine 1964-1987 in the landings data set. Here is how to set up the data for another species.

```
datdf <- subset(landings, Species == "Sardine")
dat <- ts(datdf$log.metric.tons, start = 1964)
dat <- window(dat, start = 1964, end = 1987)
```

   a. Do a Dickey-Fuller (DF) test using `ur.df()` and `adf.test()`. You will have to set the lags. What does the result tell you? *Note for `ur.df()` use `summary(ur.df(...))` and look at the bottom of the summary information for the test statistics and critical values. The first test statistic is the one you want, labeled `tau` (or `tau3`).*
   b. Do an Augmented Dickey-Fuller (ADF) test using `ur.df()`. How did you choose to set the lags? How is the ADF test different than the DF test?
   c. Do a KPSS test using `kpss.test()`. What does the result tell you?

4. Use the anchovy 1964-2007 data [Corrected 1/20. If you did the HW with 1964-1987, that's fine but part b won't have any models within 2 of the best for the shorter series.]. Fit this time series using `auto.arima()` with `trace=TRUE`.

```
forecast::auto.arima(anchovy, trace = TRUE)
```

   a. Fit each of the models listed using `Arima()` and show that you can produce the same AICc value that is shown in the trace table.
   b. What models are within ΔAICc of 2 of the best model (model with lowest AICc)? What is different about these models?

5. Repeat the stationarity tests and differencing tests for anchovy using the following two time ranges: 1964-1987 and 1988-2007. The following shows you how to subset the data:

```
datdf <- subset(landings, Species == "Anchovy")
dat <- ts(datdf$log.metric.tons, start = 1964)
dat64.87 <- window(dat, start = 1964, end = 1987)
```

   a. Plot the time series for the two time periods. For the `kpss.test()`, which null is appropriate, "Level" or "Trend"?
   b. Do the conclusions regarding stationarity and the amount of differencing needed change depending on which time period you analyze? For both time periods, use `adf.test()` with default values and `kpss.test()` with null="Trend".
   c. Fit each time period using `auto.arima()`. Do the selected models change? What do the coefficients mean? Coefficients means the mean and drifts terms and the AR and MA terms.
   d. Discuss the best models for each time period. How are they different?
   e. You cannot compare the AIC values for an Arima(0,1,0) and Arima(0,0,1). Why do you think that is? Hint when comparing AICs, the data being fit must be the same for each model.

6. For the anchovy 1964-2007 data, use `auto.arima()` with `stepwise=FALSE` to fit models.

   a. find the set of models within $\Delta AICc = 2$ of the top model.
   b. Use `Arima()` to fit the models with Inf or -Inf in the list. Does the set of models within $\Delta AICc = 2$ change?
   c. Create a 5-year forecast for each of the top 3 models according to AICc.
   d. How do the forecasts differ in trend and size of prediction intervals?

7. Using the `chinook` data set,

   a. Set up a monthly time series object for the Chinook log metric tons catch for Jan 1990 to Dec 2015.
   b. Fit a seasonal model to the Chinook Jan 1990 to Dec 1999 data using `auto.arima()`.
   c. Create a forecast through 2015 using the model in part b.
   d. Plot the forecast with the 2014 and 2015 actual landings added as data points.

e. The model from part b has drift. Fit this model using `Arima()` without drift and compare the 2015 forecast with this model.

Chapter 6

Univariate state-space models

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

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

Data and packages

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

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

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

6.1 Fitting a state-space model with MARSS

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

$$\begin{gathered}
\mathbf{x}_t = \mathbf{B}\mathbf{x}_{t-1} + \mathbf{u} + \mathbf{w}_t \text{ where } \mathbf{w}_t \sim \text{N}(0, \mathbf{Q}) \\
\mathbf{y}_t = \mathbf{Z}\mathbf{x}_t + \mathbf{a} + \mathbf{v}_t \text{ where } \mathbf{v}_t \sim \text{N}(0, \mathbf{R}) \\
\mathbf{x}_0 = \boldsymbol{\mu}
\end{gathered} \tag{6.1}$$

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

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

A basic `MARSS()` call looks like `fit=MARSS(y, model=list(...))`. The argument `model` tells the function what form the parameters take. The list has the elements with the names: `B`, `U`, `Q`, etc. The names correspond to the parameters with the same names in Equation (6.1) except that $\boldsymbol{\mu}$ is called `x0`. `tinitx` indicates whether the initial $\mathbf{x}$ is specified at $t = 0$ so $\mathbf{x}_0$ or $t = 1$ so $\mathbf{x}_1$.

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

$$\begin{gathered}
x_t = bx_{t-1} + w_t \text{ where } \mathbf{w}_t \sim \text{N}(0, q) \\
y_t = x_t + v_t \text{ where } v_t \sim \text{N}(0, r) \\
x_0 = \mu
\end{gathered} \tag{6.2}$$

To fit this with `MARSS()`, we need to write Equation (6.2) as Equation (6.1). Equation (6.1) is in MATRIX form. In the model list, the parameters must be written EXACTLY like they would be written for Equation (6.1). For example, `1` is the number 1 in R. It is not a matrix:

```
class(1)
```

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

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

With that mind, our model list for Equation (6.2) is:

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

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

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

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

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

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

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

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

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

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

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

6.2 Examples using the Nile river data

We will use the data from the Nile River (Figure 6.1). We will fit different flow models to the data and compare the models with AIC.

```
library(datasets)
dat <- Nile
```

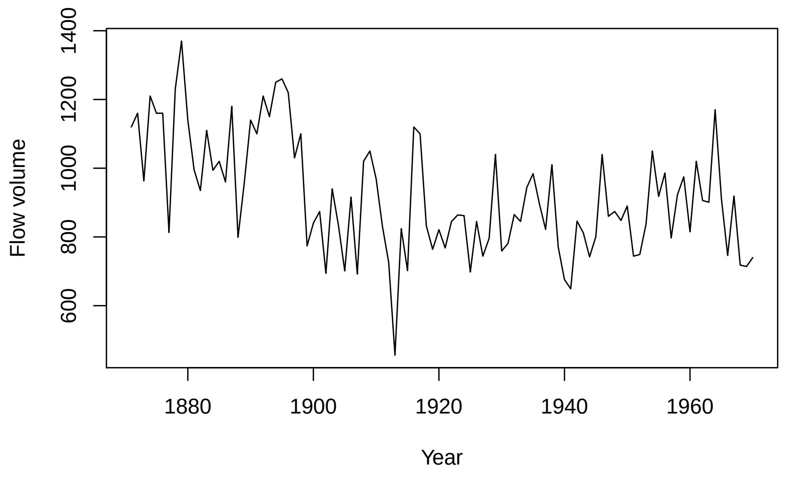


Figure 6.1: The Nile River flow volume 1871 to 1970 (`Nile` dataset in R).

6.2.1 Flat level model

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

$$y_t = \mu + v_t \text{ where } v_t \sim \text{N}(0, r) \tag{6.3}$$

where y_t is the river flow volume at year t.

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

$$\begin{gathered} x_t = 1 \times x_{t-1} + 0 + w_t \text{ where } w_t \sim \text{N}(0, 0) \\ y_t = 1 \times x_t + 0 + v_t \text{ where } v_t \sim \text{N}(0, r) \\ x_0 = \mu \end{gathered} \tag{6.4}$$

The model is specified as a list as follows:

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

We then fit the model:

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

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

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

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

6.2.2 Linear trend in flow model

Figure 6.2 shows the fit for the flat average river flow model. Looking at the data, we might expect that a declining average river flow would be better. In MARSS form, that model would be:

$$\begin{gathered} x_t = 1 \times x_{t-1} + u + w_t \text{ where } w_t \sim \text{N}(0, 0) \\ y_t = 1 \times x_t + 0 + v_t \text{ where } v_t \sim \text{N}(0, r) \\ x_0 = \mu \end{gathered} \tag{6.5}$$

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

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

We then fit the model:

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

Here are the estimates, log-likelihood and AICc:

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

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

Figure 6.2 shows the fits for the two models with deterministic models (flat and declining) for mean river flow along with their AICc values (smaller AICc is better). The AICc for the model with a declining river flow is lower by over 20 (which is a lot).

6.2.3 Stochastic level model

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

$$\begin{gathered} x_t = x_{t-1} + w_t \text{ where } w_t \sim \text{N}(0, q) \\ y_t = x_t + v_t \text{ where } v_t \sim \text{N}(0, r) \\ x_0 = \mu \end{gathered} \tag{6.6}$$

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

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

We could also use the text shortcuts to specify the model. Because **R** and **Q** are 1×1 matrices, “unconstrained”, “diagonal and unequal”, “diagonal and equal” and “equalvarcov” will all lead to a 1×1 matrix with one estimated element. For **a** and **u**, the following shortcut could be used:

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

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

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

Here are the estimates, log-likelihood and AICc:

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

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

6.2.4 Stochastic level model with drift

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

$$\begin{gathered} x_t = x_{t-1} + u + w_t \text{ where } w_t \sim \mathrm{N}(0, q) \\ y_t = x_t + v_t \text{ where } v_t \sim \mathrm{N}(0, r) \\ x_0 = \mu \end{gathered} \tag{6.7}$$

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

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

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

Here are the estimates, log-likelihood and AICc:

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

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

Figure 6.2 shows all the models along with their AICc values.

6.3 The StructTS function

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

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

```

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

Variances:
  level  epsilon
   1469    15099
```

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

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

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

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

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

```
t <- 10
fitted(fit.sts)[t]
```

```
[1] 1162.904
```

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

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

```
fitted(kem.2, type = "xtT") %>%
    subset(t == 11)
```

If you needed the one-step predictions from `MARSS()`, you can get that using "xtt1".

```
fitted(kem.2, type = "xtt1") %>%
    subset(t == 11)
```

This is the expected value of x_t conditioned on y_1 to y_{t-1}.

```
Loading required package: lattice

Loading required package: survival

Loading required package: Formula

Attaching package: 'Hmisc'

The following object is masked from 'package:quantmod':

    Lag

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

    format.pval, units
```

6.4 Comparing models with AIC and model weights

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

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

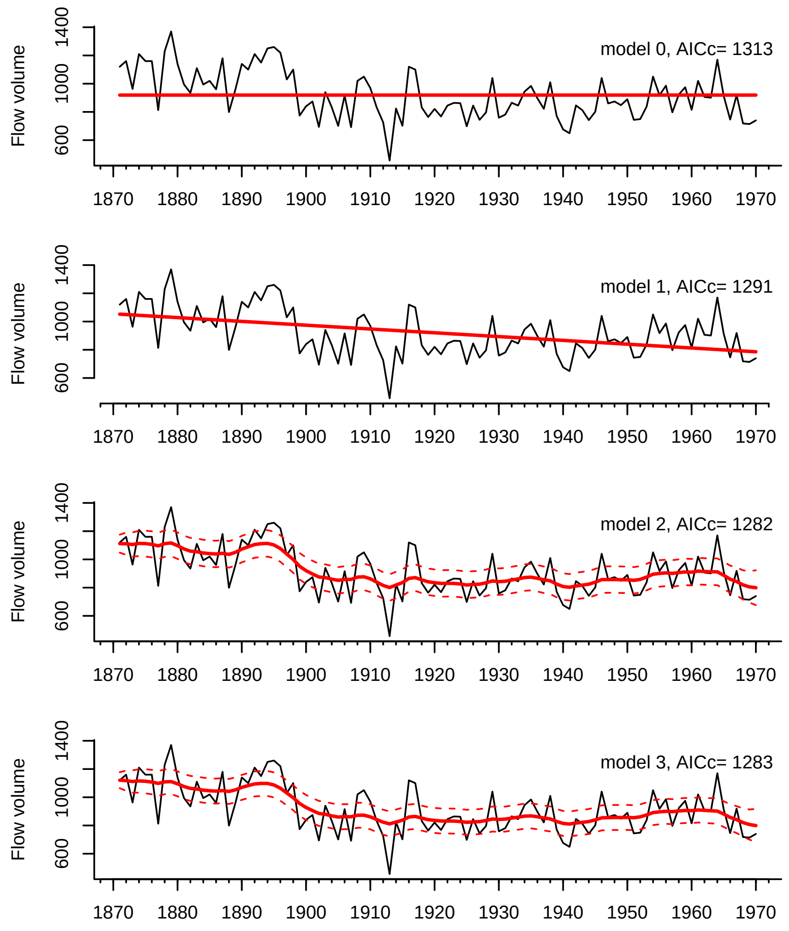


Figure 6.2: The Nile River flow volume with the model estimated flow rates (solid lines). The bottom model is a stochastic level model, meaning there isn't one level line. Rather the level line is a distribution that has a mean and standard deviation. The solid state line in the bottom plots is the mean of the stochastic level and the 2 standard deviations are shown. The other two models are deterministic level models so the state is not stochastic and does not have a standard deviation.

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

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

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

And this leads to our model weights table:

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

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

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

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

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

6.5 Basic diagnostics

The first diagnostic that you do with any statistical analysis is check that your residuals correspond to your assumed error structure. For a basic residuals diagnostic check for a state-space model, we want to use the 'innovations residuals'. This is the observed data at time t minus the value predicted using the model plus the data up to time $t-1$. Innovations residuals should be Gaussian and temporally independent (no autocorrelation). `residuals(fit)` will return the innovations residuals as a data frame.

```
head(residuals(kem.0))
```

```
  type .rownames  name    t value .fitted .resids   .sigma .std.resids
1 ytt1        Y1 model 1871  1120  919.35  200.65 168.3792   1.1916552
2 ytt1        Y1 model 1872  1160  919.35  240.65 168.3792   1.4292142
3 ytt1        Y1 model 1873   963  919.35   43.65 168.3792   0.2592362
4 ytt1        Y1 model 1874  1210  919.35  290.65 168.3792   1.7261629
5 ytt1        Y1 model 1875  1160  919.35  240.65 168.3792   1.4292142
6 ytt1        Y1 model 1876  1160  919.35  240.65 168.3792   1.4292142
```

The innovations residuals should also not be autocorrelated in time. We can check the autocorrelation with the function `acf()`. The autocorrelation plots are shown in Figure 6.3. The stochastic level model looks the best in that its innovations residuals are fine.

```
par(mfrow = c(2, 2), mar = c(2, 2, 4, 2))
resids <- residuals(kem.0)
acf(resids$.resids, main = "flat level v(t)", na.action = na.pass)
resids <- residuals(kem.1)
acf(resids$.resids, main = "linear trend v(t)", na.action = na.pass)
resids <- residuals(kem.2)
acf(resids$.resids, main = "stoc level v(t)", na.action = na.pass)
```

6.5.1 Outlier diagnostics

Another type of residual used in state-space models is smoothation residuals. This residual at time t is conditioned on all the data. Smoothation residu-

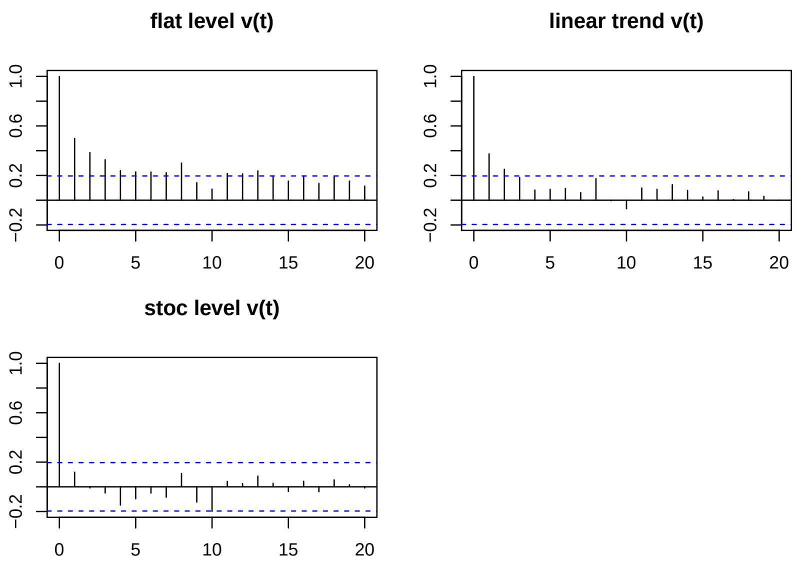


Figure 6.3: The model innovations residual acfs for the 3 models.

als are used for outlier detection and can help detect anomalous shocks in the data. Smoothation residuals can be autocorrelated but should fluctuate around 0. The should not have a trend. Looking at your smoothation residuals can help you determine if there are fundamental problems with the structure of your model.

We can get the smoothation residuals by passing in `type="tT"` to the `residuals()` call. Figure 6.4 shows the model and state smoothation residuals. The flat level and linear trend models do not have a stochastic state so their state residuals are all 0.

The flat and linear trend models show problems. The model smoothation residuals are positive early and then are slightly negative. They should fluctuate around 0 the whole time series. The stochastic level model looks fine. The residuals fluctuate around 0.

The smoothation residuals can also help us look for outliers in the data or outlier shifts in the level (sudden anolmalous changes). If we standardize by the variance of the residuals (divide by the square root of the variance), then the standardized residuals should have an approximate standard normal distribution.

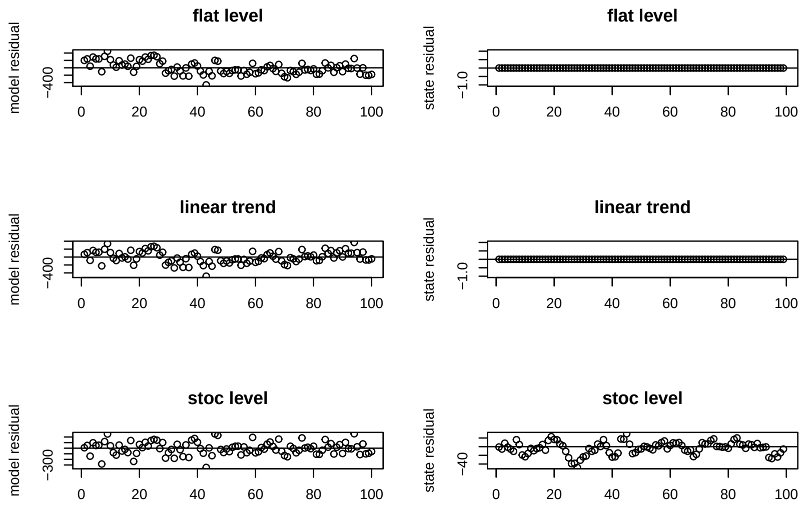


Figure 6.4: The model and state smoothations residuals for the first 3 models.

We will look just at the stochastic level model since the other models do not have a stochastic state (x). Figure 6.5 (right panel) shows us that there was a sudden level change around 1902. The Aswan Low Dam was completed in 1902 and changed the mean flow. The Aswan High Dam was completed in 1970 and also affected the flow though not as much. You can see these perturbations in Figure 6.1.

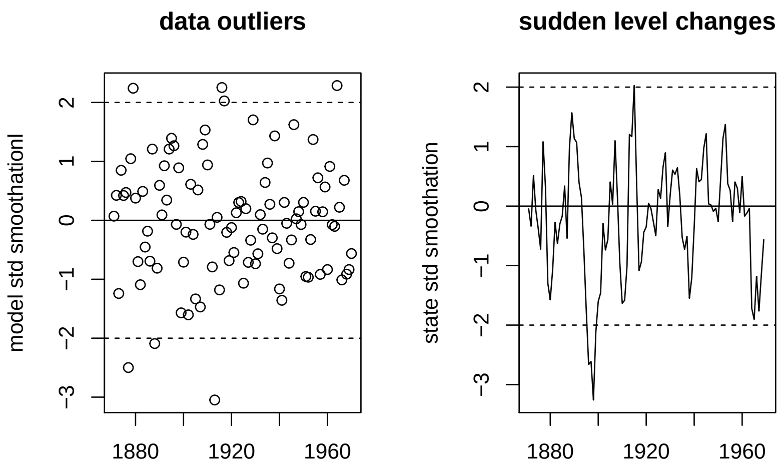


Figure 6.5: The model and state smoothations residuals for the first 3 models.

6.6 Fitting with JAGS

Here we show how to fit the stochastic level model, model 3 Equation (6.7), with JAGS. This is a model where the level is a random walk with drift and the Nile River flow is that level plus error.

```
library(datasets)
y <- Nile
```

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

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

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

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

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

   for(i in 2:TT) {
   predX[i] <- X[i-1]+u;
   X[i] ~ dnorm(predX[i],tau.q); # Process variation
   Y[i] ~ dnorm(X[i], tau.r); # Observation variation
   }
   }
   ",
    file = model.loc)
```

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

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

Now we can fit the model:

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

We can then show the posteriors along with the MLEs from MARSS on top (Figure 6.6) using the code below.

```
attach.jags(mod_ss)
```

```

The following objects in .GlobalEnv will mask
objects in the attached database:
mu
Remove these objects from .GlobalEnv?

1: YES
2: NO

Enter an item from the menu, or 0 to exit
```

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

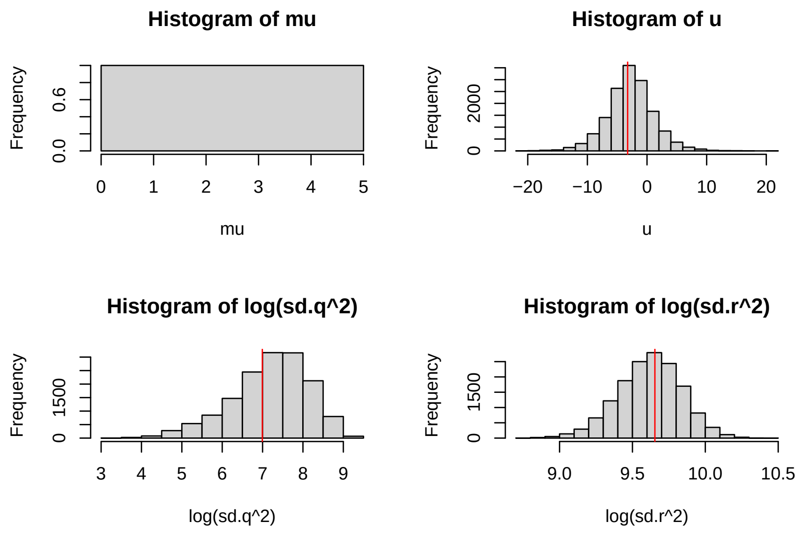


Figure 6.6: The posteriors for model 3 with MLE estimates from `MARSS()` shown in red.

```
detach.jags()
```

To plot the estimated states (Figure 6.7), we write a helper function:

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

```
plotModelOutput(mod_ss, y)
```

```
The following objects in .GlobalEnv will mask
objects in the attached database:
mu
Remove these objects from .GlobalEnv?

1: YES
2: NO

The following object is masked _by_ .GlobalEnv:

    mu
```

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

6.7 Fitting with Stan

Let's fit the same model with Stan using the **rstan** package. If you have not already, you will need to install the **rstan** package. This package depends

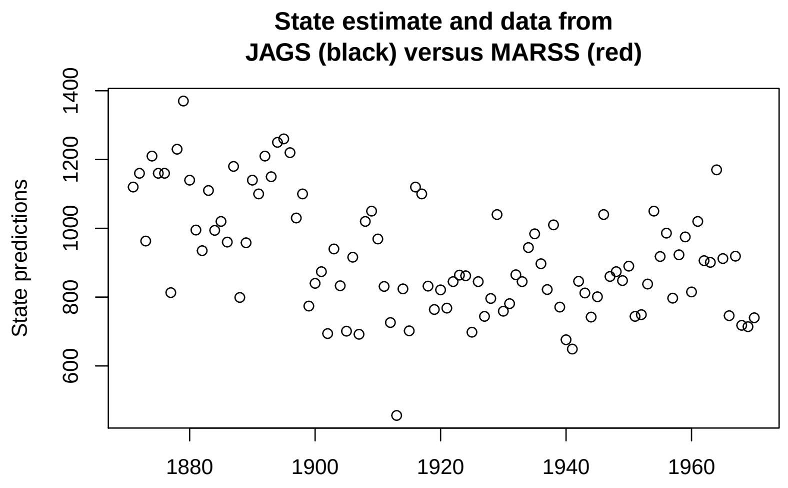


Figure 6.7: The estimated states from the Bayesian fit along with 95% credible intervals (black and grey) with the MLE states and 95% condidence intervals in red.

on a number of other packages which should install automatically when you install **rstan**.

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

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

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

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

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

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

```
pars <- rstan::extract(mod)
pred_mean <- apply(pars$x, 2, mean)
pred_lo <- apply(pars$x, 2, quantile, 0.025)
pred_hi <- apply(pars$x, 2, quantile, 0.975)
plot(pred_mean, type = "l", lwd = 3, ylim = range(c(pred_mean,
    pred_lo, pred_hi)), ylab = "Nile River Level")
lines(pred_lo)
lines(pred_hi)
points(y, col = "blue")
```

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

```
library(ggplot2)
nile <- data.frame(y = y, year = 1871:1970)
h <- ggplot(nile, aes(year))
h + geom_ribbon(aes(ymin = pred_lo, ymax = pred_hi), fill = "grey70") +
    geom_line(aes(y = pred_mean), size = 1) + geom_point(aes(y = y),
    color = "blue") + labs(y = "Nile River level")
```

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

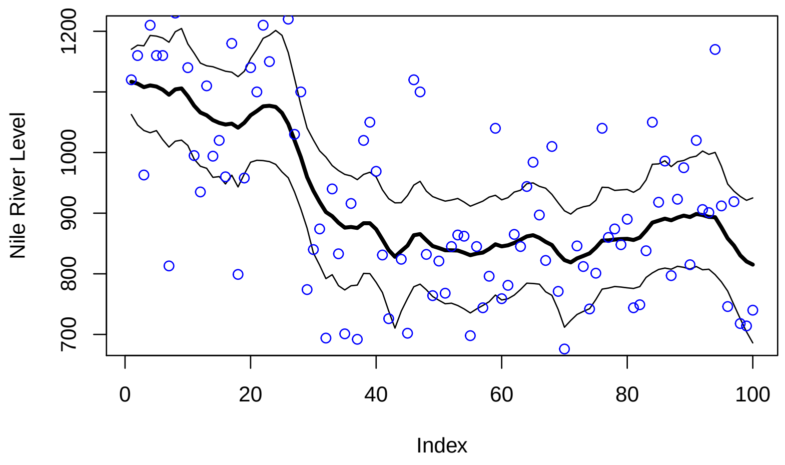


Figure 6.8: Estimated level and 95 percent credible intervals. Blue dots are the actual Nile River levels.

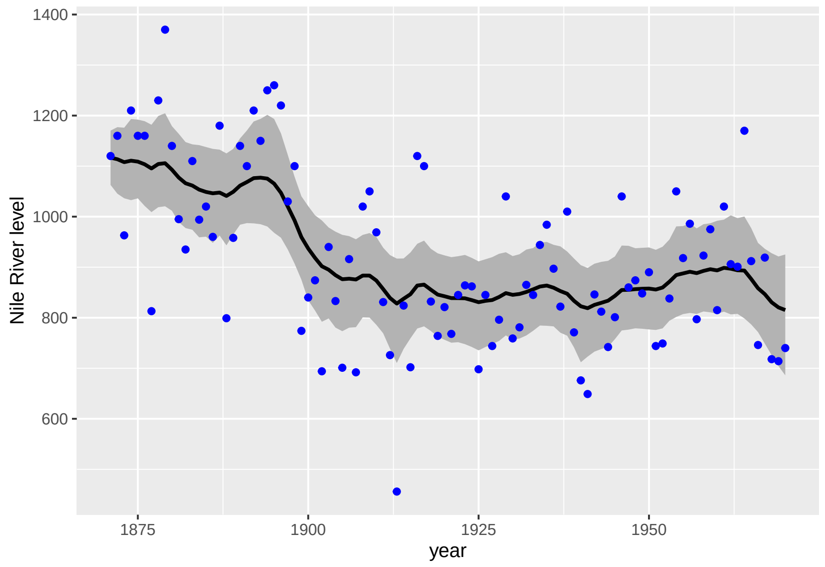


Figure 6.9: Estimated level and 95 percent credible intervals

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

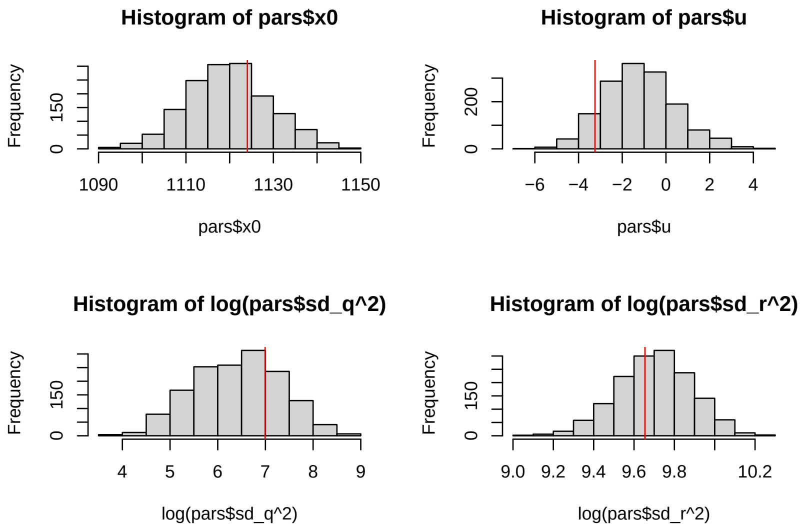


Figure 6.10: Histogram of the parameter samples versus the estimate (red line) from maximum likelihood.

6.8 A random walk model of animal movement

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

$$x_{1,t} = x_{1,t-1} + u_1 + w_{1,t}, \quad w_{1,t} \sim \text{N}(0, \sigma_1^2) \tag{6.8}$$
$$x_{2,t} = x_{2,t-1} + u_2 + w_{2,t}, \quad w_{2,t} \sim \text{N}(0, \sigma_2^2) \tag{6.9}$$

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

$$y_{1,t} = x_{1,t} + v_{1,t}, \quad v_{1,t} \sim \text{N}(0, \eta_1^2) \tag{6.10}$$
$$y_{2,t} = x_{2,t} + v_{2,t}, \quad v_{2,t} \sim \text{N}(0, \eta_2^2), \tag{6.11}$$

This model is comprised of two separate univariate state-space models. Note that y_1 depends only on x_1 and y_2 depends only on x_2. There are no actual interactions between these two univariate models. However, we can write the model down in the form of a multivariate model using diagonal variance-covariance matrices and a diagonal design ($\mathbf{Z}$) matrix. Because the variance-covariance matrices and $\mathbf{Z}$ are diagonal, the x_1:y_1 and x_2:y_2 processes will be independent as intended. Here are Equations (6.9) and (6.11) written as a MARSS model (in matrix form):

$$\begin{bmatrix}x_{1,t}\\x_{2,t}\end{bmatrix} = \begin{bmatrix}x_{1,t-1}\\x_{2,t-1}\end{bmatrix} + \begin{bmatrix}u_1\\u_2\end{bmatrix} + \begin{bmatrix}w_{1,t}\\w_{2,t}\end{bmatrix}, \ \mathbf{w}_t \sim \text{MVN}\left(0, \begin{bmatrix}\sigma_1^2 & 0\\0 & \sigma_2^2\end{bmatrix}\right) \tag{6.12}$$

$$\begin{bmatrix}y_{1,t}\\y_{2,t}\end{bmatrix} = \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}\begin{bmatrix}x_{1,t}\\x_{2,t}\end{bmatrix} + \begin{bmatrix}v_{1,t}\\v_{2,t}\end{bmatrix}, \ \mathbf{v}_t \sim \text{MVN}\left(0, \begin{bmatrix}\eta_1^2 & 0\\0 & \eta_2^2\end{bmatrix}\right) \tag{6.13}$$

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

$$\mathbf{x}_t = \mathbf{x}_{t-1} + \mathbf{u} + \mathbf{w}_t, \quad \mathbf{w}_t \sim \text{MVN}(0, \mathbf{Q}) \tag{6.14}$$
$$\mathbf{y}_t = \mathbf{x}_t + \mathbf{v}_t, \quad \mathbf{v}_t \sim \text{MVN}(0, \mathbf{R}). \tag{6.15}$$

6.9 Problems

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

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

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

Consider these two models for the data:

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

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

$$\begin{gathered} x_t = x_{t-1} + w_t \text{ where } w_t \sim \mathrm{N}(0,q) \\ x_0 = a \\ y_t = x_t \end{gathered} \tag{6.16}$$

Model 2 is almost identical except with u added

$$\begin{gathered} x_t = x_{t-1} + u + w_t \text{ where } w_t \sim \mathrm{N}(0,q) \\ x_0 = a \\ y_t = x_t \end{gathered} \tag{6.17}$$

y is the log grouse count in year t.

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

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

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

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

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

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

a. What is the order of this random walk written as ARIMA(p, d, q)? "what is the order" means "what is p, d, and q. Model"order" is how `arima()` and `Arima()` specify arima models.
b. Fit that model using `Arima()` in the **forecast** package. You'll need to specify the arguments `order` and `include.drift`. Use `?Arima` to review what that function does if needed.
c. Write out the equation for this random walk as a univariate state-space model. Notice that there is no observation error, but still write this as a state-space model.
d. Fit that model with `MARSS()`.
e. How are the two estimates from `Arima()` and `MARSS()` different?

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

```
diff.dat <- diff(dat)
```

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

a. If x_t denotes a time series. What is the first difference of x? What is the second difference?
b. What is the **x** model for `diff.dat`? Look at your answer to part (a) and the answer to part (e).
c. Fit `diff.dat` using `Arima()`. You'll need to change the arguments `order` and `include.mean`.
d. Fit with `MARSS()`. You will need to write the model for `diff.dat` as a state-space model. If you've done this right, the estimated parameters using `Arima()` and `MARSS()` will now be the same.

This question should clue you into the fact that `Arima()` is not exactly fitting Equation (6.1). It's very similar, but not quite written that way. By the way, Equation (6.1) is how structural time series observed with error are written (state-space models). To recover the estimates that a function like `arima()` or `Arima()` returns, you need to write your state-space model in a specific way (as seen above).

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

$$x_t = bx_{t-1} + u + w_t \text{ where } w_t \sim \mathrm{N}(0, q) \tag{6.18}$$

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

a. Write R code to simulate Equation (6.18). Make b less than 1 and greater than 0. Set u and x_0 to whatever you want. You can use a for loop.
b. Plot the trajectories and show that this model does not "drift" upward or downward. It fluctuates about a mean value.
c. Hold b constant and change u. How do the trajectories change?
d. Hold u constant and change b. Make sure to use a b close to 1 and another close to 0. How do the trajectories change?

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

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

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

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

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

$$\begin{gathered}
x_t = x_{t-1} + u + w_t \text{ where } w_t \sim \text{N}(0,q) \\
y_t = x_t + v_t \text{ where } v_t \sim \text{N}(0,r) \\
x_0 = a
\end{gathered} \tag{6.19}$$

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

a. Fit this model with `MARSS()`
b. Plot the estimated x as a line with the actual counts added as points. x is in `fit$states`. It is a matrix. To plot using `plot()`, you will need to change it to a vector using `as.vector()` or `fit$states[1,]`
c. Simulate 1000 sample gray whale populstion trajectories (the x in your model) using the estimated u and q starting at the estimated x in 1997. You can do this with a couple for loops or write something terse with `cumsum()` and `apply()`.
d. Using these simulated trajectories, what is your estimate of the probability that the grey whale population will be above 50,000 graywhales in 2007?
e. What kind(s) of uncertainty does your estimate above NOT include?

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

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

   a. Compute the AICc's for each model and likelihood or deviance (-2 * log likelihood). Where to find these? Try `names(fit)`. `logLik()` is the standard R function to return log-likelihood from fits.
   b. Calculate a table of ΔAICc values and AICc weights.
   c. Show the acf of the model and state residuals for the best model. You will need a vector of the residuals to do this. If `fit` is the fit from a fit call like `fit = MARSS(dat)`, you get the residuals using this code:

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

   Do the acf's suggest any problems?

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

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

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

a. Fit the following four models using `Arima()`: ARIMA(0,0,0), ARIMA(1,0,0), ARIMA(0,0,1), ARIMA(1,0,1).
b. Use `forecast()` to make 3 step ahead forecasts from each.
c. Calculate the MASE statistic for each using the `accuracy()` function in the **forecast** package. Type `?accuracy` to learn how to use this function.
d. Present the results in a table.
e. Which model is best supported based on the MASE statistic?

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

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

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

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

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

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

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

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

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

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

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

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

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

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

   The lon residuals are in row 1 and lat residuals are in row 2 (same order as the data).

Chapter 7

MARSS models

This lab will show you how to fit multivariate state-space (MARSS) models using the **MARSS** package. This class of time-series model is also called vector autoregressive state-space (VARSS) models. This chapter works through an example which uses model selection to test different population structures in west coast harbor seals. See Holmes et al. (2014) for a fuller version of this example.

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

Data and packages

All the data used in the chapter are in the **MARSS** package. For most examples, we will use the `MARSS()` function to fit models via maximum-likelihood. We also show how to fit a Bayesian model using JAGS and Stan. For these sectiosn you will need the **R2jags**, **coda** and **rstan** packages. To run the JAGS code, you will also need JAGS installed. See Chapter 12 for more details on JAGS and Chapter 13 for more details on Stan.

```
library(MARSS)
library(R2jags)
library(coda)
library(rstan)
```

7.1 Overview

As discussed in Chapter 6, the **MARSS** package fits multivariate state-space models in this form:

$$\begin{aligned}\mathbf{x}_t &= \mathbf{B}\mathbf{x}_{t-1} + \mathbf{u} + \mathbf{w}_t \text{ where } \mathbf{w}_t \sim \text{N}(0, \mathbf{Q}) \\ \mathbf{y}_t &= \mathbf{Z}\mathbf{x}_t + \mathbf{a} + \mathbf{v}_t \text{ where } \mathbf{v}_t \sim \text{N}(0, \mathbf{R}) \\ \mathbf{x}_0 &= \boldsymbol{\mu}\end{aligned} \tag{7.1}$$

where each of the bolded terms are matrices. Those that are bolded and small (not capitalized) have one column only, so are column matrices.

To fit a multivariate time series model with the **MARSS** package, you need to first determine the size and structure of each of the parameter matrices: $\mathbf{B}$, $\mathbf{u}$, $\mathbf{Q}$, $\mathbf{Z}$, $\mathbf{a}$, $\mathbf{R}$ and $\boldsymbol{\mu}$. This requires first writing down your model in matrix form. We will illustarte this with a series of models for the temporal population dynamics of West coast harbor seals.

7.2 West coast harbor seals counts

In this example, we will use multivariate state-space models to combine surveys from four survey regions to estimate the average long-term population growth rate and the year-to-year variability in that population growth rate.

We have five regions (or sites) where harbor seals were censused from 1978-1999 while hauled out of land[1]. During the period of this dataset, harbor seals were recovering steadily after having been reduced to low levels by hunting prior to protection. We will assume that the underlying population process is a stochastic exponential growth process with mean rates of increase that were not changing through 1978-1999.

The survey methodologies were consistent throughout the 20 years of the data but we do not know what fraction of the population that each region represents nor do we know the observation-error variance for each region. Given differences between the numbers of haul-outs in each region, the observation errors may be quite different. The regions have had different levels

[1]Jeffries et al. 2003. Trends and status of harbor seals in Washington State: 1978-1999. Journal of Wildlife Management 67(1):208–219

of sampling; the best sampled region has only 4 years missing while the worst has over half the years missing (Figure 7.1).

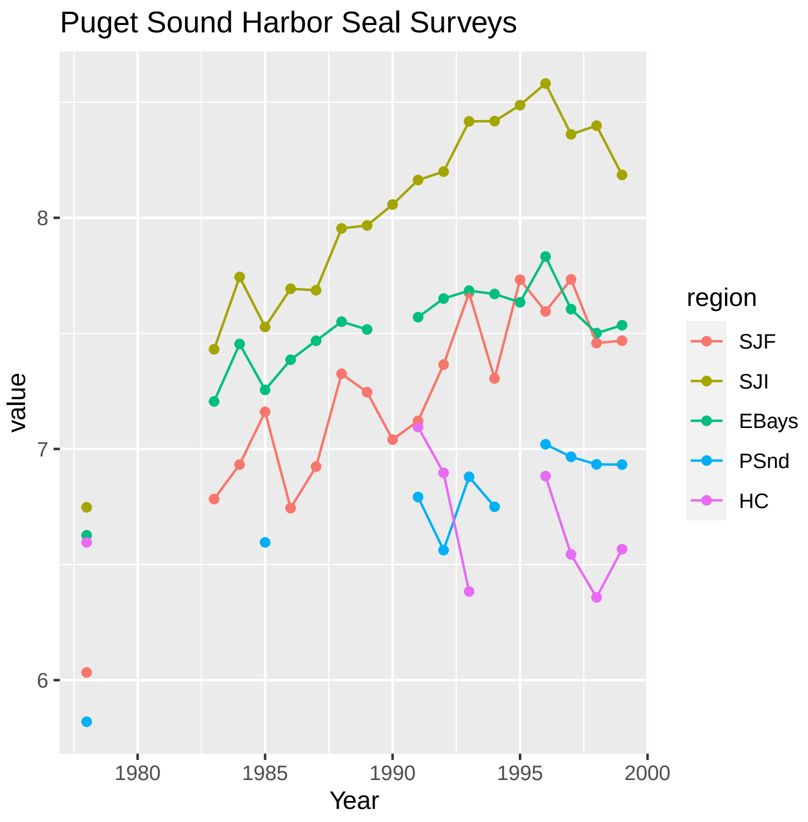


Figure 7.1: Plot of the of the count data from the five harbor seal regions (Jeffries et al. 2003). The numbers on each line denote the different regions: 1) Strait of Juan de Fuca (SJF), 2) San Juan Islands (SJI), 2) Eastern Bays (EBays), 4) Puget Sound (PSnd), and 5) Hood Canal (HC). Each region is an index of the total harbor seal population in each region.

7.2.1 Load the harbor seal data

The harbor seal data are included in the **MARSS** package as matrix with years in column 1 and the logged counts in the other columns. Let's look at the first few years of data:

```
data(harborSealWA, package = "MARSS")
print(harborSealWA[1:8, ], digits = 3)
```

```
     Year  SJF  SJI EBays PSnd  HC
[1,] 1978 6.03 6.75  6.63 5.82 6.6
[2,] 1979   NA   NA    NA   NA  NA
[3,] 1980   NA   NA    NA   NA  NA
[4,] 1981   NA   NA    NA   NA  NA
[5,] 1982   NA   NA    NA   NA  NA
[6,] 1983 6.78 7.43  7.21   NA  NA
[7,] 1984 6.93 7.74  7.45   NA  NA
[8,] 1985 7.16 7.53  7.26 6.60  NA
```

We are going to leave out Hood Canal (HC) since that region is somewhat isolated from the others and experiencing very different conditions due to hypoxic events and periodic intense killer whale predation. We will set up the data as follows:

```
dat <- MARSS::harborSealWA
years <- dat[, "Year"]
dat <- dat[, !(colnames(dat) %in% c("Year", "HC"))]
dat <- t(dat)  # transpose to have years across columns
colnames(dat) <- years
n <- nrow(dat) - 1
```

7.3 A single well-mixed population

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

$$n_t = \exp(u + w_t) n_{t-1}, \tag{7.2}$$

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

$$x_t = x_{t-1} + u + w_t, \text{ where } w_t \sim \mathrm{N}(0, q) \tag{7.3}$$

$x_t = \log n_t$. When there is one effective population, there is one x, therefore $\mathbf{x}_t$ is a 1×1 matrix. This is our **state** model and x is called the "state". This is just the jargon used in this type of model (state-space model) for the hidden state that you are estimating from the data. "Hidden" means that you observe this state with error.

7.3.1 The observation process

We assume that all four regional time series are observations of this one population trajectory but they are scaled up or down relative to that trajectory. In effect, we think of each regional survey as an index of the total population. With this model, we do not think the regions represent independent subpopulations but rather independent observations of one population. Our model for the data, $\mathbf{y}_t = \mathbf{Z}\mathbf{x}_t + \mathbf{a} + \mathbf{v}_t$, is written as:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix}_t = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} x_t + \begin{bmatrix} 0 \\ a_2 \\ a_3 \\ a_4 \end{bmatrix} + \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix}_t \tag{7.4}$$

Each y_i is the observed time series of counts for a different region. The a's are the bias between the regional sample and the total population. $\mathbf{Z}$ specifies which observation time series, y_i, is associated with which population trajectory, x_j. In this case, $\mathbf{Z}$ is a matrix with 1 column since each region is an observation of the one population trajectory.

We allow that each region could have a unique observation variance and that the observation errors are independent between regions. We assume that the observations errors on log(counts) are normal and thus the errors on (counts) are log-normal. The assumption of normality is not unreasonable since these regional counts are the sum of counts across multiple haul-outs. We specify independent observation errors with different variances by specifying that

$\mathbf{v} \sim \text{MVN}(0, \mathbf{R})$, where

$$\mathbf{R} = \begin{bmatrix} r_1 & 0 & 0 & 0 \\ 0 & r_2 & 0 & 0 \\ 0 & 0 & r_3 & 0 \\ 0 & 0 & 0 & r_4 \end{bmatrix} \tag{7.5}$$

This is a diagonal matrix with unequal variances. The shortcut for this structure in `MARSS()` is `"diagonal and unequal"`.

7.3.2 Fitting the model

We need to write the model in the form of Equation (7.1) with each parameter written as a matrix. The observation model (Equation (7.4)) is already in matrix form. Let's write the state model in matrix form too:

$$[x]_t = [1][x]_{t-1} + [u] + [w]_t, \text{ where } [w]_t \sim \text{N}(0, [q]) \tag{7.6}$$

It is very simple since all terms are 1×1 matrices.

To fit our model with `MARSS()`, we set up a list which precisely describes the size and structure of each parameter matrix. Fixed values in a matrix are designated with their numeric value and estimated values are given a character name and put in quotes. Our model list for a single well-mixed population is:

```
mod.list.0 <- list(B = matrix(1), U = matrix("u"), Q = matrix("q"),
    Z = matrix(1, 4, 1), A = "scaling", R = "diagonal and unequal",
    x0 = matrix("mu"), tinitx = 0)
```

and fit:

```
fit.0 <- MARSS(dat, model = mod.list.0)
```

```
Success! abstol and log-log tests passed at 32 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 32 iterations.
Log-likelihood: 21.62931
AIC: -23.25863   AICc: -19.02786

                Estimate
A.SJI            0.79583
A.EBays          0.27528
A.PSnd          -0.54335
R.(SJF,SJF)      0.02883
R.(SJI,SJI)      0.03063
R.(EBays,EBays)  0.01661
R.(PSnd,PSnd)    0.01168
U.u              0.05537
Q.q              0.00642
x0.mu            6.22810
Initial states (x0) defined at t=0

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

We already discussed that the short-cut `"diagonal and unequal"` means a diagonal matrix with each diagonal element having a different value. The short-cut `"scaling"` means the form of $\mathbf{a}$ in Equation (7.4) with one value set to 0 and the rest estimated. You should run the code in the list to make sure you see that each parameter in the list has the same form as in our mathematical equation for the model.

7.3.3 Model residuals

The model fits fine but look at the model residuals (Figure 7.2). They have problems.

```
par(mfrow = c(2, 2))
resids <- MARSSresiduals(fit.0, type = "tt1")
for (i in 1:4) {
    plot(resids$model.residuals[i, ], ylab = "model residuals",
        xlab = "")
    abline(h = 0)
    title(rownames(dat)[i])
}
```

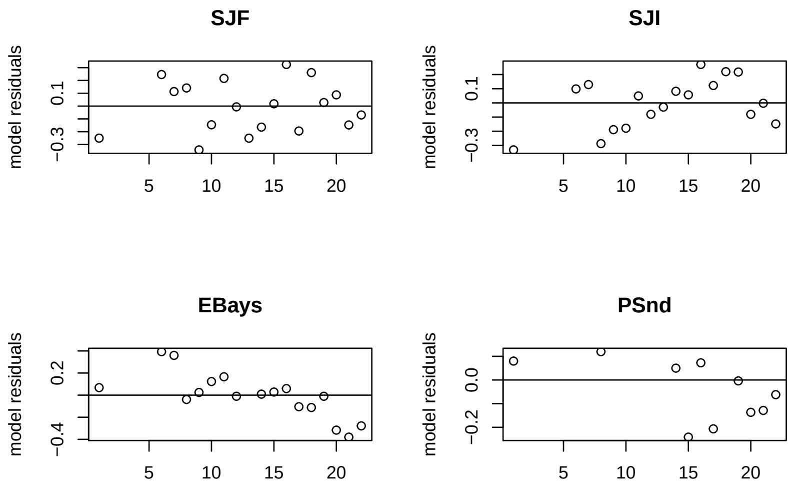


Figure 7.2: The model residuals for the first model. SJI and EBays do not look good.

7.4 Four subpopulations with temporally uncorrelated errors

The model for one well-mixed population was not very good. Another reasonable assumption is that the different census regions are measuring four different temporally independent subpopulations. We write a model of the log subpopulation abundances for this case as:

$$
\begin{gathered}
\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}_t = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}_{t-1} + \begin{bmatrix} u \\ u \\ u \\ u \end{bmatrix} + \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \end{bmatrix}_t \\
\text{where } \mathbf{w}_t \sim \text{MVN}\left(0, \begin{bmatrix} q & 0 & 0 & 0 \\ 0 & q & 0 & 0 \\ 0 & 0 & q & 0 \\ 0 & 0 & 0 & q \end{bmatrix}\right) \\
\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}_0 = \begin{bmatrix} \mu_1 \\ \mu_2 \\ \mu_3 \\ \mu_4 \end{bmatrix}_t
\end{gathered}
\tag{7.7}
$$

The **Q** matrix is diagonal with one variance value. This means that the process variance (variance in year-to-year population growth rates) is independent (good and bad years are not correlated) but the level of variability is the same across regions. We made the **u** matrix with one u value. This means that we assume the population growth rates are the same across regions.

Notice that we set the **B** matrix equal to a diagonal matrix with 1 on the diagonal. This is the "identity" matrix and it is like a 1 but for matrices. We do not need **B** for our model, but `MARSS()` requires a value.

7.4.1 The observation process

In this model, each survey is an observation of a different x:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix}_t = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}_t + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix}_t \tag{7.8}$$

No a's can be estimated since we do not have multiple observations of a given x time series. Our $\mathbf{R}$ matrix doesn't change; the observation errors are still assumed to the independent with different variances.

Notice that our $\mathbf{Z}$ matrix changed. $\mathbf{Z}$ is specifying which y_i goes to which x_j. The one we have specified means that y_1 is observing x_1, y_2 observes x_2, etc. We could have set up $\mathbf{Z}$ like so

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{7.9}$$

This would mean that y_1 observes x_2, y_2 observes x_1, y_3 observes x_4, and y_4 observes x_3. Which x goes to which y is arbitrary; we need to make sure it is one-to-one. We will stay with $\mathbf{Z}$ as an identity matrix since y_i observing x_i makes it easier to remember which x goes with which y.

7.4.2 Fitting the model

We set up the model list for `MARSS()` as:

```r
mod.list.1 <- list(B = "identity", U = "equal", Q = "diagonal and equal",
    Z = "identity", A = "scaling", R = "diagonal and unequal",
    x0 = "unequal", tinitx = 0)
```

We introduced a few more short-cuts. `"equal"` means all the values in the matrix are the same. `"diagonal and equal"` means that the matrix is diagonal with one value on the diagonal. `"unequal"` means that all values in the matrix are different.

We can then fit our model for 4 subpopulations as:

```
fit.1 <- MARSS::MARSS(dat, model = mod.list.1)
```

7.5 Four subpopulations with temporally correlated errors

Another reasonable assumption is that the different census regions are measuring different subpopulations but that the year-to-year population growth rates are correlated (good and bad year coincide). The only parameter that changes is the **Q** matrix:

$$\mathbf{Q} = \begin{bmatrix} q & c & c & c \\ c & q & c & c \\ c & c & q & c \\ c & c & c & q \end{bmatrix} \tag{7.10}$$

This **Q** matrix structure means that the process variance (variance in year-to-year population growth rates) is the same across regions and the covariance in year-to-year population growth rates is also the same across regions.

7.5.1 Fitting the model

Set up the model list for `MARSS()` as:

```
mod.list.2 <- mod.list.1
mod.list.2$Q <- "equalvarcov"
```

`"equalvarcov"` is a shortcut for the matrix form in Equation (7.10).

Fit the model with:

```
fit.2 <- MARSS::MARSS(dat, model = mod.list.2)
```

Results are not shown, but here are the AICc. This last model is much better:

```
c(fit.0$AICc, fit.1$AICc, fit.2$AICc)
```

```
[1] -19.02786 -22.20194 -41.00511
```

7.5.2 Model residuals

Look at the model residuals (Figure 7.3). They are also much better.

```
MARSSresiduals.tt1 reported warnings. See msg element of returned residuals ob
```

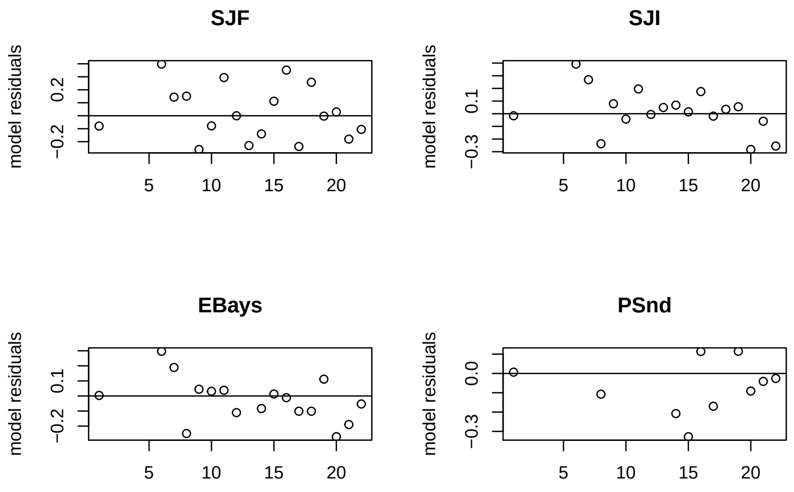


Figure 7.3: The model residuals for the model with four temporally correlated subpopulations.

Figure 7.4 shows the estimated states for each region using this code:

```
par(mfrow = c(2, 2))
for (i in 1:4) {
    plot(years, fit.2$states[i, ], ylab = "log subpopulation estimate",
        xlab = "", type = "l")
    lines(years, fit.2$states[i, ] - 1.96 * fit.2$states.se[i,
        ], type = "l", lwd = 1, lty = 2, col = "red")
    lines(years, fit.2$states[i, ] + 1.96 * fit.2$states.se[i,
        ], type = "l", lwd = 1, lty = 2, col = "red")
    title(rownames(dat)[i])
}
```

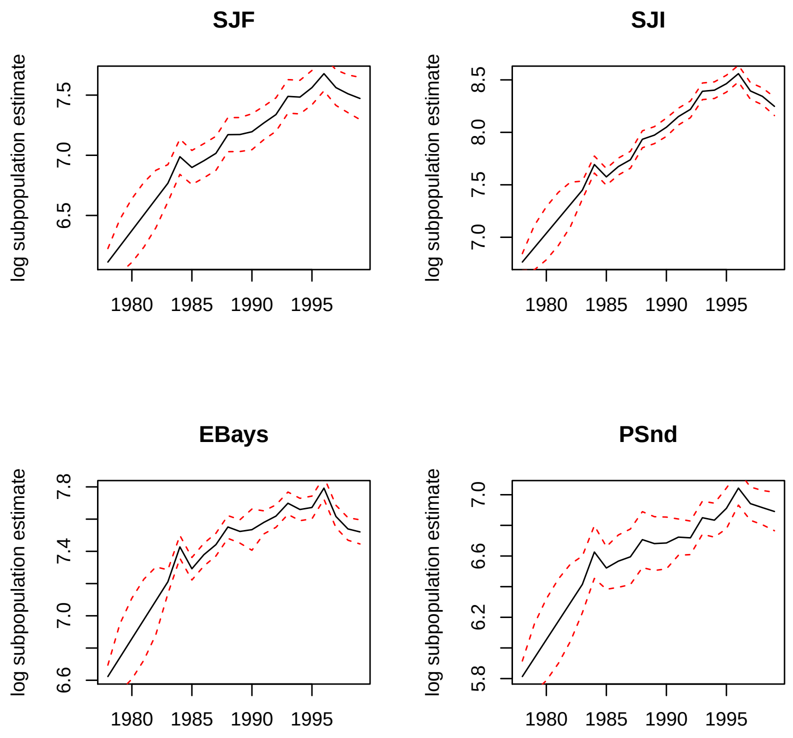


Figure 7.4: Plot of the estimate of log harbor seals in each region. The 95% confidence intervals on the population estimates are the dashed lines. These are not the confidence intervals on the observations, and the observations (the numbers) will not fall between the confidence interval lines.

7.6 Using MARSS models to study spatial structure

For our next example, we will use MARSS models to test hypotheses about the population structure of harbor seals on the west coast. For this example, we will evaluate the support for different population structures (numbers of subpopulations) using different $\mathbf{Z}$s to specify how survey regions map onto subpopulations. We will assume correlated process errors with the same magnitude of process variance and covariance. We will assume independent observations errors with equal variances at each site. We could do unequal variances but it takes a long time to fit so for this example, the observation variances are set equal.

The dataset we will use is `harborSeal`, a 29-year dataset of abundance indices for 12 regions along the U.S. west coast between 1975-2004 (Figure 7.5).

We start by setting up our data matrix. We will leave off Hood Canal.

```
dat <- MARSS::harborSeal
years <- dat[, "Year"]
good <- !(colnames(dat) %in% c("Year", "HoodCanal"))
sealData <- t(dat[, good])
```

7.7 Hypotheses regarding spatial structure

We will evaluate the data support for the following hypotheses about the population structure:

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

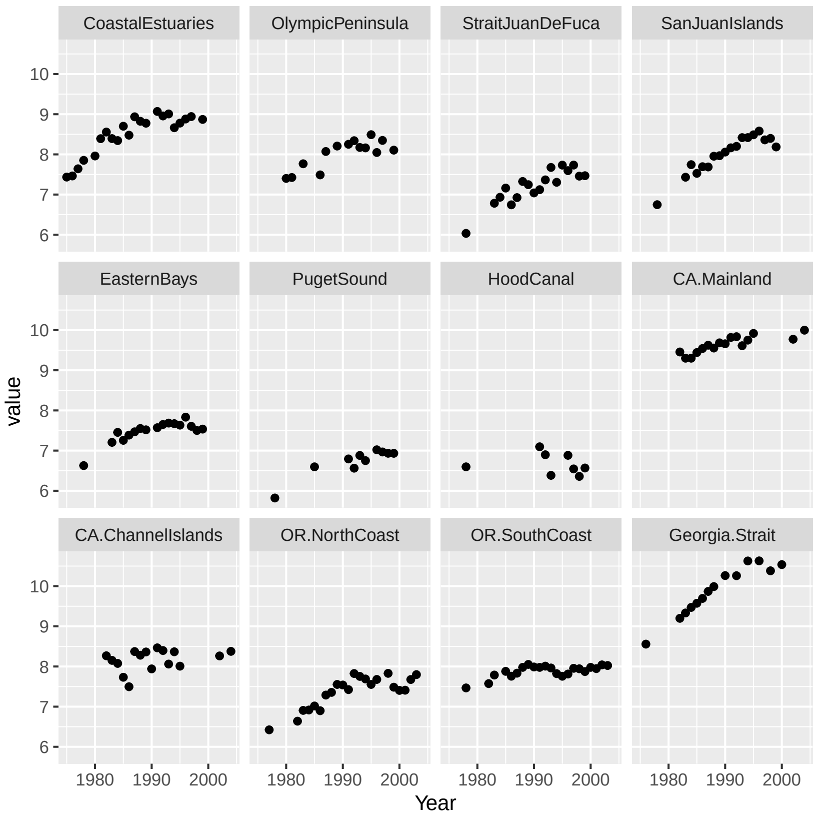


Figure 7.5: Plot of log counts at each survey region in the harborSeal dataset. Each region is an index of the harbor seal abundance in that region.

These hypotheses translate to these **Z** matrices (H6 not shown; it is an identity matrix):

$$\begin{array}{r|ccc|cc|cccc|c}
 & \multicolumn{3}{c}{H1} & \multicolumn{2}{c}{H2} & \multicolumn{4}{c}{H4} & H5 \\
 & \text{pnw} & \text{ps} & \text{ca} & \text{coast} & \text{pc} & \text{nc} & \text{is} & \text{ps} & \text{sc} & \text{pan} \\
\hline
\text{Coastal Estuaries} & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\
\text{Olympic Peninsula} & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\
\text{Str. Juan de Fuca} & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\
\text{San Juan Islands} & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\
\text{Eastern Bays} & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\
\text{Puget Sound} & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\
\text{CA Mainland} & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\
\text{CA Channel Islands} & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\
\text{OR North Coast} & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\
\text{OR South Coast} & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\
\text{Georgia Strait} & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1
\end{array}$$

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

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

Or we can use a short-cut by specifying **Z** as a factor that has the name of the subpopulation associated with each row in **y**. For hypothesis 1, this is

```r
Z1 <- factor(c("pnw", "pnw", rep("ps", 4), "ca", "ca", "pnw",
    "pnw", "ps"))
```

Notice it is 11 elements in length; one element for each row of data.

7.8 Set up the hypotheses as different models

Only the **Z** matrices change for our model. We will set up a base model list used for all models.

```
mod.list <- list(
  B = "identity",
  U = "unequal",
  Q = "equalvarcov",
  Z = "placeholder",
  A = "scaling",
  R = "diagonal and equal",
  x0 = "unequal",
  tinitx = 0
)
```

Then we set up the **Z** matrices using the factor short-cut.

```
Z.models <- list(
  H1 = factor(c("pnw", "pnw", rep("ps", 4), "ca", "ca", "pnw", "pnw", "ps")),
  H2 = factor(c(rep("coast", 2), rep("ps", 4), rep("coast", 4), "ps")),
  H3 = factor(c(rep("N", 6), "S", "S", "N", "S", "N")),
  H4 = factor(c("nc", "nc", "is", "is", "ps", "ps", "sc", "sc", "nc", "sc", "is")),
  H5 = factor(rep("pan", 11)),
  H6 = factor(1:11) # site
)
names(Z.models) <-
  c("stock", "coast+PS", "N+S", "NC+strait+PS+SC", "panmictic", "site")
```

7.8.1 Fit the models

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

```
out.tab <- NULL
fits <- list()
for (i in 1:length(Z.models)) {
    mod.list$Z <- Z.models[[i]]
    fit <- MARSS::MARSS(sealData, model = mod.list, silent = TRUE,
        control = list(maxit = 1000))
    out <- data.frame(H = names(Z.models)[i], logLik = fit$logLik,
        AICc = fit$AICc, num.param = fit$num.params, m = length(unique(Z.models[[i]]))
        num.iter = fit$numIter, converged = !fit$convergence)
    out.tab <- rbind(out.tab, out)
    fits <- c(fits, list(fit))
}
```

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

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

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

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

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

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

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

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

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

```
               H delta.AICc AIC.weight converged
 NC+strait+PS+SC       0.00      0.979      TRUE
            site       7.65      0.021      TRUE
             N+S      36.97      0.000      TRUE
           stock      47.02      0.000      TRUE
        coast+PS      48.78      0.000      TRUE
       panmictic      71.67      0.000      TRUE
```

7.9 Fitting a MARSS model with JAGS

Here we show you how to fit a MARSS model for the harbor seal data using JAGS. We will focus on four time series from inland Washington and set up the data as follows:

```
data(harborSealWA, package = "MARSS")
sites <- c("SJF", "SJI", "EBays", "PSnd")
Y <- harborSealWA[, sites]
Y <- t(Y)  # time across columns
```

We will fit the model with four temporally independent subpopulations with the same population growth rate (u) and year-to-year variance (q). This is the model in Section 7.4.

7.9.1 Writing the model in JAGS

The first step is to write this model in JAGS. See Chapter 12 for more information on and examples of JAGS models.

```
jagsscript <- cat("
model {
   U ~ dnorm(0, 0.01);
   tauQ~dgamma(0.001,0.001);
   Q <- 1/tauQ;

   # Estimate the initial state vector of population abundances
   for(i in 1:nSites) {
      X[i,1] ~ dnorm(3,0.01); # vague normal prior
   }

   # Autoregressive process for remaining years
   for(t in 2:nYears) {
      for(i in 1:nSites) {
         predX[i,t] <- X[i,t-1] + U;
         X[i,t] ~ dnorm(predX[i,t], tauQ);
      }
   }

   # Observation model
   # The Rs are different in each site
   for(i in 1:nSites) {
     tauR[i]~dgamma(0.001,0.001);
     R[i] <- 1/tauR[i];
   }
   for(t in 1:nYears) {
     for(i in 1:nSites) {
       Y[i,t] ~ dnorm(X[i,t],tauR[i]);
     }
   }
}

",
    file = "marss-jags.txt")
```

7.9.2 Fit the JAGS model

{#sec-mss-fit-jags}

Then we write the data list, parameter list, and pass the model to the `jags()` function:

```
jags.data <- list(Y = Y, nSites = nrow(Y), nYears = ncol(Y))  # named list
jags.params <- c("X", "U", "Q", "R")
model.loc <- "marss-jags.txt"  # name of the txt file
mod_1 <- jags(jags.data, parameters.to.save = jags.params, model.file = model.
    n.chains = 3, n.burnin = 5000, n.thin = 1, n.iter = 10000,
    DIC = TRUE)
```

7.9.3 Plot the posteriors for the estimated states

We can plot any of the variables we chose to return to R in the `jags.params` list. Let's focus on the `X`. When we look at the dimension of the `X`, we can use the `apply()` function to calculate the means and 95 percent CIs of the estimated states.

```
# attach.jags attaches the jags.params to our workspace
attach.jags(mod_1)
means <- apply(X, c(2, 3), mean)
upperCI <- apply(X, c(2, 3), quantile, 0.975)
lowerCI <- apply(X, c(2, 3), quantile, 0.025)
par(mfrow = c(2, 2))
nYears <- ncol(Y)
for (i in 1:nrow(means)) {
    plot(means[i, ], lwd = 3, ylim = range(c(lowerCI[i, ], upperCI[i,
        ])), type = "n", main = colnames(Y)[i], ylab = "log abundance",
        xlab = "time step")
    polygon(c(1:nYears, nYears:1, 1), c(upperCI[i, ], rev(lowerCI[i,
        ]), upperCI[i, 1]), col = "skyblue", lty = 0)
    lines(means[i, ], lwd = 3)
    title(rownames(Y)[i])
}
```

```
detach.jags()
```

7.10 Fitting a MARSS model with Stan

Let's fit the same model as in Section 7.9 with Stan using the **rstan** package. If you have not already, you will need to install the **rstan** package. This package depends on a number of other packages which should install automatically when you install **rstan**.

First we write the model. We could write this to a file (recommended), but for this example, we write as a character object. Though the syntax is different from the JAGS code, it has many similarities. Note that Stan

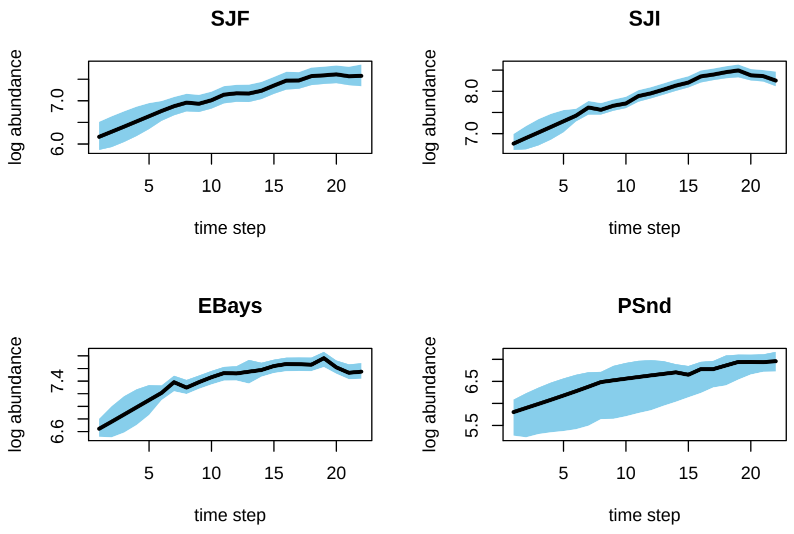


Figure 7.6: Plot of the posterior means and credible intervals for the estimated states.

does not allow missing values in the data, thus we need to pass in only the non-missing values along with the row and column indices of those values. The latter is so we can match them to the appropriate state (x) values.

```
scode <- "
data {
  int<lower=0> TT; // length of ts
  int<lower=0> N; // num of ts; rows of y
  int<lower=0> n_pos; // number of non-NA values in y
  int<lower=0> col_indx_pos[n_pos]; // col index of non-NA vals
  int<lower=0> row_indx_pos[n_pos]; // row index of non-NA vals
  vector[n_pos] y;
}
parameters {
  vector[N] x0; // initial states
  real u;
  vector[N] pro_dev[TT]; // refed as pro_dev[TT,N]
  real<lower=0> sd_q;
  real<lower=0> sd_r[N]; // obs variances are different
```

```
}
transformed parameters {
  vector[N] x[TT]; // refed as x[TT,N]
  for(i in 1:N){
    x[1,i] = x0[i] + u + pro_dev[1,i];
    for(t in 2:TT) {
      x[t,i] = x[t-1,i] + u + pro_dev[t,i];
    }
  }
}
model {
  sd_q ~ cauchy(0,5);
  for(i in 1:N){
    x0[i] ~ normal(y[i],10); // assume no missing y[1]
    sd_r[i] ~ cauchy(0,5);
    for(t in 1:TT){
    pro_dev[t,i] ~ normal(0, sd_q);
    }
  }
  u ~ normal(0,2);
  for(i in 1:n_pos){
    y[i] ~ normal(x[col_indx_pos[i], row_indx_pos[i]], sd_r[row_indx_pos[i]]);
  }
}
generated quantities {
  vector[n_pos] log_lik;
  for (n in 1:n_pos) log_lik[n] = normal_lpdf(y[n] | x[col_indx_pos[n], row_indx_pos[
}
"
```

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

```
ypos <- Y[!is.na(Y)]
n_pos <- length(ypos)  # number on non-NA ys
indx_pos <- which(!is.na(Y), arr.ind = TRUE)  # index on the non-NAs
```

```r
col_indx_pos <- as.vector(indx_pos[, "col"])
row_indx_pos <- as.vector(indx_pos[, "row"])
mod <- rstan::stan(model_code = scode, data = list(y = ypos,
    TT = ncol(Y), N = nrow(Y), n_pos = n_pos, col_indx_pos = col_indx_pos,
    row_indx_pos = row_indx_pos), pars = c("sd_q", "x", "sd_r",
    "u", "x0"), chains = 3, iter = 1000, thin = 1)
```

We use `extract()` to extract the parameters from the fitted model and then the means and 95% credible intervals.

```r
pars <- rstan::extract(mod)
means <- apply(pars$x, c(2, 3), mean)
upperCI <- apply(pars$x, c(2, 3), quantile, 0.975)
lowerCI <- apply(pars$x, c(2, 3), quantile, 0.025)
colnames(means) <- colnames(upperCI) <- colnames(lowerCI) <- rownames(Y)
```

```
No id variables; using all as measure variables
No id variables; using all as measure variables
No id variables; using all as measure variables
```

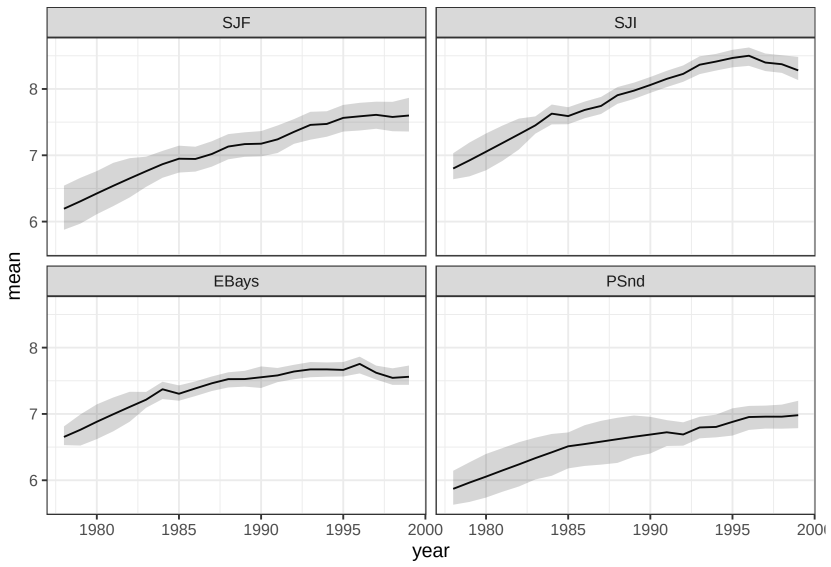


Figure 7.7: Estimated level and 95 percent credible intervals.

7.11 Problems

For these questions, use the `harborSealWA` data set in **MARSS**. The data are already logged, but you will need to remove the year column and have time going across the columns not down the rows.

```r
require(MARSS)
data(harborSealWA, package = "MARSS")
dat <- t(harborSealWA[, 2:6])
```

The sites are San Juan de Fuca (SJF 3), San Juan Islands (SJI 4), Eastern Bays (EBays 5), Puget Sound (PSnd 6) and Hood Canal (HC 7).

1. Plot the harbor seal data. Use whatever plotting functions you wish (e.g. `ggplot()`, `plot(); points(); lines()`, `matplot()`).

2. Fit a panmictic population model that assumes that each of the 5 sites is observing one "Inland WA" harbor seal population with trend u. Assume the observation errors are independent and identical. This means 1 variance on diagonal and 0s on off-diagonal. This is the default assumption for `MARSS()`.

   a. Write the **Z** for this model. The code to use for making a matrix in Rmarkdown is

   ```
   $$\begin{bmatrix}a & b & 0\\d & e & f\\0 & h & i\end{bmatrix}$$
   ```

   b. Write the **Z** matrix in R using `Z=matrix(...)` and using the factor short-cut for specifying **Z**. `Z=factor(c(...)`.

   c. Fit the model using `MARSS()`. What is the estimated trend (u)? How fast was the population increasing (percent per year) based on this estimated u?

   d. Compute the confidence intervals for the parameter estimates. Compare the intervals using the Hessian approximation and using a parametric bootstrap. What differences do you see between the two approaches? Use this code:

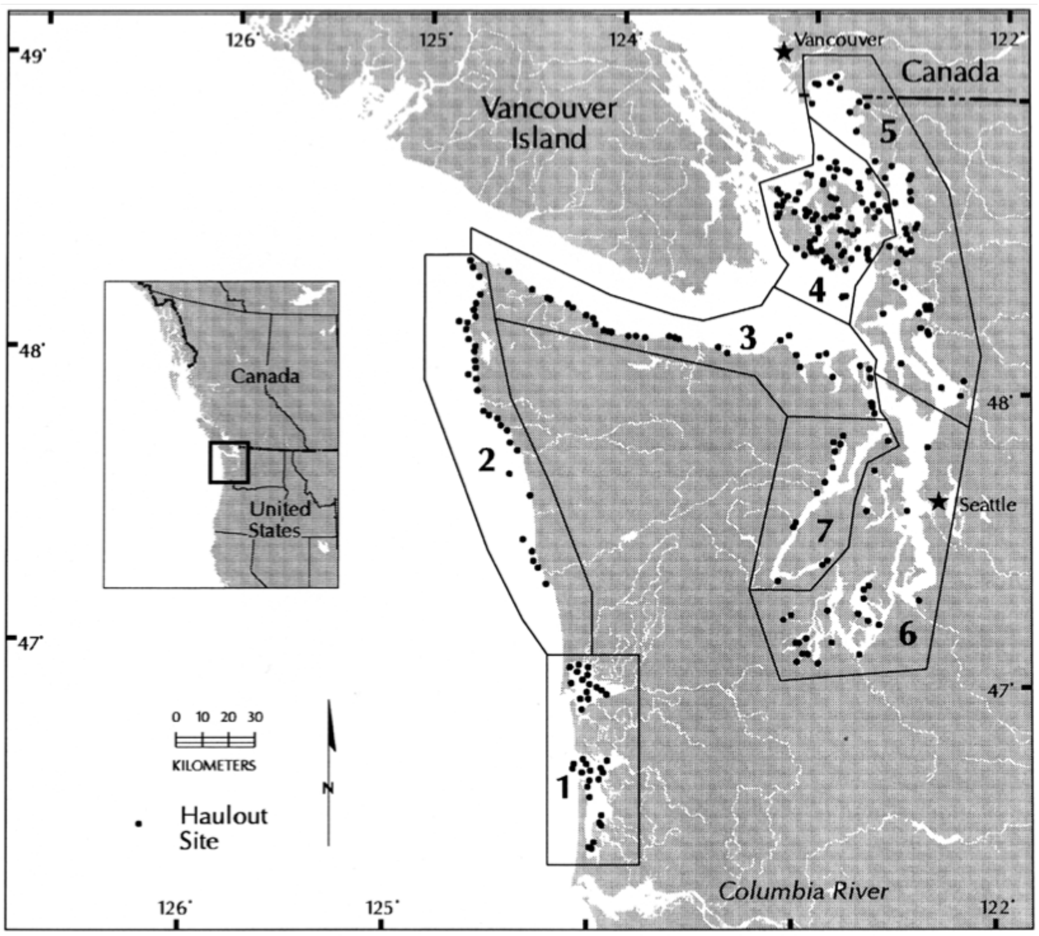


Figure 7.8: Regions in the harbor seal surveys

```
library(broom)
tidy(fit)
# set nboot low so it doesn't take forever
tidy(fit, method="parametric",nboot=100)
```

e. What does an estimate of $\mathbf{Q} = 0$ mean? What would the estimated state (x) look like when $\mathbf{Q} = 0$?

3. Using the same panmictic population model, compare 3 assumptions about the observation error structure.

   - The observation errors are independent with different variances.
   - The observation errors are independent with the same variance.
   - The observation errors are correlated with the same variance and same correlation.

   a. Write the $\mathbf{R}$ variance-covariance matrices for each assumption.
   b. Create each R matrix in R. To combine, numbers and characters in a matrix use a list matrix like so:

```
A <- matrix(list(0),3,3)
A[1,1] <- "sigma2"
```

   c. Fit each model using `MARSS()` and compute the confidence intervals (CIs) for the estimated parameters. Compare the estimated u (the population long-term trend) along with their CIs. Does the assumption about the observation errors change the u estimate?
   d. Plot the state residuals, the ACF of the state residuals, and the histogram of the state residuals for each fit. Are there any issues that you see? Use this code to get your state residuals:

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

You need the `[1,]` since the residuals are returned as a matrix.

4. Fit a model with 3 subpopulations. 1=SJF,SJI; 2=PS,EBays; 3=HC. The x part of the model is the population structure. Assume that the observation errors are identical and independent (`R="diagonal and equal"`). Assume that the process errors are unique and independent (`Q="diagonal and unequal"`). Assume that the u are unique among the 3 subpopulation.

    a. Write the $\mathbf{x}$ equation. Make sure each matrix in the equation has the right number of rows and columns.
    b. Write the $\mathbf{Z}$ matrix.
    c. Write the $\mathbf{Z}$ in R using `Z=matrix(...)` and using the factor shortcut `Z=factor(c(...))`.
    d. Fit the model with `MARSS()`.
    e. What do the estimated u and $\mathbf{Q}$ imply about the population dynamics in the 3 subpopulations?

5. Repeat the fit from Question 4 but assume that the 3 subpopulations covary. Use `Q="unconstrained"`.

    a. What does the estimated $\mathbf{Q}$ matrix tell you about how the 3 subpopulation covary?
    b. Compare the AICc from the model in Question 4 and the one with `Q="unconstrained"`. Which is more supported?
    c. Fit the model with `Q="equalvarcov"`. Is this more supported based on AICc?

6. Develop the following alternative models for the structure of the inland harbor seal population. For each model assume that the observation errors are identical and independent (`R="diagonal and equal"`). Assume that the process errors covary with equal variance and covariances (`Q="equalvarcov"`).

    - 5 subpopulations with unique u.
    - 5 subpopulations with shared (equal) u.
    - 5 subpopulations but with u shared in some regions: SJF+SJI shared, PS+EBays shared, HC unique.

- 1 panmictic population.
- 3 subpopulations, 1=SJF,SJI, 2=PS,EBays, 3=HC, with unique u
- 2 subpopulations, 1=SJF,SJI,PS,EBays, 2=HC, with unique u

a. Fit each model using `MARSS()`.

b. Prepare a table of each model with a column for the AICc values. And a column for $\Delta AICc$ (AICc minus the lowest AICc in the group). What is the most supported model?

7. Do diagnostics on the model innovations residuals for the 3 subpopulation model from question 4. Use the following code to get your model residuals. This will put NAs in the model residuals where there is missing data. Then do the tests on each row of `resids`.

```
resids <- MARSSresiduals(fit, type = "tt1")$model.residuals
resids[is.na(dat)] <- NA
```

a. Plot the model residuals.

b. Plot the ACF of the model residuals. Use `acf(..., na.action=na.pass)`.

c. Plot the histogram of the model residuals.

d. Fit an ARIMA() model to your model residuals using `forecast::auto.arima()`. Are the best fit models what you want? Note, we cannot use the Augmented Dickey-Fuller or KPSS tests when there are missing values in our residuals time series.

Chapter 8

MARSS models with covariates

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

Data and packages

For the chapter examples, we will use the green and bluegreen algae in the Lake Washington plankton data set and the covariates in that dataset. This is a 32-year time series (1962-1994) of monthly plankton counts (cells per mL) from Lake Washington, Washington, USA with the covariates total phosphorous and pH. `lakeWAplanktonTrans` is a transformed version of the raw data used for teaching purposes. Zeros have been replaced with NAs (missing). The logged (natural log) raw plankton counts have been standardized to a mean of zero and variance of 1 (so logged and then z-scored). Temperature, TP and pH were also z-scored but not logged (so z-score of the untransformed values for these covariates). The single missing temperature value was replaced with -1 and the single missing TP value was replaced with -0.3.

We will use the 10 years of data from 1965-1974 (Figure 8.1), a decade with particularly high green and bluegreen algae levels.

```
data(lakeWAplankton, package = "MARSS")
# lakeWA
fulldat <- lakeWAplanktonTrans
```

```
years <- fulldat[, "Year"] >= 1965 & fulldat[, "Year"] < 1975
dat <- t(fulldat[years, c("Greens", "Bluegreens")])
covariates <- t(fulldat[years, c("Temp", "TP")])
```

Packages:

```
library(MARSS)
library(ggplot2)
```

8.1 Overview

A multivariate autoregressive state-space (MARSS) model with covariate effects in both the process and observation components is written as:

$$\begin{gathered}\mathbf{x}_t = \mathbf{B}_t\mathbf{x}_{t-1} + \mathbf{u}_t + \mathbf{C}_t\mathbf{c}_t + \mathbf{w}_t, \text{ where } \mathbf{w}_t \sim \text{MVN}(0, \mathbf{Q}_t) \\ \mathbf{y}_t = \mathbf{Z}_t\mathbf{x}_t + \mathbf{a}_t + \mathbf{D}_t\mathbf{d}_t + \mathbf{v}_t, \text{ where } \mathbf{v}_t \sim \text{MVN}(0, \mathbf{R}_t)\end{gathered} \tag{8.1}$$

where $\mathbf{c}_t$ is the $p \times 1$ vector of covariates (e.g., temperature, rainfall) which affect the states and $\mathbf{d}_t$ is a $q \times 1$ vector of covariates (potentially the same as $\mathbf{c}_t$), which affect the observations. $\mathbf{C}_t$ is an $m \times p$ matrix of coefficients relating the effects of $\mathbf{c}_t$ to the $m \times 1$ state vector $\mathbf{x}_t$, and $\mathbf{D}_t$ is an $n \times q$ matrix of coefficients relating the effects of $\mathbf{d}_t$ to the $n \times 1$ observation vector $\mathbf{y}_t$.

With the `MARSS()` function, one can fit this model by passing in `model$c` and/or `model$d` in the `model` argument as a $p \times T$ or $q \times T$ matrix, respectively. The form for $\mathbf{C}_t$ and $\mathbf{D}_t$ is similarly specified by passing in `model$C` and/or `model$D`. $\mathbf{C}$ and $\mathbf{D}$ are matrices and are specified as 2-dimensional matrices as you would other parameter matrices.

8.2 Prepare the plankton data

We will prepare the data by z-scoring. The original data `lakeWAplanktonTrans` were already z-scored, but we changed the mean when we subsampled the years so we need to z-score again.

```
# z-score the response variables
the.mean <- apply(dat, 1, mean, na.rm = TRUE)
the.sigma <- sqrt(apply(dat, 1, var, na.rm = TRUE))
dat <- (dat - the.mean) * (1/the.sigma)
```

Next we set up the covariate data, temperature and total phosphorous. We z-score the covariates to standardize and remove the mean.

```
the.mean <- apply(covariates, 1, mean, na.rm = TRUE)
the.sigma <- sqrt(apply(covariates, 1, var, na.rm = TRUE))
covariates <- (covariates - the.mean) * (1/the.sigma)
```

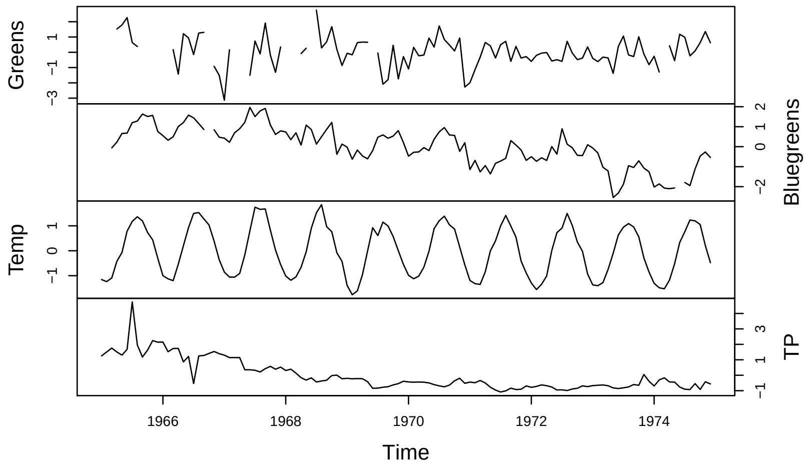


Figure 8.1: Time series of Green and Bluegreen algae abundances in Lake Washington along with the temperature and total phosporous covariates.

8.3 Observation-error only model

We can estimate the effect of the covariates using a process-error only model, an observation-error only model, or a model with both types of error. An observation-error only model is a multivariate regression, and we will start here so you see the relationship of MARSS model to more familiar linear regression models.

In a standard multivariate linear regression, we only have an observation model with independent errors (the state process does not appear in the model):

$$\mathbf{y}_t = \mathbf{a} + \mathbf{D}\mathbf{d}_t + \mathbf{v}_t, \text{ where } \mathbf{v}_t \sim \text{MVN}(0, \mathbf{R}) \tag{8.2}$$

The elements in $\mathbf{a}$ are the intercepts and those in $\mathbf{D}$ are the slopes (effects). We have dropped the t subscript on $\mathbf{a}$ and $\mathbf{D}$ because these will be modeled as time-constant. Writing this out for the two plankton and the two covariates we get:

$$\begin{bmatrix} y_g \\ y_{bg} \end{bmatrix}_t = \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} + \begin{bmatrix} \beta_{\text{g,temp}} & \beta_{\text{g,tp}} \\ \beta_{\text{bg,temp}} & \beta_{\text{bg,tp}} \end{bmatrix} \begin{bmatrix} \text{temp} \\ \text{tp} \end{bmatrix}_t + \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}_t \tag{8.3}$$

Let's fit this model with MARSS. The $\mathbf{x}$ part of the model is irrelevant so we want to fix the parameters in that part of the model. We won't set $\mathbf{B} = 0$ or $\mathbf{Z} = 0$ since that might cause numerical issues for the Kalman filter. Instead we fix them as identity matrices and fix $\mathbf{x}_0 = 0$ so that $\mathbf{x}_t = 0$ for all t.

```
Q <- U <- x0 <- "zero"
B <- Z <- "identity"
d <- covariates
A <- "zero"
D <- "unconstrained"
y <- dat  # to show relationship between dat & the equation
model.list <- list(B = B, U = U, Q = Q, Z = Z, A = A, D = D,
    d = d, x0 = x0)
kem <- MARSS(y, model = model.list)
```

```
Success! algorithm run for 15 iterations. abstol and log-log tests passed.
Alert: conv.test.slope.tol is 0.5.
Test with smaller values (<0.1) to ensure convergence.

MARSS fit is
Estimation method: kem
```

```
Convergence test: conv.test.slope.tol = 0.5, abstol = 0.001
Algorithm ran 15 (=minit) iterations and convergence was reached.
Log-likelihood: -276.4287
AIC: 562.8573   AICc: 563.1351

                     Estimate
R.diag                  0.706
D.(Greens,Temp)         0.367
D.(Bluegreens,Temp)     0.392
D.(Greens,TP)           0.058
D.(Bluegreens,TP)       0.535
Initial states (x0) defined at t=0

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

We set `A="zero"` because the data and covariates have been demeaned. Of course, one can do multiple regression in R using, say, `lm()`, and that would be much, much faster. The EM algorithm is over-kill here, but it is shown so that you see how a standard multivariate linear regression model is written as a MARSS model in matrix form.

8.4 Process-error only model

Now let's model the data as an autoregressive process observed without error, and incorporate the covariates into the process model. Note that this is much different from typical linear regression models. The $\mathbf{x}$ part represents our model of the data (in this case plankton species). How is this different from the autoregressive observation errors? Well, we are modeling our data as autoregressive so data at $t-1$ affects the data at t. Population abundances are inherently autoregressive so this model is a bit closer to the underlying mechanism generating the data. Here is our new process model for plankton abundance.

$$\mathbf{x}_t = \mathbf{x}_{t-1} + \mathbf{C}\mathbf{c}_t + \mathbf{w}_t, \text{ where } \mathbf{w}_t \sim \text{MVN}(0, \mathbf{Q}) \tag{8.4}$$

We can fit this as follows:

```
R <- A <- U <- "zero"
B <- Z <- "identity"
Q <- "equalvarcov"
C <- "unconstrained"
model.list <- list(B = B, U = U, Q = Q, Z = Z, A = A, R = R,
    C = C, c = covariates)
kem <- MARSS(dat, model = model.list)
```

```
Success! algorithm run for 15 iterations. abstol and log-log tests passed.
Alert: conv.test.slope.tol is 0.5.
Test with smaller values (<0.1) to ensure convergence.

MARSS fit is
Estimation method: kem
Convergence test: conv.test.slope.tol = 0.5, abstol = 0.001
Algorithm ran 15 (=minit) iterations and convergence was reached.
Log-likelihood: -285.0732
AIC: 586.1465   AICc: 586.8225

                      Estimate
Q.diag                  0.7269
Q.offdiag              -0.0210
x0.X.Greens            -0.5189
x0.X.Bluegreens        -0.2431
C.(X.Greens,Temp)      -0.0434
C.(X.Bluegreens,Temp)   0.0988
C.(X.Greens,TP)        -0.0589
C.(X.Bluegreens,TP)     0.0104
Initial states (x0) defined at t=0

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

Now, it looks like temperature has a strong negative effect on algae? Also our log-likelihood dropped a lot. Well, the data do not look at all like a random walk model (where $\mathbf{B} = 1$), which we can see from the plot of the data (Figure 8.1). The data are fluctuating about some mean so let's switch to a better autoregressive model—a mean-reverting model. To do this, we will allow the diagonal elements of **B** to be something other than 1.

```
model.list$B <- "diagonal and unequal"
kem <- MARSS(dat, model = model.list)
```

```
Success! algorithm run for 15 iterations. abstol and log-log tests passed.
Alert: conv.test.slope.tol is 0.5.
Test with smaller values (<0.1) to ensure convergence.

MARSS fit is
Estimation method: kem
Convergence test: conv.test.slope.tol = 0.5, abstol = 0.001
Algorithm ran 15 (=minit) iterations and convergence was reached.
Log-likelihood: -236.6106
AIC: 493.2211   AICc: 494.2638

                              Estimate
B.(X.Greens,X.Greens)           0.1981
B.(X.Bluegreens,X.Bluegreens)   0.7672
Q.diag                          0.4899
Q.offdiag                      -0.0221
x0.X.Greens                    -1.2915
x0.X.Bluegreens                -0.4179
C.(X.Greens,Temp)               0.2844
C.(X.Bluegreens,Temp)           0.1655
C.(X.Greens,TP)                 0.0332
C.(X.Bluegreens,TP)             0.1340
Initial states (x0) defined at t=0

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

Notice that the log-likelihood goes up quite a bit, which means that the mean-reverting model fits the data much better.

With this model, we are estimating $\mathbf{x}_0$. If we set `model$tinitx=1`, we will get a error message that **R** diagonals are equal to 0 and we need to fix `x0`.

Because $\mathbf{R} = 0$, if we set the initial states at $t = 1$, then they are fully determined by the data.

```
x0 <- dat[, 1, drop = FALSE]
model.list$tinitx <- 1
model.list$x0 <- x0
kem <- MARSS(dat, model = model.list)
```

```
Success! algorithm run for 15 iterations. abstol and log-log tests passed.
Alert: conv.test.slope.tol is 0.5.
Test with smaller values (<0.1) to ensure convergence.

MARSS fit is
Estimation method: kem
Convergence test: conv.test.slope.tol = 0.5, abstol = 0.001
Algorithm ran 15 (=minit) iterations and convergence was reached.
Log-likelihood: -235.4827
AIC: 486.9653   AICc: 487.6414

                              Estimate
B.(X.Greens,X.Greens)           0.1980
B.(X.Bluegreens,X.Bluegreens)   0.7671
Q.diag                          0.4944
Q.offdiag                      -0.0223
C.(X.Greens,Temp)               0.2844
C.(X.Bluegreens,Temp)           0.1655
C.(X.Greens,TP)                 0.0332
C.(X.Bluegreens,TP)             0.1340
Initial states (x0) defined at t=1

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

8.5 Both process- and observation-error

Here is an example where we have both process and observation error but the covariates only affect the process:

$$\begin{aligned} \mathbf{x}_t &= \mathbf{B}\mathbf{x}_{t-1} + \mathbf{C}_t\mathbf{c}_t + \mathbf{w}_t, \text{ where } \mathbf{w}_t \sim \text{MVN}(0, \mathbf{Q}) \\ \mathbf{y}_t &= \mathbf{x}_{t-1} + \mathbf{v}_t, \text{ where } \mathbf{v}_t \sim \text{MVN}(0, \mathbf{R}), \end{aligned} \tag{8.5}$$

$\mathbf{x}$ is the true algae abundances and $\mathbf{y}$ is the observation of the $\mathbf{x}$'s.

Let's say we knew that the observation variance on the algae measurements was about 0.16 and we wanted to include that known value in the model. To do that, we can simply add $\mathbf{R}$ to the model list from the process-error only model in the last example.

```
D <- d <- A <- U <- "zero"
Z <- "identity"
B <- "diagonal and unequal"
Q <- "equalvarcov"
C <- "unconstrained"
c <- covariates
R <- diag(0.16, 2)
x0 <- "unequal"
tinitx <- 1
model.list <- list(B = B, U = U, Q = Q, Z = Z, A = A, R = R,
    D = D, d = d, C = C, c = c, x0 = x0, tinitx = tinitx)
kem <- MARSS(dat, model = model.list)
```

```
Success! abstol and log-log tests passed at 36 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 36 iterations.
Log-likelihood: -240.3694
AIC: 500.7389   AICc: 501.7815
```

```
                              Estimate
B.(X.Greens,X.Greens)          0.30848
B.(X.Bluegreens,X.Bluegreens)  0.76101
Q.diag                         0.33923
Q.offdiag                     -0.00411
x0.X.Greens                   -0.52614
x0.X.Bluegreens               -0.32836
C.(X.Greens,Temp)              0.23790
C.(X.Bluegreens,Temp)          0.16991
C.(X.Greens,TP)                0.02505
C.(X.Bluegreens,TP)            0.14183
Initial states (x0) defined at t=1

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

Note, our estimates of the effect of temperature and total phosphorous are not that different than what you get from a simple multiple regression (our first example). This might be because the autoregressive component is small, meaning the estimated diagonals on the $\mathbf{B}$ matrix are small.

Here is an example where we have both process and observation error but the covariates only affect the observation process:

$$\begin{gathered}\mathbf{x}_t = \mathbf{B}\mathbf{x}_{t-1} + \mathbf{w}_t, \text{ where } \mathbf{w}_t \sim \text{MVN}(0, \mathbf{Q}) \\ \mathbf{y}_t = \mathbf{x}_{t-1} + \mathbf{D}\mathbf{d}_t + \mathbf{v}_t, \text{ where } \mathbf{v}_t \sim \text{MVN}(0, \mathbf{R}),\end{gathered} \tag{8.6}$$

$\mathbf{x}$ is the true algae abundances and $\mathbf{y}$ is the observation of the $\mathbf{x}$'s.

```
C <- c <- A <- U <- "zero"
Z <- "identity"
B <- "diagonal and unequal"
Q <- "equalvarcov"
D <- "unconstrained"
d <- covariates
R <- diag(0.16, 2)
x0 <- "unequal"
```

```
tinitx <- 1
model.list <- list(B = B, U = U, Q = Q, Z = Z, A = A, R = R,
    D = D, d = d, C = C, c = c, x0 = x0, tinitx = tinitx)
kem <- MARSS(dat, model = model.list)
```

```
Success! abstol and log-log tests passed at 45 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 45 iterations.
Log-likelihood: -239.5879
AIC: 499.1759   AICc: 500.2185

                              Estimate
B.(X.Greens,X.Greens)            0.428
B.(X.Bluegreens,X.Bluegreens)    0.859
Q.diag                           0.314
Q.offdiag                       -0.030
x0.X.Greens                     -0.121
x0.X.Bluegreens                 -0.119
D.(Greens,Temp)                  0.373
D.(Bluegreens,Temp)              0.276
D.(Greens,TP)                    0.042
D.(Bluegreens,TP)                0.115
Initial states (x0) defined at t=1

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

8.6 Including seasonal effects in MARSS models

Time-series data are often collected at intervals with some implicit "seasonality." For example, quarterly earnings for a business, monthly rainfall totals, or hourly air temperatures. In those cases, it is often helpful to extract any recurring seasonal patterns that might otherwise mask some of the other temporal dynamics we are interested in examining.

Here we show a few approaches for including seasonal effects using the Lake Washington plankton data, which were collected monthly. The following examples will use all five phytoplankton species from Lake Washington. First, let's set up the data.

```
years <- fulldat[, "Year"] >= 1965 & fulldat[, "Year"] < 1975
phytos <- c("Diatoms", "Greens", "Bluegreens", "Unicells", "Other.algae")
dat <- t(fulldat[years, phytos])

# z.score data because we changed the mean when we
# subsampled
the.mean <- apply(dat, 1, mean, na.rm = TRUE)
the.sigma <- sqrt(apply(dat, 1, var, na.rm = TRUE))
dat <- (dat - the.mean) * (1/the.sigma)
# number of time periods/samples
TT <- dim(dat)[2]
```

8.6.1 Seasonal effects as fixed factors

One common approach for estimating seasonal effects is to treat each one as a fixed factor, such that the number of parameters equals the number of "seasons" (e.g., 24 hours per day, 4 quarters per year). The plankton data are collected monthly, so we will treat each month as a fixed factor. To fit a model with fixed month effects, we create a $12 \times T$ covariate matrix $\mathbf{c}$ with one row for each month (Jan, Feb, ...) and one column for each time point. We put a 1 in the January row for each column corresponding to a January time point, a 1 in the February row for each column corresponding to a February time point, and so on. All other values of **c** equal 0. The following code will create such a **c** matrix.

```
# number of 'seasons' (e.g., 12 months per year)
period <- 12
# first 'season' (e.g., Jan = 1, July = 7)
per.1st <- 1
# create factors for seasons
c.in <- diag(period)
for (i in 2:(ceiling(TT/period))) {
    c.in <- cbind(c.in, diag(period))
}
# trim c.in to correct start & length
c.in <- c.in[, (1:TT) + (per.1st - 1)]
# better row names
rownames(c.in) <- month.abb
```

Next we need to set up the form of the **C** matrix which defines any constraints we want to set on the month effects. **C** is a 5×12 matrix. Five taxon and 12 month effects. If we wanted each taxon to have the same month effect, i.e. there is a common month effect across all taxon, then we have the same value in each **C** column[1]:

```
C <- matrix(month.abb, 5, 12, byrow = TRUE)
C
```

```
     [,1]  [,2]  [,3]  [,4]  [,5]  [,6]  [,7]  [,8]  [,9]  [,10] [,11] [,12]
[1,] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov" "Dec"
[2,] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov" "Dec"
[3,] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov" "Dec"
[4,] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov" "Dec"
[5,] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov" "Dec"
```

Notice, that **C** only has 12 values in it, the 12 common month effects. However, for this example, we will let each taxon have a different month effect

[1]'month.abb' is a R constant that gives month abbreviations in text.

thus allowing different seasonality for each taxon. For this model, we want each value in **C** to be unique:

```
C <- "unconstrained"
```

Now **C** has $5 \times 12 = 60$ separate effects.

Then we set up the form for the rest of the model parameters. We make the following assumptions:

```
# Each taxon has unique density-dependence
B <- "diagonal and unequal"
# Assume independent process errors
Q <- "diagonal and unequal"
# We have demeaned the data & are fitting a mean-reverting
# model by estimating a diagonal B, thus
U <- "zero"
# Each obs time series is associated with only one process
Z <- "identity"
# The data are demeaned & fluctuate around a mean
A <- "zero"
# We assume observation errors are independent, but they
# have similar variance due to similar collection methods
R <- "diagonal and equal"
# We are not including covariate effects in the obs
# equation
D <- "zero"
d <- "zero"
```

Now we can set up the model list for MARSS and fit the model (results are not shown since they are verbose with 60 different month effects).

```
model.list <- list(B = B, U = U, Q = Q, Z = Z, A = A, R = R,
    C = C, c = c.in, D = D, d = d)
seas.mod.1 <- MARSS(dat, model = model.list, control = list(maxit = 1500))

# Get the estimated seasonal effects rows are taxa, cols
# are seasonal effects
```

```r
seas.1 <- coef(seas.mod.1, type = "matrix")$C
rownames(seas.1) <- phytos
colnames(seas.1) <- month.abb
```

The top panel in Figure 8.2 shows the estimated seasonal effects for this model. Note that if we had set U="unequal", we would need to set one of the columns of **C** to zero because the model would be under-determined (infinite number of solutions). If we substracted the mean January abundance off each time series, we could set the January column in **C** to 0 and get rid of 5 estimated effects.

8.6.2 Seasonal effects as a polynomial

The fixed factor approach required estimating 60 effects. Another approach is to model the month effect as a 3rd-order (or higher) polynomial: $a + b \times m + c \times m^2 + d \times m^3$ where m is the month number. This approach has less flexibility but requires only 20 estimated parameters (i.e., 4 regression parameters times 5 taxa). To do so, we create a $4 \times T$ covariate matrix $\mathbf{c}$ with the rows corresponding to 1, m, m^2, and m^3, and the columns again corresponding to the time points. Here is how to set up this matrix:

```r
# number of 'seasons' (e.g., 12 months per year)
period <- 12
# first 'season' (e.g., Jan = 1, July = 7)
per.1st <- 1
# order of polynomial
poly.order <- 3
# create polynomials of months
month.cov <- matrix(1, 1, period)
for (i in 1:poly.order) {
    month.cov = rbind(month.cov, (1:12)^i)
}
# our c matrix is month.cov replicated once for each year
c.m.poly <- matrix(month.cov, poly.order + 1, TT + period, byrow = FALSE)
# trim c.in to correct start & length
c.m.poly <- c.m.poly[, (1:TT) + (per.1st - 1)]
```

```
# Everything else remains the same as in the previous
# example
model.list <- list(B = B, U = U, Q = Q, Z = Z, A = A, R = R,
    C = C, c = c.m.poly, D = D, d = d)
seas.mod.2 <- MARSS(dat, model = model.list, control = list(maxit = 1500))
```

The effect of month m for taxon i is $a_i + b_i \times m + c_i \times m^2 + d_i \times m^3$, where a_i, b_i, c_i and d_i are in the i-th row of $\mathbf{C}$. We can now calculate the matrix of seasonal effects as follows, where each row is a taxon and each column is a month:

```
C.2 = coef(seas.mod.2, type = "matrix")$C
seas.2 = C.2 %*% month.cov
rownames(seas.2) <- phytos
colnames(seas.2) <- month.abb
```

The middle panel in Figure 8.2 shows the estimated seasonal effects for this polynomial model.

Note: Setting the covariates up like this means that our covariates are collinear since m, m^2 and m^3 are correlated, obviously. A better approach is to use the `poly()` function to create an orthogonal polynomial covariate matrix `c.m.poly.o`:

```
month.cov.o <- cbind(1, poly(1:period, poly.order))
c.m.poly.o <- matrix(t(month.cov.o), poly.order + 1, TT + period,
    byrow = FALSE)
c.m.poly.o <- c.m.poly.o[, (1:TT) + (per.1st - 1)]
```

8.6.3 Seasonal effects as a Fourier series

The factor approach required estimating 60 effects, and the 3rd order polynomial model was an improvement at only 20 parameters. A third option is to use a discrete Fourier series, which is combination of sine and cosine waves; it would require only 10 parameters. Specifically, the effect of month m on taxon i is $a_i \times \cos(2\pi m/p) + b_i \times \sin(2\pi m/p)$, where p is the period (e.g., 12 months, 4 quarters), and a_i and b_i are contained in the i-th row of $\mathbf{C}$.

We begin by defining the $2 \times T$ seasonal covariate matrix $\mathbf{c}$ as a combination of 1 cosine and 1 sine wave:

```
cos.t <- cos(2 * pi * seq(TT)/period)
sin.t <- sin(2 * pi * seq(TT)/period)
c.Four <- rbind(cos.t, sin.t)
```

Everything else remains the same and we can fit this model as follows:

```
model.list <- list(B = B, U = U, Q = Q, Z = Z, A = A, R = R,
    C = C, c = c.Four, D = D, d = d)
seas.mod.3 <- MARSS(dat, model = model.list, control = list(maxit = 1500))
```

We make our seasonal effect matrix as follows:

```
C.3 <- coef(seas.mod.3, type = "matrix")$C
# The time series of net seasonal effects
seas.3 <- C.3 %*% c.Four[, 1:period]
rownames(seas.3) <- phytos
colnames(seas.3) <- month.abb
```

The bottom panel in Figure 8.2 shows the estimated seasonal effects for this seasonal-effects model based on a discrete Fourier series.

Rather than rely on our eyes to judge model fits, we should formally assess which of the 3 approaches offers the most parsimonious fit to the data. Here is a table of AICc values for the 3 models:

```
data.frame(Model = c("Fixed", "Cubic", "Fourier"), AICc = round(c(seas.mod.1$AICc,
    seas.mod.2$AICc, seas.mod.3$AICc), 1))
```

```
    Model   AICc
1   Fixed 1188.4
2   Cubic 1144.9
3 Fourier 1127.4
```

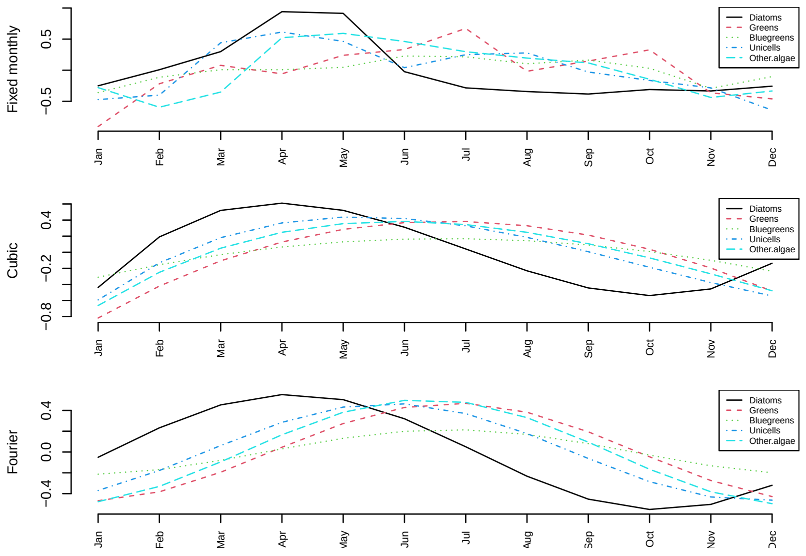


Figure 8.2: Estimated monthly effects for the three approaches to estimating seasonal effects. Top panel: each month modelled as a separate fixed effect for each taxon (60 parameters); Middle panel: monthly effects modelled as a 3rd order polynomial (20 parameters); Bottom panel: monthly effects modelled as a discrete Fourier series (10 parameters).

The model selection results indicate that the model with monthly seasonal effects estimated via the discrete Fourier sequence is the best of the 3 models. Its AICc value is much lower than either the polynomial or fixed-effects models.

8.7 Model diagnostics

We will examine some basic model diagnostics for these three approaches by looking at plots of the model residuals and their autocorrelation functions (ACFs) for all five taxa using the following code:

```
for (i in 1:3) {
    dev.new()
    modn <- paste("seas.mod", i, sep = ".")
    for (j in 1:5) {
        resid.j <- MARSSresiduals(get(modn), type = "tt1")$model.residuals[j,
            ]
        plot.ts(resid.j, ylab = "Residual", main = phytos[j])
        abline(h = 0, lty = "dashed")
        acf(resid.j, na.action = na.pass)
    }
}
```

8.8 Homework data and discussion

For these problems, use the following code to load in 1980-1994 phytoplankton data, covariates, and z-score all the data. Run the code below and use `dat` and `covars` directly in your code.

```
library(MARSS)
spp <- c("Cryptomonas", "Diatoms", "Greens", "Unicells", "Other.algae",
    "Daphnia")
yrs <- lakeWAplanktonTrans[, "Year"] %in% 1980:1994
dat <- t(lakeWAplanktonTrans[yrs, spp])
```

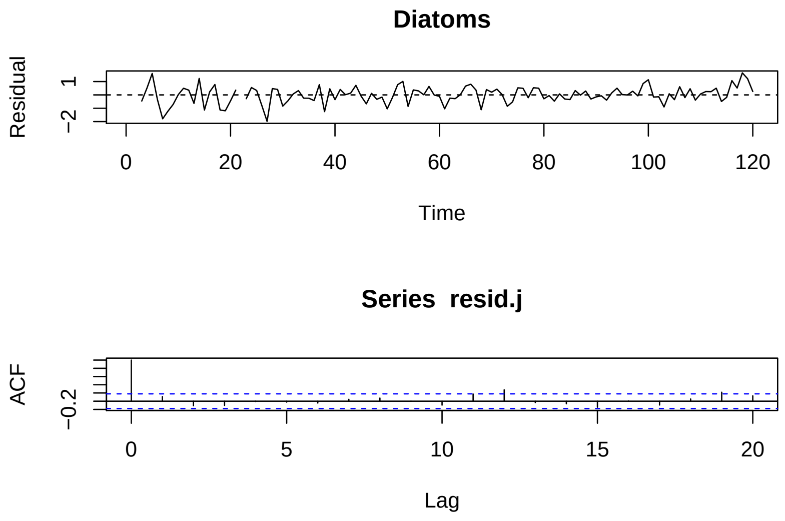


Figure 8.3: Residuals for model with season modelled as a discrete Fourier series.

```
# z-score the data
avg <- apply(dat, 1, mean, na.rm = TRUE)
sd <- sqrt(apply(dat, 1, var, na.rm = TRUE))
dat <- (dat - avg)/sd
rownames(dat) = spp
# always check that the mean and variance are 1 after
# z-scoring
apply(dat, 1, mean, na.rm = TRUE)  #this should be 0
apply(dat, 1, var, na.rm = TRUE)  #this should be 1
```

For the covariates, you'll use temperature and TP.

```
covars <- rbind(Temp = lakeWAplanktonTrans[yrs, "Temp"], TP = lakeWAplanktonTr
    "TP"])
avg <- apply(covars, 1, mean)
sd <- sqrt(apply(covars, 1, var, na.rm = TRUE))
covars <- (covars - avg)/sd
rownames(covars) <- c("Temp", "TP")
```

```
# always check that the mean and variance are 1 after
# z-scoring
apply(covars, 1, mean, na.rm = TRUE)  #this should be 0
apply(covars, 1, var, na.rm = TRUE)  #this should be 1
```

Here are some guidelines to help you answer the questions:

- Use a MARSS model that allows for both observation and process error.
- Assume that the observation errors are independent and identically distributed with known variance of 0.10.
- Assume that the process errors are independent from one another, but the variances differ by taxon.
- Assume that each group is an observation of its own process. This means `Z="identity"`.
- Use `B="diagonal and unequal"`. This implies that each of the taxa are operating under varying degrees of density-dependence, and they are not allowed to interact.
- All the data have been de-meaned and **Z** is identity, therefore use `U="zero"` and `A="zero"`. Make sure to check that the means of the data are 0 and the variance is 1.
- Use `tinitx=1`. It makes **B** estimation more stable. It goes in your model list.
- Include a plot of residuals versus time and acf of residuals for each question. You only need to show these for the top (best) model if the question involves comparing multiple models.
- Use AICc to compare models.
- Some of the models may not converge, however use for the purpose of the homework, use the unconverged models. Thus use the output from `MARSS()` without any additional arguments. If you want, you can try using `control=list(maxit=1000)` to increase the number of iterations. Or you can try `method="BFGS"` in your `MARSS()` call. This will use the BFGS optimization method, however it may throw an error for these data.

8.9 Problems

Read Section 8.8 for the data and tips on answering the questions and setting up your models. Note the questions asking about the effects on *growth rate* are asking about the C matrix in

$$\mathbf{x}_t = \mathbf{B}\mathbf{x}_{t-1} + \mathbf{C}\mathbf{c}_t + \mathbf{w}_t$$

The $\mathbf{C}\mathbf{c}_t + \mathbf{w}_t$ are the process errors and represent the growth rates (growth above or below what you would expect given $\mathbf{x}_{t-1}$). Use your raw data in the MARSS model. You do not need to difference the data to get at the growth rates since the process model is modeling that.

1. How does month affect the mean phytoplankton population growth rates? Show a plot of the estimated mean growth rate versus month for each taxon using three approaches to estimate the month effect (factor, polynomial, Fourier series). Estimate seasonal effects without any covariate (Temp, TP) effects.

2. It is likely that both temperature and total phosphorus (TP) affect phytoplankton population growth rates. Using MARSS models, estimate the effect of Temp and TP on growth rates of each taxon. Leave out the seasonal covariates from question 1, i.e. only use Temp and TP as covariates. Make a plot of the point estimates of the Temp and TP effects with the 95% CIs added to the plot. `tidy()` is an easy way to get the parameters CIs.

3. Estimate the Temp and TP effects using `B="unconstrained"`.

   a. Compare the **B** matrix for the fit from question 2 and from question 3. Describe the species interactions modeled by the **B** matrix when `B="unconstrained"`. How is it different than the **B** matrix from question 2? Note, you can retrieve the matrix using `coef(fit, type="matrix")$B`.

   b. Do the Temp and TP effects change when you use `B="unconstrained"`? Make sure to look at the CIs also.

4. Using MARSS models, evaluate which (Temp or TP) is the more important driver or if both are important. Again, leave out the seasonal

covariates from question 1, i.e. only use Temp and TP as covariates. Compare two approaches: comparison of effect sizes in a model with both Temp and TP and model selection using a set of models.

5. Evaluate whether the effect of temperature (Temp) on the taxa manifests itself via their underlying physiology (by affecting growth rates and thus abundance) or because physical changes in the water stratification makes them easier/harder to sample in some months. Leave out the seasonal covariates from question 1, i.e. only use Temp and TP as the covariates. For TP, assume it always affects growth rates, never the observation errors.

6. Is there support for temperature or TP affecting all functional groups' growth rates the same, or are the effects on one taxon different from another? Make sure to test all possibilities: the Temp and TP effects are the same for all taxa, and one covariate effect is the same across taxa while the other's effects are unique across taxa.

7. Compare your results for question 2 using linear regression, by using the `lm()` function. You'll need to look at the response of each taxon separately, i.e. one response variable. You can have a multivariate response variable with `lm()` but the functions will be doing 6 independent linear regressions. In your `lm()` model, use only Temp and TP (and intercept) as covariates. Compare the estimated effects to those from question 2. How are they different? How is this model different from the model you fit in question 2?

8. Temp and TP are negatively correlated (cor = -0.66). A common threshold for collinearity in regression models is 0.7. Temp and TP fall below that but are close. One approach to collinearity is sequential regression (Dormann et al., 2013). The first (most influential) covariate is included 'as is' and the second covariate appears as the residuals of a regression of the second against the first. The covariates are now orthogonal however the second covariate is conditioned on the first. If we see an effect of the residuals covariate, it is the effect of TP additional to the contribution it already made through its relationship with temperature. Rerun question 2 using sequential regression (see code below).

Make your Temp and TP covariates orthogonal using sequential regression. Do your conclusions about the effects of Temperature and TP change?

Below is code to construct your orthogonal covariates for sequential regression.

```
fit <- lm(covars[1, ] ~ covars[2, ])
seqcovs <- rbind(covars[1, ], residuals(fit))
avg <- apply(seqcovs, 1, mean)
sd <- sqrt(apply(seqcovs, 1, var, na.rm = TRUE))
seqcovs <- (seqcovs - avg)/sd
rownames(seqcovs) <- c("Temp", "TPresids")
```

9. Compare the AICc's of the 3 seasonal models from question 1 and the 4 Temp/TP models from question 5. What does this tell you about the Temp and TP only models?

10. We cannot just fit a model with season and Temp plus TP since Temp and TP are highly seasonal. That will cause problems if we have something that explain season (a polynomial) and a covariate that has seasonality. Instead, use sequential regression to fit a model with seasonality, Temp and TP. Use a 3rd order polynomial with the `poly()` function to create orthogonal season covariates and then use sequential regression (code in problem 8) to create Temp and TP covariates that are orthogonal to your season covariates. Fit the model and compare a model with only season to a model with season and Temp plus TP.

11. Another approach to looking at effects of covariates which have season cycles is to examine if the seasonal anomalies of the independent variable can be explained by the seasonal anomalies of the dependent variables. In other words, can an unusually high February abundance (higher than expected) be explained by an unusually high or low February temperature? In this approach, you remove season so you do not need to model it (with factor, polynomial, etc). The `stl()` function can be used to decompose a time series using LOESS. We'll use `stl()` since it can handle missing values.

    a. Decompose the Diatom time series using `stl()` and plot. Use `na.action=zoo::na.approx` to deal with the NAs. Use `s.window="periodic"`. Other than that you can use the defaults.

```r
i <- "Diatoms"
dati <- ts(dat[i, ], frequency = 12)
a <- stl(dati, "periodic", na.action = zoo::na.approx)
```

b. Create dependent variables and covariates that are anomolies by modifying the following code. For the anomaly, you will use the remainder plus the trend. You will need to adapt this code to create the anomalies for Temp and TP and for `dat` (your data).

```r
i <- "Diatoms"
a <- stl(ts(dat[i, ], frequency = 12), "periodic", na.action = zoo::na.approx)
anom <- a$time.series[, "remainder"] + a$time.series[, "trend"]
```

c. Notice that you have simply removed the seasonal cycle from the data. Using the seasonal anomalies (from part b), estimate the effect of Temp and TP on each taxon's growth rate. You will use the same model as in question 2, but use the seasonal anomalies as data and covariates.

```r
anoms <- matrix(NA, dim(dat)[1] + dim(covars)[1], dim(dat)[2])
rownames(anoms) <- c(rownames(dat), rownames(covars))
for (i in 1:dim(dat)[1]) {
    a <- stl(ts(dat[i, ], frequency = 12), "periodic", na.action = zoo::na.approx)
    anoms[i, ] <- a$time.series[, "remainder"] + a$time.series[,
        "trend"]
}
for (i in 1:dim(covars)[1]) {
    a <- stl(ts(covars[i, ], frequency = 12), "periodic", na.action = zoo::na.approx)
    anoms[i + dim(dat)[1], ] <- a$time.series[, "remainder"] +
        a$time.series[, "trend"]
}
```

Chapter 9

Dynamic linear models

Dynamic linear models (DLMs) are a type of linear regression model, wherein the parameters are treated as time-varying rather than static. DLMs are used commonly in econometrics, but have received less attention in the ecological literature (c.f. Lamon et al., 1998; Scheuerell and Williams, 2005). Our treatment of DLMs is rather cursory—we direct the reader to excellent textbooks by Pole et al. (1994) and Petris et al. (2009) for more in-depth treatments of DLMs. The former focuses on Bayesian estimation whereas the latter addresses both likelihood-based and Bayesian estimation methods.

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

Data

Most of the data used in the chapter are from the **MARSS** package. Install the package, if needed, and load:

```
library(MARSS)
```

The problem set uses an additional data set on spawners and recruits (`KvichakSockeye`) in the `atsalibrary` package.

9.1 Overview

We begin our description of DLMs with a *static* regression model, wherein the i^{th} observation (response variable) is a linear function of an intercept, predictor variable(s), and a random error term. For example, if we had one predictor variable (f), we could write the model as

$$y_i = \alpha + \beta f_i + v_i, \tag{9.1}$$

where the α is the intercept, β is the regression slope, f_i is the predictor variable matched to the i^{th} observation (y_i), and $v_i \sim \text{N}(0, r)$. It is important to note here that there is no implicit ordering of the index i. That is, we could shuffle any/all of the (y_i, f_i) pairs in our dataset with no effect on our ability to estimate the model parameters.

We can write Equation (9.1) using matrix notation, as

$$\begin{aligned} y_i &= \begin{bmatrix} 1 & f_i \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} + v_i \\ &= \mathbf{F}_i^\top \boldsymbol{\theta} + v_i, \end{aligned} \tag{9.2}$$

where $\mathbf{F}_i^\top = \begin{bmatrix} 1 & f_i \end{bmatrix}$ and $\boldsymbol{\theta} = \begin{bmatrix} \alpha \\ \beta \end{bmatrix}$.

In a DLM, however, the regression parameters are *dynamic* in that they "evolve" over time. For a single observation at time t, we can write

$$(\#eq: dlm - dlm_1) y_t = \mathbf{F}_t^\top \boldsymbol{\theta}_t + v_t, \tag{9.3}$$

where $\mathbf{F}_t$ is a column vector of predictor variables (covariates) at time t, $\boldsymbol{\theta}_t$ is a column vector of regression parameters at time t and $v_t \sim \text{N}(0, r)$. This formulation presents two features that distinguish it from Equation (9.2). First, the observed data are explicitly time ordered (i.e., $\mathbf{y} = \{y_1, y_2, y_3, \dots, y_T\}$), which means we expect them to contain implicit information. Second, the relationship between the observed datum and the predictor variables are unique at every time t (i.e., $\boldsymbol{\theta} = \{\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \boldsymbol{\theta}_3, \dots, \boldsymbol{\theta}_T\}$).

However, closer examination of Equation @ref(eq:dlm-dlm_1) reveals an apparent complication for parameter estimation. With only one datum at each time step t, we could, at best, estimate only one regression parameter, and even then, the 1:1 correspondence between data and parameters would preclude any estimation of parameter uncertainty. To address this shortcoming, we return to the time ordering of model parameters. Rather than assume the regression parameters are independent from one time step to another, we instead model them as an autoregressive process where

$$\boldsymbol{\theta}_t = \mathbf{G}_t \boldsymbol{\theta}_{t-1} + \mathbf{w}_t, \tag{9.4}$$

$\mathbf{G}_t$ is the parameter "evolution" matrix, and $\mathbf{w}_t$ is a vector of process errors, such that $\mathbf{w}_t \sim \text{MVN}(\mathbf{0}, \mathbf{Q})$. The elements of $\mathbf{G}_t$ may be known and fixed *a priori*, or unknown and estimated from the data. Although we could allow $\mathbf{G}_t$ to be time-varying, we will typically assume that it is time invariant or assume $\mathbf{G}_t$ is an $m \times m$ identity matrix $\mathbf{I}_m$.

The idea is that the evolution matrix $\mathbf{G}_t$ deterministically maps the parameter space from one time step to the next, so the parameters at time t are temporally related to those before and after. However, the process is corrupted by stochastic error, which amounts to a degradation of information over time. If the diagonal elements of $\mathbf{Q}$ are relatively large, then the parameters can vary widely from t to $t+1$. If $\mathbf{Q} = \mathbf{0}$, then $\boldsymbol{\theta}_1 = \boldsymbol{\theta}_2 = \boldsymbol{\theta}_T$ and we are back to the static model in Equation (9.1).

9.2 DLM in state-space form

A DLM is a state-space model and can be written in MARSS form:

$$\begin{gathered}
y_t = \mathbf{F}_t^\top \boldsymbol{\theta}_t + e_t \\
\boldsymbol{\theta}_t = \mathbf{G} \boldsymbol{\theta}_{t-1} + \mathbf{w}_t \\
\Downarrow \\
y_t = \mathbf{Z}_t \mathbf{x}_t + v_t \\
\mathbf{x}_t = \mathbf{B} \mathbf{x}_{t-1} + \mathbf{w}_t
\end{gathered} \tag{9.5}$$

Note that DLMs include predictor variables (covariates) in the observation equation much differently than other forms of MARSS models. In a DLM,

$\mathbf{Z}$ is a matrix of predictor variables and $\mathbf{x}_t$ are the time-evolving regression parameters.

$$y_t = \boxed{\mathbf{Z}_t\mathbf{x}_t} + v_t. \tag{9.6}$$

In many other MARSS models, $\mathbf{d}_t$ is a time-varying column vector of covariates and $\mathbf{D}$ is the matrix of covariate-effect parameters.

$$y_t = \mathbf{Z}_t\mathbf{x}_t + \boxed{\mathbf{D}\mathbf{d}_t} + v_t. \tag{9.7}$$

9.3 Stochastic level models

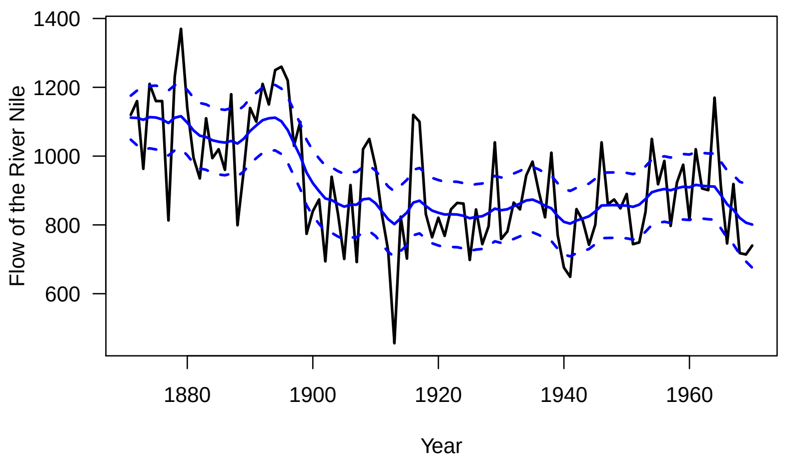
The most simple DLM is a stochastic level model, where the level is a random walk without drift, and this level is observed with error. We will write it first in using regression notation where the intercept is α and then in MARSS notation. In the latter, $\alpha_t = x_t$.

$$\begin{gathered}
y_t = \alpha_t + e_t \\
\alpha_t = \alpha_{t-1} + w_t \\
\Downarrow \\
y_t = x_t + v_t \\
x_t = x_{t-1} + w_t
\end{gathered} \tag{9.8}$$

Using this model, we can model the Nile River level and fit the model using `MARSS()`.

```
## load Nile flow data
data(Nile, package = "datasets")

## define model list
mod_list <- list(B = "identity", U = "zero", Q = matrix("q"),
    Z = "identity", A = matrix("a"), R = matrix("r"))

## fit the model with MARSS
fit <- MARSS(matrix(Nile, nrow = 1), mod_list)
```

```
Success! abstol and log-log tests passed at 82 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 82 iterations.
Log-likelihood: -637.7569
AIC: 1283.514   AICc: 1283.935

       Estimate
A.a      -0.338
R.r   15135.796
Q.q    1381.153
x0.x0  1111.791
Initial states (x0) defined at t=0

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



9.3.1 Stochastic level with drift

We can add a drift term to the level model to allow the level to tend upward or downward with a deterministic rate η. This is a random walk with bias.

$$\begin{gathered} y_t = \alpha_t + e_t \\ \alpha_t = \alpha_{t-1} + \eta + w_t \\ \Downarrow \\ y_t = x_t + v_t \\ x_t = x_{t-1} + u + w_t \end{gathered} \tag{9.9}$$

We can allow that the drift term η evolves over time along with the level. In this case, η is modeled as a random walk along with α. This model is

$$\begin{gathered} y_t = \alpha_t + e_t \\ \alpha_t = \alpha_{t-1} + \eta_{t-1} + w_{\alpha,t} \\ \eta_t = \eta_{t-1} + w_{\eta,t} \end{gathered} \tag{9.10}$$

Equation (9.10) can be written in matrix form as:

$$\begin{gathered} y_t = \begin{bmatrix}1 & 0\end{bmatrix} \begin{bmatrix}\alpha \\ \eta\end{bmatrix}_t + v_t \\ \begin{bmatrix}\alpha \\ \eta\end{bmatrix}_t = \begin{bmatrix}1 & 1 \\ 0 & 1\end{bmatrix} \begin{bmatrix}\alpha \\ \eta\end{bmatrix}_{t-1} + \begin{bmatrix}w_\alpha \\ w_\eta\end{bmatrix}_t \end{gathered} \tag{9.11}$$

Equation (9.11) is a MARSS model.

$$y_t = \mathbf{Z}\mathbf{x}_t + v_t\mathbf{x}_t = \mathbf{B}\mathbf{x}_{t-1} + \mathbf{w}_t \tag{9.12}$$

where $\mathbf{B} = \begin{bmatrix}1 & 1 \\ 0 & 1\end{bmatrix}$, $\mathbf{x} = \begin{bmatrix}\alpha \\ \eta\end{bmatrix}$ and $\mathbf{Z} = \begin{bmatrix}1 & 0\end{bmatrix}$.

See Section 6.2 for more discussion of stochastic level models and Section @ref() to see how to fit this model with the `StructTS(sec-uss-the-structts-function)` function in the **stats** package.

9.4 Stochastic regression model

The stochastic level models in Section 9.3 do not have predictor variables (covariates). Let's add one predictor variable f_t and write a simple DLM where the intercept α and slope β are stochastic. We will specify that α and β evolve according to a simple random walk. Normally x is used for the predictor variables in a regression model, but we will avoid that since we are using x for the state equation in a state-space model. This model is

$$y_t = \alpha_t + \beta_t f_t + v_t \alpha_t = \alpha_{t-1} + w_{\alpha,t} \beta_t = \beta_{t-1} + w_{\beta,t} \tag{9.13}$$

Written in matrix form, the model is

$$y_t = \begin{bmatrix}1 & f_t\end{bmatrix}\begin{bmatrix}\alpha\\ \beta\end{bmatrix}_t + v_t \begin{bmatrix}\alpha\\ \beta\end{bmatrix}_t = \begin{bmatrix}\alpha\\ \beta\end{bmatrix}_{t-1} + \begin{bmatrix}w_\alpha\\ w_\beta\end{bmatrix}_t \tag{9.14}$$

Equation (9.14) is a MARSS model:

$$y_t = \mathbf{Z}\mathbf{x}_t + v_t \mathbf{x}_t = \mathbf{x}_{t-1} + \mathbf{w}_t \tag{9.15}$$

where $\mathbf{x} = \begin{bmatrix}\alpha\\ \beta\end{bmatrix}$ and $\mathbf{Z} = \begin{bmatrix}1 & f_t\end{bmatrix}$.

9.5 DLM with seasonal effect

Let's add a simple fixed quarter effect to the regression model:

$$y_t = \alpha_t + \beta_t x_t + \gamma_{qtr} + e_t \gamma_{qtr} = \begin{cases}\gamma_1 & \text{if } qtr = 1\\ \gamma_2 & \text{if } qtr = 2\\ \gamma_3 & \text{if } qtr = 3\\ \gamma_4 & \text{if } qtr = 4\end{cases} \tag{9.16}$$

We can write Equation (9.16) in matrix form. In our model for γ, we will set the variance to 0 so that the γ does not change with time.

$$y_t = \begin{bmatrix}1 & x_t & 1\end{bmatrix} \begin{bmatrix}\alpha \\ \beta \\ \gamma_{qtr}\end{bmatrix}_t + e_t \begin{bmatrix}\alpha \\ \beta \\ \gamma_{qtr}\end{bmatrix}_t = \begin{bmatrix}\alpha \\ \beta \\ \gamma_{qtr}\end{bmatrix}_{t-1} + \begin{bmatrix}w_\alpha \\ w_\beta \\ 0\end{bmatrix}_t \Downarrow y_t = \mathbf{Z}_t\mathbf{x}_t + v_t\mathbf{x}_t = \mathbf{x}_{t-1} + \mathbf{w}_t \tag{9.17}$$

How do we select the right quarterly effect? Let's separate out the quarterly effects and add them to **x**. We could then select the right γ using 0s and 1s in the $\mathbf{Z}_t$ matrix. For example, if t is in quarter 1, our model would be

$$y_t = \begin{bmatrix}1 & x_t & 1 & 0 & 0 & 0\end{bmatrix} \begin{bmatrix}\alpha_t \\ \beta_t \\ \gamma_1 \\ \gamma_2 \\ \gamma_3 \\ \gamma_4\end{bmatrix} \tag{9.18}$$

While if t is in quarter 2, the model is

$$y_t = \begin{bmatrix}1 & x_t & 0 & 1 & 0 & 0\end{bmatrix} \begin{bmatrix}\alpha_t \\ \beta_t \\ \gamma_1 \\ \gamma_2 \\ \gamma_3 \\ \gamma_4\end{bmatrix} \tag{9.19}$$

This would work, but we would have to have a different $\mathbf{Z}_t$ matrix and it might get cumbersome to keep track of the 0s and 1s. If we wanted the γ to evolve with time, we might need to do this. However, if the γ are fixed, i.e. the quarterly effect does not change over time, a less cumbersome approach is possible.

We could instead keep the $\mathbf{Z}_t$ matrix the same, but reorder the γ_i within **x**. If t is in quarter 1,

$$y_t = \begin{bmatrix} 1 & x_t & 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \alpha_t \\ \beta_t \\ \gamma_1 \\ \gamma_2 \\ \gamma_3 \\ \gamma_4 \end{bmatrix} \tag{9.20}$$

While if t is in quarter 2,

$$y_t = \begin{bmatrix} 1 & x_t & 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \alpha_t \\ \beta_t \\ \gamma_2 \\ \gamma_3 \\ \gamma_4 \\ \gamma_1 \end{bmatrix} \tag{9.21}$$

We can use a non-diagonal $\mathbf{G}$ to to shift the correct quarter effect within $\mathbf{x}$.

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

With this $\mathbf{G}$, the γ rotate within $\mathbf{x}$ with each time step. If t is in quarter 1, then $t+1$ is in quarter 2, and we want γ_2 to be in the 3rd row.

$$\begin{bmatrix} \alpha \\ \beta \\ \gamma_2 \\ \gamma_3 \\ \gamma_4 \\ \gamma_1 \end{bmatrix}_{t+1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \\ \gamma_1 \\ \gamma_2 \\ \gamma_3 \\ \gamma_4 \end{bmatrix}_t + \begin{bmatrix} w_\alpha \\ w_\beta \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}_t \tag{9.22}$$

At $t+2$, we are in quarter 3 and γ_3 will be in row 3.

$$\begin{bmatrix}\alpha\\ \beta\\ \gamma_3\\ \gamma_4\\ \gamma_1\\ \gamma_2\end{bmatrix}_{t+2} = \begin{bmatrix}1&0&0&0&0&0\\0&1&0&0&0&0\\0&0&0&1&0&0\\0&0&0&0&1&0\\0&0&0&0&0&1\\0&0&1&0&0&0\end{bmatrix}\begin{bmatrix}\alpha\\ \beta\\ \gamma_2\\ \gamma_3\\ \gamma_4\\ \gamma_1\end{bmatrix}_{t+1} + \begin{bmatrix}w_\alpha\\ w_\beta\\ 0\\ 0\\ 0\\ 0\end{bmatrix}_t \tag{9.23}$$

9.6 Analysis of salmon survival

Let's see an example of a DLM used to analyze real data from the literature. Scheuerell and Williams (2005) used a DLM to examine the relationship between marine survival of Chinook salmon and an index of ocean upwelling strength along the west coast of the USA. Upwelling brings cool, nutrient-rich waters from the deep ocean to shallower coastal areas. Scheuerell & Williams hypothesized that stronger upwelling in April should create better growing conditions for phytoplankton, which would then translate into more zooplankton. In turn, juvenile salmon ("smolts") entering the ocean in May and June should find better foraging opportunities. Thus, for smolts entering the ocean in year t,

$$survival_t = \alpha_t + \beta_t f_t + v_t \text{ with } v_t \sim \text{N}(0, r), \tag{9.24}$$

and f_t is the coastal upwelling index (cubic meters of seawater per second per 100 m of coastline) for the month of April in year t.

Both the intercept and slope are time varying, so

$$\alpha_t = \alpha_{t-1} + w_{\alpha,t} \text{ with } w_{\alpha,t} \sim \text{N}(0, q_\alpha) \tag{9.25}$$

$$\beta_t = \beta_{t-1} + w_{\beta,t} \text{ with } w_{\beta,t} \sim \text{N}(0, q_\beta). \tag{9.26}$$

If we define $\boldsymbol{\theta}_t = \begin{bmatrix}\alpha\\ \beta\end{bmatrix}_t$, $\mathbf{G}_t = \mathbf{I}$, $\mathbf{w}_t = \begin{bmatrix}w_\alpha\\ w_\beta\end{bmatrix}_t$, and $\mathbf{Q} = \begin{bmatrix}q_\alpha & 0\\ 0 & q_\beta\end{bmatrix}$, we get Equation (9.4). If we define $y_t = survival_t$ and $\mathbf{F}_t = \begin{bmatrix}1\\ f_t\end{bmatrix}$, we can write out the full DLM as a state-space model with the following form:

$$y_t = \mathbf{F}_t^\top \boldsymbol{\theta}_t + v_t \text{ with } v_t \sim \text{N}(0, r)\boldsymbol{\theta}_t = \mathbf{G}_t \boldsymbol{\theta}_{t-1} + \mathbf{w}_t \text{ with } \mathbf{w}_t \sim \text{MVN}(\mathbf{0}, \mathbf{Q})\boldsymbol{\theta}_0 = \boldsymbol{\pi}_0. \tag{9.27}$$

Equation (9.27) is equivalent to our standard MARSS model:

$$\mathbf{y}_t = \mathbf{Z}_t \mathbf{x}_t + \mathbf{a} + \mathbf{v}_t \text{ with } \mathbf{v}_t \sim \text{MVN}(0, \mathbf{R}_t)\mathbf{x}_t = \mathbf{B}_t \mathbf{x}_{t-1} + \mathbf{u} + \mathbf{w}_t \text{ with } \mathbf{w}_t \sim \text{MVN}(0, \mathbf{Q}_t)\mathbf{x}_0 = \boldsymbol{\pi} \tag{9.28}$$

where $\mathbf{x}_t = \boldsymbol{\theta}_t$, $\mathbf{B}_t = \mathbf{G}_t$, $\mathbf{y}_t = y_t$ (i.e., $\mathbf{y}_t$ is 1×1), $\mathbf{Z}_t = \mathbf{F}_t^\top$, $\mathbf{a} = \mathbf{u} = \mathbf{0}$, and $\mathbf{R}_t = r$ (i.e., $\mathbf{R}_t$ is 1×1).

9.7 Fitting with `MARSS()`

Now let's go ahead and analyze the DLM specified in Equations (9.24)–(9.27). We begin by loading the data set (which is in the **MARSS** package). The data set has 3 columns for 1) the year the salmon smolts migrated to the ocean (`year`), 2) logit-transformed survival [1] (`logit.s`), and 3) the coastal upwelling index for April (`CUI.apr`). There are 42 years of data (1964–2005).

```r
## load the data
data(SalmonSurvCUI, package = "MARSS")
## get time indices
years <- SalmonSurvCUI[, 1]
## number of years of data
TT <- length(years)
## get response variable: logit(survival)
dat <- matrix(SalmonSurvCUI[, 2], nrow = 1)
```

As we have seen in other case studies, standardizing our covariate(s) to have zero-mean and unit-variance can be helpful in model fitting and interpretation. In this case, it's a good idea because the variance of `CUI.apr` is orders of magnitude greater than `logit.s`.

[1] Survival in the original context was defined as the proportion of juveniles that survive to adulthood. Thus, we use the logit function, defined as $logit(p) = log_e(p/[1-p])$, to map survival from the open interval (0,1) onto the interval $(-\infty, \infty)$, which allows us to meet our assumption of normally distributed observation errors.

```
## get predictor variable
CUI <- SalmonSurvCUI[, 3]
## z-score the CUI
CUI_z <- matrix((CUI - mean(CUI))/sqrt(var(CUI)), nrow = 1)
## number of regr params (slope + intercept)
m <- dim(CUI_z)[1] + 1
```

Plots of logit-transformed survival and the z-scored April upwelling index are shown in Figure 9.1.

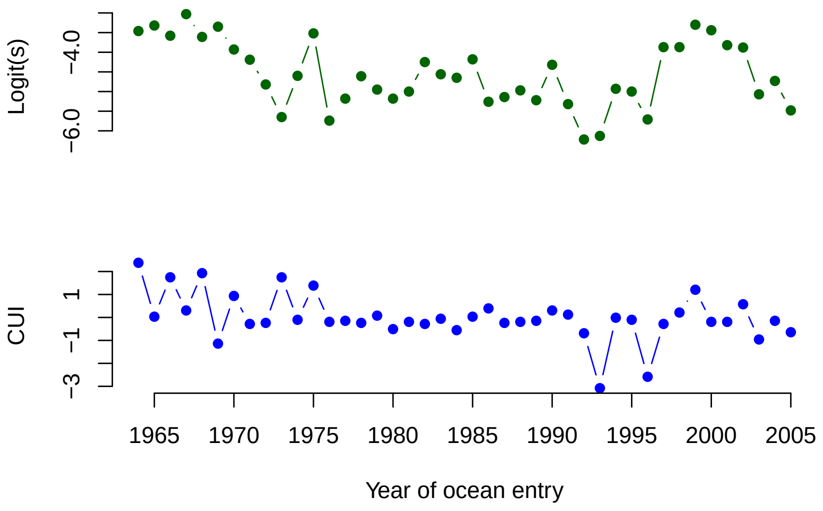


Figure 9.1: Time series of logit-transformed marine survival estimates for Snake River spring/summer Chinook salmon (top) and z-scores of the coastal upwelling index at 45N 125W (bottom). The x-axis indicates the year that the salmon smolts entered the ocean.

Next, we need to set up the appropriate matrices and vectors for MARSS. Let's begin with those for the process equation because they are straightforward.

```
## for process eqn
B <- diag(m)  ## 2x2; Identity
U <- matrix(0, nrow = m, ncol = 1)  ## 2x1; both elements = 0
```

```
Q <- matrix(list(0), m, m)  ## 2x2; all 0 for now
diag(Q) <- c("q.alpha", "q.beta")  ## 2x2; diag = (q1,q2)
```

Defining the correct form for the observation model is a little more tricky, however, because of how we model the effect(s) of predictor variables. In a DLM, we need to use $\mathbf{Z}_t$ (instead of $\mathbf{d}_t$) as the matrix of predictor variables that affect $\mathbf{y}_t$, and we use $\mathbf{x}_t$ (instead of $\mathbf{D}_t$) as the regression parameters. Therefore, we need to set $\mathbf{Z}_t$ equal to an $n \times m \times T$ array, where n is the number of response variables (= 1; y_t is univariate), m is the number of regression parameters (= intercept + slope = 2), and T is the length of the time series (= 42).

```
## for observation eqn
Z <- array(NA, c(1, m, TT))  ## NxMxT; empty for now
Z[1, 1, ] <- rep(1, TT)  ## Nx1; 1's for intercept
Z[1, 2, ] <- CUI_z  ## Nx1; predictor variable
A <- matrix(0)  ## 1x1; scalar = 0
R <- matrix("r")  ## 1x1; scalar = r
```

Lastly, we need to define our lists of initial starting values and model matrices/vectors.

```
## only need starting values for regr parameters
inits_list <- list(x0 = matrix(c(0, 0), nrow = m))

## list of model matrices & vectors
mod_list <- list(B = B, U = U, Q = Q, Z = Z, A = A, R = R)
```

And now we can fit our DLM with MARSS.

```
## fit univariate DLM
dlm_1 <- MARSS(dat, inits = inits_list, model = mod_list)
```

```
Success! abstol and log-log tests passed at 115 iterations.
Alert: conv.test.slope.tol is 0.5.
Test with smaller values (<0.1) to ensure convergence.
```

```
MARSS fit is
Estimation method: kem
Convergence test: conv.test.slope.tol = 0.5, abstol = 0.001
Estimation converged in 115 iterations.
Log-likelihood: -40.03813
AIC: 90.07627   AICc: 91.74293

          Estimate
R.r        0.15708
Q.q.alpha  0.11264
Q.q.beta   0.00564
x0.X1     -3.34023
x0.X2     -0.05388
Initial states (x0) defined at t=0

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

Notice that the MARSS output does not list any estimates of the regression parameters themselves. Why not? Remember that in a DLM the matrix of states ($\mathbf{x}$) contains the estimates of the regression parameters ($\boldsymbol{\theta}$). Therefore, we need to look in `dlm_1$states` for the MLEs of the regression parameters, and in `dlm_1$states.se` for their standard errors.

Time series of the estimated intercept and slope are shown in Figure 9.2. It appears as though the intercept is much more dynamic than the slope, as indicated by a much larger estimate of process variance for the former (`Q.q1`). In fact, although the effect of April upwelling appears to be increasing over time, it doesn't really become important as a predictor variable until about 1990 when the approximate 95% confidence interval for the slope no longer overlaps zero.

9.8 Forecasting

Forecasting from a DLM involves two steps:

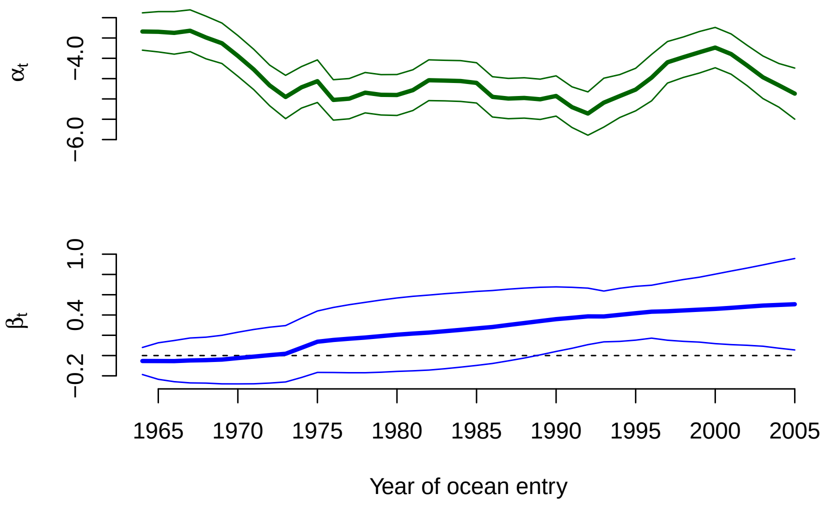


Figure 9.2: Time series of estimated mean states (thick lines) for the intercept (top) and slope (bottom) parameters from the DLM specified by Equations (9.24)–(9.27). Thin lines denote the mean ± 2 standard deviations.

1. Get an estimate of the regression parameters at time t from data up to time $t-1$. These are also called the one-step ahead forecast (or prediction) of the regression parameters.

2. Make a prediction of y at time t based on the predictor variables at time t and the estimate of the regression parameters at time t (step 1). This is also called the one-step ahead forecast (or prediction) of the observation.

9.8.1 Estimate of the regression parameters

For step 1, we want to compute the distribution of the regression parameters at time t conditioned on the data up to time $t-1$, also known as the one-step ahead forecasts of the regression parameters. Let's denote $\boldsymbol{\theta}_{t-1}$ conditioned on $y_{1:t-1}$ as $\boldsymbol{\theta}_{t-1|t-1}$ and denote $\boldsymbol{\theta}_t$ conditioned on $y_{1:t-1}$ as $\boldsymbol{\theta}_{t|t-1}$. We will start by defining the distribution of $\boldsymbol{\theta}_{t|t}$ as follows

$$\boldsymbol{\theta}_{t|t} \sim \text{MVN}(\boldsymbol{\pi}_t, \boldsymbol{\Lambda}_t) \tag{9.29}$$

where $\boldsymbol{\pi}_t = \text{E}(\boldsymbol{\theta}_{t|t})$ and $\boldsymbol{\Lambda}_t = \text{Var}(\boldsymbol{\theta}_{t|t})$.

Now we can compute the distribution of $\boldsymbol{\theta}_t$ conditioned on $y_{1:t-1}$ using the process equation for $\boldsymbol{\theta}$:

$$\boldsymbol{\theta}_t = \mathbf{G}_t\boldsymbol{\theta}_{t-1} + \mathbf{w}_t,\ \mathbf{w}_t \sim \text{MVN}(\mathbf{0}, \mathbf{Q}) \tag{9.30}$$

The expected value of $\boldsymbol{\theta}_{t|t-1}$ is thus

$$\text{E}(\boldsymbol{\theta}_{t|t-1}) = \mathbf{G}_t\text{E}(\boldsymbol{\theta}_{t-1|t-1}) = \mathbf{G}_t\boldsymbol{\pi}_{t-1} \tag{9.31}$$

The variance of $\boldsymbol{\theta}_{t|t-1}$ is

$$\text{Var}(\boldsymbol{\theta}_{t|t-1}) = \mathbf{G}_t\text{Var}(\boldsymbol{\theta}_{t-1|t-1})\mathbf{G}_t^\top + \mathbf{Q} = \mathbf{G}_t\boldsymbol{\Lambda}_{t-1}\mathbf{G}_t^\top + \mathbf{Q} \tag{9.32}$$

Thus the distribution of $\boldsymbol{\theta}_t$ conditioned on $y_{1:t-1}$ is

$$\text{E}(\boldsymbol{\theta}_{t|t-1}) \sim \text{MVN}(\mathbf{G}_t\boldsymbol{\pi}_{t-1}, \mathbf{G}_t\boldsymbol{\Lambda}_{t-1}\mathbf{G}_t^\top + \mathbf{Q}) \tag{9.33}$$

9.8.2 Prediction of the response variable y_t

For step 2, we make the prediction of y_t given the predictor variables at time t and the estimate of the regression parameters at time t. This is called the one-step ahead prediction for the observation at time t. We will denote the prediction of y as $\hat{y}$ and we want to compute its distribution (mean and variance). We do this using the equation for y_t but substituting the expected value of $\boldsymbol{\theta}_{t|t-1}$ for $\boldsymbol{\theta}_t$.

$$\hat{y}_{t|t-1} = \mathbf{F}_t^\top\text{E}(\boldsymbol{\theta}_{t|t-1}) + e_t,\ e_t \sim \text{N}(0, r) \tag{9.34}$$

Our prediction of y at t has a normal distribution with mean (expected value) and variance. The expected value of $\hat{y}_{t|t-1}$ is

$$\text{E}(\hat{y}_{t|t-1}) = \mathbf{F}_t^\top\text{E}(\boldsymbol{\theta}_{t|t-1}) = \mathbf{F}_t^\top(\mathbf{G}_t\boldsymbol{\pi}_{t-1}) \tag{9.35}$$

and the variance of $\hat{y}_{t|t-1}$ is

$$\begin{aligned}\text{Var}(\hat{y}_{t|t-1}) &= \mathbf{F}_t^\top \text{Var}(\boldsymbol{\theta}_{t|t-1})\mathbf{F}_t + r &(9.36)\\ &= \mathbf{F}_t^\top(\mathbf{G}_t\boldsymbol{\Lambda}_{t-1}\mathbf{G}_t^\top + \mathbf{Q})\mathbf{F}_t + r &(9.37)\end{aligned}$$

9.8.3 Computing the prediction

The expectations and variance of $\boldsymbol{\theta}_t$ conditioned on $y_{1:t}$ and $y_{1:t-1}$ are standard output from the Kalman filter. Thus to produce the predictions, all we need to do is run our DLM state-space model through a Kalman filter to get $\text{E}(\boldsymbol{\theta}_{t|t-1})$ and $\text{Var}(\boldsymbol{\theta}_{t|t-1})$ and then use Equation (9.35) to compute the mean prediction and Equation (9.36) to compute its variance.

The Kalman filter will need $\mathbf{F}_t$, $\mathbf{G}_t$ and estimates of $\mathbf{Q}$ and r. The latter are calculated by fitting the DLM to the data $y_{1:t}$, using for example the `MARSS()` function.

Let's see an example with the salmon survival DLM. We will use the Kalman filter function in the **MARSS** package and the DLM fit from `MARSS()`.

9.8.4 Forecasting salmon survival

Scheuerell and Williams (2005) were interested in how well upwelling could be used to actually *forecast* expected survival of salmon, so let's look at how well our model does in that context. To do so, we need the predictive distribution for the survival at time t given the upwelling at time t and the predicted regression parameters at t.

In the salmon survival DLM, the $\mathbf{G}_t$ matrix is the identity matrix, thus the mean and variance of the one-step ahead predictive distribution for the observation at time t reduces to (from Equations (9.35) and (9.36))

$$\begin{gathered}\text{E}(\hat{y}_{t|t-1}) = \mathbf{F}_t^\top \text{E}(\boldsymbol{\theta}_{t|t-1})\\ \text{Var}(\hat{y}_{t|t-1}) = \mathbf{F}_t^\top \text{Var}(\boldsymbol{\theta}_{t|t-1})\mathbf{F}_t + \hat{r}\end{gathered} \tag{9.38}$$

where

$$\mathbf{F}_t = \begin{bmatrix} 1 \\ f_t \end{bmatrix}$$

and f_t is the upwelling index at $t+1$. $\hat{r}$ is the estimated observation variance from our model fit.

9.8.5 Forecasting using MARSS

Working from Equation (9.38), we can compute the expected value of the forecast at time t and its variance using the Kalman filter. For the expectation, we need $\mathbf{F}_t^\top \text{E}(\boldsymbol{\theta}_{t|t-1})$. $\mathbf{F}_t^\top$ is called $\mathbf{Z}_t$ in MARSS notation. The one-step ahead forecasts of the regression parameters at time t, the $\text{E}(\boldsymbol{\theta}_{t|t-1})$, are calculated as part of the Kalman filter algorithm—they are termed $\tilde{x}_t^{t-1}$ in MARSS notation and stored as `xtt1` in the list produced by the `MARSSkfss()` Kalman filter function.

Using the `Z` defined in 9.6, we compute the mean forecast as follows:

```r
## get list of Kalman filter output
kf_out <- MARSSkfss(dlm_1)

## forecasts of regr parameters; 2xT matrix
eta <- kf_out$xtt1

## ts of E(forecasts)
fore_mean <- vector()
for (t in 1:TT) {
    fore_mean[t] <- Z[, , t] %*% eta[, t, drop = FALSE]
}
```

For the variance of the forecasts, we need $\mathbf{F}_t^\top \text{Var}(\boldsymbol{\theta}_{t|t-1})\mathbf{F}_t + \hat{r}$. As with the mean, $\mathbf{F}_t^\top \equiv \mathbf{Z}_t$. The variances of the one-step ahead forecasts of the regression parameters at time t, $\text{Var}(\boldsymbol{\theta}_{t|t-1})$, are also calculated as part of the Kalman filter algorithm—they are stored as `Vtt1` in the list produced by the `MARSSkfss()` function. Lastly, the observation variance $\hat{r}$ was estimated when we fit the DLM to the data using `MARSS()` and can be extracted from the `dlm_1` fit.

Putting this together, we can compute the forecast variance:

```r
## variance of regr parameters; 1x2xT array
Phi <- kf_out$Vtt1

## obs variance; 1x1 matrix
R_est <- coef(dlm_1, type = "matrix")$R

## ts of Var(forecasts)
fore_var <- vector()
for (t in 1:TT) {
    tZ <- matrix(Z[, , t], m, 1)  ## transpose of Z
    fore_var[t] <- Z[, , t] %*% Phi[, , t] %*% tZ + R_est
}
```

Plots of the model mean forecasts with their estimated uncertainty are shown in Figure 9.3. Nearly all of the observed values fell within the approximate prediction interval. Notice that we have a forecasted value for the first year of the time series (1964), which may seem at odds with our notion of forecasting at time t based on data available only through time $t-1$. In this case, however, MARSS is actually estimating the states at $t = 0$ ($\boldsymbol{\theta}_0$), which allows us to compute a forecast for the first time point.

Although our model forecasts look reasonable in logit-space, it is worthwhile to examine how well they look when the survival data and forecasts are back-transformed onto the interval [0,1] (Figure 9.4). In that case, the accuracy does not seem to be affected, but the precision appears much worse, especially during the early and late portions of the time series when survival is changing rapidly.

Notice that we passed the DLM fit to all the data to `MARSSkfss()`. This meant that the Kalman filter used estimates of $\mathbf{Q}$ and r using all the data in the `xtt1` and `Vtt1` calculations. Thus our predictions at time t are not entirely based on only data up to time $t-1$ since the $\mathbf{Q}$ and r estimates were from all the data from 1964 to 2005.

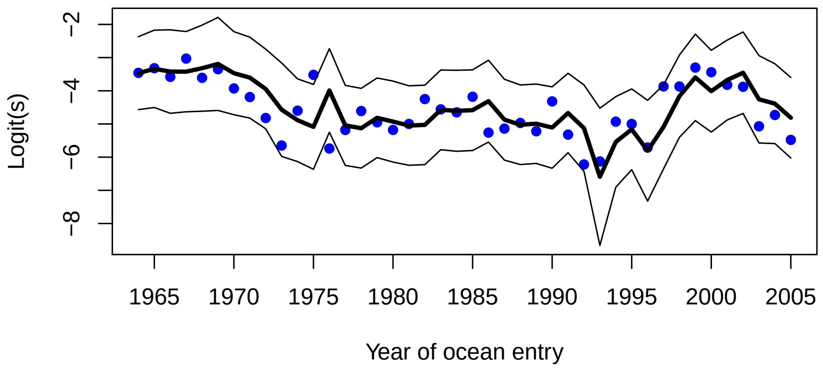


Figure 9.3: Time series of logit-transformed survival data (blue dots) and model mean forecasts (thick line). Thin lines denote the approximate 95% prediction intervals.

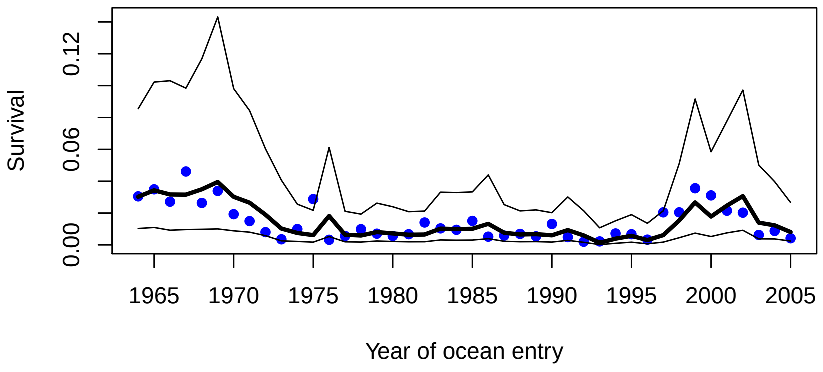


Figure 9.4: Time series of survival data (blue dots) and model mean forecasts (thick line). Thin lines denote the approximate 95% prediction intervals.

9.9 Forecast diagnostics

As with other time series models, evaluation of a DLM should include diagnostics. In a forecasting context, we are often interested in the forecast errors, which are simply the observed data minus the forecasts $e_t = y_t - \text{E}(y_t|y_{1:t-1})$. In particular, the following assumptions should hold true for e_t:

1. $e_t \sim \text{N}(0, \sigma^2)$; 2. $\text{cov}(e_t, e_{t-k}) = 0$.

In the literature on state-space models, the set of e_t are commonly referred to as "innovations". `MARSS()` calculates the innovations as part of the Kalman filter algorithm—they are stored as `Innov` in the list produced by the `MARSSkfss()` function.

```
## forecast errors
innov <- kf_out$Innov
```

Let's see if our innovations meet the model assumptions. Beginning with (1), we can use a Q-Q plot to see whether the innovations are normally distributed with a mean of zero. We'll use the `qqnorm()` function to plot the quantiles of the innovations on the y-axis versus the theoretical quantiles from a Normal distribution on the x-axis. If the 2 distributions are similar, the points should fall on the line defined by $y = x$.

```
## Q-Q plot of innovations
qqnorm(t(innov), main = "", pch = 16, col = "blue")
## add y=x line for easier interpretation
qqline(t(innov))
```

The Q-Q plot (Figure 9.5) indicates that the innovations appear to be more-or-less normally distributed (i.e., most points fall on the line). Furthermore, it looks like the mean of the innovations is about 0, but we should use a more reliable test than simple visual inspection. We can formally test whether the mean of the innovations is significantly different from 0 by using a one-sample t-test. based on a null hypothesis of $\text{E}(e_t) = 0$. To do so, we will use the function `t.test()` and base our inference on a significance value of $\alpha = 0.05$.

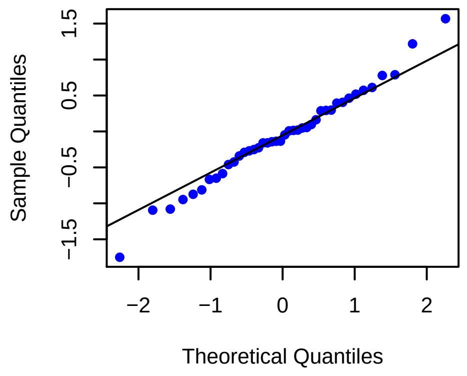


Figure 9.5: Q-Q plot of the forecast errors (innovations) for the DLM specified in Equations (9.24)–(9.27).

```
## p-value for t-test of H0: E(innov) = 0
t.test(t(innov), mu = 0)$p.value
```

```
[1] 0.4840901
```

The p-value >> 0.05 so we cannot reject the null hypothesis that $\text{E}(e_t) = 0$.

Moving on to assumption (2), we can use the sample autocorrelation function (ACF) to examine whether the innovations covary with a time-lagged version of themselves. Using the `acf()` function, we can compute and plot the correlations of e_t and e_{t-k} for various values of k. Assumption (2) will be met if none of the correlation coefficients exceed the 95% confidence intervals defined by $\pm\, z_{0.975}/\sqrt{n}$.

```
## plot ACF of innovations
acf(t(innov), lag.max = 10)
```

The ACF plot (Figure 9.6) shows no significant autocorrelation in the innovations at lags 1–10, so it looks like both of our model assumptions have indeed been met.

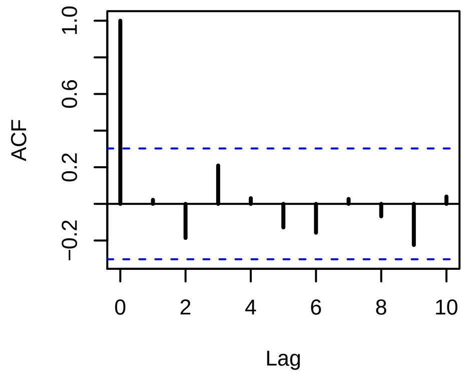


Figure 9.6: Autocorrelation plot of the forecast errors (innovations) for the DLM specified in Equations (9.24)–(9.27). Horizontal blue lines define the upper and lower 95% confidence intervals.

9.10 Homework discussion and data

For the homework this week we will use a DLM to examine some of the time-varying properties of the spawner-recruit relationship for Pacific salmon. Much work has been done on this topic, particularly by Randall Peterman and his students and post-docs at Simon Fraser University. To do so, researchers commonly use a Ricker model because of its relatively simple form, such that the number of recruits (offspring) born in year t (R_t) from the number of spawners (parents) (S_t) is

$$R_t = aS_t e^{-bS+v_t}. \tag{9.39}$$

The parameter a determines the maximum reproductive rate in the absence of any density-dependent effects (the slope of the curve at the origin), b is the strength of density dependence, and $v_t \sim N(0,\sigma)$. In practice, the model is typically log-transformed so as to make it linear with respect to the predictor variable S_t, such that

$$\begin{aligned}
\log(R_t) &= \log(a) + \log(S_t) - bS_t + v_t && (9.40)\\
\log(R_t) - \log(S_t) &= \log(a) - bS_t + v_t && (9.41)\\
\log(R_t/S_t) &= \log(a) - bS_t + v_t. && (9.42)
\end{aligned}$$

Substituting $y_t = \log(R_t/S_t)$, $x_t = S_t$, and $\alpha = \log(a)$ yields a simple linear regression model with intercept α and slope b.

Unfortunately, however, residuals from this simple model typically show high-autocorrelation due to common environmental conditions that affect overlapping generations. Therefore, to correct for this and allow for an index of stock productivity that controls for any density-dependent effects, the model may be re-written as

$$\begin{aligned} \log(R_t/S_t) &= \alpha_t - bS_t + v_t, && (9.43) \\ \alpha_t &= \alpha_{t-1} + w_t, && (9.44) \end{aligned}$$

and $w_t \sim N(0, q)$. By treating the brood-year specific productivity as a random walk, we allow it to vary, but in an autocorrelated manner so that consecutive years are not independent from one another.

More recently, interest has grown in using covariates (*e.g.*, sea-surface temperature) to explain the interannual variability in productivity. In that case, we can can write the model as

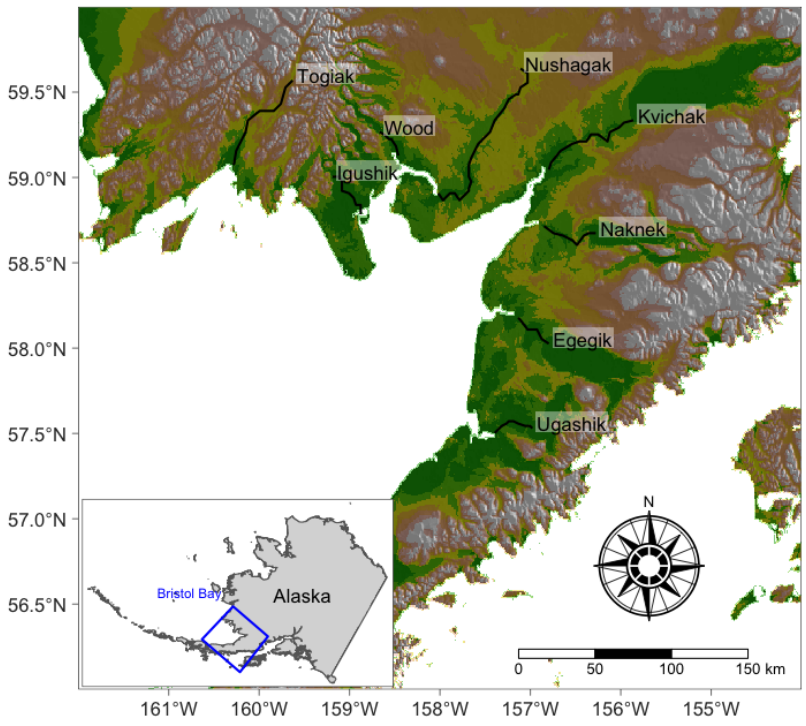
$$\log(R_t/S_t) = \alpha + \delta_t X_t - bS_t + v_t. \tag{9.45}$$

In this case we are estimating some base-level productivity (α) plus the time-varying effect of some covariate X_t (δ_t).

9.10.1 Spawner-recruit data

The data come from a large public database begun by Ransom Myers many years ago. If you are interested, you can find lots of time series of spawning-stock, recruitment, and harvest for a variety of fishes around the globe. Here is the website:
https://www.ramlegacy.org/

For this exercise, we will use spawner-recruit data for sockeye salmon (*Oncorhynchus nerka*) from the Kvichak River in SW Alaska that span the years 1952-1989. In addition, we'll examine the potential effects of the Pacific Decadal Oscillation (PDO) during the salmon's first year in the ocean, which is widely believed to be a "bottleneck" to survival.



These data are in the **atsalibrary** package on GitHub. If needed, install using the **devtools** package.

```
library(devtools)
## Windows users will likely need to set this
## Sys.setenv('R_REMOTES_NO_ERRORS_FROM_WARNINGS' = 'true')
devtools::install_github("nwfsc-timeseries/atsalibrary")
```

Load the data.

```
data(KvichakSockeye, package = "atsalibrary")
SR_data <- KvichakSockeye
```

The data are a dataframe with columns for brood year (`brood_year`), number of spawners (`spawners`), number of recruits (`recruits`) and PDO at year $t-2$ in summer (`pdo_summer_t2`) and in winter (`pdo_winter_t2`).

```
## head of data file
head(SR_data)
```

```
# A tibble: 6 x 5
# Groups:   brood_year [6]
  brood_year spawners recruits pdo_summer_t2 pdo_winter_t2
       <dbl>    <dbl>    <dbl>         <dbl>         <dbl>
1       1952       NA    20200         -2.79         -1.68
2       1953       NA      593         -1.2          -1.05
3       1954       NA      799         -1.85         -1.25
4       1955       NA     1500         -0.6          -0.68
5       1956     9440    39000         -0.5          -0.31
6       1957     2840     4090         -2.36         -1.78
```

9.11 Problems

Use the information and data in the previous section to answer the following questions. Note that if any model is not converging, then you will need to increase the `maxit` parameter in the `control` argument/list that gets passed to `MARSS()`. For example, you might try `control=list(maxit=2000)`.

1. Begin by fitting a reduced form of Equation (9.43) that includes only a time-varying level (α_t) and observation error (v_t). That is,

$$\begin{aligned}\log(R_t) &= \alpha_t + \log(S_t) + v_t\\ \log(R_t/S_t) &= \alpha_t + v_t\end{aligned}$$

This model assumes no density-dependent survival in that the number of recruits is an ascending function of spawners. Plot the ts of α_t and note the AICc for this model. Also plot appropriate model diagnostics. `residuals()` will return the innovations residuals for your fits.

2. Fit the full model specified by Equation (9.43). For this model, obtain the time series of α_t, which is an estimate of the stock productivity in the absence of density-dependent effects. How do these estimates of productivity compare to those from the previous question? Plot the ts of α_t and note the AICc for this model. Also plot appropriate model diagnostics. (*Hint*: If you don't want a parameter to vary with time, what does that say about its process variance?)

3. Fit the model specified by Equation (9.45) with the summer PDO index as the covariate (`pdo_summer_t2`). What is the mean level of productivity? Plot the ts of δ_t and note the AICc for this model. Also plot appropriate model diagnostics.

4. Fit the model specified by Equation (9.45) with the winter PDO index as the covariate (`pdo_winter_t2`). What is the mean level of productivity? Plot the ts of δ_t and note the AICc for this model. Also plot appropriate model diagnostics.

5. Based on AICc, which of the models above is the most parsimonious? Is it well behaved (*i.e.*, are the model assumptions met)? Plot the model forecasts for the best model. Is this a good forecast model?

Chapter 10

Dynamic Factor Analysis

Here we will use the **MARSS** package to do Dynamic Factor Analysis (DFA), which allows us to look for a set of common underlying processes among a relatively large set of time series (Zuur et al., 2003). There have been a number of recent applications of DFA to ecological questions surrounding Pacific salmon (Stachura et al., 2014; Jorgensen et al., 2016; Ohlberger et al., 2016) and stream temperatures (Lisi et al., 2015). For a more in-depth treatment of potential applications of MARSS models for DFA, see Chapter 9 in the MARSS User's Guide.

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

Data and packages

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

```
library(MARSS)
```

10.1 Introduction

DFA is conceptually different than what we have been doing in the previous applications. Here we are trying to explain temporal variation in a set of n

observed time series using linear combinations of a set of m hidden random walks, where $m << n$. A DFA model is a type of MARSS model with the following structure:

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

This equation should look rather familiar as it is exactly the same form we used for estimating varying number of processes from a set of observations in Lesson II. The difference with DFA is that rather than fixing the elements within $\mathbf{Z}$ at 1 or 0 to indicate whether an observation does or does not correspond to a trend, we will instead estimate them as “loadings” on each of the states/processes.

10.2 Example of a DFA model

The general idea is that the observations $\mathbf{y}$ are modeled as a linear combination of hidden processes $\mathbf{x}$ and factor loadings $\mathbf{Z}$ plus some offsets $\mathbf{a}$. Imagine a case where we had a data set with five observed time series ($n = 5$) and we want to fit a model with three hidden processes ($m = 3$). If we write out our DFA model in MARSS matrix form, the observation equation would look like

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

and the process model would look like

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}_t = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}_{t-1} + \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}_t \tag{10.3}$$

The observation errors would be

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

And the process errors would be

$$\begin{bmatrix}w_1\\w_2\\w_3\end{bmatrix}_t \sim \text{MVN}\left(\begin{bmatrix}0\\0\\0\end{bmatrix}, \begin{bmatrix}q_{11}&q_{12}&q_{13}\\q_{12}&q_{22}&q_{23}\\q_{13}&q_{23}&q_{33}\end{bmatrix}\right). \tag{10.5}$$

10.3 Constraining a DFA model

If **a**, **Z**, and **Q** are not constrained, the DFA model above is unidentifiable. Nevertheless, we can use the following parameter constraints to make the model identifiable:

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

Using these constraints, the observation equation for the DFA model above becomes

$$\begin{bmatrix}y_1\\y_2\\y_3\\y_4\\y_5\end{bmatrix}_t = \begin{bmatrix}z_{11}&0&0\\z_{21}&z_{22}&0\\z_{31}&z_{32}&z_{33}\\z_{41}&z_{42}&z_{43}\\z_{51}&z_{52}&z_{53}\end{bmatrix} \begin{bmatrix}x_1\\x_2\\x_3\end{bmatrix}_t + \begin{bmatrix}0\\0\\0\\0\\0\end{bmatrix} + \begin{bmatrix}v_1\\v_2\\v_3\\v_4\\v_5\end{bmatrix}_t. \tag{10.6}$$

and the process equation becomes

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}_t = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}_{t-1} + \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}_t \tag{10.7}$$

The distribution of the observation errors would stay the same, such that

$$\begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{bmatrix}_t \sim \text{MVN}\left(\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} & r_{15} \\ r_{12} & r_{22} & r_{23} & r_{24} & r_{25} \\ r_{13} & r_{23} & r_{33} & r_{34} & r_{35} \\ r_{14} & r_{24} & r_{34} & r_{44} & r_{45} \\ r_{15} & r_{25} & r_{35} & r_{45} & r_{55} \end{bmatrix} \right). \tag{10.8}$$

but the distribution of the process errors would become

$$\begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}_t \sim \text{MVN}\left(\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \right), \tag{10.9}$$

10.4 Different error structures

The example observation equation we used above had what we refer to as an "unconstrained" variance-covariance matrix **R** wherein all of the parameters are unique. In certain applications, however, we may want to change our assumptions about the forms for **R**. For example, we might have good reason to believe that all of the observations have different error variances and they were independent of one another (e.g., different methods were used for sampling), in which case

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

Alternatively, we might have a situation where all of the observation errors had the same variance r, but they were not independent from one another. In that case we would have to include a covariance parameter k, such that

$$\mathbf{R} = \begin{bmatrix} r & k & k & k & k \\ k & r & k & k & k \\ k & k & r & k & k \\ k & k & k & r & k \\ k & k & k & k & r \end{bmatrix}.$$

Any of these options for **R** (and other custom options as well) are available to us in a DFA model, just as they were in the MARSS models used in previous chapters.

10.5 Lake Washington phytoplankton data

For this exercise, we will use the Lake Washington phytoplankton data contained in the **MARSS** package. Let's begin by reading in the monthly values for all of the data, including metabolism, chemistry, and climate.

```
## load the data (there are 3 datasets contained here)
data(lakeWAplankton, package = "MARSS")
## we want lakeWAplanktonTrans, which has been transformed
## so the 0s are replaced with NAs and the data z-scored
all_dat <- lakeWAplanktonTrans
## use only the 10 years from 1980-1989
yr_frst <- 1980
yr_last <- 1989
plank_dat <- all_dat[all_dat[, "Year"] >= yr_frst & all_dat[,
    "Year"] <= yr_last, ]
## create vector of phytoplankton group names
phytoplankton <- c("Cryptomonas", "Diatoms", "Greens", "Unicells",
    "Other.algae")
## get only the phytoplankton
dat_1980 <- plank_dat[, phytoplankton]
```

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

```
## transpose data so time goes across columns
dat_1980 <- t(dat_1980)
## get number of time series
N_ts <- dim(dat_1980)[1]
## get length of time series
TT <- dim(dat_1980)[2]
```

It will be easier to estimate the real parameters of interest if we de-mean the data, so let's do that.

```
y_bar <- apply(dat_1980, 1, mean, na.rm = TRUE)
dat <- dat_1980 - y_bar
rownames(dat) <- rownames(dat_1980)
```

10.5.1 Plots of the data

Here are time series plots of all five phytoplankton functional groups.

```
spp <- rownames(dat_1980)
clr <- c("brown", "blue", "darkgreen", "darkred", "purple")
cnt <- 1
par(mfrow = c(N_ts, 1), mai = c(0.5, 0.7, 0.1, 0.1), omi = c(0,
    0, 0, 0))
for (i in spp) {
    plot(dat[i, ], xlab = "", ylab = "Abundance index", bty = "L",
        xaxt = "n", pch = 16, col = clr[cnt], type = "b")
    axis(1, 12 * (0:dim(dat_1980)[2]) + 1, yr_frst + 0:dim(dat_1980)[2])
    title(i)
    cnt <- cnt + 1
}
```

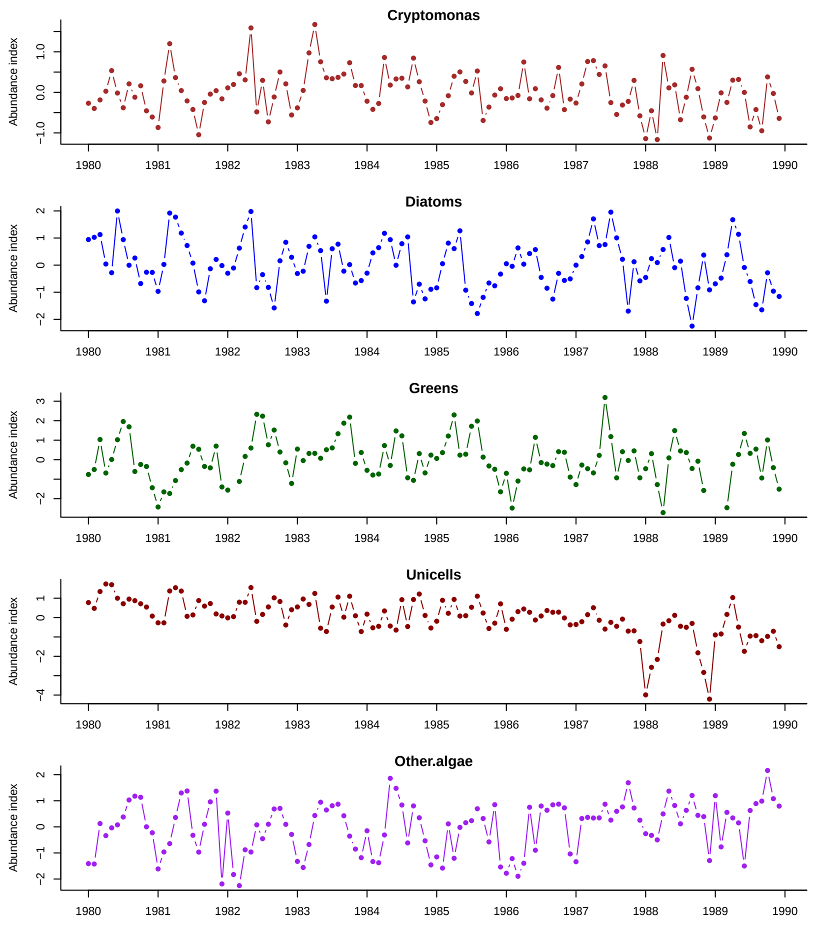


Figure 10.1: Demeaned time series of Lake Washington phytoplankton.

10.6 Fitting DFA models with the MARSS package

The **MARSS** package is designed to work with the fully specified matrix form of the multivariate state-space model we wrote out in Sec 3. Thus, we will need to create a model list with forms for each of the vectors and matrices. Note that even though some of the model elements are scalars and vectors, we will need to specify everything as a matrix (or array for time series of matrices).

Notice that the code below uses some of the **MARSS** shortcuts for specifying forms of vectors and matrices. We will also use the `matrix(list(),nrow,ncol)` trick we learned previously.

10.6.1 The observation model

Here we will fit the DFA model above where we have `R N_ts` observed time series and we want 3 hidden states. Now we need to set up the observation model for `MARSS`. Here are the vectors and matrices for our first model where each nutrient follows its own process. Recall that we will need to set the elements in the upper R corner of **Z** to 0. We will assume that the observation errors have different variances and they are independent of one another.

```
## 'ZZ' is loadings matrix
Z_vals <- list("z11", 0, 0, "z21", "z22", 0, "z31", "z32", "z33",
    "z41", "z42", "z43", "z51", "z52", "z53")
ZZ <- matrix(Z_vals, nrow = N_ts, ncol = 3, byrow = TRUE)
ZZ
```

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

```
## 'aa' is the offset/scaling
aa <- "zero"
## 'DD' and 'd' are for covariates
DD <- "zero"  # matrix(0,mm,1)
dd <- "zero"  # matrix(0,1,wk_last)
## 'RR' is var-cov matrix for obs errors
RR <- "diagonal and unequal"
```

10.6.2 The process model

We need to specify the explicit form for all of the vectors and matrices in the full form of the MARSS model we defined in Sec 3.1. Note that we do not have to specify anything for the states ($\mathbf{x}$) – those are elements that `MARSS` will identify and estimate itself based on our definitions of the other vectors and matrices.

```
## number of processes
mm <- 3
## 'BB' is identity: 1's along the diagonal & 0's elsewhere
BB <- "identity"  # diag(mm)
## 'uu' is a column vector of 0's
uu <- "zero"  # matrix(0, mm, 1)
## 'CC' and 'cc' are for covariates
CC <- "zero"  # matrix(0, mm, 1)
cc <- "zero"  # matrix(0, 1, wk_last)
## 'QQ' is identity
QQ <- "identity"  # diag(mm)
```

10.6.3 Fit the model in MARSS

Now it's time to fit our first DFA model To do so, we need to create three lists that we will need to pass to the `MARSS()` function:

1. A list of specifications for the model's vectors and matrices;
2. A list of any initial values – `MARSS` will pick its own otherwise;

3. A list of control parameters for the `MARSS()` function.

```
## list with specifications for model vectors/matrices
mod_list <- list(Z = ZZ, A = aa, D = DD, d = dd, R = RR, B = BB,
    U = uu, C = CC, c = cc, Q = QQ)
## list with model inits
init_list <- list(x0 = matrix(rep(0, mm), mm, 1))
## list with model control parameters
con_list <- list(maxit = 3000, allow.degen = TRUE)
```

Now we can fit the model.

```
## fit MARSS
dfa_1 <- MARSS(y = dat, model = mod_list, inits = init_list,
    control = con_list)
```

```
Success! abstol and log-log tests passed at 246 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 246 iterations.
Log-likelihood: -692.9795
AIC: 1425.959   AICc: 1427.42

                            Estimate
Z.z11                         0.2738
Z.z21                         0.4487
Z.z31                         0.3170
Z.z41                         0.4107
Z.z51                         0.2553
Z.z22                         0.3608
Z.z32                        -0.3690
Z.z42                        -0.0990
Z.z52                        -0.3793
```

```
Z.z33                          0.0185
Z.z43                         -0.1404
Z.z53                          0.1317
R.(Cryptomonas,Cryptomonas)    0.1638
R.(Diatoms,Diatoms)            0.2913
R.(Greens,Greens)              0.8621
R.(Unicells,Unicells)          0.3080
R.(Other.algae,Other.algae)    0.5000
x0.X1                          0.2218
x0.X2                          1.8155
x0.X3                         -4.8097
Initial states (x0) defined at t=0

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

10.7 Interpreting the MARSS output

By now the `MARSS()` output should look familiar. The first 12 parameter estimates `Z.z##` are the loadings of each observed time series on the 3 hidden states. The next 5 estimates `R.(,)` are the variances of the observation errors ($v_{i,t}$). The last 3 values, `x0.X#`, are the estimates of the initial states at $t = 0$.

Recall that the estimates of the processes themselves (i.e., $\mathbf{x}$) are contained in one of the list elements in our fitted `MARSS` object. Specifically, they are in `mod_fit$states`, and their respective standard errors are in `mod_fit$states.se`. For the names of all of the other objects, type `names(dfa_1)`.

10.8 Rotating trends and loadings

Before proceeding further, we need to address the constraints we placed on the DFA model in Sec 2.2. In particular, we arbitrarily constrained $\mathbf{Z}$ in such a way to choose only one of these solutions, but fortunately the different solutions are equivalent, and they can be related to each other by a rotation matrix $\mathbf{H}$. Let $\mathbf{H}$ be any $m \times m$ non-singular matrix. The following are then equivalent DFA models:

$$\mathbf{y}_t = \mathbf{Z}\mathbf{x}_t + \mathbf{a} + \mathbf{v}_t \mathbf{x}_t = \mathbf{x}_{t-1} + \mathbf{w}_t \tag{10.10}$$

and

$$\mathbf{y}_t = \mathbf{Z}\mathbf{H}^{-1}\mathbf{x}_t + \mathbf{a} + \mathbf{v}_t \mathbf{H}\mathbf{x}_t = \mathbf{H}\mathbf{x}_{t-1} + \mathbf{H}\mathbf{w}_t. \tag{10.11}$$

There are many ways of doing factor rotations, but a common method is the "varimax"" rotation, which seeks a rotation matrix $\mathbf{H}$ that creates the largest difference between the loadings in $\mathbf{Z}$. For example, imagine that row 3 in our estimated $\mathbf{Z}$ matrix was (0.2, 0.2, 0.2). That would mean that green algae were a mixture of equal parts of processes 1, 2, and 3. If instead row 3 was (0.8, 0.1, 0.05), this would make our interpretation of the model fits easier because we could say that green algae followed the first process most closely. The varimax rotation would find the $\mathbf{H}$ matrix that makes the rows in $\mathbf{Z}$ more like (0.8, 0.1, 0.05) and less like (0.2, 0.2, 0.2).

The varimax rotation is easy to compute because R has a built in function for this: `varimax()`. Interestingly, the function returns the inverse of $\mathbf{H}$, which we need anyway.

```r
## get the estimated ZZ
Z_est <- coef(dfa_1, type = "matrix")$Z
## get the inverse of the rotation matrix
H_inv <- varimax(Z_est)$rotmat
```

We can now rotate both $\mathbf{Z}$ and $\mathbf{x}$.

```r
## rotate factor loadings
Z_rot = Z_est %*% H_inv
## rotate processes
proc_rot = solve(H_inv) %*% dfa_1$states
```

10.9 Estimated states and loadings

Here are plots of the three hidden processes (left column) and the loadings for each of phytoplankton groups (right column).

```r
ylbl <- phytoplankton
w_ts <- seq(dim(dat)[2])
layout(matrix(c(1, 2, 3, 4, 5, 6), mm, 2), widths = c(2, 1))
## par(mfcol=c(mm,2), mai = c(0.5,0.5,0.5,0.1), omi =
## c(0,0,0,0))
par(mai = c(0.5, 0.5, 0.5, 0.1), omi = c(0, 0, 0, 0))
## plot the processes
for (i in 1:mm) {
    ylm <- c(-1, 1) * max(abs(proc_rot[i, ]))
    ## set up plot area
    plot(w_ts, proc_rot[i, ], type = "n", bty = "L", ylim = ylm,
        xlab = "", ylab = "", xaxt = "n")
    ## draw zero-line
    abline(h = 0, col = "gray")
    ## plot trend line
    lines(w_ts, proc_rot[i, ], lwd = 2)
    lines(w_ts, proc_rot[i, ], lwd = 2)
    ## add panel labels
    mtext(paste("State", i), side = 3, line = 0.5)
    axis(1, 12 * (0:dim(dat_1980)[2]) + 1, yr_frst + 0:dim(dat_1980)[2])
}
## plot the loadings
minZ <- 0
ylm <- c(-1, 1) * max(abs(Z_rot))
for (i in 1:mm) {
    plot(c(1:N_ts)[abs(Z_rot[, i]) > minZ], as.vector(Z_rot[abs(Z_rot[,
        i]) > minZ, i]), type = "h", lwd = 2, xlab = "", ylab = "",
        xaxt = "n", ylim = ylm, xlim = c(0.5, N_ts + 0.5), col = clr)
    for (j in 1:N_ts) {
        if (Z_rot[j, i] > minZ) {
            text(j, -0.03, ylbl[j], srt = 90, adj = 1, cex = 1.2,
                col = clr[j])
    }
    if (Z_rot[j, i] < -minZ) {
      text(j, 0.03, ylbl[j], srt = 90, adj = 0, cex = 1.2,
        col = clr[j])
    }
    abline(h = 0, lwd = 1.5, col = "gray")
  }
  mtext(paste("Factor loadings on state", i), side = 3, line = 0.5)
}
```

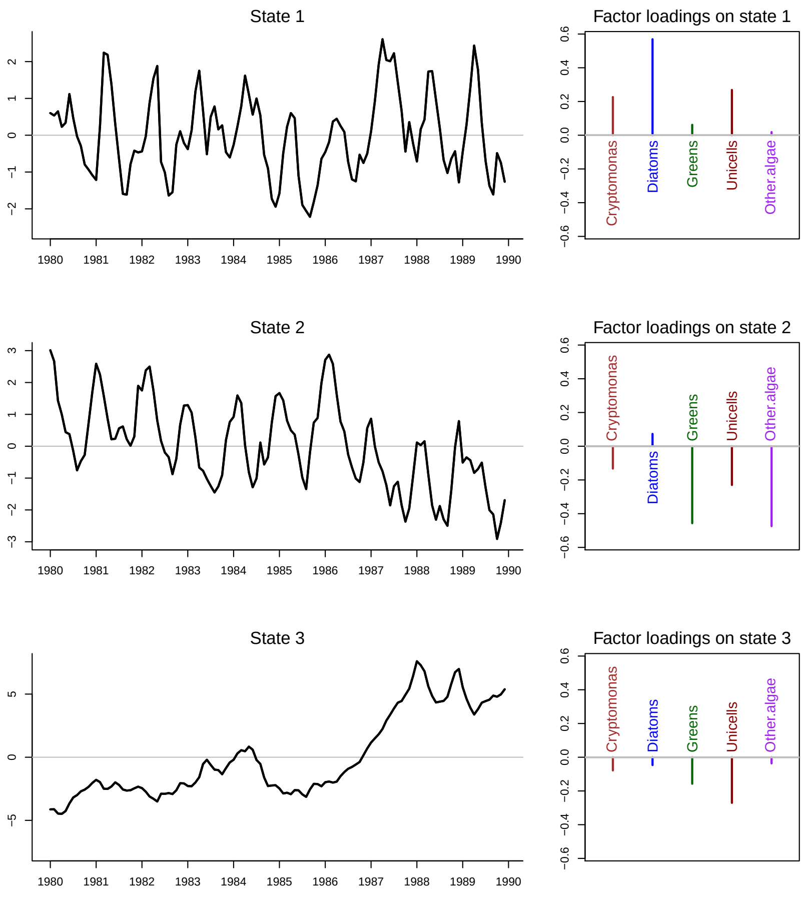


Figure 10.2: Estimated states from the DFA model.

It looks like there are strong seasonal cycles in the data, but there is some indication of a phase difference between some of the groups. We can use `ccf()` to investigate further.

```r
par(mai = c(0.9, 0.9, 0.1, 0.1))
ccf(proc_rot[1, ], proc_rot[2, ], lag.max = 12, main = "")
```

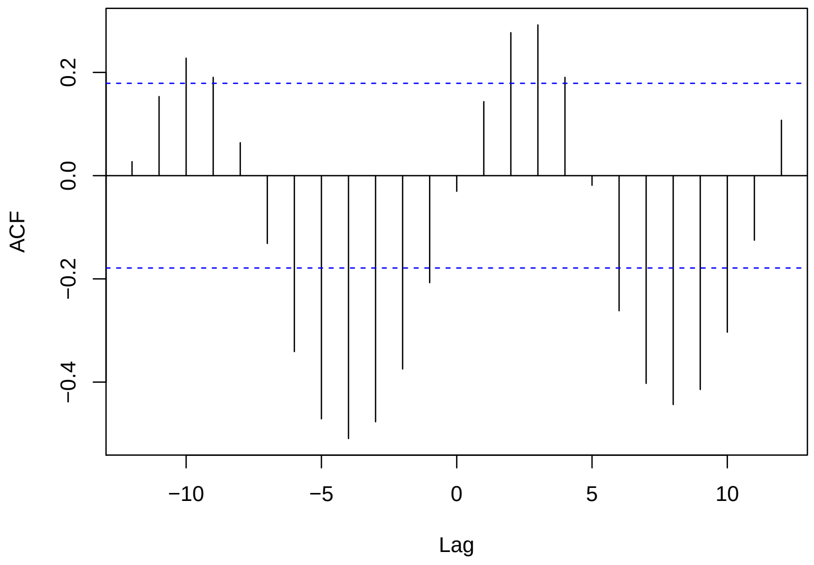


Figure 10.3: Cross-correlation plot of the two rotations.

10.10 Plotting the data and model fits

We can plot the fits for our DFA model along with the data. The following function will return the fitted values $\pm$ (1-α)% confidence intervals.

```r
get_DFA_fits <- function(MLEobj, dd = NULL, alpha = 0.05) {
    ## empty list for results
    fits <- list()
    ## extra stuff for var() calcs
    Ey <- MARSS:::MARSShatyt(MLEobj)
    ## model params
    ZZ <- coef(MLEobj, type = "matrix")$Z
    ## number of obs ts
    nn <- dim(Ey$ytT)[1]
    ## number of time steps
    TT <- dim(Ey$ytT)[2]
    ## get the inverse of the rotation matrix
    H_inv <- varimax(ZZ)$rotmat
    ## check for covars
    if (!is.null(dd)) {
        DD <- coef(MLEobj, type = "matrix")$D
        ## model expectation
        fits$ex <- ZZ %*% H_inv %*% MLEobj$states + DD %*% dd
    } else {
        ## model expectation
        fits$ex <- ZZ %*% H_inv %*% MLEobj$states
    }
    ## Var in model fits
    VtT <- MARSSkfss(MLEobj)$VtT
    VV <- NULL
    for (tt in 1:TT) {
        RZVZ <- coef(MLEobj, type = "matrix")$R - ZZ %*% VtT[,
            , tt] %*% t(ZZ)
        SS <- Ey$yxtT[, , tt] - Ey$ytT[, tt, drop = FALSE] %*%
            t(MLEobj$states[, tt, drop = FALSE])
        VV <- cbind(VV, diag(RZVZ + SS %*% t(ZZ) + ZZ %*% t(SS)))
    }
    SE <- sqrt(VV)
    ## upper & lower (1-alpha)% CI
    fits$up <- qnorm(1 - alpha/2) * SE + fits$ex
    fits$lo <- qnorm(alpha/2) * SE + fits$ex
    return(fits)
}
```

Here are time series of the five phytoplankton groups (points) with the mean of the DFA fits (black line) and the 95% confidence intervals (gray lines).

```r
## get model fits & CI's
mod_fit <- get_DFA_fits(dfa_1)
## plot the fits
ylbl <- phytoplankton
par(mfrow = c(N_ts, 1), mai = c(0.5, 0.7, 0.1, 0.1), omi = c(0,
    0, 0, 0))
for (i in 1:N_ts) {
    up <- mod_fit$up[i, ]
    mn <- mod_fit$ex[i, ]
    lo <- mod_fit$lo[i, ]
    plot(w_ts, mn, xlab = "", ylab = ylbl[i], xaxt = "n", type = "n",
        cex.lab = 1.2, ylim = c(min(lo), max(up)))
    axis(1, 12 * (0:dim(dat_1980)[2]) + 1, yr_frst + 0:dim(dat_1980)[2])
    points(w_ts, dat[i, ], pch = 16, col = clr[i])
    lines(w_ts, up, col = "darkgray")
    lines(w_ts, mn, col = "black", lwd = 2)
    lines(w_ts, lo, col = "darkgray")
}
```

10.11 Covariates in DFA models

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

$$\begin{aligned} \mathbf{y}_t &= \mathbf{Z}\mathbf{x}_t + \mathbf{a} + \mathbf{D}\mathbf{d}_t + \mathbf{v}_t \text{ where } \mathbf{v}_t \sim \text{MVN}(0, \mathbf{R}) \\ \mathbf{x}_t &= \mathbf{x}_{t-1} + \mathbf{w}_t \text{ where } \mathbf{w}_t \sim \text{MVN}(0, \mathbf{Q}) \end{aligned} \tag{10.12}$$

where the $q \times 1$ vector $\mathbf{d}_t$ contains the covariate(s) at time t, and the $n \times q$ matrix $\mathbf{D}$ contains the effect(s) of the covariate(s) on the observations. Using `form = "dfa"` and `covariates=<covariate name(s)>`, we can easily add covariates to our DFA, but this means that the covariates are input, not data, and there can be no missing values (see Chapter 6 in the **MARSS** User Guide for how to include covariates with missing values).

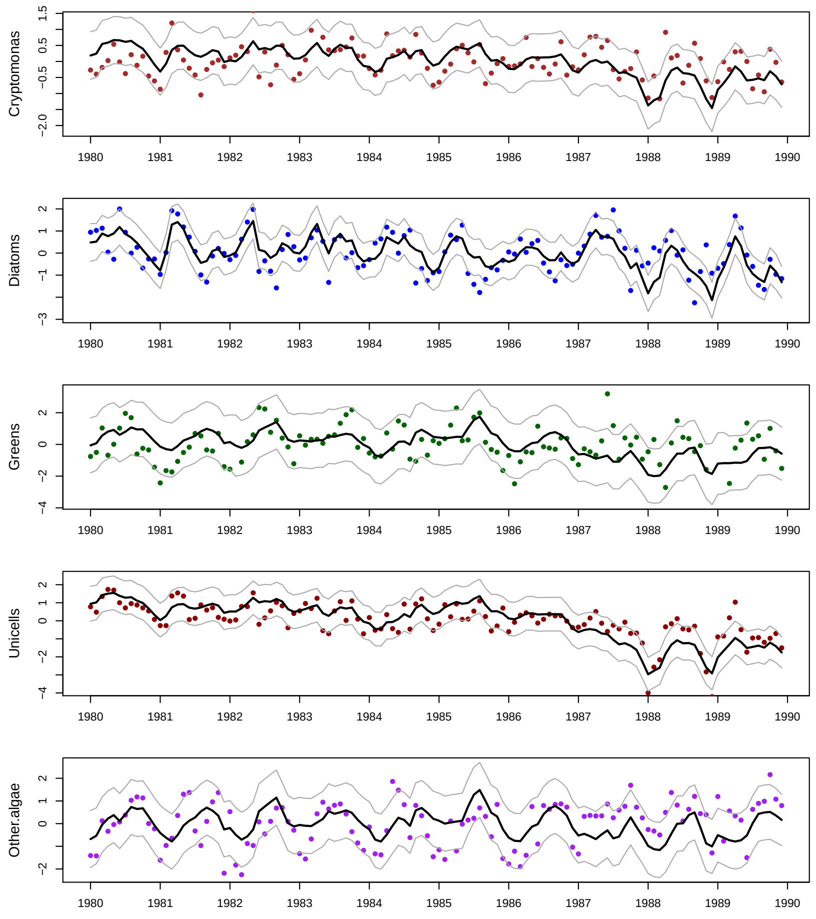


Figure 10.4: Data and fits from the DFA model.

10.12 Example from Lake Washington

The Lake Washington dataset has two environmental covariates that we might expect to have effects on phytoplankton growth, and hence, abundance: temperature (`Temp`) and total phosphorous (`TP`). We need the covariate inputs to have the same number of time steps as the variate data, and thus we limit the covariate data to the years 1980-1994 also.

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

We will now fit three different models that each add covariate effects (i.e., `Temp`, `TP`, `Temp` and `TP`) to our existing model above where $m = 3$ and $\mathbf{R}$ is `"diagonal and unequal"`.

```
mod_list = list(m = 3, R = "diagonal and unequal")
dfa_temp <- MARSS(dat, model = mod_list, form = "dfa", z.score = FALSE,
    control = con_list, covariates = temp)
dfa_TP <- MARSS(dat, model = mod_list, form = "dfa", z.score = FALSE,
    control = con_list, covariates = TP)
dfa_both <- MARSS(dat, model = mod_list, form = "dfa", z.score = FALSE,
    control = con_list, covariates = rbind(temp, TP))
```

Next we can compare whether the addition of the covariates improves the model fit.

```
print(cbind(model = c("no covars", "Temp", "TP", "Temp & TP"),
    AICc = round(c(dfa_1$AICc, dfa_temp$AICc, dfa_TP$AICc, dfa_both$AICc))),
    quote = FALSE)
```

```
     model     AICc
[1,] no covars 1427
[2,] Temp      1356
[3,] TP        1414
[4,] Temp & TP 1362
```

This suggests that adding temperature or phosphorus to the model, either alone or in combination with one another, does seem to improve overall model fit. If we were truly interested in assessing the “best” model structure that includes covariates, however, we should examine all combinations of 1-4 trends and different structures for **R**.

Now let’s try to fit a model with a dummy variable for season, and see how that does.

```
cos_t <- cos(2 * pi * seq(TT)/12)
sin_t <- sin(2 * pi * seq(TT)/12)
dd <- rbind(cos_t, sin_t)
dfa_seas <- MARSS(dat, model = mod_list, form = "dfa", z.score = FALSE,
    control = con_list, covariates = dd)
```

```
Success! abstol and log-log tests passed at 451 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 451 iterations.
Log-likelihood: -633.1283
AIC: 1320.257   AICc: 1322.919

                            Estimate
Z.11                         0.26207
Z.21                         0.24762
Z.31                         0.03689
Z.41                         0.51329
Z.51                         0.18479
Z.22                         0.04819
Z.32                        -0.08824
Z.42                         0.06454
Z.52                         0.05905
Z.33                         0.02673
Z.43                         0.19343
```

```
Z.53                          -0.10528
R.(Cryptomonas,Cryptomonas)    0.14406
R.(Diatoms,Diatoms)            0.44205
R.(Greens,Greens)              0.73113
R.(Unicells,Unicells)          0.19533
R.(Other.algae,Other.algae)    0.50127
D.(Cryptomonas,cos_t)         -0.23244
D.(Diatoms,cos_t)             -0.40829
D.(Greens,cos_t)              -0.72656
D.(Unicells,cos_t)            -0.34666
D.(Other.algae,cos_t)         -0.41606
D.(Cryptomonas,sin_t)          0.12515
D.(Diatoms,sin_t)              0.65621
D.(Greens,sin_t)              -0.50657
D.(Unicells,sin_t)            -0.00867
D.(Other.algae,sin_t)         -0.62474
Initial states (x0) defined at t=0

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

```
dfa_seas$AICc
```

```
[1] 1322.919
```

The model with a dummy seasonal factor does much better than the covariate models. The model fits for the seasonal effects model are shown below.

```
## get model fits & CI's
mod_fit <- get_DFA_fits(dfa_seas, dd = dd)
## plot the fits
ylbl <- phytoplankton
par(mfrow = c(N_ts, 1), mai = c(0.5, 0.7, 0.1, 0.1), omi = c(0,
    0, 0, 0))
for (i in 1:N_ts) {
    up <- mod_fit$up[i, ]
    mn <- mod_fit$ex[i, ]
    lo <- mod_fit$lo[i, ]
    plot(w_ts, mn, xlab = "", ylab = ylbl[i], xaxt = "n", type = "n",
        cex.lab = 1.2, ylim = c(min(lo), max(up)))
    axis(1, 12 * (0:dim(dat_1980)[2]) + 1, yr_frst + 0:dim(dat_1980)[2])
    points(w_ts, dat[i, ], pch = 16, col = clr[i])
    lines(w_ts, up, col = "darkgray")
    lines(w_ts, mn, col = "black", lwd = 2)
    lines(w_ts, lo, col = "darkgray")
}
```

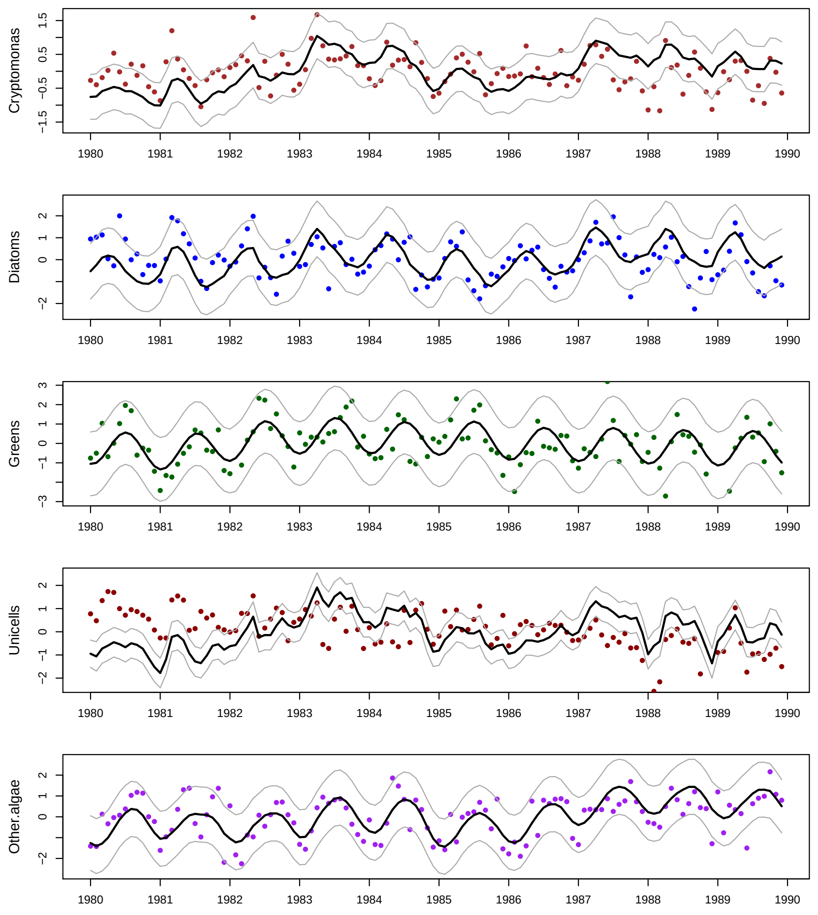


Figure 10.5: Data and model fits for the DFA with covariates.

10.13 Problems

For questions 1-3, use the Lake Washington plankton data from the chapter. `dat` is the data to use.

```r
library(MARSS)
data(lakeWAplankton, package = "MARSS")
all_dat <- lakeWAplanktonTrans
yr_frst <- 1980
yr_last <- 1989
plank_dat <- all_dat[all_dat[, "Year"] >= yr_frst & all_dat[,
    "Year"] <= yr_last, ]
phytoplankton <- c("Cryptomonas", "Diatoms", "Greens", "Unicells",
    "Other.algae")
dat_1980 <- plank_dat[, phytoplankton]
## transpose data so time goes across columns
dat_1980 <- t(dat_1980)
## remove the mean
dat <- zscore(dat_1980, mean.only = TRUE)
```

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

   Plot the model states (latent trends) and loadings as in Section 10.9. Describe the general patterns in the states and the ways the different taxa load onto those trends.

   Also plot the the model fits as in Section 10.10. Do they look reasonable? Are there any particular problems or outliers?

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

   Plot the model states (latent trends) and loadings as in Section 10.9. Describe the general patterns in the states and the ways the different taxa load onto those trends.

Also plot the the model fits as in Section 10.10. Do they look reasonable? Are there any particular problems or outliers?

3. Fit a DFA model that includes temperature as a covariate and 3 trends (as in Section 10.12), but with`R="unconstrained"`? How does this model compare to the model with `R="diagonal and unequal"`? How does it compare to the model in Question 2?

   Plot the model states and loadings as in Section 10.9. Describe the general patterns in the states and the ways the different taxa load onto those trends.

   Also plot the the model fits as in Section 10.10. Do they look reasonable? Are there any particular problems or outliers?

Chapter 11

Covariates with Missing Values

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

Data and packages

This chapter will use a SNOTEL dataset. These are data on snow water equivalency at locations throughtout the state of Washington. The data are in the **atsalibrary** package.

```
data(snotel, package = "atsalibrary")
```

The main packages used in this chapter are **MARSS** and **forecast**.

```
library(MARSS)
library(forecast)
library(ggplot2)
library(ggmap)
library(broom)
```

11.1 Covariates with missing values or observation error

The specific formulation of Equation (8.1) creates restrictions on the assumptions regarding the covariate data. You have to assume that your covariate data has no error, which is probably not true. You cannot have missing values in your covariate data, again unlikely. You cannot combine instrument time series; for example, if you have two temperature recorders with different error rates and biases. Also, what if you have one noisy temperature sensor in the first part of your time series and then you switch to a much better sensor in the second half of your time series? All these problems require pre-analysis massaging of the covariate data, leaving out noisy and gappy covariate data, and making what can feel like arbitrary choices about which covariate time series to include.

To circumvent these potential problems and allow more flexibility in how we incorporate covariate data, one can instead treat the covariates as components of an auto-regressive process by including them in both the process and observation models. Beginning with the process equation, we can write

$$\begin{gathered}\begin{bmatrix}\mathbf{x}^{(v)}\\ \mathbf{x}^{(c)}\end{bmatrix}_t = \begin{bmatrix}\mathbf{B}^{(v)} & \mathbf{C}\\ 0 & \mathbf{B}^{(c)}\end{bmatrix}\begin{bmatrix}\mathbf{x}^{(v)}\\ \mathbf{x}^{(c)}\end{bmatrix}_{t-1} + \begin{bmatrix}\mathbf{u}^{(v)}\\ \mathbf{u}^{(c)}\end{bmatrix} + \mathbf{w}_t,\\ \mathbf{w}_t \sim \text{MVN}\left(0, \begin{bmatrix}\mathbf{Q}^{(v)} & 0\\ 0 & \mathbf{Q}^{(c)}\end{bmatrix}\right)\end{gathered} \tag{11.1}$$

The elements with superscript (v) are for the k variate states and those with superscript (c) are for the q covariate states. The dimension of $\mathbf{x}^{(c)}$ is $q \times 1$ and q is not necessarily equal to p, the number of covariate observation time series in your dataset. Imagine, for example, that you have two temperature sensors and you are combining these data. Then you have two covariate observation time series ($p = 2$) but only one underlying covariate state time series ($q = 1$). The matrix $\mathbf{C}$ is dimension $k \times q$, and $\mathbf{B}^{(c)}$ and $\mathbf{Q}^{(c)}$ are dimension $q \times q$. The dimension of $\mathbf{x}^{(v)}$ is $k \times 1$, and $\mathbf{B}^{(v)}$ and $\mathbf{Q}^{(v)}$ are dimension $k \times k$. The dimension of $\mathbf{x}$ is always denoted m. If your process model includes only variates, then $k = m$, but now your process model includes k variates and q covariate states so $m = k + q$.

Next, we can write the observation equation in an analogous manner, such that

$$\begin{gathered}\begin{bmatrix}\mathbf{y}^{(v)}\\ \mathbf{y}^{(c)}\end{bmatrix}_t = \begin{bmatrix}\mathbf{Z}^{(v)} & \mathbf{D}\\ 0 & \mathbf{Z}^{(c)}\end{bmatrix}\begin{bmatrix}\mathbf{x}^{(v)}\\ \mathbf{x}^{(c)}\end{bmatrix}_t + \begin{bmatrix}\mathbf{a}^{(v)}\\ \mathbf{a}^{(c)}\end{bmatrix} + \mathbf{v}_t,\\ \mathbf{v}_t \sim \text{MVN}\left(0, \begin{bmatrix}\mathbf{R}^{(v)} & 0\\ 0 & \mathbf{R}^{(c)}\end{bmatrix}\right)\end{gathered} \tag{11.2}$$

The dimension of $\mathbf{y}^{(c)}$ is $p \times 1$, where p is the number of covariate observation time series in your dataset. The dimension of $\mathbf{y}^{(v)}$ is $l \times 1$, where l is the number of variate observation time series in your dataset. The total dimension of $\mathbf{y}$ is $l + p$. The matrix $\mathbf{D}$ is dimension $l \times q$, $\mathbf{Z}^{(c)}$ is dimension $p \times q$, and $\mathbf{R}^{(c)}$ are dimension $p \times p$. The dimension of $\mathbf{Z}^{(v)}$ is dimension $l \times k$, and $\mathbf{R}^{(v)}$ are dimension $l \times l$.

The $\mathbf{D}$ matrix would presumably have a number of all zero rows in it, as would the $\mathbf{C}$ matrix. The covariates that affect the states would often be different than the covariates that affect the observations. For example, mean annual temperature might affect population growth rates for many species while having little or no affect on observability, and turbidity might strongly affect observability in many types of aquatic surveys but have little affect on population growth rate.

Our MARSS model with covariates now looks on the surface like a regular MARSS model:

$$\begin{gathered}\mathbf{x}_t = \mathbf{B}\mathbf{x}_{t-1} + \mathbf{u} + \mathbf{w}_t, \text{ where } \mathbf{w}_t \sim \text{MVN}(0, \mathbf{Q})\\ \mathbf{y}_t = \mathbf{Z}\mathbf{x}_t + \mathbf{a} + \mathbf{v}_t, \text{ where } \mathbf{v}_t \sim \text{MVN}(0, \mathbf{R})\end{gathered} \tag{11.3}$$

with the $\mathbf{x}_t$, $\mathbf{y}_t$ and parameter matrices redefined as in Equations (11.1) and (11.2):

$$\begin{gathered}\mathbf{x} = \begin{bmatrix}\mathbf{x}^{(v)}\\ \mathbf{x}^{(c)}\end{bmatrix} \quad \mathbf{B} = \begin{bmatrix}\mathbf{B}^{(v)} & \mathbf{C}\\ 0 & \mathbf{B}^{(c)}\end{bmatrix} \quad \mathbf{u} = \begin{bmatrix}\mathbf{u}^{(v)}\\ \mathbf{u}^{(c)}\end{bmatrix} \quad \mathbf{Q} = \begin{bmatrix}\mathbf{Q}^{(v)} & 0\\ 0 & \mathbf{Q}^{(c)}\end{bmatrix}\\ \mathbf{y} = \begin{bmatrix}\mathbf{y}^{(v)}\\ \mathbf{y}^{(c)}\end{bmatrix} \quad \mathbf{Z} = \begin{bmatrix}\mathbf{Z}^{(v)} & \mathbf{D}\\ 0 & \mathbf{Z}^{(c)}\end{bmatrix} \quad \mathbf{a} = \begin{bmatrix}\mathbf{a}^{(v)}\\ \mathbf{a}^{(c)}\end{bmatrix} \quad \mathbf{R} = \begin{bmatrix}\mathbf{R}^{(v)} & 0\\ 0 & \mathbf{R}^{(c)}\end{bmatrix}\end{gathered} \tag{11.4}$$

Note $\mathbf{Q}$ and $\mathbf{R}$ are written as block diagonal matrices, but you could allow covariances if that made sense. $\mathbf{u}$ and $\mathbf{a}$ are column vectors here. We can fit the model (Equation (11.4)) as usual using the `MARSS()` function.

The log-likelihood that is returned by MARSS will include the log-likelihood of the covariates under the covariate state model. If you want only the the log-likelihood of the non-covariate data, you will need to subtract off the log-likelihood of the covariate model:

$$\begin{aligned}\mathbf{x}_t^{(c)} &= \mathbf{B}^{(c)}\mathbf{x}_{t-1}^{(c)} + \mathbf{u}^{(c)} + \mathbf{w}_t, \text{ where } \mathbf{w}_t \sim \text{MVN}(0, \mathbf{Q}^{(c)}) \\ \mathbf{y}_t^{(c)} &= \mathbf{Z}^{(c)}\mathbf{x}_t^{(c)} + \mathbf{a}^{(c)} + \mathbf{v}_t, \text{ where } \mathbf{v}_t \sim \text{MVN}(0, \mathbf{R}^{(c)})\end{aligned} \tag{11.5}$$

An easy way to get this log-likelihood for the covariate data only is use the augmented model (Equation (11.2) with terms defined as in Equation (11.4) but pass in missing values for the non-covariate data. The following code shows how to do this.

```
y.aug = rbind(data, covariates)
fit.aug = MARSS(y.aug, model = model.aug)
```

`fit.aug` is the MLE object that can be passed to `MARSSkf()`. You need to make a version of this MLE object with the non-covariate data filled with NAs so that you can compute the log-likelihood without the covariates. This needs to be done in the `marss` element since that is what is used by `MARSSkf()`. Below is code to do this.

```
fit.cov = fit.aug
fit.cov$marss$data[1:dim(data)[1], ] = NA
extra.LL = MARSSkf(fit.cov)$logLik
```

Note that when you fit the augmented model, the estimates of $\mathbf{C}$ and $\mathbf{B}^{(c)}$ are affected by the non-covariate data since the model for both the non-covariate and covariate data are estimated simultaneously and are not independent (since the covariate states affect the non-covariates states). If you want the covariate model to be unaffected by the non-covariate data, you can fit the covariate model separately and use the estimates for $\mathbf{B}^{(c)}$ and $\mathbf{Q}^{(c)}$ as fixed values in your augmented model.

11.2 Example: Snotel Data

Let's see an example using the Washington SNOTEL data. The data we will use is the snow water equivalent percent of normal. This represents the snow water equivalent compared to the average value for that site on the same day. We will look at a subset of sites in the Central Cascades in our `snotel` dataset (Figure 11.1).

```r
y <- snotelmeta
# Just use a subset
y = y[which(y$Longitude < -121.4), ]
y = y[which(y$Longitude > -122.5), ]
y = y[which(y$Latitude < 47.5), ]
y = y[which(y$Latitude > 46.5), ]
```

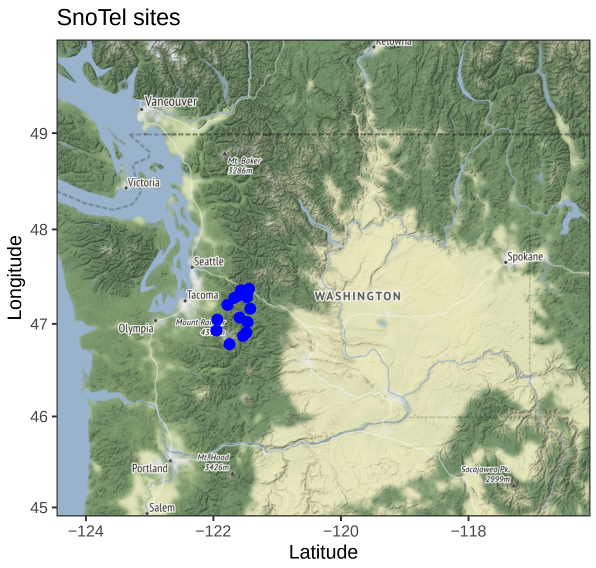


Figure 11.1: Subset of SNOTEL sties used in this chapter.

For the first analysis, we are just going to look at February Snow Water Equivalent (SWE). Our subset of stations is `y$Station.Id`. There are many missing years among some of our stations (Figure 11.2).

```r
swe.feb <- snotel
swe.feb <- swe.feb[swe.feb$Station.Id %in% y$Station.Id & swe.feb$Month ==
    "Feb", ]
p <- ggplot(swe.feb, aes(x = Date, y = SWE)) + geom_line()
p + facet_wrap(~Station)
```

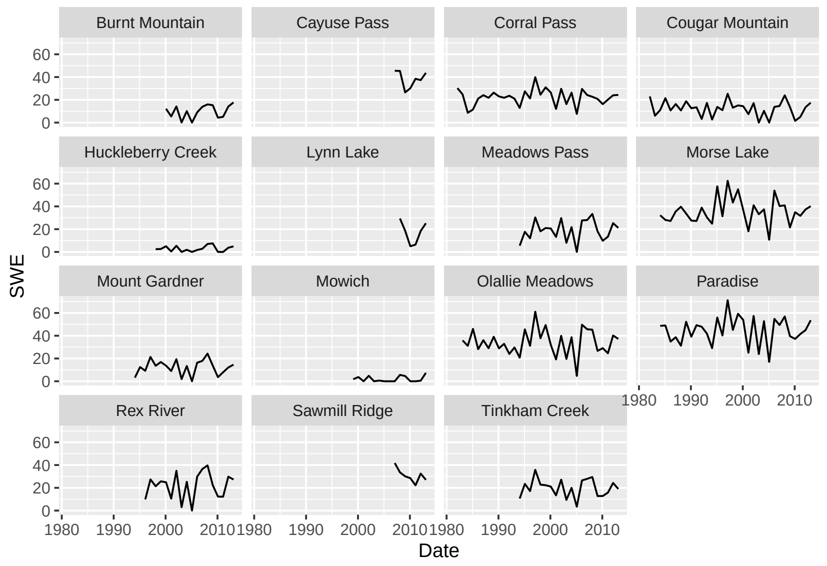


Figure 11.2: Snow water equivalent time series from each SNOTEL station.

11.2.1 Estimate missing Feb SWE using AR(1) with spatial correlation

Imagine that for our study we need an estimate of SWE for all sites. We will use the information from the sites with full data to estimate the missing SWE for other sites. We will use a MARSS model to use all the available data.

$$\begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_{15} \end{bmatrix}_t = \begin{bmatrix} b & 0 & \dots & 0 \\ 0 & b & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & b \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_{15} \end{bmatrix}_{t-1} + \begin{bmatrix} w_1 \\ w_2 \\ \dots \\ w_{15} \end{bmatrix}_t$$
$$\begin{bmatrix} y_1 \\ y_2 \\ \dots \\ y_{15} \end{bmatrix}_t = \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_{15} \end{bmatrix}_t + \begin{bmatrix} a_1 \\ a_2 \\ \dots \\ a_{15} \end{bmatrix}_t + \begin{bmatrix} v_1 \\ v_2 \\ \dots \\ v_{15} \end{bmatrix}_t \tag{11.6}$$

We will use an unconstrained variance-covariance structure for **w** and assume that **v** is identical and independent and very low (SNOTEL instrument variability). The a_i determine the level of the x_i.

We need our data to be in rows. We will use `reshape2::acast()`.

```
dat.feb <- reshape2::acast(swe.feb, Station ~ Year, value.var = "SWE")
```

We set up the model for MARSS so that it is the same as (11.6). We will fix the measurement error to be small; we could use 0 but the fitting is more stable if we use a small variance instead. When estimating $\mathbf{B}$, setting the initial value to be at $t = 1$ instead of $t = 0$ works better.

```
ns <- length(unique(swe.feb$Station))
B <- "diagonal and equal"
Q <- "unconstrained"
R <- diag(0.01, ns)
U <- "zero"
A <- "unequal"
x0 <- "unequal"
mod.list.ar1 = list(B = B, Q = Q, R = R, U = U, x0 = x0, A = A,
    tinitx = 1)
```

Now we can fit a MARSS model and get estimates of the missing SWEs. Convergence is slow. We set $\mathbf{a}$ equal to the mean of the time series to speed convergence.

```
library(MARSS)
m <- apply(dat.feb, 1, mean, na.rm = TRUE)
fit.ar1 <- MARSS(dat.feb, model = mod.list.ar1, control = list(maxit = 5000),
    inits = list(A = matrix(m, ns, 1)))
```

The b estimate is "".

Let's plot the estimated SWEs for the missing years (Figure 11.3). These estimates use all the information about the correlation with other sites and uses information about correlation with the prior and subsequent years. We will use the `tidy()` function to get the estimates and the 95% prediction intervals. The prediction interval is for the range of SWE values we might observe for that site. Notice that for some sites, intervals are low in early years as these sites are highly correlated with site for which there are data. In other sites, the uncertainty is high in early years because the sites with data in those years are not highly correlated. There are no intervals for sites with data. We have data for those sites, so we are not uncertain about the observed SWE for those.

```
fit <- fit.ar1
d <- fitted(fit, interval = "prediction", type = "ytT")
d$Year <- d$t + 1980
d$Station <- d$.rownames
p <- ggplot(data = d) + geom_line(aes(Year, .fitted)) + geom_point(aes(Year,
    y)) + geom_ribbon(aes(x = Year, ymin = .lwr, ymax = .upr),
    linetype = 2, alpha = 0.2, fill = "blue") + facet_wrap(~Station) +
    xlab("") + ylab("SWE (demeaned)")
p
```

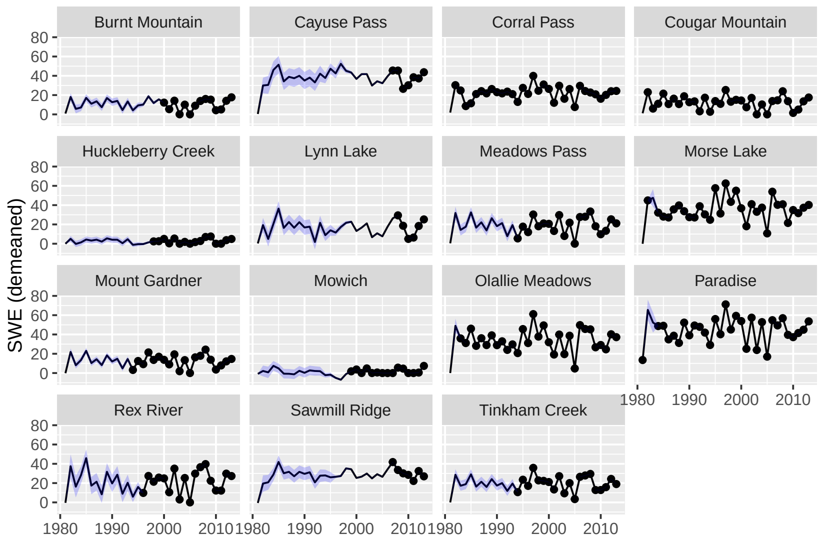


Figure 11.3: Estimated SWEs for the missing sites with prediction intervals.

If we were using these SWE as covariates in a site specific model, we could then use the estimates as our covariates, however this would not incorporate parameter uncertainty. Alternatively we could use Equation (11.1) and set the parameters for the covariate process to those estimated for our covariate-only model. This approach will incorporate the uncertainty in the SWE estimates in the early years for the sites with no data.

Note, we should do some cross-validation (fitting with data left out) to ensure that the estimated SWEs are well-matched to actual measurements. It would probably be best to do 'leave-three' out instead of 'leave-one' out since the estimates for time t uses information from $t-1$ and $t+1$ (if present).

11.2.1.1 Diagnostics

The model residuals have a tendency for negative autocorrelation at lag-1 (Figure 11.4).

```
fit <- fit.ar1
par(mfrow = c(4, 4), mar = c(2, 2, 1, 1))
apply(MARSSresiduals(fit, type = "tt1")$model.residuals[, 1:30],
    1, acf, na.action = na.pass)
```

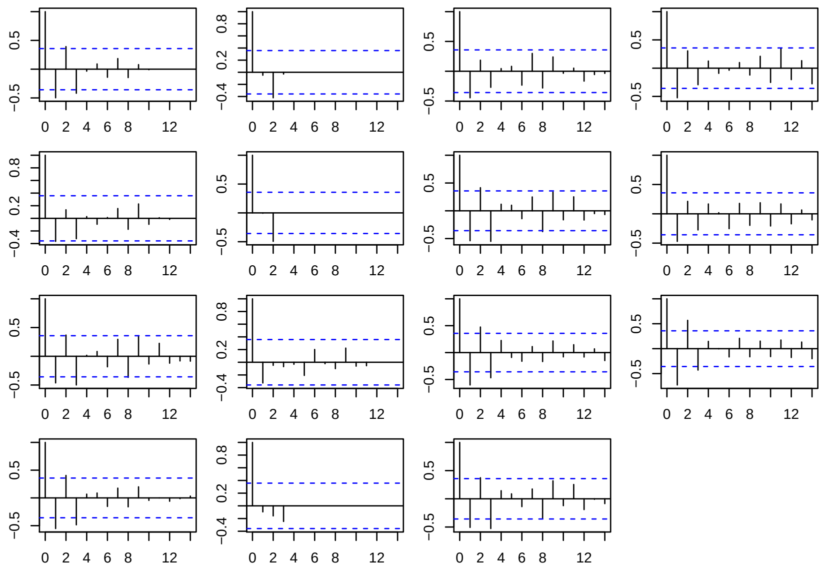


Figure 11.4: Model residuals for the AR(1) model.

11.2.2 Estimate missing Feb SWE using only correlation

Another approach is to treat the February data as temporally uncorrelated. The two longest time series (Paradise and Olallie Meadows) show minimal autocorrelation so we might decide to just use the correlation across stations for our estimates. In this case, the state of the missing SWE values at time t is the expected value conditioned on all the stations with data at time t given the estimated variance-covariance matrix $\mathbf{Q}$.

We could set this model up as

$$\begin{bmatrix} y_1 \\ y_2 \\ \dots \\ y_{15} \end{bmatrix}_t = \begin{bmatrix} a_1 \\ a_2 \\ \dots \\ a_{15} \end{bmatrix}_t + \begin{bmatrix} v_1 \\ v_2 \\ \dots \\ v_{15} \end{bmatrix}_t, \begin{bmatrix} \sigma_1 & \zeta_{1,2} & \dots & \zeta_{1,15} \\ \zeta_{2,1} & \sigma_2 & \dots & \zeta_{2,15} \\ \dots & \dots & \dots & \dots \\ \zeta_{15,1} & \zeta_{15,2} & \dots & \sigma_{15} \end{bmatrix} \tag{11.7}$$

However the EM algorithm used by `MARSS()` runs into numerical issues. Instead we will set the model up as follows. Allowing a hidden state observed with small error makes the estimation more stable.

$$\begin{gathered} \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_{15} \end{bmatrix}_t = \begin{bmatrix} w_1 \\ w_2 \\ \dots \\ w_{15} \end{bmatrix}_t, \begin{bmatrix} w_1 \\ w_2 \\ \dots \\ w_{15} \end{bmatrix}_t \sim \begin{bmatrix} \sigma_1 & \zeta_{1,2} & \dots & \zeta_{1,15} \\ \zeta_{2,1} & \sigma_2 & \dots & \zeta_{2,15} \\ \dots & \dots & \dots & \dots \\ \zeta_{15,1} & \zeta_{15,2} & \dots & \sigma_{15} \end{bmatrix} \\ \begin{bmatrix} y_1 \\ y_2 \\ \dots \\ y_{15} \end{bmatrix}_t = \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_{15} \end{bmatrix}_t + \begin{bmatrix} a_1 \\ a_2 \\ \dots \\ a_{15} \end{bmatrix}_t + \begin{bmatrix} v_1 \\ v_2 \\ \dots \\ v_{15} \end{bmatrix}_t, \begin{bmatrix} 0.01 & 0 & \dots & 0 \\ 0 & 0.01 & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & 0.01 \end{bmatrix} \end{gathered} \tag{11.8}$$

Again $\mathbf{a}$ is the mean level in the time series. Note that the expected value of $\mathbf{x}$ is zero if there are no data, so $E(\mathbf{x}_0) = 0$.

```r
ns <- length(unique(swe.feb$Station))
B <- "zero"
Q <- "unconstrained"
R <- diag(0.01, ns)
U <- "zero"
```

```r
A <- "unequal"
x0 <- "zero"
mod.list.corr = list(B = B, Q = Q, R = R, U = U, x0 = x0, A = A,
    tinitx = 0)
```

Now we can fit a MARSS model and get estimates of the missing SWEs. Convergence is slow. We set **a** equal to the mean of the time series to speed convergence.

```r
m <- apply(dat.feb, 1, mean, na.rm = TRUE)
fit.corr <- MARSS(dat.feb, model = mod.list.corr, control = list(maxit = 5000),
    inits = list(A = matrix(m, ns, 1)))
```

The estimated SWEs for the missing years uses the information about the correlation with other sites only.

```r
fit <- fit.corr
d <- fitted(fit, type = "ytT", interval = "prediction")
d$Year <- d$t + 1980
d$Station <- d$.rownames
p <- ggplot(data = d) + geom_line(aes(Year, .fitted)) + geom_point(aes(Year,
    y)) + geom_ribbon(aes(x = Year, ymin = .lwr, ymax = .upr),
    linetype = 2, alpha = 0.2, fill = "blue") + facet_wrap(~Station) +
    xlab("") + ylab("SWE (demeaned)")
p
```

11.2.2.1 Diagnostics

The model residuals have no tendency towards negative autocorrelation now that we removed the autoregressive component from the process (x) model.

```r
fit <- fit.corr
par(mfrow = c(4, 4), mar = c(2, 2, 1, 1))
apply(MARSSresiduals(fit, type = "tt1")$model.residuals, 1, acf,
    na.action = na.pass)
mtext("Model Residuals ACF", outer = TRUE, side = 3)
```

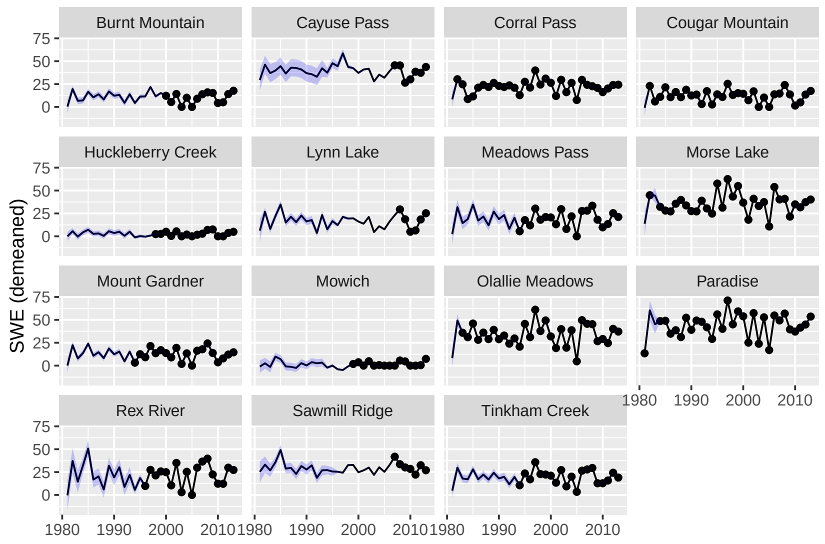
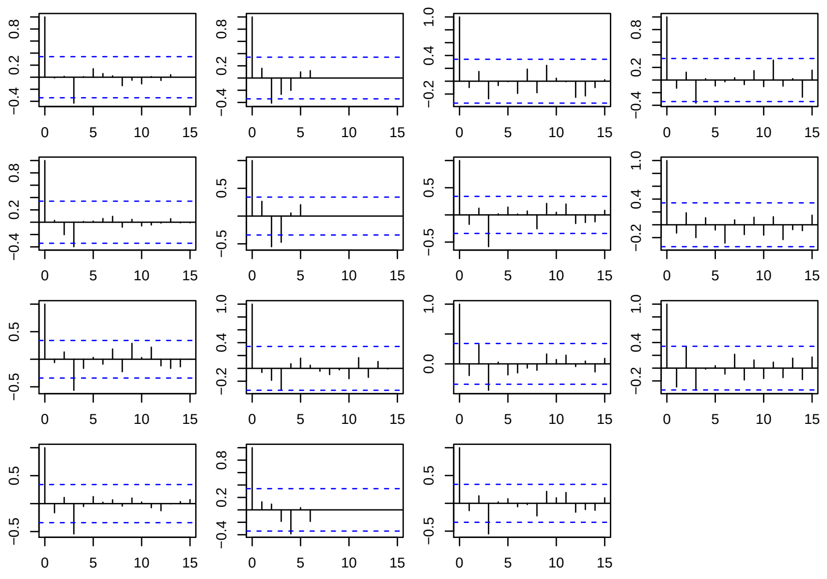


Figure 11.5: Estimated SWEs from the expected value of the states $\hat{x}$ conditioned on all the data for the model with only correlation across stations at time t.



11.2.3 Estimate missing Feb SWE using DFA

Another approach we might take is to model SWE using Dynamic Factor Analysis. Our model might take the following form with two factors, modeled as AR(1) processes. **a** is the mean level of the time series.

$$
\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}_t = \begin{bmatrix} b_1 & 0 \\ 0 & b_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}_{t-1} + \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}_t
$$

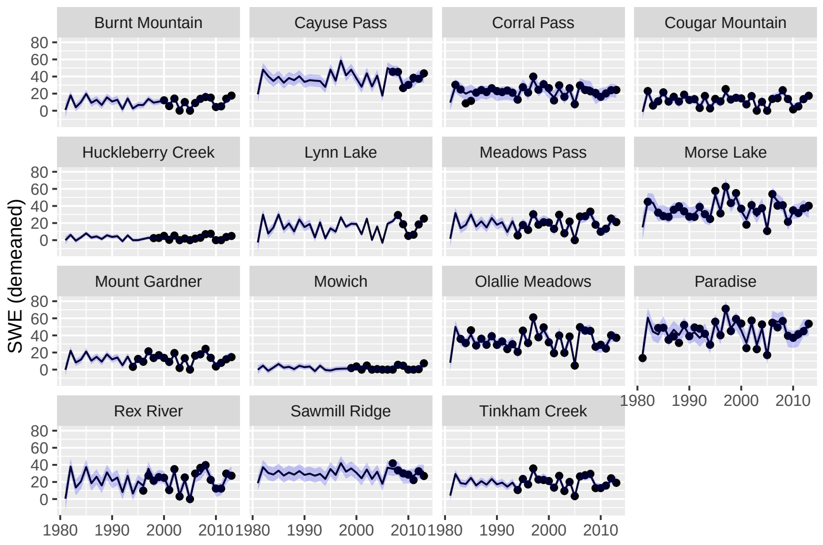
$$
\begin{bmatrix} y_1 \\ y_2 \\ \dots \\ y_{15} \end{bmatrix}_t = \begin{bmatrix} z_{1,1} & 0 \\ z_{2,1} & z_{2,2} \\ \dots & \\ z_{3,1} & z_{3,2} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}_t + \begin{bmatrix} a_1 \\ a_2 \\ \dots \\ a_{15} \end{bmatrix} + \begin{bmatrix} v_1 \\ v_2 \\ \dots \\ v_{15} \end{bmatrix}_t \tag{11.9}
$$

The model is set up as follows:

```r
ns <- dim(dat.feb)[1]
B <- matrix(list(0), 2, 2)
B[1, 1] <- "b1"
B[2, 2] <- "b2"
Q <- diag(1, 2)
R <- "diagonal and unequal"
U <- "zero"
x0 <- "zero"
Z <- matrix(list(0), ns, 2)
Z[1:(ns * 2)] <- c(paste0("z1", 1:ns), paste0("z2", 1:ns))
Z[1, 2] <- 0
A <- "unequal"
mod.list.dfa = list(B = B, Z = Z, Q = Q, R = R, U = U, A = A,
    x0 = x0)
```

Now we can fit a MARSS model and get estimates of the missing SWEs. We pass in the initial value for **a** as the mean level so it fits easier.

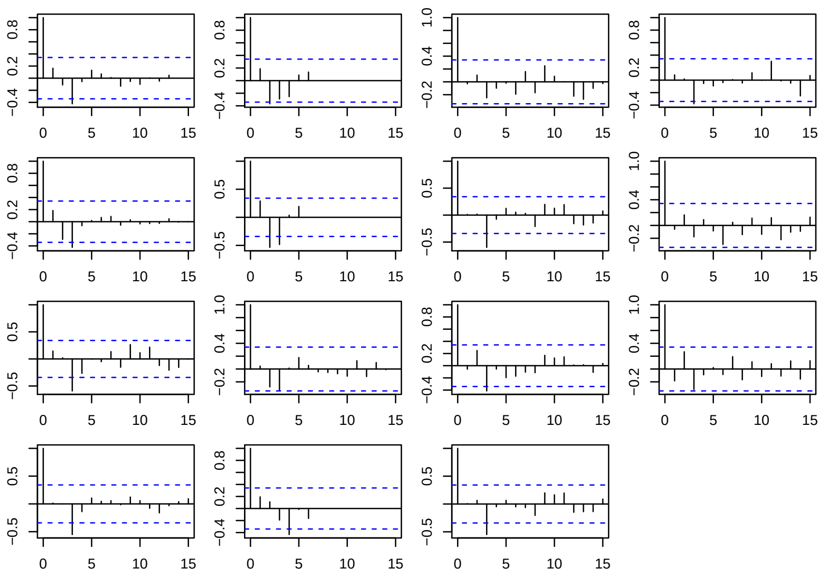
```r
library(MARSS)
m <- apply(dat.feb, 1, mean, na.rm = TRUE)
fit.dfa <- MARSS(dat.feb, model = mod.list.dfa, control = list(maxit = 1000),
    inits = list(A = matrix(m, ns, 1)))
```



11.2.4 Diagnostics

The model residuals are uncorrelated.

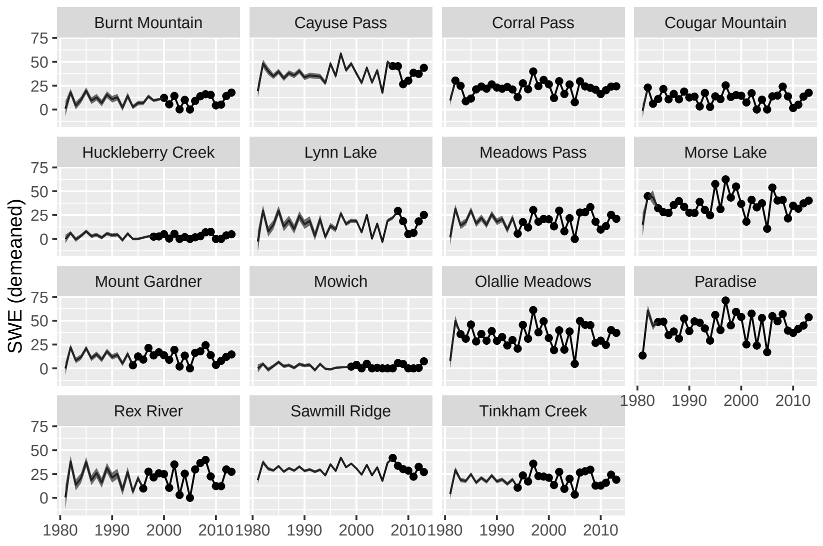
```
par(mfrow = c(4, 4), mar = c(2, 2, 1, 1))
apply(MARSSresiduals(fit, type = "tt1")$model.residual, 1, function(x) {
    acf(x, na.action = na.pass)
})
```



11.2.5 Plot the fitted or mean Feb SWE using DFA

The plots showed the estimate of the missing Feb SWE values, which is the expected value of $\mathbf{y}$ conditioned on all the data. For the non-missing SWE, this expected value is just the observation. Many times we want the model fit for the covariate. If the measurements have observation error, the fitted value is the estimate without this observation error.

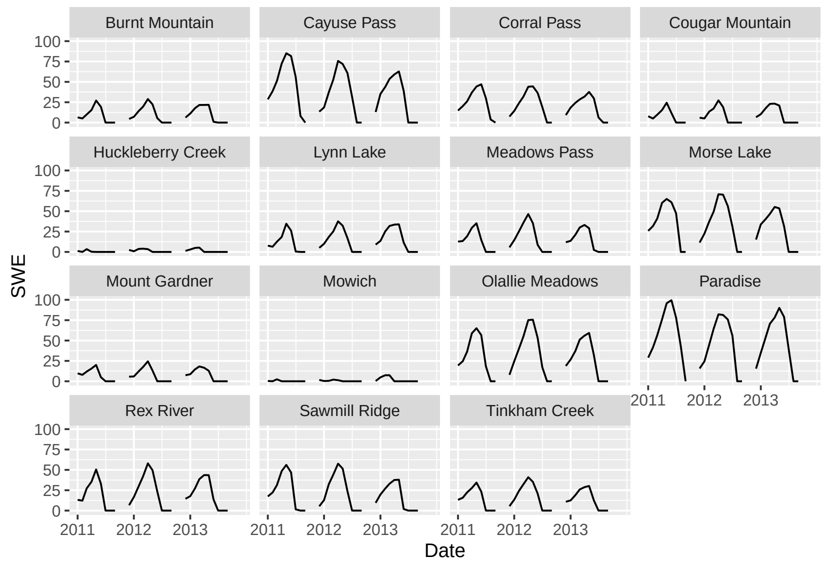
For the estimated states conditioned on all the data we want `tsSmooth()`. We will not show the prediction intervals which would be for new data. We will just show the confidence intervals on the fitted estimate for the missing values. The confidence intervals are small so they are a bit hard to see.



11.3 Modeling Seasonal SWE

When we look at all months, we see that SWE is highly seasonal. Note October and November are missing for all years.

```r
swe.yr <- snotel
swe.yr <- swe.yr[swe.yr$Station.Id %in% y$Station.Id, ]
swe.yr$Station <- droplevels(swe.yr$Station)
```



Set up the data matrix of monthly SNOTEL data:

```
dat.yr <- snotel
dat.yr <- dat.yr[dat.yr$Station.Id %in% y$Station.Id, ]
dat.yr$Station <- droplevels(dat.yr$Station)
dat.yr$Month <- factor(dat.yr$Month, level = month.abb)
dat.yr <- reshape2::acast(dat.yr, Station ~ Year + Month, value.var = "SWE")
```

We will model the seasonal differences using a periodic model. The covariates are

```
period <- 12
TT <- dim(dat.yr)[2]
cos.t <- cos(2 * pi * seq(TT)/period)
sin.t <- sin(2 * pi * seq(TT)/period)
c.seas <- rbind(cos.t, sin.t)
```

11.3.1 Modeling season across sites

We will create a state for the seasonal cycle and each station will have a scaled effect of that seasonal cycle. The observations will have the seasonal effect plus a mean and residuals (observation - season - mean) will be allowed to correlate across stations.

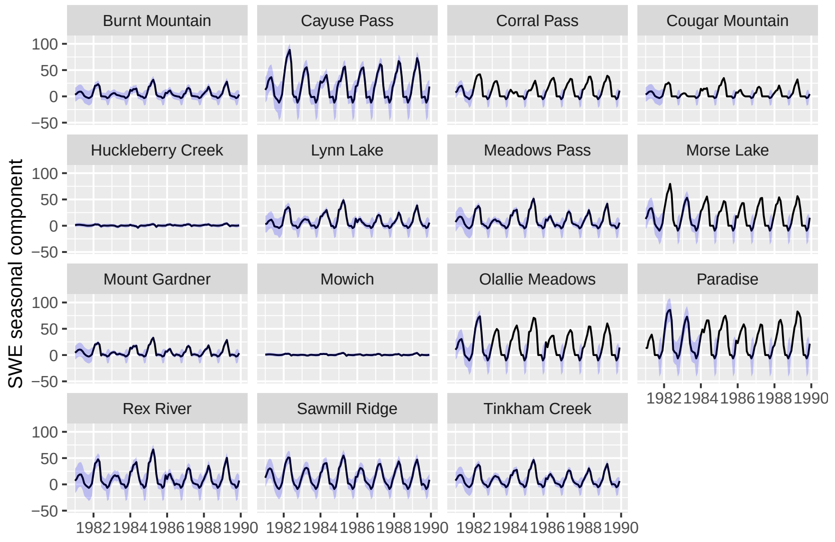
```
ns <- dim(dat.yr)[1]
B <- "zero"
Q <- matrix(1)
R <- "unconstrained"
U <- "zero"
x0 <- "zero"
Z <- matrix(paste0("z", 1:ns), ns, 1)
A <- "unequal"
mod.list.dfa = list(B = B, Z = Z, Q = Q, R = R, U = U, A = A,
    x0 = x0)
C <- matrix(c("c1", "c2"), 1, 2)
c <- c.seas
mod.list.seas <- list(B = B, U = U, Q = Q, A = A, R = R, Z = Z,
    C = C, c = c, x0 = x0, tinitx = 0)
```

Now we can fit the model:

```
m <- apply(dat.yr, 1, mean, na.rm = TRUE)
fit.seas <- MARSS(dat.yr, model = mod.list.seas, control = list(maxit = 500),
    inits = list(A = matrix(m, ns, 1)))
```

The seasonal patterns

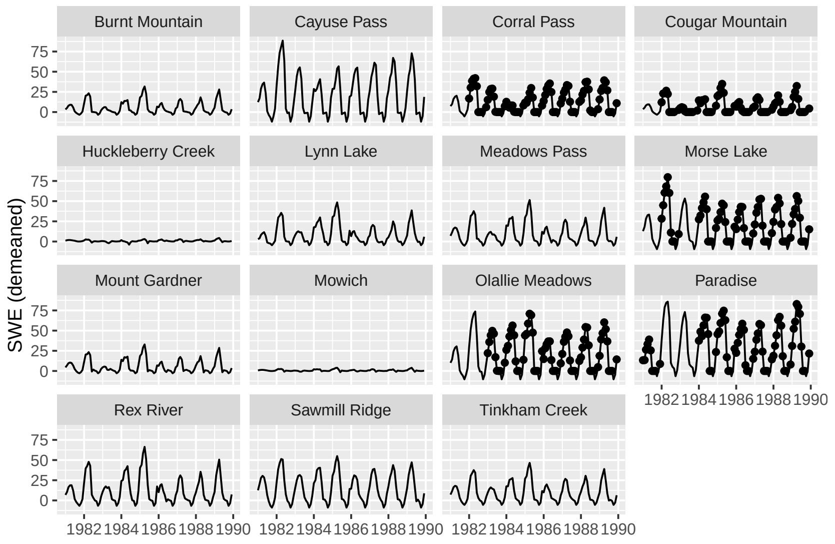
Figure @ref{fig:mssmiss-seas} shows the seasonal estimate plus prediction intervals for each station. This is $z_i x_i + a_i$. The prediction interval shows our estimate of the range of the data we would see around the seasonal estimate.



Estimates for the missing years

The estimated mean SWE at each station is $E(y_{t,i}|y_{1:T})$. This is the estimate of $y_{t,i}$ conditioned on all the data and includes the seasonal component plus the information from the data from other stations. If $y_{t,i}$ is observed, $E(y_{t,i}|y_{1:T}) = y_{t,i}$, i.e. just the observed value. But if $y_{t,i}$ is unobserved, the stations with data at time t help inform $y_{t,i}$, the value of the station without data at time t. Note this is not the case when we computed the fitted value for $y_{t,i}$. In that case, the data inform $\mathbf{R}$ but we do not treat the observed data at $t = i$ as 'observed' and influencing the missing the missing $y_{t,i}$ through $\mathbf{R}$.

Only years up to 1990 are shown, but the model is fit to all years. The stations with no data before 1990 are being estimated based on the information in the later years when they do have data. We did not constrain the SWE to be positive, so negative estimates are possible and occurs in the months in which we have no SWE data (because there is no snow).


Burnt Mountain
Cayuse Pass
Corral Pass
Cougar Mountain
Huckleberry Creek
Lynn Lake
Meadows Pass
Morse Lake
Mount Gardner
Mowich
Olallie Meadows
Paradise
Rex River
Sawmill Ridge
Tinkham Creek
SWE (demeaned)
75
50
25
0
1982
1984
1986
1988
1990

Chapter 12

JAGS for Bayesian time series analysis

In this lab, we will illustrate how to use JAGS to fit time series models with Bayesian methods. The purpose of this chapter is to teach you some basic JAGS models. To go beyond these basics, study the wide variety of software tools to do time series analysis using Bayesian methods, e.g. packages listed on the R Cran TimeSeries task view.

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

Data and packages

For data for this lab, we will use a dataset on air quality in New York. For the majority of our models, we are going to treat wind speed as the response variable for our time series models.

```r
data(airquality, package = "datasets")
Wind <- airquality$Wind  # wind speed
Temp <- airquality$Temp  # air temperature
N <- dim(airquality)[1]  # number of data points
```

To run this code, you will need to install JAGS for your operating platform using the instructions here. Click on JAGS, then the most recent folder, then the platform of your machine. You will also need the **coda**, **rjags** and **R2jags** packages.

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

12.1 Overview

In this chapter, we will be working up to simple univariate state-space JAGS models. We will write each of the models with the same univariate state-space form.

$$\begin{gathered} x_t = bx_{t-1} + u + w_t,\ w_t \sim N(0,q) \\ y_t = x_t + a + v_t,\ v_t \sim \text{ N}(0,r) \end{gathered} \tag{12.1}$$

We will be fitting linear regressions with this form, and this will mean the JAGS code is more verbose than necessary, but the goal is to build up to our univariate state-space code by building off simpler models.

12.2 Univariatate response models

12.2.1 Linear regression with no covariates

We will start with a linear regression with only an intercept. We will write the model in the form of Equation (12.1). Our model is

$$\begin{gathered} x_t = u \\ y_t = x_t + v_t, v_t \sim \text{ N}(0,r) \end{gathered} \tag{12.2}$$

An equivalent way to think about this model is

$$Y_t \sim \text{ N}(E[Y_t], r) \tag{12.3}$$

$E[Y_t] = x_t$ where $x_t = u$. In this linear regression model, we will treat the residual error as independent and identically distributed Gaussian observation error.

To run the JAGS model, we will need to start by writing the model in JAGS notation. We can construct the model in Equation (12.2) as

```
# LINEAR REGRESSION intercept only.

model.loc <- "lm_intercept.txt"  # name of the txt file
jagsscript <- cat("
model {
   # priors on parameters
   u ~ dnorm(0, 0.01);
   inv.r ~ dgamma(0.001,0.001); # This is inverse gamma
   r <- 1/inv.r; # derived value

   # likelihood
   for(i in 1:N) {
      X[i] <- u
      EY[i] <- X[i]; # derived value
      Y[i] ~ dnorm(EY[i], inv.r);
   }
}
",
    file = model.loc)
```

The JAGS code has three parts: our parameter priors, our data model and derived parameters.

Parameter priors There are two parameters in the model (u, the mean, and r, the variance of the observation error). We need to set a prior on both of these. We will set a vague prior of a Gaussian with varianc 10 on u. In JAGS instead of specifying the normal distribution with the variance, $N(0, 10)$, you specify it with the precision (1/variance), so our prior on u is `dnorm(0, 0.01)`. For r, we need to set a prior on the precision $1/r$, which we call `inv.r` in the code. The precision receives a gamma prior, which is equivalent to the variance receiving an inverse gamma prior (fairly common for standard Bayesian regression models).

Likelihood Our data distribution is $Y_t \sim \mathrm{N}(E[Y_t], r)$. We use the `dnorm()` distribution with the precision $(1/r)$ instead of r. So our data model is `Y[t] = dnorm(EY[t], inv.r)`. JAGS is not vectorized so we need to use for loops (instead of matrix multiplication) and use the for loop to specify the distribution for each `Y[t]`. For, this model we didn't actually need `X[t]` but we use it because we are building up to a state-space model which has both x_t and y_t.

Derived values Derived values are things we want output so we can track them. In this example, our derived values are a bit useless but in more complex models they will be quite handy. Also they can make your code easier to understand.

To run the model, we need to create several new objects, representing (1) a list of data that we will pass to JAGS `jags.data`, (2) a vector of parameters that we want to monitor and have returned back to R `jags.params`, and (3) the name of our text file that contains the JAGS model we wrote above. With those three things, we can call the `jags()` function.

```
jags.data <- list(Y = Wind, N = N)
jags.params <- c("r", "u")  # parameters to be monitored
mod_lm_intercept <- R2jags::jags(jags.data, parameters.to.save = jags.params,
    model.file = model.loc, n.chains = 3, n.burnin = 5000, n.thin = 1,
    n.iter = 10000, DIC = TRUE)
```

The function from the **R2jags** package that we use to run the model is `jags()`. There is a parallel version of the function called `jags.parallel()` which is useful for larger, more complex models. The details of both can be found with `?jags` or `?jags.parallel`.

Notice that the `jags()` function contains a number of other important arguments. In general, larger is better for all arguments: we want to run multiple MCMC chains (maybe 3 or more), and have a burn-in of at least 5000. The total number of samples after the burn-in period is n.iter-n.burnin, which in this case is 5000 samples. Because we are doing this with 3 MCMC chains, and the thinning rate equals 1 (meaning we are saving every sample), we will retain a total of 1500 posterior samples for each parameter.

The saved object storing our model diagnostics can be accessed directly, and includes some useful summary output.

```r
mod_lm_intercept
```

```
Inference for Bugs model at "lm_intercept.txt", fit using jags,
 3 chains, each with 10000 iterations (first 5000 discarded)
 n.sims = 15000 iterations saved
         mu.vect sd.vect    2.5%     25%     50%     75%   97.5%  Rhat n.eff
r         12.574   1.466  10.027  11.550  12.460  13.502  15.759 1.001 12000
u          9.950   0.286   9.382   9.759   9.951  10.142  10.504 1.001  6700
deviance 820.556   2.006 818.596 819.119 819.937 821.354 825.980 1.001 10000

For each parameter, n.eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor (at convergence, Rhat=1).

DIC info (using the rule, pD = var(deviance)/2)
pD = 2.0 and DIC = 822.6
DIC is an estimate of expected predictive error (lower deviance is better).
```

The last two columns in the summary contain `Rhat` (which we want to be close to 1.0), and `neff` (the effective sample size of each set of posterior draws). To examine the output more closely, we can pull all of the results directly into R,

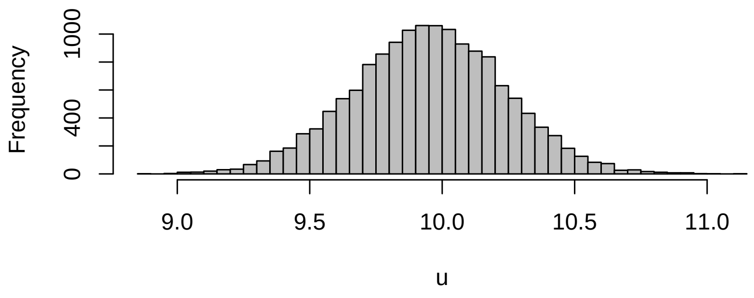
```r
R2jags::attach.jags(mod_lm_intercept)
```

Attaching the **R2jags** object loads the posteriors for the parameters and we can call them directly, e.g. `u`. If we don't want to attach them to our workspace, we can find the posteriors within the model object.

```r
post.params <- mod_lm_intercept$BUGSoutput$sims.list
```

We make a histogram of the posterior distributions of the parameters `u` and `r` with the following code,

```r
# Now we can make plots of posterior values
par(mfrow = c(2, 1))
hist(post.params$u, 40, col = "grey", xlab = "u", main = "")
hist(post.params$r, 40, col = "grey", xlab = "r", main = "")
```



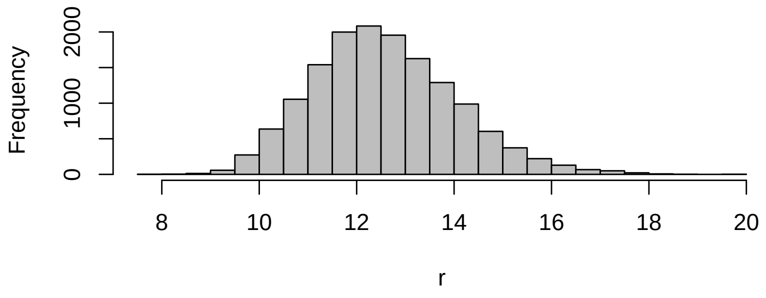


Figure 12.1: Plot of the posteriors for the linear regression model.

We can run some useful diagnostics from the **coda** package on this model output. We have written a small function to make the creation of a MCMC list (an argument required for many of the diagnostics). The function is

```
createMcmcList <- function(jagsmodel) {
    McmcArray <- as.array(jagsmodel$BUGSoutput$sims.array)
    McmcList <- vector("list", length = dim(McmcArray)[2])
    for (i in 1:length(McmcList)) McmcList[[i]] <- as.mcmc(McmcArray[,
        i, ])
    McmcList <- mcmc.list(McmcList)
    return(McmcList)
}
```

Creating the MCMC list preserves the random samples generated from each chain and allows you to extract the samples for a given parameter (such as μ) from any chain you want. To extract μ from the first chain, for example, you could use the following code. Because `createMcmcList()` returns a list of **mcmc** objects, we can summarize and plot these directly. Figure 12.2 shows the plot from `plot(myList[[1]])`.

```
myList <- createMcmcList(mod_lm_intercept)
summary(myList[[1]])
```

```

Iterations = 1:5000
Thinning interval = 1
Number of chains = 1
Sample size per chain = 5000

1. Empirical mean and standard deviation for each variable,
   plus standard error of the mean:

            Mean     SD Naive SE Time-series SE
deviance 820.526 1.9933 0.028189       0.028189
r         12.558 1.4396 0.020359       0.021002
u          9.944 0.2866 0.004053       0.004053

2. Quantiles for each variable:
```

```
             2.5%     25%     50%    75%  97.5%
deviance 818.599 819.104 819.893 821.36 825.95
r         10.073  11.542  12.436  13.48  15.71
u          9.377   9.757   9.946  10.14  10.50
```

```
plot(myList[[1]])
```

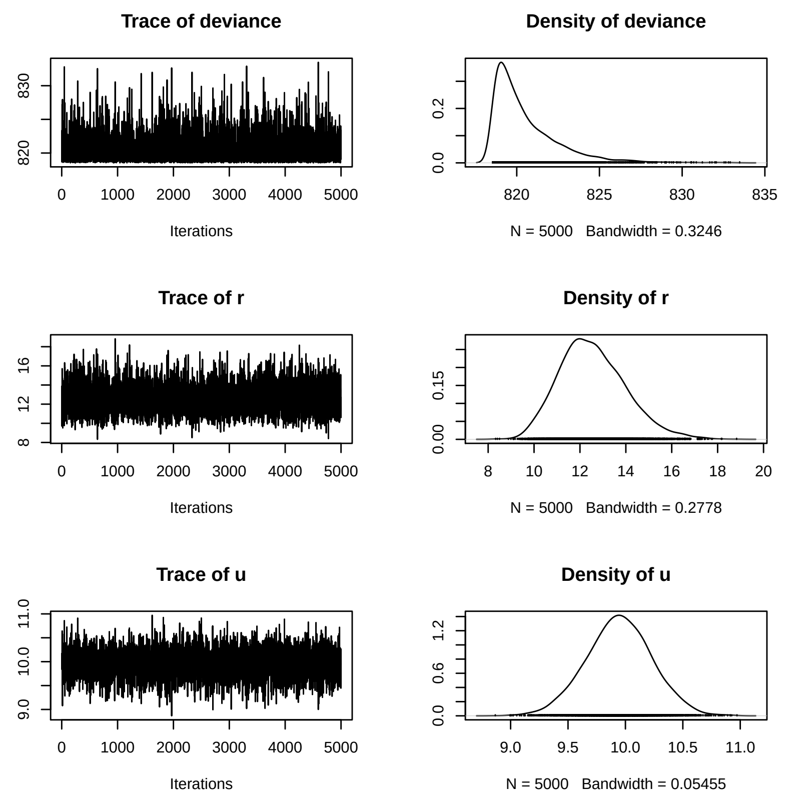


Figure 12.2: Plot of an object output from `creatMcmcList`.

For more quantitative diagnostics of MCMC convergence, we can rely on the **coda** package in R. There are several useful statistics available, including the Gelman-Rubin diagnostic (for one or several chains), autocorrelation diagnostics (similar to the ACF you calculated above), the Geweke diagnostic, and Heidelberger-Welch test of stationarity.

```
library(coda)
gelmanDiags <- coda::gelman.diag(createMcmcList(mod_lm_intercept),
    multivariate = FALSE)
autocorDiags <- coda::autocorr.diag(createMcmcList(mod_lm_intercept))
gewekeDiags <- coda::geweke.diag(createMcmcList(mod_lm_intercept))
heidelDiags <- coda::heidel.diag(createMcmcList(mod_lm_intercept))
```

12.2.2 Linear regression with covariates

We can introduce `Temp` as the covariate explaining our response variable `Wind`. Our new equation is

$$\begin{gathered} x_t = u + C\,c_t \\ y_t = x_t + v_t, v_t \sim \text{N}(0, r) \end{gathered} \tag{12.4}$$

To create JAGS code for this model, we (1) add a prior for our new parameter `C`, (2) update `X[i]` equation to include the new covariate, and (3) we include the new covariate in our named data list.

```
# 1. LINEAR REGRESSION with covariates

model.loc <- ("lm_covariate.txt")
jagsscript <- cat("
model {
   # priors on parameters
   u ~ dnorm(0, 0.01);
   C ~ dnorm(0,0.01);
   inv.r ~ dgamma(0.001,0.001);
   r <- 1/inv.r;

   # likelihood
   for(i in 1:N) {
      X[i] <- u + C*c[i];
      EY[i] <- X[i]
      Y[i] ~ dnorm(EY[i], inv.r);
   }
}
",
    file = model.loc)

jags.data <- list(Y = Wind, N = N, c = Temp)
jags.params <- c("r", "EY", "u", "C")
mod_lm <- R2jags::jags(jags.data, parameters.to.save = jags.params,
    model.file = model.loc, n.chains = 3, n.burnin = 5000, n.thin = 1,
    n.iter = 10000, DIC = TRUE)
```

We can show the the posterior fits (the model fits) to the data. Here is a simple function whose arguments are one of our fitted models and the raw data. The function is:

```
plotModelOutput <- function(jagsmodel, Y) {
    # attach the model
    EY <- jagsmodel$BUGSoutput$sims.list$EY
    x <- seq(1, length(Y))
    summaryPredictions <- cbind(apply(EY, 2, quantile, 0.025),
        apply(EY, 2, mean), apply(EY, 2, quantile, 0.975))
    plot(Y, col = "white", ylim = c(min(c(Y, summaryPredictions)),
        max(c(Y, summaryPredictions))), xlab = "", ylab = "95% CIs of predicti
        main = paste("JAGS results:", jagsmodel$model.file))
    polygon(c(x, rev(x)), c(summaryPredictions[, 1], rev(summaryPredictions[,
        3])), col = "grey70", border = NA)
    lines(summaryPredictions[, 2])
    points(Y)
}
```

We can use the function to plot the predicted posterior mean with 95% CIs, as well as the raw data. Note that the shading is for the CIs on the expected value of y_t so will look narrow relative to the data. For example, try

```
plotModelOutput(mod_lm, Wind)
```

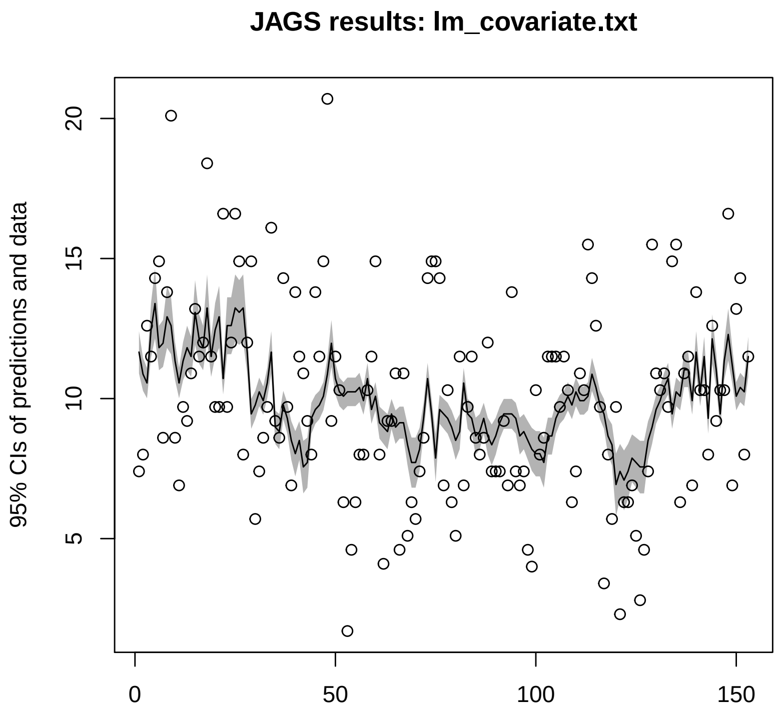


Figure 12.3: Predicted posterior mean with 95% CIs

12.2.3 Random walk with drift

The previous models were observation error only models. Switching gears, we can create process error models. We will start with a random walk model. In this model, the assumption is that the underlying state x_t is measured perfectly. All stochasticity is originating from process variation: variation in x_t to x_{t+1}.

For this simple model, we will assume that wind behaves as a random walk. We will call this process x to prepare for the state-space model to come. We have no y_t part of the equation in this model.

$$x_t = x_{t-1} + u + w_t, \text{ where } w_t \sim \text{N}(0, q) \tag{12.5}$$

Now x_t is stochastic and $E[X_t] = x_{t-1} + u$ and $X_t \sim \text{N}(E[X_t], q)$.

We are going to need to put a prior on x_0, which appears in $E[X_1]$. We could start with $t = 2$ and skip this but we will start at $t = 1$ since we will need to do that for later problems. The question is what prior should we put on x_0? This is not a stationary process. We will just put a vague prior on x_0.

The JAGS random walk model is:

```
# RANDOM WALK with drift

model.loc <- ("rw_intercept.txt")
jagsscript <- cat("
model {
   # priors on parameters
   u ~ dnorm(0, 0.01);
   inv.q ~ dgamma(0.001,0.001);
   q <- 1/inv.q;
   X0 ~ dnorm(0, 0.001);

   # likelihood
   X[1] ~ dnorm(X0 + u, inv.q);
   for(i in 2:N) {
      X[i] ~ dnorm(X[i-1] + u, inv.q);
   }
}
",
    file = model.loc)
```

To fit this model, we need to change `jags.data` to pass in `X = Wind` instead of `Y = Wind`. Obvioously we could have written the JAGS code with `Y` in place of `X` and kept our `jags.data` code the same as before, but we are working up to a state-space model where we have a hidden random walk called `X` and an observation of that called `Y`.

```
jags.data <- list(X = Wind, N = N)
jags.params <- c("q", "u")
mod_rw_intercept <- R2jags::jags(jags.data, parameters.to.save = jags.params,
    model.file = model.loc, n.chains = 3, n.burnin = 5000, n.thin = 1,
    n.iter = 10000, DIC = TRUE)
```

12.2.4 Autoregressive AR(1) time series models

A variation of the random walk model is the autoregressive time series model of order 1, AR(1). This model introduces a coefficient, which we will call b. The parameter b controls the degree to which the random walk reverts to the mean. When $b = 1$, the model is identical to the random walk, but at smaller values, the model will revert back to the mean (which in this case is zero). Also, b can take on negative values.

$$x_t = b\,x_{t-1} + u + w_t, \text{ where } w_t \sim \mathrm{N}(0, q) \tag{12.6}$$

Now $E[X_t] = b\,x_{t-1} + u$.

Once again we need to put a prior on x_0, which appears in $E[X_1]$. An AR(1) with $|b| < 1$ is a stationary process and the variance of the stationary distribution of x_t is $q/(1 - b^2)$. If you think that x_0 has the stationary distribution (does your data look stationary?) then you can use the variance of the stationary distribution of x_t for your prior. We specify priors with the precision (1 over the variance) instead of the variance. Thus the precision of the stationary distribution of x_0 is $(1/q)(1 - b^2)$. In the code, `inv.q` is $1/q$ and the precision is `inv.q * (1-b*b)`.

```
# AR(1) MODEL WITH AND ESTIMATED AR COEFFICIENT

model.loc <- ("ar1_intercept.txt")
jagsscript <- cat("
model {
   # priors on parameters
   u ~ dnorm(0, 0.01);
   inv.q ~ dgamma(0.001,0.001);
   q <- 1/inv.q;
   b ~ dunif(-1,1);
   X0 ~ dnorm(0, inv.q * (1 - b * b));

   # likelihood
   X[1] ~ dnorm(b * X0 + u, inv.q);
   for(t in 2:N) {
      X[t] ~ dnorm(b * X[t-1] + u, inv.q);
   }
}
",
    file = model.loc)

jags.data <- list(X = Wind, N = N)
jags.params <- c("q", "u", "b")
mod_ar1_intercept <- R2jags::jags(jags.data, parameters.to.save = jags.params,
    model.file = model.loc, n.chains = 3, n.burnin = 5000, n.thin = 1,
    n.iter = 10000, DIC = TRUE)
```

12.2.5 Regression with AR(1) errors

The AR(1) model in the previous section suggests a way that we could include correlated errors in our linear regression. We could use the x_t AR(1) process as our errors for y_t. Here is an example of modifying the intercept only linear regression model. We will set u to 0 so that our AR(1) errors have a mean of 0.

$$\begin{gathered} x_t = b\,x_{t-1} + w_t, \text{ where } w_t \sim \text{N}(0, q) \\ y_t = a + x_t \end{gathered} \tag{12.7}$$

The problem with this is that we need a distribution for y_t. We cannot use `Y[t] <- a + X[t]` in our JAGS code (Y_t is a random varible with a distribution; you cannot assign it a value). We need to re-write this as $Y_t \sim N(a + b\,x_{t-1}, q)$.

$$\begin{gathered} Y_t \sim N(a + b\,x_{t-1}, q) \\ x_t = y_t - a \end{gathered} \tag{12.8}$$

We will create the variable `EY` so we can keep track of the expected value of Y_t, conditioned on $t - 1$.

```
# LINEAR REGRESSION with autocorrelated errors no
# covariates, intercept only.

model.loc <- ("lm_intercept_ar1b.txt")
jagsscript <- cat("
model {
   # priors on parameters
   a ~ dnorm(0, 0.01);
   inv.q ~ dgamma(0.001,0.001);
   q <- 1/inv.q;
   b ~ dunif(-1,1);
   X0 ~ dnorm(0, inv.q * (1 - b * b));

   # likelihood
   EY[1] <- a + b * X0;
   Y[1] ~ dnorm(EY[1], inv.q);
   X[1] <- Y[1] - a;
   for(t in 2:N) {
      EY[t] <- a + b * X[t-1];
      Y[t] ~ dnorm(EY[1], inv.q);
      X[t] <- Y[t]-a;
   }
}
",
    file = model.loc)

jags.data <- list(Y = Wind, N = N)
jags.params <- c("q", "EY", "a", "b")
mod_ar1_intercept <- R2jags::jags(jags.data, parameters.to.save = jags.params,
    model.file = model.loc, n.chains = 3, n.burnin = 5000, n.thin = 1,
    n.iter = 10000, DIC = TRUE)
```

12.2.6 Univariate state space model

Now we will combine the process and observation models to create a univariate state-space model. This is the classic stochastic level model.

$$\begin{gathered} x_t = x_{t-1} + u + w_t,\, w_t \sim N(0,q) \\ y_t = x_t + v_t,\, v_t \sim \mathrm{N}(0,r) \end{gathered} \tag{12.9}$$

Because x is a random walk model not a stationary AR(1), we will place a vague weakly informative prior on x_0: $x_0 \sim \mathrm{N}(y_1, 1000)$. We had to pass in `Y1` as data because JAGS would complain if we used `Y[1]` in our prior (because have `X0` in our model for $Y[1]$). `EY` is added so that we can track the model fits for y. In this case it is just `X` but in more complex models it will involve more parameters.

```
model.loc <- ("ss_model.txt")
jagsscript <- cat("
model {
   # priors on parameters
   u ~ dnorm(0, 0.01);
   inv.q ~ dgamma(0.001,0.001);
   q <- 1/inv.q;
   inv.r ~ dgamma(0.001,0.001);
   r <- 1/inv.r;
   X0 ~ dnorm(Y1, 0.001);

   # likelihood
   X[1] ~ dnorm(X0 + u, inv.q);
   EY[1] <- X[1];
   Y[1] ~ dnorm(EY[1], inv.r);
   for(t in 2:N) {
      X[t] ~ dnorm(X[t-1] + u, inv.q);
      EY[t] <- X[t];
      Y[t] ~ dnorm(EY[t], inv.r);
   }
}
",
    file = model.loc)
```

We fit as usual with the addition of `Y1` in `jags.data`.

```
jags.data <- list(Y = Wind, N = N, Y1 = Wind[1])
jags.params <- c("q", "r", "EY", "u")
mod_ss <- jags(jags.data, parameters.to.save = jags.params, model.file = model.loc,
    n.chains = 3, n.burnin = 5000, n.thin = 1, n.iter = 10000,
    DIC = TRUE)
```

12.3 Multivariate state-space models

In the multivariate state-space model, our observations and hidden states can be multivariate along with all the parameters:

$$\begin{gathered}\mathbf{x}_t = \mathbf{B}\mathbf{x}_{t-1} + \mathbf{u} + \mathbf{w}_t \text{ where } \mathbf{w}_t \sim \mathrm{N}(0, \mathbf{Q}) \\ \mathbf{y}_t = \mathbf{Z}\mathbf{x}_t + \mathbf{a} + \mathbf{v}_t \text{ where } \mathbf{v}_t \sim \mathrm{N}(0, \mathbf{R}) \\ \mathbf{x}_0 = \boldsymbol{\mu}\end{gathered} \tag{12.10}$$

12.3.1 One hidden state

Let's start with a very simple MARSS model with JAGS: two observation time-series and one hidden state. Our $\mathbf{x}_t$ model is $x_t = x_{t-1} + u + w_t$ and our $\mathbf{y}_t$ model is

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix}_t = \begin{bmatrix} 1 \\ 1 \end{bmatrix} x_t + \begin{bmatrix} 0 \\ a_2 \end{bmatrix} + \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}_t, \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}_t \sim \mathrm{MVN}\left(0, \begin{bmatrix} r_1 & 0 \\ 0 & r_2 \end{bmatrix}\right) \tag{12.11}$$

We need to put a prior on our x_0 (initial x). Since $b = 1$, we have a random walk rather than a stationary process and we will put a vague prior on the x_0. We need to deal with the **a** so that our code doesn't run in circles by trying to match x up with different y_t time series. We force x_t to track the mean of $y_{1,t}$ and then use a_2 to scale the other y_t relative to that. The problem is that a random walk is very flexible and if we tried to estimate a_1 then we would have infinite solutions.

To keep our JAGS code organized, let's separate the **x** and **y** parts of the code.

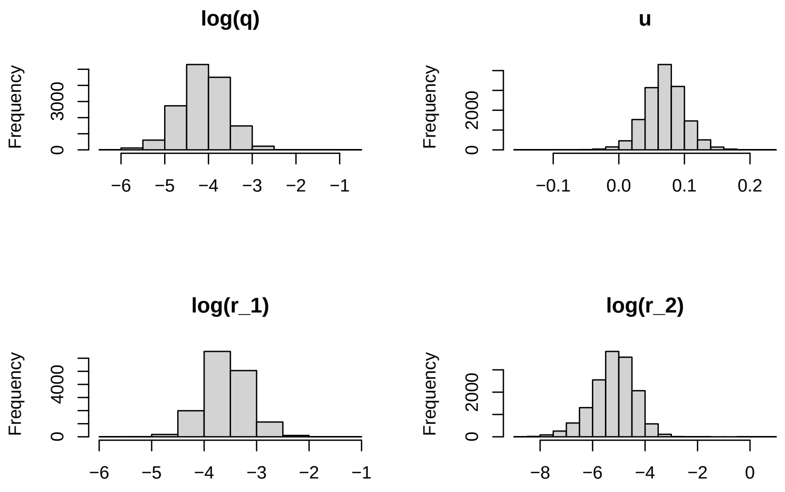
```
jagsscript <- cat("
model {
   # process model priors
   u ~ dnorm(0, 0.01); # one u
   inv.q~dgamma(0.001,0.001);
   q <- 1/inv.q; # one q
   X0 ~ dnorm(Y1,0.001); # initial state
   # process model likelihood
   EX[1] <- X0 + u;
   X[1] ~ dnorm(EX[1], inv.q);
   for(t in 2:N) {
         EX[t] <- X[t-1] + u;
         X[t] ~ dnorm(EX[t], inv.q);
   }

   # observation model priors
   for(i in 1:n) { # r's differ by site
     inv.r[i]~dgamma(0.001,0.001);
     r[i] <- 1/inv.r[i];
   }
   a[1] <- 0; # first a is 0, rest estimated
   for(i in 2:n) {
     a[i]~dnorm(0,0.001);
   }
   # observation model likelihood
   for(t in 1:N) {
     for(i in 1:n) {
       EY[i,t] <- X[t]+a[i]
       Y[i,t] ~ dnorm(EY[i,t], inv.r[i]);
     }
   }
}

",
    file = "marss-jags1.txt")
```

To fit the model, we write the data list, parameter list, and pass the model to the `jags()` function.

```
data(harborSealWA, package = "MARSS")
dat <- t(harborSealWA[, 2:3])
jags.data <- list(Y = dat, n = nrow(dat), N = ncol(dat), Y1 = dat[1,
    1])
jags.params <- c("EY", "u", "q", "r")
model.loc <- "marss-jags1.txt"  # name of the txt file
mod_marss1 <- R2jags::jags(jags.data, parameters.to.save = jags.params,
    model.file = model.loc, n.chains = 3, n.burnin = 5000, n.thin = 1,
    n.iter = 10000, DIC = TRUE)
```

We can make a plot of our estimated parameters:

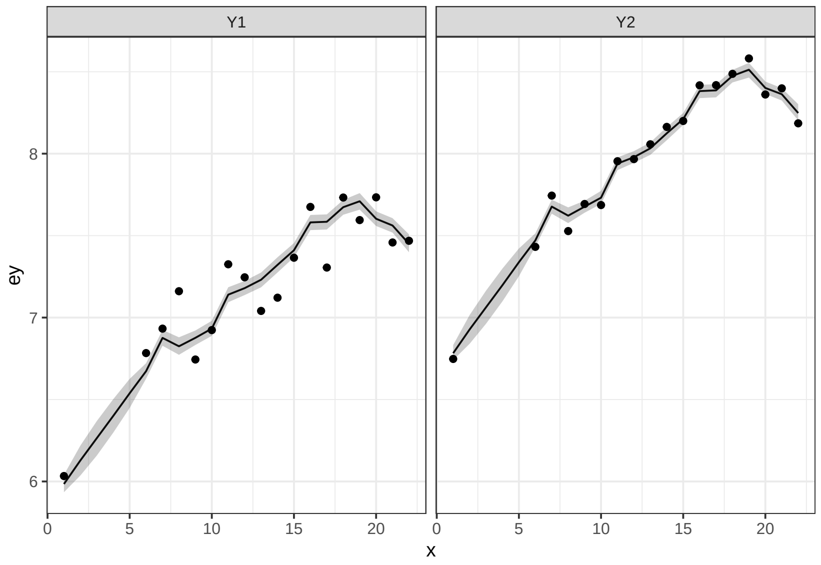
```
post.params <- mod_marss1$BUGSoutput$sims.list
par(mfrow = c(2, 2))
hist(log(post.params$q), main = "log(q)", xlab = "")
hist(post.params$u, main = "u", xlab = "")
hist(log(post.params$r[, 1]), main = "log(r_1)", xlab = "")
hist(log(post.params$r[, 2]), main = "log(r_2)", xlab = "")
```



We can make a plot of the model fitted y_t with 50% credible intervals and the data. Note that the credible intervals are for the expected value of $y_{i,t}$ so will be narrower than the data.

```r
make.ey.plot <- function(mod, dat) {
    library(ggplot2)
    EY <- mod$BUGSoutput$sims.list$EY
    n <- nrow(dat)
    N <- ncol(dat)
    df <- c()
    for (i in 1:n) {
        tmp <- data.frame(n = paste0("Y", i), x = 1:N, ey = apply(EY[,
            i, , drop = FALSE], 3, median), ey.low = apply(EY[,
            i, , drop = FALSE], 3, quantile, probs = 0.25), ey.up = apply(EY[,
            i, , drop = FALSE], 3, quantile, probs = 0.75), y = dat[i,
            ])
        df <- rbind(df, tmp)
    }
    ggplot(df, aes(x = x, y = ey)) + geom_line() + geom_ribbon(aes(ymin = ey.l
        ymax = ey.up), alpha = 0.25) + geom_point(data = df,
        aes(x = x, y = y)) + facet_wrap(~n) + theme_bw()
}
```

```
make.ey.plot(mod_marss1, dat)
```



12.3.2 m hidden states

Let's add multiple hidden states. We'll say that each y_t is observing its own x_t but the x_t share the same q but not u. Our $\mathbf{x}_t$ model is

$$\begin{bmatrix}x_1\\x_2\end{bmatrix}_t = \begin{bmatrix}1&0\\0&1\end{bmatrix}\begin{bmatrix}x_1\\x_2\end{bmatrix}_{t-1} + \begin{bmatrix}u_1\\u_2\end{bmatrix} + \begin{bmatrix}w_1\\w_2\end{bmatrix}_t, \begin{bmatrix}w_1\\w_2\end{bmatrix}_t \sim \text{MVN}\left(0, \begin{bmatrix}q&0\\0&q\end{bmatrix}\right) \quad (12.12)$$

Here is the JAGS model. Note that a_i is 0 for all i because each y_t is associated with its own x_t.

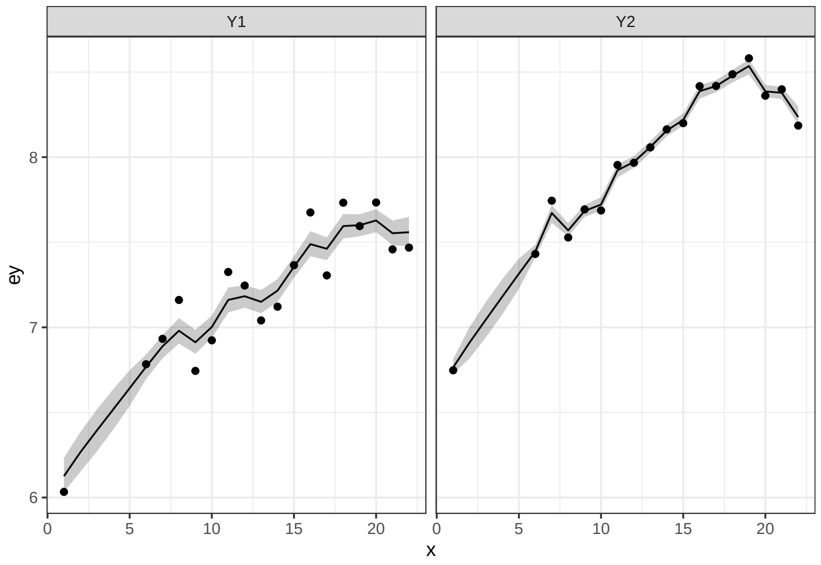
```
jagsscript <- cat("
model {
   # process model priors
   inv.q~dgamma(0.001,0.001);
   q <- 1/inv.q; # one q
   for(i in 1:n) {
      u[i] ~ dnorm(0, 0.01);
      X0[i] ~ dnorm(Y1[i],0.001); # initial states
   }
   # process model likelihood
   for(i in 1:n) {
     EX[i,1] <- X0[i] + u[i];
     X[i,1] ~ dnorm(EX[i,1], inv.q);
   }
   for(t in 2:N) {
      for(i in 1:n) {
         EX[i,t] <- X[i,t-1] + u[i];
         X[i,t] ~ dnorm(EX[i,t], inv.q);
      }
   }

   # observation model priors
   for(i in 1:n) { # The r's are different by site
     inv.r[i]~dgamma(0.001,0.001);
     r[i] <- 1/inv.r[i];
   }
   # observation model likelihood
   for(t in 1:N) {
     for(i in 1:n) {
       EY[i,t] <- X[i,t]
       Y[i,t] ~ dnorm(EY[i,t], inv.r[i]);
     }
   }
}

",
    file = "marss-jags2.txt")
```

Our code to fit the model changes a little.

```
data(harborSealWA, package = "MARSS")
dat <- t(harborSealWA[, 2:3])
jags.data <- list(Y = dat, n = nrow(dat), N = ncol(dat), Y1 = dat[,
    1])
```

```
jags.params <- c("EY", "u", "q", "r")
model.loc <- "marss-jags2.txt"  # name of the txt file
mod_marss1 <- R2jags::jags(jags.data, parameters.to.save = jags.params,
    model.file = model.loc, n.chains = 3, n.burnin = 5000, n.thin = 1,
    n.iter = 10000, DIC = TRUE)
```

```
make.ey.plot(mod_marss1, dat)
```



12.4 Non-Gaussian observation errors

12.4.1 Poisson observation errors

So far we have used the following observation model $y_t \sim \text{N}(x_t, r)$. We can change this to a Poisson observation error model: $Y_t \sim \text{Pois}(\lambda_t)$ where $E[Y_t] = \lambda_t$. $\log(\lambda_t) = x_t$ where x_t is our process model.

All we need to change to allow Poisson errors is to change the `Y[t]` part to

```
log(EY[t]) <- X[i]
Y[t] ~ dpois(EY[t])
```

We also need to ensure that our data are integers and we remove the `r` part from our model code since the Poisson does not have that.

Our univariate state-space code with Poisson observation errors is the following:

```
# SS MODEL with Poisson errors

model.loc <- ("ss_model_pois.txt")
jagsscript <- cat("
model {
   # priors on parameters
   u ~ dnorm(0, 0.01);
   inv.q ~ dgamma(0.001,0.001);
   q <- 1/inv.q;
   X0 ~ dnorm(0, 0.001);

   # likelihood
   X[1] ~ dnorm(X0 + u, inv.q);
   log(EY[1]) <- X[1]
   Y[1] ~ dpois(EY[1])
   for(t in 2:N) {
      X[t] ~ dnorm(X[t-1] + u, inv.q);
      log(EY[t]) <- X[t]
      Y[t] ~ dpois(EY[t]);
   }
}
",
    file = model.loc)
```

We will fit this to the wild dogs data in the **MARSS** package.

```
data(wilddogs, package = "MARSS")
jags.data <- list(Y = wilddogs[, 2], N = nrow(wilddogs))
jags.params <- c("q", "EY", "u")
mod_ss <- jags(jags.data, parameters.to.save = jags.params, model.file = model
    n.chains = 3, n.burnin = 5000, n.thin = 1, n.iter = 10000,
    DIC = TRUE)
```

When we use this univariate state-space model with population data, like the wild dogs, we would log the data†, and our y_t in our JAGS code is really $log(y_t)$. In that case, $E[log(Y_t)]) = f(x_t)$. So there is a log-link that we are not really explicit about when we pass in the log of our data. In the Poisson model, that log relationship is explicit, aka we specify $log(E[Y_t]) = x_t$ and we pass in the raw count data not the log of the data.

† Why would we typically log population data in this case? Because we would typically think of population processes as multiplicative. Population size at time t is growth **times** population size at time $t-1$. By logging the data, we convert to an additive process. Log population size at time t is log growth **plus** log population size at time $t-1$.

12.4.2 Negative binomial observation errors

In the Poisson distribution, the mean and variance are the same. Using the negative binomial distribution, we can relax that assumption and allow the mean and variance to be different. The negative binomial distribution has two parameters, r and p. r is the dispersion parameter. As $r \to \infty$, the distribution becomes the Poisson distribution and when r is small, the distribution is overdispersed (higher variance) relative to the Poisson. In practice, $r > 30$ is going to be very close to the Poisson. p is the success parameter, $p = r/(r + E[Y_t])$. As for the Poisson, $logE[Y_t] = x_t$—for the univariate state-space model in this example with one state, $z = 1$ and $a = 0$.

To allow negative binomial errors we change the `Y[t]` part to

```
log(EY[t]) <- X[t]
p[t] <- r/(r + EY[t])
Y[t] ~ dnegbin(p[t], r)
```

Now that we have r again in the model, we will need to put a prior on it. r is positive and 50 is close to infinity. The following is a sufficiently vague prior.

```
r ~ dunif(0,50)
```

Our univariate state-space code with negative binomial observation errors is the following:

```
# SS MODEL with negative binomial errors

model.loc <- ("ss_model_negbin.txt")
jagsscript <- cat("
model {
   # priors on parameters
   u ~ dnorm(0, 0.01);
   inv.q ~ dgamma(0.001,0.001);
   q <- 1/inv.q;
   r ~ dunif(0,50);
   X0 ~ dnorm(0, 0.001);

   # likelihood
   X[1] ~ dnorm(X0 + u, inv.q);
   log(EY[1]) <- X[1]
   p[1] <- r/(r + EY[1])
   Y[1] ~ dnegbin(p[1], r)
   for(t in 2:N) {
      X[t] ~ dnorm(X[t-1] + u, inv.q);
      log(EY[t]) <- X[t]
      p[t] <- r/(r + EY[t])
      Y[t] ~ dnegbin(p[t], r)
   }
}
",
    file = model.loc)
```

We will fit this to the wild dogs data in the **MARSS** package.

```
data(wilddogs, package = "MARSS")
jags.data <- list(Y = wilddogs[, 2], N = nrow(wilddogs))
jags.params <- c("q", "EY", "u", "r")
mod_ss <- jags(jags.data, parameters.to.save = jags.params, model.file = model
    n.chains = 3, n.burnin = 5000, n.thin = 1, n.iter = 10000,
    DIC = TRUE)
```

12.5 Forecasting with JAGS models

There are a number of different approaches to using Bayesian time series models to perform forecasting. One approach might be to fit a model, and use those posterior distributions to forecast as a secondary step (say within R). A more streamlined approach is to do this within the JAGS code itself. We can take advantage of the fact that JAGS allows you to include NAs in the response variable (but never in the predictors). Let's use the same Wind dataset, and the univariate state-space model described above to forecast three time steps into the future. We can do this by including 3 more NAs in the dataset, and incrementing the variable `N` by 3.

```
jags.data <- list(Y = c(Wind, NA, NA, NA), N = (N + 3), Y1 = Wind[1])
jags.params <- c("q", "r", "EY", "u")
model.loc <- ("ss_model.txt")
mod_ss_forecast <- jags(jags.data, parameters.to.save = jags.params,
    model.file = model.loc, n.chains = 3, n.burnin = 5000, n.thin = 1,
    n.iter = 10000, DIC = TRUE)
```

We can inspect the fitted model object, and see that `EY` contains the 3 new predictions for the forecasts from this model.

12.6 Problems

1. Fit the intercept only model from section 12.2.1. Set the burn-in to 3, and when the model completes, plot the time series of the parameter `u` for the first MCMC chain.

   a. Based on your visual inspection, has the MCMC chain convered?

   b. What is the ACF of the first MCMC chain?

2. Increase the MCMC burn-in for the model in question 1 to a value that you think is reasonable. After the model has converged, calculate the Gelman-Rubin diagnostic for the fitted model object.

3. Compare the results of the `plotModelOutput()` function for the intercept only model from section 12.2.1. You will to add "predY" to your JAGS model and to the list of parameters to monitor, and re-run the model.

4. Plot the posterior distribution of b for the AR(1) model in section 12.2.4. Can this parameter be well estimated for this dataset?

5. Plot the posteriors for the process and observation variances (not standard deviation) for the univariate state-space model in section 12.2.6. Which is larger for this dataset?

6. Add the effect of temperature to the AR(1) model in section 12.2.4. Plot the posterior for `C` and compare to the posterior for `C` from the model in section 12.2.2.

7. Plot the fitted values from the model in section 12.5, including the forecasts, with the 95% credible intervals for each data point.

8. The following is a dataset from the Upper Skagit River (Puget Sound, 1952-2005) on salmon spawners and recruits:

```
Spawners <- c(2662, 1806, 1707, 1339, 1686, 2220, 3121, 5028,
    9263, 4567, 1850, 3353, 2836, 3961, 4624, 3262, 3898, 3039,
    5966, 5931, 7346, 4911, 3116, 3185, 5590, 2485, 2987, 3829,
    4921, 2348, 1932, 3151, 2306, 1686, 4584, 2635, 2339, 1454,
    3705, 1510, 1331, 942, 884, 666, 1521, 409, 2388, 1043, 3262,
    2606, 4866, 1161, 3070, 3320)
Recruits <- c(12741, 15618, 23675, 37710, 62260, 32725, 8659,
    28101, 17054, 29885, 33047, 20059, 35192, 11006, 48154, 35829,
    46231, 32405, 20782, 21340, 58392, 21553, 27528, 28246, 35163,
    15419, 16276, 32946, 11075, 16909, 22359, 8022, 16445, 2912,
    17642, 2929, 7554, 3047, 3488, 577, 4511, 1478, 3283, 1633,
    8536, 7019, 3947, 2789, 4606, 3545, 4421, 1289, 6416, 3647)
logRS <- log(Recruits/Spawners)
```

a. Fit the following Ricker model to these data using the following linear form of this model with normally distributed errors:

$$log(R_t/S_t) = a + b \times S_t + e_t, \text{ where } e_t \sim \text{N}(0, \sigma^2)$$

You will recognize that this form is exactly the same as linear regression, with independent errors (very similar to the intercept only model of Wind we fit in section 12.2.1).

b. Within the constraints of the Ricker model, think about other ways you might want to treat the errors. The basic model described above has independent errors that are not correlated in time. Approaches to analyzing this dataset might involve

    - modeling the errors as independent (as described above)
    - modeling the errors as autocorrelated
    - fitting a state-space model, with independent or correlated process errors

Fit each of these models, and compare their performance (either using their predictive ability, or forecasting ability).

Chapter 13

Stan for Bayesian time series analysis

For this lab, we will use Stan for fitting models. These examples are primarily drawn from the Stan manual and previous code from this class.

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

Data and packages

You will need the **atsar** and **bayesdfa** packages we have written for fitting state-space time series models with Stan. Install using the **devtools** package.

```r
library(devtools)
# Windows users will likely need to set this
# Sys.setenv('R_REMOTES_NO_ERRORS_FROM_WARNINGS' = 'true')
devtools::install_github("nwfsc-timeseries/atsar")
devtools::install_github("nwfsc-timeseries/tvvarss")
devtools::install_github("fate-ewi/bayesdfa")
```

In addition, you will need the **rstan**, **datasets**, **parallel** and **loo** packages. After installing, if needed, load the packages:

```r
library(atsar)
library(rstan)
library(loo)
```

Once you have Stan and **rstan** installed, optimize Stan on your machine:

```r
rstan_options(auto_write = TRUE)
options(mc.cores = parallel::detectCores())
```

For this lab, we will use a data set on air quality in New York from the **datasets** package. Load the data and create a couple new variables for future use.

```r
data(airquality, package = "datasets")
Wind <- airquality$Wind  # wind speed
Temp <- airquality$Temp  # air temperature
```

13.1 Linear regression

We'll start with the simplest time series model possible: linear regression with only an intercept, so that the predicted values of all observations are the same. There are several ways we can write this equation. First, the predicted values can be written as $E[Y_t] = \beta x$, where $x = 1$. Assuming that the residuals are normally distributed, the model linking our predictions to observed data is written as

$$y_t = \beta x + e_t, e_t \sim N(0, \sigma), x = 1$$

An equivalent way to think about this model is that instead of the residuals as normally distributed with mean zero, we can think of the data y_t as being drawn from a normal distribution with a mean of the intercept, and the same residual standard deviation:

$$Y_t \sim N(E[Y_t], \sigma)$$

Remember that in linear regression models, the residual error is interpreted as independent and identically distributed observation error.

To run this model using our package, we'll need to specify the response and predictor variables. The covariate matrix with an intercept only is a matrix of 1s. To double check, you could always look at

```
x <- model.matrix(lm(Temp ~ 1))
```

Fitting the model using our function is done with this code,

```
lm_intercept <- atsar::fit_stan(y = as.numeric(Temp), x = rep(1,
    length(Temp)), model_name = "regression")
```

Coarse summaries of `stanfit` objects can be examined by typing one of the following

```
lm_intercept
# this is huge
summary(lm_intercept)
```

But to get more detailed output for each parameter, you have to use the `extract()` function,

```
pars <- rstan::extract(lm_intercept)
names(pars)
```

```
[1] "beta"    "sigma"   "pred"    "log_lik" "lp__"
```

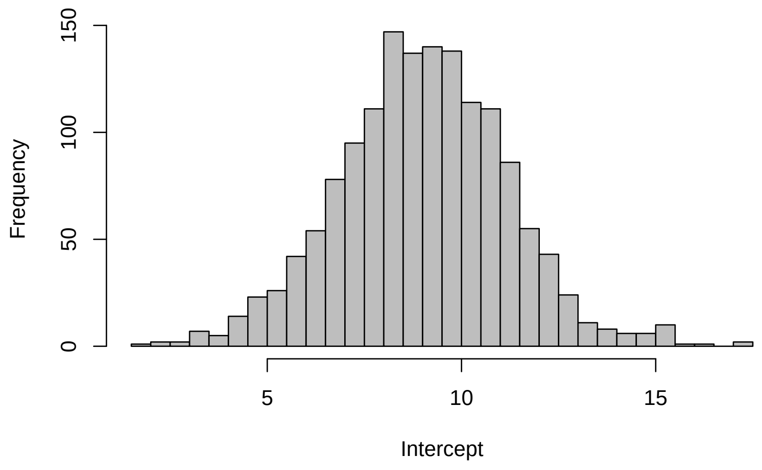
`extract()` will return the draws from the posterior for your parameters and any derived variables specified in your stan code. In this case, our model is

$$y_t = \beta \times 1 + e_t, e_t \sim N(0, \sigma)$$

so our estimated parameters are β and σ. Our stan code computed the derived variables: predicted y_t which is $\hat{y}_t = \beta \times 1$ and the log-likelihood. lp___ is the log posterior which is automatically returned.

We can then make basic plots or summaries of each of these parameters,

```r
hist(pars$beta, 40, col = "grey", xlab = "Intercept", main = "")
```



```r
quantile(pars$beta, c(0.025, 0.5, 0.975))
```

```
     2.5%       50%     97.5% 
 4.620617  9.016637 13.326209 
```

One of the other useful things we can do is look at the predicted values of our model ($\hat{y}_t = \beta \times 1$) and overlay the data. The predicted values are *pars$pred*.

```r
plot(apply(pars$pred, 2, mean), main = "Predicted values", lwd = 2,
    ylab = "Temp", ylim = c(min(pars$pred), max(pars$pred)),
    type = "l")
lines(apply(pars$pred, 2, quantile, 0.025))
lines(apply(pars$pred, 2, quantile, 0.975))
points(Temp, col = "red")
```

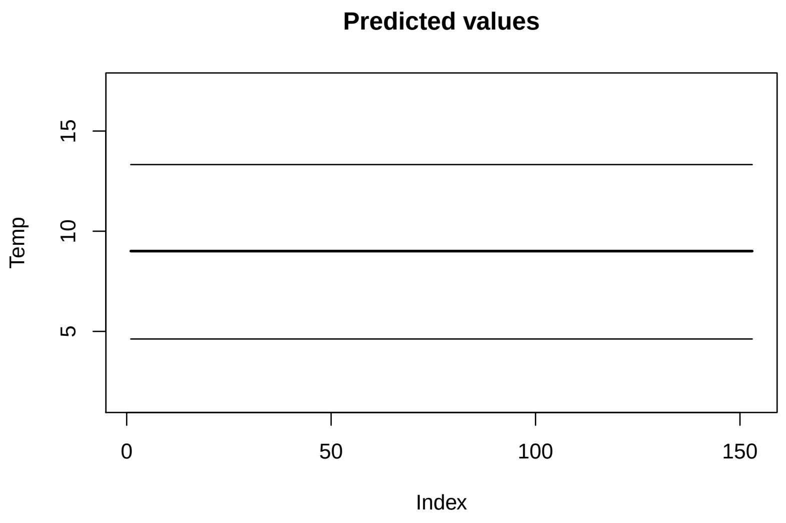


Figure 13.1: Data and predicted values for the linear regression model.

13.1.1 Burn-in and thinning

To illustrate the effects of the burn-in/warmup period and thinning, we can re-run the above model, but for just 1 MCMC chain (the default is 3).

```
lm_intercept <- atsar::fit_stan(y = Temp, x = rep(1, length(Temp)),
    model_name = "regression", mcmc_list = list(n_mcmc = 1000,
        n_burn = 1, n_chain = 1, n_thin = 1))
```

```
Warning: There were 999 divergent transitions after warmup. See
http://mc-stan.org/misc/warnings.html#divergent-transitions-after-warmup
to find out why this is a problem and how to eliminate them.
```

```
Warning: Examine the pairs() plot to diagnose sampling problems
```

Here is a plot of the time series of `beta` with one chain and no burn-in. Based on visual inspection, when does the chain converge?

```r
pars <- rstan::extract(lm_intercept)
plot(pars$beta)
```

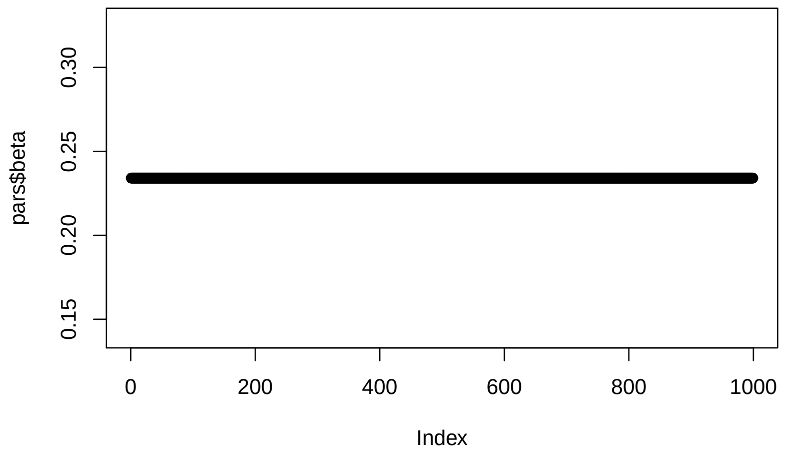


Figure 13.2: A time series of our posterior draws using one chain and no burn-in.

13.2 Linear regression with correlated errors

In our first model, the errors were independent in time. We're going to modify this to model autocorrelated errors. Autocorrelated errors are widely used in ecology and other fields – for a greater discussion, see Morris and Doak (2002) Quantitative Conservation Biology. To make the errors autocorrelated, we start by defining the error in the first time step, $e_1 = y_1 - \beta$. The expectation of Y_t in each time step is then written as

$$E[Y_t] = \beta + \phi e_{t-1}$$

In addition to affecting the expectation, the correlation parameter ϕ also affects the variance of the errors, so that

$$\sigma^2 = \psi^2 \left(1 - \phi^2\right)$$

Like in our first model, we assume that the data follows a normal likelihood (or equivalently that the residuals are normally distributed), $y_t = E[Y_t] + e_t$, or $Y_t \sim N(E[Y_t], \sigma)$. Thus, it is possible to express the subsequent deviations as $e_t = y_t - E[Y_t]$, or equivalently as $e_t = y_t - \beta - \phi e_{t-1}$.

We can fit this regression with autocorrelated errors by changing the model name to 'regression_cor'

```r
lm_intercept_cor <- atsar::fit_stan(y = Temp, x = rep(1, length(Temp)),
    model_name = "regression_cor", mcmc_list = list(n_mcmc = 1000,
        n_burn = 1, n_chain = 1, n_thin = 1))
```

13.3 Random walk model

All of the previous three models can be interpreted as observation error models. Switching gears, we can alternatively model error in the state of nature, creating process error models. A simple process error model that many of you may have seen before is the random walk model. In this model, the assumption is that the true state of nature (or latent states) are measured perfectly. Thus, all uncertainty is originating from process variation (for ecological problems, this is often interpreted as environmental variation). For this simple model, we'll assume that our process of interest (in this case, daily wind speed) exhibits no daily trend, but behaves as a random walk.

$$y_t = y_{t-1} + e_t$$

And the $e_t \sim N(0, \sigma)$. Remember back to the autocorrelated model (or MA(1) models) that we assumed that the errors e_t followed a random walk. In contrast, this model assumes that the errors are independent, but that the state of nature follows a random walk. Note also that this model as written doesn't include a drift term (this can be turned on / off using the `est_drift` argument).

We can fit the random walk model using argument `model_name = 'rw'` passed to the `fit_stan()` function.

```
rw <- atsar::fit_stan(y = Temp, est_drift = FALSE, model_name = "rw")
```

13.4 Autoregressive models

A variation of the random walk model described previously is the autoregressive time series model of order 1, AR(1). This model is essentially the same as the random walk model but it introduces an estimated coefficient, which we will call ϕ. The parameter ϕ controls the degree to which the random walk reverts to the mean – when $\phi = 1$, the model is identical to the random walk, but at smaller values, the model will revert back to the mean (which in this case is zero). Also, ϕ can take on negative values, which we'll discuss more in future lectures. The math to describe the AR(1) model is:

$$y_t = \phi y_{t-1} + e_t$$

.

The `fit_stan()` function can fit higher order AR models, but for now we just want to fit an AR(1) model and make a histogram of phi.

```
ar1 <- atsar::fit_stan(y = Temp, x = matrix(1, nrow = length(Temp),
    ncol = 1), model_name = "ar", est_drift = FALSE, P = 1)
```

13.5 Univariate state-space models

At this point, we've fit models with observation or process error, but we haven't tried to estimate both simultaneously. We will do so here, and introduce some new notation to describe the process model and observation model. We use the notation x_t to denote the latent state or state of nature (which is unobserved) at time t and y_t to denote the observed data. For introductory purposes, we'll make the process model autoregressive (similar to our AR(1) model),

$$x_t = \phi x_{t-1} + e_t, e_t \sim N(0, q)$$

For the process model, there are a number of ways to parameterize the first 'state', and we'll talk about this more in the class, but for the sake of this model, we'll place a vague weakly informative prior on x_0, $x_0 \sim N(0, 10)$.Second, we need to construct an observation model linking the estimate unseen states of nature x_t to the data y_t. For simplicitly, we'll assume that the observation errors are indepdendent and identically distributed, with no observation component. Mathematically, this model is

$$Y_t \sim N(x_t, r)$$

In the two above models, we'll refer to q as the standard deviation of the process variance and r as the standard deviation of the observation error variance

We can fit the state-space AR(1) and random walk models using the `fit_stan()` function:

```r
ss_ar <- atsar::fit_stan(y = Temp, est_drift = FALSE, model_name = "ss_ar")
ss_rw <- atsar::fit_stan(y = Temp, est_drift = FALSE, model_name = "ss_rw")
```

13.6 Dynamic factor analysis

First load the plankton dataset from the **MARSS** package.

```r
library(MARSS)
data(lakeWAplankton, package = "MARSS")
# we want lakeWAplanktonTrans, which has been transformed
# so the 0s are replaced with NAs and the data z-scored
dat <- lakeWAplanktonTrans
# use only the 10 years from 1980-1989
plankdat <- dat[dat[, "Year"] >= 1980 & dat[, "Year"] < 1990,
    ]
# create vector of phytoplankton group names
phytoplankton <- c("Cryptomonas", "Diatoms", "Greens", "Unicells",
    "Other.algae")
# get only the phytoplankton
dat.spp.1980 <- t(plankdat[, phytoplankton])
```

```
# z-score the data since we subsetted time
dat.spp.1980 <- MARSS::zscore(dat.spp.1980)
# check our z-score
apply(dat.spp.1980, 1, mean, na.rm = TRUE)
```

```
  Cryptomonas       Diatoms        Greens      Unicells   Other.algae
 4.740855e-17 -5.592676e-18 -4.486354e-19 -2.699663e-18  6.517410e-18
```

```
apply(dat.spp.1980, 1, var, na.rm = TRUE)
```

```
Cryptomonas     Diatoms      Greens    Unicells Other.algae
          1           1           1           1           1
```

Plot the data.

```
# make into ts since easier to plot
dat.ts <- ts(t(dat.spp.1980), frequency = 12, start = c(1980,
    1))
par(mfrow = c(3, 2), mar = c(2, 2, 2, 2))
for (i in 1:5) {
    plot(dat.ts[, i], type = "b", main = colnames(dat.ts)[i],
        col = "blue", pch = 16)
}
```

Run a 3 trend model on these data.

```
mod_3 <- bayesdfa::fit_dfa(y = dat.spp.1980, num_trends = 3,
    chains = 1, iter = 1000)
```

Rotate the estimated trends and look at what it produces.

```
rot <- bayesdfa::rotate_trends(mod_3)
names(rot)
```

```
[1] "Z_rot"          "trends"         "Z_rot_mean"     "Z_rot_median"
[5] "trends_mean"    "trends_median"  "trends_lower"   "trends_upper"
```

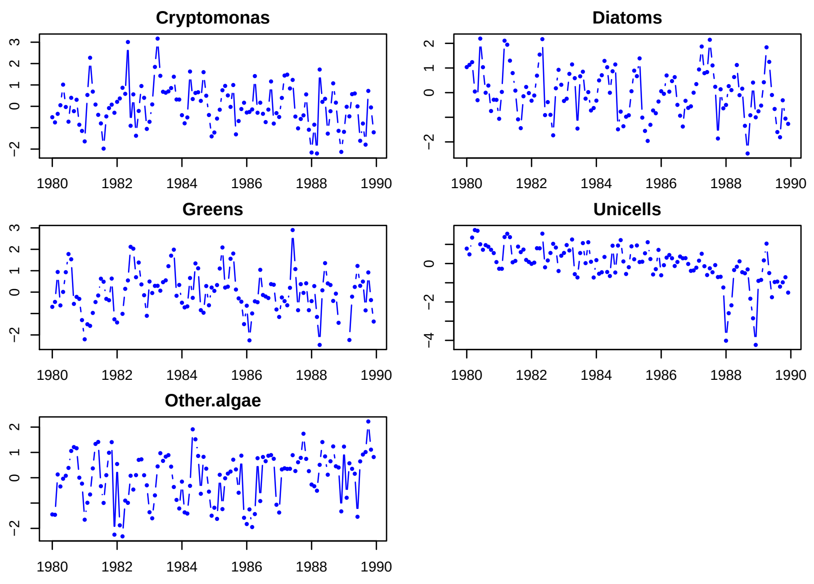


Figure 13.3: Phytoplankton data.

Plot the estimate of the trends.

```
matplot(t(rot$trends_mean), type = "l", lwd = 2, ylab = "mean trend")
```

13.6.1 Using leave one out cross-validation to select models

We will fit multiple DFA with different numbers of trends and use leave one out (LOO) cross-validation to choose the best model.

```
mod_1 <- bayesdfa::fit_dfa(y = dat.spp.1980, num_trends = 1,
    iter = 1000, chains = 1)
```

```
Warning: Bulk Effective Samples Size (ESS) is too low, indicating posterior means and
Running the chains for more iterations may help. See
http://mc-stan.org/misc/warnings.html#bulk-ess
```

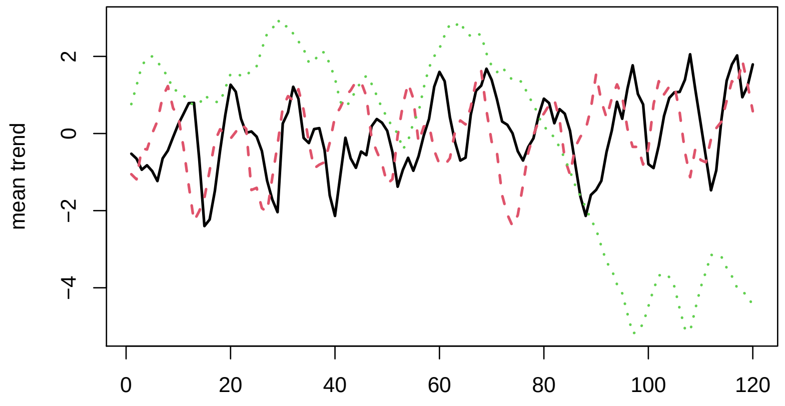


Figure 13.4: Trends.

```
mod_2 <- bayesdfa::fit_dfa(y = dat.spp.1980, num_trends = 2,
    iter = 1000, chains = 1)
```

```
Warning: Bulk Effective Samples Size (ESS) is too low, indicating posterior me
Running the chains for more iterations may help. See
http://mc-stan.org/misc/warnings.html#bulk-ess
```

```
mod_3 <- bayesdfa::fit_dfa(y = dat.spp.1980, num_trends = 3,
    iter = 1000, chains = 1)
```

```
Warning: Bulk Effective Samples Size (ESS) is too low, indicating posterior me
Running the chains for more iterations may help. See
http://mc-stan.org/misc/warnings.html#bulk-ess
```

```
mod_4 <- bayesdfa::fit_dfa(y = dat.spp.1980, num_trends = 4,
    iter = 1000, chains = 1)
```

```
Warning: Bulk Effective Samples Size (ESS) is too low, indicating posterior means and
Running the chains for more iterations may help. See
http://mc-stan.org/misc/warnings.html#bulk-ess

Warning: Tail Effective Samples Size (ESS) is too low, indicating posterior variances
Running the chains for more iterations may help. See
http://mc-stan.org/misc/warnings.html#tail-ess
```

```
# mod_5 = bayesdfa::fit_dfa(y = dat.spp.1980, num_trends=5)
```

We will compute the Leave One Out Information Criterion (LOOIC) using the **loo** package. Like AIC, lower is better.

```
loo(mod_1)$estimates["looic", "Estimate"]
```

```
[1] 1613.758
```

Table of the LOOIC values:

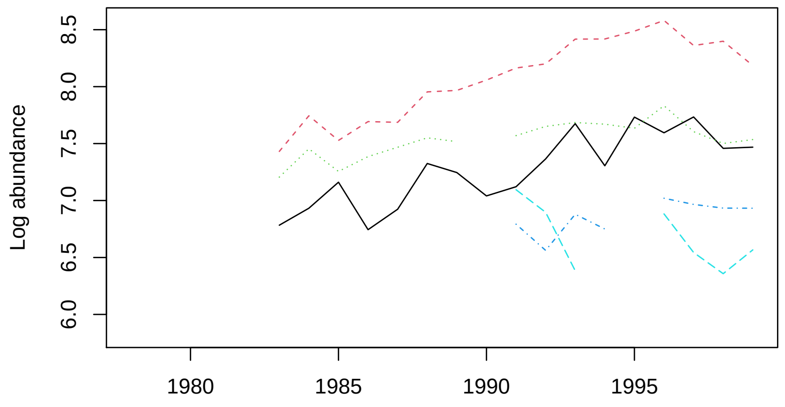
```
looics <- c(loo(mod_1)$estimates["looic", "Estimate"], loo(mod_2)$estimates["looic",
    "Estimate"], loo(mod_3)$estimates["looic", "Estimate"], loo(mod_4)$estimates["loo
    "Estimate"])
looic.table <- data.frame(trends = 1:4, LOOIC = looics)
looic.table
```

```
  trends    LOOIC
1      1 1613.758
2      2 1540.511
3      3 1477.859
4      4 1457.230
```

13.7 Uncertainty intervals on states

We will look at the effect of missing data on the uncertainty intervals on estimates states using a DFA on the harbor seal dataset.

```
data(harborSealWA, package = "MARSS")
# the first column is year
matplot(harborSealWA[, 1], harborSealWA[, -1], type = "l", ylab = "Log abundan
    xlab = "")
```



Assume they are all observing a single trend.

```
seal.mod <- bayesdfa::fit_dfa(y = t(harborSealWA[, -1]), num_trends = 1,
    chains = 1, iter = 1000)
```

```
pars <- rstan::extract(seal.mod$model)
```

```
pred_mean <- c(apply(pars$x, c(2, 3), mean))
pred_lo <- c(apply(pars$x, c(2, 3), quantile, 0.025))
pred_hi <- c(apply(pars$x, c(2, 3), quantile, 0.975))

plot(pred_mean, type = "l", lwd = 3, ylim = range(c(pred_mean,
    pred_lo, pred_hi)), main = "Trend")
lines(pred_lo)
lines(pred_hi)
```

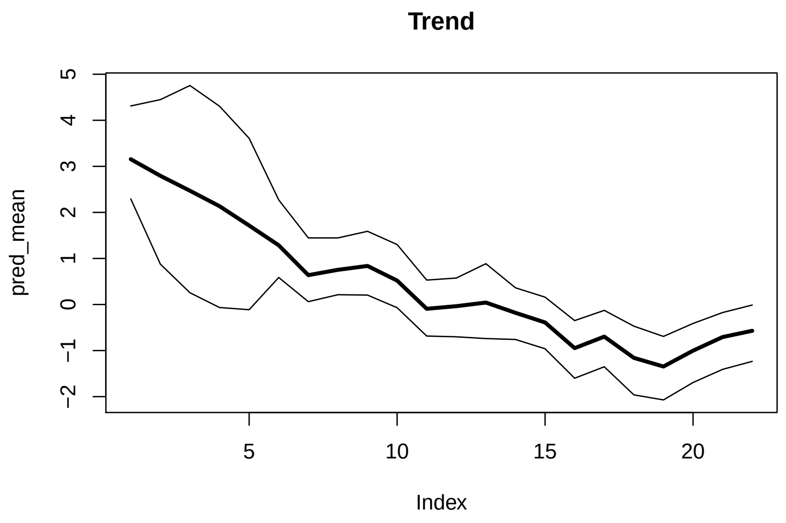


Figure 13.5: Estimated states and 95 percent credible intervals.

13.8 NEON EFI Aquatics Challenge

The data for the aquatics challenge comes from a NEON site at Lake Barco (Florida). More about the data and challenge is here and the Github repository for getting all the necessary data is here.

We pulled in the data with the code from the challenge and saved the data for Lake Barco in the atsalibrary package. See `?neon_barc` for more details.

```
data(neon_barc, package = "atsalibrary")
```

Familiarize yourself with the oxygen and temperature data from Lake Barco by looking at `neon_barc`.

Question: what sort of models do you think would be good candidates for forecasting?

13.9 NEON EFI Aquatics Challenge

Before modeling temperature and oxygen jointly, we'll start working with just the oxygen data alone.

We'll just with just a AR(p) state space model. The Lake Barco data has associated standard errors with the oxygen data, so we'll use that as a known observation error (rather than estimating that variance parameter, in our previous work).

A script that describes the model is called `model_01.stan`. You can download this file here. This should be familiar, following from the `atsar` code with a couple modifications. First, Stan doesn't like NAs being passed in as data, and we've had to do some indexing to avoid that. Second, notice that the model is flexible and can be used to fit models with any numbers of lags.

To get the data prepped for Stan, we need to do a few things. First, specify the lag and forecast horizon

```r
data <- neon_barc
data$indx <- seq(1, nrow(data))
n_forecast <- 7
n_lag <- 1
```

Next, we'll drop observations with missing oxygen. If you wanted to do some validation, we could also split the data into a training and test set.

```r
# As a first model, we'll just work with modeling oxygen
o2_dat <- dplyr::filter(data, !is.na(oxygen))

# split the test and training data
last_obs <- max(data$indx) - n_forecast
o2_train <- dplyr::filter(o2_dat, indx <= last_obs)
test <- dplyr::filter(data, indx > last_obs)

o2_x <- o2_train$indx
o2_y <- o2_train$oxygen
o2_sd <- o2_train$oxygen_sd
n_o2 <- nrow(o2_train)
```

Remember that the Stan data needs to be in a list,

```
stan_data <- list(n = last_obs, n_o2 = n_o2, n_lag = n_lag, n_forecast = n_forecast,
    o2_x = o2_x, o2_y = o2_y, o2_sd = o2_sd)
```

Finally we can compile the model. If we wanted to do fully Bayesian estimates, we could do that with

```
fit <- stan(file = "model_01.stan", data = stan_data)
```

Try fitting the Bayesian model with a short chain length (maybe 1000 iterations) and 1 MCMC chain.

But because we're interested in doing this quickly, and running the model a bunch of times, we'll try Stan's optimizing function for MAP estimation.

```
m <- stan_model(file = "model_01.stan")
o2_model <- rstan::optimizing(m, data = stan_data, hessian = TRUE)
```

Let's extract predictions from the fitted object,

```
data$pred <- o2_model$par[grep("pred", names(o2_model$par))]
ggplot(data, aes(date, pred)) + geom_line() + geom_point(aes(date,
    oxygen), col = "red", alpha = 0.5)
```

Question how do we evaluate predictions? We'll talk about that more next week, but for today we can think about it as RMSE(observations,predictions)

Question: modify the code above (just on the R side, not Stan) to fit a lag-2 or lag-3 model. Are the predictions similar?

13.10 NEON EFI Aquatics Challenge

In generating forecasts, it's often a good idea to use multiple training / test sets. One question with this challenge is what's the best lag (or order of AR model) for generating forecasts? Starting with lag-1, we can evaluate the effects of different lags by starting at some point in the dataset (e.g. using 50% of the observations), and generate and evaluate forecasts. We then iterate, adding a day of data, generating and evaluating forecasts, and repeating through the rest of the data. [An alternative approach would be to adopt a moving window, where each prediction would be based on the same number of historical data points].

I've generalized the code above into a function that takes the initial Lake Barco dataset, and

```r
create_stan_data <- function(data, last_obs, n_forecast, n_lag) {
    o2_test <- dplyr::filter(data, indx %in% seq(last_obs + 1,
        (last_obs + n_forecast)))

    o2_train <- dplyr::filter(data, indx <= last_obs, !is.na(oxygen))
    o2_x <- o2_train$indx
    o2_y <- o2_train$oxygen
    o2_sd <- o2_train$oxygen_sd
    n_o2 <- nrow(o2_train)

    stan_data <- list(n = last_obs, n_o2 = n_o2, n_lag = n_lag,
        n_forecast = n_forecast, o2_x = o2_x, o2_y = o2_y, o2_sd = o2_sd)

    return(list(train = o2_train, stan_data = stan_data, test = o2_test))
}
```

Now we can try iterating over sets of data with a lag 1 model

```r
n_forecast <- 7
n_lag <- 1
rmse <- NA
for (i in 500:(nrow(data) - n_lag)) {
    dat_list <- create_stan_data(data, last_obs = i, n_forecast = n_forecast,
        n_lag = n_lag)

    # fit the model. opimizing can be sensitive to starting
    # values, so let's try
    best_map <- -1e+100
    for (j in 1:10) {
        test_fit <- rstan::optimizing(m, data = dat_list$stan_data)
        if (test_fit$value > best_map) {
            fit <- test_fit
            best_map <- test_fit$value
        }
    }
    if (fit$return_code == 0) {
        # extract forecasts
        pred <- fit$par[grep("forecast", names(fit$par))]

        # evaluate predictions
        rmse[i] <- sqrt(mean((dat_list$test$oxygen - pred)^2,
            na.rm = T))
    }
}
```

13.11 NEON EFI Aquatics Challenge

So far, these models have only included oxygen. We can easily bring in the temperature data, which has it's own standard error associated with it. We can set up a null model modeling temperature and oxygen independently, with different lags allowed

First we have to modify our data function to also generate temperature data.

```
create_stan_data <- function(data, last_obs, n_forecast, n_lag_o2,
    n_lag_temp) {
    # create test data
    o2_test <- dplyr::filter(data, indx %in% seq(last_obs + 1,
        (last_obs + n_forecast)))
    temp_test <- dplyr::filter(data, indx %in% seq(last_obs +
        1, (last_obs + n_forecast)))

    o2_train <- dplyr::filter(data, indx <= last_obs, !is.na(oxygen))
    o2_x <- o2_train$indx
    o2_y <- o2_train$oxygen
    o2_sd <- o2_train$oxygen_sd
    n_o2 <- nrow(o2_train)

    temp_train <- dplyr::filter(data, indx <= last_obs, !is.na(temperature))
    temp_x <- temp_train$indx
    temp_y <- temp_train$temperature
    temp_sd <- temp_train$temperature_sd
    n_temp <- nrow(temp_train)

    stan_data <- list(n = last_obs, n_lag_o2 = n_lag_o2, n_lag_temp = n_lag_te
        n_forecast = n_forecast, n_o2 = n_o2, o2_x = o2_x, o2_y = o2_y,
        o2_sd = o2_sd, n_temp = n_temp, temp_x = temp_x, temp_y = temp_y,
        temp_sd = temp_sd)

    return(list(o2_train = o2_train, temp_train = temp_train,
        stan_data = stan_data, o2_test = o2_test, temp_test = temp_test))
}
```

```
m <- stan_model(file = "model_02.stan")
```

where `model_02.stan` is the Stan model file that you can download here.

```
# Now we can try iterating over sets of data with a lag 1
# model
n_forecast <- 7
n_lag <- 1
rmse <- NA
for (i in 500:(nrow(data) - n_lag)) {
    dat_list <- create_stan_data(data, last_obs = i, n_forecast = n_forecast,
        n_lag_o2 = n_lag, n_lag_temp = n_lag)

    # fit the model. opimizing can be sensitive to starting
    # values, so let's try
    best_map <- -1e+100
    for (j in 1:10) {
        test_fit <- rstan::optimizing(m, data = dat_list$stan_data)
        if (test_fit$value > best_map) {
            fit <- test_fit
            best_map <- test_fit$value
        }
    }

    # extract forecasts
    o2_pred <- fit$par[grep("o2_forecast", names(fit$par))]
    temp_pred <- fit$par[grep("temp_forecast", names(fit$par))]

    pred <- c(o2_pred, temp_pred)
    obs <- c(dat_list$o2_test$oxygen, dat_list$temp_test$temperature)
    # evaluate predictions
    rmse[i] <- sqrt(mean((obs - pred)^2, na.rm = T))
    print(rmse)
}
```

Question: why do the future predictions of either temp or oxygen sometimes fall to 0, and appear to do poorly?

Question: modify the script above to loop over various oxygen and temperature lags. Using a subset of data, say observations 500:800, which combination yields the best predictions?

13.12 Problems

1. By adapting the code in Section 13.1, fit a regression model that includes the intercept and a slope, modeling the effect of Wind. What is the mean wind effect you estimate?

2. Using the results from the linear regression model fit with no burn-in (Section 13.1.1), calculate the ACF of the `beta` time series using `acf()`. Would thinning more be appropriate? How much?

3. Using the fit of the random walk model to the temperature data (Section 13.3), plot the predicted values (states) and 95% CIs.

4. To see the effect of this increased flexibility in estimating the autocorrelation, make a plot of the predictions from the AR(1) model (Section 13.4 and the RW model (13.3).

5. Fit the univariate state-space model (Section 13.5) with and without the autoregressive parameter ϕ and compare the estimated process and observation error variances. Recall that AR(1) without the ϕ parameter is a random walk.

6. Run the examples related to the EFI challenge. Work through the questions associated with each exercise

More examples

Short examples of a variety of multivariate state-space models.

knitr::opts_knit$set(unnamed.chunk.label = "dlm-")knitr :: opts_chunk$set(echo = TRUE, comment=NA, cache=TRUE, tidy.opts=list(width.cutoff=60), tidy=TRUE, fig.align='center', out.width='80%')

Chapter 14

Detecting a signal from noisy sensors

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

14.1 Overview

We have 3 sensors that are tracking some signal. One sensor is good (low error). The other 2 sensors are horrible in different ways. One has high auto-correlated error. The third is basically a random walk and not tracking our signal at all. However, we do not know which ones are bad or if in fact any are bad.

What we do know is that for these sensors an AR-1 error model is a good approximation: $y_t = a + e_t$ where a is our signal and $e_t = be_{t_1} + w_t$. w_t is white noise with some unknown standard deviation and mean 0.

We will create some simulated data with this set-up and estimate the signal.

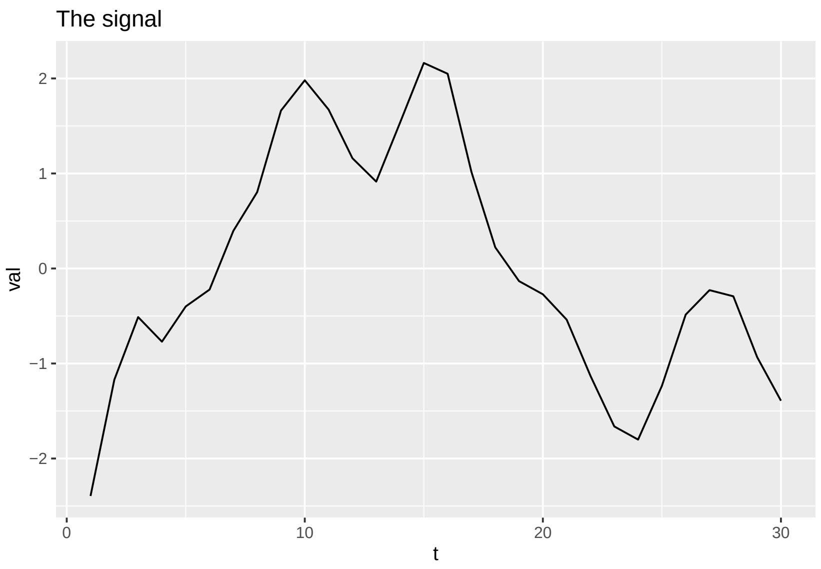
14.2 Prep

```r
library(ggplot2)
library(MARSS)
library(stringr)
set.seed(1234)
```

14.3 Create a signal

I want something fairly smooth but still stochastic. It is not important how you create it. We are not interested in its parameters but only its shape. The idea is that there is some average signal (imagine a mean temperature, say) that is slowly changing. We are trying to pick up that signal and separate it from the noise added by our sensors. Our sensors might not be noisy because they are *bad*. It may be that what we are measuring has a local noisy component (from say local winds or currents or something). That noise might be autoregressive (temporally correlated) because whatever we are measuring is temporally correlated (like temperature).

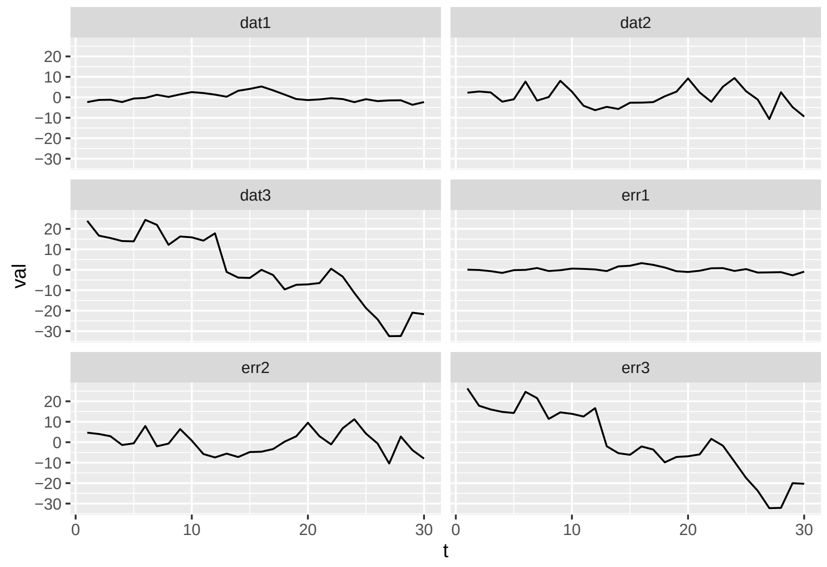
Here I use `arima.sim()` to create a signal and then then smooth it with `filter()`. I set the seed. Change to make a new data set.



14.4 Create data

Plot the data and the error added to the signal by each sensor. The data is the error (on right) plus the signal. The signal is definitely not obvious in the data. The data look mostly like the error, which is autocorrelated so has trends in it unlike white noise error.

```r
p1 <- ggplot(subset(df, name != "signal"), aes(x = t, y = val)) +
    geom_line() + facet_wrap(~name, ncol = 2)
p1
```



14.5 Model Shared stochastic level with AR-1 observation errors

This time I will allow AR-1 errors that are not a random walk, so classic AR-1 errors. This means estimating the diagonals of a B matrix. To do that I will need to get rid of the mean of the data since trying to estimate a B matrix and mean levels is hard (there is a huge ridge in the likelihood and the problem is poorly defined).

$$\begin{bmatrix} a \\ x1 \\ x2 \\ x3 \end{bmatrix}_t = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & b_1 & 0 & 0 \\ 0 & 0 & b_2 & 0 \\ 0 & 0 & 0 & b_3 \end{bmatrix} \begin{bmatrix} a \\ x1 \\ x2 \\ x3 \end{bmatrix}_{t-1} + \begin{bmatrix} e \\ w1 \\ w2 \\ w3 \end{bmatrix}_t, \quad \begin{bmatrix} e \\ w1 \\ w2 \\ w3 \end{bmatrix}_t \sim MVN\left(0, \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & q_1 & 0 & 0 \\ 0 & 0 & q_2 & 0 \\ 0 & 0 & 0 & q_3 \end{bmatrix}\right)$$

Here is the data model BUT the y will be demeaned. Each sensor observes a plus their own independent local AR-1 trend. Notice no v_t. The model error comes through the AR-1 x processes.

$$\begin{bmatrix} y1 \\ y2 \\ y3 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ x1 \\ x2 \\ x3 \end{bmatrix}_t$$

14.6 Fit the model

We specify this one-to-one in R for `MARSS()`:

```r
makemod <- function(n) {
    B <- matrix(list(0), n + 1, n + 1)
    diag(B)[2:(n + 1)] <- paste0("b", 1:n)
    B[1, 1] <- 1
    A <- "zero"
    Z <- cbind(1, diag(1, n))
    Q <- matrix(list(0), n + 1, n + 1)
    Q[1, 1] <- 1
    diag(Q)[2:(n + 1)] <- paste0("q", 1:n)
    R <- "zero"
    U <- "zero"
    x0 <- "zero"
    mod.list <- list(B = B, A = A, Z = Z, Q = Q, R = R, U = U,
        x0 = x0, tinitx = 0)
    return(mod.list)
}
mod.list1 <- makemod(3)
```

Demean the data.

```r
dat2 <- dat - apply(dat, 1, mean) %*% matrix(1, 1, TT)
```

Fit to that

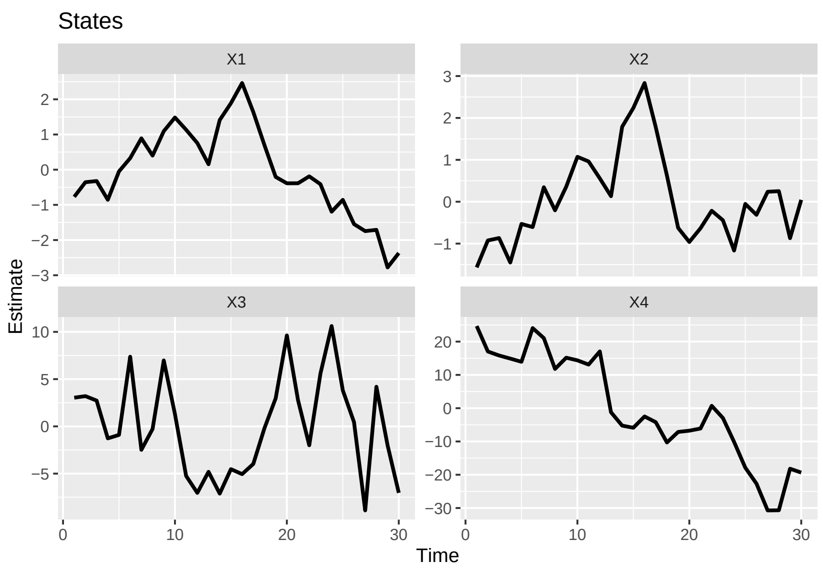
```
fit.mod1 <- MARSS(dat2, model = mod.list1)
```

```
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: -244.5957
AIC: 501.1914   AICc: 502.2035

     Estimate
B.b1    0.790
B.b2    0.406
B.b3    0.905
Q.q1    1.004
Q.q2   23.042
Q.q3   55.892
Initial states (x0) defined at t=0

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

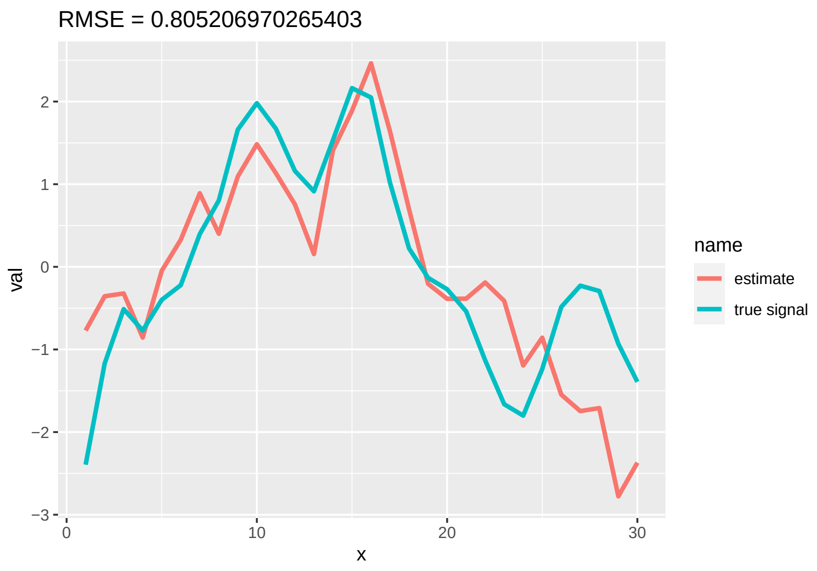
14.7 Show the fits

X1 is the estimate of the signal. The mean has been removed. X2, X3 and X4 are the AR-1 errors for our sensors.

```
require(ggplot2)
autoplot(fit.mod1, plot.type = "xtT", conf.int = FALSE)
```

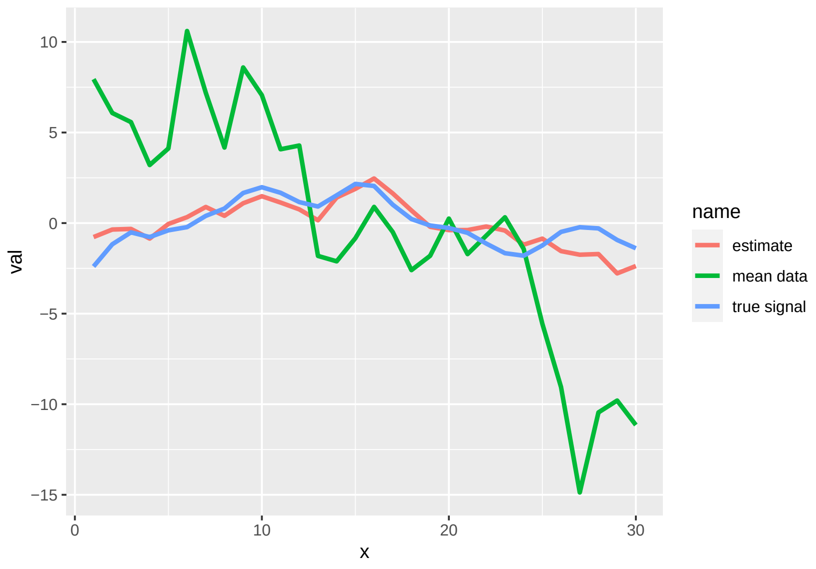


14.8 Estimated common trend from state-space model



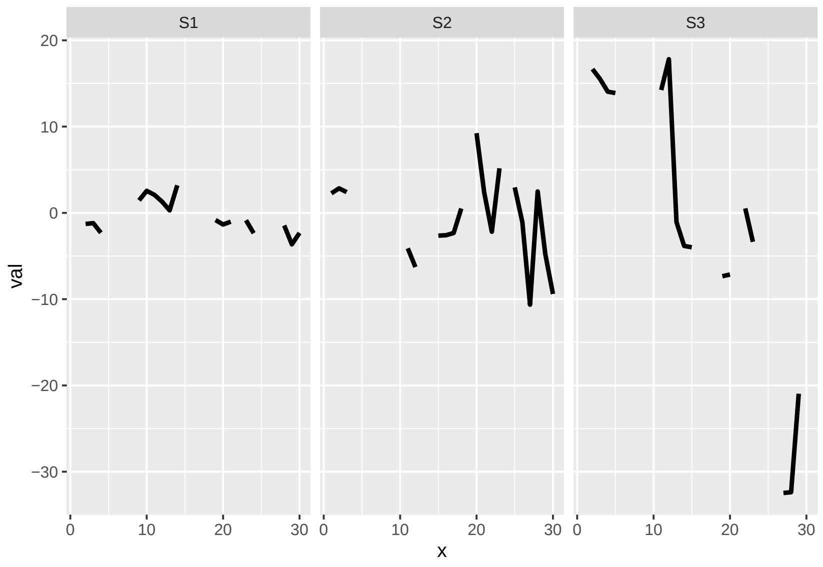
14.9 Compare the common trend to mean of data

You could just take the average of the 3 sensors assuming they were independent with similar error levels. With some of the sensors being really bad, this would not give a good estimate of the signal. The model allowed us to estimate b for each sensor and q (variance) thus allowing us to estimate how much to weight each sensor.



14.10 Missing data

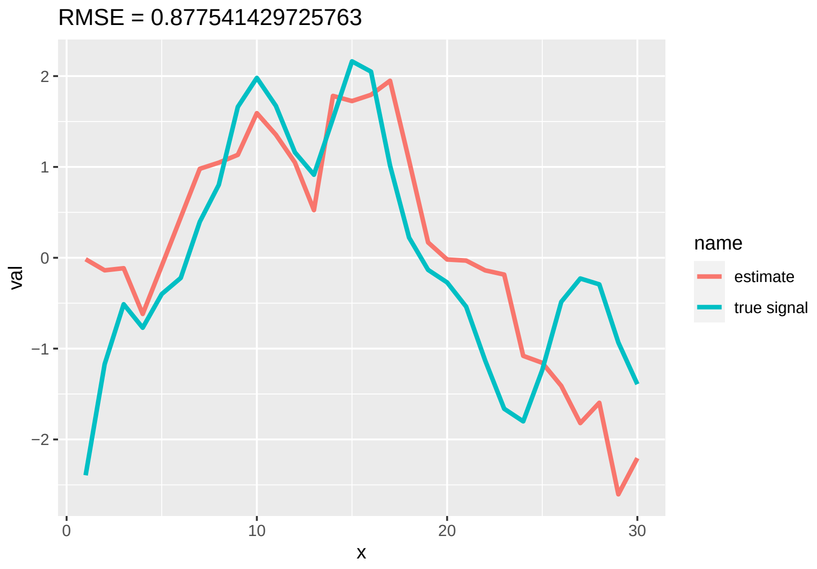
One nice features of this approach is that it is robust to a fair bit of missing data. Here I delete a third of the data. I do this randomly throughout the dataset. The data look pretty hopeless. No signal to be seen.



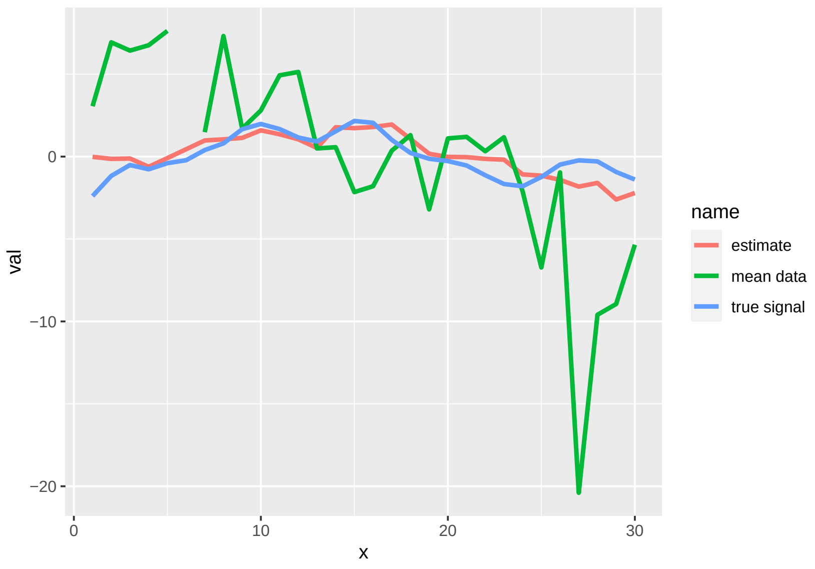
Fit as usual:

```r
fit <- MARSS(dat2.miss, model = mod.list1, silent = TRUE)
```

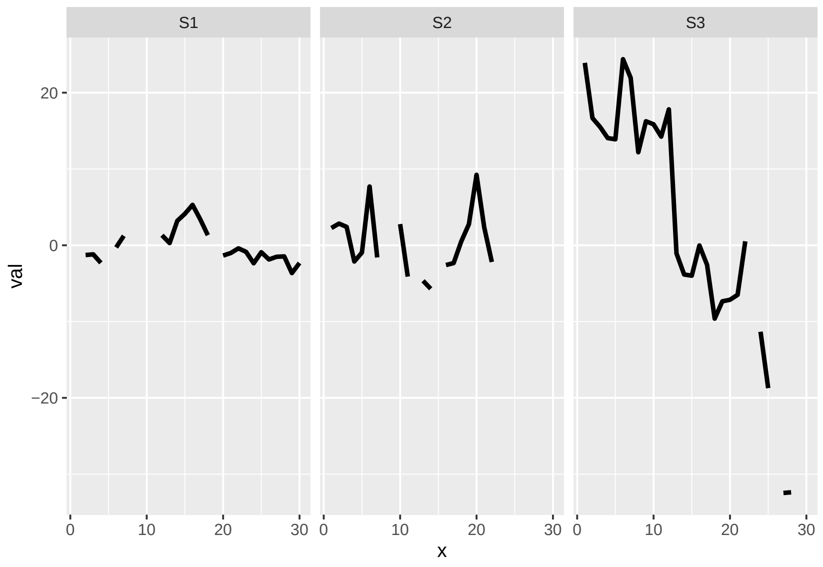
But though we can't see the signal in the data, it is there.



Averaging our sensors doesn't work since there are so many missing values and we will have missing values in our average.

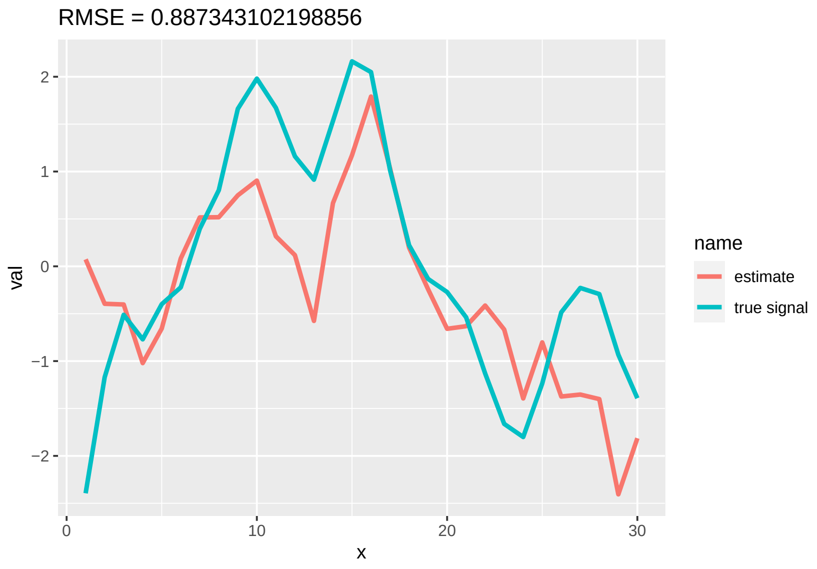


Another type of missing data are strings of missing data. Here I create a data set with random strings of missing values. Again the data look really hopeless and definitely cannot average across the data since we'd be averaging across different data sets.



We can fit as usual and see that it is possible to recover the signal.

```
fit <- MARSS(dat2.miss, model = mod.list1, silent = TRUE)
```



14.11 Correlated noise

In the simulated data, the AR-1 errors were uncorrelated. Each error time series was independent of the others. But we might want to test a model where the errors are correlated. The processes that drive variability in sensors can sometimes be a factor that are common across all our sensors, like say average wind speed or rainfall.

Our AR-1 errors would look like so with covariance c.

$$\begin{bmatrix} e \\ w_1 \\ w_2 \\ w_3 \end{bmatrix}_t, \quad \begin{bmatrix} e \\ w_1 \\ w_2 \\ w_3 \end{bmatrix}_t \sim MVN\left(0, \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & q_1 & c_1 & c_2 \\ 0 & c_1 & q_2 & c_3 \\ 0 & c_2 & c_3 & q_3 \end{bmatrix}\right)$$

To fit this model, we need to create a Q matrix that looks like the above. It's a bit of a hassle.

```
Q <- matrix(list(0), n + 1, n + 1)
Q[1, 1] <- 1
Q2 <- matrix("q", n, n)
diag(Q2) <- paste0("q", 1:n)
Q2[upper.tri(Q2)] <- paste0("c", 1:n)
Q2[lower.tri(Q2)] <- paste0("c", 1:n)
Q[2:(n + 1), 2:(n + 1)] <- Q2
Q
```

```
     [,1] [,2] [,3] [,4]
[1,] 1    0    0    0   
[2,] 0    "q1" "c1" "c2"
[3,] 0    "c1" "q2" "c3"
[4,] 0    "c2" "c3" "q3"
```

Now we can fit as usual using this Q in our model list.

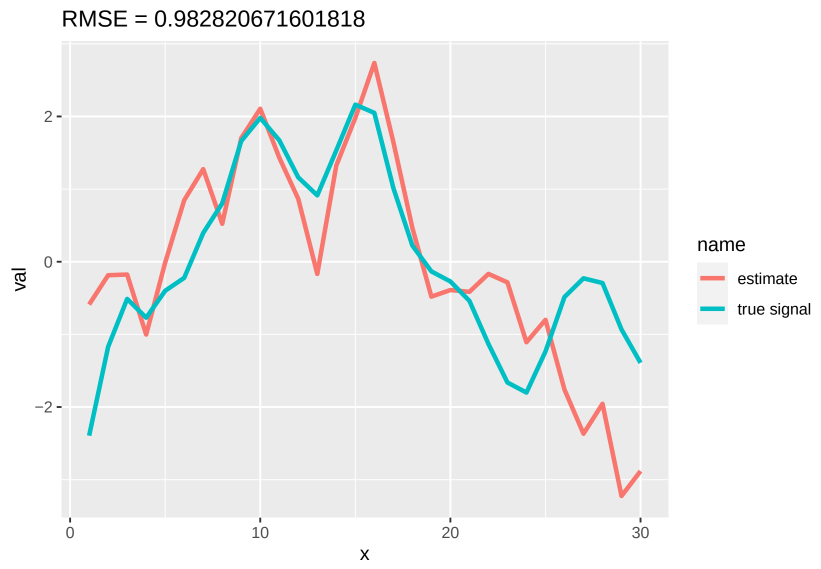
```
mod.list2 <- mod.list1
mod.list2$Q <- Q
fit <- MARSS(dat2, model = mod.list2)
```

```
Success! abstol and log-log tests passed at 127 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 127 iterations.
Log-likelihood: -242.6383
AIC: 503.2765   AICc: 505.5265

     Estimate
B.b1    0.745
B.b2    0.389
B.b3    0.907
Q.q1    0.800
Q.c1   -1.721
Q.c2   -2.130
Q.q2   21.680
Q.c3    4.210
Q.q3   54.143
Initial states (x0) defined at t=0

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

The AIC is larger indicating that this model is not more supported, which is not surprising given that the data are not correlated with each other.

```
c(fit$AIC, fit.mod1$AIC)
```

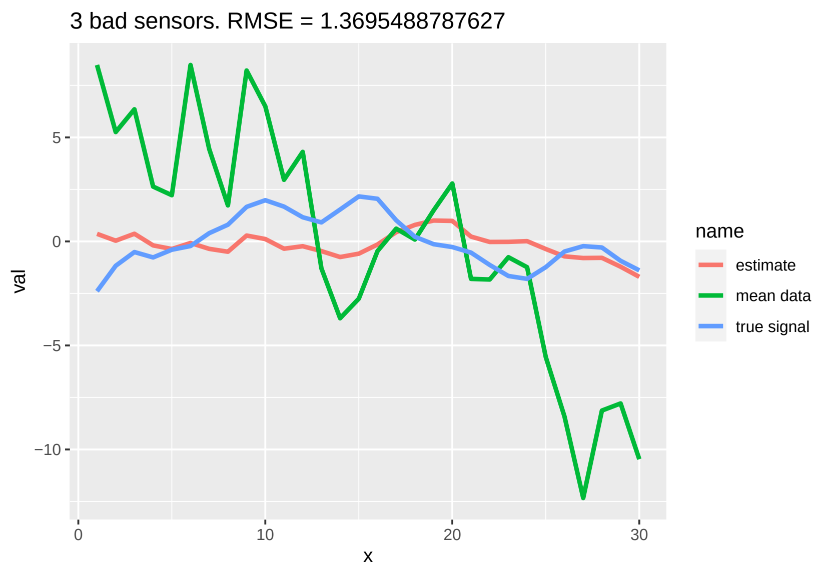
```
[1] 503.2765 501.1914
```



14.12 Discussion

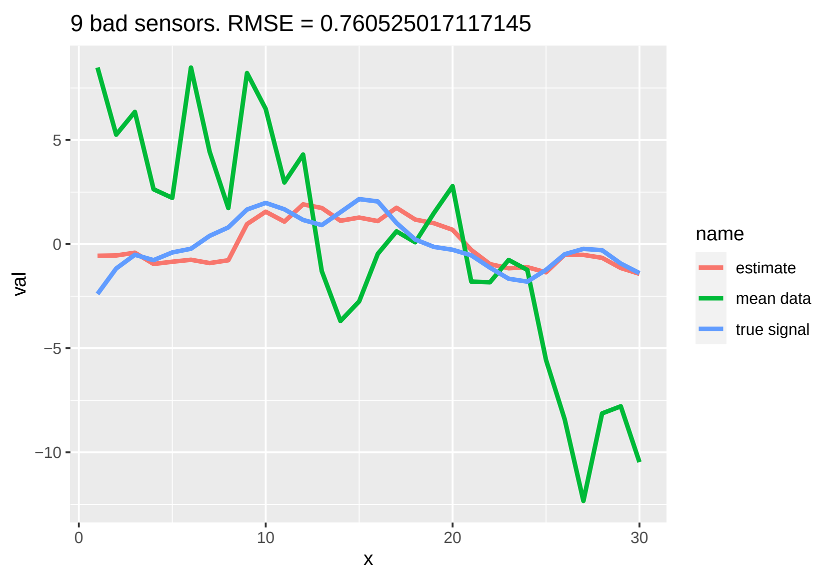
This example worked because I had a sensor that was quite a bit better than the others with a much smaller level of observation error variance (sd=1 versus 28 and 41 for the others). I didn't know which one it was, but I did have at least one good sensor. If I up the observation error variance on the first (good) sensor, then my signal estimate is not so good. The variance of the signal estimate is better than the average, but it is still bad. There is only so much that can be done when the sensor adds so much error.

```r
sd <- sqrt(c(10, 28, 41))
dat[1, ] <- signal + arima.sim(TT, model = list(ar = ar[1]),
    sd = sd[1])
dat2 <- dat - apply(dat, 1, mean) %*% matrix(1, 1, TT)

fit <- MARSS(dat2, model = mod.list1, silent = TRUE)
```



One solution is to have more sensors. They can all be horrible but now that I have more, I can get a better estimate of the signal. In this example I have 12 bad sensors instead of 3. The properties of the sensors are the same as in the example above. I will add the new data to the existing data.

```
set.seed(123)
datm <- dat
for (i in 1:2) {
    tmp <- createdata(n, TT, ar, sd)
    datm <- rbind(datm, tmp$dat)
}
datm2 <- datm - apply(datm, 1, mean) %*% matrix(1, 1, TT)

fit <- MARSS(datm2, model = makemod(dim(datm2)[1]), silent = TRUE)
```



Some more caveats are that I simulated data that was the same as the model that I fit, except the signal. However an AR-1 with b and q (sd) estimated is quite flexible and this will likely work for data that is roughly AR-1. A common exception is very smooth data that you get from sensors that record dense data (like every second). That kind of sensor data may need to be subsampled (every 10 or 20 or 30 data point) to get AR-1 like data.

Lastly I set the seed to 1234 to have an example that looks *ok*. If you comment that out and rerun the code, you'll quickly see that the example I used is not one of the bad ones. It's not unusually good, just not unusually bad.

On the otherhand, I poised a difficult problem with two quite awful sensors. A sensor with a random walk error would be really alarming and hopefully you would not have that type of error. But you might. IT can happen when local conditions are undergoing a random walk with slow reversion to the mean. Many natural systems look like that. If you have that problem, subsampling that *random walk* sensor might be a good idea.

Chapter 15

Modeling changing seasonality

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

15.1 Overview

As discussed in Section 8.6.3 in the covariates chapter, we can model season with sine and cosine covariates.

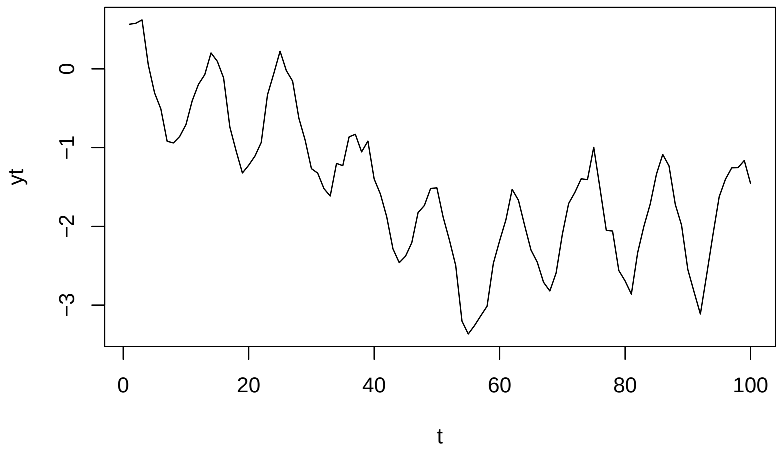
$$y_t = x_t + \beta_1 \sin(2\pi t/p) + \beta_2 \cos(2\pi t/p) + e_t$$

where t is the time step (1 to length of the time series) and p is the frequency of the data (e.g. 12 for monthly data). α_t is the mean level about which the data y_t are fluctuating.

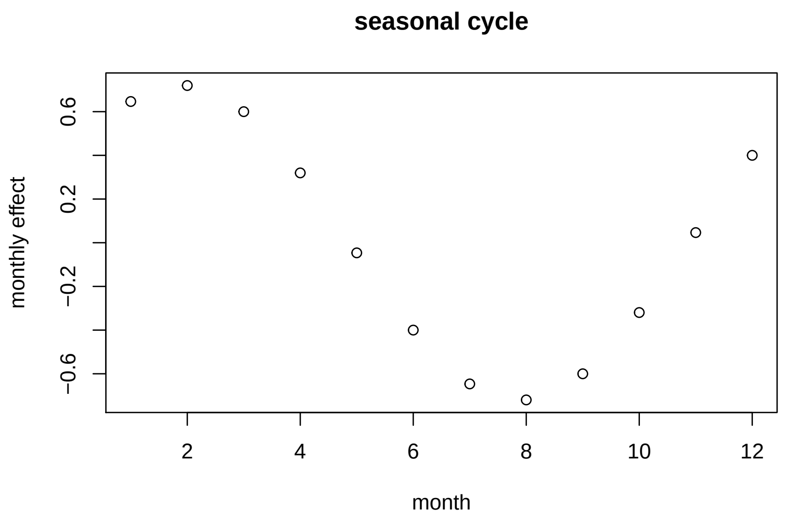
We can simulate data like this as follows:

```r
set.seed(1234)
TT <- 100
q <- 0.1
r <- 0.1
beta1 <- 0.6
beta2 <- 0.4
cov1 <- sin(2 * pi * (1:TT)/12)
```

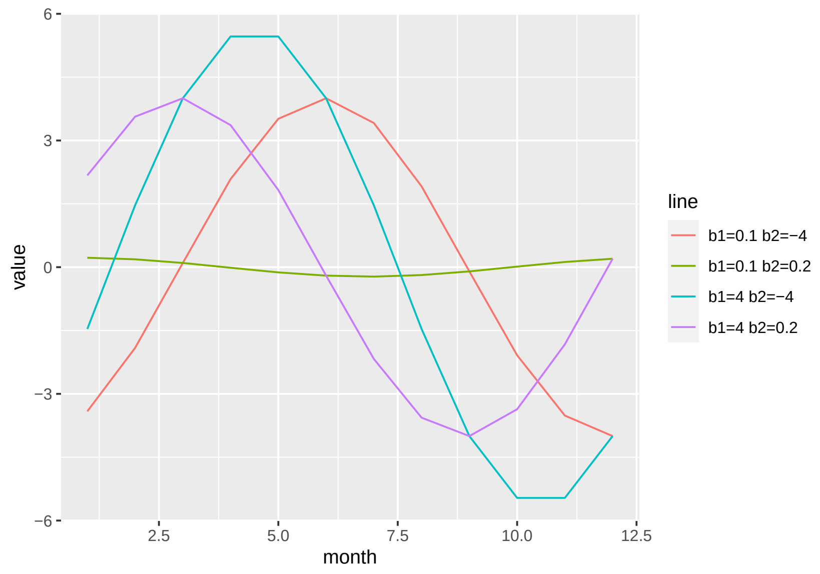
```
cov2 <- cos(2 * pi * (1:TT)/12)
xt <- cumsum(rnorm(TT, 0, q))
yt <- xt + beta1 * cov1 + beta2 * cov2 + rnorm(TT, 0, r)
plot(yt, type = "l", xlab = "t")
```



In this case, the seasonal cycle is constant over time since β_1 and β_2 are fixed (not varying in time).



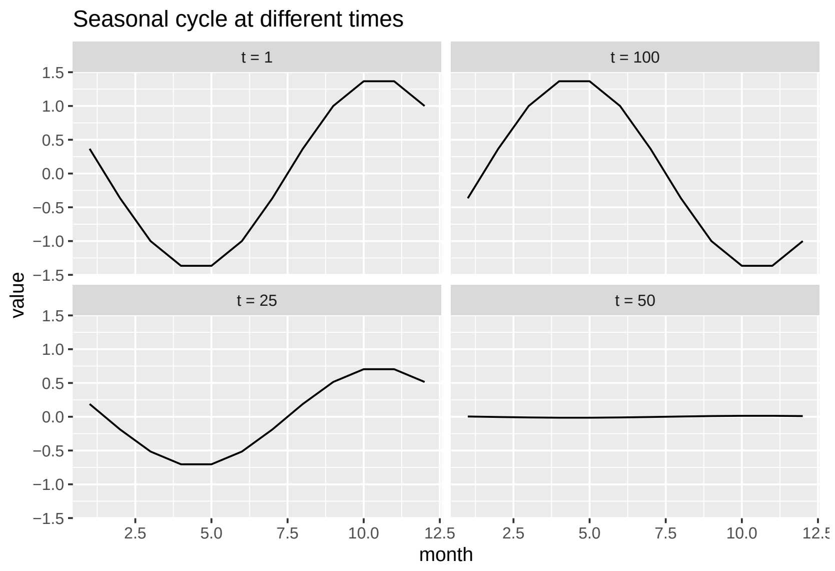
The β determine the shape and amplitude of the seasonal cycle—though in this case we will only have one peak per year.



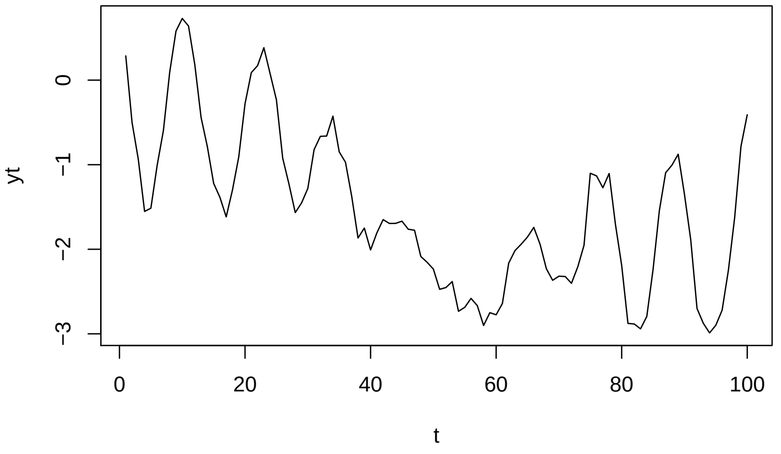
15.2 Time-varying seasonality and amplitude

If β_1 and β_2 vary in time then the seasonal cycle also varies in time. Let's imagine that β_1 varies from -1 to 1 over our 100 time steps while β_2 varies from 1 to -1.

So the seasonal cycle flips between the start and end of our time series.



A time series simulated with that flip looks like so:



15.3 Using DLMs to estimate changing season

Here is a DLM model with the level and season modeled as a random walk.

$$
\begin{bmatrix} x \\ \beta_1 \\ \beta_2 \end{bmatrix}_t = \begin{bmatrix} x \\ \beta_1 \\ \beta_2 \end{bmatrix}_{t-1} + \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}_t
$$

$$
y_t = \begin{bmatrix} 1 & \sin(2\pi t/p) & \cos(2\pi t/p) \end{bmatrix} \begin{bmatrix} x \\ \beta_1 \\ \beta_2 \end{bmatrix}_t + v_t
$$

We can fit the model to the y_t data and estimate the β's and α. We specify this one-to-one in R for `MARSS()`.

`Z` is time-varying and we set this up with an array with the 3rd dimension being time.

```
Z <- array(1, dim = c(1, 3, TT))
Z[1, 2, ] <- sin(2 * pi * (1:TT)/12)
Z[1, 3, ] <- cos(2 * pi * (1:TT)/12)
```

Then we make our model list. We need to set `A` since `MARSS()` doesn't like the default value of `scaling` when `Z` is time-varying.

```
mod.list <- list(U = "zero", Q = "diagonal and unequal", Z = Z,
    A = "zero")
```

When we fit the model we need to give `MARSS()` initial values for `x0`. It cannot come up with default ones for this model. It doesn't really matter what you pick.

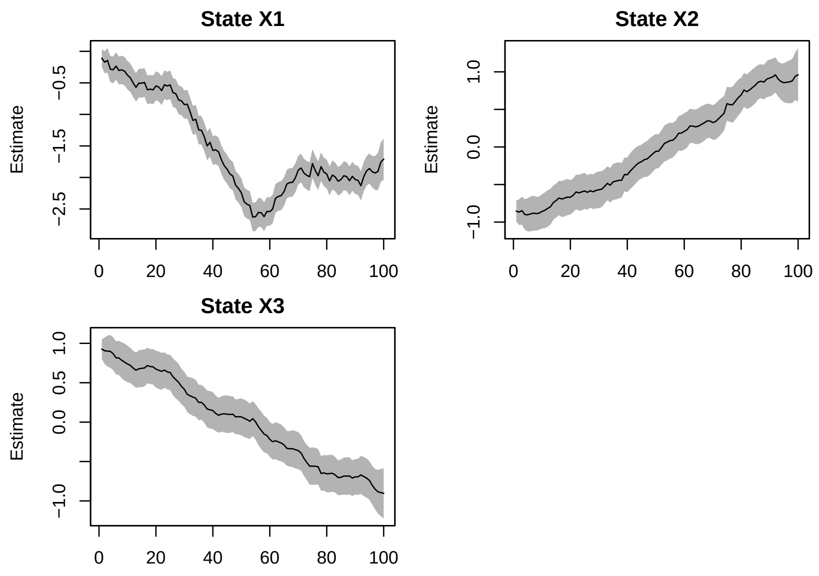
```
require(MARSS)
fit <- MARSS(yt, model = mod.list, inits = list(x0 = matrix(0,
    3, 1)))
```

```
Success! abstol and log-log tests passed at 45 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 45 iterations.
Log-likelihood: 19.0091
AIC: -24.01821   AICc: -22.80082

          Estimate
R.R        0.00469
Q.(X1,X1)  0.01476
Q.(X2,X2)  0.00638
Q.(X3,X3)  0.00580
x0.X1     -0.11002
x0.X2     -0.85340
```

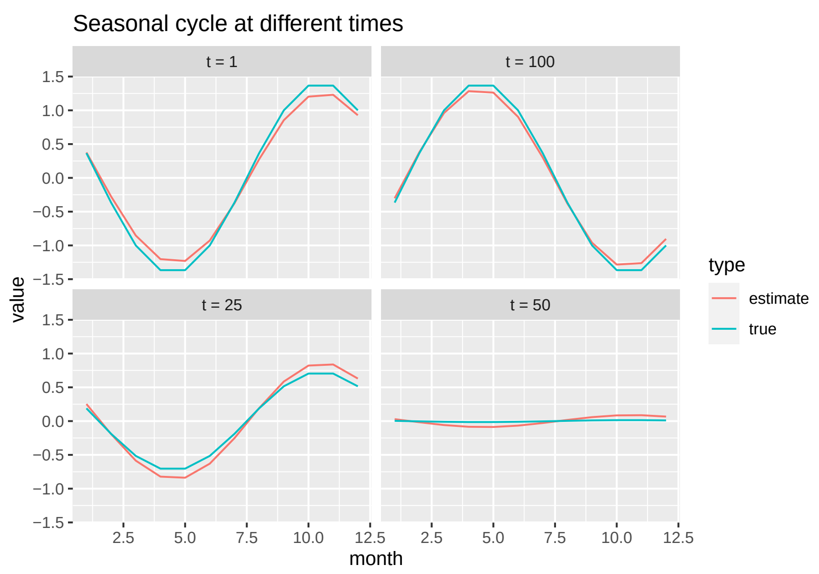
```
x0.X3      0.92787
Initial states (x0) defined at t=0

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

The β_1 estimate is State X2 and β_2 is State X3. The estimates match what we put into the simulated data.

```
plot type = "xtT" Estimated states
```

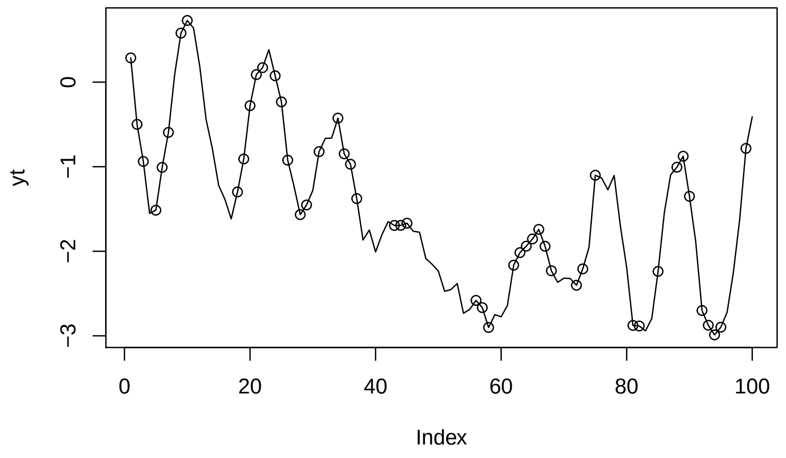


We can compare the estimated cycles to the ones used in the simulation.



We can make this a bit harder by imagining that our data have missing values. Let's imagine that we only observe half the months.

```
yt.miss <- yt
yt.miss[sample(100, 50)] <- NA
plot(yt, type = "l")
points(yt.miss)
```



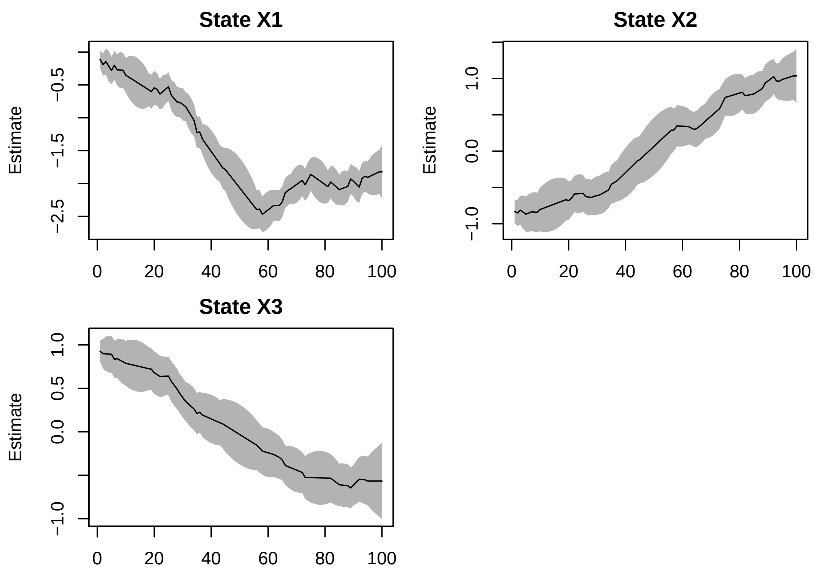
```
require(MARSS)
fit.miss <- MARSS(yt.miss, model = mod.list, inits = list(x0 = matrix(0,
    3, 1)))
```

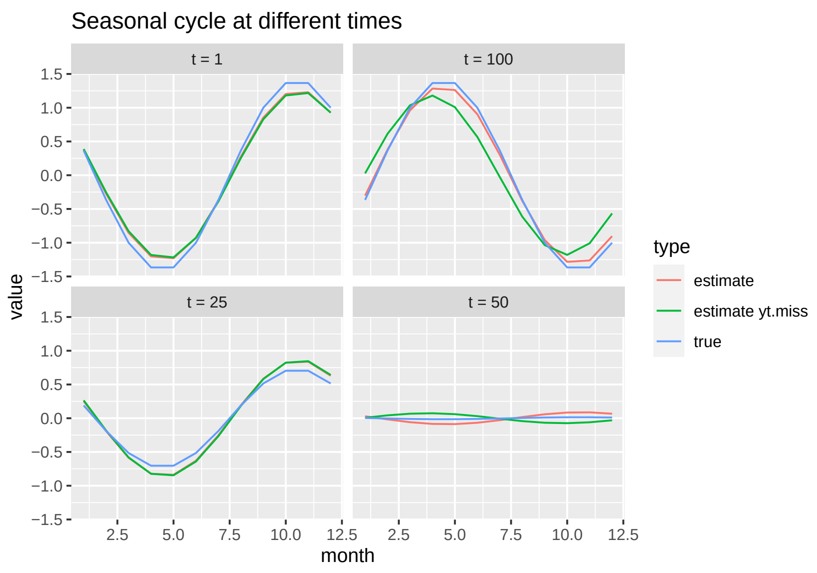
```
Success! abstol and log-log tests passed at 108 iterations.
Alert: conv.test.slope.tol is 0.5.
Test with smaller values (<0.1) to ensure convergence.

MARSS fit is
Estimation method: kem
Convergence test: conv.test.slope.tol = 0.5, abstol = 0.001
Estimation converged in 108 iterations.
Log-likelihood: 0.1948933
AIC: 13.61021   AICc: 16.27688

          Estimate
R.R        0.00243
Q.(X1,X1)  0.01342
Q.(X2,X2)  0.00794
Q.(X3,X3)  0.00580
x0.X1     -0.11497
```

```
x0.X2     -0.82921
x0.X3      0.92696
Initial states (x0) defined at t=0

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

The model still can pick out the changing seasonal cycle.

```
plot type = "xtT" Estimated states
```



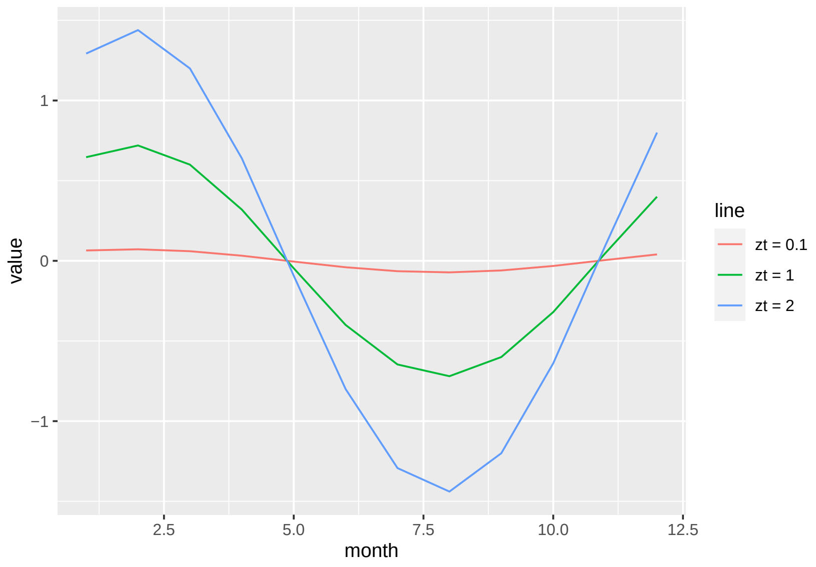


15.4 Time-varying amplitude

Instead of a constant seasonality, we can imagine that it varies in time. The first way it might vary is in the amplitude of the seasonality. So the location of the peak is the same but the difference between the peak and valley changes.

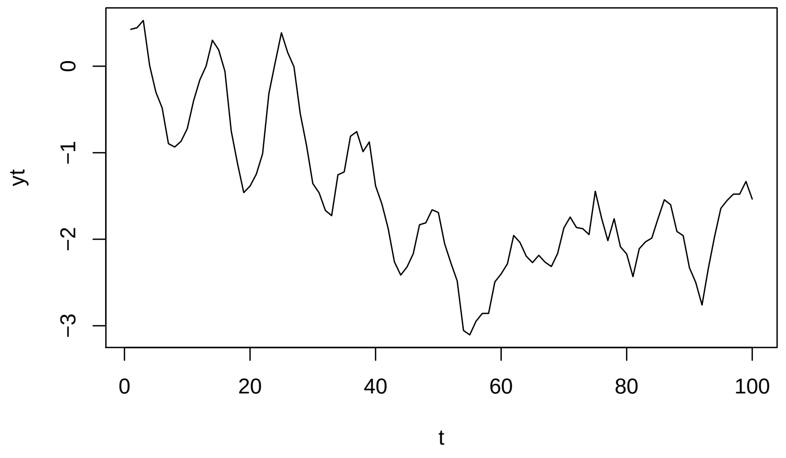
$$z_t\,(\beta_1 \sin(2\pi t/p) + \beta_2 \cos(2\pi t/p))$$

In this case, the β's remain constant but the sum of the sines and cosines is multiplied by a time-varying scaling factor.



Here we simulate some data where z_t is sinusoidal and is largest in the beginning of the time-series. Note we want z_t to stay positive otherwise our peak will become a valley when z_t goes negative.

```r
set.seed(1234)
TT <- 100
q <- 0.1
r <- 0.1
beta1 <- 0.6
beta2 <- 0.4
zt <- 0.5 * sin(2 * pi * (1:TT)/TT) + 0.75
cov1 <- sin(2 * pi * (1:TT)/12)
cov2 <- cos(2 * pi * (1:TT)/12)
xt <- cumsum(rnorm(TT, 0, q))
yt <- xt + zt * beta1 * cov1 + zt * beta2 * cov2 + rnorm(TT,
    0, r)
plot(yt, type = "l", xlab = "t")
```



15.4.1 Fitting the model

When the seasonality is written as

$$z_t\,(\beta_1 \sin(2\pi t/p) + \beta_2 \cos(2\pi t/p))$$

our model is under-determined because we have $z_t\beta_1$ and $z_t\beta_2$. We can scale the $z_t up and the β's correspondingly down and have the same values (so multiply z_t by 2 and divide the β's by 2, say). We can fix that by multiplying the z_t and dividing the seaonal part by β_1. Then our seasonal model becomes *Recognizing when your model is under-determined takes some experience. If you work in a Bayesian framework, it is a bit easier because it is easy to look at the posterior distributions and look for ridges.*

$$(z_t/\beta_1)\,(\sin(2\pi t/p) + (\beta_2/\beta_1)\cos(2\pi t/p)) = x_{2,t}\,(\sin(2\pi t/p) + \beta\cos(2\pi t/p))$$

The seasonality (peak location) will be the same for $(\sin(2\pi t/p) + \beta\cos(2\pi t/p))$ and $(\beta_1 \sin(2\pi t/p) + \beta_2 \cos(2\pi t/p))$. The only thing that is different is the amplitude and we are using $x_{2,t}$ to determine the amplitude.

Now our x and y models look like this. Notice that the $\mathbf{Z}$ is 1×2 instead of 1×3.

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

$$y_t = \begin{bmatrix} 1 & \sin(2\pi t/p) + \beta\cos(2\pi t/p) \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}_t + v_t$$

To set up the `Z` matrix, we can pass in values like `"1 + 0.5*beta"`. `MARSS()` will translate that to $1 + 0.5\beta$.

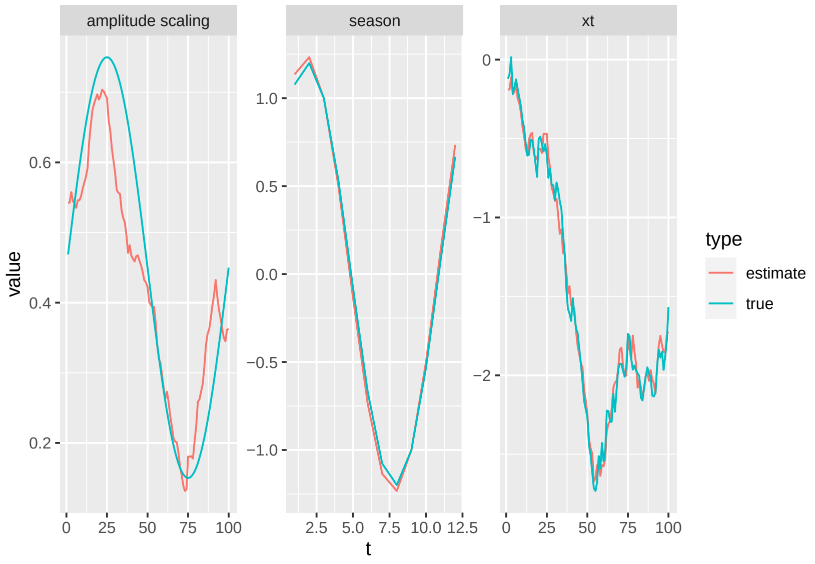
```
Z <- array(list(1), dim = c(1, 2, TT))
Z[1, 2, ] <- paste0(sin(2 * pi * (1:TT)/12), " + ", cos(2 * pi *
    (1:TT)/12), "*beta")
```

Then we make our model list. We need to set `A` since `MARSS()` doesn't like the default value of `scaling` when `Z` is time-varying.

```
mod.list <- list(U = "zero", Q = "diagonal and unequal", Z = Z,
    A = "zero")
```

```
require(MARSS)
fit <- MARSS(yt, model = mod.list, inits = list(x0 = matrix(0,
    2, 1)))
```

We are able to recover the level, seasonality and changing amplitude of the seasonality.



15.5 Multivariate responses

Let's imagine that we have two sites y_1 and y_2. We can model relationship between the seasonality in these two sites in many different ways. How we model it depends on our assumptions about our site or might reflect different relationships that we want to test.

15.5.1 Same seasonality and same level

In this case x_t is shared and the β's. We can allow the data to be scaled (translated up or down) relative to each other however.

$$\begin{bmatrix} x \\ \beta_1 \\ \beta_2 \end{bmatrix}_t = \begin{bmatrix} x \\ \beta_1 \\ \beta_2 \end{bmatrix}_{t-1} + \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}_t$$

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix}_t = \begin{bmatrix} 1 & \sin(2\pi t/p) & \cos(2\pi t/p) \\ 1 & \sin(2\pi t/p) & \cos(2\pi t/p) \end{bmatrix} \begin{bmatrix} x \\ \beta_1 \\ \beta_2 \end{bmatrix}_t + \begin{bmatrix} 0 \\ a_2 \end{bmatrix} + \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}_t$$

15.5.2 Same seasonality and same level but scaled

Same as the model above but we allow that the y level are stretched versions of x (so zx_t)). We only change one of the y to have a scaled x or else the model would have infinite number of solutions.

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix}_t = \begin{bmatrix} 1 & \sin(2\pi t/p) & \cos(2\pi t/p) \\ z & \sin(2\pi t/p) & \cos(2\pi t/p) \end{bmatrix} \begin{bmatrix} x \\ \beta_1 \\ \beta_2 \end{bmatrix}_t + \begin{bmatrix} 0 \\ a_2 \end{bmatrix} + \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}_t$$

15.5.3 Scaled seasonality and same level

We could also say that the are affect by the same seasonality but the amplitude is different. So covariate will be $z(\sin(2\pi t/12) + \cos(2\pi t/12))$ for the second y.

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix}_t = \begin{bmatrix} 1 & \sin(2\pi t/p) & \cos(2\pi t/p) \\ 1 & z\sin(2\pi t/p) & z\cos(2\pi t/p) \end{bmatrix} \begin{bmatrix} x \\ \beta_1 \\ \beta_2 \end{bmatrix}_t + \begin{bmatrix} 0 \\ a_2 \end{bmatrix} + \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}_t$$

15.5.4 Different seasonality but correlated

We might imagine that the seasonality is different between the sites but that the changes in seasonality are allowed to covary.

$$\begin{bmatrix} x \\ \beta_{1a} \\ \beta_{1b} \\ \beta_{2a} \\ \beta_{2b} \end{bmatrix}_t = \begin{bmatrix} x \\ \beta_{1a} \\ \beta_{1b} \\ \beta_{2a} \\ \beta_{2b} \end{bmatrix}_{t-1} + \begin{bmatrix} w \\ w_1 \\ w_2 \\ w_3 \\ w_4 \end{bmatrix}_t$$

$$\begin{bmatrix} w \\ w_1 \\ w_2 \\ w_3 \\ w_4 \end{bmatrix}_t \sim \text{MVN}\left(0, \begin{bmatrix} q & 0 & 0 & 0 & 0 \\ 0 & q_1 & c_1 & 0 & 0 \\ 0 & c_1 & q_1 & 0 & 0 \\ 0 & 0 & 0 & q_2 & c_2 \\ 0 & 0 & 0 & c_2 & q_2 \end{bmatrix}\right)$$

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix}_t = \begin{bmatrix} 1 & \sin(2\pi t/p) & \cos(2\pi t/p) & 0 & 0 \\ 1 & 0 & 0 & \sin(2\pi t/p) & \cos(2\pi t/p) \end{bmatrix} \begin{bmatrix} x \\ \beta_{1a} \\ \beta_{1b} \\ \beta_{2a} \\ \beta_{2b} \end{bmatrix}_t + \begin{bmatrix} 0 \\ a_2 \end{bmatrix} + \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}_t$$

15.5.5 Same seasonality and different level

In this case there is a different x_t for each y_t.

$$\begin{bmatrix} x_1 \\ x_2 \\ \beta_1 \\ \beta_2 \end{bmatrix}_t = \begin{bmatrix} x_1 \\ x_2 \\ \beta_1 \\ \beta_2 \end{bmatrix}_{t-1} + \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \end{bmatrix}_t$$

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix}_t = \begin{bmatrix} 1 & 0 & \sin(2\pi t/p) & \cos(2\pi t/p) \\ 0 & 1 & \sin(2\pi t/p) & \cos(2\pi t/p) \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \beta_1 \\ \beta_2 \end{bmatrix}_t + \begin{bmatrix} 0 \\ a_2 \end{bmatrix} + \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}_t$$

15.6 Summary

The MARSS structure allows you to model many different relationships between your multivariate observations. Fitting the models with `MARSS()` is just a matter of carefully setting up the matrices for the model list. In the next chapter, you will see an example of using these models to look at some real data.

Chapter 16

Modeling cyclic sockeye

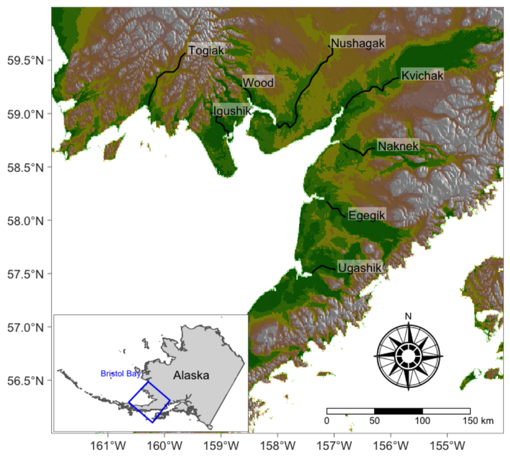
A script with all the R code in the chapter can be downloaded here. The Rmd for this chapter can be downloaded here

Data and packages

```
library(atsalibrary)
library(ggplot2)
library(MARSS)
```

16.1 Overview

In this chapter we will use DLMs to account for cyclicity in Bristol Bay sockeye spawner escapement (returning spawners minus catch).

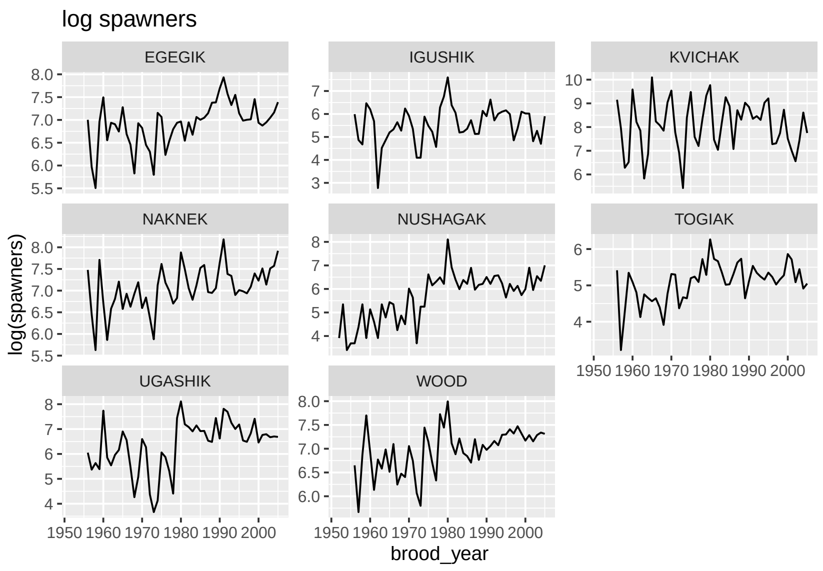


16.2 Analysis goal

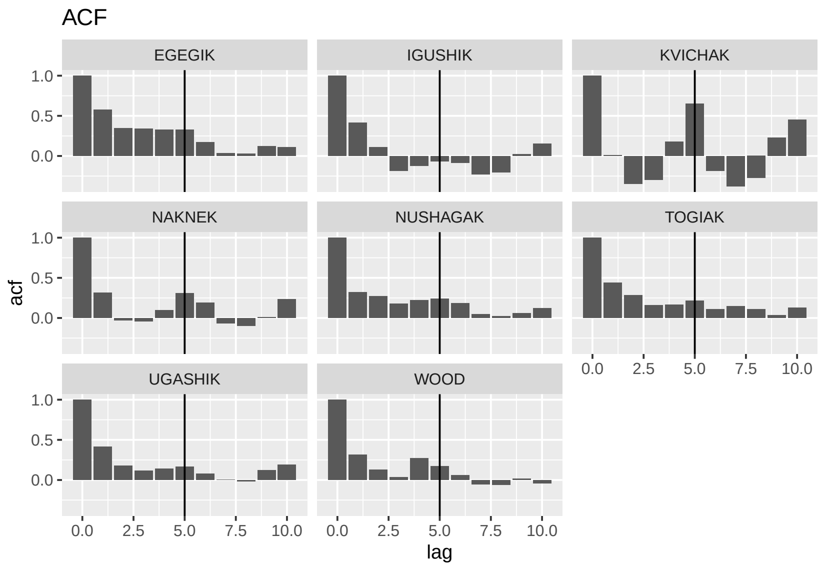
Our goal is to look for the underlying level (trends) in the rivers and studying whether there is evidence of synchronicity (correlated trend changes across rivers). In looking at the data, we have a couple problems.

1. There are strong cycles in the annual data due to the life cycle of sockeye which return 4-5 years after their brood year (year they were spawned). But unfortunately the cycles do not occur regularly every 5 years. Sockeye produce cycles that are 5 years apart but the cycle might break down for a year or two and then restart after a year or two. Thus the cycle appears to shift.
2. The amplitude of the cycle changes over time.

Both of these will cause us problems when we try to estimate a stochastic level.



ACF of the spawners showing the 5-year cycle in most of the rivers.



16.3 Modeling the cycle

As discussed in Chapter 15 in the covariates chapter, we can model changes in seasonality with a DLM with sine and cosine covariates. Here is a DLM model with the level and season modeled as a random walk.

$$\begin{bmatrix} x \\ \beta_1 \\ \beta_2 \end{bmatrix}_t = \begin{bmatrix} x \\ \beta_1 \\ \beta_2 \end{bmatrix}_{t-1} + \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}_t$$

$$y_t = \begin{bmatrix} 1 & \sin(2\pi t/p) & \cos(2\pi t/p) \end{bmatrix} \begin{bmatrix} x \\ \beta_1 \\ \beta_2 \end{bmatrix}_t + v_t$$

We can fit the model to the Kvichak River log spawner data and estimate the β's and stochastic level (x). This is annual data so what does p mean? p is the time steps between peaks. For sockeye that is 5 years. So we set $p = 5$. If p were changing, that would cause problems, but it is not for these data (which you can confirm by looking at the ACF for different parts of the time series).

16.3.1 Set up the data

```
river <- "KVICHAK"
df <- subset(sockeye, region == river)
yt <- log(df$spawners)
TT <- length(yt)
p <- 5
```

16.3.2 Specify the Z matrix

Z is time-varying and we set this up with an array with the 3rd dimension being time.

```
Z <- array(1, dim = c(1, 3, TT))
Z[1, 2, ] <- sin(2 * pi * (1:TT)/p)
Z[1, 3, ] <- cos(2 * pi * (1:TT)/p)
```

16.3.3 Specify the model list

Then we make our model list. We need to set **A** since `MARSS()` doesn't like the default value of `scaling` when **Z** is time-varying.

```
mod.list <- list(U = "zero", Q = "diagonal and unequal", Z = Z,
    A = "zero")
```

16.3.4 Fit the model

When we fit the model we need to give `MARSS()` initial values for `x0`. It cannot come up with default ones for this model. It doesn't really matter what you pick.

```
m <- dim(Z)[2]
fit <- MARSS(yt, model = mod.list, inits = list(x0 = matrix(0,
    m, 1)))
```

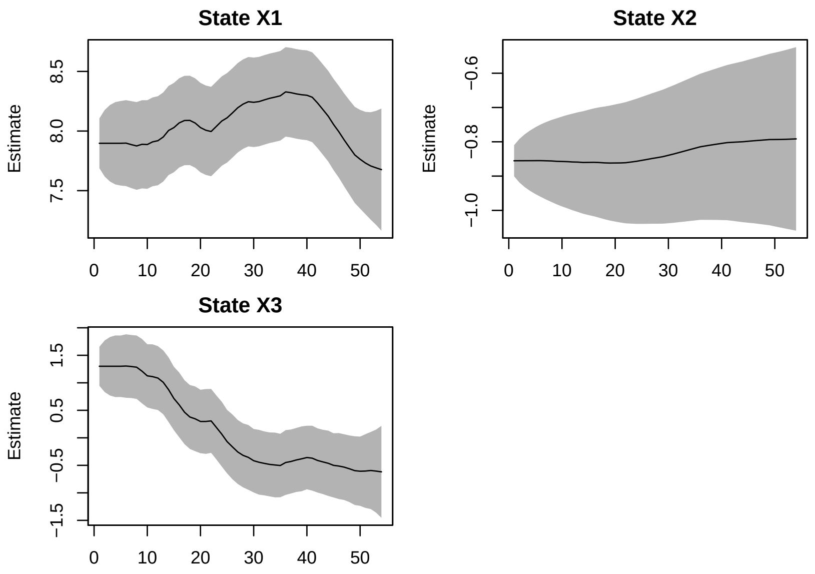
```
Warning! Abstol convergence only. Maxit (=500) reached before log-log convergence.

MARSS fit is
Estimation method: kem
Convergence test: conv.test.slope.tol = 0.5, abstol = 0.001
WARNING: Abstol convergence only no log-log convergence.
 maxit (=500) reached before log-log convergence.
 The likelihood and params might not be at the ML values.
 Try setting control$maxit higher.
Log-likelihood: -58.61614
AIC: 131.2323   AICc: 133.899

           Estimate
```

```
R.R        0.415426
Q.(X1,X1)  0.012584
Q.(X2,X2)  0.000547
Q.(X3,X3)  0.037008
x0.X1      7.897380
x0.X2     -0.855359
x0.X3      1.300854
Initial states (x0) defined at t=0

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

Convergence warnings
 Warning: the  Q.(X2,X2)  parameter value has not converged.
 Type MARSSinfo("convergence") for more info on this warning.
```

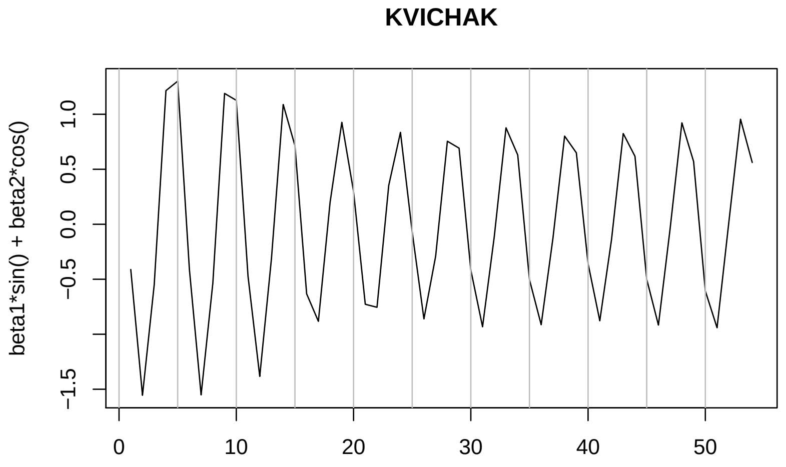
16.3.5 Plot the output

The β_1 estimate is State X2 and β_2 is State X3. The estimates match what we put into the simulated data.

```
plot type = "xtT" Estimated states
```



We can plot our cycle estimates and see that the peak has shifted over time. The peak has not been regularly every 5 years.



Let's look at the other rivers. Write a function to do the fit.

```
fitriver <- function(river, p = 5) {
    df <- subset(sockeye, region == river)
    yt <- log(df$spawners)
    TT <- length(yt)
    Z <- array(1, dim = c(1, 3, TT))
    Z[1, 2, ] <- sin(2 * pi * (1:TT)/p)
    Z[1, 3, ] <- cos(2 * pi * (1:TT)/p)
    mod.list <- list(U = "zero", Q = "diagonal and unequal",
        Z = Z, A = "zero")
    fit <- MARSS(yt, model = mod.list, inits = list(x0 = matrix(0,
        3, 1)), silent = TRUE)
    return(fit)
}
```

The make a list with all the fits.

```
fits <- list()
for (river in names(a)) {
    fits[[river]] <- fitriver(river)
}
```

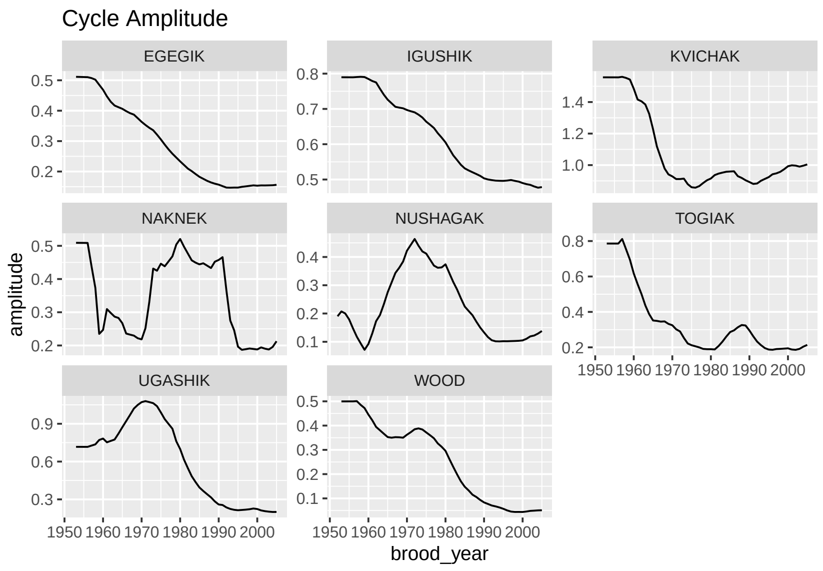
Create a data frame of the amplitude of the cycle ($\sqrt{\beta_1^2 + \beta_2^2}$) and the stochastic level (x).

```
dfz <- data.frame()
for (river in names(a)) {
    fit <- fits[[river]]
    tmp <- data.frame(amplitude = sqrt(fit$states[2, ]^2 + fit$states[3,
        ]^2), trend = fit$states[1, ], river = river, brood_year = subset(sock
        region == river)$brood_year)
    dfz <- rbind(dfz, tmp)
}
```

16.4 Univariate results

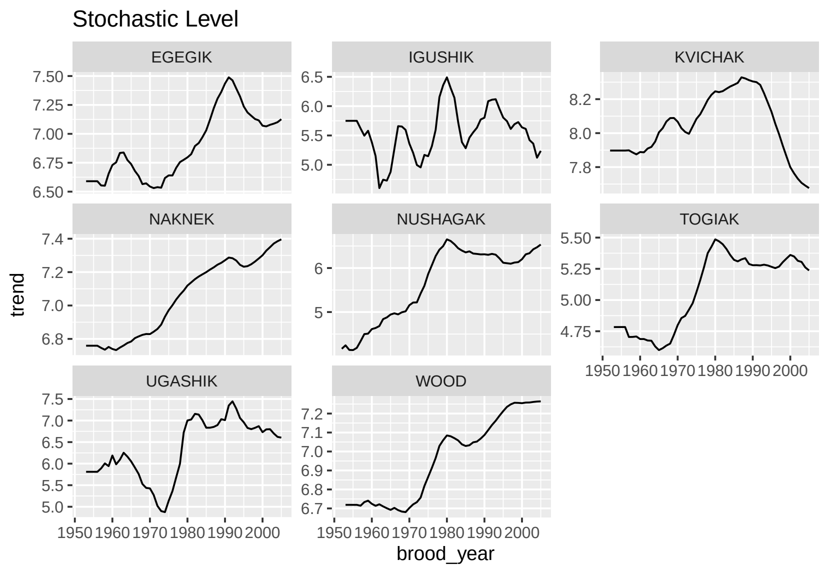
Plot of the amplitude of the cycles. All the rivers were analyzed independently. It certainly looks like there are common patterns in the amplitude of the cycles with many showing a steady decline in amplitude. Note the counts were not decreasing so this is not due to fewer spawners.

```
ggplot(dfz, aes(x = brood_year, y = amplitude)) + geom_line() +
    facet_wrap(~river, scales = "free_y") + ggtitle("Cycle Amplitude")
```



Plot of the stochastic level. Again all the rivers were analyzed independently. It certainly looks like there are common patterns in the trends. In the next step, we can test this.

```
ggplot(dfz, aes(x = brood_year, y = trend)) + geom_line() + facet_wrap(~river,
    scales = "free_y") + ggtitle("Stochastic Level")
```



16.5 Multivariate DLM 1: Synchrony in levels

In the first analysis, we will look at whether the stochastic levels (underlying trends) are correlated. We will analyze all the rivers together but in the equations, we will show just two rivers to keep the equations concise.

16.5.1 State model

The hidden states model will have the following components:

- Each trend x will be modeled as separate but allowed to be correlated. This means either an unconstrained **Q** or an equal variance and equal covariance matrix.
- Each seasonal trend, the β's, will also be treated as separate but independent. This means either a diagonal and equal variance **Q** or diagona and unequal variances.

The **x** equation is then:

$$\begin{bmatrix} x_a \\ x_b \\ \beta_{1a} \\ \beta_{1b} \\ \beta_{2a} \\ \beta_{2b} \end{bmatrix}_t = \begin{bmatrix} x_a \\ x_b \\ \beta_{1a} \\ \beta_{1b} \\ \beta_{2a} \\ \beta_{2b} \end{bmatrix}_{t-1} + \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \\ w_5 \\ w_6 \end{bmatrix}_t$$

$$\begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \\ w_5 \\ w_6 \end{bmatrix}_t \sim \text{MVN}\left(0, \begin{bmatrix} q_a & c & 0 & 0 & 0 & 0 \\ c & q_b & 0 & 0 & 0 & 0 \\ 0 & 0 & q_1 & 0 & 0 & 0 \\ 0 & 0 & 0 & q_2 & 0 & 0 \\ 0 & 0 & 0 & 0 & q_3 & 0 \\ 0 & 0 & 0 & 0 & 0 & q_4 \end{bmatrix}\right)$$

16.5.2 Observation model

The observation model will have the following components:

- Each spawner count time series will be treated as independent with independent error (equal or unequal variance).

$$\begin{bmatrix} y_a \\ y_b \end{bmatrix}_t = \begin{bmatrix} 1 & 0 & \sin(2\pi t/p) & 0 & \cos(2\pi t/p) & 0 \\ 0 & 1 & 0 & \sin(2\pi t/p) & 0 & \cos(2\pi t/p) \end{bmatrix} \begin{bmatrix} x_a \\ x_b \\ \beta_{1a} \\ \beta_{1b} \\ \beta_{2a} \\ \beta_{2b} \end{bmatrix}_t + \mathbf{v}_t$$

16.5.3 Fit model

Set the number of rivers.

```
n <- 2
```

The following code will create the **Z** for a model with n rivers. The first **Z** is shown.

```
Z <- array(1, dim = c(n, n * 3, TT))
Z[1:n, 1:n, ] <- diag(1, n)
for (t in 1:TT) {
    Z[, (n + 1):(2 * n), t] <- diag(sin(2 * pi * t/p), n)
    Z[, (2 * n + 1):(3 * n), t] <- diag(cos(2 * pi * t/p), n)
}
Z[, , 1]
```

```
     [,1] [,2]      [,3]      [,4]     [,5]     [,6]
[1,]    1    0 0.9510565 0.0000000 0.309017 0.000000
[2,]    0    1 0.0000000 0.9510565 0.000000 0.309017
```

And this code will make the **Q** matrix:

```
Q <- matrix(list(0), 3 * n, 3 * n)
Q[1:n, 1:n] <- "c"
diag(Q) <- c(paste0("q", letters[1:n]), paste0("q", 1:(2 * n)))
Q
```

```
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,] "qa" "c"  0    0    0    0   
[2,] "c"  "qb" 0    0    0    0   
[3,] 0    0    "q1" 0    0    0   
[4,] 0    0    0    "q2" 0    0   
[5,] 0    0    0    0    "q3" 0   
[6,] 0    0    0    0    0    "q4"
```

We will write a function to prepare the model matrices and fit. It takes the names of the rivers.

```
fitriver.m <- function(river, p = 5) {
    require(tidyr)
    require(dplyr)
    require(MARSS)
    df <- subset(sockeye, region %in% river)
    df <- df %>%
        pivot_wider(id_cols = brood_year, names_from = "region",
            values_from = spawners) %>%
        ungroup() %>%
        select(-brood_year)
    yt <- t(log(df))
    TT <- ncol(yt)
    n <- nrow(yt)
    Z <- array(1, dim = c(n, n * 3, TT))
    Z[1:n, 1:n, ] <- diag(1, n)
    for (t in 1:TT) {
        Z[, (n + 1):(2 * n), t] <- diag(sin(2 * pi * t/p), n)
        Z[, (2 * n + 1):(3 * n), t] <- diag(cos(2 * pi * t/p),
            n)
    }
    Q <- matrix(list(0), 3 * n, 3 * n)
    Q[1:n, 1:n] <- paste0("c", 1:(n^2))
    diag(Q) <- c(paste0("q", letters[1:n]), paste0("q", 1:(2 *
        n)))
    Q[lower.tri(Q)] <- t(Q)[lower.tri(Q)]
    mod.list <- list(U = "zero", Q = Q, Z = Z, A = "zero")
    fit <- MARSS(yt, model = mod.list, inits = list(x0 = matrix(0,
        3 * n, 1)), silent = TRUE)
    return(fit)
}
```

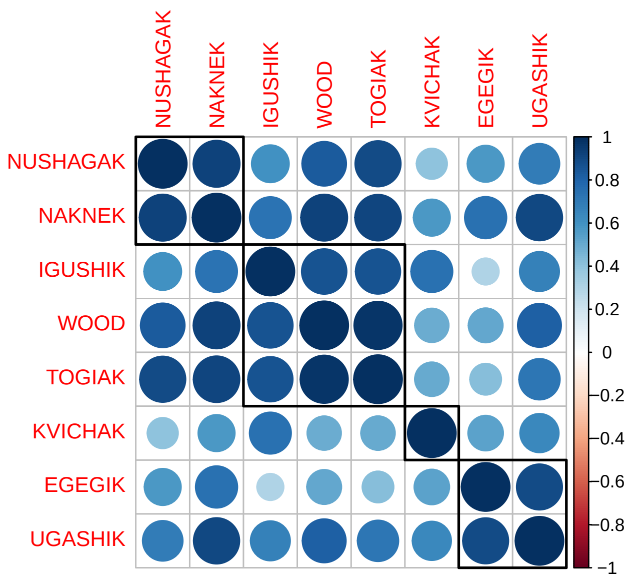
Now we can fit for two (or more) rivers. Note it didn't quite converge as some of the variances for the β's are going to 0 (constant β value).

```
river <- unique(sockeye$region)
n <- length(river)
fit <- fitriver.m(river)
```

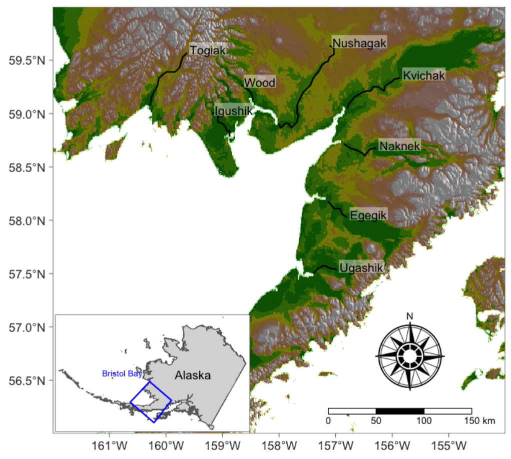
16.5.4 Look at the results

We will look at the correlation plot for the trends.

```r
require(corrplot)
Qmat <- coef(fit, type = "matrix")$Q[1:n, 1:n]
rownames(Qmat) <- colnames(Qmat) <- river
M <- cov2cor(Qmat)
corrplot(M, order = "hclust", addrect = 4)
```



We can compare to the locations and see that this suggests that there is small scale regional correlation in the spawner counts.


Togiak
Nushagak
Wood
Kvichak
Igushik
Naknek
Egegik
Ugashik
Bristol Bay
Alaska
N
0
50
100
150 km
59.5°N
59.0°N
58.5°N
58.0°N
57.5°N
57.0°N
56.5°N
161°W
160°W
159°W
158°W
157°W
156°W
155°W

Bibliography

Dormann, C. F., Elith, J., Bacher, S., Buchmann, C., Carl, G., Carré, G., Marquéz, J. R. G., Gruber, B., Lafourcade, B., Leitão, P. J., Münkemüller, T., McClean, C., Osborne, P. E., Reineking, B., Schröder, B., Skidmore, A. K., Zurell, D., and Lautenbach, S. (2013). Collinearity: a review of methods to deal with it and a simulation study evaluating their performance. *Ecography*, 36:027–046.

Holmes, E. E., Ward, E. J., and Scheuerell, M. D. (2014). Analysis of multivariate time-series using the marss package. Technical report, Northwest Fisheries Science Center, Seattle, WA.

Jorgensen, J. C., Ward, E. J., Scheuerell, M. D., and Zabel, R. W. (2016). Assessing spatial covariance among time series of abundance. *Ecology and Evolution*, 6:2472–2485.

Lamon, E. I., Carpenter, S., and Stow, C. (1998). Forecasting pcb concentrations in lake michigan salmonids: a dynamic linear model approach. *Ecological Applications*, 8:659–668.

Lisi, P. J., Schindler, D. E., Cline, T. J., Scheuerell, M. D., and Walsh, P. B. (2015). Watershed geomorphology and snowmelt control stream thermal sensitivity to air temperature. *Geophysical Research Letters*, 42(9):3380–3388.

Ohlberger, J., Scheuerell, M. D., and Schindler, D. E. (2016). Population coherence and environmental impacts across spatial scales: a case study of Chinook salmon. *Ecosphere*, 7:e01333.

Petris, G., Petrone, S., and Campagnoli, P. (2009). *Dynamic Linear Models with R*. Use R! Springer, London.

Pole, A., West, M., and Harrison, J. (1994). *Applied Bayesian forecasting and time series analysis*. Chapman and Hall, New York.

Scheuerell, M. D. and Williams, J. G. (2005). Forecasting climate induced changes in the survival of snake river spring/summer chinook salmon (oncorhynchus tshawytscha). *Fisheries Oceanography*, 14(6):448–457.

Stachura, M. M., Mantua, N. J., and Scheuerell, M. D. (2014). Oceanographic influences on patterns in North Pacific salmon abundance. *Canadian Journal of Fisheries and Aquatic Sciences*, 71(2):226–235.

Zuur, A. F., Fryer, R. J., Jolliffe, I. T., Dekker, R., and Beukema, J. J. (2003). Estimating common trends in multivariate time series using dynamic factor analysis. *Environmetrics*, 14(7):665–685.