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

Analysis of multivariate time-series using the MARSS package

version 3.10.2

April 15, 2016

Northwest Fisheries Science Center, NOAA
Seattle, WA, USA

Holmes, E. E., E. J. Ward and M. D. Scheuerell. Analysis of multivariate time-series using the MARSS package. NOAA Fisheries, Northwest Fisheries Science Center, 2725 Montlake Blvd E., Seattle, WA 98112. Contacts eli.holmes@noaa.gov, eric.ward@noaa.gov, and mark.scheuerell@noaa.gov

Preface

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

The application chapters were developed originally as part of workshops on analysis of multivariate time-series data given at the Ecological Society of America meetings since 2005 and taught by us along with Yasmin Lucero, Stephanie Hampton, and Brice Semmens. The chapter on extinction estimation and trend estimation was initially developed by Brice Semmens and later extended by us for this user guide. The algorithm behind the TMU figure in Chapter **??** was developed during a collaboration with Steve Ellner (**?**). Later we further developed the chapters as part of a course we taught on analysis of fisheries and environmental time-series data at the University of Washington, where we are affiliate faculty. You can find online versions of the workshops and the time-series analysis course on EEH's website `http://faculty.washington.edu/eeholmes`.

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

You are welcome to use the code and adapt it with full attribution. You should use citation **?** for the MARSS package. It may not be used in any commercial applications nor may it be copyrighted. Use of the EM algorithm should cite **?**. Links to more code and publications on MARSS applications can be found by following the links at our our academic websites:

- `http://faculty.washington.edu/eeholmes`
- `http://faculty.washington.edu/scheuerl`
- `https://sites.google.com/site/ericward2`

Contents

1

Standard errors, confidence intervals and prediction intervals for MARSS models

This chapter discusses standard errors, confidence intervals and prediction intervals for the states and observations in a MARSS model and shows you how to get these outputs with the MARSS functions. We will start by illustrating these concepts with a simple linear regression and then discuss them in the more complicated state-space context. When discussing confidence interval and prediction intervals, it is important to recognize that they concern properties of data we have not collected rather than data we have collected. When we write of the expected value of $\tilde{y}$ (new data) conditioned on the observed data, these two datasets are different. We do not need to calculate the expected value of data we have collected; the expected value is simply its value. To distinguish these two types of data, the data collected is called y and new data (and whose expected value we are interested in) is called $\tilde{y}$.

For this chapter, we will use a data set on the date of first spring flight observed in two butterfly species in the California Sierra from 1972 to 2002[1]. We will regress this data against the average maximum daily temperature in February (Figure 1).

We will assume that these data are a random sample from this generating process:

$$y = \alpha + \beta x + e, \qquad e \sim N(0, \sigma) \tag{1.1}$$

That is, that the day of first spring flight relative to the mean is a linear function of the average winter maximum temperature. We will assume that α = 299.655, β= -5.064, and σ=9.492 based on the 1972-2002 data. The red dots in Figure 1 show hypothetical data generated from this model with the observed data shown in blue. The red line is the expected value of data generated with Equation 1.1 at each x value on the x-axis. We will denote this expected value as $E[\tilde{y}|x]$.

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

[1] Collected by Dr. A. M. Shapiro at UC Davis. http://butterfly.ucdavis.edu/

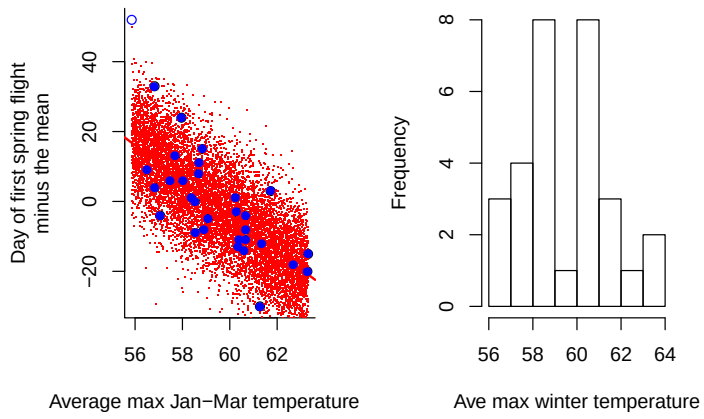

Fig. 1.1. First flight data for Common Skippers at field sites in Northern California 1972-2002. Left) The day of first observed spring flight (y) relative to the mean versus the average maximum daily temperature (x) recorded at the Willow Slough site in January to March. On the y axis, 0 indicates the average observed day of first flight from 1972 to 2002; 20 means 20 days after the average and -20 means before. The blue dots are the observed data and the red dots are the 'hypothetical' data generated from the fitted relationship: $y = \alpha + \beta x + e$ where $e \sim N(0, \sigma)$. The point marked with an open circle is 1983 and was removed as an outlier year. Right) Histogram of average maximum winter temperatures in the 30-year butterfly dataset.

1.1 Definition and properties of confidence intervals

A 95% confidence interval on some test statistic is an interval constructed in such a way that for 95% of the possible new data sets, the confidence interval includes (or covers) the true value of the statistic. We will be computing confidence intervals for $E[\tilde{y}|x = 58]$, the value of the red regression line at $x = 58$, based on a sample (the observed data) from the generating model (Equation 1.1).

Let's say we observe 10 data points from the generating model (Equation **??**). We will denote these observations $\mathbf{y}_j$ and through this chapter j is referring to this particular observed set of data. To generate a $\mathbf{y}_j$ for this chapter, we will set the random seed to 123, and then will generate both y (first flight day relative to mean) and x (average max winter temperature):

```
set.seed(123)
nsamp=10 #sample size
x.j = runif(nsamp, min(dat$x), max(dat$x))
y.j =  alpha + beta*x.j + rnorm(nsamp,0,sigma)
dat.j = data.frame(x=x.j, y=y.j)
```

```
#we will use the fit to the j-data throughout the chapter
fit.j = lm(y~x, data=dat.j)
df.j = fit.j$df.residual
alpha.j = coef(fit.j)[1]
beta.j = coef(fit.j)[2]
sigma.j = sqrt(sum(fit.j$residual^2)/df.j) #unbiased not MLE
```

There are other possible y data we could have collected with the same x values, say different years with the same set of temperatures or same years but a different schedule of site visits. These hypothetical samples will be denoted $\tilde{\mathbf{y}}_j$; the the j subscript reminds us that the x values are $\mathbf{x}_j$.

Figure 1.2c shows a regression line fit to $\mathbf{y}_j$. The value of this regression at $x = 58$ is our estimate of the expected value of new or hypothetical data generated at the value $x = 58$, denoted $E[\tilde{y}|\widehat{x = 58}]$. Our estimate of $E[\tilde{y}|x = 58]$ is $\hat{\alpha}_j + \hat{\beta}_j 58$. We can construct a confidence interval for $E[\tilde{y}|x = 58]$.

This 95% confidence interval is an interval constructed in such a way that for 95% of the possible random samples of 10, the confidence interval includes (or covers) the true expected value: $\alpha + \beta 58$. What does this mean? We can imagine other samples of 10 and other regression lines that we would compute with those samples. Here is code to create a large number of α, β and σ^2 estimates from samples of 10:

```
nsim = 5000
i.results=matrix(NA,nsim,3)
for(i in 1:nsim){
  x = runif(nsamp, min(dat$x), max(dat$x))
  y = alpha + beta*x + rnorm(nsamp,0,sigma)
  dat.i=data.frame(x=x, y=y)
  fit.i=lm(y~x, data=dat.i)
  i.results[i,]=c(coef(fit.i),
                  sqrt(sum(fit.i$residual^2)/fit.i$df.residual))
}
```

For each sample, we could compute $E[\tilde{y}|x = 58] = \hat{\alpha} + \hat{\beta} 58$ and we could construct a confidence interval for it just like in Figure 1.2c. If the confidence interval is constructed properly, for 95% of the regression lines in Figure 1.2b, the interval will include $\alpha + \beta 58$. Figure 1.2d shows that this is the case for the interval constructed as in panel b. The y-axis is the fraction of CIs that cover $\alpha + \beta 58$ minus the correct fraction (99%, 95% or 75%).

1.1.1 Logic behind the construction of confidence intervals

Figure 1.3 walks through the logic behind algorithms for construction of confidence intervals. There are two approaches you could take based on the distribution of regression lines from fixed x values or from random x values. The

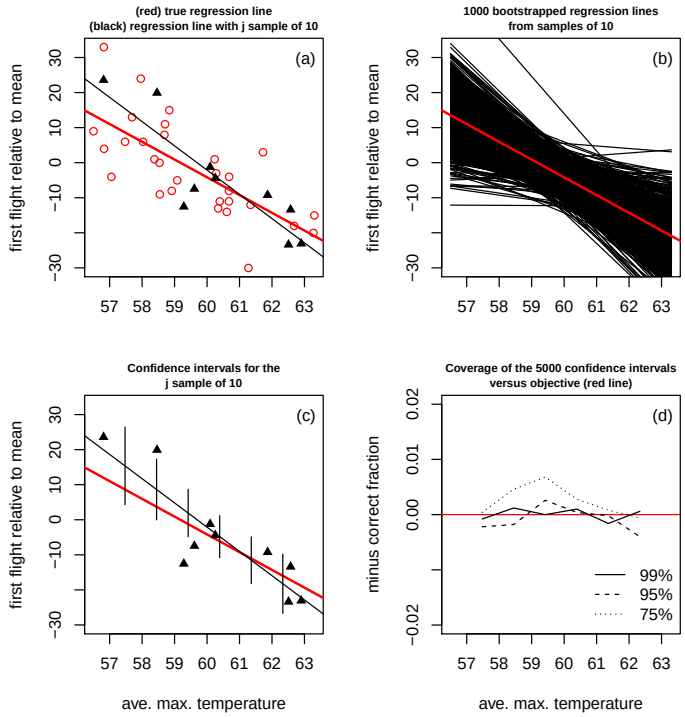

Fig. 1.2. Properties of confidence intervals

former is the typical way to construct CIs but the later arises in some types of bootstrapping.

Using the left-hand strategy (panels a-c), we start by thinking about a particular set of x values, the $\mathbf{x}_j$ which are marked as green lines in panel a). At those values, we can imagine a sets of y generated with the model:

$$\mathbf{y} = \alpha + \beta \mathbf{x}_j + \mathbf{e}$$

where the e in $\mathbf{e}$ are drawn from a Normal distribution with variance σ^2. We could then estimate the regression lines from the $\mathbf{y}$ datasets. These are shown by the grey lines in panel a). We could construct a confidence interval at $x = 58$ using the 95% range of values of the grey lines at $x = 58$ (panel b). If we were to center that blue line on each of regression lines in panel a), it would cover the red line 95% of the time.

The strategy in the right-hand panels is similar except the regression lines are generated from sets of x that are drawn randomly from the possible set of x. In the 30-year butterfly dataset, the x have bi-modal distribution (Figure 1). We imagine drawing 10 x randomly from that distribution and generate $\mathbf{y}$ with

$$\mathbf{y} = \alpha + \beta\mathbf{x} + \mathbf{e}$$

that would lead to a different distribution of regression lines shown in panel d. We would construct the blue line from the distribution of lines in panel d and use that for the CI for any set of 10 x we might draw.

The trick is to come up with a strategy for estimating the blue line—that is the distribution of regression lines in panels a) or d). The blue line is determined by the generating model. The left panel of Figure 1.1.1 shows the true bivariate distribution of $\hat{\alpha}$ and $\hat{\beta}$. We need to know the shape of this, but we do not need to know the true α and β that determine the location of the maximum. That's because the shape determines the length of the blue line and that's all we need to construct our CI. The different strategies for constructing CIs are based on estimating the shape of the distribution on the left from the data. The right hand panel illustrates this. This is the distribution of $\hat{\alpha}$ and $\hat{\beta}$ estimates from data sets generated from parametric bootstaps from one set of data[2]. You can see that the shape is similar to that on the left. All CI strategies are based on this idea of using the observed data to estimate the shape of the true distribution of $\hat{\alpha}$ and $\hat{\beta}$.

1.2 Different ways to compute confidence intervals

We cover five approaches for constructing confidence intervals. The first four use a parametric model: the linear model with Gaussian independent errors. The first of these is the analytical CI using a known σ^2, the second is the analytical CI using an estimate of σ^2, the third is a parametric bootstrap, and the fourth uses a numerically estimated Hessian matrix. The fifth approach uses resampling from the data.

All the above methods are asymptotically correct, as n gets large. This means as sample size gets large, all methods will produce x-CIs (e.g. 95% CIs) that cover the true value x% of the time.

1.2.1 Analytical construction of CIs using a known σ

We constructed the blue line in panel b) of Figure 1.3 by simulation. The code is shown below. Notice that we keep the x values constant and simulate new residual errors.

```
nsim=5000
ij.results=matrix(NA,nsim,2)
for(i in 1:nsim){
  y.ij=alpha+beta*x.j+rnorm(nsamp,0,sigma)
  fit.ij=lm(y.ij ~ x.j)
```

[2] Fit model to observed data. Use that model to generate data. Fit model to bootstrap data to get parameter estimates. Repeat.

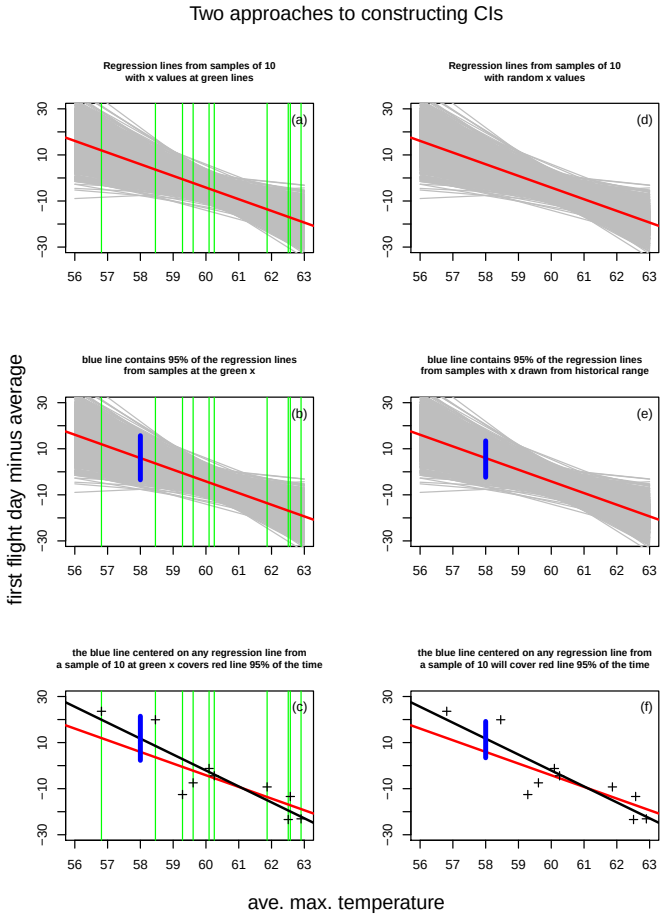

Fig. 1.3. Two approaches to construction CIs that will cover the red line (the true regression line) for 95% of any sample of 10. On the left side, we hold the x values constant and on the right, we chose them randomly from the possible sets of x values. In panel a) we show the distribution of regression lines for a particular set of x values, $\mathbf{x}_j$, shown by the green lines. The grey lines are random regression lines from samples generated from $y = \alpha + \beta\mathbf{x}_j + \mathbf{e}$, where the e in $\mathbf{e}$ are drawn from a Normal distribution with variance σ^2. In panel b) the blue line is a 95% CI constructed from the 95% range of the grey line values at $x = 58$. In panel c) we show one particular regression line (from the grey lines in panels a and b) from one particular set of y (the crosses) at $\mathbf{x}_j$. We center the blue line from panel b) on the black regression line value at $x = 58$. For 95% of the regression lines (the grey lines) in panel a), the blue line centered in this way will cover the red line. This works for any set of $\mathbf{x}$ values and thus this approach will properly define a CI for any $\mathbf{x}$. On the right side the strategy is similar, but instead of generating regressions at one set of $\mathbf{x}_j$, the regressions are from $\mathbf{x}$ drawn randomly from the possible sets of $\mathbf{x}$.

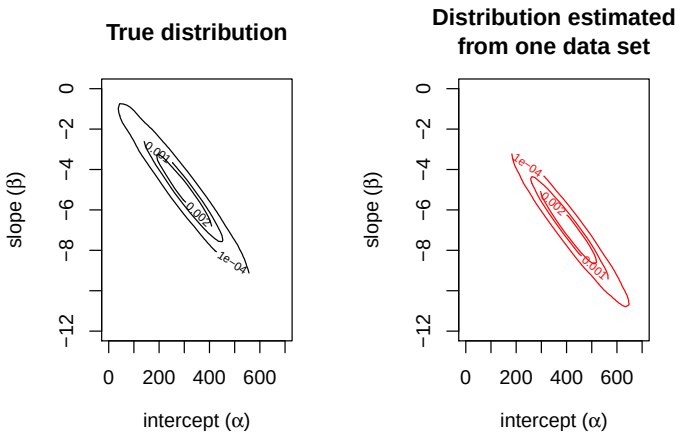

Fig. 1.4. True distribution of $\hat{\alpha}$ and $\hat{\beta}$ from random samples of 10 from the true generating model (left) versus the distribution from fitting a model to observed data and then estimating α and β from bootstrap data sets created with the model estimated from the observed data.

```
  ij.results[i,]=coef(fit.ij)
}
x1=58
CI=quantile(ij.results[,1]+ij.results[,2]*x1,probs=c(0.025,.975))
```

`CI` in the code is the blue line in panel b) in Figure 1.3.

However, we do not need to simulate for this problem because there is an analytical solution. The asymptotic[3] distribution of $\hat{\alpha}$ and $\hat{\beta}$ values that define the grey lines in panels a) and b) in Figure 1.3 is a multivariate normal distribution:

$$\begin{bmatrix}\hat{\alpha}\\ \hat{\beta}\end{bmatrix} \sim \text{MVN}(\theta, \Sigma)$$
$$\theta = \begin{bmatrix}\alpha\\ \beta\end{bmatrix} \text{ and } \Sigma = \mathbf{I}(\theta)^{-1} \tag{1.2}$$

where $\mathbf{I}(\theta)$ the Fisher information matrix. The Fisher information matrix is the second derivative of the negative log-likelihood function:

$$\mathbf{I}(\theta) = \begin{bmatrix}\frac{\partial^2 -\log L}{\partial\alpha\partial\alpha} & \frac{\partial^2 -\log L}{\partial\alpha\partial\beta}\\ \frac{\partial^2 -\log L}{\partial\alpha\partial\beta} & \frac{\partial^2 -\log L}{\partial\beta\partial\beta}\end{bmatrix}$$

For our linear regression model with Gaussian errors, the Fisher information matrix has a simple equation:

[3] meaning large n.

$$\mathbf{I}(\theta) = \frac{1}{\sigma^2}\mathbf{X}_j^\top \mathbf{X}_j$$

where $\mathbf{X}_j$ is a 2 column matrix with 1s in column 1 and the predictor variables (the $\mathbf{x}_j$) in column 2. The asymptotic variance-covariance matrix of the MLEs is the inverse of this, i.e. $\sigma^2(\mathbf{X}_j^\top \mathbf{X}_j)^{-1}$. Thus for our $\mathbf{x}_j$ in panel a (the green lines), the estimated α's and β's will be normally distributed with a variance of

```
Xj=cbind(alpha=1,beta=x.j)
analytical.Sigma=sigma^2*solve(t(Xj)%*%Xj)
analytical.Sigma
```

```
          alpha        beta
alpha 9105.7640 -150.517899
beta  -150.5179    2.490519
```

We can use the variance-covariance matrix of the MLEs, Σ, to compute an interval around $\alpha + 58\beta$ that contains 95% of the grey lines at $x = 58$. We know that

$$\text{var}(\hat{\alpha} + 58\hat{\beta}) = \text{var}(\mathbf{X}\hat{\theta}) = \mathbf{X}\,\text{var}(\hat{\theta})\mathbf{X}^\top = \mathbf{X}\Sigma\mathbf{X}^\top \tag{1.3}$$

where $\mathbf{X}$ is a matrix with 1 in column 1 and 58 in column 2 (58 is where we are construction the CI), , $\Sigma = \sigma^2(\mathbf{X}_j^\top \mathbf{X}_j)^{-1}$, $\mathbf{X}_j$ is a matrix with 1s in column 1 and $\mathbf{x}_j$ in column 2, and $\hat{\theta} = \begin{bmatrix}\hat{\alpha}\\ \hat{\beta}\end{bmatrix}$. An interval that contains 95% of these is

$$\mathbf{X}\theta + z^{.05/2}\sqrt{\mathbf{X}\Sigma\mathbf{X}^\top} = \mathbf{X}\theta + 1.96\sqrt{\mathbf{X}\Sigma\mathbf{X}^\top}$$

where $z^{.05/2}$ is the quantile of the unit Normal at $.05/2$.

Here is code to compute the analytical 95% interval of the regression lines at $x = 58$:

```
Xj = cbind(1, dat.j$x)
XjXj.inv = solve(t(Xj)%*%Xj)
Sigma = sigma^2*XjXj.inv
theta = rbind(alpha, beta)
x=58; X = cbind(1, x)
#the analytical CI
X%*%theta + qnorm(c(0.025,.975)) * sqrt(X%*%Sigma%*%t(X))
```

```
[1] -3.604446 15.516128
```

We can compare this to the real distribution of the regression lines at $x = 58$:

```
quantile(ij.results[,1]+ij.results[,2]*x, probs=c(0.025, 0.975))
```

```
    2.5%    97.5%
-3.487706 15.664808
```

They are pretty close even though n is quite small and the asymptotic solution is an approximation.

This gives us a way to calculate confidence intervals for $\mathrm{E}(\tilde{y}|x=58)$ if we know the true σ. The width of the CI is specified as above but we center it on $\mathbf{X}\hat{\theta}$ which is our estimate of $\mathrm{E}(\tilde{y}|x=58)$:

$$\mathbf{X}\hat{\theta} + z^{.05/2}\sqrt{\mathbf{X}\Sigma\mathbf{X}^\top}$$

where $\Sigma = \sigma^2(\mathbf{X}_j^\top \mathbf{X}_j)^{-1}$.

The problem is we don't know σ; we only have an estimate of σ.

1.2.2 Analytical construction of CIs using an estimate of σ

The parametric approach gives us a simple equation for Σ: $\sigma^2(\mathbf{X}_j^\top \mathbf{X}_j)^{-1}$. Why not just use that with an estimate of σ? So why not use the *observed* Fisher information matrix

$$\mathbf{I}(\hat{\theta}) = \frac{1}{\hat{\sigma}^2}(\mathbf{X}_j^\top \mathbf{X}_j)$$

in Equation 1.2. The problem is the distribution of $\tilde{y}_j$ conditioned on σ (known) is Normal but the distribution of $\tilde{y}_j$ conditioned on an estimate of σ has a t-distribution. For large n, approximating a t-distribution by a Normal distribution is not too bad, but for small n (like 10), it will lead to overly narrow CIs (too low coverage).

The corrected CIs are

$$\mathbf{X}\hat{\theta} + t_d^{.05/2}\sqrt{\mathbf{X}\hat{\Sigma}\mathbf{X}^\top} \tag{1.4}$$

where $t_d^{.05/2}$ is the t-distribution quantile at 0.025 with the degrees of freedom for the σ estimation (in our case $d = n-2$) and $\hat{\Sigma} = \hat{\sigma}^2(\mathbf{X}_j^\top \mathbf{X}_j)^{-1}$. The R code to compute this is

```
Xj = cbind(1, x.j)
XjXj.inv = solve(t(Xj)%*%Xj)
Sigma.j = sigma.j^2*XjXj.inv
theta.j = matrix(c(alpha.j,beta.j), ncol=1)
fit.df = fit.j$df.residual
#Compute CI at x=58
x=58; X = cbind(1, x)
EyX = X%*%theta.j
CI = EyX + qt(c(0.025,.975), df=fit.df) * sqrt(X%*%Sigma.j%*%t(X))
correct.ci.j = c(fit=EyX, lwr=CI[1], upr=CI[2])
```

It is this confidence interval that R's `predict` function returns:

```
correct.ci.j
```

```
     fit      lwr      upr
11.71982  1.99248 21.44715
```

```
predict(lm(y~x, data=dat.j), new=data.frame(x=x),interval="confidence")
```

```
       fit     lwr      upr
1 11.71982 1.99248 21.44715
```

We can simulate to show that analytical CIs with true σ are correct and those with estimated σ have under-coverage. We simulate y's at the $\mathbf{x}_j$ shown by the green lines in Figure 1.3 and compute CIs at $x = 58$ using true or estimated σ and then using the correction. Then we will see if these CIs cover the red line 95% of the time (or not).

```
Xj = cbind(1, x.j)
XjXj.inv = solve(t(Xj)%*%Xj)
true.Sigma = sigma^2*XjXj.inv
#we are going to compute CI at x=58
x=58; X = cbind(1, x)
#holders
nsim=5000
i.CIs.bad=i.CIs.true=i.CIs.corr=matrix(NA,nsim,2)
for(i in 1:nsim){
  tilde.y=alpha+beta*x.j+rnorm(nsamp,0,sigma)
  fit.i=lm(tilde.y ~ x.j)
  hat.theta=matrix(coef(fit.i),ncol=1)
  hat.sigma = sqrt(sum(fit.i$residual^2)/fit.i$df.residual)
  hat.Sigma = hat.sigma^2*XjXj.inv
  meanCI = X%*%hat.theta
  normaldist = qnorm(c(0.025,0.975))
  tdist = qt(c(0.025,0.975),df=fit.i$df.residual)
  #CI using asymptotic equation and estimated Sigma
  i.CIs.bad[i,] = meanCI + normaldist * sqrt(X%*%hat.Sigma%*%t(X))
  #CI using asymptotic equation and true Sigma
  i.CIs.true[i,] = meanCI + normaldist * sqrt(X%*%true.Sigma%*%t(X))
  #Corrected CI using t-distribution
  i.CIs.corr[i,] = meanCI + tdist * sqrt(X%*%hat.Sigma%*%t(X))
}
```

The true value (red line) at $x = 58$ is $\alpha + 58\beta$. The correct CI using the true σ has the correct coverage:

```
x=58
true.val = alpha+beta*x
100*sum(i.CIs.true[,1]<true.val & i.CIs.true[,2]>true.val)/nsim
```

```
[1] 94.96
```

But the CIs using the estimated σ^2 are too narrow. They have low coverage (less than 95%):

```
100*sum(i.CIs.bad[,1]<true.val & i.CIs.bad[,2]>true.val)/nsim
```

```
[1] 91.54
```

If we use the t-distribution's 95% intervals to compute our CIs, the coverage is correct again:

```
100*sum(i.CIs.corr[,1]<true.val & i.CIs.corr[,2]>true.val)/nsim
```

```
[1] 94.66
```

This illustrates the problem of using an estimate of σ instead of the true value. This same problem will arise when we look at other approaches to computing confidence intervals.

1.2.3 Constructing confidence intervals using a numerically estimated information matrix

To construct the analytical CIs, we estimated the distribution of $\alpha+58\beta$ using an estimate of the distribution of $\hat{\alpha}$ and $\hat{\beta}$ based on the observed Fisher information matrix: $\hat{\Sigma} = \mathbf{I}(\hat{\theta})^{-1}$. We have an analytical solution for $\mathbf{I}(\hat{\theta})^{-1}$, but we could also use R to generate a numerical estimate of the information matrix. This doesn't make much sense here since we have an analytical solution, but it is useful when we do not know or have the analytical solution.

Another term for the observed Fisher information matrix is the Hessian of the negative log-likelihood function at $\hat{\theta}$. There are a number of R functions that will estimate the Hessian of a function. We will use `optim()`. First we define function to return the negative log-likelihood for our model, a linear regression with Gaussian errors.

```
# Define the log likelihood function for a linear regression
# parm is the alpha, beta, sigma vector
NLL <- function(parm,  dat=NULL){
  #parm is alpha, beta, sigma
  resids = dat$y - dat$x * parm[2] - parm[1]
  dresids = suppressWarnings(dnorm(resids, 0, parm[3], log = TRUE))
  -sum(dresids)
}
```

Then we can pass this function into `optim()` with `hessian=TRUE`. To work well, `optim()` needs good starting values. We pass in really good ones, i.e. the output from `lm()`. Remember that j here is referring to the observed j sample of 10 ('the data'). α_j is the α estimate from that sample.

```
start.pars = c(alpha.j, beta.j, sigma.j)
ofit.j=optim(start.pars, NLL, dat=dat.j, hessian=TRUE)
parSigma = solve(ofit.j$hessian)[1:2,1:2]
parMean = ofit.j$par[1:2]
```

This will output the observed Fisher information matrix at the MLEs. The σ that `lm()` uses is different; it is the unbiased estimate. Thus the observed Fisher information matrix output here is different because it is using σ_{MLE}. We could change our code to use the unbiased estimate, but often one computes the Hessian at the MLEs, including the MLE of σ.

We can then use the Hessian to compute CIs:

```
X = cbind(1, 58)
EyX = X%*%parMean
hessian.cis = c(EyX, EyX + qnorm(c(0.025,.975))*sqrt(X%*%parSigma%*%t(X)))
```

Because we use the estimated variance instead of the true variance, the confidence intervals from the numerical estimate of the observed Fisher information matrix should also be too narrow:

```
rbind(hessian=hessian.cis, correct=correct.ci.j)
```

```
             fit      lwr      upr
hessian 11.71696 4.321769 19.11216
correct 11.71982 1.992480 21.44715
```

Though not necessary here, often one computes CIs by simulating from the estimated Σ, generating a large number of estimates of the metric of interest, and then using the quantiles of that. Section 1.6 includes a functions `hessian.boot` and `hessian.boot.cis` to generate CIs using parameter estimates generated from an estimated information matrix.

1.2.4 Constructing CIs via bootstrapping

The basic idea behind a bootstrap CI is that the data are used to generate new data sets (bootstrap data sets) from which parameters are estimated to give a large set of bootstrap parameter estimates and thus regression lines. On average, the variance-covariance matrix of the bootstrapped parameter estimates will be close to the variance-covariance matrix of the MLE parameter estimates (Σ for our example). Thus the bootstrapped parameter estimates can used to generate CIs.

The procedure is simple. First you generate a large number of bootstrapped data sets, then estimate the model parameters from each data set to get the bootstrap parameter estimates. For each set of bootstrapped parameter estimates, compute the metric of interest. We are interested in CIs for the fitted value at x, so we compute $\hat{\alpha}_b + \hat{\beta}_b x$ for each bootstrap. The 95% quantiles of the $\hat{\alpha}_b + \hat{\beta}_b x$ define the bootstrap CIs. Here is a function to do this:

```
#takes a set of boot parameters and makes CIs from them at x
boot.CI=function(boot.params, x, alp=0.05){
  CIs=apply(
    boot.params[,c("alpha","beta"),drop=FALSE]%*%rbind(1,x),
    2,quantile, c(0.5, alp/2, 1-alp/2) )
```

```
  colnames(CIs)=x
  t(CIs) #to look like predict output
}
```

The are a different ways to generate the bootstrap data sets. We cover parametric bootstrapping, residuals resampling, and data resampling.

1.2.5 Constructing CIs via parametric bootstrapping

In a parametric bootstrap, the estimated model is used to generate bootstrapped data, and the parameters are estimated from that bootstrapped data. The bootstrapped parameter estimates are then used to construct CIs. It is the same idea as a traditional bootstrap, but we are not sampling from the data to generate new bootstrap data sets but rather using the estimated model to generate new data. This is similar to the analytical CI approach and CIs from an estimated Hessian, in that one uses the model and the parameter estimates to estimate the distribution of parameter estimates.

Why use a parametric bootstrap instead of the analytical CIs? You use it when you want CIs based on your parametric model, but you do not know the analytical solution of Σ or you want to approximate the small n distribution rather than use the large n approximation. Why not use an estimate of the observed Fisher information matrix (Hessian)? You do not want to use the large n approximation or estimation of the Hessian is unstable.

To generate bootstrap parameter estimates via a parametric bootstrap, we generate data from our estimated model: $\mathbf{y}_b = \hat{\alpha} + \hat{\beta}\mathbf{x} + \mathbf{e}$ where each e in $\mathbf{e}$ is drawn from a Normal distribution with mean 0 and variance $\hat{\sigma}^2$. From each $\mathbf{y}_b$, we estimate α and β as usual, e.g. using `lm`, and construct the CIs using the set of bootstrap estimates.

Section 1.6 shows functions `parametric.boot` and `parametric.boot.cis` to do this. It takes the original data, gets the MLEs, and then uses those to generate data and a set of bootstrapped parameter estimates. Notice that the function holds the x values equal to our observed values when simulating new data. That's rather important.

Parametric bootstrapping uses the estimated σ to generate new data. Thus, as you would expect, the confidence intervals should be too narrow:

```
rbind(
  parametric=parametric.boot.cis(dat.j, x=58)[1,],
  correct=correct.ci.j
)
```

```
               50%     2.5%    97.5%
parametric 11.61505 4.119191 19.58689
correct    11.71982 1.992480 21.44715
```

However the problem diminishes as sample size increases; 10 is a very small sample size. We will address how to correct this bias after covering all the bootstrapping approaches.

We can compare the bivariate distributions of parameter estimates from a parametric bootstrap to parameters drawn from the estimated parameter distribution using the numerically estimated Hessian matrix at the MLEs (Figure 1.5). They should be very similar. The only difference (besides the parametric bootstrap using simulation) is that the Hessian approach is based on a large n approximation and our parametric bootstrap used the unbiased estimate of σ.

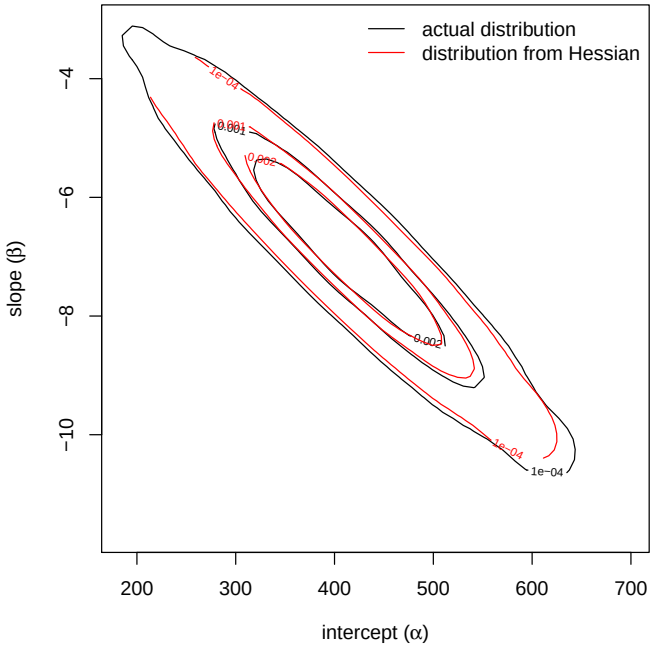

Fig. 1.5. Comparison of the distribution of α and β estimates from parametrically bootstrapping using the data $\mathbf{y}_j$ to the distribution from the Hessian of the log-likelihood function at the MLE values.

1.2.6 Constructing CIs via resampling from the residuals

The analytical and parametric bootstrap CIs are based on assuming that the residual errors can be described by a particular statistical distribution. In our example we use a Normal distribution. Sampling from the residuals allows us to compute CIs when we are unwilling to make a specific assumption about the distribution but are willing to treat the structure of the data as known:

$\mathbf{y} = \alpha + \beta\mathbf{x} + \mathbf{e}$. Instead of assuming the $\mathbf{e}$ come from a specific distribution, we generate $\mathbf{e}$ for our bootstrap data by sampling with replacement from the residual errors from the fit to the original data. Section 1.6 shows functions to do this.

Once a large number of bootstrap data sets are generated, the construction of confidence intervals proceeds as for the parametric bootstrap. The model is fit to each bootstrap dataset $\mathbf{y}_b$ and α_b and β_b are estimated. Those parameters are then used to compute CIs.

However, once again we used the variance in the observed residuals to generate samples of residuals. So, again you would expect the confidence intervals should be too narrow. And they are:

```
rbind(
  residuals=residuals.boot.cis(dat.j, x=58)[1,],
  correct=correct.ci.j
)
```

```
               50%     2.5%    97.5%
residuals 11.99081 3.566076 18.43254
correct   11.71982 1.992480 21.44715
```

1.2.7 Constructing confidence intervals via resampling the data

The last approach we will cover for creating bootstrap datasets is to sample, with replacement, from the data. This approach follows the logic of panels d-f in Figure 1.3. We are trying to estimate, via resampling, the distribution of regression lines that we would see if we had many samples of data drawn from the 'universe' of possible y and x.

Note that in this approach, the predictor variables, the x, in your bootstrap data (your $\mathbf{y}_b$) change from bootstrap to bootstrap. That might be problem. The variance of your regressions depends on your x values. What if your x values must be fixed—because you choose them to be a particular value, for example, as part of your experimental design. Using anything other than the x values you chose would give the wrong CIs. In this case, resampling from the residuals would make more sense.

In other cases, the x values you observed were random (you did not choose them). If your sample size is large enough, resampling from the data has some benefits. Like resampling from the residuals, you are not assuming a specific distribution for the residuals, unlike the analytical CIs or parametric bootstrap CIs. Unlike for sampling from the residuals, you also allow that the residual distribution could be different for different x values. For example, this approach would allow the residual variance to be smaller for large x and larger for small x, say.

Section 1.6 shows functions to create bootstrap data by resampling the data. Once a large number of bootstrap data sets are generated, the CIs are created as usual. As usual we are using the variance in the data as a proxy for

the variance in the 'universe' of data and the CIs will tend to be too narrow. However, this is countered by the fact that the x values in our resamples will have lower spread (because we are resampling from the observed x) and this tends to cause the CIs to be larger than they should be. This problem is severe for small samples, like $n = 10$.

```
rbind(
  resampling=resampling.boot.cis(dat.j, x=58)[1,],
  correct=correct.ci.j
)
```

```
                50%     2.5%    97.5%
resampling 11.74418 -3.16275 19.17828
correct    11.71982  1.99248 21.44715
```

But this problem diminishes as sample size increases:

```
#create a larger sample
nsamp=50
x = runif(nsamp, min(dat$x), max(dat$x))
y = alpha + beta*x + rnorm(nsamp,0,sigma)
dat.big=data.frame(x=x, y=y)
rbind(
  residuals=resampling.boot.cis(dat.big, x=x1)[1,],
  correct=predict(lm(y~x, data=dat.big), new=data.frame(x=58),interval="conf")
)
```

```
               50%     2.5%    97.5%
residuals 4.901254 1.666834 7.864881
1         4.920030 1.777711 8.062349
```

1.2.8 Correcting under-coverage of CIs

Ignoring the fact that we used an estimate of σ (either explicitly or implicitly) rather than the true value leads to overly small CIs with under-coverage. We saw this when the CIs from the parametric, non-parametric and Hessian bootstraps were compared to the correct analytical CIs coming from the `predict` function.

How do we fix this? We need to estimate the correction factor, i.e how much to increase the width of our estimated CIs. From the analytical CIs, we know that this correction factor is $t_{5\%/2,df}/z_{5\%/2}$ or in R `qt(0.025, df=8)/qnorm(0.025)` for our example with 10 data points and Gaussian errors. We can estimate the correction factor via simulating from the estimated model. The basic idea is to get a correct CI for one set of parameters, the estimated parameters, and then generate bootstrap data and estimate bootstrap CIs. Then you estimate how small, on average, the bootstrap CIs are to the CI estimated from

the observed data. That mean bias is the correction factor. Section 1.6 has a function to do this.

Here is the

```
#the correct adjustment:
qt(c(0.025),df=fit.j$df.residual)/qnorm(c(0.025))
```

```
[1] 1.176554
```

```
#adjustment computed via bootstrapping
ci.adj(dat.j, 58, type="analytical")
```

```
[1] 1.070423
```

Unfortunately computing the CI adjustment with bootstrapping takes a large number of bootstraps and is thus quite slow particular for the parametric and residuals bootstraps. Also it is an estimate of the bias. The expected value of the estimate will be the correct bias, but for any one data set, the estimated bias will not be precisely correct.

Why not draw σ from its bootstrapped distribution and use that in our CI construction? So instead of using $\hat{\alpha}_b$ and $\hat{\beta}_b$ only, we also use $\hat{\sigma}_b^2$. Wouldn't that properly account for the fact that we use an estimate of σ? No, unfortunately it does not.

1.3 Prediction intervals on the fitted $\tilde{y}$

The confidence intervals in Figure **??** are for $\alpha + \beta x$, or the expected value of $\tilde{y}$. The prediction intervals show the (estimated) 95% interval of $\tilde{y}$, not the expected value of $\tilde{y}$. The 95% prediction intervals should contain (or cover) 95% of $\tilde{y}$ (mpg) observed for cars of weight x. Prediction intervals are wider than confidence intervals because prediction intervals predict the distribution of $\tilde{y}$ while confidence intervals predict the expected value of $\tilde{y}$.

Let's go back to our true relationship between mpg and weight (the red line in Figure 1.2a). This is the relationship for our 32 car dataset, but let's imagine it holds for all models of cars. Let's also say that the residual variance we see in Figure 1.2a (the difference between the dots and the red line) characterizes the variability in the relationship for all models of cars. We could imagine many more data points (car models) around the red line, which we could generate like so:

```
fit = lm(mpg~wt, data=mtcars)
s2 = sum(fit$residual^2)/fit$df.residual
alpha=coef(fit)[1]
beta=coef(fit)[2]
plot(mtcars$wt,mtcars$mpg,ylim=c(0,60),xlab="car weight",ylab="mpg")
x = runif(1000,1,6)
```

```
y = alpha+beta*x+rnorm(1000,0,sqrt(s2))
points(x,y)
abline(fit, lwd=2, col="red")
```

Then we could construct an interval at any x (car weight) that would contain 95% of the data points at that x (Figure 1.6a). For any new $\tilde{y}$, there would be a 95% chance it would fall within our constructed interval. Since our errors are Gaussian, this is just $\alpha+\beta x \pm 1.96\sigma$. We could add these to our plot using:

```
lines(c(0,10),alpha+beta*c(0,10)+1.96*sqrt(s2),col="red",lty=2,lwd=1)
lines(c(0,10),alpha+beta*c(0,10)-1.96*sqrt(s2),col="red",lty=2,lwd=1)
```

However, we do not know α, β or σ^2, so we cannot construct this perfect prediction interval. But we can devise a method of constructing a prediction interval such that 95% of the constructed prediction intervals will cover new $\tilde{y}$. This is the same idea as constructing confidence intervals except that now we are predicting $\tilde{y}$ instead of the expected value of $\tilde{y}$.

The analytical equation for the prediction intervals assumes the data come from the linear model with Gaussian errors and can be generated using the predict function `predict`:

```
npred=1000
plot(mtcars$wt,mtcars$mpg,ylim=c(0,60),type="n",xlab="car weight",ylab="mpg")
x = runif(npred,1,6)
y = alpha+beta*x+rnorm(npred,0,sqrt(s2))
points(x,y, col="grey")
abline(fit, lwd=2, col="red")
abline(fit.j, lwd=2)
points(mtcars.j$wt, mtcars.j$mpg, pch=3)
preds = predict(lm(mpg~wt,data=mtcars.j), newdata= data.frame(wt=pred.wt), interval="predi
lines(pred.wt,preds[,2],lty=2)
lines(pred.wt,preds[,3],lty=2)
```

Figure **??**b shows the prediction intervals computed using our j sample (the crosses). These are a bit wide and cover a little too much of the $\tilde{y}$ distribution (the grey circles). The prediction interval is worse, meaning too much coverage, the farther we try to predict from the center of the prediction variable (car weight):

```
#get the 95% pred intervals for each x
preds = predict(fit.j, newdata= data.frame(wt=x), interval="prediction")
#see how many y fall outside that
1-sum(y>preds[,3] | y<preds[,2])/npred
#see how many fall inside at different x values
1-tapply(y>preds[,3] | y<preds[,2],cut(x,breaks=1:6),mean)
```

However that was just for one sample of 10 cars. If we look at large number of 10 car samples, we see that on average the prediction interval does cover 95% of the $\tilde{y}$:

```
#j sample of 10 cars
nsim = 5000
pi.coverage=rep(NA, nsim)
for(i in 1:nsim){
  tmp.fit=lm(mpg~wt, data=mtcars, subset=sample(dim(mtcars)[1], 10))
  preds = predict(tmp.fit, newdata= data.frame(wt=x), interval="prediction")
  pi.coverage[i] = 1-sum(y>preds[,3] | y<preds[,2])/npred
}
sum(pi.coverage)/nsim
```

In the above code, the $\tilde{y}$ were generated using a Gaussian distribution and this gave us a bit of an unfair advantage. We could also generate $\tilde{y}$ by sampling from the residuals. This is fairer since the data are not exactly normal. However, the average coverage is close to 95% even with new data generated by sampling from the residuals.

```
y = alpha+beta*x+sample(residuals(fit), npred, replace=TRUE)
nsim = 5000
pi.coverage=rep(NA, nsim)
for(i in 1:nsim){
  tmp.fit=lm(mpg~wt, data=mtcars, subset=sample(dim(mtcars)[1], 10))
  preds = predict(tmp.fit, newdata= data.frame(wt=x), interval="prediction")
  pi.coverage[i] = 1-sum(y>preds[,3] | y<preds[,2])/npred
}
sum(pi.coverage)/nsim
```

1.3.1 Computing prediction intervals via bootstrapping

In the same way that we generated confidence intervals via bootstrapping, we also can generate prediction intervals from bootstrapping. The idea is fairly simple. Say we want to generate a prediction interval for weight x. We fit a model to the data and then use the fitted model to generate bootstrap data. We fit to that bootstrap data to get bootstrap parameter estimates including an estimate of the residual variance. We then generate a new data point for x using those parameter estimates. We repeat this thousands of times to get many predictions for x and the 95% quantiles are the bootstapped prediction intervals at x.

1.3.2 Prediction interval includes two types of uncertainty

Includes uncertainty about the relationship between mpg and weight, i.e. in the regression parameters, and uncertainty in what $\tilde{y}$ will be due to observation error.

We will not get the right prediction interval if we simply use our estimated regression line and add on the estimated residual variance. I think this might work though: bootstrapping a bunch of regression lines and simulating new data based on $\hat{\alpha}_b + \hat{\beta}_b x + e$ with $e \sim \mathrm{N}(0, \hat{\sigma^2}_b)$.

Fig. 1.6. Comparison of four ways to compute prediction intervals. There is no one true CI as there are many CI algorithms that would could produce proper coverage; i.e. that the x-CI would cover the true relationship x percent of the time. The red line CIs are based on the true distribution of the α and β from all the 10 car samples from the original 32 car dataset. A CI based on this true distribution is simple one of the CIs that will have proper coverage.

1.3.3 CI computation for the parameters of MARSS models

The MARSS package will allow you to construct CIs via an estimated Hessian, parametric bootstrapping, or residuals bootstrapping. Construction of CIs using an estimated Hessian distribution is the fastest and it the default approach for producing CIs for the parameter estimates via function `MARSSparamCIs()`. The residuals bootstrap, called innovations bootstrapping for MARSS models, is useful if one is unwilling to assume a particular distribution for the errors. Residuals bootstrapping is selected by passing in `method="innovations"` to `MARSSparamCIs()`. The parametric bootstrap is useful when the estimate of the Hessian is numerically difficult or the multivariate normality assumption for the parameters is dubious. Parametric bootstrapping is selected by passing in `method="parametric"` to `MARSSparamCIs()`. It should be kept in mind that the methods discussed here do not work that well if the likelihood surface is multi-modal.

If one needs CIs for a metric that is some function of the estimated parameters, then CIs can be constructed using the intervals (e.g. 95%) from a large number of bootstrapped parameter estimates. The function `MARSSboot` will generate bootstrap parameter estimates via an estimated Hessian, residuals bootstrapping or parametric bootstrapping.

1.4 Expected values of states and data in MARSS models

Unlike a linear regression, a MARSS model has two types of random variables: the new data ($\tilde{y}$) like a regression but also the state ($\mathbf{x}$). In a linear regression, we are uncertain about the relationship (the red line i Figure **??**) because we are uncertain about the parameters that describe that line. In a MARSS model, we have the uncertainty about the parameters, but even if we did not,

we would still be uncertain about the $\mathbf{x}$ that the $\tilde{y}$ are observations of because $\mathbf{x}$ is a random process.

$$\begin{aligned} \mathbf{x}_t &= \mathbf{x}_{t-1} + \mathbf{w}_t \text{ where } \mathbf{w}_t \sim \text{MVN}(0, \mathbf{Q}) \\ \mathbf{y}_t &= \mathbf{Z}\mathbf{x}_t + \mathbf{a} + \mathbf{v}_t \text{ where } \mathbf{v}_t \sim \text{MVN}(0, \mathbf{R}) \\ \mathbf{x}_0 &\sim \text{MVN}(\boldsymbol{\pi}, \boldsymbol{\Lambda}) \end{aligned} \tag{1.5}$$

In a linear regression, you, typically, plot the regression line on to the data (as in Figure **??**a). The regression line is the expected value of new y at a given x given the estimated parameter values. You can get the expected value of $\tilde{y}$ from a linear regression fit using either the `fitted` or `predict` function.

In a MARSS model, the expected value of $\tilde{y}$ has the same interpretation but the calculation of the expected value involves both the estimated parameters and the expected value of $\mathbf{x}$:

$$\text{E}[\tilde{y}_t] = \hat{\mathbf{Z}}\,\text{E}[\mathbf{x}_t] + \hat{\mathbf{a}} \tag{1.6}$$

where the expectation is conditioned on the data ($\mathbf{y}$). You can get the expected value of $\tilde{y}$ from a MARSS fit using either the `fitted` or `predict` function:

```
#show example
```

Often it is the case that the objective of the analysis is to estimate $\mathbf{x}$ and it is its expected value that is desired.

Show standard error of $\mathbf{x}$. That makes sense. We don't know what $\mathbf{x}$ is.

We can also the show the stand. error for functions of $\mathbf{x}$. We could show the standard error of $\hat{\mathbf{Z}}\text{E}[\mathbf{x}_t] + \hat{\mathbf{a}}$. That however is NOT the variability of $\text{E}[\tilde{y}]$. If we just collected a bunch of new data for the same time period, the $\mathbf{x}$ stays the same. The s.e. reflects our uncertainty but the $\mathbf{x}$ is not changing. $\text{E}[\tilde{y}] = \mathbf{Z}\mathbf{x} + \mathbf{u}$ and uncertainty about that is due to our uncertainty in both $\mathbf{x}$ and the parameters. We are uncertain about $\mathbf{x}$ in the same way as we are uncertain about the red line. Confidence interval not standard error. The standard error of $\mathbf{y}$ is from $\mathbf{R}$. E(param estimate) have standard errors and $\mathbf{x}_t$ has a standard error.

Doesn't make sense to use that to compute the standard error of $\mathbf{y}$ unless we wanted to show the stand. error if the whole process were run again or run forward. We assume the parameters are at their estimated values and run forward. But why would the new process be governed by the estimated parameters?

Expected value of $\mathbf{y}$ if we ran the process over and over and each time generated a new $\mathbf{x}$ using the estimated parameters. ??? $\mathbf{x}_t = \hat{\mathbf{B}}\mathbf{x}_{t-1} +$

Forecasting using the estimated values. Yes, standard thing to do but keep in mind that it the prediction intervals will be too narrow.

why show variability in the E(y)? You are uncertain about the parameters. s.e. of the states has nothing to do with the uncertainty in the parameters. You are uncertain about the states even if you are certain about the

1.5 Confidence intervals and prediction intervals for MARSS models

Fig. 1.7. Distribution of α and β estimates from parametrically bootstrapping using the fit to $\mathbf{y}_j$. You can see that they are approximately multivariate normal even for $n = 10$ (so not n large). Note, these estimates are from lm() which is using least-squares estimation, but the parameter estimates are the same as the maximum-likelihood estiamates for this problem (linear regression with Gaussian errors).

1.6 Functions used in the chapter

1.6.1 Analytical CIs

This function computes the analytical CIs with the Fisher information matrix using the known (true) σ if passed in or estimated σ if not.

```
CI.analytical.true.s2=function(dat, x, sigma, alp=0.05){
  fit=lm(y~x, data=dat)
  Xi = cbind(1, dat$x)
  XiXi.inv = solve(t(Xi)%*%Xi)
  theta = matrix(coef(fit), ncol=1)
  if(is.null(sigma)){
    sigma = sqrt(sum(fit$residual^2)/fit.df)
  }
  Sigma = sigma^2*XiXi.inv
  X = cbind(1, x)
  EyX = X%*%theta
  #the analytical CI
  CI = EyX + qnorm(c(alp/2,1-alp/2)) * sqrt(X%*%Sigma%*%t(X))
  c(fit=EyX, lwr=CI[1], upr=CI[2])
}
```

This function computes the analytical CIs with the observed Fisher information matrix using the estimated σ. This is biased.

```
CI.analytical.est.s2=function(dat, x, alp=0.05){
  fit=lm(y~x, data=dat)
  Xi = cbind(1, dat$x)
  XiXi.inv = solve(t(Xi)%*%Xi)
  theta = matrix(coef(fit), ncol=1)
  fit.df=fit$df.residual
```

```
  sigma = sqrt(sum(fit$residual^2)/fit.df)
  Sigma = sigma^2*XiXi.inv
  X = cbind(1, x)
  EyX = X%*%theta
  #the analytical CI
  CI = EyX + qnorm(c(alp/2,1-alp/2)) * sqrt(X%*%Sigma%*%t(X))
  c(fit=EyX, lwr=CI[1], upr=CI[2])
}
```

This function computes analytical CIs with the observed Fisher information matrix and corrects using the quantiles for the t-distribution.

```
CI.analytical.corrected=function(dat, x, alp=0.05){
  fit=lm(y~x, data=dat)
  fit.df = fit$df.residual
  Xi = cbind(1, dat$x)
  XiXi.inv = solve(t(Xi)%*%Xi)
  hat.sigma = sqrt(sum(fit$residual^2)/fit.df)
  hat.Sigma = hat.sigma^2*XiXi.inv
  hat.theta = matrix(coef(fit), ncol=1)
  X = cbind(1, x)
  EyX = X%*%hat.theta
  CI = EyX + qt(c(alp/2,1-alp/2), df=fit.df) * sqrt(X%*%hat.Sigma%*%t(X))
  c(fit=EyX, lwr=CI[1], upr=CI[2])
}
```

1.6.2 Numerical estimation of the Hessian of the negative log-likelihood function

The following function numerically estimates the Hessian of the negative of the log-likelihood functions and inverts that to give us an estimate of Σ:

```
#Get the MLEs and MLE Sigma
hessian.parm=function(dat){
  library(MASS)
  NLL <- function(parm, y=NULL, x=NULL) {
    resids = y - x * parm[2] - parm[1]
    dresids = suppressWarnings(dnorm(resids, 0, parm[3], log = TRUE))
    -sum(dresids)
  }
  fit=lm(y~x, data=dat)
  sigma = sqrt(sum(fit$residual^2)/fit$df.residual)
  alpha=coef(fit)[1]
  beta=coef(fit)[2]
  pars=c(alpha, beta, sigma)
  names(pars)=c("alpha","beta","sigma")
```

```
  fit.tmp=optim(pars, NLL, y=dat$y, x=dat$x, hessian=TRUE)
  parSigma = solve(fit.tmp$hessian)
  parMean = matrix(fit.tmp$par, ncol=1)
  names(parMean)=c("alpha","beta","sigma")

  list(parMean=parMean, parSigma=parSigma)
}
```

1.6.3 Functions for producing bootstrapped parameter estimates

This function produces parameter estimates by drawing from the estimated Σ:

```
hessian.boot=function(dat, nboot=1000){
  hes=hessian.parm(dat)
  #generate alpah and beta from Sigma
  boot.params = mvrnorm(nboot, mu = hes$parMean, Sigma = hes$parSigma)
  colnames(boot.params)=c("alpha","beta","sigma")
  boot.params
}
```

This function produces parameter estimates by parametric bootstrapping:

```
parametric.boot=function(dat, nboot=1000){
  #first fit model to data
  fit=lm(y~x, data=dat)
  #x's at which to generate new data
  x=dat$x
  #matrix to store the estimates
  boot.params=matrix(NA,nboot,3)
  sigma = sqrt(sum(fit$residual^2)/fit$df.residual)
  alpha=coef(fit)[1]
  beta=coef(fit)[2]
  for(i in 1:nboot){
    y=alpha + beta*x + rnorm(nrow(dat),0,sigma)
    tmp.fit=lm(y~x)
    boot.params[i,]=c(coef(tmp.fit),
                      sqrt(sum(tmp.fit$residual^2)/tmp.fit$df.residual))
  }
  colnames(boot.params)=c("alpha","beta","sigma")
  boot.params
}
parametric.boot.cis=function(dat, x, nboot=1000){
  boot.params=parametric.boot(dat, nboot=nboot)
  boot.CI(boot.params, x)
}
```

This function produces parameter estimates by resampling from the residuals:

```
residuals.boot=function(dat, nboot=1000){
  fit=lm(y~x, data=dat)
  resids=residuals(fit)
  alpha=coef(fit)[1]
  beta=coef(fit)[2]
  boot.params=matrix(NA,nboot,3)
  n = nrow(dat) #number of data points
  for(i in 1:nboot){
    tmp = sample(n, replace=TRUE)
    tmp.y=alpha + beta*dat$x + resids[tmp]
    tmp.fit=lm(tmp.y~dat$x)
    boot.params[i,]=c(coef(tmp.fit),
                      sqrt(sum(tmp.fit$residual^2)/tmp.fit$df.residual))
  }
  colnames(boot.params)=c("alpha","beta","sigma")
  boot.params
}
```

This function produces parameter estimates by resampling the data:

```
resampling.boot=function(dat, nboot=1000){
  boot.params=matrix(NA,nboot,3)
  n = nrow(dat) #number of data points
  for(i in 1:nboot){
    #sample with replacement
    tmp = sample(n, replace=TRUE)
    #tmp.fit is the fit to this bootstrapped data
    tmp.fit=lm(y~x, data=dat, subset=tmp)
    boot.params[i,]=c(
      coef(tmp.fit),
      sqrt(sum(tmp.fit$residuals^2)/tmp.fit$df.residual))
  }
  colnames(boot.params)=c("alpha","beta","sigma")
  boot.params
}
```

1.6.4 Functions for producing CIs from bootstrapped parameter estimates

This function produces CIs at x from a set of bootstrapped parameter estimates:

```
boot.CI=function(boot.params, x, alp=0.05){
  CIs=apply(
```

```
    boot.params[,c("alpha","beta"),drop=FALSE]%*%rbind(1,x),
    2,quantile, c(0.5, alp/2, 1-alp/2) )
  colnames(CIs)=x
  t(CIs) #to look like predict output
}
```

This function is then combined with the functions for producing bootstrapped parameter estimates to produces CIs for the different bootstrap methods:

```
hessian.boot.cis=function(dat, x, nboot=1000){
  boot.params=hessian.boot(dat, nboot=nboot)
  boot.CI(boot.params, x)
}
parametric.boot.cis=function(dat, x, nboot=1000){
  boot.params=parametric.boot(dat, nboot=nboot)
  boot.CI(boot.params, x)
}
residuals.boot.cis=function(dat, x, nboot=1000){
  boot.params=residuals.boot(dat, nboot=nboot)
  boot.CI(boot.params, x)
}
resampling.boot.cis=function(dat, x, nboot=5000){
  boot.params=resampling.boot(dat, nboot=nboot)
  boot.CI(boot.params, x)
}
```

1.6.5 Estimating the correction for a CI via bootstrapping

ci.adj

```
function(dat, x, nboot1=5000, nboot2=1000, type="hessian"){
  #x is the x where the ci is computed; one value
  #type is analytical, hessian, parametric or residuals
  nsamp=nrow(dat)
  fit=lm(y~x, data=dat)
  sigma = sqrt(sum(fit$residuals^2)/fit$df.residual)
  alpha=coef(fit)[1]
  beta=coef(fit)[2]
  #this is the correct width of the 95\% CI
  #for a model with the parameters estimated from dat
  ci=parametric.boot.cis(dat,x,nboot=nboot1)[2:3]
  ci.width=ci[2]-ci[1]
  #ci.width.boot will hold a set of bootstrapped CIs
  ci.width.boot=rep(NA,nboot2)
  for(i in 1:nboot2){
```

```
    if(type %in% c("parametric", "hessian", "analytical")){
      y=alpha + beta*dat$x + rnorm(nsamp,0,sigma)
      tmp.dat=data.frame(x=dat$x, y=y)
      if(type=="analytical") tmp.ci=CI.analytical.est.s2(tmp.dat,x)[2:3]
      if(type=="parametric") tmp.ci=parametric.boot.cis(tmp.dat,x,nboot1)[2:3]
      if(type=="hessian") tmp.ci=hessian.boot.cis(tmp.dat,x,nboot1)[2:3]
    }
    if(type == "residuals"){
      y=alpha + beta*dat$x + sample(fit$residuals, nsamp)
      tmp.ci=residuals.boot.cis(tmp.dat,x,nboot1)[2:3]
    }
    if(type == "resampling"){
      tmp.dat = dat[sample(nsamp, replace=TRUE),]
      tmp.ci=resampling.boot.cis(tmp.dat,x,nboot1)[2:3]
    }
    ci.width.boot[i]=tmp.ci[2]-tmp.ci[1]
  }
  #Trim to deal with random ouliers
  mean(ci.width/ci.width.boot, trim=.1)
}
```

References

Index