Fitting Process Models
Part 1: Fitting a regression (as if that were a process model)

Carsten F. Dormann[1]

[1]Biometry & Environmental System Analysis, University of Freiburg, Germany
carsten.dormann@biom.uni-freiburg.de

July 12, 2019

Fitting a process model to data is, fundamentally, similar to fitting a linear regression: We need to specify the relationship (for process model read "processes") between the predictors ("input") and the observed response ("output"); the likelihood for these data; and then we use an optimiser to find the best parameters ("constants"). In practice, however, optimisation may have some disadvantages, so we may want to resort to a Bayesian calibration framework. I here demonstrate how these steps work, for a simple linear regression. That provides us with a framework to use in the next step, when we fit an actual process model.

Contents

1 Function and distribution

A process model really is only a (complicated) function, $f(x)$. So we can also take a simple function, to start with:

$$f(x) = ax + b \tag{1}$$

It is customary to write vectors in bold, so that this becomes $f(\mathbf{x}) = a\mathbf{x} + b$ for real data sets. It also makes clear that a and b are scalar and in this case the stuff we want to estimate from the data.

Data? Right.

In a regression, we have a response, $\mathbf{y}$, and one or more predictors, $\mathbf{x}$. And these are linked, in the regression, by a distribution $\mathcal{D}$, which we assume when we employ the GLM:

$$\mathbf{y} \sim \mathcal{D}(\theta = g(a\mathbf{x} + b), \xi = h(c)), \tag{2}$$

where θ and (depending on the distribution) ξ are distribution parameters (such as shape, scale, location, spread and whatever their names are), and $g(.)$ and $h(.)$ are possible link functions.

For the normal distribution, this then looks like this:

$$\mathbf{y} \sim \mathcal{N}(\mu = a\mathbf{x} + b, \sigma = c). \tag{3}$$

Let's take a simple example:

```
data(trees)
attach(trees)
# plot(Volume ~ Girth) # see later
fm <- lm(Volume ~ Girth)
summary(fm)
```

```
Call:
lm(formula = Volume ~ Girth)

Residuals:
   Min     1Q Median     3Q    Max
-8.065 -3.107  0.152  3.495  9.587

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -36.9435     3.3651  -10.98 7.62e-12 ***
Girth         5.0659     0.2474   20.48  < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 4.252 on 29 degrees of freedom
Multiple R^2:  0.9353,   Adjusted R^2:  0.9331
F-statistic: 419.4 on 1 and 29 DF,  p-value: < 2.2e-16
```

Our model, in this case coded in R, looks like this:

```
model <- function(par) {
    # our regression model
    par[1] * Girth + par[2]
}
```

When we plug in some values for par, we get an output; that's what a model does: converting input to output.

To fit a model to data, we need a score function that tells the computer what to minimise or maximise. Typically this means we define a so-called *metric*[1], which quantifies the distance between the model predictions and the data, and minimise this distance. The best-known measure for regression fit is, of course, the likelihood (even though it is not always metric (the adjective); see footnote).

Let's define the likelihood as an R-function to be minimised:

```
optfun1 <- function(params) {
    mu <- model(params)
    sigma <- exp(params[3])  # trick to avoid negative or 0 SD!!
    # now compute the likelihood of a normal distribution:
    sum(dnorm(Volume, mean = mu, sd = sigma, log = T))
}
```

Next, we have to set up and run the optimiser for our problem:

[1]Mathematically, a metric is a function, typically called "distance", that maps two same-sized sets to a value between 0 and ∞. Or, in mathy writing, which we really should get familiar with at some point:

$$d : X \times X \to [0, \infty) \tag{4}$$

Any metric must meet four conditions (non-negativity, identity of indiscernibles, symmetry and subadditivity, see `https://en.wikipedia.org/wiki/Metric_(mathematics)`). Interestingly, symmetry, i.e. $d(x, y) = d(y, x)$ is *not* met by the Kullbach-Leibler divergence, which hence is not called a "distance" and denoted by a different writing style ($D_{KL}(x \mid\mid y)$, rather than $d_{\text{RMSE}}(x, y)$). Note also that the likelihood of symmetric distributions, such as the normal, Cauchy or t are metric, while those of asymmetric and discrete distributions are not. Remember that KL-divergence and likelihood are closely linked, particular when used as target for minimisation/maximisation (`https://wiseodd.github.io/techblog/2017/01/26/kl-mle`).

```
opt ← optim(fn = optfun1, par = c(1, 2, 3), control = list(fnscale = -1),
    hessian = T)
opt
```

```
$par
[1]   5.065789 -36.943216   1.414206

$value
[1] -87.82236

$counts
function gradient
     196       NA

$convergence
[1] 0

$message
NULL

$hessian
             [,1]          [,2]          [,3]
[1,] -339.06819412 -24.275096935  -0.033733036
[2,]  -24.27509694  -1.832305834  -0.002369193
[3,]   -0.03373304  -0.002369193 -61.979606848
```

This output means: the algorithm converged (indicated stupidly by a "0"), the parameter estimates are 5.07 for the slope, -36.9 for the intercept and $e^{1.41} = 4.1$ for the standard deviation. Their standard errors can be extracted by inverting the negative hessian and taking the square-root of the diagonal, i.e.

```
sqrt(diag(solve(-opt$hessian)))
```

```
[1] 0.2393035 3.2553197 0.1270210
```

which compares well to the GLM-output.

1.1 *Other metrics*

As a very brief excursion, we'll look at two alternative metrics, the mean absolute difference (MAD) and the root mean squared error (RMSE). Note that both MAD and RMSE want to be minimised, unlike our likelihood. So we have to remove the $-$ from the 'fnscale'-option.

```
dmad ← function(par) {
    # mean absolute difference median is often used to quantify error, but
    # NOT useful for optimisation: why not?
    mean(abs(model(par) - Volume))
}
# minimise!
optim(par = c(1, 1, 1), fn = dmad, control = list(fnscale = 1))
```

```
$par
[1]   4.560606 -30.590908  29.570993

$value
[1] 3.197703

$counts
function gradient
     188       NA

$convergence
[1] 0

$message
NULL
```

Works fine, but the optimal parameters are slightly different to the maximum likelihood estimate (MLE).

```
drmse ← function(par) {
    # root mean squared error
    sqrt(mean((model(par) − Volume)^2))
}
optim(par = c(1, 1, 1), fn = drmse, control = list(fnscale = 1))
```

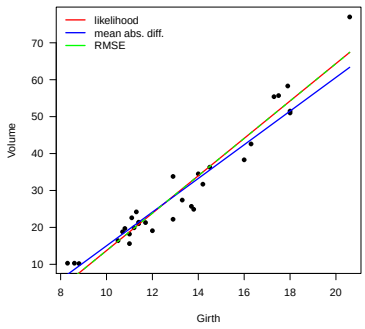
```
$par
[1]   5.065957 -36.943443  53.122623

$value
[1] 4.11254

$counts
function gradient
     140       NA

$convergence
[1] 0

$message
NULL
```

Again, works smoothly, and parameter estimates are very similar to the MLE.

1.2 *Comparing fits*

Let's put them all together next to the figures:

```
plot(Volume ~ Girth, pch = 16, las = 1)
curve(5.065371 * x − 36.935176, add = T, col = "red", lwd = 2)
curve(4.560606 * x − 30.590908, add = T, col = "blue", lwd = 2)
curve(5.065957 * x − 36.943443, add = T, col = "green", lty = 2, lwd = 2)
legend("topleft", legend = c("likelihood", "mean abs. diff.", "RMSE"),
    col = c("red", "blue", "green"), lwd = 2, bty = "n")
```



Both MLE and RMSE are (actually identical and) less accepting of extreme deviations than MAD. The MAD-line is not wrong! It merely is optimal for a different score function; and who am I to say what the correct score function is for *your* questions?

2 *Generalisation: Going Bayesian*

So, in principle, we can use an optimiser to get the best-fitting parameters for our model. That's what people in hydrology and meteorology have done, and are doing, all the time, and that's perfectly fine. Well, most of the time.

There are two issues that could be improved upon:

1. Using guesses, or rather, prior knowledge about the expected parameters, e.g. from previous analyses.

2. Relaxing the (implicit) assumption(s) behind the Hessian, such as symmetry at the optimum and smooth (quadratic) slopes.[2]

The first issue calls for a Bayesian solution, as prior knowledge is one of its selling points (i.e. the explicit mathematical inclusion of priors in the computation of the solution).

The second issue calls for a more, well, brute-force approach, which represents the optimum more realistically.[3] The typical algorithm for such problems is the MCMC (or indeed the bootstrap). Since Bayesian statistics often is implemented as MCMC, we can feed two birds with one scone.[4]

There are probably dozens of ways to implement fitting process models in a Bayesian framework. Here, we go with **BayesianTools** by Florian Hartig, which is great, although it requires us to think in exactly his way (small price to pay).

```
library(BayesianTools)
`?`(createBayesianSetup)
```

We have to construct a `bayesianSetup`, a set of objects that defines the entire fitting process, incl. priors, a "sampler" for randomly drawing from the prior, and the likelihood.

2.1 *BayesianTools: the likelihood*

We take again our model as starting point

```
model <- function(par) {
    # our regression model
    par[1] * Girth + par[2]
}
```

and define the likelihood, just like above for the optimisation:

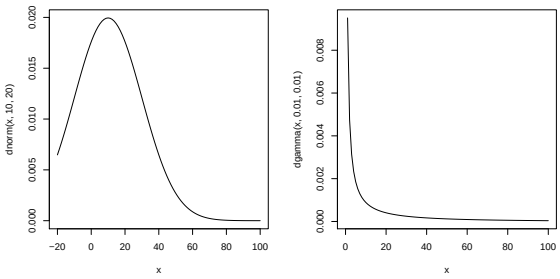
```
ell <- function(par) {
    # the likelihood
    sum(dnorm(Volume, mean = model(par), sd = exp(par[3]), log = T))
    # sum(log( abs( (Volume - model(par)) + 1e-8)))
}
```

2.2 *BayesianTools: the prior setup*

The prior definition requires a bit of care and thought. A probability density function is required (unless we use `createUniformPrior` or `createTruncatedNormalPrior` or `createBetaPrior`); it defines the distributions of the priors for each parameter in the model. Remember that the prior returns the log-likelihood of the current values of our parameters.

So, what would we expect for the slope, or the standard deviation? Maybe:

```
par(mfrow = c(1, 2), mar = c(5, 5, 1, 1))
curve(dnorm(x, 10, 20), -20, 100)
curve(dgamma(x, 0.01, 0.01), 0, 100)
```



[2] These assumptions typically lead to optimistic standard errors, which can be very inaccurate, as demonstrated here: `https://stackoverflow.com/questions/14581358/getting-standard-errors-on-fitted-parameters-using-the-optimize-leastsq-method-i`.

[3] Think of the optimum as the summit of a mountain. The Hessian implies a sugar-cone-like summit, which is rare. Ideally, our algorithm would be able to represent asymmetric, rough, cracky summits, too.

[4] Actually, this is one of the better PETA-replacements for "killing two birds with one stone". I am less fond of their other proposal (`https://www.peta.org/teachkind/lesson-plans-activities/animal-friendly-idioms/`), although "Feeding a fed horse." to replace "Flogging a dead horse." is also well-balanced.

We'll run with that and define a function, that returns the sum of the log-likelihoods for a given set of parameter values:

```
densPriors ← function(par) {
    d1 ← dnorm(par[1], mean = 10, sd = 10, log = T)  # what we would expect for the slope: pos
    d2 ← dnorm(par[2], mean = 0, sd = 10, log = T)  # what we would expect for the intercept
    d3 ← dgamma(exp(par[3]), shape = 0.01, scale = 0.01, log = T)  # the SD
    return(d1 + d2 + d3)
}
```

Now imagine a function that would draw n random values from these densities:

```
samplerPriors ← function(n = 1) {
    d1 ← rnorm(n, mean = 10, sd = 10)  # draw a random value from the prior of slope
    d2 ← rnorm(n, mean = 0, sd = 10)  # same for intercept
    d3 ← rgamma(n, shape = 0.01, scale = 0.01)  # and for the SD
    return(cbind(d1, d2, d3))
}
```

This function can be used to compute starting values for the MCMC.

In BayesianTools, we now have to put them together like so:

```
priors ← createPrior(density = densPriors, sampler = samplerPriors)  # no best/upper/lower are
```

2.3 *BayesianTools: the BayesianSetup*

Now we pull likelihood and priors together, and specify how many cores we can allocate to this computation:

```
treesSetup ← createBayesianSetup(likelihood = ell, prior = priors, parallel = 3)
```

```
parallel function execution created with3cores.
```

```
[1] "Info is missing upper/lower/best. This can cause plotting and sensitivity analysis functions to fail. If yo
```

The function checks whether all information is proper and correct (which we could also do explicitly with `checkBayesianSetup(treesSetup)`, if we wanted). We can check the structure of this setup:

```
str(treesSetup)
```

```
List of 9
 $ prior       :List of 7
  ..$ density        :function (x)
  ..$ sampler        :function (n = NULL)
  ..$ lower          : NULL
  ..$ upper          : NULL
  ..$ best           : NULL
  ..$ originalDensity:function (par)
  .. ..- attr(*, "srcref")= 'srcref' int [1:8] 1 15 6 1 15 1 1 6
  .. .. ..- attr(*, "srcfile")=Classes 'srcfilecopy', 'srcfile' <environment: 0x7ff8a0f8c610>
  ..$ checkStart     :function (x = NULL, z = FALSE)
  ..- attr(*, "class")= chr "prior"
 $ likelihood  :List of 5
  ..$ density     :function (x, ...)
  ..$ sampler     : NULL
  ..$ cl          :List of 3
  .. ..$ :List of 3
  .. .. ..$ con : 'sockconn' int 5
  .. .. .. ..- attr(*, "conn_id")=<externalptr>
  .. .. ..$ host: chr "localhost"
  .. .. ..$ rank: int 1
  .. .. ..- attr(*, "class")= chr "SOCKnode"
  .. ..$ :List of 3
  .. .. ..$ con : 'sockconn' int 6
  .. .. .. ..- attr(*, "conn_id")=<externalptr>
  .. .. ..$ host: chr "localhost"
  .. .. ..$ rank: int 2
```

```
  .. .. ..- attr(*, "class")= chr "SOCKnode"
  .. ..$ :List of 3
  .. .. ..$ con : 'sockconn' int 7
  .. .. .. ..- attr(*, "conn_id")=<externalptr>
  .. .. ..$ host: chr "localhost"
  .. .. ..$ rank: int 3
  .. .. ..- attr(*, "class")= chr "SOCKnode"
  .. ..- attr(*, "class")= chr [1:2] "SOCKcluster" "cluster"
  ..$ pwLikelihood: logi FALSE
  ..$ parNames    : NULL
  ..- attr(*, "class")= chr "likelihood"
 $ posterior   :List of 1
  ..$ density:function (x, returnAll = F)
  ..- attr(*, "class")= chr "posterior"
 $ names       : chr [1:3] "par 1" "par 2" "par 3"
 $ numPars     : int 3
 $ model       : NULL
 $ parallel    : num 3
 $ pwLikelihood: logi FALSE
 $ info        :List of 8
  ..$ priorLower: NULL
  ..$ priorUpper: NULL
  ..$ priorBest : NULL
  ..$ plotLower : NULL
  ..$ plotUpper : NULL
  ..$ plotBest  : NULL
  ..$ parNames  : chr [1:3] "par 1" "par 2" "par 3"
  ..$ numPars   : int 3
 - attr(*, "class")= chr "BayesianSetup"
```

We could add lower and upper limits, but we don't have to:

```
# priors ← createPrior(density=densPriors, sampler=samplerPriors,
# lower=c(-50, -50, -50), upper=c(50, 50, 50))
```

2.4 *BayesianTools: run the algorithm*

With everything set up, we can now choose the sampler and its settings, in our case we start with the simplest option, the Metropolis algorithm, run for 5000 steps (invoking parallel will not work for this algorithm's implementation).

```
### run the calibration
settings ← list(iterations = 5000, nrChains = 3, parallel = T)  # not properly documented!
# settings of the sampler, e.g. ?Metropolis
fit ← runMCMC(bayesianSetup = treesSetup, sampler = "Metropolis", settings = settings)
```

```
BT runMCMC: Optimization finished, setting startValues to 5.02345321242382 -36.3688366913278 0.492291543325061
- Setting covariance to 0.00893016045365648 -0.118212015917592 -0.000150354012500905 -0.118212015917592 1.651090
```

```
runMCMC terminated after 0.951seconds
```

```
BT runMCMC: Optimization finished, setting startValues to 5.02480651859315 -36.3691613405764 0.496341018264447
- Setting covariance to 0.00900077761365977 -0.119150967582385 -0.000146113598291017 -0.119150967582385 1.664281
```

```
runMCMC terminated after 0.796seconds
```

```
BT runMCMC: Optimization finished, setting startValues to 5.02102085596067 -36.3279552095489 0.492780646369207
- Setting covariance to 0.00894023388241125 -0.118348231866561 -0.000159044029037816 -0.118348231866561 1.653017
```

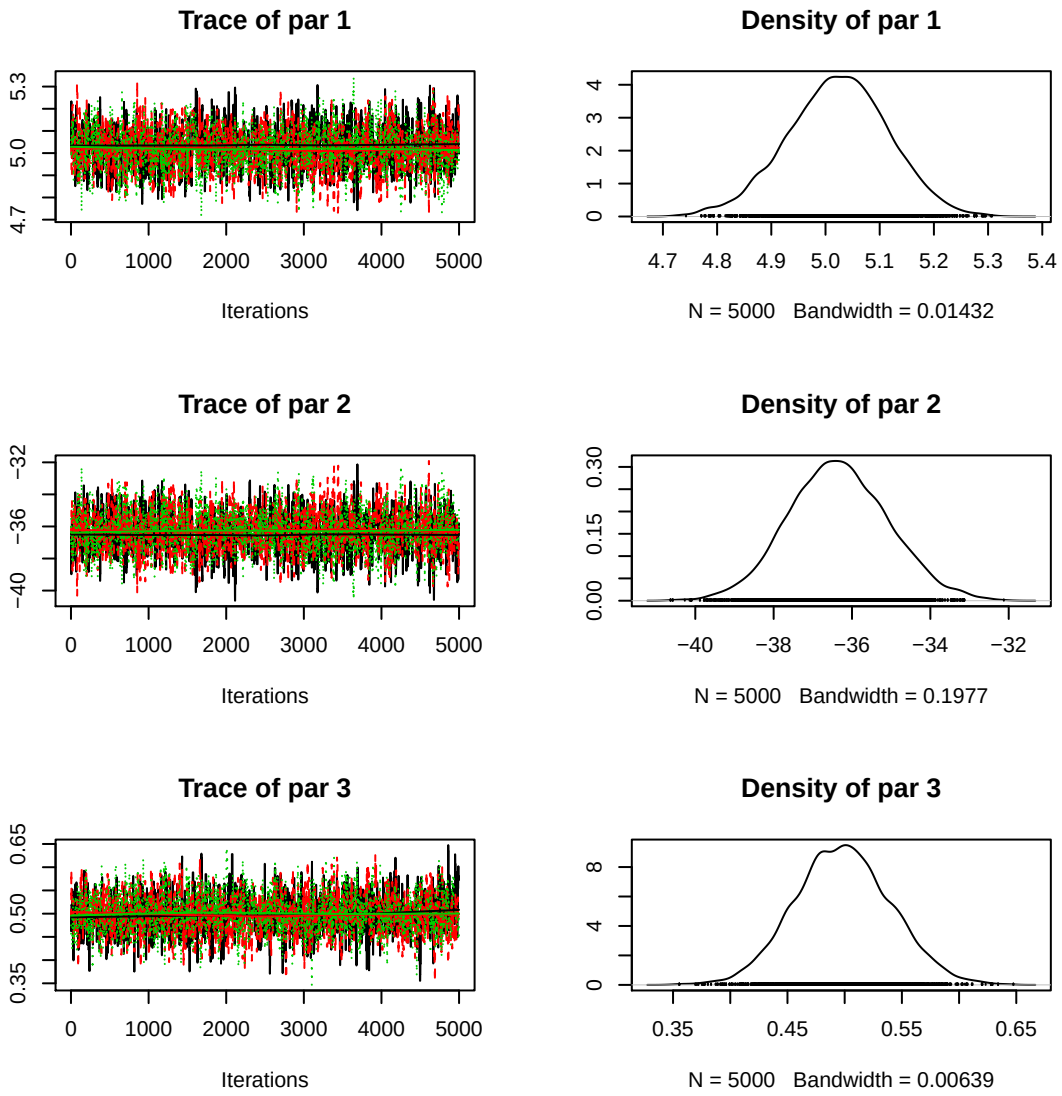
```
runMCMC terminated after 0.832000000000001seconds
```

The output was cut, as it is rather long, reporting on the progress of the MCMC sampling. We get:

```
summary(fit)
```

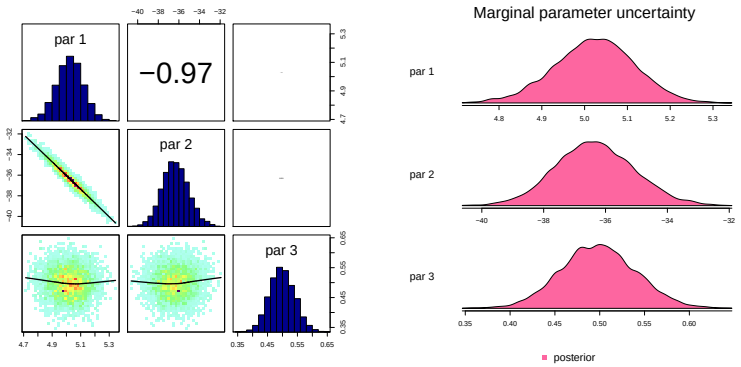
```
# # # # # # # # # # # # # # # # # # # # # # # # # # # #
## MCMC chain summary ##
# # # # # # # # # # # # # # # # # # # # # # # # # # # #

# MCMC sampler:  Metropolis
# Nr. Chains:  3
# Iterations per chain:  5000
# Rejection rate:  0.68
# Effective sample size:  1445
# Runtime:  2.579  sec.

# Parameters
        psf     MAP    2.5%  median   97.5%
par 1 1.016   5.021   4.833   5.025   5.202
par 2 1.015 -36.328 -38.796 -36.362 -33.767
par 3 1.001   0.493   0.418   0.498   0.582

## DIC:  286.77
## Convergence
 Gelman Rubin multivariate psrf:  1.016

## Correlations
       par 1  par 2  par 3
par 1  1.000 -0.973 -0.051
par 2 -0.973  1.000  0.051
par 3 -0.051  0.051  1.000
```

```
plot(fit)  # same as tracePlot(fit)
```



So we get the estimates for the parameters, the convergence diagnostics and, with the figures, a nice feeling for the mixing of the chains (excellent) and the posterior (a bit Matterhorn-like). We can now do all sorts of model diagnostics, and indeed here are a few:

```
correlationPlot(fit)
# gelmanDiagnostics(fit)
marginalPlot(fit)  # only plots parameters
```



2.5 *BayesianTools: plotting the model onto the data*

More interesting here is how to then plot the regression line and its credible interval from this fit. Here's how it works:

First, we draw a subset from the large number of parameter sets fitted in the MCMC:

```
samples ← getSample(fit, start = 1000, thin = 3)  # a matrix of values
# 4002 parameter sets, while 1000 would have done the job, too.
```

Next, we plug this into the model we fitted (first only one, as an example:

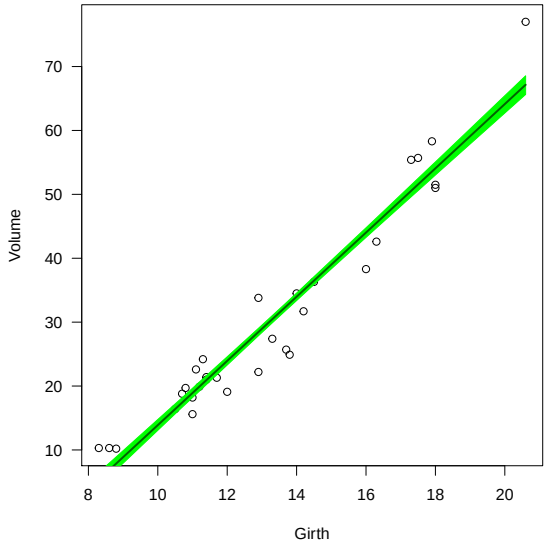
```
model(samples[1, ])  # the 31 fitted values for ONE set of parameters
```

```
 [1]  5.440187  6.935298  7.932039 16.404335 17.401076 17.899446 18.896187
 [8] 18.896187 19.394557 19.892927 20.391298 20.889668 20.889668 22.384779
[15] 23.879890 28.365223 28.365223 30.358705 32.352186 32.850556 33.847297
[22] 34.844038 36.339149 43.814704 45.309815 50.293519 51.290259 53.283741
[29] 53.782111 53.782111 66.739740
```

```
preds ← apply(samples, 1, model)  # for all 4002 parameter sets
```

Finally, we can plot the data and add the quantiles from our model "simulations" to it:

```
par(mar = c(5, 5, 1, 1))
plot(Volume ~ Girth, las = 1)
quants ← apply(preds, 1, quantile, c(0.025, 0.5, 0.975))  # CI and median
# lines(Girth, quants[1,], lwd=3, col='darkgreen') lines(Girth,
# quants[3,], lwd=3, col='darkgreen')
polygon(x = c(Girth, rev(Girth)), y = c(quants[3, ], rev(quants[1, ])),
    border = "green", col = "green")
lines(Girth, quants[2, ], lwd = 2, col = "darkgreen")
```



That's it. Remember that our function `model` could in principle be replaced by any model, including a process model. That's what we'll turn to in the next session!