A tutorial on the use of the model fitting engines JAGS, NIMBLE, Stan and TMB and on Do-it-yourself maximum likelihood (DIY-MLE)

Bayesian Intro Course,
UZH, Switzerland,
12–14 April 2021

Written by
Marc Kéry (Swiss Ornithological Institute) and
Ken F. Kellner (ESF, SUNY Syracuse)

1 Introduction

In this tutorial we will familiarize ourselves with the way in which we use JAGS (Plummer, 2003), NIMBLE (de Valpine et al., 2017; see also https://r-nimble.org/) and Stan (Carpenter et al., 2017; see also https://mc-stan.org/), all three for Bayesian inference, and TMB (Kristensen et al., 2016; see also https://www.admb-project.org/), for likelihood inference, for the fitting of parametric models. We also include an illustration of key features of a Bayesian analysis such as prior, likelihood, MCMC settings, initial values, updates, convergence and so on. In addition, we establish a unified format for the workflow of our analyses. Finally, we give a practical illustration of the hugely important estimation method known as maximum likelihood. Arguably, this is the most widely used statistical estimation method, but is not well understood among most nonstatisticians.

We will start with an analysis of one of the simplest possible models for a normal response–for what we here call the "model of the mean". This is a normal linear regression model with just an intercept in the deterministic part of the response. Another description of this model would be that we estimate the mean and the variance of a normal distribution, which we assume as a statistical model for a sample of measurements taken from a population. Our first example will deal with body mass of male peregrines (Fig. 1).

Figure 1: Male peregrine falcon (*Falco peregrinus*) wintering in the French Mediterranean, Sète, 2008 (Photo by Jean-Marc Delaunay).

Hence, the parametric statistical model assumed for our data will be

$$y_i \sim Normal(\mu, \sigma^2),$$

where y_i is the mass measurement made on the *i*-th male peregrine in our data set, the tilde/twiddle (~) can be read as 'is a draw from' or 'is considered to be distributed as' and μ and σ^2 are the two parameters of the Normal (or Gaussian) distribution: the mean and the variance. As we will see, the latter is often expressed at the scale of the standard deviation instead.

This algebraic expression is really important to understand, but it is only a shorthand for the probability density function of a Normal distribution. That is, this equation stands for the following:

$$f(y_i \mid \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{y_i - \mu}{\sigma}\right)^2}$$

This expression enables us to determine the probability density (broadly, *"the probability of observing a value of response Y in some infinitesimally small range of values around y*$_i$*"*) if we know the two parameters of the density, the mean μ and the standard deviation σ.

When we fit a parametric statistical model, we claim that we know exactly the stochastic mechanism that has produced the observed data. This mechanism is described as a probability density or a probability mass function. This then serves to express the joint probability of the entire data set, usually (i.e., under the assumption of independence) as a product of the probability density or mass of each single datum. The joint probability of the data set can then be "turned around" and used as a likelihood function to estimate the parameters of the model using either Bayesian methods or maximum likelihood estimation, both of which are 'likelihood-based' in this sense.

Here is a broad outline of this tutorial:

- First, we simulate a data set in R. This serves as both an explanation of the statistical model that we cover, as well as providing a data set for us to play with.
- After that, we start with using a canned, nonbayesian model fitting function in R to fit the model and to generate maximum likelihood estimates (MLEs). Specifically, we will be using the function `lm` for this. This actually uses the least-squares method, but for the model considered this produces estimates of the mean that are identical to the MLE (while the variance estimate is only very slightly different).
- Once we have obtained this first set of solutions (i.e., parameter estimates) from fitting a model using methods that we assume most of you will know and have used many times before, we will then fit the same model using Bayesian posterior inference with JAGS (Plummer, 2003), NIMBLE (de Valpine et al., 2017) and Stan (Carpenter et al., 2017). We think that it will be insightful for you to compare the Bayesian and the nonbayesian estimates and see for yourself how numerically similar they virtually always turn out to be when vague priors are specified in a Bayesian analysis. We hope that it will also be helpful and interesting for you to compare the working of, and the results among, the three Bayesian model-fitting engines.
- After the Bayesian model fitting we will produce what we call "do-it-yourself (DIY) MLEs" by defining the likelihood of our model explicitly as an R function and then maximizing it using another R function, `optim`. We are convinced that this will be an incredibly valuable exercise for your general statistical education, because we hope that it will really help to clarify to you what this crucial estimation method of maximum likelihood is all about (see also Bolker, 2008, for an important book that is written in exactly that spirit). Furthermore, we believe that you will then feel wonderfully empowered by actually being able to obtain your very own personal MLEs for a statistical model :) In addition, the DIY-MLE section will be insightful to better understand all the other methods of analysis. And finally, obtaining a much more "natural" relationship with maximum likelihood estimation is required for you to progress to the more advanced formats of using NIMBLE, Stan and also JAGS (for instance, when you must marginalize latent variables from the likelihood to gain speed; Yackulic et al, 2020), as well as for fitting any kind of statistical model in that very powerful frequentist software TMB, see next.
- Thus, last in the tutorial, you will see how the same model is fit using TMB (Kristensen et al., 2016), which is a very powerful piece of software for obtaining MLEs for even complex statistical models. Arguably, for a nonstatistician this is probably the most challenging software among the model-fitting engines considered here. But we believe that when you understand the DIY-MLEs and have seen the code for fitting a model in the other software here, then the step towards using TMB for model fitting will be a little less daunting (note also a recent "hybrid" between Stan and TMB; Monnahan & Kristensen, PlosOne, 2018).

Throughout the tutorial, we will make plenty of numerical comparisons between the estimates obtained by the different methods and software to increase your confidence in them and to emphasize that at least at some superficial level, you can view JAGS, NIMBLE, Stan and TMB just as very powerful "solvers" for fitting general statistical models in practice.

2 Data generation

But first, we need a data set. Remember that the R code to generate a data set can be viewed as just another, very explicit description of a parametric statistical model, using the R language. We believe that R code to describe a statistical model is an incredibly powerful explanation of a statistical model, especially to nonstatisticians (Kéry, 2010; Kéry & Royle, 2016, 2021).

Male peregrines in Western Europe weigh on average about 600 g and Monneret (2006) gives a range of 500–680 g. The assumption of a normal distribution of body mass implies a standard deviation of about 30 g. We will create a small and a large variant of our data.

```
set.seed(39)

# Generate two samples of body mass measurements of male peregrines
y10 <- rnorm(n = 10, mean = 600, sd = 30)       # Sample of 10 birds
y1000 <- rnorm(n = 1000, mean = 600, sd = 30)   # Sample of 1000 birds

# Save the data-generating values of the parameters for later comparisons
truth <- c(mean=600, sd=30)

# Plot data (Fig. 2)
xlim = c(450, 750)
par(mfrow = c(1, 2), mar = c(6, 6, 6, 3), cex.lab = 1.5, cex.axis = 1.5)
hist(y10, col = 'grey ', xlim = xlim, main = 'Body mass (g) of 10 male
peregrines')
hist(y1000, col = 'grey', xlim = xlim, main = ' Body mass (g) of 1000 male
peregrines')
```

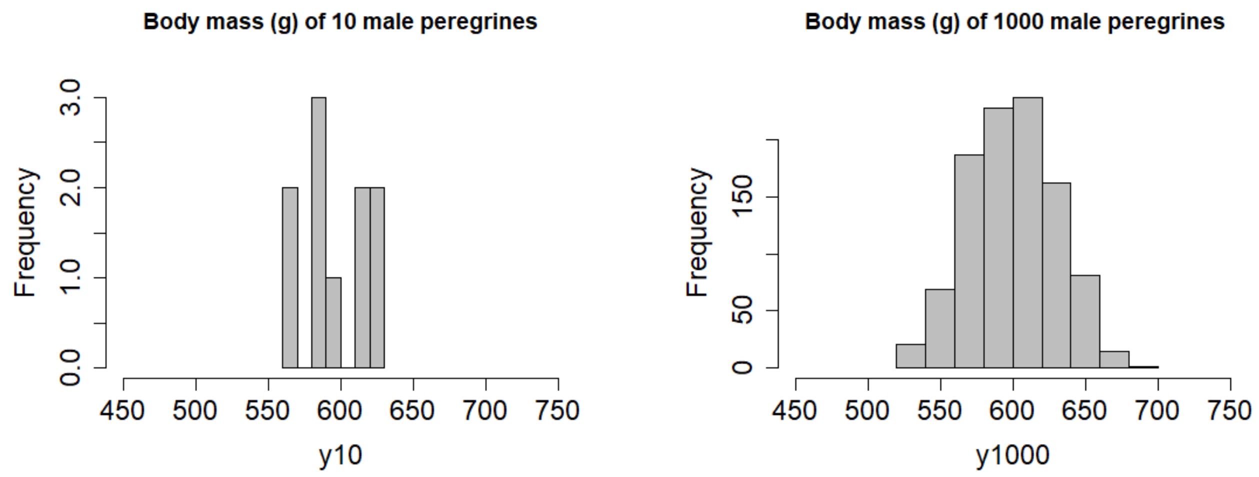

Figure 2: Histograms of the small (left) and the large (right) sample of weights of male peregrine falcons, which were all simulated as draws from a normal distribution with mean 600 and standard deviation 30.

Remember that when you want the same results as we show here, both in terms of the simulated data and to some degree also in the analysis later, then you must use the same seed as we do here (i.e., 39). However, it is also *extremely instructive* to execute the previous set of statements repeatedly to *experience* sampling error – the variation in one's data that stems from the fact that only part but not the whole of a variable population has been measured. Sampling error, or sampling variance, is something absolutely central to statistics and yet, it is among the most difficult concepts for ecologists to grasp, especially, since in practice we only ever observe a single sample from the distribution that characterizes this variation! It is astonishing to observe how *different* repeated realizations from the exact same random process can be – here, the sampling of 10 or 1000 male peregrines from an assumed infinite population of male peregrines. Also surprising is how far from normal the distribution of values in the smaller sample may look.

3 MLEs using canned functions in R

We can conduct a quick classical (or frequentist) analysis of this model using the linear regression facilities in R implemented in function `lm`. This actually uses the least-squares method, but for GLMs with normal response the parameter estimates in the mean correspond to the MLEs and the variance estimate closely matches the MLE of the variance, too.

```
summary(out3 <- lm(y10 ~ 1))          # small data set: not shown
summary(out3 <- lm(y1000 ~ 1))

Call:
lm(formula = y1000 ~ 1)
```

```
Residuals:
    Min      1Q  Median      3Q     Max 
-76.425 -21.581  -0.368  20.922  84.051 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept) 599.7482     0.9376   639.6   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 29.65 on 999 degrees of freedom
```

```
# Compare estimates with truth
lm_est <- c(coef(out3), sigma=sigma(out3))
cbind(truth=truth, lm=lm_est)

     truth        lm
mean   600 599.74824
sd      30  29.65055
```

We recognize well the estimates of the population mean and of the population SD, which is called the residual standard error in the regression output. We now compare these estimates with the posterior inference produced by our Bayesian model fitting engines, before we turn again to maximum likelihood inference, but in a very general and basic manner, first by writing our own likelihood function and optimizing it with the R function `optim` (our "DIY MLE method"), and then using program TMB.

4 Bayesian analysis with JAGS

Remember that you need to load the `jagsUI` package first, which is our interface of choice for running JAGS from R

```
library(jagsUI)
```

To run a model in JAGS from R using the `jagsUI` interface we use a function that is called `jags` (no surprise here ...). Here is its usage, with only the most commonly used arguments shown (do `?jags` to see all arguments).

```
  jags(data, inits, parameters.to.save, model.file,
     n.chains, n.adapt=NULL, n.iter, n.burnin=0, n.thin=1,
     parallel=FALSE)
```

Before we call `jags`, we will deal with each of these arguments which mean this:

- `data` is a list of the data for the analysis
- `inits` is a function that generates random starting values for some or all parameters
- `parameters.to.save` (which we abbreviate to 'params') is a list with the names of the parameters which we want to estimate

- The `model.file` is a text file that contains the model description in the BUGS language
- The next four arguments define what we call the MCMC settings and give the number of Markov chains run by JAGS, the length of the adaptive phase before the production run starts, the number of post-adaptation iterations that are discarded as a burnin and the thinning rate.
- `parallel` is a switch for turning on parallel computation (see below).

First, we package in a list the data that JAGS uses for the analysis. We do this by creating a "bundle" that contains both the data themselves and a count of the number of data points. We typically wrap the data bundle into a call to `str`, since you can much better understand the model descriptions for JAGS, NIMBLE, Stan and TMB, when you first see this useful concise overview of the data.

```
# Bundle and summarize data (here the larger variant)
str(dataList <- list(mass = y1000, n = length(y1000)))

List of 2
 $ mass: num [1:1000] 603 598 621 577 584 ...
 $ n   : int 1000
```

Next, we create the text file that contains the BUGS language description of the model that we want to fit. Executing the following block of code will write into our R working directory a text file named “`model4.txt`“ which contains the BUGS description of the model. Perhaps a little confusingly, this code block contains pieces that are R code and pieces that are BUGS language, so we mark the BUGS code part with a grey square. It is also useful at the beginning if you go into the R working directory (which you can identify by typing into R `getwd()`) and inspect the text file thus created.

Basically, in the BUGS model we define the likelihood of the model and, since we're conducting a Bayesian analysis, we must also define the priors. We do this by defining 'quantities', also called nodes, which are data, parameters, missing values, predictions and so forth, along with relationships between these nodes. Relations can be either deterministic or stochastic. We denote the former by the arrow sign, or assignment operator, and the latter by the twiddle, or tilde. A twiddle always defines the quantity on its left as a draw from the distribution on the right.

```
# Write JAGS model file
cat(file="model4.txt", "          # This code line is R
model {                           # Starting here, we have BUGS code
# Priors
pop.mean ~ dunif(0, 5000)         # Normal parameterized by precision
precision <- 1 / pop.var          # Precision = 1/variance
pop.var <- pop.sd * pop.sd
pop.sd ~ dunif(0, 100)

# Likelihood
for(i in 1:n){
  mass[i] ~ dnorm(pop.mean, precision)
}
}                                 # This is the last line of BUGS code
")                                # ... and this is R again
```

One feature of a model written in the BUGS language is that the order of the components doesn't matter as long as you don't swap things in- or outside of a loop. Hence, we could define all priors after the likelihood or we could put just the line starting with 'precision' after the likelihood and define there a third block entitled 'Derived quantities' (as we will often do in later examples). A second feature of JAGS is that the normal distribution is defined in terms of the precision, which is the reciprocal of the variance. In our model, we aim for clarity by choosing a nice layout of the model code with section titles, which are hashed out as comments, and with additional comments on the right in some of the lines. Clean layout, titles and comments are all optional, but highly encouraged, since they all greatly increase the readability of computer code.

Next, we define a function that creates random starting values, the inits function. This function will then be executed once for each chain requested in the MCMC settings (see below), ensuring that chains start at different places. This is desirable for convergence assessment.

```
# Function to generate starting values
inits <- function()
  list (pop.mean = rnorm(1, 600), pop.sd = runif(1, 1, 30))
```

We must also tell JAGS for which parameters it should save the posterior draws. Let's say we want to obtain estimates of the variance also.

```
# Parameters monitored (= to be estimated)
params <- c("pop.mean", "pop.sd", "pop.var")
```

Then, the MCMC settings need to be selected.

```
# MCMC settings
na <- 1000          # Number of iterations in the adaptive phase
ni <- 12000         # Number of draws from the posterior (in each chain)
nb <- 2000          # Number of draws to discard as burn-in
nc <- 3             # Number of chains
nt <- 1             # Thinning rate (nt = 1 means we do not thin)
```

Now, we have completed all the preparations required to run the analysis. We call function `jags` to carry out the analysis in JAGS and afterwards to put the results into an R object which we call `out4`, because we're in section 4.

```
# Call JAGS (ART 1 min) and marvel at JAGS' progress bar
out4 <- jags(data = dataList, inits = inits, parameters.to.save = params,
model.file = "model5.4.txt", n.iter = ni, n.burnin = nb, n.chains = nc,
n.thin = nt, n.adapt = na, parallel = FALSE)
```

Executing JAGS with `parallel = FALSE` lets us see the wonderful progress bar, staring at which gives us a powerful sense of achievement. However, for any analysis that takes longer than just a couple of minutes, we will usually run JAGS in parallel. This will distribute each chain to its own core on your computer (and you can set the number of cores with an argument in the function). With three chains you will reduce the run time by a factor of about 2.5; not quite 3x, since there's some overhead. All that we need to run JAGS in parallel is to set the `parallel` 'switch' to TRUE.

```
# Call JAGS in parallel (ART <1 min) and check convergence
```

```
out4 <- jags(data = dataList, inits = inits, parameters.to.save = params,
model.file = "model5.4.txt", n.iter = ni, n.burnin = nb, n.chains = nc,
n.thin = nt, n.adapt = na, parallel = TRUE)
par(mfrow=c(2, 2)); jagsUI::traceplot(out4)          # Produce Fig. 3
```

The very first thing that we should always do after any MCMC algorithm has finished is to assess whether the chains have converged. Most of all, we do this visually by trace plots (Fig. 3). Our trace plots oscillate around a horizontal average level, without any long or even short-term trends being visible, and this clearly suggests that the chains have converged. This impression is confirmed also by the value of the Brooks-Gelman-Rubin (BGR) or Rhat statistic. This is a convergence test statistic that is related to the F-test in a one-way analysis of variance testing for a chain effect. At convergence, there should no longer be any chain effect and all the variance should be within rather than between the chains, resulting in an Rhat of 1. For better or worse, a common rule of thumb takes 1.1 as a threshold for the acceptance of chain convergence. Rhat is useful for a quick screening of convergence in models with many parameters, for instance by producing a plot of the values of Rhat. However, we should always visually skim trace plots for at least the main parameters in the model, because very rarely the Rhat as computed in JAGS is unable to pick up non-convergence in some pathological cases.

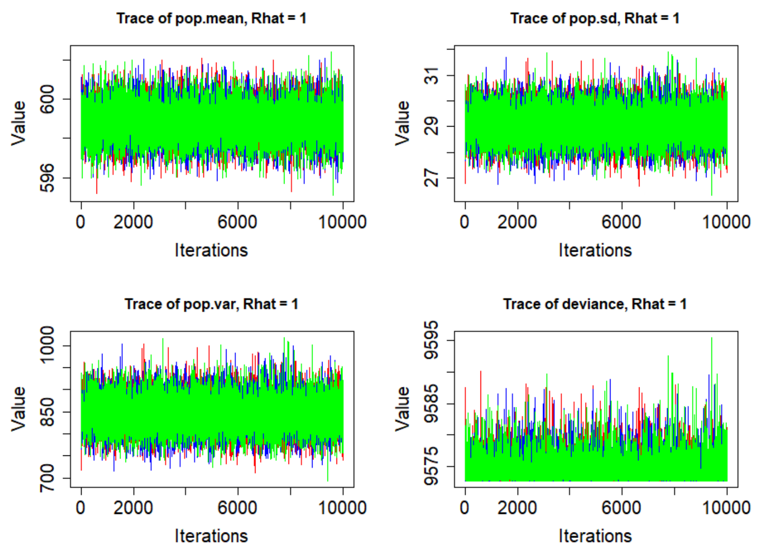

Figure 3: Trace plots for the parameters in the model of the mean, including the variance as a derived quantity and the deviance. In the header of each plot we see the value of the Brooks-Gelman-Rubin (BGR) convergence statistic, called Rhat here, which at convergence is near to 1.

Only once we have assessed convergence do we look at a summary of the posterior distributions.

```
print(out4, 3) # Produce a summary of the fitted model object
print(out4$summary, 3)     # Same, but only main body of table
```

```
JAGS output for model 'model5.4.txt', generated by jagsUI.
Estimates based on 3 chains of 12000 iterations,
adaptation = 1000 iterations (sufficient),
burn-in = 2000 iterations and thin rate = 1,
yielding 30000 total samples from the joint posterior.
```

```
MCMC ran in parallel for 0.202 minutes at time 2021-04-07 08:27:42.

             mean     sd     2.5%      50%    97.5% overlap0 f Rhat n.eff
pop.mean  599.765  0.950  597.921  599.764  601.637    FALSE 1    1 30000
pop.sd     29.680  0.670   28.406   29.666   31.032    FALSE 1    1 30000
pop.var   881.355 39.849  806.908  880.055  962.955    FALSE 1    1 30000
deviance 9617.882  2.069 9615.890 9617.262 9623.444    FALSE 1    1 30000

Successful convergence based on Rhat values (all < 1.1).
Rhat is the potential scale reduction factor (at convergence, Rhat=1).
For each parameter, n.eff is a crude measure of effective sample size.

overlap0 checks if 0 falls in the parameter's 95% credible interval.
f is the proportion of the posterior with the same sign as the mean;
i.e., our confidence that the parameter is positive or negative.

DIC info: (pD = var(deviance)/2)
pD = 2.1 and DIC = 9620.022
DIC is an estimate of expected predictive error (lower is better).
```

You see an output in three parts, where the header gives some general information about the MCMC settings and about the timing of the run. The latter becomes important when your models take hours to run. Then, you will want to know how many iterations you can run overnight so that you have some results to study the next morning, or over the weekend to have something on Monday morning to chew on.

The middle part is the crucial part: it gives sample statistics for the draws produced by the MCMC random-number-generator for each estimated quantity. Typically, we take the mean as a point estimate, the posterior sd as a quantity akin to the standard error and the 2.5% and 97.5% percentiles as a 95% credible interval. For skewed posterior distributions we should better report the posterior median or the mode – actually, it is the posterior mode which numerically corresponds to the MLEs. Here, mean and median are almost identical, suggesting a symmetric posterior distribution for the population mean, though a little less so for the population variance. We will see below how we can access the posterior draws and produce plots of the posterior distributions.

The columns with heading 'overlap0' and 'f' are explained in the footer of the output and can be used to conduct a type of Bayesian significance test (see later). The column with Rhat has just been explained. The final column is the effective sample size from the posterior, which is an estimate of the size of a hypothetical independent sample that contains the same amount of information as the dependent sample produced by our MCMC algorithm. Increasing serial autocorrelation in the chain output will reduce the size of `n.eff` relative to the nominal sample size given by the MCMC settings. To minimize the contribution of simulation error (also called MC error) to the parameter estimates, we want large effective sample sizes, especially when we are interested in the credible intervals.

The footer of this output gives some explanations and also includes the value of the Bayesian deviance information criterion or DIC (Spiegelhalter et al. 2002), which is a Bayesian analogue to the frequentist Akaike information criterion (AIC; Akaike 197x) and can be used for model selection. As the AIC, the DIC is computed as a deviance that is penalized by model complexity, where pD is the estimated number of parameters. For non-hierarchical models such as all GLMs, the DIC is useful, but it should in general not be used for most hierarchical models.

So, we have now achieved a major feat: we have obtained the first parameters estimates for a 'real' model fit with JAGS. We might summarize our inferences in a paper as follows: *"We estimate the mean body mass of male peregrines at 599.8 g (posterior mean), with a 95% Bayesian credible interval* *of 597.9–601.6 g. The standard deviation was estimated at 29.7 g, with a 95% CRI of the standard deviation of 28.4–31.0 g".*

But actually, the object `out4` produced by `jagsUI` based on the posterior draws produced by JAGS contains *a lot* more information. We can see this by listing all objects contained within `out4` by typing `names(out4)`.

```
names(out4)

 [1] "sims.list"   "mean"        "sd"          "q2.5"
 [5] "q25"         "q50"         "q75"         "q97.5"
 [9] "overlap0"    "f"           "Rhat"        "n.eff"
[13] "pD"          "DIC"         "summary"     "samples"
[17] "modfile"     "model"       "parameters"  "mcmc.info"
[21] "run.date"    "parallel"    "bugs.format" "calc.DIC"
```

You can look into any of these objects by typing their name, which will usually print their full content, or by applying some summarizing function such as `summary`, `names` or `str` on them. (Again, when not sure what an R function does, just type `?names` or `?str` into your R console.) One of our favourite R commands is `str`, which we now use to provide is with an overview of the R object `out4`, which the function `jags` in `jagsUI` has created in our R workspace using the results of the analysis conducted in JAGS.

```
str(out4)           # Full resolution of overview (not shown)
str(out4, 1)        # Reduced resolution of overview of out4

List of 24
 $ sims.list  :List of 4
 $ mean       :List of 4
 $ sd         :List of 4
 $ q2.5       :List of 4
 $ q25        :List of 4
 $ q50        :List of 4
 $ q75        :List of 4
 $ q97.5      :List of 4
 $ overlap0   :List of 4
 $ f          :List of 4
 $ Rhat       :List of 4
 $ n.eff      :List of 4
 $ pD         : num 2.07
 $ DIC        : num 9667
 $ summary    : num [1:4, 1:11] 600.57 30.41 924.95 9665.09 0.96 ...
  ..- attr(*, "dimnames")=List of 2
 $ samples    :List of 3
  ..- attr(*, "class")= chr "mcmc.list"
 $ modfile    : chr "model4.txt"
 $ model      :List of 3
 $ parameters : chr [1:4] "pop.mean" "pop.sd" "pop.var" "deviance"
 $ mcmc.info  :List of 10
 $ run.date   : POSIXct[1:1], format: "2021-04-07 11:14:37"
 $ parallel   : logi TRUE
 $ bugs.format: logi FALSE
 $ calc.DIC   : logi TRUE
 - attr(*, "class")= chr "jagsUI"
```

We see that `out4` is a huge list of 24 elements. Some of the key output is highlighted in bold font and is this:

- `sims.list` contains the merged output from all chains of the posterior draws for each monitored parameter. This is useful for plotting or for producing the posterior distributions of derived quantities, which are functions of one or more parameters.
- The following 11 elements contain the same information as we just saw in the posterior summary table. These are particularly useful for plotting, where, for instance, you will use the posterior mean and the lower and upper bound of the 95% Bayesian credible interval, which are contained in the elements named `mean`, `q2.5` and `q97.5`. Element `Rhat` may be useful for quick screening of convergence in models that are very parameter-rich.
- Object `summary` is just the posterior summary that we have just seen.
- `samples` contains the individual posterior draws for each chain kept separate, i.e., before they are merged to become the `sims.list`.
- One of the elements in the `mcmc.info` is the elapsed time, which, again, is important for your planning of JAGS runs over the night, the week-end or the vacations.

Remember that the fundamental output from an MCMC (or HMC) algorithm is a stream of numbers for every estimated quantity --- you can think of these algorithms as RNGs (random number generators) for the joint posterior distributions of all the parameters in your models ! To illustrate this, let's look at that fundamental output from the analysis.

```
str(out4$sims.list)

List of 4
 $ pop.mean: num [1:30000] 600 599 597 597 600 ...
 $ pop.sd  : num [1:30000] 29.5 29.9 30.3 29.7 29 ...
 $ pop.var : num [1:30000] 870 894 918 881 840 ...
 $ deviance: num [1:30000] 9616 9617 9625 9622 9617 ...
```

This shows that JAGS has produced for us 30,000 random draws from the joint posterior distribution of the parameters, including the model deviance. We say joint distribution, because these draws contain information also about any correlations among the parameters. You can access these sampled values using standard R functionality, e.g., as follows:

```
print(out4$sims.list$pop.mean)

...... [output STRONGLY truncated !]
[29985] 600.8376 601.9977 602.0186 600.6135
[29989] 600.5735 601.3516 601.0133 598.4386
[29993] 601.0906 601.6634 598.1749 599.6469
[29997] 599.4382 599.4730 600.5569 599.5235
```

As an example of some things that you can do with the output in terms of the posterior summary given above: for a quick check whether any of the parameters has a BGR diagnostic greater than 1.1, i.e., has Markov chains that have not converged, you can type this:

```
hist(out4$summary[,8]) # Rhat values in the eighth column of the summary
which(out4$summary[,8] > 1.1) # None in this case
```

For trace plots "by hand" for the entire chains do:

```
par(mfrow = c(3,1), mar = c(5,5,4,2))
matplot(cbind(out4$samples[[1]][,1], out4$samples[[2]][,1],
out4$sample[[3]][,1]), type = 'l', lty = 1, col = c('red', 'blue',
'green'), xlab = 'Iteration', ylab = 'Value', main = 'Population mean',
frame = FALSE)
matplot(cbind(out4$samples[[1]][,2], out4$samples[[2]][,2],
out4$sample[[3]][,2]), type = 'l', lty = 1, col = c('red', 'blue',
'green'), xlab = 'Iteration', ylab = 'Value', main = 'Population sd', frame
= FALSE)
matplot(cbind(out4$samples[[1]][,3], out4$samples[[2]][,3],
out4$sample[[3]][,3]), type = 'l', lty = 1, col = c('red', 'blue',
'green'), xlab = 'Iteration', ylab = 'Value', main = 'Population variance',
frame = FALSE)
```

... or just for the start of the chains, to see how rapidly they converge ...

```
n <- 10         # Choose last iteration in plot
par(mfrow = c(3,1), mar = c(5,5,4,2))
matplot(cbind(out4$samples[[1]][1:n,1], out4$samples[[2]][ 1:n,1],
out4$sample[[3]][1:n,1]), type = 'l', lty = 1, col = c('red', 'blue',
'green'), xlab = 'Iteration', ylab = 'Value', main = 'Population mean',
frame = FALSE)
matplot(cbind(out4$samples[[1]][1:n,2], out4$samples[[2]][1:n,2],
out4$sample[[3]][1:n,2]), type = 'l', lty = 1, col = c('red', 'blue',
'green'), xlab = 'Iteration', ylab = 'Value', main = 'Population sd', frame
= FALSE)
matplot(cbind(out4$samples[[1]][1:n,3], out4$samples[[2]][1:n,3],
out4$sample[[3]][1:n,3]), type = 'l', lty = 1, col = c('red', 'blue',
'green'), xlab = 'Iteration', ylab = 'Value', main = 'Population variance',
frame = FALSE)
```

We can produce graphical summaries, e.g., histograms or density plots of the posterior distributions for each parameter (Fig. 4), working on the contents of the object `sims.list` that we just inspected above:

```
par(mfrow = c(3, 2), mar = c(5,5,4,2))     # Fig. 4
hist(out4$sims.list$pop.mean, col = "grey", breaks = 100, xlab =
"pop.mean")
plot(density(out4$sims.list$pop.mean), type = 'l', lwd = 3, col =
rgb(0,0,0,0.6), main = "", frame = FALSE)
hist(out4$sims.list$pop.sd, col = "grey", breaks = 100, xlab = "pop.sd")
plot(density(out4$sims.list$pop.sd), type = 'l', lwd = 3, col =
rgb(0,0,0,0.6), main = "", frame = FALSE)
hist(out4$sims.list$pop.var, col = "grey", breaks = 100, xlab = "pop.var")
plot(density(out4$sims.list$pop.var), type = 'l', lwd = 3, col =
rgb(0,0,0,0.6), main = "", frame = FALSE)
```

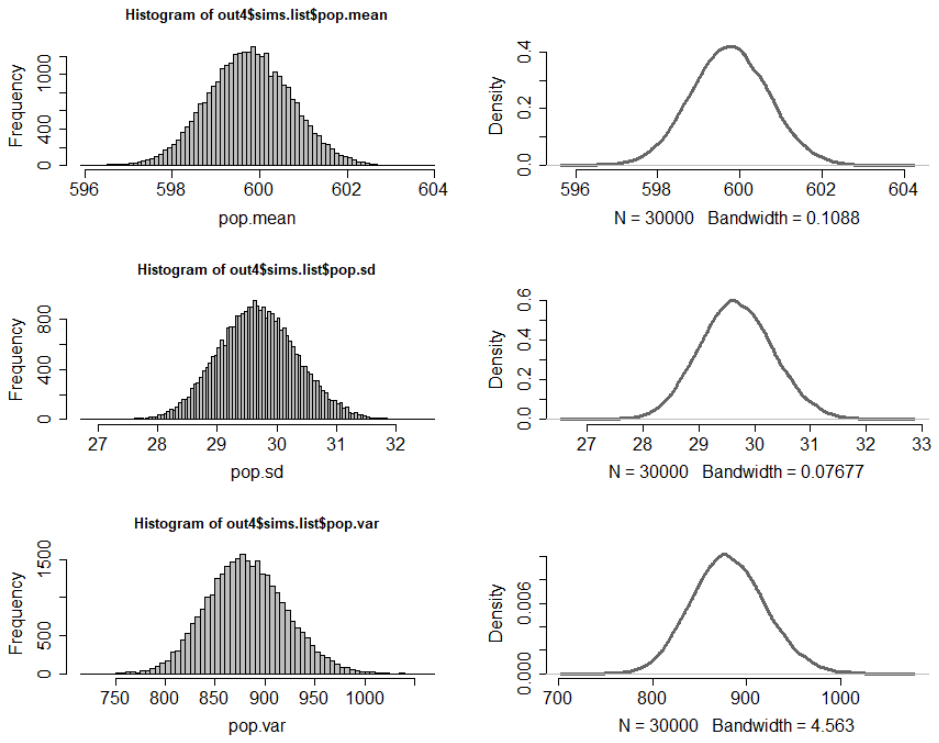

Figure 4: Two variants of density plots for the posterior distributions of the posterior mean, the posterior standard deviation and the posterior variance of the population of male peregrines from which our sample of 1000 birds was drawn. We could also overlay the density estimator (the black line on the right) on the histogram, but only if we choose the argument `freq = FALSE` in the latter.

We can play "probability games" based on the posterior samples produced to estimate the probability that a parameter has a certain range of values. For instance, we can evaluate the probability that mean male peregrine body mass is less than 599 g by conducting a logical test on each draw from the posterior distribution and then tally up the proportion of draws for which the test evaluates to true (using the mean function for the latter as a shortcut) ... we find it is near 0.21.

```
sims <- out4$sims.list
sims$pop.mean < 599        # MANY logical tests ! (not shown)
mean(sims$pop.mean < 599) # Prob(mu < 599)
```

We can also look at the bivariate posterior distribution for two parameters simultaneously ... for instance to check whether the estimates of two parameters are correlated (and here we find they are not --- there is a round cloud and not an elongated one; Fig. 5). If we want, we can also play probability games in two dimensions. Just to give one, wild, example, the probability that the mean mass of male peregrines is less than 600 g, but the standard deviation is more than 30g, can be estimated at about 0.19.

```
# Fig. 5: compute 2d probability game and plot bivariate posterior
test.true <- sims$pop.mean < 600 & sims$pop.sd > 30
mean(test.true)          # [1] 0.1851333
```

```
plot(sims$pop.mean, sims$pop.sd, pch = 16, col = rgb(0,0,0,0.3), cex = 0.8,
frame = FALSE)
points(sims$pop.mean[test.true], sims$pop.sd[test.true], pch = 16, col =
rgb(1,0,0,0.6), cex = 0.8)
abline(h = 30, col = 'red', lwd = 1)
abline(v = 600, col = 'red', lwd = 1)

# alternatively do this to see bivariate posterior plots
pairs(cbind(out4$sims.list$pop.mean, out4$sims.list$pop.sd,
out4$sims.list$pop.var))
```

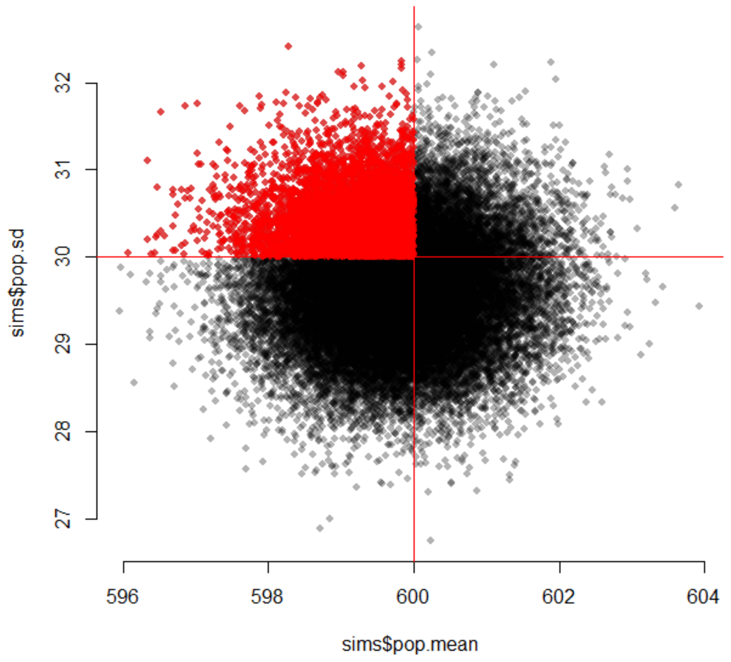

Figure 5: A bivariate representation of the joint posterior distribution with the red showing those draws for which a two-dimensional condition is met: for them, the mean is less than 600 and the standard deviation is more than 30 (this is the top left corner). The probability of this is evaluated at 0.20, which is the proportion of red points among the total number of points.

Numerical summaries of the posterior distribution can be obtained, with the standard deviation requested separately:

```
summary(out4$sims.list$pop.mean)
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  596.0   599.1   599.8   599.8   600.4   603.9

summary(out4$sims.list$pop.sd)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  26.75   29.22   29.67   29.68   30.12   32.63

sd(out4$sims.list$pop.mean)
[1] 0.9499512
```

```
sd(out4$sims.list$pop.sd)
[1] 0.6704534
```

Comparison of these Bayesian estimates again with the classical analysis using maximum likelihood (or, least-squares, see above) show results that are numerically almost indistinguishable, at least for practical purposes.

```
# Compare likelihood with Bayesian estimates and with truth
jags_est <- out4$summary[1:2,1]
tmp <- cbind(truth=truth, lm=lm_est, JAGS=jags_est)
print(tmp, 3)

     truth    lm  JAGS
mean   600 599.7 599.8
sd      30  29.7  29.7
```

After using `jags` the R workspace contains all the results from your Bayesian analysis. If you want to keep these results, *you must save the R workspace* or else output and save the MCMC samples elsewhere.

5 Bayesian analysis with NIMBLE

One major advantage of NIMBLE over JAGS is that you can do much more with it in terms of customization of the model and of the algorithms used to fit the models. However, one drawback over JAGS, currently at least, is that NIMBLE is in some ways *a little* less user-friendly. For instance, as of now it is not just so easy to run NIMBLE in parallel as it is for JAGS using `jagsUI` for instance (see https://mmeredith.net/blog/2021/parallel_chains_foreach.htm for one solution). Also, we don't have the same useful and easy output summary functions as we have with `jagsUI` for JAGS. Therefore, we provide our own little functions for this purpose.

To fit a model in NIMBLE, there is a set of preparations that must be made beforehand. We can either do all individual steps of this (for model building, construction of the algorithm, compilation etc) individually or else we can use the user-friendly function `nimbleMCMC`, which does all in one step, but then allows much less user control. In a first step we will show the easiest manner in which we can fit a model using NIMBLE, where we use function `nimbleMCMC`. This is easier for beginners, but has the disadvantage that the model must be built and compiled every time anew when you want to re-run it, and this is time-consuming. Therefore, in a second step we will show how model building, compilation and MCMC running can all be accomplished in separate steps. This allows much more user control and moreover allows us to avoid unnecessary re-compilation of an already compiled model. This will likely be the way in which you will later operate NIMBLE for the most part for more "serious" modeling, but in this introductory tutorial we emphasize use of the wrapper function `nimbleMCMC`.

In this section, we demonstrate our workflow for using NIMBLE to fit a model. This is very similar to our JAGS model-fitting workflow. And most importantly, you will see below that for this simple model of the mean, the NIMBLE model code is essentially identical to the model code used for JAGS. This similarity will hold for most statistical models.

```
# Load NIMBLE
library(nimble)
```

Next we specify the data list, again in the same format as we used for JAGS. In principle NIMBLE makes a distinction between `constants` (which as the name suggests cannot change and must be specified when the model is defined) and `data` (which may change, useful e.g. when applying the same model to many datasets). In contrast JAGS (and OpenBUGS or WinBUGS) do not allow this. The distinction is not important for this tutorial, so to keep things simple, we will make use of a NIMBLE shortcut and provide both data and constants together to the `constants` argument of `nimbleMCMC`.

```
# Bundle and summarize data (same as for JAGS)
str(dataList <- list(mass = y1000, n = length(y1000)))
```

Next, we write the model in the BUGS language dialect for NIMBLE.

```
# Write Nimble model file
model5 <- nimbleCode( {
# Priors and linear models
pop.mean ~ dunif(0, 5000)         # Normal parameterized by precision
precision <- 1 / pop.variance     # Precision = 1/variance
pop.variance <- pop.sd * pop.sd
pop.sd ~ dunif(0, 100)

# Likelihood
for(i in 1:n){
  mass[i] ~ dnorm(pop.mean, precision)
}
} )
```

Here we don't specify the model in a text file outside of R, but in an additional object inside of the R workspace called `model5` (you can now do `ls()` to see this new object).

Specifying a function to generate initial values and a vector of quantities we want to save is done in the same way as in our JAGS workflow.

```
# Can use same function to generate starting values as for JAGS
inits <- function()
  list (pop.mean = rnorm(1, 600), pop.sd = runif(1, 1, 30))

# Parameters monitored: same as before
params <- c("pop.mean", "pop.sd", "pop.variance")
```

Then, we select the MCMC settings for use in `nimbleMCMC` below. NIMBLE does not require specifying the length of an adaptive phase as JAGS does. The number of posterior draws returned will be floor((niter - nburnin) / nthin) (and we abbreviate these as usual).

```
# MCMC settings
```

```
ni <- 3000   ;   nb <- 1000   ;  nc <- 3   ;  nt <- 1
```

We are ready now to fit the model in NIMBLE in a single step using function `nimbleMCMC`. We wrap the call to `nimbleMCMC` into a call to `system.time` to be able to time the model run (always important when working with MCMC algorithms).

```
# Call NIMBLE (ART 20 sec), check convergence and summarize posteriors
system.time(
  out5 <- nimbleMCMC(code = model5,
    constants = dataList, inits = inits, monitors = params,
    nburnin = nb, niter = ni, thin = nt, nchains = nc,
    samplesAsCodaMCMC = TRUE) )
```

From the output appearing in the R console, we get a feeling that a whole lot of things now happen under the hood, such as compiling, building the model and so on. Below we will see this in a little more detail when we do all this step by step.

When we first look at the object created by NIMBLE we see that it is a list of posterior random draws produced by three runs of the MCMC algorithm.

```
str(out5, 1)

List of 3
 $ chain1: 'mcmc' num [1:2000, 1:3] 601 598 599 600 599 ...
  ..- attr(*, "dimnames")=List of 2
  ..- attr(*, "mcpar")= num [1:3] 1 2000 1
 $ chain2: 'mcmc' num [1:2000, 1:3] 600 601 601 601 601 ...
  ..- attr(*, "dimnames")=List of 2
  ..- attr(*, "mcpar")= num [1:3] 1 2000 1
 $ chain3: 'mcmc' num [1:2000, 1:3] 602 600 600 600 600 ...
  ..- attr(*, "dimnames")=List of 2
  ..- attr(*, "mcpar")= num [1:3] 1 2000 1
 - attr(*, "class")= chr "mcmc.list"
```

To produce the usual summaries, we can write our own R code or use functions in package `coda`. We have written our own function `nimble_summary` for producing a summary table.

```
(nsum <- nimble_summary(out5, params))   # Produce posterior summary table

Estimates based on 3 chains of 2000 iterations
                mean     sd    2.5%     50%   97.5%  rhat
pop.mean     599.753  0.952 597.849 599.741 601.563 1.002
pop.sd        29.694  0.650  28.472  29.676  30.989 1.001
pop.variance 882.171 38.683 810.682 880.688 960.331 1.001
```

We can produce trace plots using the function in R package `coda`.

```
par(mfrow=c(1, 3)); coda::traceplot(out5)   # not shown
```

This manner of running NIMBLE is very practical for a beginner, but later on you will want to dissect the different steps that are combined in this single call to `nimbleMCMC` for better control and to save time by avoiding re-compilation of the model each time you run it. Thus, here we repeat the model fitting by separating all these steps.

To fit our NIMBLE model in this way we will perform several steps: (1) Build a model; (2) Configure and build an MCMC algorithm to use when sampling; (3) convert (i.e. compile) our model and MCMC algorithm from R code to C++ to increase speed; and (4) generate posterior samples. First, we will construct a complete NIMBLE model, which we will call `rawModel`, using the `nimbleModel` function. We need to provide our model code (contained in object `model5`), our list of constants (i.e. input data; `dataList`), and a set of initial values for each model parameter.

```
# Create a NIMBLE model from BUGS code
rawModel <- nimbleModel(code = model5, constants=dataList, inits=inits())
```

Next, we will configure our desired MCMC algorithm for our model, and tell NIMBLE which parameters we want to monitor (`params`). We will use the default MCMC settings here, but deep customization is possible in this step. See the NIMBLE manual, section 7, for more details.

```
# Configure the MCMC algorithm: create a default MCMC configuration
# and monitor our selected list of parameters
mcmcConfig <- configureMCMC(rawModel, monitors=params)
```

We then supply this configuration object to the `buildMCMC` function to create our MCMC algorithm, called `rawMCMC`.

```
# Build the MCMC algorithm function
rawMCMC <- buildMCMC(mcmcConfig)
```

It is now possible to generate posterior samples with this MCMC function. However, since it is written in R code, it will be quite slow.

```
system.time(rawMCMC$run(30)) # Take 30 samples
as.matrix(rawMCMC$mvSamples)
```

The solution is to convert (i.e., compile) our slow R code into fast C++ code. NIMBLE can do this for us automatically. We start by compiling our model specification (`rawModel`) from step 1 using the `compileNimble` function.

```
compModel <- compileNimble(rawModel)
```

Next we compile our MCMC function (`rawMCMC`) into C++ as well. We must also provide our original model object (`rawModel`) as an argument.

```
compMCMC <- compileNimble(rawMCMC, project = rawModel)
```

We can now take samples from the compiled MCMC function, which will be significantly faster.

```
system.time(compMCMC$run(30))
```

Finally, we will use `compMCMC` and the `runMCMC` function to generate posterior samples from 3 chains while specifying total iterations, burn-in and thinning. The output object will be an `mcmc.list`, same as with the `nimbleMCMC` function.

```
system.time(
samples <- runMCMC(compMCMC, niter=ni, nburnin=nb, thin=nt, nchains=nc,
                   samplesAsCodaMCMC=TRUE) )  # Takes almost no time !

# Peak at samples
lapply(samples, head)

# Produce marginal posterior summaries
(nsum <- nimble_summary(samples))

# Traceplots
par(mfrow=c(1,3)); coda::traceplot(samples)
```

Thus, this manner of operating NIMBLE lets the more initiated exert much more control, which is good if you know what you have to do and have a complex model where you will want to do things to speed up the computations. And importantly, you only have to do the compilation steps a single time, which greatly speeds things up. However, it also means that you have to exert more of it even for simple cases.

We close by making our usual comparison with truth for all estimates up to now.

```
# Compare estimates with truth
nimble_est <- nsum[1:2,1]
tmp <- cbind(truth=truth, lm=lm_est, JAGS=jags_est, NIMBLE=nimble_est)
print(tmp, 4)

     truth     lm   JAGS NIMBLE
mean   600 599.75 599.77 599.76
sd      30  29.65  29.68  29.67
```

6 Bayesian analysis with Stan

Next, we fit the "model of the mean" with Stan. We load the Stan package for R, which will mask some of our earlier functionality, such as the `traceplot` function, for which `rstan` has yet another one of its own.

```
# Load Stan R package
library(rstan)
```

We run Stan in a workflow format that again resembles as much as possible that which we established above for JAGS and NIMBLE. That is, we first prepare some R objects, including the model definition file in the Stan language (which is similar to BUGS and of course inspired by it), before we use function `stan` to send all that over to program Stan. Stan then fits the model using Hamiltonian Monte Carlo (HMC; Carpenter et al. 2017) and returns all sampled values of the posterior distributions of the fitted model back to R. Here, we can again summarize this basic Stan output in the usual manner by producing point and interval estimates, plots, and so on.

```
# Bundle and summarize data
str(dataList <- list(n = length(y1000), y = y1000)) # same as before, not
shown
```

As with JAGS, we write a text file containing the Stan model code, which will be saved into the R working directory. The Stan language includes syntax that resembles both C++ and BUGS. For instance, from the former, we have the double slash indicating a comment, and a semicolon is required at the end of each line of code. From the latter, we have the basic definition of the statistical model using the tilde for stochastic relationships. Below, we code to specify the model of the mean in Stan, and then provide further description afterwards.

```
# Write text file with model description in Stan language
cat(file = "model6.stan",  # This line is still R code
"data {                      // This is the first line of Stan code
  int n;                     // Define the format of all data
  vector[n] y;               // ... including the dimension of vectors
}
parameters {                 // Same for the parameters
  real mu;
  real<lower=0> sigma;
}
model {
  // Priors
  mu ~ normal(0, 1000);
  sigma ~ cauchy(0, 10);
  // Likelihood
  y ~ normal(mu, sigma);
}                            // This is the last line of Stan code
" )
```

Recall from our JAGS workflow that our BUGS model code was contained within `model{}`. As you can see Stan is similar, but divides the model file into additional parts. In the `data{}` section, we must provide detailed information about each element of our data list `dataList`, including both the *type* and *dimensions* of each data element as appropriate. In this case we have just two pieces of data: (1) the number of data points `n`, which must be a round number and thus has type integer or `int`, and (2) the actual vector of datapoints `y`, which has type `vector` and has one dimension with `n` values.

Next, the `parameters{}` section contains the type and dimensions of parameters we are estimating in the model: the mean `mu` and the standard deviation `sigma`. Both are of type `real`, meaning they are each a single value that can take on any real value (as opposed to `int` which can only be round numbers). However, we also tell Stan to force `sigma` to have a lower bound at 0, since you can't have a negative standard deviation.

Finally, the `model{}` section should look familiar to our JAGS model. We provide priors on both `mu` and `sigma` using the tilde. Note that the normal density is defined in terms of the standard deviation rather than in terms of the precision as in JAGS. We use a Cauchy prior for the standard deviation, a distribution which resembles a normal distribution but has fatter tails. The model file is a text file which should have the extension `.stan` and which must not use in its name any periods (hence use underlines for the goal of structuring a model name).

HMC is usually slower than MCMC per iteration, but is more efficient in the sense of producing posterior draws with (much) less serial correlation. Hence, we will usually be fine with shorter chains (and sometimes much shorter ones) than in JAGS and NIMBLE and with little or no thinning.

```
# HMC settings
ni <- 1200   ;   nb <- 200   ;  nc <- 3   ;  nt <- 1

# Call STAN (ART 28 / 2 sec)
system.time(
  out6 <- stan(file = "model6.stan", data=dataList,
    chains=nc, iter=ni, warmup=nb, thin=nt)  )     # Ignore the warnings
```

On the first execution, this takes about 30 sec, but it runs and produces our first analysis in Stan. On re-fitting the model we will observe that run time is reduced to about 2 sec only. This is mainly due to the fact that some overhead related to compiling the model needs only be done once, on the first run of that particular model.

```
# Check convergence and print posterior summaries
par(mfrow = c(2, 2))                 # not sure why this is ignored !
traceplot(out6, c("mu", "sigma"))    # Wilted-flower color plots ...
print(out6, c("mu", "sigma"))
print(out6, dig = 1)

Inference for Stan model: model6.
3 chains, each with iter=1200; warmup=200; thin=1;
post-warmup draws per chain=1000, total post-warmup draws=3000.

        mean se_mean   sd  2.5%    25%    50%    75%  97.5% n_eff Rhat
mu    599.73    0.02 0.96 597.8 599.08 599.73 600.38 601.64  3362    1
sigma  29.62    0.02 0.65  28.4  29.16  29.60  30.05  30.90   785    1

Samples were drawn using NUTS(diag_e) at Wed Apr 07 09:37:37 2021.
For each parameter, n_eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor on split chains (at
convergence, Rhat=1).
```

Up to MC error, these estimates should be numerically identical to those from the other model fitting engines. And as for JAGS and NIMBLE, the output object in R is huge and can and should be inspected.

```
str(out6)            # not shown
```

Here is how we can first extract only the posterior samples from the chains produced and then produce visual output using plotting with coda.

```
library(coda)
sims <- extract(out6)
str(sims)
plot(mcmc(cbind(sims[[1]], sims[[2]], sims[[3]])))
```

Hence, with Stan we now have a third engine for Bayesian model fitting at our avail. We finish by making our usual comparisons of all sets of estimates up to now.

```
# Compare estimates with truth
stan_est <- summary(out6)$summary[1:2,1]
tmp <- cbind(truth=truth, lm=lm_est, JAGS=jags_est, NIMBLE=nimble_est,
Stan=stan_est)
print(tmp, 4)

     truth     lm   JAGS NIMBLE   Stan
mean   600 599.75 599.77 599.76 599.73
sd      30  29.65  29.68  29.67  29.62
```

7 Do-it-yourself maximum likelihood estimates (DIY-MLEs)

We want to define a likelihood function for "the model of the mean" such that we can use it, along with any data set, in R's function optimizer `optim` to identify those values of the parameters that maximize the likelihood of our model for the data set at hand. The likelihood to be maximized for this model is the joint probability of all dataunits in the data set, where the contribution from each datum comes from a normal density. In other words, we define a likelihood that is the product of *n* normal random variables, where *n* is our sample size. Since we assume independence of the mass of all measured peregrines, we can work with the product of the densities of these random variables, or, in practice, we will work with the sum of the log densities. Actually: since optimisation works by function minimization, we will work with the *negative* of the sum of the log densities. The parameter values that minimize the negative log-likelihood function are the same as those that maximize the likelihood function.

Based on our probability model, the contribution to the joint likelihood of the data set coming from one peregrine (with datum y_i) is that of a normal (or Gaussian) probability density:

$$f(y_i \mid \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{y_i-\mu}{\sigma}\right)^2}$$

This is exactly what we get in R by writing the following, which you can think of as a shorthand for the ugly expression on the right-hand side of the equation above.

```
dnorm(y[i], mean, sd)
```

Hence, the joint likelihood of all *n* body mass measurements, and thus the likelihood that we want to maximize, is the product over *n* such terms:

$$L(\mathbf{y} \mid \mu, \sigma) = \prod_{i=1}^{n} \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{y_i-\mu}{\sigma}\right)^2} ,$$

and in R, we could write this:

```
    L <- dnorm(y[1], mean, sd) * dnorm(y[2], mean, sd) * ... *
dnorm(y[n], mean, sd)
```

(or more concisely, using the `prod()` function). Below, we show explicitly the likelihood and the log-likelihood for one observation and then the negative sum of the log-likelihoods over all observations in the data set. We denote these by L, LL and NLL, respectively, in the definition of the likelihood function (which will actually define the negative log-likelihood). This is meant to make things more transparent. For the log-likelihood we take the log of L explicitly, even though a numerically more accurate variant is to set `log = TRUE` as an argument in the density function in R (Bolker 2008).

In our likelihood function `NLL` we define the relationship between the parameters to be estimated (which for `optim` we must put into a vector, which we call `param` and make an argument of our function `NLL`) and the observed data (which we call `y`) under the assumed probabilistic model, which here is that of an independent normal random variable for every observation `y[i]`.

```
?optim                    # Check out the optim function
```

Once we have defined the function for the negative log-likelihood (`NLL`) we use `optim` to identify those values for the elements in `param` which lead to the highest function value for the likelihood, or, and this is the same, to the lowest value for `NLL`. For `optim` we must provide starting values (`inits`) and a data set (`y`). In addition, we ask for the Hessian matrix to be computed. The Hessian is the matrix containing the second partial derivatives of the log-likelihood function with respect to the estimated parameters when evaluated at the MLEs. When the `NLL` is minimized as we do here, then we actually obtain the *negative* of the Hessian (though this is not visible in the output from `optim`). The Hessian carries the information about the local curvature of the likelihood function around the MLEs: when the curvature is steep, we have a lot of information about the estimate of a parameter and when it is rather flat, we do not have much information. This is the basis for obtaining the asymptotic variance-covariance matrix (`VC`) of the estimates. Taking the square root of the diagonal of `VC` yields the asymptotic standard errors (`ASE`, or simply `SE` for short) of the estimated parameters.

Executing the following beige-colored code block will yield the MLEs for the mean and the standard deviation of the model fit to the small data set with 10 observations only.

```
# Definition of NLL for a normal linear model with an intercept only
NLL <- function(param, y) {
  mu <- param[1]           # Define the first element to be the mean ..
  sigma <- param[2]        # ... and the second the standard deviation
  L <- dnorm(y, mu, sigma) # Likelihood contribution for 1 observation
  LL <- log(L)             # Loglikelihood for 1 observation
  NLL <- -sum(LL)          # NLL for all observations in vector y
  return(NLL)
}

# An alternative, which is more numerically accurate
NLL <- function(param, y) {
  mu <- param[1]
  sigma <- param[2]
  # log-likelihood contribution for 1 observation
  LL <- dnorm(y, mu, sigma, log = TRUE)
  NLL <- -sum(LL)          # NLL for all observations in vector y
  return(NLL)
}

# Minimize NLL and use quadratic approximation of SEs using the Hessian
inits <- c('mu' = 500, 'sigma' = 10)
sol <- optim(inits, NLL, y=y10, hessian=TRUE)  # Small data set
MLE <- sol$par                  # Grab MLEs
VC <- solve(sol$hessian)        # Get variance-covariance matrix
ASE <- sqrt(diag(VC))           # Extract asymptotic SEs
print(cbind(MLE, ASE), 3)       # Print MLEs and SE's
```

```
        MLE  ASE
mu    595.8 6.92
sigma  21.9 4.87

sol        # Look at output from optimizer
summary(lm(y10~1))       # Compare with least-squares solution

$par
       mu     sigma
595.82624  21.87164

$value
[1] 45.06595

$counts
function gradient
      49       NA

$convergence
[1] 0

$message
NULL

$hessian
                 mu         sigma
mu     2.090438e-02 -1.878764e-05
sigma -1.878764e-05  4.211802e-02



Call:
lm(formula = y10 ~ 1)

Residuals:
    Min      1Q  Median      3Q     Max
-32.694 -14.292  -5.008  17.861  33.032

Coefficients:
```

```
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  595.816      7.308   81.52 3.18e-14 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 23.11 on 9 degrees of freedom

### Note slight difference in RSD estimate and MLE of that parameter
```

We package part of the code into a new function which we call `getMLE` and then repeat the analysis for the large data. Argument `sol` (for 'solution') in the `getMLE` function is the result of our call to `optim` above, which moreover is assumed to have `hessian = TRUE`.

```
# Define function to get asymptotic SEs and print out MLEs and SEs
getMLE <- function(sol, dig = 3){
  MLE <- sol$par
  VC <- solve(sol$hessian)
  ASE <- sqrt(diag(VC))
  print(cbind(MLE, ASE), dig)
}

out7 <- optim(inits, NLL, y=y1000, hessian=TRUE)  # Large data set
getMLE(out7, 5)

         MLE     ASE
mu    599.75 0.93761
sigma  29.65 0.66346

# Compare again with least-squares solution for the big data set
summary(lm(y1000~1))

Call:
lm(formula = y1000 ~ 1)

Residuals:
    Min      1Q  Median      3Q     Max
-76.425 -21.581  -0.368  20.922  84.051

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 599.7482     0.9376   639.6   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 29.65 on 999 degrees of freedom

### Slight difference in variance estimator no longer matter with n=1000
```

A comparison with the estimates from the smaller data set makes sense: with a sample size *n* that is 100 times larger than in the small data set, the MLEs are now much closer to the truth and the magnitude of the SE is reduced by a factor of about $\sqrt{100}$.

```
# Compare estimates with truth and previous estimates
diy_est <- out7$par
tmp <- cbind(truth=truth, lm=lm_est, JAGS=jags_est,
   NIMBLE=nimble_est, Stan=stan_est, DIY=diy_est)
print(tmp, 4)

     truth     lm   JAGS NIMBLE   Stan    DIY
mean   600 599.75 599.77 599.76 599.73 599.75
sd      30  29.65  29.68  29.67  29.62  29.65
```

8 Get the MLEs with TMB instead

TMB is a very powerful model-fitting engine that enables MLEs to be obtained for even complex hierarchical models (Kristensen et al. 2017; Monnahan & Kristensen, 2018). However, for a beginner, it is hard to understand how TMB works, especially in the context of such complicated statistical models for which TMB is typically used. So, here you will see TMB being used in a really simple model, which will hopefully help you understand to see how TMB "works", at least in a practical sense. So, let's now marvel at what is perhaps the most complicated manner in which you have ever estimated the mean and the sd of a sample from one single normal distribution :) As you will see, fitting a model with TMB combines code syntax similar to Stan (because like Stan, it is based on C++) and math similar to the DIY likelihood we demonstrated in the previous section.

```
library(TMB)
```

For this model we can use exactly the same data bundle as before.

```
# Bundle and summarize data
str(dataList <- list(y = y1000, n = length(y1000)))
```

As writing a model in TMB is complicated, we will present the code in several steps below before combining all the pieces into a single model file at the end. Our code consists of a function written in C++, analogous to the DIY likelihood function we wrote in the last section. The first several lines of the model code contain boilerplate, but required, setup code. The only thing we will point out here is that we specify that the output value of this function will be of type `Type` (confusingly). This is because TMB uses C++ templates (hence the name Template Model Builder), which are useful for writing flexible and re-usable code. A detailed explanation of the use and value of templates is beyond the scope of this tutorial.

```
#include <TMB.hpp>

template<class Type>
Type objective_function<Type>::operator() ()
{
```

In the first section of our model function, we describe the types of our input data in a manner similar to Stan. We identify `n` as an integer and `y` as a vector using special functions `DATA_INTEGER` and `DATA_VECTOR` respectively (and end every line in a semicolon):

```
DATA_INTEGER(n);
DATA_VECTOR(y);
```

Note that unlike with Stan we do not have to specify the dimensions here. We will do that later in R.

Next, and again like Stan, we need to tell TMB about our parameters `mu` and `sigma`. However instead of sigma we are specifying our parameter as log(sigma) or `log_sigma`, so we don't have to worry about accounting for sigma's lower bound at 0. In both cases the parameters are real-valued scalars, so we use the PARAMETER function.

```
PARAMETER(mu);
PARAMETER(log_sigma);
```

After specifying `log_sigma`, we then calculate sigma for later use:

```
Type sigma = exp(log_sigma);
```

Note that we specify the type of `sigma` as `Type`, matching the output type of our function. What this means in a practical sense is that `sigma` will be a real number. We know `sigma` will be greater than 0, as required, because exp(any number) is always greater than 0. Next, we initialize the total log-likelihood at 0, again using `Type`:

```
Type loglik = 0.0;
```

Finally we are ready to actually calculate the log-likelihood. As with our DIY likelihood function, we will iterate over each of our `n` datapoints, calculate the log-likelihood of that datapoint using `dnorm`, and add it to our total log likelihood using a for loop. This loop will look a little different than how it is done in R, but the similarities should be clear. The most important thing to remember is that in C++, and thus in TMB, the index of the first element in a vector is 0, not 1 as it is in R, JAGS, NIMBLE, and Stan.

```
for (int i=0; i<n; i++){
    loglik += dnorm(y(i), mu, sigma, true); //Add log-lik of obs i
}
```

The `loglik += something` syntax is equivalent to saying `loglik = loglik + something`. Also note that the final `true` argument to the `dnorm` function indicates that we want the log-likelihood to be returned. Finally, we return the negative of total log-likelihood:

```
return -loglik;
```

With our model function complete, we will write out the whole thing to a file ending in the extension .cpp (for C plus plus) using `cat`.

```
# Write TMB model file
cat(file="model8.cpp",
"#include <TMB.hpp>

template<class Type>
Type objective_function<Type>::operator() ()
{
  //Describe input data
  DATA_INTEGER(n);      //Sample size
  DATA_VECTOR(y);       //response

  //Describe parameters
  PARAMETER(mu);        //Mean
  PARAMETER(log_sigma); //log(standard deviation)

  Type sigma = exp(log_sigma);  //Type = match type of function output
(double)

  Type loglik = 0.0;    //Initialize total log likelihood at 0

  for (int i=0; i<n; i++){ //Note index starts at 0 instead of 1!
    //Calculate log-likelihood of observation
    //value of true in final argument indicates function should return
log(lik)
    loglik += dnorm(y(i), mu, sigma, true); //Add log-lik of obs i
  }

  return -loglik; //Return negative of total log likelihood
}
")
```

We next compile our model file and load the result into our R session.

```
# Compile and load TMB function
compile("model8.cpp") # Produces a lot of gibberish ... have no fear !
dyn.load(dynlib("model8"))

?TMB::compile               # Check out that function if you wish
```

As we mentioned earlier, we need to tell TMB about the dimensions of our parameters, which we do by creating an initial value for each parameter and combining them in a list.

```
# Provide dimensions and starting values for parameters
params <- list(mu=0, log_sigma=0)
```

Using our input data, parameter description, and compiled model object, we then create a TMB object with the function `MakeADFun` that is ready to be optimized.

```
# Create TMB object
out8 <- MakeADFun(data = dataList,
            parameters = params, random=NULL,
            DLL = "model8", silent=TRUE)

?TMB::MakeADFun                # Check out that function
```

The TMB object contains both a function to optimize and a corresponding gradient function, both of which we pass to `optim`. Otherwise, this is similar to what we did in the previous section.

```
# Optimize TMB object and print results
starts <- rep(0, length(unlist(params)))
sol <- optim(starts, fn=out8$fn, gr=out8$gr, method="BFGS", hessian = TRUE)
```

We have written a utility function `tmb_summary` that automatically summarizes parameter estimates and standard errors from an optimized TMB object.

```
(tsum <- tmb_summary(out8))   # Use our summary function

            Estimate Std. Error
mu        599.748227 0.93716246
log_sigma   3.388979 0.02236064
```

9 Comparison of the parameter estimates

We believe that it is important to realize that while the difference between Bayesian and likelihood inference matters very much philosophically, in practical terms, we very often get estimates that from a simplistic numerical perspective are almost indistinguishable. Hence, we like to emphasize that most of the times our choice of whether to use a Bayesian or a likelihood model fitting engine should be guided by which one gets us what we want for the least cost. So, here we will now compare the point estimates of the two parameters in our very simple statistical model, which we produced by all six fitting methods that we have seen in this tutorial. For the Bayesian engines we will be using the posterior means, but re-iterate that sometimes you may want to use the median or the mode as your point estimate, and that strictly, it is the latter which corresponds to the MLEs when you use vague priors.

```
# Compare results with truth and previous estimates
tmb_est <- c(sol$par[1], exp(sol$par[2])) # backtransform SD estimate!
tmp <- cbind(cbind(truth=truth, lm=lm_est, JAGS=jags_est,
   NIMBLE=nimble_est, Stan=stan_est, DIY=diy_est, TMB=tmb_est))
print(tmp, 4)

     truth     lm   JAGS NIMBLE   Stan    DIY    TMB
mean   600 599.75 599.77 599.76 599.73 599.75 599.75
sd      30  29.65  29.68  29.67  29.62  29.65  29.64
```

We would argue that you can consider these estimates as numerically identical for virtually all practical purposes.

10 Summary

We have considered one of the simplest possible statistical models and called it the "model of the mean". This is a normal distribution with two parameters, one to describe the average or the mean of a response, and the other to quantify the amount of noise or variability of the individual datum around that mean. Another name for the parametric statistical model in this tutorial would be a normal linear model with just an intercept. The typical nonbayesian fitting algorithm for this model is also sometimes known as OLS (ordinary least squares).

We used R to simulate two variants of such a data set, a smaller one and a much larger one. As always, you can consider R code for data simulation as just another method to describe a parametric statistical model, although for our trivially simple model here, you may perhaps not be overly convinced by our claim that data simulation code in R (or any other language) is a wonderful manner to explain a certain statistical model (Kéry, 2010; Kéry & Royle, 2016, 2021). We believe that this is perhaps more obvious with more complex models.

We have gone through a whole series of model fitting exercises. We have first fit the data-generating model to our data set(s) using a traditional model fitting function in R. Here, this was function `lm`, which applies a method called (ordinary) least-squares and which for a linear model with normal response yields the same estimates of the mean parameters as those that we would get if we used the maximum likelihood method (and there is only a slight discrepancy in the variance estimate). After that, we used our three Bayesian model fitting engines JAGS, NIMBLE and Stan to produce posterior inferences for the same model. We have seen how to fit the model in these, you have to do much more coding and also that the model description will appear, at first at least, much more complicated than what we did when fitting the same model using `lm` in R. After that, we looked at the actual likelihood of our model and defined our own function for it, which we then maximised using the R function `optim` to produce what we jokingly call our DIY-MLEs. And for a final set of solutions to our model fitting comparison, we used TMB to produce what again are the MLEs for the same model. We have strived for a consistent format and workflow for the three Bayesian model fitting engines and hope that this will help you understand them better and also, to be better able to see some of the differences between them.

After fitting our model in six different ways, we compared our estimates for the mean and standard deviation in our model and found them to be numerically identical for practical purposes. This is what we will see most of the times when we use vague priors in our Bayesian analyses, as we did throughout in this tutorial. We do not want to argue for an anything-goes philosophy in statistical modeling: in terms of what they mean, there are deep philosophical differences between estimates obtained using Bayesian posterior inference and maximum likelihood and you must know and understand these (and ideally read some Bayesian intro books such as Hobbs & Hooten, 2015). But at the same time, most statisticians would agree that both estimation methods are valid to fit a statistical model. Hence, you can take one or the other and this in practice means that you have a choice to take that method, and hence, that fitting engine, which is easiest for you or which you find most interesting or fascinating. Just keep in mind that you may raise some eyebrows among the reviewers of your paper when you write "we used TMB to estimate the average in our data set of bird masses".