Linear Regression 303

Alfonso R. Reyes

2019-09-18

Contents

Prerequisites **5**

1 Generalised additive models (GAMs): an introduction **7**
1.1 Introduction . . . 7
1.2 Running the analysis . . . 7
1.3 Communicating the results . . . 11

2 Fitting GAMs with brms: part 1 **13**
2.1 Introduction . . . 13
2.2 Load packages . . . 13
2.3 MASS motorcycle dataset . . . 14
2.4 Bayesian Approach . . . 16
2.5 Comparison . . . 18

3 Regression with Stan **21**
3.1 Regression Models . . . 21
3.2 Stan Code . . . 25
3.3 Running the Model . . . 25

Prerequisites

This is a *sample* book written in **Markdown**. You can use anything that Pandoc's Markdown supports, e.g., a math equation $a^2 + b^2 = c^2$.

The **bookdown** package can be installed from CRAN or Github:

```
install.packages("bookdown")
# or the development version
# devtools::install_github("rstudio/bookdown")
```

Remember each Rmd file contains one and only one chapter, and a chapter is defined by the first-level heading `#`.

To compile this example to PDF, you need XeLaTeX. You are recommended to install TinyTeX (which includes XeLaTeX): https://yihui.name/tinytex/.

Chapter 1

Generalised additive models (GAMs): an introduction

1.1 Introduction

Source: http://environmentalcomputing.net/intro-to-gams/

Many data in the environmental sciences do not fit simple linear models and are best described by "wiggly models", also known as Generalised Additive Models (GAMs).

Let's start with a famous tweet by one Gavin Simpson, which amounts to:

1. GAMs are just GLMs
2. GAMs fit wiggly terms
3. use `+ s(x)` not `x` in your syntax
4. use method `= "REML"`
5. always look at `gam.check()`

This is basically all there is too it - an extension of generalised linear models (GLMs) with a smoothing function. Of course, there may be many sophisticated things going on when you fit a model with smooth terms, but you only need to understand the rationale and some basic theory. There are also lots of what would be apparently magic things happening when we try to understand what is under the hood of say lmer or glmer, but we use them all the time without reservation!

1.2 Running the analysis

Before we consider a GAM, we need to load the package mgcv – the choice for running GAMs in R.

```
library(mgcv)
#> Loading required package: nlme
#> This is mgcv 1.8-28. For overview type 'help("mgcv-package")'.
library(ggplot2)
#> Registered S3 methods overwritten by 'ggplot2':
#>   method         from
#>   [.quosures     rlang
#>   c.quosures     rlang
#>   print.quosures rlang
```

We'll now look at a quick real example – we'll just scratch the surface, and in a future tutorial we will look at it in more detail. We're going to look at some CO2 data from Manua Loa (it's used elsewhere in this series). We will fit a couple GAMs to the data to try and pick apart the intra- and inter-annual trends.

First load the data – you can download it here.

```
CO2 <- read.csv(file.path(data_raw_dir, "mauna_loa_co2.csv"))
```

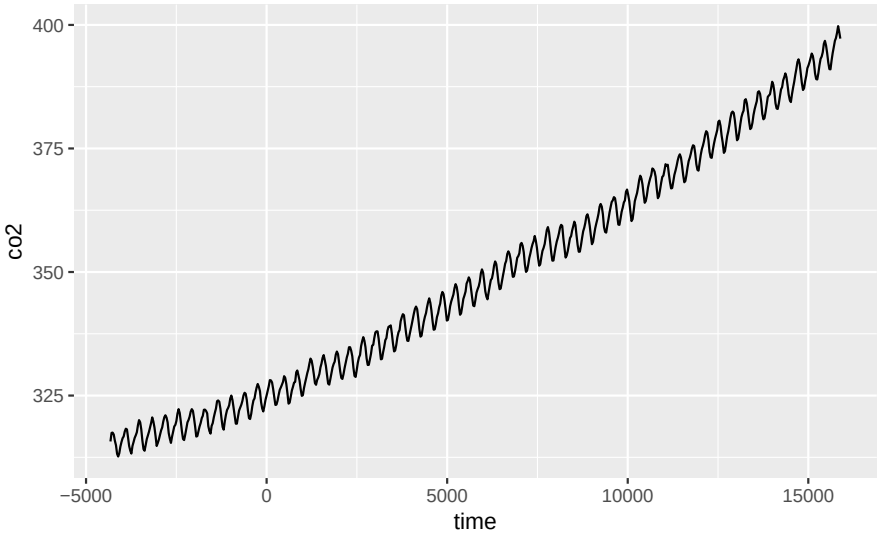
We want to look at inter-annual trend first, so let's convert the date into a continuous time variable (take a subset for visualisation).

```
CO2$time <- as.integer(as.Date(CO2$Date, format = "%d/%m/%Y"))
CO2_dat <- CO2
CO2 <- CO2[CO2$year %in% (2000:2010),]
```

OK, so let's plot it and look at a smooth term for time.

$$y = \beta_0 + f_{\text{trend}}(time) + \varepsilon, \quad \varepsilon \sim N(0, \sigma^2)$$

```
ggplot(CO2_dat, aes(time, co2)) + geom_line()
```

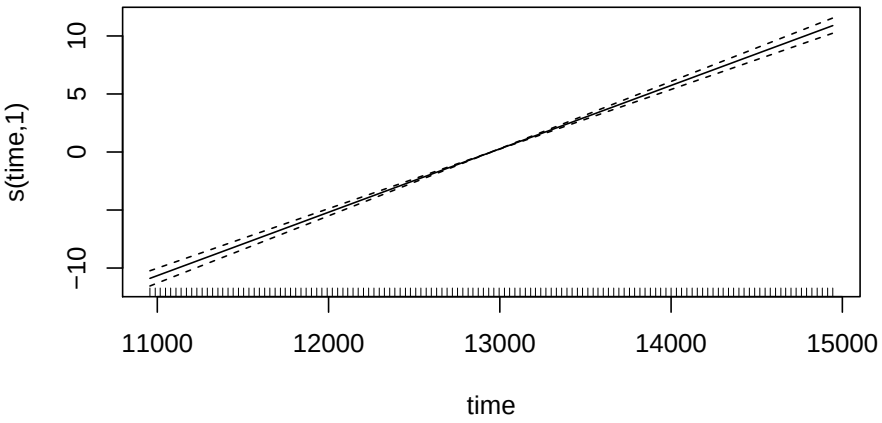


We can fit a GAM for these data using:

```
CO2_time <- gam(co2 ~ s(time), data = CO2, method = "REML")
```

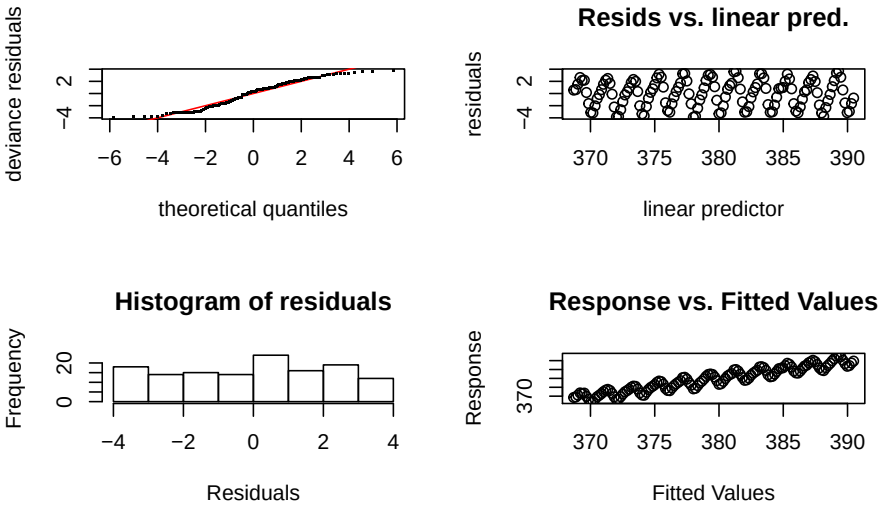
which fits a model with a single smooth term for time. We can look at the predicted values for this:

```
plot(CO2_time)
```



Note how the smooth term actually reduces to a 'normal' linear term here (with an edf of 1) – that's the nice thing about penalised regression splines. But if we check the model, then we see something is amuck.

```
par(mfrow = c(2,2))
gam.check(CO2_time)
#>
#> Method: REML   Optimizer: outer newton
#> full convergence after 8 iterations.
#> Gradient range [-0.000145,6.46e-05]
#> (score 291 & scale 4.79).
#> Hessian positive definite, eigenvalue range [0.000145,65].
#> Model rank =  10 / 10
#>
#> Basis dimension (k) checking results. Low p-value (k-index<1) may
#> indicate that k is too low, especially if edf is close to k'.
#>
#>         k' edf k-index p-value
#> s(time)  9   1    0.16  <2e-16 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```



The residual plots have a very odd looking rise-and-fall pattern – clearly there is some dependance structure (and we can probably guess it has something to do with intra-annual fluctuations). Let's try again, and introduce something called a cyclical smoother.

$$y = \beta_0 + f_{\text{intrannual}}(month) + f_{\text{trend}}(time) + \varepsilon, \quad \varepsilon \sim N(0, \sigma^2)$$

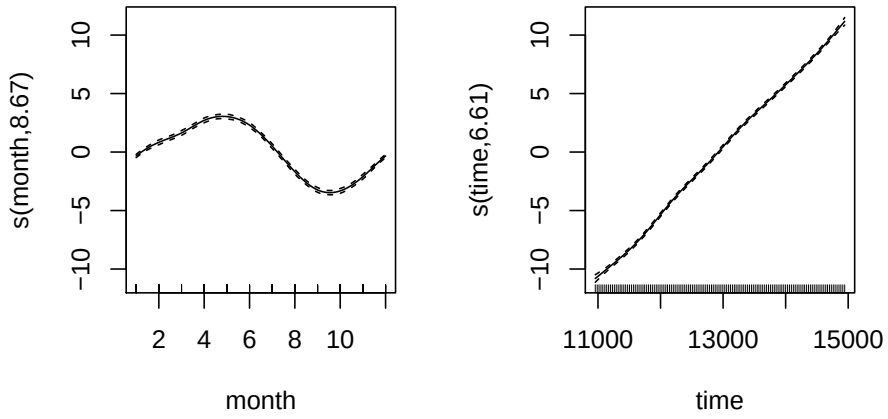
The cyclical smooth term, $f_{\text{intrannual}}(month)$, is comprised of basis functions just the same as we have seen already, except that the end points of the spline are constrained to be equal – which makes sense when we're modelling a variable that is cyclical (across months/years).

We'll now see the bs = argument to choose the type of smoother, and the k = argument to choose the number of knots, because cubic regression splines have a set number of knots. We use 12 knots, because there are 12 months.

```
CO2_season_time <- gam(co2 ~ s(month, bs = 'cc', k = 12) + s(time),
                       data = CO2,
                       method = "REML")
```

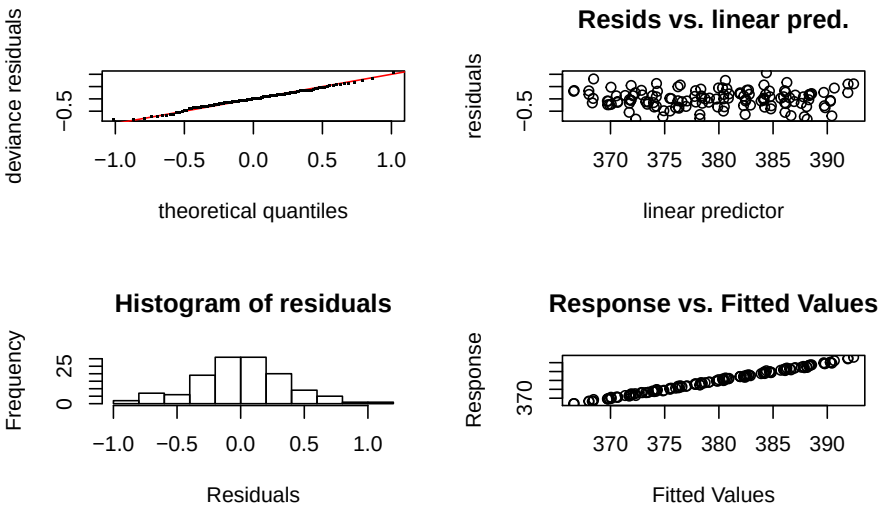
Let's look at the fitted smooth terms:

```
par(mfrow = c(1,2))
plot(CO2_season_time)
```



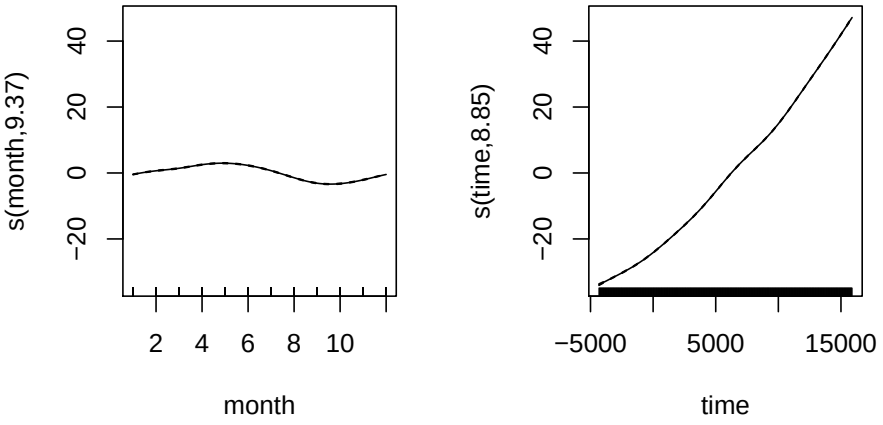
Looking at both smooth terms, we can see that the monthly smoother is picking up that monthly rise and fall of CO2 – looking at the relative magnitudes (i.e. monthly fluctuation vs. long-term trend), we can see how important it is to disintangle the components of the time series. Let's see how the model diagnostics look now:

```
par(mfrow = c(2,2))
gam.check(CO2_season_time)
#>
#> Method: REML   Optimizer: outer newton
#> full convergence after 6 iterations.
#> Gradient range [-2.64e-06,5.26e-08]
#> (score 87.7 & scale 0.144).
#> Hessian positive definite, eigenvalue range [1.03,65.4].
#> Model rank =  20 / 20
#>
#> Basis dimension (k) checking results. Low p-value (k-index<1) may
#> indicate that k is too low, especially if edf is close to k'.
#>
#>              k'   edf k-index p-value
#> s(month) 10.00  8.67    0.72  <2e-16 ***
#> s(time)   9.00  6.61    0.87    0.09 .
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```



Much better. Let's look at how the seasonal component stacks up against the full long term trend.

```
CO2_season_time <- gam(co2 ~ s(month, bs = 'cc', k = 12) + s(time),
                       data = CO2_dat,
                       method = "REML")
par(mfrow = c(1,2))
plot(CO2_season_time)
```



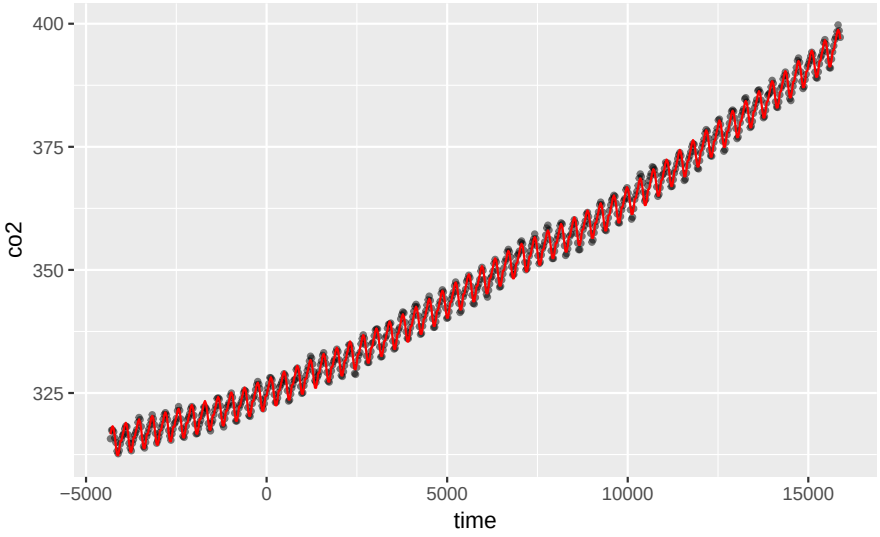
There's more to the story – pephaps spatial autocorrelations of some kind? gam can make use of the spatial autocorrelation structures available in the nlme package, more on that next time. Hopefully for the meantime GAMs now don't seem qutie so scary or magical, and you can start to make use of what is really an inrecibly flexible and powerful modelling framework.

1.3 Communicating the results

You can essentially present model results from a GAM as if it were any other linear model, the main difference being that for the smooth terms, there is no single coefficient you can make inference from (i.e. negative, positive, effect size etc.). So you need to rely on either interpretting the parital effects of the smooth terms visually (e.g. from a call to plot(gam_model)) or make inference from the predicted values. You can of course include normal linear terms in the model (either continuous or categorical, and in an ANOVA type framework even) and make inference from them like you normally would. Indeed, GAMs are often useful for accounting for a non-linear phenomonon that is not directly of interest, but needs to be acocunted for when making inferece about other variables.

You can plot the partial effects by calling the plot function on a fitted gam model, and you can look at the parametric terms too, possibly using the termplot function too. You can use ggplot for simple models like we did earlier in this tutorial, but for more complex models, it's good to know how to make the data using predict. We just use the existing time-series here, but you would generate your own data for the `newdata=` argument.

```
CO2_pred <- data.frame(time = CO2_dat$time,
                       co2 = CO2_dat$co2,
                       predicted_values = predict(CO2_season_time,
                                                  newdata = CO2_dat))
ggplot(CO2_pred, aes(x = time)) +
  geom_point(aes(y = co2), size = 1, alpha = 0.5) +
    geom_line(aes(y = predicted_values), colour = "red")
```



Chapter 2

Fitting GAMs with brms: part 1

2.1 Introduction

Source: https://www.fromthebottomoftheheap.net/2018/04/21/fitting-gams-with-brms/

Regular readers will know that I have a somewhat unhealthy relationship with GAMs and the `mgcv` package. I use these models all the time in my research but recently we've been hitting the limits of the range of models that `mgcv` can fit. So I've been looking into alternative ways to fit the GAMs I want to fit but which can handle the kinds of data or distributions that have been cropping up in our work. The `brms` package (Bürkner, 2017) is an excellent resource for modellers, providing a high-level R front end to a vast array of model types, all fitted using Stan. `brms` is the perfect package to go beyond the limits of mgcv because `brms` even uses the smooth functions provided by `mgcv`, making the transition easier. In this post I take a look at how to fit a simple GAM in `brms` and compare it with the same model fitted using `mgcv`.

2.2 Load packages

In this post we'll use the following packages.

```
## packages
library(mgcv)
#> Loading required package: nlme
#> This is mgcv 1.8-28. For overview type 'help("mgcv-package")'.
library(brms)
#> Loading required package: Rcpp
#> Registered S3 methods overwritten by 'ggplot2':
#>   method         from
#>   [.quosures     rlang
#>   c.quosures     rlang
#>   print.quosures rlang
#> Registered S3 method overwritten by 'xts':
#>   method     from
#>   as.zoo.xts zoo
#> Loading 'brms' package (version 2.8.0). Useful instructions
#> can be found by typing help('brms'). A more detailed introduction
#> to the package is available through vignette('brms_overview').
#>
#> Attaching package: 'brms'
```

```
#> The following objects are masked from 'package:mgcv':
#>
#>     s, t2
library(ggplot2)
library(schoenberg)

theme_set(theme_bw())
```

If you don't know `schoenberg`, it's a package I'm writing to provide `ggplot` versions of plots that can be produced by `mgcv` from fitted GAM objects. `schoenberg` is in early development, but it currently works well enough to plot the models we fit here. If you've never come across this package before, you can install it from Github using `devtools::install_github('gavinsimpson/schoenberg')`.

2.3 MASS motorcycle dataset

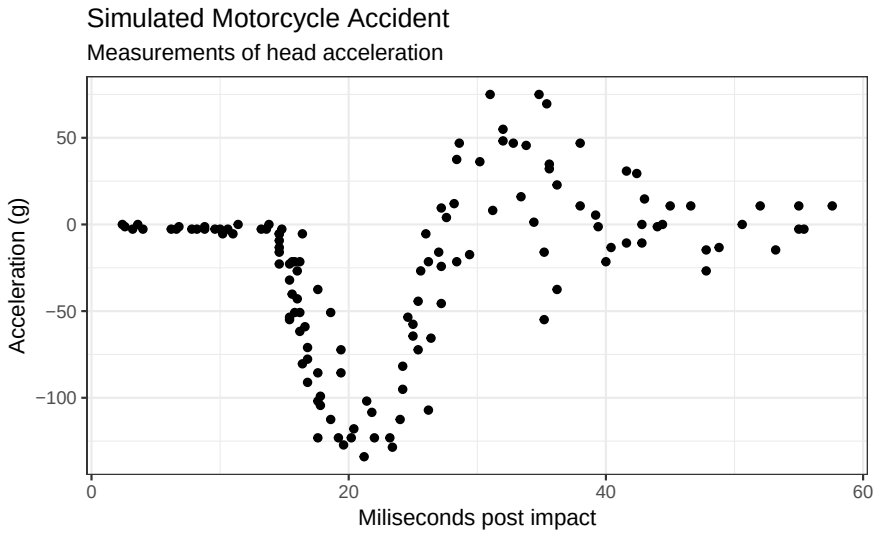
To illustrate brms's GAM-fitting chops, we'll use the `mcycle` data set that comes with the `MASS` package. It contains a set of measurements of the acceleration force on a rider's head during a simulated motorcycle collision and the time, in milliseconds, post collision. The data are loaded using `data()` and we take a look at the first few rows:

```
## load the example data mcycle
data(mcycle, package = 'MASS')

## show data
head(mcycle)
#>   times accel
#> 1   2.4   0.0
#> 2   2.6  -1.3
#> 3   3.2  -2.7
#> 4   3.6   0.0
#> 5   4.0  -2.7
#> 6   6.2  -2.7
```

The aim is to model the acceleration force (`accel`) as a function of time post collision (`times`). The plot below shows the data.

```
ggplot(mcycle, aes(x = times, y = accel)) +
    geom_point() +
    labs(x = "Miliseconds post impact", y = "Acceleration (g)",
         title = "Simulated Motorcycle Accident",
         subtitle = "Measurements of head acceleration")
```



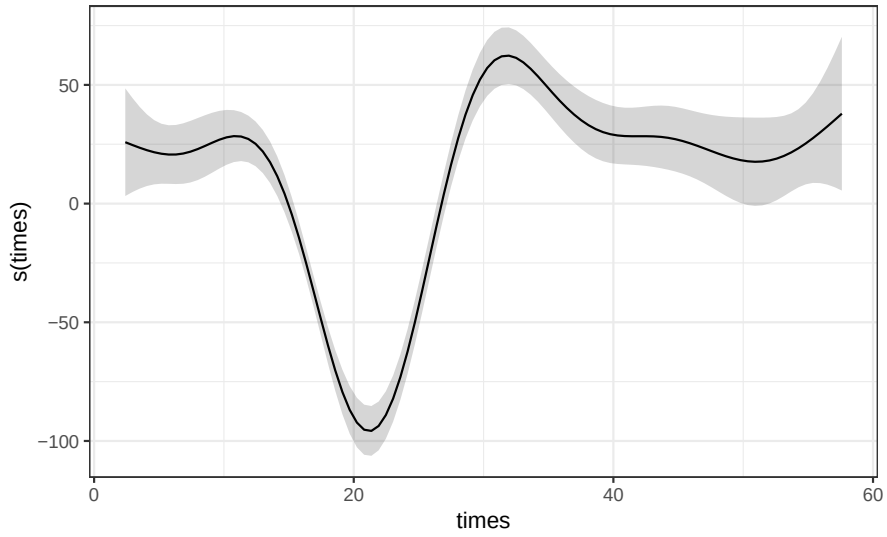
We'll model acceleration as a smooth function of time using a GAM and the default thin plate regression spline basis. This can be done using the `gam()` function in `mgcv` and, for comparison with the fully bayesian model we'll fit shortly, we use `method = "REML"` to estimate the smoothness parameter for the spline in mixed model form using REML.

```
m1 <- gam(accel ~ s(times), data = mcycle, method = "REML")
summary(m1)
#>
#> Family: gaussian
#> Link function: identity
#>
#> Formula:
#> accel ~ s(times)
#>
#> Parametric coefficients:
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept)   -25.55       1.95   -13.1   <2e-16 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Approximate significance of smooth terms:
#>           edf Ref.df    F p-value
#> s(times) 8.62   8.96 53.4  <2e-16 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> R-sq.(adj) =  0.783   Deviance explained = 79.7%
#> -REML = 616.14  Scale est. = 506.35    n = 133
```

As we can see from the model summary, the estimated smooth uses about 8.5 effective degrees of freedom and in the test of zero effect, the null hypothesis is strongly rejected. The fitted spline explains about 80% of the variance or deviance in the data.

To plot the fitted smooth we could use the `plot()` method provided by `mgcv`, but this uses base graphics. Instead we can use the `draw()` method from `schoenberg`, which can currently handle most of the univariate smooths in `mgcv` plus 2-d tensor product smooths

```
draw(m1)
```



2.4 Bayesian Approach

The equivalent model can be estimated using a fully-bayesian approach via the `brm()` function in the `brms` package. In fact, `brm()` will use the smooth specification functions from `mgcv`, making our lives much easier. The major difference though is that you can't use `te()` or `ti()` smooths in `brm()` models; you need to use `t2()` tensor product smooths instead. This is because the smooths in the model are going to be treated as random effects and the model is estimated as a GLMM, which exploits the duality of splines as random effects. In this representation, the wiggly parts of the spline basis are treated as a random effect and their associated variance parameter controls the degree of wiggliness of the fitted spline. The perfectly smooth parts of the basis are treated as a fixed effect. In this form, the GAM can be estimated using standard GLMM software; it's what allows the `gamm4()` function to fit GAMMs using the `lme4` package for example. This is also the reason why we can't use `te()` or `ti()` smooths; those smooths do not have nicely separable penalties which means they can't be written in the form required to be fitted using typical mixed model software.

The `brm()` version of the GAM is fitted using the code below. Note that I have changed a few things from their default values as:

1. the model required more than the default number of MCMC samples - `iter = 4000`,
2. the samples needed thinning to deal with some strong autocorrelation in the Markov chains - `thin = 10`,
3. the `adapt.delta` parameter, a tuning parameter in the NUTS sampler for Hamiltonian Monte Carlo, potentially needed raising - there was a warning about a potential divergent transition but I should have looked to see if it was one or not; instead I just increased the tuning parameter to `0.99`,
4. four chains fitted by default but I wanted these to be fitted using 4 CPU cores,
5. `seed` sets the internal random number generator seed, which allows reproducibility of models, and
6. for this post I didn't want to print out the progress of the sampler - `refresh = 0` - typically you won't want to do this so you can see how sampling is progressing.

The rest of the model is pretty similar to the `gam()` version we fitted earlier. The main difference is that I use the `bf()` function to create a special `brms` formula specifying the model. You don't actually need to do this for such a simple model, but in a later post we'll use this to fit distributional GAMs. Note that I'm

leaving all the priors in the model at the default values. I'll look at defining priors in a later post; for now I'm just going to use the default priors that `brm()` uses.

```
m2 <- brm(bf(accel ~ s(times)),
          data = mcycle, family = gaussian(), cores = 4, seed = 17,
          iter = 4000, warmup = 1000, thin = 10, refresh = 0,
          control = list(adapt_delta = 0.99))
#> Compiling the C++ model
#> Start sampling
```

Once the model has finished compiling and sampling we can output the model summary:

```
summary(m2)
#>  Family: gaussian
#>   Links: mu = identity; sigma = identity
#> Formula: accel ~ s(times)
#>    Data: mcycle (Number of observations: 133)
#> Samples: 4 chains, each with iter = 4000; warmup = 1000; thin = 10;
#>          total post-warmup samples = 1200
#>
#> Smooth Terms:
#>               Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
#> sds(stimes_1)   717.02    184.01   460.62  1147.78        991 1.00
#>
#> Population-Level Effects:
#>           Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
#> Intercept   -25.52      1.98   -29.38   -21.53       1190 1.00
#> stimes_1     17.06     37.98   -59.02    88.95       1144 1.00
#>
#> Family Specific Parameters:
#>       Estimate Est.Error l-95% CI u-95% CI Eff.Sample Rhat
#> sigma    22.73      1.53    20.02    26.16       1153 1.00
#>
#> Samples were drawn using sampling(NUTS). For each parameter, Eff.Sample
#> is a crude measure of effective sample size, and Rhat is the potential
#> scale reduction factor on split chains (at convergence, Rhat = 1).
```

This output details of the model fitted plus parameter estimates (as posterior means), standard errors, (by default) 95% credible intervals and two other diagnostics:

1. `Eff.Sample` is the effective sample size of the posterior samples in the model, and
2. `Rhat` is the potential scale reduction factor or Gelman-Rubin diagnostic and is a measure of how well the chains have converged and ideally should be equal to 1.

The summary includes two entries for the smooth of times:

1. `sds(stimes_1)` is the variance parameter, which has the effect of controlling the wiggliness of the smooth - the larger this value the more wiggly the smooth. We can see that the credible interval doesn't include 0 so there is evidence that a smooth is required over and above a linear parametric effect of times, details of which are given next,
2. `stimes_1` is the fixed effect part of the spline, which is the linear function that is perfectly smooth.

The final parameter table includes information on the variance of the data about the conditional mean of the response.

2.5 Comparison

How does this model compare with the one fitted using gam()? We can use the `gam.vcomp()` function to compute the variance component representation of the smooth estimated via `gam()`. To make it comparable with the value shown for the `brms` model, we don't undo the rescaling of the penalty matrix that `gam()` performs to help with numeric stability during model fitting.

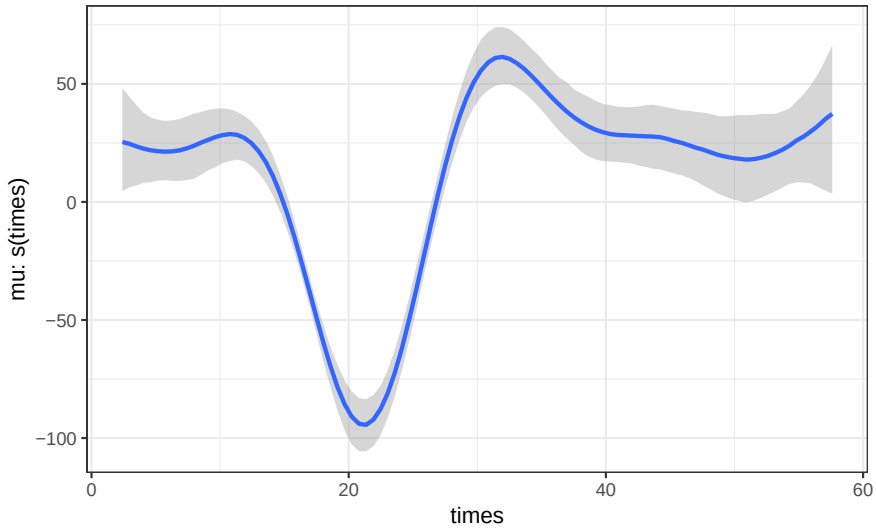
```
gam.vcomp(m1, rescale = FALSE)
#>
#> Standard deviations and 0.95 confidence intervals:
#>
#>           std.dev lower  upper
#> s(times)   807.9 480.7 1357.9
#> scale       22.5  19.9   25.5
#>
#> Rank: 2/2
```

This gives a posterior mean of 807.89 with 95% confidence interval of 480.66–1357.88, which compares well with posterior mean and credible interval of the `brm()` version of 722.44 (450.17 – 1150.27).

The `marginal_smooths()` function is used to extract the marginal effect of the spline.

This function extracts enough information about the estimated spline to plot it using the `plot()` method

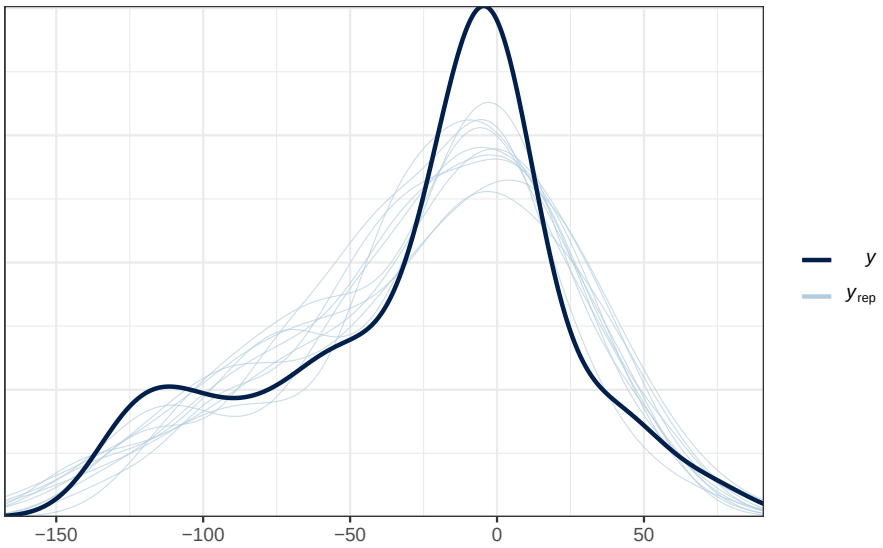
```
msms <- marginal_smooths(m2)
plot(msms)
```



Given the similarity in the variance components of the two models it is not surprising the two estimated smooth also look similar. The `marginal_smooths()` function is effectively the equivalent of the `plot()` method for mgcv-based GAMs.

There's a lot that we can and should do to check the model fit. For now, we'll look at two posterior predictive check plots that `brms`, via the `bayesplot` package (Gabry and Mahr, 2018), makes very easy to produce using the `pp_check()` function.

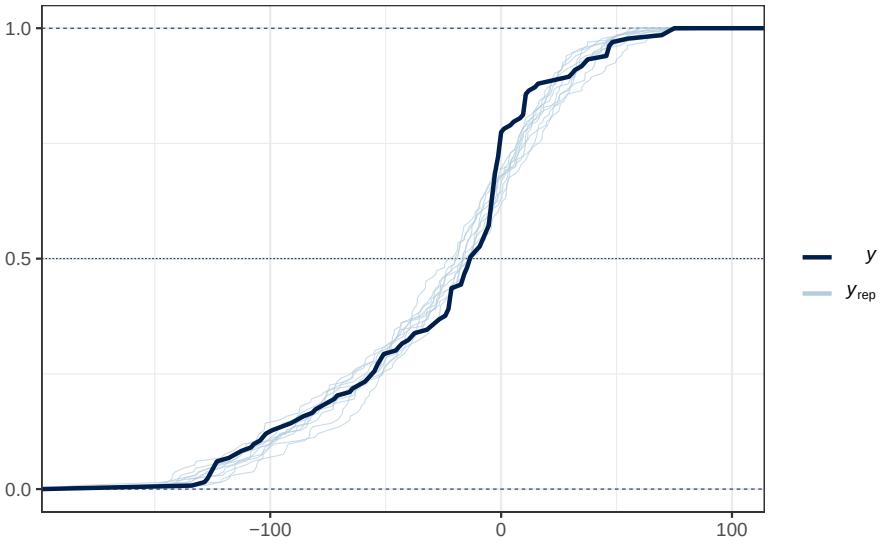
```
pp_check(m2)
#> Using 10 posterior samples for ppc type 'dens_overlay' by default.
```



The default produces a density plot overlay of the original response values (the thick black line) with 10 draws from the posterior distribution of the model. If the model is a good fit to the data, samples of data sampled from it at the observed values of the covariate(s) should be similar to one another.

Another type of posterior predictive check plot is the empirical cumulative distribution function of the observations and random draws from the model posterior, which we can produce with `type = "ecdf_overlay"`.

```
pp_check(m2, type = "ecdf_overlay")
#> Using 10 posterior samples for ppc type 'ecdf_overlay' by default.
```



Both plots show significant deviations between the the posterior simulations and the observed data. The poor posterior predictive check results are in large part due to the non-constant variance of the acceleration data conditional upon the covariate. Both models assumed that the observation are distributed Gaussian with means equal to the fitted values (estimated expectation of the response) with the same variance σ^2. The observations appear to have different variances, which we can model with a distributional model, which allow all parameters of the distribution of the response to be modelled with linear predictors. We'll take a look at these models in a future post.

Chapter 3

Regression with Stan

https://m-clark.github.io/bayesian-basics/models.html

3.1 Regression Models

Now armed with a conceptual understanding of the Bayesian approach, we will actually investigate a regression model using it. To keep things simple, we start with a standard linear model for regression. Later, we will show how easy it can be to add changes to the sampling distribution or priors for alternative modeling techniques. But before getting too far, you should peruse the Modeling Languages section of the appendix to get a sense of some of the programming approaches available. We will be using the programming language Stan via R and the associated R package `rstan`. If you prefer to keep things conceptual rather than worry about the code, you can read through the following data description and then skip to running the model.

3.1.1 Example: Linear Regression Model

In the following we will have some initial data set up and also run the model using the standard lm function for later comparison. I choose simulated data so that not only should you know what to expect from the model, it can easily be modified to enable further understanding. I will also use some matrix operations, and if these techniques are unfamiliar to you, you'll perhaps want to do some refreshing or learning on your own beforehand.

3.1.2 Setup

First we need to create the data we'll use here and for most of the other examples in this document. I use simulated data so that there is no ambiguity about what to expect.

```
library(rstan)
#> Loading required package: ggplot2
#> Registered S3 methods overwritten by 'ggplot2':
#>   method         from
#>   [.quosures     rlang
#>   c.quosures     rlang
#>   print.quosures rlang
#> Loading required package: StanHeaders
#> rstan (Version 2.18.2, GitRev: 2e1f913d3ca3)
#> For execution on a local, multicore CPU with excess RAM we recommend calling
```

```
#> options(mc.cores = parallel::detectCores()).
#> To avoid recompilation of unchanged Stan programs, we recommend calling
#> rstan_options(auto_write = TRUE)
library(rstanarm)
#> Loading required package: Rcpp
#> Registered S3 method overwritten by 'xts':
#>   method     from
#>   as.zoo.xts zoo
#> rstanarm (Version 2.18.2, packaged: 2018-11-08 22:19:38 UTC)
#> - Do not expect the default priors to remain the same in future rstanarm versions.
#> Thus, R scripts should specify priors explicitly, even if they are just the defaults.
#> - For execution on a local, multicore CPU with excess RAM we recommend calling
#> options(mc.cores = parallel::detectCores())
#> - Plotting theme set to bayesplot::theme_default().
#>
#> Attaching package: 'rstanarm'
#> The following object is masked from 'package:rstan':
#>
#>     loo
library(dplyr)
#>
#> Attaching package: 'dplyr'
#> The following objects are masked from 'package:stats':
#>
#>     filter, lag
#> The following objects are masked from 'package:base':
#>
#>     intersect, setdiff, setequal, union

rstan_options(auto_write = TRUE)
options(mc.cores = parallel::detectCores())
```

```
# set seed for replicability
set.seed(8675309)

# create a N x k matrix of covariates
N = 250
K = 3

covariates = replicate(K, rnorm(n=N))
colnames(covariates) = c('X1', 'X2', 'X3')

# create the model matrix with intercept
X = cbind(Intercept=1, covariates)
glimpse(X)
#>  num [1:250, 1:4] 1 1 1 1 1 1 1 1 1 1 ...
#>  - attr(*, "dimnames")=List of 2
#>   ..$ : NULL
#>   ..$ : chr [1:4] "Intercept" "X1" "X2" "X3"
head(X)
#>      Intercept     X1     X2     X3
#> [1,]         1 -0.997 -0.333 -0.391
#> [2,]         1  0.722 -0.174  2.213
#> [3,]         1 -0.617 -0.187 -1.563
```

```
#> [4,]         1  2.029  1.156 -0.770
#> [5,]         1  1.065 -0.938 -0.555
#> [6,]         1  0.987  0.524  1.411
```

```
# create a normally distributed variable that is a function of the covariates
coefs = c(5, .2, -1.5, .9)
mu = X %*% coefs
glimpse(mu)
#>  num [1:250, 1] 4.95 7.4 3.75 2.98 6.12 ...
```

```
# same as
# y = 5 + .2*X1 - 1.5*X2 + .9*X3 + rnorm(N, mean=0, sd=2)

sigma = 2
y = rnorm(N, mu, sigma)
glimpse(y)
#>  num [1:250] 5.21 9.76 3.71 3.7 2.92 ...
```

Just to make sure we're on the same page, at this point we have three covariates, and a y that is a normally distributed, linear function of them, and with standard deviation equal to 2. The population values for the coefficients including the intercept are 5, 0.2, -1.5, and 0.9, though with the noise added, the actual estimated values for the sample are slightly different. Now we are ready to set up an R list object of the data for input into Stan, as well as the corresponding Stan code to model this data. I will show all the Stan code, which is implemented in R via a single character string, and then provide some detail on each corresponding model block. However, the goal here isn't to focus on tools as it is to focus on concepts. Related code for this same model in BUGS and JAGS is provided in the appendix here. I don't think there is an easy way to learn these programming languages except by diving in and using them yourself with models and data you understand.

```
# Run lm for later comparison; but go ahead and examine now if desired
modlm = lm(y~., data=data.frame(X[,-1]))
summary(modlm)
#>
#> Call:
#> lm(formula = y ~ ., data = data.frame(X[, -1]))
#>
#> Residuals:
#>    Min     1Q Median     3Q    Max
#> -6.863 -1.470  0.243  1.421  5.041
#>
#> Coefficients:
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept)   4.8978     0.1284   38.13  < 2e-16 ***
#> X1            0.0841     0.1296    0.65     0.52
#> X2           -1.4686     0.1261  -11.64  < 2e-16 ***
#> X3            0.8196     0.1207    6.79  8.2e-11 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 2.02 on 246 degrees of freedom
#> Multiple R-squared:  0.452,  Adjusted R-squared:  0.446
#> F-statistic: 67.8 on 3 and 246 DF,  p-value: <2e-16
```

The data list for Stan should include any matrix, vector, or value that might be used in the Stan code. For example, along with the data one can include things like sample size, group indicators (e.g. for mixed models) and so forth. Here we can get by with just the sample size (N), the number of columns in the model matrix

(K), the target variable (y) and the model matrix itself (X).

```
# Create the data list object for Stan input
dat = list(N=N, K=ncol(X), y=y, X=X)
```

Next comes the Stan code. In `R2OpenBugs` or `rjags` one would call a separate text file with the code, and one can do the same with rstan, but for our purposes, we'll display it within the R code. The first thing to note then is the model code. Next, Stan has programming blocks that have to be called in order. I will have all of the blocks in the code to note their order and discuss each in turn, even though we won't use them all. Anything following a `//` or `#`, or between `/* */`, are comments pertaining to the code. Assignments in Stan are `=`, while distributions are specified with a `~` , e.g. `y ~ normal(0, 1)` means y is normally distributed with mean 0 and standard deviation of 1.

The primary goal here is to get to the results and beyond, but one should examine the Stan manual for details about the code. In addition, to install rstan one will need to do so via CRAN or GitHub (quickstart guide). It does not require a separate installation of Stan itself, but it does take a couple steps and does require a C++ compiler. Once you have rstan installed it is called like any other R package as will see shortly.

```
# Create the stan model object using Stan's syntax
stanmodelcode = "
data {                      // Data block
  int<lower=1> N;           // Sample size
  int<lower=1> K;           // Dimension of model matrix
  matrix[N, K] X;           // Model Matrix
  vector[N] y;              // Target variable
}

/*
transformed data {          // Transformed data block. Not used presently.
}
*/

parameters {                // Parameters block
  vector[K] beta;           // Coefficient vector
  real<lower=0> sigma;      // Error scale
}

model {                     // Model block
  vector[N] mu;
  mu = X * beta;            // Creation of linear predictor

  // priors
  beta  ~ normal(0, 10);
  sigma ~ cauchy(0, 5);     // With sigma bounded at 0, this is half-cauchy

  // likelihood
  y ~ normal(mu, sigma);
}

/*
generated quantities {      // Generated quantities block. Not used presently.
}
*/
"
```

3.2 Stan Code

The first section is the data block, where we tell Stan the data it should be expecting from the data list. It is useful to put in bounds as a check on the data input, and that is what is being done between the < > (e.g. we should at least have a sample size of 1). The first two variables declared are N and K, both as integers. Next the code declares the model matrix and target vector respectively. As you'll note here and for the next blocks, we declare the type and dimensions of the variable and then its name. In Stan, everything declared in one block is available to subsequent blocks, but those declared in a block may not be used in earlier blocks. Even within a block, anything declared, such as N and K, can then be used subsequently, as we did to specify dimensions of the model matrix X.

For a reference, the following is from the Stan manual, and notes variables of interest and the associated blocks where they would be declared.

The transformed data block is where you could do such things as log or center variables and similar, i.e. you can create new data based on the input data or just in general. If you are using R though, it would almost always be easier to do those things in R first and just include them in the data list. You can also declare any unmodeled parameters here, e.g. those you want fixed at some value.

The primary parameters of interest that are to be estimated go in the parameters block. As with the data block you can only declare these variables, you cannot make any assignments. Here we note the β and σ to be estimated, with a lower bound of zero on the latter. In practice, you might prefer to split out the intercept or other coefficients to be modeled separately if they are on notably different scales.

The transformed parameters block is where optional parameters of interest might be included. What might go here is fairly open, but for efficiency's sake you will typically want to put things only of specific interest that are dependent on the parameters block. These are evaluated along with the parameters, so if the objects are not of special interest you can instead generate them in the model or generated quantities block to save time.

The model block is where your priors and likelihood are specified, along with the declaration of any variables necessary. As an example, the linear predictor is included here, as it will go towards the likelihood. Note that we could have instead put the linear predictor in the transformed parameters section, but this would slow down the process, and again, we're not so interested in those specific values.

I use a normal prior for the coefficients with a zero mean and a very large standard deviation to reflect my notable ignorance here. For the
σ estimate I use a Cauchy distribution. Many regression examples using BUGS will use an inverse gamma prior, which is perfectly okay for this model, though it would not work so well for other variance parameters. Had we not specified anything for the prior distribution for the parameters, vague (discussed more in the Choice of Prior section), uniform distributions would be the default. The likelihood is specified in a similar manner as one would with R. BUGS style languages would actually use `dnorm` as in R, though Stan uses normal for the function name.

Finally, we get to the generated quantities, which is kind of a fun zone. Anything you want to calculate can go here - predictions on new data, ratios of parameters, how many times a parameter is greater than `x`, transformations of parameters for reporting purposes, and so forth. We will demonstrate this later.

3.3 Running the Model

Now that we have an idea of what the code is doing, let's put it to work. Bayesian estimation, like maximum likelihood, starts with initial guesses as starting points and then runs in an iterative fashion, producing simulated draws from the posterior distribution at each step, and then correcting those draws until finally getting to some target, or stationary distribution. This part is key and different from classical statistics. We are aiming for a distribution, not a point estimate.

The simulation process is referred to as Markov Chain Monte Carlo, or MCMC for short. The specifics of this process are what sets many of the Bayesian programming languages/approaches apart, and something we will cover in more detail in a later section (see Sampling Procedure). In MCMC, all of the simulated draws from the posterior are based on and correlated with previous draws, as the process moves along the path toward a stationary distribution. We will typically allow the process to warm up, or rather get a bit settled down from the initial starting point, which might be way off, and thus the subsequent estimates will also be way off for the first few iterations. Rest assured, assuming the model and data are otherwise acceptable, the process will get to where it needs to go. However, as a further check, we will run the whole thing multiple times, i.e. have more than one chain. As the chains will start from different places, if multiple chains get to the same place in the end, we can feel more confident about our results.

While this process may sound like it might take a long time to complete, for the following you'll note that it will likely take more time for Stan to compile its code to C++ than it will to run the model, and on my computer each chain only takes only a little more than a second. However, the Bayesian approach used to take a very long time even for a standard regression such as this, and that is perhaps the primary reason why Bayesian analysis only caught on in the last couple decades; we simply didn't have the machines to do it efficiently. Even now though, for highly complex models and large data sets it can still take a long time to run, though typically not prohibitively so.

In the following code, we note the object that contains the Stan model code, the data list, how many iterations we want (5000), how long we want the process to run before we start to keep any estimates (`warmup=2500`), how many of the post-warmup draws of the posterior we want to keep (`thin=10` means every tenth draw), and the number of chains (`chains=4`). In the end, we will have four chains of 100028 draws from the posterior distribution of the parameters. Stan spits out a lot of output to the R console even with `verbose = FALSE`, and I omit it here, but you will see some initial info about the compiling process, updates as each chain gets through 10% of iterations specified in the iter argument, and finally an estimate of the elapsed time. You may also see informational messages which, unless they are highly repetitive, should not be taken as an error.

```
library(rstan)

### Run the model and examine results
fit = stan(model_code = stanmodelcode,
           data = dat,
           iter = 5000,
           warmup = 2500,
           thin = 10,
           chains = 4)
```

With the model run, we can now examine the results. In the following, we specify the digit precision to display, which parameters we want (not necessary here), and which quantiles of the posterior draws we want, which in this case are the median and those that would produce a 95% interval estimate.

```
# summary
print(fit, pars=c('beta', 'sigma'), digits=3, prob=c(.025,.5,.975))
#> Inference for Stan model: 207e716b728c6bce448b2aa8a19bbe44.
#> 4 chains, each with iter=5000; warmup=2500; thin=10;
#> post-warmup draws per chain=250, total post-warmup draws=1000.
#>
#>           mean se_mean    sd   2.5%    50%  97.5% n_eff Rhat
#> beta[1]  4.894   0.005 0.130  4.642  4.890  5.134   713    1
#> beta[2]  0.086   0.004 0.132 -0.169  0.081  0.336   996    1
#> beta[3] -1.463   0.004 0.122 -1.703 -1.466 -1.232   955    1
#> beta[4]  0.822   0.004 0.124  0.584  0.822  1.069  1038    1
#> sigma    2.034   0.003 0.092  1.858  2.033  2.211   960    1
#>
```

```
#> Samples were drawn using NUTS(diag_e) at Wed Sep 18 14:39:40 2019.
#> 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).
```

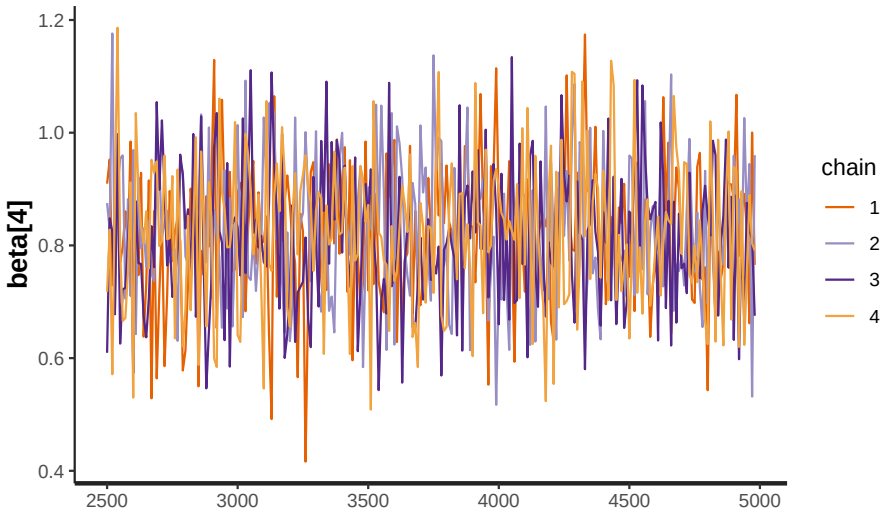
So far so good. The mean estimates reflect the mean of posterior draws for the parameters of interest, and are the **typical coefficients** reported in standard regression analysis. The 95% probability, or, credible intervals are worth noting, because they are not confidence intervals as you know them. There is no repeated sampling interpretation here. The probability interval is more intuitive. It means simply that, based on the results of this model, there is a 95% chance the true value will fall between those two points. The other values printed out I will return to in just a moment.

Comparing the results to those from R's `lm` function, we can see we obtain similar estimates, as they are identical to two decimal places. In fact, had we used uniform priors, we would be doing essentially the same model as what is being conducted with standard maximum likelihood estimation. Here, we have a decent amount of data for a model that isn't complex, so we would expect the likelihood to notably outweigh the prior, as we demonstrated previously with our binomial example.

```
summary(modlm)
#>
#> Call:
#> lm(formula = y ~ ., data = data.frame(X[, -1]))
#>
#> Residuals:
#>     Min      1Q  Median      3Q     Max
#> -6.863 -1.470  0.243  1.421  5.041
#>
#> Coefficients:
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept)   4.8978     0.1284   38.13  < 2e-16 ***
#> X1            0.0841     0.1296    0.65     0.52
#> X2           -1.4686     0.1261  -11.64  < 2e-16 ***
#> X3            0.8196     0.1207    6.79  8.2e-11 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 2.02 on 246 degrees of freedom
#> Multiple R-squared:  0.452,  Adjusted R-squared:  0.446
#> F-statistic: 67.8 on 3 and 246 DF,  p-value: <2e-16
```

But how would we know if our model was working out okay otherwise? There are several standard diagnostics, and we will talk about them in more detail in the next section, but let's take a look at some presently. In the summary, `se_mean` is the Monte Carlo error, and is an estimate of the uncertainty contributed by only having a finite number of posterior draws. `n_eff` is effective sample size given all chains, and essentially accounts for autocorrelation in the chain, i.e. the correlation of the estimates as we go from one draw to the next. It actually doesn't have to be very large, but if it was small relative to the total number of draws desired that might be cause for concern. `Rhat` is a measure of how well chains mix, and goes to 1 as chains are allowed to run for an infinite number of draws. In this case, `n_eff` and `Rhat` suggest we have good convergence, but we can also examine this visually with a traceplot.

```
# Visualize
stan_trace(fit, pars=c('beta[4]'))
```



I only show one parameter for the current demonstration, but one should always look at the traceplots for all parameters. What we are looking for after the warmup period is a "fat hairy caterpillar" or something that might be labeled as "grassy", and this plot qualifies as such. One can see that the estimates from each chain find their way from the starting point to a more or less steady state quite rapidly (initial warmup iterations in gray). Furthermore, all three chains, each noted by a different color, are mixing well and bouncing around the same conclusion. The statistical measures and traceplot suggest that we are doing okay.

The Stan development crew has made it easy to interactively explore diagnostics via the `shinystan` package, and one should do so with each model. In addition, there are other diagnostics available in the coda package, and Stan model results can be easily converted to work with it. The following code demonstrates how to get started.

```
library(coda)
#>
#> Attaching package: 'coda'
#> The following object is masked from 'package:rstan':
#>
#>     traceplot
betas = extract(fit, pars='beta')$beta
betas.mcmc = as.mcmc(betas)
# plot(betas.mcmc)
```

```
if (interactive()) launch_shinystan(fit)
```

So there you have it. Aside from the initial setup with making a data list and producing the language-specific model code, it doesn't necessarily take much to running a Bayesian regression model relative to standard models. The main thing perhaps is simply employing a different mindset, and interpreting the results from within that new perspective. For the standard models you are familiar with, it probably won't take too long to be as comfortable here as you were with those, and now you will have a flexible tool to take you into new realms with deeper understanding.