Getting Stan set up on your computer

Stan uses Hamiltonian Monte Carlo to sample from the posterior. (For more on HMC see Michael Betancourt's paper "A Conceptual Introduction to Hamiltonian Monte Carlo"). HMC has many advantages over Gibbs and Metropolis-Hastings samplers. I'm not terribly qualified to make this argument, but here is an intro to the discussion: https://elevanth.org/2017_09_28_why_walk_when_flow.html.

In addition to the efficiency gained by using HMC, Stan code gets translated to `C++` and compiled into an executable program. This means that Stan programs are very fast (although caution: they won't feel fast compared to some of the maximum likelihood algorithms, but this is not an apples to apples comparison). This can lead to some difficulty setting up because we will need to install a `C++` toolchain. Let's get started.

Calling Stan from R

Since we are an `R` shop here in hacky hours, we will use `R` to interact with Stan (but note that you can use Python and the command line as well, and maybe even Julia, though I'm not sure on that one). There are two main packages to do this: `RStan` and `CmdStanR`. We will be using `CmdStanR` in hacky hours, it gives us access to the most up to date version of Stan, but feel free to play with `RStan`! It's been a long time since I've installed Stan, but I think we can use `CmdStanR` to do most of it. So let's install `CmdStanR` first!

Installing CmdStanR

```r
install.packages("cmdstanr", repos = c('https://stan-dev.r-universe.dev',
getOption("repos")))
```

Hopefully that installed correctly! If it did, we can check to see if the `C++` toolchain is already set up on your computer.

Installing the C++ toolchain

It's possible that you already have the toolchain setup, check with:

```r
cmdstanr::check_cmdstan_toolchain(fix = T)
```

```
The C++ toolchain required for CmdStan is setup properly!
```

My toolchain is good to go! If yours is as well, you can skip down to "Installing CmdStan", if not , the next step is to setup the toolchain—*Deep breaths!* **Linux** and **Mac** users, follow the instructions here: https://mc-stan.org/install/#prerequisite-c17-toolchain and then run the above command again.

For **Windows** folks, we need to go on a bit of an adventure. Go here https://cran.r-project.org/bin/windows/Rtools/rtools44/rtools.html and download and run the Rtools44 installer. Once it is installed, you will need to update your `PATH` variable, see the instructions here: https://helpdeskgeek.com/add-windows-path-environment-variable/ and add: `C:rtools44\usr\bin` to the paths list.

Once you have done that, re-run `cmdstanr::check_cmdstan_toolchain(fix = T)`. Hopefully it will set up your toolchain. If you get errors, google them and try to solve the problem. But if you end up with a problem you can't figure out, let me know so we can try to sort it out.

Once the toolchain is good, you can install `CmdStan`

Installing CmdStan

You can install `CmdStan` using `CmdStanR` by:

```r
cmdstanr::install_cmdstan(cores = 2)
```

This will take a little bit. Once it is complete, let's check that everything is ok. `CmdStan` comes installed with an example model. Let's look at it:

```r
library(cmdstanr)
# save this as an object called file
file <- file.path(cmdstan_path(), "examples", "bernoulli", "bernoulli.stan")
# translate and compile the model to C++
mod <- cmdstan_model(file)
```

No we have a stan model ready to go! Let's see what it looks like:

```r
mod$print()
```

```
data {
  int<lower=0> N;
  array[N] int<lower=0, upper=1> y;
}
parameters {
  real<lower=0, upper=1> theta;
}
model {
  theta ~ beta(1, 1); // uniform prior on interval 0,1
  y ~ bernoulli(theta);
}
```

Let's give it some data and run it (we'll talk about all this more in hacky hours, let's just make sure it runs for now):

```r
dat <- list(
  N = 10,
  y = c(0,1,0,0,0,0,0,0,0,1)
)

fit <- mod$sample(
  data = dat,
  chains = 4,
  parallel_chains = 4,
  refresh = 1000
)
```

```
Running MCMC with 4 parallel chains...

Chain 1 Iteration:    1 / 2000 [  0%]  (Warmup)
Chain 1 Iteration: 1000 / 2000 [ 50%]  (Warmup)
Chain 1 Iteration: 1001 / 2000 [ 50%]  (Sampling)
Chain 1 Iteration: 2000 / 2000 [100%]  (Sampling)
Chain 2 Iteration:    1 / 2000 [  0%]  (Warmup)
Chain 2 Iteration: 1000 / 2000 [ 50%]  (Warmup)
Chain 2 Iteration: 1001 / 2000 [ 50%]  (Sampling)
Chain 2 Iteration: 2000 / 2000 [100%]  (Sampling)
Chain 3 Iteration:    1 / 2000 [  0%]  (Warmup)
Chain 3 Iteration: 1000 / 2000 [ 50%]  (Warmup)
Chain 3 Iteration: 1001 / 2000 [ 50%]  (Sampling)
Chain 3 Iteration: 2000 / 2000 [100%]  (Sampling)
Chain 4 Iteration:    1 / 2000 [  0%]  (Warmup)
Chain 4 Iteration: 1000 / 2000 [ 50%]  (Warmup)
Chain 4 Iteration: 1001 / 2000 [ 50%]  (Sampling)
Chain 4 Iteration: 2000 / 2000 [100%]  (Sampling)
Chain 1 finished in 0.0 seconds.
Chain 2 finished in 0.0 seconds.
Chain 3 finished in 0.0 seconds.
Chain 4 finished in 0.0 seconds.

All 4 chains finished successfully.
Mean chain execution time: 0.0 seconds.
Total execution time: 0.5 seconds.
```

Awesome, everything looks like it is working well. Here are some other packages we should install that make working with Stan and the posterior a little easier:

```
# install these packages if you haven't yet
install.packages("posterior")
install.packages("bayesplot")
```