Bayesian Estimation of Recruitment Trends in Alberta (BERTA) Tutorial

Christopher L. Cahill

31 December 2021

Goals

- Import the FWIN and stocking data, and demonstrate how to add new data to the original data used in Cahill et al. 2021
- Become familiar with the `run.R` script
- Gain intuition for how .R and .stan scripts are working together to subset data, fit a Bayesian population dynamics model to those data, and then save a model fit with a unique file name identifier
- Improve understanding of tidyverse sub-setting, `get_fit()`, `future_pwalk()`, and `plan()` functions
- Launching shinystan diagnostics
- Practice debugging using `browser()`

Packages

Let's load the packages we will use:

```
library(tidyverse)
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.1 --

## v ggplot2 3.3.5     v purrr   0.3.4
## v tibble  3.1.3     v dplyr   1.0.7
## v tidyr   1.1.3     v stringr 1.4.0
## v readr   2.0.1     v forcats 0.5.1

## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

```
library(rstan)
```

```
## Warning: package 'rstan' was built under R version 4.1.2

## Loading required package: StanHeaders

## rstan (Version 2.21.3, 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)

## Do not specify '-march=native' in 'LOCAL_CPPFLAGS' or a Makevars file

##
## Attaching package: 'rstan'

## The following object is masked from 'package:tidyr':
##
##     extract
```

```
library(purrr)
library(furrr)
```

```
## Loading required package: future
```

```
library(future)
```

Data

We will work with the Fall Walleye Index Netting (FWIN) dataset used in Cahill et al. (2021), which included all Alberta lakes with ≥ 3 FWIN surveys during 2000-2018. Life history parameters ω, A_{50}, L_∞, vbk, and β_{wl} were obtained using hierarchical modeling methods described in Cahill et al. (2020), and these values represent lake-specific averages.

```
data <- readRDS(here::here("data/BERTA-wide-0-25.rds"))
glimpse(data)
```

```
## Rows: 236
## Columns: 46
## Groups: name [55]
## $ WBID       <int> 3526, 3526, 3526, 3526, 3916, 3916, 3916, 3969, 3969, 3969,~
## $ year       <dbl> 5, 6, 13, 19, 6, 11, 14, 9, 12, 15, 17, 7, 12, 17, 4, 11, 1~
## $ name       <chr> "milk river ridge reservoir", "milk river ridge reservoir",~
## $ nnet       <int> 18, 20, 11, 12, 8, 12, 10, 11, 12, 12, 12, 18, 15, 15, 4, 6~
## $ n          <int> 201, 283, 232, 158, 189, 117, 132, 357, 171, 186, 201, 373,~
## $ effort     <dbl> 18.0, 20.0, 11.0, 12.0, 8.0, 6.0, 5.0, 11.0, 6.0, 6.0, 6.0,~
## $ X_TTM_c    <dbl> 676088.4, 676088.4, 676088.4, 676088.4, 652638.4, 652638.4,~
## $ Y_TTM_c    <dbl> 5469124, 5469124, 5469124, 5469124, 6050150, 6050150, 60501~
## $ p_aged     <dbl> 1.0000000, 1.0000000, 1.0000000, 0.9430380, 1.0000000, 0.98~
## $ omega      <dbl> 12.22278, 12.22278, 12.22278, 12.22278, 13.93477, 13.93477,~
## $ linf       <dbl> 56.72357, 56.72357, 56.72357, 56.72357, 51.38603, 51.38603,~
## $ vbk        <dbl> 0.2154797, 0.2154797, 0.2154797, 0.2154797, 0.2711781, 0.27~
## $ a50        <dbl> 7, 7, 7, 7, 4, 4, 4, 5, 5, 5, 5, 5, 5, 5, 4, 4, 4, 4, 5, 5,~
## $ beta_wl    <dbl> 3.409773, 3.409773, 3.409773, 3.409773, 3.100920, 3.100920,~
## $ X_long     <dbl> -112.5735, -112.5735, -112.5735, -112.5735, -112.6363, -112~
## $ Y_lat      <dbl> 49.36904, 49.36904, 49.36904, 49.36904, 54.59751, 54.59751,~
```

```
## $ Area_Ha    <dbl> 1355.0, 1355.0, 1355.0, 1355.0, 527.1, 527.1, 527.1, 970.7,~
## $ DD5        <int> 1605, 1605, 1605, 1605, 1293, 1293, 1293, 1293, 1293, 1293,~
## $ Max_Depth  <dbl> NA, NA, NA, NA, 27.5, 27.5, 27.5, 27.4, 27.4, 27.4, 27.4, 1~
## $ Mean_Depth <dbl> NA, NA, NA, NA, 14.3, 14.3, 14.3, 9.2, 9.2, 9.2, 9.2, 6.9, ~
## $ `1`        <dbl> 23, 21, 5, 2, 0, 4, 4, 2, 10, 3, 23, 24, 5, 6, 1, 18, 0, 0,~
## $ `2`        <dbl> 13, 67, 43, 6, 0, 4, 7, 24, 4, 14, 41, 11, 5, 10, 7, 14, 0,~
## $ `3`        <dbl> 18, 55, 13, 8, 5, 17, 44, 53, 5, 12, 14, 16, 5, 4, 15, 13, ~
## $ `4`        <dbl> 30, 46, 10, 11, 3, 16, 13, 78, 4, 8, 17, 50, 11, 2, 7, 5, 1~
## $ `5`        <dbl> 27, 35, 17, 13, 41, 3, 8, 15, 12, 0, 8, 55, 5, 37, 17, 9, 1~
## $ `6`        <dbl> 26, 20, 16, 30, 80, 1, 23, 6, 26, 11, 7, 38, 5, 18, 0, 7, 7~
## $ `7`        <dbl> 16, 11, 18, 40, 45, 2, 11, 8, 37, 1, 4, 27, 6, 15, 0, 5, 8,~
## $ `8`        <dbl> 23, 10, 14, 11, 9, 1, 3, 4, 4, 12, 4, 39, 4, 5, 2, 1, 6, 45~
## $ `9`        <dbl> 16, 7, 20, 8, 2, 1, 1, 14, 1, 23, 0, 7, 8, 12, 0, 4, 4, 85,~
## $ `10`       <dbl> 6, 1, 14, 3, 2, 18, 1, 26, 3, 38, 7, 10, 17, 7, 3, 4, 3, 3,~
## $ `11`       <dbl> 0, 1, 3, 6, 0, 30, 1, 38, 6, 9, 16, 23, 6, 1, 0, 5, 3, 1, 0~
## $ `12`       <dbl> 0, 0, 0, 3, 0, 12, 0, 45, 2, 3, 25, 39, 1, 3, 0, 2, 1, 0, 0~
## $ `13`       <dbl> 0, 0, 1, 2, 1, 0, 5, 3, 14, 2, 5, 3, 9, 0, 2, 0, 0, 0, 0, 0~
## $ `14`       <dbl> 0, 1, 1, 0, 1, 1, 6, 18, 7, 5, 3, 1, 0, 4, 0, 1, 1, 1, 0, 0~
## $ `15`       <dbl> 0, 0, 0, 1, 0, 0, 2, 10, 5, 4, 1, 0, 1, 5, 0, 0, 0, 0, 0, 0~
## $ `16`       <dbl> 0, 0, 0, 1, 0, 0, 2, 2, 3, 8, 2, 0, 1, 4, 0, 0, 0, 0, 0, 0,~
## $ `17`       <dbl> 0, 0, 0, 0, 0, 0, 0, 3, 3, 12, 3, 0, 7, 3, 0, 0, 0, 0, 0, 0~
## $ `18`       <dbl> 0, 0, 0, 0, 0, 0, 0, 1, 2, 10, 3, 0, 1, 4, 0, 0, 0, 0, 0, 0~
## $ `19`       <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 1, 0, 3, 0, 0, 0, 0, 0, 0, 0,~
## $ `20`       <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 1, 2, 4, 0, 0, 1, 0, 0, 0, 0, 0, 0,~
## $ `21`       <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,~
## $ `22`       <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 2, 0, 0, 0, 0, 0, 0,~
## $ `23`       <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,~
## $ `24`       <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,~
## $ `25`       <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,~
## $ lake       <dbl> 1, 1, 1, 1, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 5, 5, 5, 5, 6, 6,~
```

Now read in the stocking data, which was used for plotting and not fitted in the .stan model. Note these stocking records go from 1980-2018, and values represent the number of Walleye stocked per hectare:

```
stocking <- readRDS(here::here("data/stocking_matrix_ha.rds"))
glimpse(stocking)
```

```
##  num [1:106, 1:39] 0 0 0 0 0 0 0 0 0 0 ...
##  - attr(*, "dimnames")=List of 2
##   ..$ : chr [1:106] "berry creek reservoir" "jensen reservoir" "milk river ridge reservoir" "travers
##   ..$ : NULL
```

Adding new data to the .rds

Here are a few important notes to how one might add new data into this .rds file setup.

First, `year` in the BERTA .rds is indexed from year 2000, which was when the first FWIN survey in Alberta occurred. Thus, year = 1 is 2000, year = 2 is 2001, etc.

Consequently, if one wanted to add new data to the `BERTA-wide-0-25.rds` file from e.g., 2022, they would create a new row in the .rds that

- declares the value in the `year` slot as 2022 - 2000 + 1 = 23

- repeats the name for that lake as per the names in `name` slot
- sets the number of nets (nnets) as the sum of full nets (each net equal to 1) and half nets (each worth 0.5)
- calculates the proportion of fish with age structure estimates in that survey in the `p_aged` slot
- puts the catch at each age from 0 to 25 in columns labelled '0', '1', ..., '25'
- all other columns just need to be copied for the lake of interest

To add new information to the stocking dataframe, note that rows are lakes and columns are the number of Walleye stocked per hectare during 1980-2018 in that lake. Thus, if you want to add stocking values after 2018 you'd add new columns for the years of interest. These are indexed from 1980 because BERTA hindcasts to reconstruct recruitment trends in years pre-FWIN.

Create a wrapper function

Once the data are read into R, we can write a wrapper function called `get_fit()` that does the following:

- subsets all the data to data for a specific lake
- creates the appropriate tagged list data structures as input into the Stan model
- creates appropriate input for parameters for our Stan model
- runs the model for a particular combination of priors (e.g., which α_r), structural control parameters (e.g., Ricker vs. Beverton-Holt stock-recruit), and Stan run parameter values (i.e., number of iterations, warmup period, number of chains)
- saves this model run with a unique identifier file name

This may seem like a pain, but coding this way will help us later on when we need to run multiple models on different data sets. The wrapper function is:

```
get_fit <- function(which_lake = "pigeon lake",
                    rec_ctl = c("bev-holt", "ricker"),
                    cr_prior = c(6, 12),
                    n_iter = n_iter, n_chains = n_chains,
                    n_warmup = n_iter / 2,
                    ...) {
  # Some outrageous amount of code here in the run.R script
}
```

There is too much code in this function to show it all here. The entire function can be seen at:

https://github.com/ChrisFishCahill/managing-irf-complexity/blob/main/r-code/run.R#L28-L187

Let's break `get_fit()` down into pieces. Note these next lines won't run because we are jumping inside a function.

The first few lines of `get_fit()` are:

```
rec_ctl <- match.arg(rec_ctl)
cat(
    crayon::green(
      clisymbols::symbol$tick
    ),
    fitted = "model fitted = ", which_lake, rec_ctl,
    sep = " "
)
cat("\n")
```

This code is ensuring that the variable rec_ctl is either "ricker" or "bev-holt" via `match_arg()`. Next, this code is printing which lake and recruitment model is being fitted to the console via cat(). This isn't super important and realistically can be omitted or ignored by most users.

The next chunk of code is:

```
#filter the run data from all data, re-order it
run_data <- data %>% filter(name %in% which_lake)
 run_data <-
    within(run_data, lake <-
      as.numeric(interaction(
        run_data$WBID,
        drop = TRUE, lex.order = F
    )))
run_data <- run_data[order(run_data$lake), ]
```

This first line subsets all available data via `%>%` and `filter()`, and returns data corresponding to the variable `which_lake`.

The next few lines of code are simply re-ordering the data to ensure FWIN catch data for a given lake are in ascending order in terms of years via `within()` and `order()`

Once we have subset the FWIN data, we subset the stocking data in a similar way:

```
# stocking stuff was run in different versions, now just for plotting:
run_stocking <- stocking[which(rownames(stocking) %in% which_lake), ]

# Add ten years of zero for short term projections
proj_stock <- rep(0, 10)
run_stocking <- round(c(run_stocking, proj_stock)) #add to stocking data (for plots)
```

The extra lines for `proj_stock` simply add zeros to the end of the stocking data for a given lake. These were important when we were attempting to fit a stocking survival parameter in previous versions of the model, but are no longer used except for plotting.

The next chunk of `get_fit()` is

```
# Set up the Rbar years
suppressMessages(
  survey_yrs <- run_data %>%
      group_by(lake) %>%
      summarise(
        min_yr = min(year) + length(initial_yr:(t - 1)),
        max_yr = max(year) + length(initial_yr:(t - 1))
      ) %>%
  as.numeric()
)
survey_yrs <- survey_yrs[2:3]
# summarize the life history relationships
suppressMessages(
  life_hist <- run_data %>%
    group_by(lake) %>%
    summarize(
      a50 = unique(a50),
      vbk = unique(vbk),
```

```
      linf = unique(linf),
      wl_beta = unique(beta_wl)
    )
)
```

This code is calculating the first and last FWIN survey years for the lake of interest (top chunk) and summarizing that lake's life history parameters (bottom chunk). These chunks are wrapped in `supressMessages()` because dplyr was returning some goofy messages and it was driving me bonkers.

Now that we have subsetted the data and calculated the necessary values from all available data, we can create a tagged data list for Stan:

```
Fseq <- seq(from = 0.01, to = 1.0, by = 0.01)

# declare the tagged data list for stan
stan_data <- list(
  n_surveys = nrow(run_data),
  n_ages = length(Ages),
  n_obs = nrow(run_data) * length(Ages),
  n_years = length(initial_yr:2028),
  n_lakes = length(unique(run_data$lake)),
  caa = run_data[, which(colnames(run_data) %in% Ages)],
  prop_aged = run_data$p_aged,
  effort = run_data$effort,
  lake = run_data$lake,
  year = run_data$year + length(initial_yr:(t - 1)),
  ages = Ages,
  survey_yrs = survey_yrs,
  which_year = 1996 - initial_yr + 2, # which integer corresponds to year = 1997
  v_prior_early = 0.3,
  v_prior_late = 0.1,
  prior_sigma_v = c(0.1, 0.5),
  R0_mean = log(6),
  R0_sd = log(3),
  ar_sd = 0.1,
  prior_mean_w = 0,
  prior_sigma_w = 1.2,
  vbk = life_hist$vbk,
  linf = life_hist$linf,
  a50 = life_hist$a50,
  wl_beta = life_hist$wl_beta,
  lbar = 57.57, # From cahill et al. 2020
  M = 0.1,
  theta = 0.85, # Lorenzen M exponent
  phi = 2.02, # vulnerability parameter (nets)
  psi = 2, # vulnerability parameter (angling)
  G_bound = c(0, Inf),
  get_SSB_obs = 1L,
  obs_cv_prior = 0.15,
  SSB_penalty = 0,
  prior_sigma_G = 1,
  Rinit_ctl = 0,
  length_Fseq = length(Fseq),
  Fseq = Fseq,
```

```
  rec_ctl = ifelse(rec_ctl == "ricker", 0, 1),
  cr_prior = cr_prior
)
```

The stuff in `stan_data` corresponds to the inputs in the data{} section of the `BERTA_singe_lake.stan` file that we will ultimately call below. Before we do that, we need to declare one more function to pass start values for our estimated parameters to Stan:

```
# create a function for start values
vk <- c(0.3, 0.3)
inits <- function() {
  list(
    v = jitter(vk, amount = 0.1),
    R0 = jitter(15, amount = 2),
    G = jitter(1, amount = 0.1),
    w = jitter(rep(
      0,
      length(1980:2028)-2
    ),
    amount = 0.1
    ),
    sigma_w = jitter(0.5, amount = 0.05),
    ar = jitter(0.5, amount = 0.01)
  )
}
```

Technically, this `inits()` function is not needed to run the model because rstan will randomly choose start values if the user does not declare them explicitly, but specifying starting values *greatly* improves the numerical performance of the chains in this particular model. Also note that `jitter()` simply "jitters" the values by a small amount:

```
# run it once
inits()$v
```

```
## [1] 0.2200028 0.3828572
```

```
# run it again and note slighly different results vs. previous call
inits()$v
```

```
## [1] 0.3017774 0.3278404
```

This is useful for initializing starting values for parameters on different chains at slightly different locations.

Now we have successfully subset the data for a specific lake, created the appropriate `stan_data` list, and an `inits()` function to declare starting values for the Bayesian model. We now call a compiled Stan model `m` via:

```
# run the model
fit <-
  rstan::sampling(
    m,
    data = stan_data,
```

```
    pars =
      c(
        "ar_mean_kick", "F_ratio", "Fmsy", "MSY",
        "G", "cr", "ar", "SPR", "br",
        "SBR", "sbr0_kick", "R0", "v", "SSB",
        "R2", "SSB_obs", "caa_pred", "b_ratio", "w"
      ),
    iter = n_iter,
    warmup = n_warmup,
    chains = n_chains,
    init = inits,
    control = list(
      adapt_delta = 0.999,
      max_treedepth = 15
    )
  )
```

Here, the `pars` argument simply tells Stan which parameters to monitor, `iter` determines the number of iterations to run the model for, `chains` specifies the number of chains to run, and `init` is an argument that accepts a function or list of start values for parameters in your model.

The parameters in the `control` part of `sampling()` control technical aspects of the MCMC sampling algorithm. **We should discuss selection of the control arguments in person.**

Once the stan model runs, we save a fit via:

```
# create name and save .rds files for each run
if (rec_ctl == "ricker") {
  my_name <- paste0(which_lake, "_ricker.rds")
}
if (rec_ctl == "bev-holt") {
  my_name <- paste0(which_lake, "_bh.rds")
}
stan_file <- "fits/"
stan_file <- str_c(stan_file, my_name)
stan_file <- stan_file %>% gsub(" ", "_", .)
if (cr_prior == 6) {
  stan_file <- stan_file %>% gsub(".rds", "_cr_6.rds", .)
}
if (cr_prior == 12) {
  stan_file <- stan_file %>% gsub(".rds", "_cr_12.rds", .)
}
if (file.exists(stan_file)) {
  return(NULL)
} else {
  saveRDS(fit, file = stan_file)
}
```

There is a lot of hogwash going on in here about manipulating strings via `gsub()` and `paste0()`, but what you need to know is that this chunk of code saves the Bayesian model fit to a specific .rds file in the fits folder. These .rds files have names like `pigeon_lake_ricker_cr6.rds`. We can call a specific .rds file for plotting or harvest control rule development or whatever later on. Note how this chunk of code will fail to save a fit for a model run using informative priors for compensation ratio other than 6 or 12. How could you fix this issue?

Now that we've done all this work, it is dead simple to fit a single BERTA model to any lake in the FWIN database:

```
# do a short run with one chain for a lake chosen at random
fit <- get_fit(which_lake = "unnamed 5",
               rec_ctl = "ricker",
               cr_prior = 6,
               n_iter = 30, n_chains = 1,
               n_warmup = 15)
```

That's it for `get_fit()`.

Functional programming with {purrr} and {furrr}

`purrr` and `furrr` are functional programming toolkits for R.

purrr

This package is extremely powerful and helps with iteration. If we wanted to iterate a simple function for multiple inputs we might go about it in base R by

- writing a simple function
- looping through that function
- examining the output of looping through that function

The following silly example demonstrates this:

```
# let's write a silly function
silly_func <- function(x){
  result <- x + 1
  result
}

# does the function behave as expected?
silly_func(1)
```

```
## [1] 2
```

```
silly_func(2)
```

```
## [1] 3
```

```
# now let's loop through silly_func() 10 times:
n_times <- 10
result <- rep(NA, n_times)

for(i in 1:n_times){
 result[i] <- silly_func(i)
}

# print the output
print(result)
```

```
##  [1]  2  3  4  5  6  7  8  9 10 11
```

Now let's do this exact same thing again using a simple `map()` function from {purrr}. Why we do this will make more sense in a bit.

```
# create an input vector to iterate across
which_i <- 1:n_times

# iterate across that vector using map(), store as `result2`
result2 <-
  which_i %>%
  map(silly_func)

#view the output
print(unlist(result2))
```

```
##  [1]  2  3  4  5  6  7  8  9 10 11
```

A few things are important here. The first is that `map()` takes in a single argument (in our case `which_i`), and iterates `silly_func()` for each value in the vector `which_i`. It then returns a *list* corresponding to the outputs for each iteration. This is why I used `unlist()` to print `result2`. With the exception of this list output, we can see that the for-loop above and `map()` produce the same output.

The `map()` function is one of the simplest {purrr} functions, but there is a whole family of useful functions in this package. Some of these functions return lists, others return data frames, and some are even able to pass multiple values to functions that require more than a single input. See this link and poke around to explore some of the different options available:

https://purrr.tidyverse.org/

The functions in {purrr} that allow users to pass more than one argument to a function are preceded by the letter 'p'. This is handy because the `get_fit()` function we wrote above has many function arguments. Two such functions in {purrr} are called `pmap()` and `pwalk()`. Here's how to use them:

```
# create another silly function that requires two inputs and outputs their sum:
silly_func_2 <- function(x,y){
  return(x + y)
}

# create an input vector for each x, y
input_df <- tibble(x = 1:5, y = 1:5) # tibble more or less the same as data.frame
str(input_df)
```

```
## tibble [5 x 2] (S3: tbl_df/tbl/data.frame)
##  $ x: int [1:5] 1 2 3 4 5
##  $ y: int [1:5] 1 2 3 4 5
```

```
# iterate across tibble using pmap()
input_df %>%
  pmap(silly_func_2)
```

```
## [[1]]
## [1] 2
```

```
##
## [[2]]
## [1] 4
##
## [[3]]
## [1] 6
##
## [[4]]
## [1] 8
##
## [[5]]
## [1] 10
```

```
# iterate across tibble using pwalk()
input_df %>%
  pwalk(silly_func_2)

# note: no output
```

Thus, the 'walk' bit just means that this function is the silent analog of 'map'. Phrased differently, `pwalk()` is running exactly the same thing as `pmap()`, but it isn't storing the results anywhere or printing them out to the console. Why might this be useful?

Okay, but why all this {purrr} stuff?

So far nothing that we have done is particularly necessary or beneficial; however, when we have complex functions and lots of things to iterate across {purrr} becomes very useful.

First, it is often (slightly) faster than using for loops in R:

```
# increase amount of things to iterate across to show timing difference:
input_df <- tibble(x = 1:500, y = 1:500)

# loop through silly_func_2 the old way and time it:
system.time(
  for (i in 1:nrow(input_df)) {
    for(j in 1:nrow(input_df)){
    silly_func_2(i,j)
    }
  }
)
```

```
##    user  system elapsed
##    0.21    0.00    0.21
```

```
# compare with purrr
system.time(
    input_df %>%
      pmap(silly_func_2)
)
```

```
##    user  system elapsed
##       0       0       0
```

So that's pretty darn cool. Speedy iteration. Why is one of these faster than the other?

The second reason you might want to code things using this functional programming logic is because there is a second package, called {furrr}, that extends most of the {purrr} commands into a 'future' supported backend.

This means any of the {furrr} commands can be used in conjunction with the {future} package to run your iteration computations in parallel in a really easy way. This can be done using the following code, which first runs a version using base R and then runs a version using {furrr} functionals and compares the run time for both methods:

```
# let's make the number of iterations even bigger!
input_df <- tibble(x = 1:50000, y = 1:50000)

# loop through silly_func_2 the old way and time it:
system.time(
  for (i in 1:nrow(input_df)) {
    silly_func_2(x = input_df[i, 1], y = input_df[i, 2])
  }
)
```

```
##    user  system elapsed
##   16.50    0.02   16.56
```

```
# set up a parallel processing plan using future and plan
future::plan(multisession)

# compare with furrr--note the future_pmap vs. pmap from previous code chunk
system.time(
    input_df %>%
      future_pmap(silly_func_2)
)
```

```
##    user  system elapsed
##    0.12    0.00    1.11
```

Okay, so now we are just playing dirty. Using {furrr} we just ran the same calculation on a laptop in a fraction of the time it took to do that same calculation using a base R loop. Obviously this isn't entirely an apples to apples comparison, but the point stands that this is fast and powerful way to code. How is this working?

How does all that relate to using BERTA and `get_fit()`

Rather than looping through silly functions, can we use this functional programming knowledge to do something useful like assess lakes across a landscape?

Here's how we can use all of the tricks from above to greatly speed up our computations for the BERTA assessment models. First we declare a few variables for the Stan model, and then a `to_fit` dataframe to iterate across:

```
# declare some indeces for stan model
Ages <- 2:20
t <- 2000 # first survey year
max_a <- max(Ages)
rec_a <- min(Ages)
initial_yr <- t - max_a + rec_a - 2
add_year <- initial_yr - 1

# declare HMC run parameters
n_iter = 2000
n_chains = 4
n_warmup = n_iter/2

# which lakes were the contract lakes?
contract_lakes <- c("lac ste. anne", "baptiste lake",
                    "pigeon lake", "calling lake",
                    "moose lake", "lake newell"
)

# create a tibble to iterate through, where columns correspond
# to function arguments in get_fit()
to_fit <- tibble(which_lake = contract_lakes)
to_fit$n_iter <- n_iter
to_fit$n_chains <- n_chains
to_fit$n_warmup <- n_warmup
to_fit$rec_ctl <- "ricker"
to_fit$cr_prior <- 6

to_fit2 <- to_fit
to_fit2$rec_ctl <- "bev-holt"
to_fit <- rbind(to_fit, to_fit2)

to_fit3 <- to_fit
to_fit3$cr_prior <- 12
to_fit <- rbind(to_fit, to_fit3)

str(to_fit)
```

```
## tibble [24 x 6] (S3: tbl_df/tbl/data.frame)
##  $ which_lake: chr [1:24] "lac ste. anne" "baptiste lake" "pigeon lake" "calling lake" ...
##  $ n_iter    : num [1:24] 2000 2000 2000 2000 2000 2000 2000 2000 2000 2000 ...
##  $ n_chains  : num [1:24] 4 4 4 4 4 4 4 4 4 4 ...
##  $ n_warmup  : num [1:24] 1000 1000 1000 1000 1000 1000 1000 1000 1000 1000 ...
##  $ rec_ctl   : chr [1:24] "ricker" "ricker" "ricker" "ricker" ...
##  $ cr_prior  : num [1:24] 6 6 6 6 6 6 6 6 6 6 ...
```

Note that the columns in `to_fit` *must* match the function arguments in `get_fit()` *exactly*.

Once we have this set up, we call `plan()` to tell R we want to conduct our analysis on multiple cores, and then use `future_pwalk()` to pass the `to_fit` dataframe through the `get_fit()` function:

```
# set up parallel processing plan
future::plan(multisession)
```

```
# Run models and save fits -- 5.5 hours on my laptop
system.time({
  future_pwalk(to_fit, get_fit,
               .options = furrr_options(seed = TRUE)
  )
})
```

Boom, we just ran 6 lakes x 2 stock-recruitment relationships x 2 informative priors for compensation ratio = 24 Bayesian models in only a few lines of code. We also saved each of these runs in the fits folder.

On my laptop this takes 5.5 hours using the functionals and {future}, and around 8.7 hours if you don't use these tricks.

How could you add another lake to the analysis? Or an entirely different set of lakes?

Launching `shinystan`'s diagnostic browser

`shinystan` is a diagnostic and interactive browser for Stan models:

```
# pick a lake in the fits folder to examine
which_file <- "fits/lac_ste._anne_ricker_cr_12.rds"

# load the .rds file
fit <- readRDS(which_file)

# launch the shinystan browser
shinystan::launch_shinystan(fit)
```

After running these lines, a browser opens and you can explore the performance of chains and a variety of other Bayesian MCMC model diagnostics. See also

http://mc-stan.org/shinystan/

Debugging basics using `browser()`

For all the benefits of functional programming, it is certainly harder to debug when errors arise. This means you should have a working knowledge of `browser()` and a few other Rstudio tricks. **Discuss this in person**.

Resources:

https://purrr.tidyverse.org/

https://homerhanumat.github.io/r-notes/purrr-higher-order-functions-for-iteration.html

http://mc-stan.org/shinystan/

https://adv-r.hadley.nz/debugging.html

https://whattheyforgot.org/debugging-r-code.html