Replication document for 'Propagating chronological uncertainty with Bayesian regression models'

W. Christopher Carleton[*] Dave Campbell[†]

Libraries

```
library(ggplot2)
library(ggpubr)
library(IntCal)
library(coda)
```

Run this script?

The analyses conducted using the code below can take a long time to run. On a desktop with 32 cores and 32G memory, the whole script required several hours to complete. So, when originally prepared, this code was run in chunks, outputs were saved, and then reloaded into the R workspace as needed for plotting results. This made it possible to disentangle code processing from minor changes to the wording or layout of this document.

To facilitate that separation, a set of variables beginning with `run_it_*` were used as toggles and passed as an option to each R Markdown chunk below. When a `run_it_*` is `TRUE`, the code in the relevant chunk runs. When false, it is assumed that certain results (big matrices and R objects) are already in the R workspace and or can be loaded by the relevant code chunk. The toggles include code chunks used for plotting and each MCMC analysis has been separated. On first running this replication script, you will need to set all of these toggles to `TRUE` in order to fully replicate the analysis described in the paper associated with this document.

```
run_it_simdata_pre = F
run_it_simdata_chronup = F
run_it_mcmc = c(F,  #J=5
                F,  #J=10
                F,  #J=50
                F,  #J=100
                F)  #chronup
run_it_plot = T
run_it_simdata_big = F
run_it_mcmc_big = c(F, #J=50
                    F) #chronup
run_it_plot_big = T
```

[*]Extreme Events Research Group, Max Planck Institutes for Chemical Ecology, Science of Human History, and Biogeochemistry, Jena Germany

[†]School of Mathematics and Statistics, Carleton University, Ottawa Canada

Directory structure

R markdown typically runs code chunks in the directory containing the relevant .Rmd file (this file). But, it is common practice to separate data and source files into different subdirectories within a project folder. This .Rmd file assumes that the chunks below are being run in the `Src` folder within a parent project folder. It also assumes there is another subdirectory in the parent project folder called `Data`. All data files and output will be saved to the `Data` folder and it is assumed that any data loaded into the script is in that folder as well. To change this default behaviour, alter the path below as needed (include trailing slash).

```
# relative to the directory containing this .Rmd file
datapath <- "../Data/"
```

Simulating Count Data

True Process

As described in the associated paper, the simulation process is based on a Poisson distribution with a mean defined by a regression function with one covariate. Thus, the count y at time t is a random variable with the following conditional distribution,

$$y_t|\lambda_t \sim Pois(\lambda_t), \tag{1}$$
$$\lambda_t = e^{\beta X_t}. \tag{2}$$

To produce samples from this process while simulating the effect of chronological uncertainty, we need to set several parameters. These include the number of desired intervals in the sequence (tau), the total number of events in the sequence (n), a regression coefficient (β), covariate sequence (x), and start and end times for the count sequence chronology. For the sake of simplicity, we will assume that the time dimension is measured using a Before Present (BP) system and that the intervals correspond to years. As a result, the passage of time in the analysis that follows corresponds with decreasing age (a given number of years before the present).

```
nevents <- 1000
tau <- 1000
beta <- 0.004
x <- 0:(tau - 1)
start <- 5000
end <- (start - tau) + 1
```

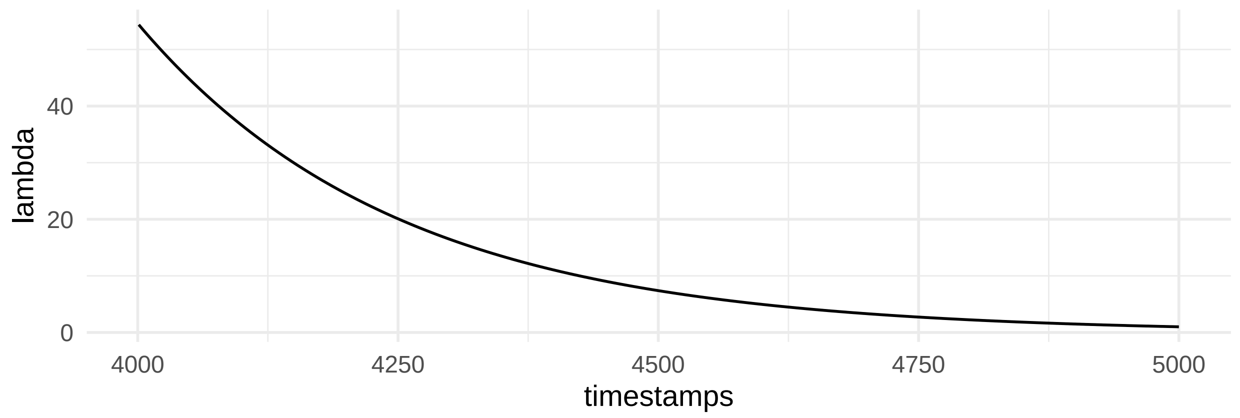
With these parameters set, we can calculate the conditional mean for the Poisson process. We will then randomly sample n ages from $[start, end]$ where the probability of sampling any given age in that range corresponds to the conditional mean function. This can be accomplished by passing the mean function evaluated at a discrete evenly-spaced set of times to R's built-in `sample()` function. Then, we can use the built-in `graphics::hist()` function to bin the sampled ages into a count sequence with the help of the `breaks` argument to define the temporal bin edges.

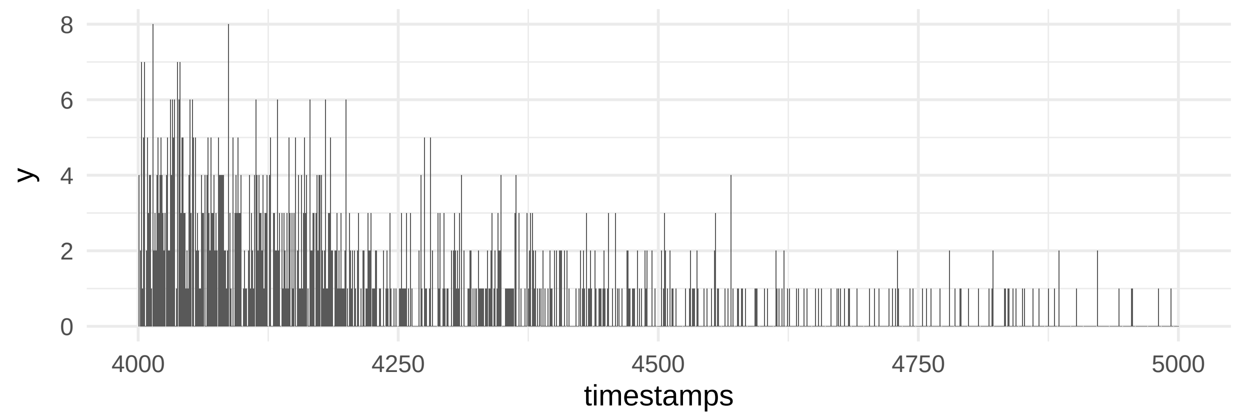
```
lambda <- exp(beta * x)
pois_process <- data.frame(lambda = lambda)
ages <- sample(x = start:end,
             size = nevents,
             prob = lambda,
             replace = T)
y <- graphics::hist(ages,
```

```r
                    breaks = seq(start, end - 1, -1),
                    include.lowest = FALSE,
                    plot = FALSE)$counts
simdata <- data.frame(timestamps = start:end,
                    lambda = lambda,
                    y = rev(y),
                    x = x)
save(simdata,
    file = paste(datapath,
                "simdata.RData",
                sep = ""))
```

Just to be certain that the sample looks sensible, plot the sample along with the underlying process (conditional Poisson mean).

```r
load(paste(datapath,
            "simdata.RData",
            sep = ""))
p1 <- ggplot(simdata) +
            geom_path(aes(y = lambda, x = timestamps)) +
            theme_minimal()
p2 <- ggplot(simdata) +
            geom_col(aes(y = y, x = timestamps)) +
            theme_minimal()
ggarrange(p1,
        p2,
        ncol = 1,
        nrow = 2,
        align = "v")
```





We can also quickly check to see whether the sample size is sufficient to estimate β by running a simple Poisson regression with R's `glm()` function.

```
glm_check <- glm(y~x,
                 data = simdata,
                 family = "poisson")
summary(glm_check)
```

```
##
## Call:
## glm(formula = y ~ x, family = "poisson", data = simdata)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.8011  -0.8247  -0.4829   0.4459   3.2839
##
## Coefficients:
##               Estimate Std. Error z value Pr(>|z|)
## (Intercept) -2.5528366  0.1195663  -21.35   <2e-16 ***
## x            0.0039473  0.0001506   26.22   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for poisson family taken to be 1)
##
##     Null deviance: 1930.71  on 999  degrees of freedom
## Residual deviance:  990.22  on 998  degrees of freedom
## AIC: 2163.3
##
## Number of Fisher Scoring iterations: 5
```

Introducing Chronological Uncertainty

Adding chronological uncertainty to the simulated data could be done in many ways. But, since radiocarbon dating is probably the most commonly used chronometric method in the Palaeo Sciences (and very popular recently with respect to event count sequences), we will simulate radiocarbon uncertainties.

To begin, pass the sampled ages from above to the IntCal packages's `calBP.14C` function. The function returns plausible C14 determinations (ages in uncalibrated radiocarbon years before present) given a set of calendar ages (the ages we sampled). These simulated C14 ages must then be calibrated and can be stored conveniently in a chronological uncertainty matrix called `chronun_matrix`. In order to align the calibrated date densities (discrete estimates provided by calibration software) for storage in the matrix, we also need to determine the maximum span of time (`sample_time_range`) covered by all the simulated sample dates. Lastly, the range of times corresponding to that enclosing interval can be stored in a vector separate from the `chronun_matrix` for use later on. The following code accomplishes all of these steps.

```r
simc14 <- t(sapply(ages, IntCal::calBP.14C))

c14post <- lapply(1:nevents,
                function(x, dates){
                    IntCal::caldist(age = dates[x, 1],
                                    error = dates[x, 2],
                                    BCAD = F)},
                    dates = simc14)
sample_time_range <- range(
                        unlist(
                            lapply(c14post,
                                function(x)range(x[, 1])
                            )
                        )
                    )
expanded_times <- sample_time_range[2]:sample_time_range[1]
c14post_expanded <- lapply(c14post,
                            function(caldate, times){
                                    approx(caldate,
                                        xout = times)$y
                                    }, times = expanded_times)
chronun_matrix <- do.call(cbind,c14post_expanded)
chronun_matrix[which(is.na(chronun_matrix))] <- 0
```

Next, the `chronun_matrix` is used to create an `event_count_ensemble`. The ensemble is just a sample of probable event counts given the radiocarbon dating uncertainty we just simulated. Each member of the ensemble (one probable count sequence) will be stored as a column in a matrix. Since we want to create a large ensemble, two small functions will be required. One function will sample the `chronun_matrix` for a probable set of event ages and the other will count those events, binning them into a probable count sequence with the given resolution (defined by temporal bin edges passed as the `breaks` argument to the `graphics::hist()` function used earlier).

```r
# first function - sample event times
sample_event_times <- function(x = NULL,
                                chronun_matrix,
                                times){
```

```
    times_sample <- apply(chronun_matrix,
                          2,
                          function(j)sample(times, size=1, prob=j))
    return(times_sample)
}
# second function - count events
count_events <- function(x,
                        breaks){
    counts <- graphics::hist(x,
                          breaks = breaks,
                          include.lowest = FALSE,
                          plot = FALSE)$counts
    return(rev(counts))
}
# create breaks vector
start <- expanded_times[1]
end <- expanded_times[length(expanded_times)] - 1
breaks <- seq(from = start, to = end, by = -1)
```

With the helpful functions in hand, sample 1000 probable event ages:

```
event_time_sample <- sapply(1:1000,
                        sample_event_times,
                        chronun_matrix = chronun_matrix,
                        times = expanded_times)
```

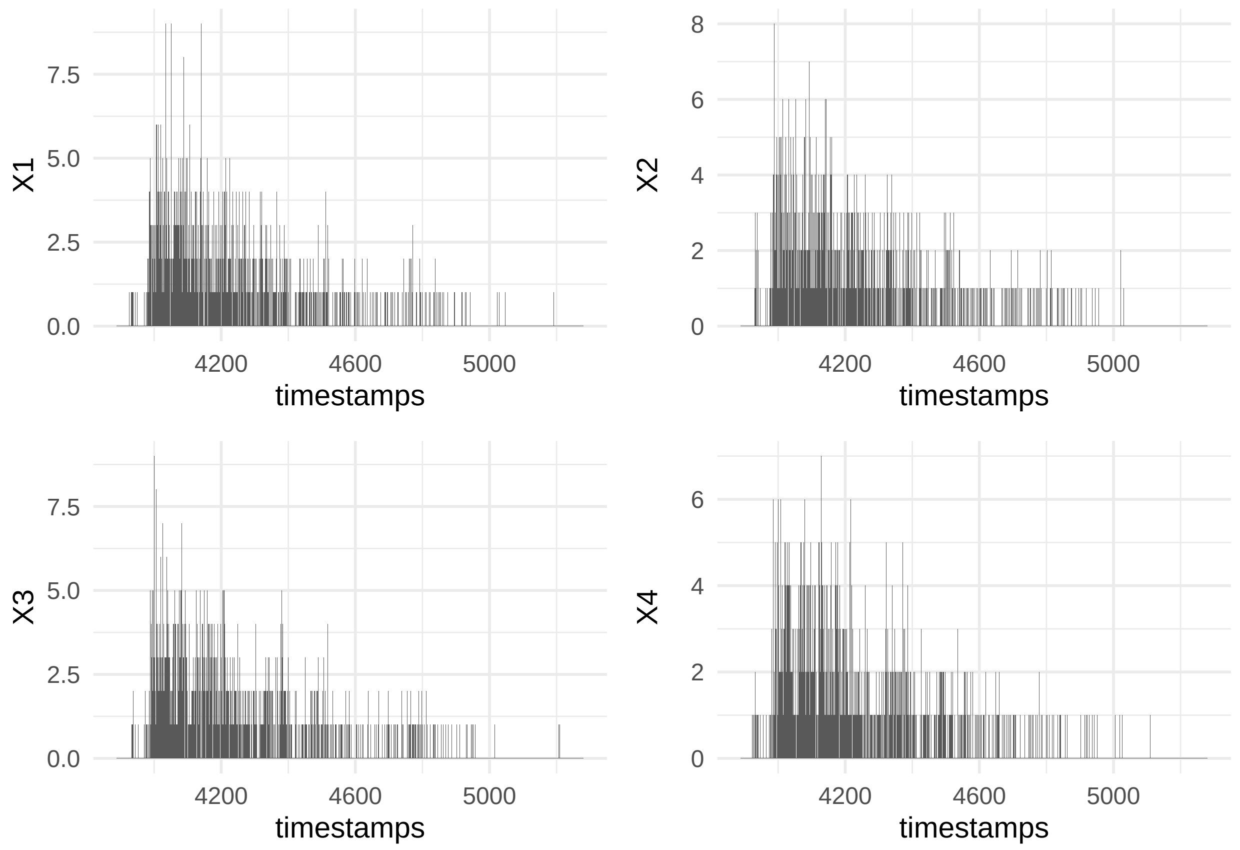
And, then, compile the count sequences:

```
event_count_ensemble <- apply(event_time_sample,
                          2,
                          count_events,
                          breaks = breaks)
save(event_count_ensemble,
    file = paste(datapath,
                "count_ensemble_1.RData",
                sep = ""))

save(expanded_times,
    file = paste(datapath,
                "count_ensemble_1_expanded_times.RData",
                sep = ""))
```

As before, it's a good idea to plot some of the sequences to make sure they are as they should be (increasing with the passage of time in a roughly exponential-looking way). To help keep variable names to a reasonable length, we can refer to any event count ensemble as an `ece` and if the events in question are radiocarbon dated we can use `rece` (radiocarbon-dated event count ensemble).

```
load(paste(datapath,
            "count_ensemble_1.RData",
            sep = ""))

load(paste(datapath,
            "count_ensemble_1_expanded_times.RData",
            sep = ""))
```

```r
rece <- data.frame(timestamps = expanded_times,
                    event_count_ensemble[,1:4])
p1 <-ggplot(rece) +
            geom_col(aes(y = X1, x = timestamps)) +
            theme_minimal()
p2 <- ggplot(rece) +
            geom_col(aes(y = X2, x = timestamps)) +
            theme_minimal()
p3 <- ggplot(rece) +
            geom_col(aes(y = X3, x = timestamps)) +
            theme_minimal()
p4 <- ggplot(rece) +
            geom_col(aes(y = X4, x = timestamps)) +
            theme_minimal()
ggarrange(p1,
        p2,
        p3,
        p4,
        ncol = 2,
        nrow = 2,
        align = "v")
```



In the figure above, each plot shows one probable event count sequence given the uncertainty in the simulated radiocarbon dates for the corresponding events. Time in the plots proceeds right to left on the BP timescale (easy enough to change by reversing the axes if desired). Individually, the sequences include no information about chronological uncertainty. In isolation a given ensemble member is effectively assuming a certain set of ages for the events (sampled at random from the corresponding calibrated radiocarbon-date distributions). Collectively, though, the ensemble contains the information needed to propagate the chronological uncertainty into any further analyses.

Simulating Count Data with 'chronup'

All of the steps taken so far to simulate data can be repeated more easily with an R package called chronup. The package is currently under development and in a pre-release state (not yet on CRAN). But, the code is open and available on GitHub. The package `R::devtools` can be used to download and install the latest version of `chronup`. To facilitate large sample of events and very large ensembles, the package can make use of parallelization (built-in R functions) and big matrix objects (from `R::bigmemory`). So, it is assumed from here on that this script is being run in an environment that supports both.

```
library(devtools)
install_github("wccarleton/chronup")
```

As before, we can define some key parameters to begin. In addition to the same ones defined above, a new parameter called `nsamples` will determine how many probable event count sequences to create.

```
nevents <- 1000
nsamples <- 200000
tau <- 1000
beta <- 0.004
x <- 0:(tau - 1)
lambda <- exp(beta * x)
start <- 5000
end <- (start - tau) + 1
times <- start:end
```

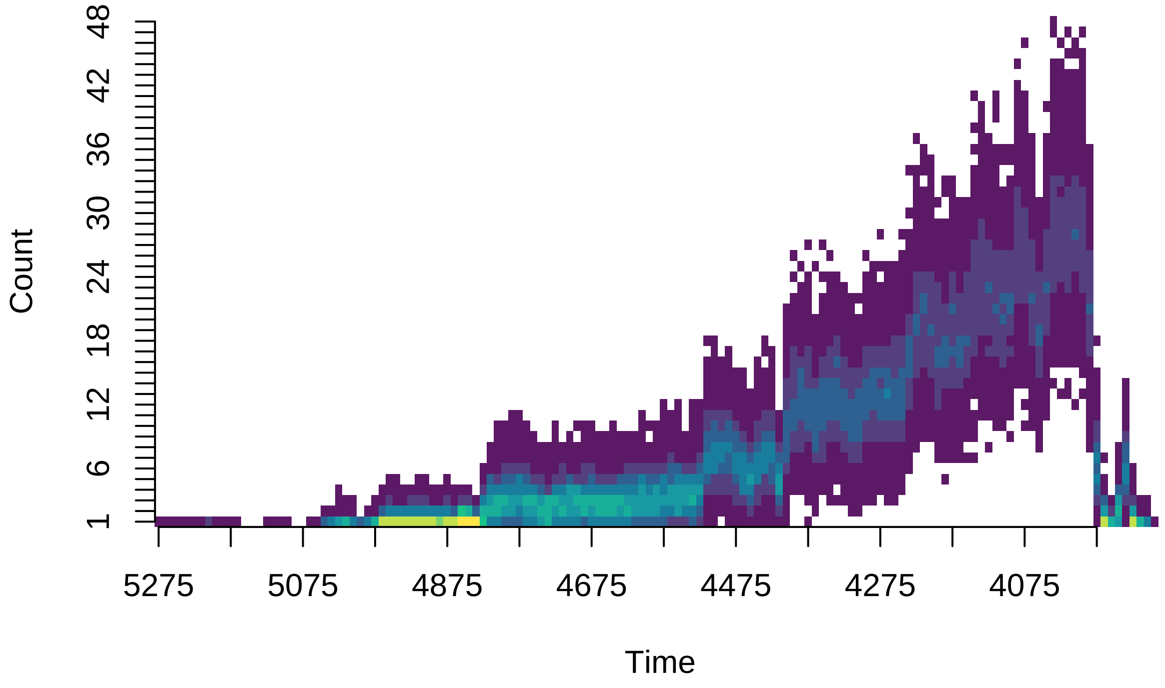
With these parameters set, one function call to `chronup::simulate_event_counts` will produce 200,000 probable event count sequences assuming radiocarbon-dated uncertainties:

```
simdata2 <- chronup::simulate_event_counts(process = lambda,
                                            times = times,
                                            nevents = nevents,
                                            nsamples = nsamples,
                                            binning_resolution = -10,
                                            bigmatrix = datapath)
save(simdata2,
    file = paste(datapath,
                "simdata2.RData",
                sep = ""))
```

The `chronup` package also includes a convenient way to plot a `rece` or count ensemble. To speed up the plotting process, we will plot only a subset of the probable sequences. Since the ensemble has been stored in a big memory matrix, we will first have to attach that matrix. The path to the matrix is stored as an element in the `simdata2` object (`simdata$count_ensemble`).

```
load(paste(datapath,
        "simdata2.RData",
        sep = ""))

rece <- bigmemory::attach.big.matrix(simdata2$count_ensemble)
rece_sub <- rece[,1:1000]
```

```
chronup::plot_count_ensemble(count_ensemble = rece_sub,
                             times = simdata2$new_timestamps,
                             axis_x_res = 10)
```



```
## NULL
```

Chronological uncertainty is represented in the plot above by a heat map. The colours indicate the relative probability of count-time pairings based on the sample represented by the count ensemble. Warmer colours indicate higher relative probabilities whereas cooler colours indicate lower relative probabilities. So, the most probable true event count sequence will be located somewhere in the hottest, brightest region of the plot. By default, the plotting function presents the passage of time moving left to right. Remember that this is effectively a density estimate not a model, and it needs to be treated as such.

Introducing Chronological Uncertainty in the Covariate

In almost all practical Palaeo Science analyses both the count data and any potential explanatory covariates will have chronological uncertainties. The covariate (independent variable) used in the simulation analyses described here is simply a linear function of time—essentially just the passage of time itself. The values of the covariate are, therefore, just a series of increasing integers, each corresponding to one of the intervals into which the events fall. Still, it would be useful to include some kind of covariate uncertainty since it would be present in real Palaeo Science research.

With that in mind, we can create another ensemble matrix that reflects covariate uncertainty. Each column in the matrix will refer to one probable covariate sequence. These could be, for instance, probable palaeoclimate reconstructions involving isotopes measured in a layer-counted depositional sequence, like a speleothem or a lake core. Each sequence, then, would be one probable sequence of isotope measurements given the chronological and depth measurement uncertainties in a relevant age-depth model. Since the covariate used in the present analysis is a simple series of integers, some uncertainty can be introduced by adding a random deviation to each value using R's built-in probability distributions.

It should be noted that since the span of time occupied by the RECE samples is longer than the span of the original process. So, we are confronted with two possible solutions. One is to extend the simulated covariate series forward and backward (start the sequence at 0 where that corresponds to the first element of any given RECE sample). The other is to crop both the dependent and independent variables to the span of the true process defined by the `start` and `end` parameters. For this simulation exercise the simplest solution is the second one, so that is what we will do.

To begin, establish a bigmatrix object to hold the covariate samples. The matrix will have the same number of rows as the RECE matrix and twice the number of columns. We are using twice the number of columns in order to include an intercept in the model, which will simply be a column of `1`'s. As the MCMC proceeds, columns from the independent variable matrix (`X`) will be selected two at a time to account for the intercept.

```
X <- bigmemory::filebacked.big.matrix(
                    nrow = dim(rece)[1],
                    ncol = dim(rece)[2] * 2,
                    backingpath = datapath,
                    backingfile = "X_mat",
                    descriptorfile = "X_mat_desc")

X_loc <- bigmemory::describe(X)
```

For each observation in the covariate sequence, we add a random deviation and then store the result in the `X` matrix. We can make use of parallelization and a progress bar to speed up this process and monitor it. Note that in order to preserve the temporal scale and obtain estimates of `beta` at the correct scale, we have to multiply the time indeces by the resolution of the temporal bins, which means multiplying in this case by 10. Likewise, it is important to note that the uncertainty we wish to introduce into the covariate would need to be scaled accordingly to ensure that a sensible level of uncertainty is being included. For simplicity, we will use a standard deviation 2, which corresponds to 20 years at the original temporal resolution of the simulated data. In a real-world analysis, forgetting to account for the temporal width of the bins can lead to scaled parameter estimates. The estimates would be a multiple of the temporal resolution in the case of a regression coefficient referring only to the passage of time. Correcting the scale after obtaining parameter estimates would involve dividing those estimates by the temporal resolution of the bins, but here we can avoid the problem by scaling appropriately at the outset.

```
library(pbapply)
library(parallel)

cl <- makeCluster(detectCores() - 1)

x_error = 2

pbsapply(cl = cl,
        X = 1:dim(rece)[2],
        FUN = function(j,
                    x_error,
                    n,
                    nX,
                    X_loc){
                library(bigmemory)
                X <- attach.big.matrix(X_loc)
                x_indeces <- 1:nX + (nX * (j - 1))
                X[, x_indeces[1]] <- 1
                xx <- rnorm(n = n,
                            mean = 0:(n - 1),
                            sd = x_error)
                X[,x_indeces[2]] <- xx * 10
                return()},
        x_error = x_error,
        n = dim(rece)[1],
        nX = 2,
        X_loc = X_loc)

stopCluster(cl)
```

Accounting for chronological uncertainty in regression

Recently, attempts have been made to handle chronological uncertainty in Palaeo Science regression analyses. As described in the paper associated with this replication document, these recent attempts have involved a hierarchical arrangement of parameters in count-based regression models. In this section, we run one of these analyses on the data simulated above and then explore an alternate approach using the same data.

Hierarchical arrangement

For the hierarchical model, we use a Bayesian R package called Nimble. Nimble is a system for creating Bayesian models and estimating model parameters with a flexible, sophisticated MCMC engine. The core of Nimble model specification is based on an older language for writing Bayesian models called "BUGS". The first step is to write the model using a Nimble R function and Nimble's slightly adapted BUGS language. Then, the model definition along with other important variables are used to create a Nimble model object, which is subsequently passed to Nimble's MCMC engine (after MCMC settings are established). Nimble allows for all of the relevant R code and objects to be compiled to C++, which massively reduces computation times.

The following code is used to create a hierarchical model in Nimble and run an MCMC given the data simulated above,

```
library(nimble)

load(paste(datapath,
        "simdata2.RData",
        sep = ""))

rece <- bigmemory::attach.big.matrix(simdata2$count_ensemble)
X <- bigmemory::attach.big.matrix(paste(datapath,
                                "X_mat_desc",
                                sep = ""))

nbCode <- nimbleCode({
   ###top-level regression
   B0 ~ dnorm(0, 100)
   B1 ~ dnorm(0, 100)
   sigB0 ~ dunif(1e-10, 100)
   sigB1 ~ dunif(1e-10, 100)
   for (j in 1:J) {
      ###low-level regression
      b0[j] ~ dnorm(mean = B0, sd = sigB0)
      b1[j] ~ dnorm(mean = B1, sd = sigB1)
      for (n in 1:N){
        p[n, j] ~ dunif(1e-10, 1)
        r[n, j] <- exp(b0[j] + X[n, j] * b1[j])
        Y[n, j] ~ dnegbin(size = r[n, j], prob = p[n, j])
      }
   }
})
```

With the Nimble model defined, sample the data matrices, setup the MCMC, and run it. First, we will create a vector (`span_index`) for subsetting the data to the interval of time covered by the simlations process (i.e., 5000:4001 BP). We will also use Nimble's slice samplers for the sub-regressions (probable regressions at the lower level of the hierarchy) to improve the sampling of the model posteriors and then run the MCMC for at least `niter = 1000000` iterations (thinned to retain only every 10th sample). Some of these MCMC parameters can be set up globally, while others have to be established again for each new MCMC run because of the way Nimble compiles code to C++, as described earlier. The global parameters are,

```
span_index <- which(simdata2$new_timestamps <= start &
                    simdata2$new_timestamps >= end)
n <- length(span_index)

#number of MCMC iterations
niter <- 1000000

#thinning to save space
thinning <- 10

#set seed for replicability
set.seed(1)
```

To explore the impact of the size of the sample of probable sequences on the posterior density estimates, we will run 4 separate MCMC simulations: one simulation for each of $J \in [5, 10, 50, 100]$. Only the code for the first of these—`J = 5`—is printed in the PDF/md document produced by this script, but the code for each is in the script below.

Importantly, we need to select the J draws from the RECE and the corresponding J draws from the matrix containing the covariate (X) values with uncertainty added in. The J RECE columns will be randomly sampled without replacement (they are already a random sample of probable count sequences and we do not want to select a given sequence more than once). For the 'X' matrix, we need to select every other column, though, because that matrix contains alternating columns of 1's representing the model intercept and covariate values. The intercept is included in the Nimble model specification internally, and so needs to be excluded from the covariate matrix.

The code for the MCMC involving `J = 5` probable sequences (and matching covariate sequences) is as follows,

```
span_index <- which(simdata2$new_timestamps <= start &
                    simdata2$new_timestamps >= end)
n <- length(span_index)
J = 5

y_sample <- sample(1:dim(rece)[2],
                   size = J,
                   replace = F)
x_sample <- sample(seq(2, dim(rece)[2], 2),
```

```r
                size = J,
                replace = F)

nbData <- list(Y = rece[span_index, y_sample],
                X = X[span_index, x_sample])

nbConsts <- list(N = n,
                J = J)

nbInits <- list(B0 = 0,
                B1 = 0,
                b0 = rep(0, J),
                b1 = rep(0, J),
                sigB0 = 0.0001,
                sigB1 = 0.0001)

nbModel <- nimbleModel(code = nbCode,
                        data = nbData,
                        inits = nbInits,
                        constants = nbConsts)

#compile nimble model to C++ code-much faster runtime
C_nbModel <- compileNimble(nbModel, showCompilerOutput = FALSE)

#configure the MCMC
nbModel_conf <- configureMCMC(nbModel)

nbModel_conf$monitors <- c("B0", "B1", "sigB0", "sigB1")
nbModel_conf$addMonitors2(c("b0", "b1"))

#samplers
nbModel_conf$removeSamplers(c("B0", "B1", "sigB0", "sigB1", "b0", "b1"))
nbModel_conf$addSampler(target = c("B0", "B1", "sigB0", "sigB1"),
                        type = "AF_slice")
for(j in 1:J){
   nbModel_conf$addSampler(target = c(paste("b0[", j, "]", sep = ""),
                                      paste("b1[", j, "]", sep = "")),
                        type = "AF_slice")
}

#thinning to conserve memory when the samples are saved below
nbModel_conf$setThin(thinning)
nbModel_conf$setThin2(thinning)

#build MCMC
nbModelMCMC <- buildMCMC(nbModel_conf)

#compile MCMC to C++-much faster
C_nbModelMCMC <- compileNimble(nbModelMCMC, project = nbModel)

#save the MCMC chain (monitored variables) as a matrix
nimble_samples_5 <- runMCMC(C_nbModelMCMC, niter = niter)
```

```r
#save samples
save(nimble_samples_5,
        file = paste(datapath,
                    "mcmc_samples_nimble_5.RData",
                    sep = ""))
```

```r
span_index <- which(simdata2$new_timestamps <= start &
                    simdata2$new_timestamps >= end)
n <- length(span_index)
J = 50

y_sample <- sample(1:dim(rece)[2],
                size = J,
                replace = F)
x_sample <- sample(seq(2, dim(rece)[2], 2),
                size = J,
                replace = F)

nbData <- list(Y = rece[span_index, y_sample],
                X = X[span_index, x_sample])

nbConsts <- list(N = n,
                J = J)

nbInits <- list(B0 = 0,
                B1 = 0,
                b0 = rep(0, J),
                b1 = rep(0, J),
                sigB0 = 0.0001,
                sigB1 = 0.0001)

nbModel <- nimbleModel(code = nbCode,
                        data = nbData,
                        inits = nbInits,
                        constants = nbConsts)

#compile nimble model to C++ code-much faster runtime
C_nbModel <- compileNimble(nbModel, showCompilerOutput = FALSE)

#configure the MCMC
nbModel_conf <- configureMCMC(nbModel)

nbModel_conf$monitors <- c("B0", "B1", "sigB0", "sigB1")
nbModel_conf$addMonitors2(c("b0", "b1"))

#samplers
nbModel_conf$removeSamplers(c("B0", "B1", "sigB0", "sigB1", "b0", "b1"))
nbModel_conf$addSampler(target = c("B0", "B1", "sigB0", "sigB1"),
                        type = "AF_slice")
for(j in 1:J){
   nbModel_conf$addSampler(target = c(paste("b0[", j, "]", sep = ""),
                                    paste("b1[", j, "]", sep = "")),
                        type = "AF_slice")
}
```

```
#thinning to conserve memory when the samples are saved below
nbModel_conf$setThin(thinning)
nbModel_conf$setThin2(thinning)

#build MCMC
nbModelMCMC <- buildMCMC(nbModel_conf)

#compile MCMC to C++-much faster
C_nbModelMCMC <- compileNimble(nbModelMCMC, project = nbModel)

#save the MCMC chain (monitored variables) as a matrix
nimble_samples_50 <- runMCMC(C_nbModelMCMC, niter = niter)

#save samples
save(nimble_samples_50,
        file = paste(datapath,
                     "mcmc_samples_nimble_50.RData",
                     sep = ""))
```

Alternative with `chronup`

Next, we can use a regression function from the `chronup` package to perform another count-based regression on these simulated data while propagating the uncertainty in the data up to the posterior estimates of the model's parameters. In this case, though, the parameters are not hierarchically arranged as they were for the Nimble analysis.

```
span_index <- which(simdata2$new_timestamps <= start &
                    simdata2$new_timestamps >= end)

startvals <- c(0, 0, rep(0.1, 100))
startscales <- c(0.1, 0.0002, rep(0.5, 100))

adapt_limit <- floor(nsamples * 0.25)

adapt_sample_y <- 1:adapt_limit
adapt_sample_x <- 1:(adapt_limit * 2)

mcmc_samples_adapt <- chronup::regress(Y = rece[span_index, adapt_sample_y],
                                        X = X[span_index, adapt_sample_x],
                                        model = "nb",
                                        startvals = startvals,
                                        scales = startscales,
                                        adapt = T)

burnin <- floor(dim(mcmc_samples_adapt$samples)[1] * 0.2)
indeces <- seq(burnin, dim(mcmc_samples_adapt$samples)[1], 1)
adapt_startvals <- colMeans(mcmc_samples_adapt$samples[indeces, ])

burnin <- floor(dim(mcmc_samples_adapt$scales)[1] * 0.2)
indeces <- seq(burnin, dim(mcmc_samples_adapt$scales)[1], 1)
adapt_startscales <- colMeans(mcmc_samples_adapt$scales[indeces, ])
```

```r
chronup_samples <- chronup::regress(Y = rece[span_index, ],
                                X = X[span_index, ],
                                model = "nb",
                                startvals = adapt_startvals,
                                scales = adapt_startscales,
                                adapt = F)
save(chronup_samples,
        file = paste(datapath,
                    "mcmc_samples_chronup.RData",
                    sep = ""))
```

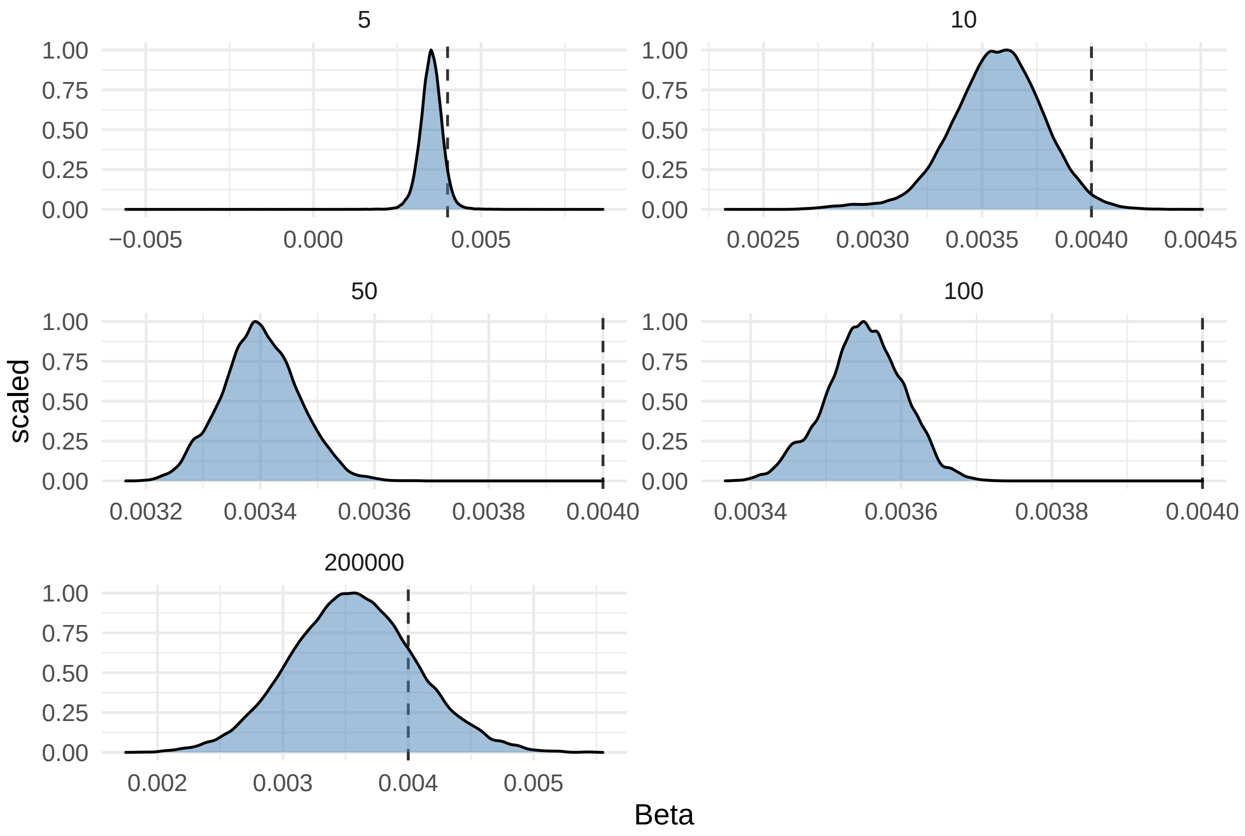
Comparing results

Finally, we can compare the results of the two approaches. The key parameter of interest is `beta`. It is the main regression coefficient in the models and represents the rate of the conditional counting process. In a practical case, it would indicate the growth rate (or, conversely, decay rate) of whatever process produced concentrations of radiocarbon samples in the sediments excavated during a Palaeo Science research project. If covariates other than time are involved, the regression coefficients would reflect the impact of those covariates on that growth/decay rate.

To see clearly how the two different approaches perform, we will compare density estimates based on the posterior samples from the various MCMC simulations. Using the following code, a side-by-side comparison can be plotted with the help of `ggplot2`. First, though, we will combine the samples into a single long dataframe. We will also discard the first 10% of samples from each as burn-in and scale the density estimates to a maximum of one to make visual comparisons easier. Initially, plot the densities without any x-axis constraints, but then produce the same plot again with the x-axis limits set to focus on the area around the target parameter estimate to facilitate comparison.

```r
load(paste(datapath,
            "mcmc_samples_nimble_5.RData",
            sep = ""))
load(paste(datapath,
            "mcmc_samples_nimble_10.RData",
            sep = ""))
load(paste(datapath,
            "mcmc_samples_nimble_50.RData",
            sep = ""))
load(paste(datapath,
            "mcmc_samples_nimble_100.RData",
            sep = ""))
load(paste(datapath,
            "mcmc_samples_chronup.RData",
            sep = ""))

n_hier <- dim(nimble_samples_5$samples)[1]
n_alt <- dim(chronup_samples)[1]

burnin_hier <- floor(n_hier * 0.1)
burnin_alt <- floor(n_alt * 0.1)

iteration_hier <- (burnin_hier + 1):n_hier
iteration_hier_all <- rep(iteration_hier, 4)
iteration_alt <- (burnin_alt + 1):n_alt
```

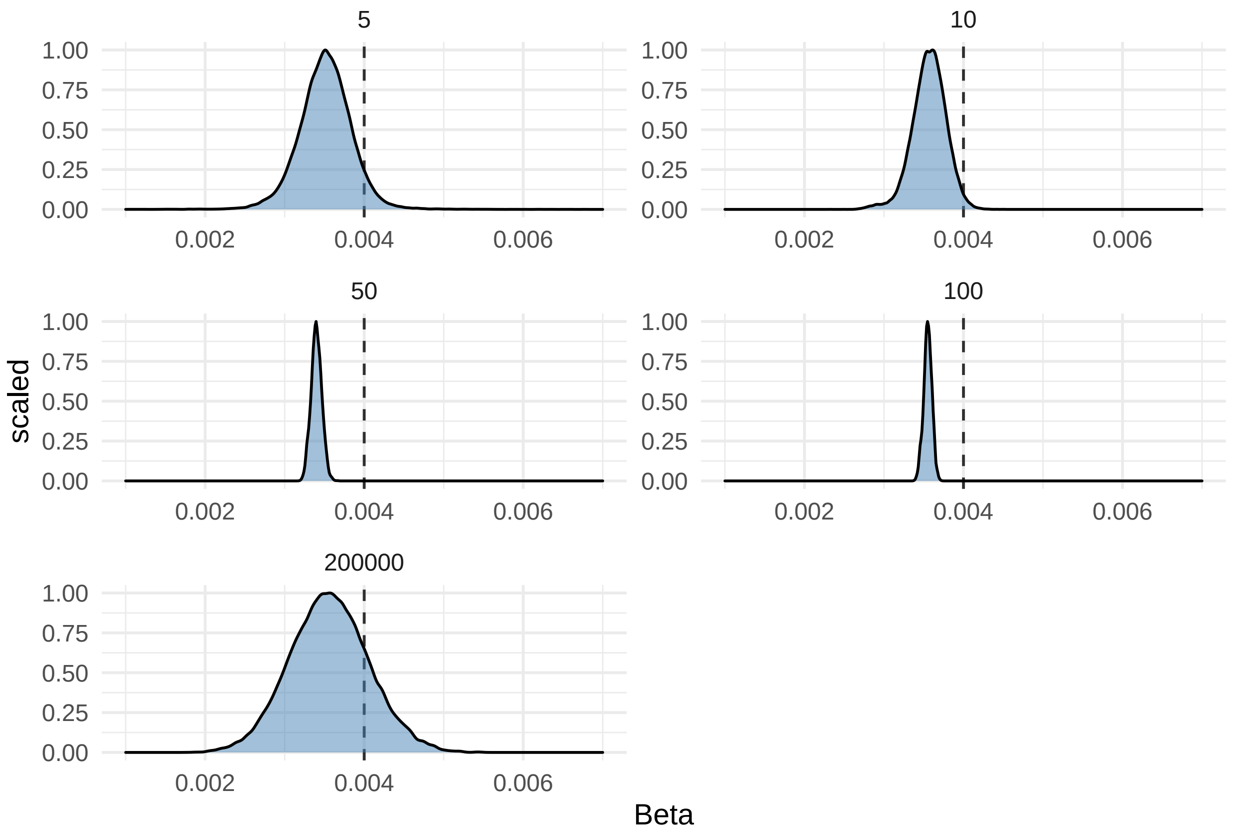
```r
n_retained_hier <- length(iteration_hier)
n_retained_alt <- length(iteration_alt)

plot_samples_hier_5 <- nimble_samples_5$samples[iteration_hier, 2]
plot_samples_hier_10 <- nimble_samples_10$samples[iteration_hier, 2]
plot_samples_hier_50 <- nimble_samples_50$samples[iteration_hier, 2]
plot_samples_hier_100 <- nimble_samples_100$samples[iteration_hier, 2]
plot_samples_hier <- c(plot_samples_hier_5,
                    plot_samples_hier_10,
                    plot_samples_hier_50,
                    plot_samples_hier_100)
plot_samples_alt <- chronup_samples[iteration_alt, 2]

model_name_hier <- rep("hier", n_retained_hier * 4)
model_name_alt <- rep("alt", n_retained_alt)

J_hier <- rep(c("5", "10", "50", "100"), each = n_retained_hier)
J_alt <- rep("200000", n_retained_alt)

J_both <- factor(c(J_hier, J_alt), levels = c("5", "10", "50", "100", "200000"))

mcmc_samples_both <- data.frame(Iteration = c(iteration_hier_all,
                                              iteration_alt),
                            Model = c(model_name_hier,
                                      model_name_alt),
                            J = J_both,
                            Beta = c(plot_samples_hier,
                                     plot_samples_alt))

ggplot(data = mcmc_samples_both) +
    geom_vline(xintercept = beta,
               linetype = 2,
               alpha = 0.8) +
    geom_density(mapping = aes(y = ..scaled.., x = Beta),
               fill = "steelblue",
               alpha = 0.5) +
    facet_wrap(~ J, ncol = 2, scales = "free") +
    theme_minimal()
```



To make comparisons easier, we can re-produce the same plot, but this time limit the x-axis to a region around the target value for β and align the plot axes:

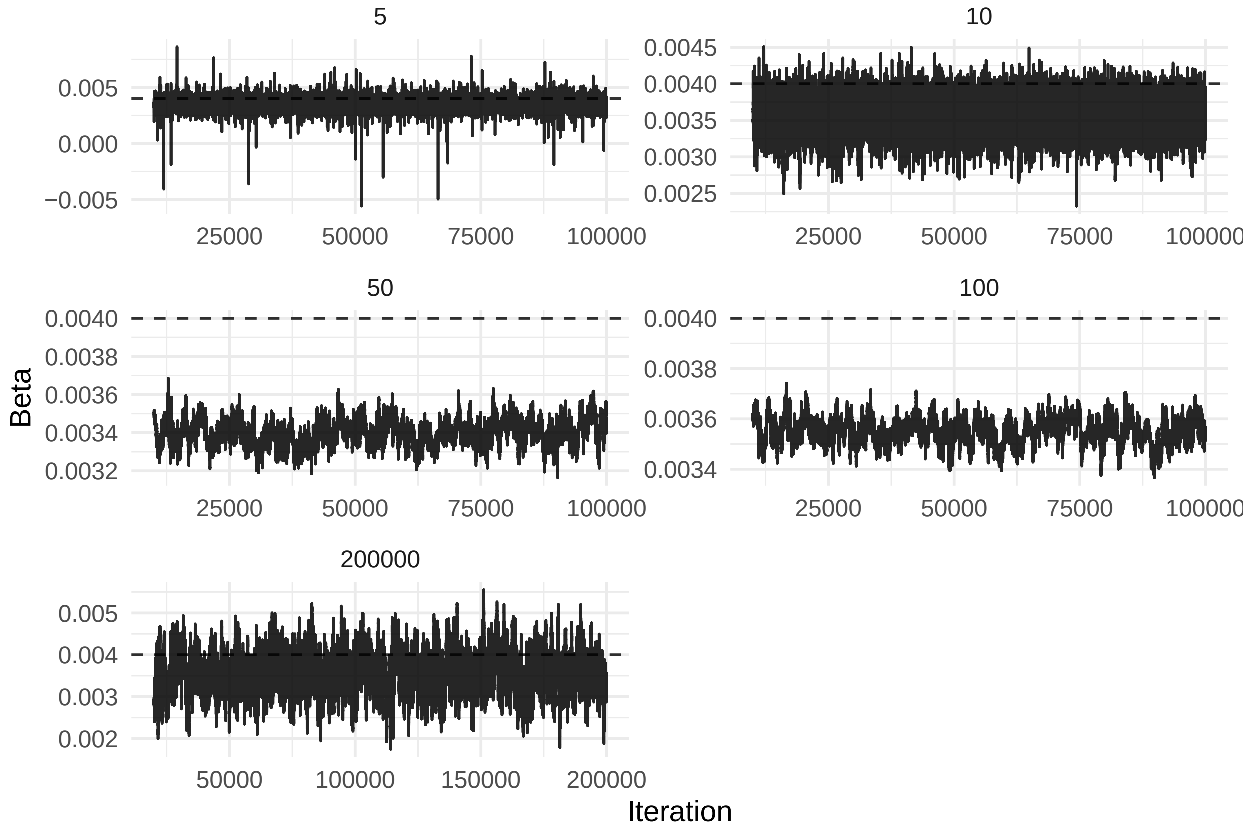
```r
ggplot(data = mcmc_samples_both) +
    geom_vline(xintercept = beta,
               linetype = 2,
               alpha = 0.8) +
    geom_density(mapping = aes(y = ..scaled.., x = Beta),
               fill = "steelblue",
               alpha = 0.5) +
    facet_wrap(~ J, ncol = 2, scales = "free") +
    scale_x_continuous(limits = c(0.001, 0.007)) +
    theme_minimal()
```

```
## Warning: Removed 31 rows containing non-finite values (stat_density).
```



Lastly, we should also inspect the mcmc trace plots for each model/estimate, and then check the Geweke statistics for the main parameters,

```r
ggplot(data = mcmc_samples_both) +
    geom_hline(yintercept = beta,
               linetype = 2,
               alpha = 0.8) +
    geom_path(mapping = aes(y = Beta, x = Iteration),
               alpha = 0.85) +
    facet_wrap(~ J, ncol = 2, scales = "free") +
    theme_minimal()
```



```r
geweke.diag(cbind(plot_samples_hier_5,
                  plot_samples_hier_10,
                  plot_samples_hier_50,
                  plot_samples_hier_100,
                  plot_samples_alt))
```

```
## Warning in cbind(plot_samples_hier_5, plot_samples_hier_10,
## plot_samples_hier_50, : number of rows of result is not a multiple of vector
## length (arg 1)
```

```
##
## Fraction in 1st window = 0.1
## Fraction in 2nd window = 0.5
##
##   plot_samples_hier_5  plot_samples_hier_10  plot_samples_hier_50
##                0.5707               -0.4789                0.5714
## plot_samples_hier_100      plot_samples_alt
##                0.8431                0.1215
```

As the plots show, both methods yielded estimates for `beta` that either include the true value (0.004, indicated by the vertical dashed line) within their 99% and/or 95% credible intervals or are at least quite close. But, the hierarchical model (`heir` in the plot legend) produced posterior density estimates with a much lower variance than the alternate method and can occasionally exclude the target value when large samples of probable sequences are involved (i.e., `J` is large).

Sample Size

A feature of hierarchical models is that they allow for borrowing of information across samples. This leads to a phenomenon sometimes called "shrinkage" whereby the variance in the posterior density for model parameters tends to be smaller than the variance of the same parameters estimated separately for each data sample. This appears to be happening with the posteriors of the hierarchical REC model estimated in Nimble. And, as the size of the sample (i.e., `J`, referring to the number of probable count sequences analyzed) increases, the variance of the posterior densities decreases, as shown in the plots above. A bias is being introduced because a single β parameter is being used to estimate the likelihoods of all `J` probable regression models simultaneously, which results in the estimates for β_j being pulled together. This convergence, in turn, leads to a lower variance in the upper level hyperparameter for β's mean. Any lingering bias in the underlying event sample, though, will cause that mean estimate to deviate from the true underlying value. Subsequently, the hyperparameter estimate for the target value will be a slightly biased, but high-precision estimate.

To see the phenomenon more clearly, we can run an additional analysis that involves a larger sample of dates. Whereas before a sample of 1000 dates was drawn from the simulated process, we can instead draw 5000 dates and re-run some of the analyses from above. The larger sample will reduce the effect of sampling bias, and we should see an improvement in the hierarchical model's estimate of the target value (β). It will, however, likely still have a bias (from sampling and perhaps the effect of chronological uncertainty on count sequences) and a lower variance than either those involving a smaller sample of probable sequences or a similar analysis conducted using the alternative sampling framework implemented with `chronup`.

First, draw more samples from the simulation process,

```r
nevents <- 5000
nsamples <- 200000
tau <- 1000
beta <- 0.004
x <- 0:(tau - 1)
lambda <- exp(beta * x)
start <- 5000
end <- (start - tau) + 1
times <- start:end
```

```r
simdata3 <- chronup::simulate_event_counts(process = lambda,
                                           times = times,
                                           nevents = nevents,
                                           nsamples = nsamples,
                                           binning_resolution = -10,
                                           bigmatrix = datapath,
                                           bigfileprefix = "count_big")
save(simdata3,
    file = paste(datapath,
                 "simdata3.RData",
                 sep = ""))
```

Next, setup and run a Nimble model as before,

```r
library(nimble)

load(paste(datapath,
        "simdata3.RData",
        sep = ""))

rece <- bigmemory::attach.big.matrix(simdata3$count_ensemble)
X <- bigmemory::attach.big.matrix(paste(datapath,
                                "X_mat_desc",
                                sep = ""))

nbCode <- nimbleCode({
   ###top-level regression
   B0 ~ dnorm(0, 100)
   B1 ~ dnorm(0, 100)
   sigB0 ~ dunif(1e-10, 100)
   sigB1 ~ dunif(1e-10, 100)
   for (j in 1:J) {
      ###low-level regression
      b0[j] ~ dnorm(mean = B0, sd = sigB0)
      b1[j] ~ dnorm(mean = B1, sd = sigB1)
      for (n in 1:N){
        p[n, j] ~ dunif(1e-10, 1)
        r[n, j] <- exp(b0[j] + X[n, j] * b1[j])
        Y[n, j] ~ dnegbin(size = r[n, j], prob = p[n, j])
      }
   }
})

span_index <- which(simdata2$new_timestamps <= start &
                    simdata2$new_timestamps >= end)
n <- length(span_index)

J = 5

y_sample <- sample(1:dim(rece)[2],
                size = J,
                replace = F)
x_sample <- sample(seq(2, dim(rece)[2], 2),
                size = J,
                replace = F)

nbData <- list(Y = rece[span_index, y_sample],
                X = X[span_index, x_sample])

nbConsts <- list(N = n,
                J = J)

nbInits <- list(B0 = 0,
                B1 = 0,
                b0 = rep(0, J),
                b1 = rep(0, J),
                sigB0 = 0.0001,
                sigB1 = 0.0001)

nbModel <- nimbleModel(code = nbCode,
                       data = nbData,
                       inits = nbInits,
                       constants = nbConsts)

#compile nimble model to C++ code-much faster runtime
```

```r
C_nbModel <- compileNimble(nbModel, showCompilerOutput = FALSE)

#configure the MCMC
nbModel_conf <- configureMCMC(nbModel)

nbModel_conf$monitors <- c("B0", "B1", "sigB0", "sigB1")
nbModel_conf$addMonitors2(c("b0", "b1"))

#samplers
nbModel_conf$removeSamplers(c("B0", "B1", "sigB0", "sigB1", "b0", "b1"))
nbModel_conf$addSampler(target = c("B0", "B1", "sigB0", "sigB1"),
                        type = "AF_slice")
for(j in 1:J){
   nbModel_conf$addSampler(target = c(paste("b0[", j, "]", sep = ""),
                                      paste("b1[", j, "]", sep = "")),
                        type = "AF_slice")
}

#thinning to conserve memory when the samples are saved below
thinning <- 10
nbModel_conf$setThin(thinning)
nbModel_conf$setThin2(thinning)

set.seed(1)

#build MCMC
nbModelMCMC <- buildMCMC(nbModel_conf)

#compile MCMC to C++-much faster
C_nbModelMCMC <- compileNimble(nbModelMCMC, project = nbModel)

niter <- 100000

#save the MCMC chain (monitored variables) as a matrix
nimble_samples_big <- runMCMC(C_nbModelMCMC, niter = niter)

#save samples
save(nimble_samples_big,
        file = paste(datapath,
                    "mcmc_samples_nimble_big.RData",
                    sep = ""))
```

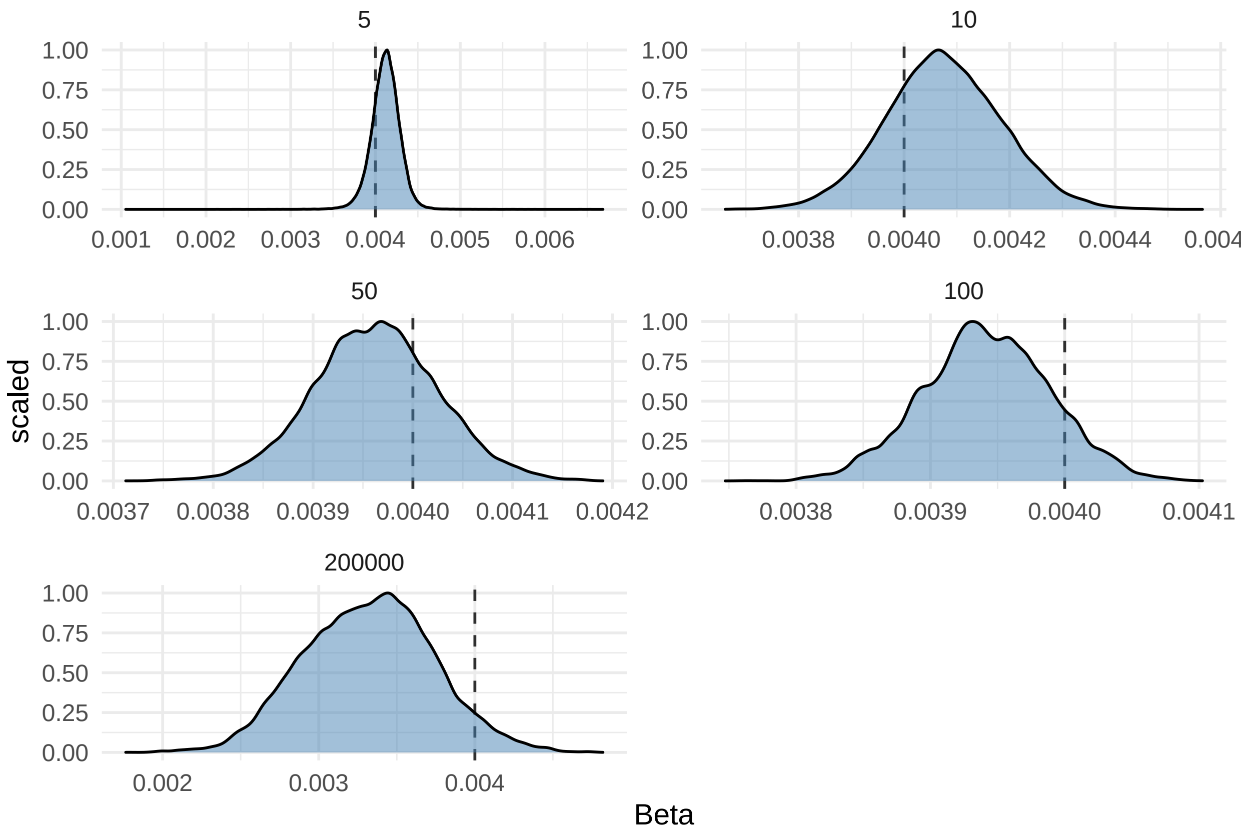
Next, run the chronup analysis on the same large-sample dataset,

```r
load(paste(datapath,
        "simdata3.RData",
        sep = ""))

rece <- bigmemory::attach.big.matrix(simdata3$count_ensemble)
X <- bigmemory::attach.big.matrix(paste(datapath,
                                "X_mat_desc",
                                sep = ""))

nsamples <- dim(rece)[2]
```

```r
span_index <- which(simdata3$new_timestamps <= start &
                    simdata3$new_timestamps >= end)

N <- length(span_index)

startvals <- c(0, 0, rep(0.1, N))
startscales <- c(0.1, 0.0002, rep(0.5, N))

adapt_limit <- floor(nsamples * 0.25)

adapt_sample_y <- 1:adapt_limit
adapt_sample_x <- 1:(adapt_limit * 2)

mcmc_samples_adapt <- chronup::regress(Y = rece[span_index, adapt_sample_y],
                                        X = X[span_index, adapt_sample_x],
                                        model = "nb",
                                        startvals = startvals,
                                        scales = startscales,
                                        adapt = T)

burnin <- floor(dim(mcmc_samples_adapt$samples)[1] * 0.2)
indeces <- seq(burnin, dim(mcmc_samples_adapt$samples)[1], 1)
adapt_startvals <- colMeans(mcmc_samples_adapt$samples[indeces, ])

burnin <- floor(dim(mcmc_samples_adapt$scales)[1] * 0.2)
indeces <- seq(burnin, dim(mcmc_samples_adapt$scales)[1], 1)
adapt_startscales <- colMeans(mcmc_samples_adapt$scales[indeces, ])

chronup_samples_big <- chronup::regress(Y = rece[span_index, ],
                                X = X[span_index, ],
                                model = "nb",
                                startvals = adapt_startvals,
                                scales = adapt_startscales,
                                adapt = F)
save(chronup_samples_big,
        file = paste(datapath,
                    "mcmc_samples_chronup_big.RData",
                    sep = ""))
```

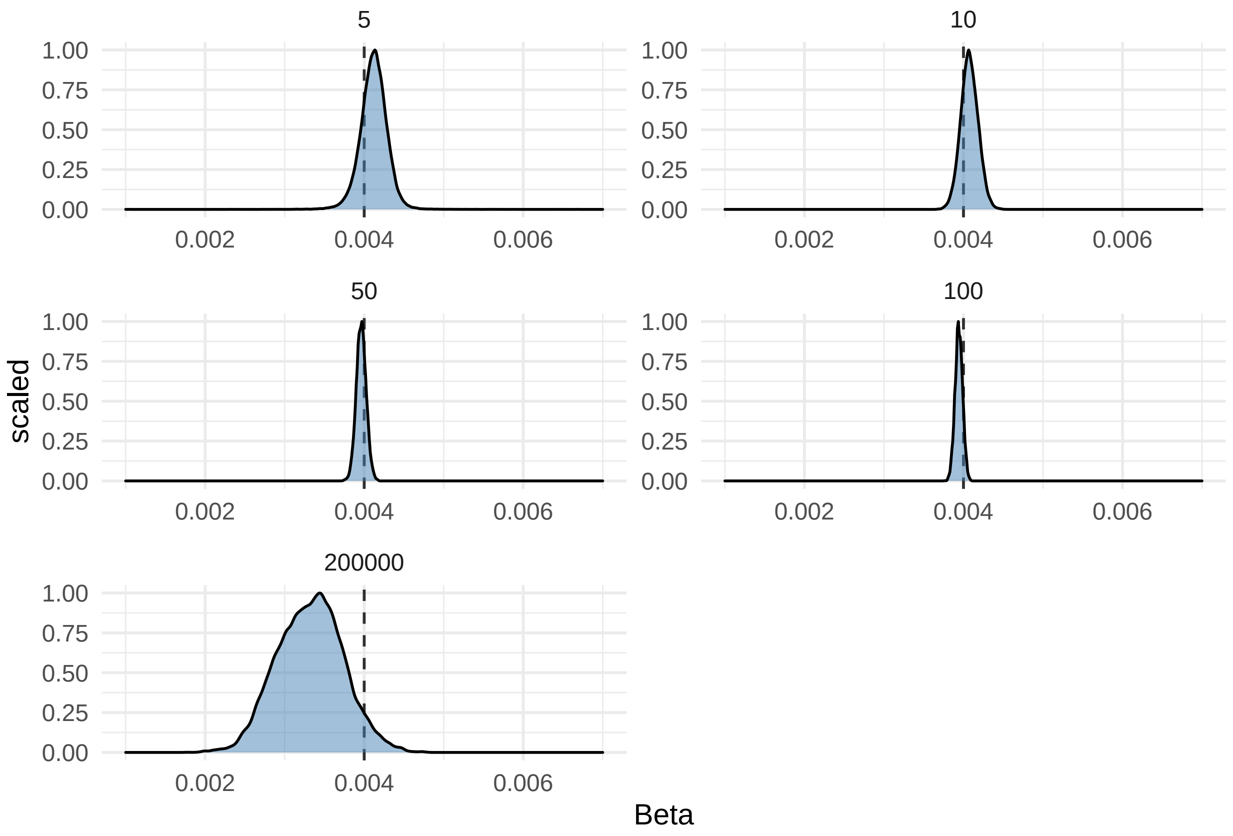
Then, like last time, plot the results,

```r
load(paste(datapath,
            "mcmc_samples_nimble_5_big.RData",
            sep = ""))
load(paste(datapath,
            "mcmc_samples_nimble_10_big.RData",
            sep = ""))
load(paste(datapath,
            "mcmc_samples_nimble_50_big.RData",
            sep = ""))
load(paste(datapath,
            "mcmc_samples_nimble_100_big.RData",
            sep = ""))
load(paste(datapath,
            "mcmc_samples_chronup_big.RData",
            sep = ""))

n_hier <- dim(nimble_samples_5_big$samples)[1]
n_alt <- dim(chronup_samples_big)[1]

burnin_hier <- floor(n_hier * 0.1)
burnin_alt <- floor(n_alt * 0.1)

iteration_hier <- (burnin_hier + 1):n_hier
iteration_hier_all <- rep(iteration_hier, 4)
iteration_alt <- (burnin_alt + 1):n_alt

n_retained_hier <- length(iteration_hier)
n_retained_alt <- length(iteration_alt)

plot_samples_hier_5 <- nimble_samples_5_big$samples[iteration_hier, 2]
plot_samples_hier_10 <- nimble_samples_10_big$samples[iteration_hier, 2]
plot_samples_hier_50 <- nimble_samples_50_big$samples[iteration_hier, 2]
plot_samples_hier_100 <- nimble_samples_100_big$samples[iteration_hier, 2]
plot_samples_hier <- c(plot_samples_hier_5,
                    plot_samples_hier_10,
                    plot_samples_hier_50,
                    plot_samples_hier_100)
plot_samples_alt <- chronup_samples_big[iteration_alt, 2]

model_name_hier <- rep("hier", n_retained_hier * 4)
model_name_alt <- rep("alt", n_retained_alt)

J_hier <- rep(c("5", "10", "50", "100"), each = n_retained_hier)
J_alt <- rep("200000", n_retained_alt)

J_both <- factor(c(J_hier, J_alt), levels = c("5", "10", "50", "100", "200000"))

mcmc_samples_both <- data.frame(Iteration = c(iteration_hier_all,
                                              iteration_alt),
                            Model = c(model_name_hier,
                                      model_name_alt),
                            J = J_both,
                            Beta = c(plot_samples_hier,
                                     plot_samples_alt))

ggplot(data = mcmc_samples_both) +
    geom_vline(xintercept = beta,
                linetype = 2,
                alpha = 0.8) +
    geom_density(mapping = aes(y = ..scaled.., x = Beta),
                fill = "steelblue",
                alpha = 0.5) +
    facet_wrap(~ J, ncol = 2, scales = "free") +
    theme_minimal()
```



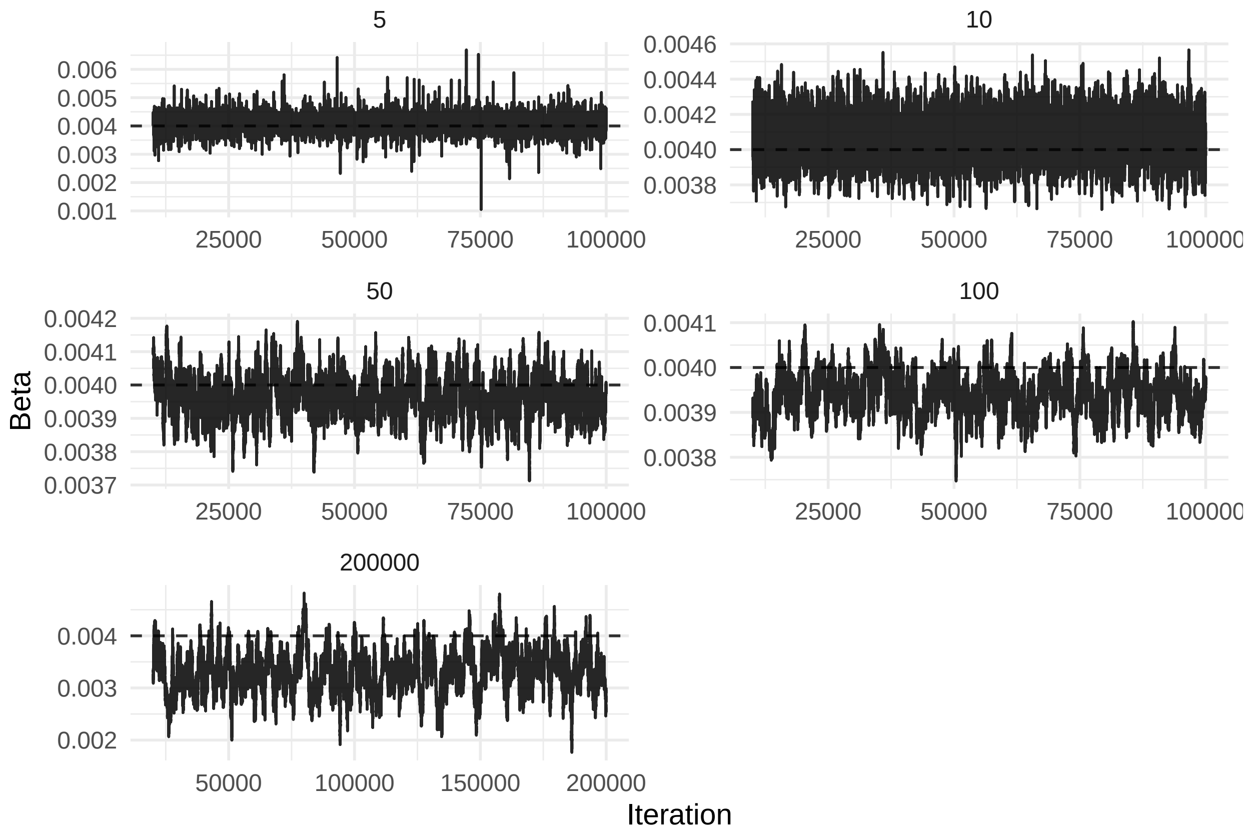
As before, to make comparisons easier, we can re-produce the same plot, but this time limit the x-axis to a region around the target value for β and align the plot axes:

```r
ggplot(data = mcmc_samples_both) +
    geom_vline(xintercept = beta,
               linetype = 2,
               alpha = 0.8) +
    geom_density(mapping = aes(y = ..scaled.., x = Beta),
               fill = "steelblue",
               alpha = 0.5) +
    facet_wrap(~ J, ncol = 2, scales = "free") +
    scale_x_continuous(limits = c(0.001, 0.007)) +
    theme_minimal()
```



Lastly, we should also inspect the mcmc trace plots for each model/estimate, and then examine the Geweke statistics for the main parameter,

```r
ggplot(data = mcmc_samples_both) +
    geom_hline(yintercept = beta,
               linetype = 2,
               alpha = 0.8) +
    geom_path(mapping = aes(y = Beta, x = Iteration),
               alpha = 0.85) +
    facet_wrap(~ J, ncol = 2, scales = "free") +
    theme_minimal()
```



```r
geweke.diag(cbind(plot_samples_hier_5,
                  plot_samples_hier_10,
                  plot_samples_hier_50,
                  plot_samples_hier_100,
                  plot_samples_alt))
```

```
## Warning in cbind(plot_samples_hier_5, plot_samples_hier_10,
## plot_samples_hier_50, : number of rows of result is not a multiple of vector
## length (arg 1)
```

```
##
## Fraction in 1st window = 0.1
## Fraction in 2nd window = 0.5
##
##   plot_samples_hier_5  plot_samples_hier_10  plot_samples_hier_50
##             -0.347998             -0.004298              0.037246
## plot_samples_hier_100      plot_samples_alt
##             -0.159882             -1.389440
```