Lifetime Simulation Writing

Piper Zimmerman

March 25, 2023

The purpose of this simulation is to show that if lifetime data actually comes from a Weibull distribution, which is expected, then modeling it with an exponential distribution is not ideal. Inverting it is even less ideal. People invert lifetime data because they assume that the lifetime is just the inverse of the mortality parameter, which is only true if it comes from an exponential distribution. We are questioning the current practices of how to model lifetime and mortality.

Individual Data:

This chunk of the code loads the necessary packages, `ggplot2, HDInterval, rjags,` and `dplyr`. I set the true values of a, b, and c. These true values were obtained by finding a mortality curve in a previous text, and finding the values of a, b, and c, to get that quadratic, $aT^2 + bT + c$. The parameter, λ, is a function of temperature, of the form $f(T) = aT^2 + bT + c$, was taken from Figure B.1 (d) of *Understanding uncertainty in temperature effects on vector-borne disease: A Bayesian approach*, and the parameters a, b, and c were found. The result was $f(T) = 0.001094 * T^2 - 0.0488 * T + 0.609$. I then create the function, `f`, that is this quadratic. `data.list` is an empty list so that I can store 100 different data sets, to run these models 100 times. The loop then creates 100 datasets, with columns `T` for the temperature, and `trait`, which is lifetime values simulated from an exponential distribution with a rate parameter of `f(T)`. We use 5 temperatures, with 100 simulated observations at each temperature. A data frame (500x2) was made with the simulated value as the response, and the temperature as the dependent variable. `b` simply is the length of `data.list` to use in the upcoming loop. `epsilon` is the value that is used in the truncation while creating the JAGS model, since you cannot have a lifetime of below 0. We chose a rounded value of 0.01 since we expect the longest lifetime to be around 90-100 days. `true_values` is a vector of the true values of a, b, and c. Both a and b are in log space due to their small values.

```
library(ggplot2)
library(HDInterval)
library(rjags)
library(dplyr)

true.a<-0.001094
true.b<-0.0488
true.c<-0.609
f<-function(x,a=0.001094,b=0.0488,c=0.609){
  a*x^2-b*x+c
}

data.list<-list()
set.seed(52)
for(i in 1:100){
  T1<-rexp(n=100,f(10))
  T2<-rexp(n=100,f(17))
  T3<-rexp(n=100,f(21))
  T4<-rexp(n=100,f(25))
  T5<-rexp(n=100,f(32))
```

```
  data.list[[i]]<-data.frame(
    T=c(rep(10,100),rep(17,100),rep(21,100),rep(25,100),rep(32,100)),
    trait=c(T1,T2,T3,T4,T5)
  )
}


b<-length(data.list)

epsilon<-0.01

true_values<-c(log(true.a),log(true.b),true.c, NA,NA)
```

This chunk of the code makes a list of matrices in order to store the posterior draws of the parameters a, b, and c. The list of length 4 is made to store the 3 parameters, as well as the deviance, one in each list. Within each list, there are 100 matrices, that are each 5x4. This way, for each of the 100 models in the upcoming loop, and each of the 5 chains, the mean, median, lower hdi bound, and upper hdi bound can all be stored. This is done for each parameter, as well as the deviance.

```
param_list123 <- vector("list", length = 4)
for (i in 1:4) {
  # Initialize the inner list
  inner_list123 <- vector("list", length = b)
  # Populate the inner list
  for (j in 1:b) {
    # Initialize a 5x4 matrix
    matrix_data123 <- matrix(0, nrow = 5, ncol = 4)
    inner_list123[[j]] <- matrix_data123
  }
  param_list123[[i]] <- inner_list123
}
```

This chunk of code is the loop to make the 100 different `JAGS` models. This model is from using individual data points generated from an exponential distribution. As mentioned previously, a and b are in log space. Since μ has to be positive, we added an indicator function. If μ is negative, the value will default to `epsilon`, which is 0.01. `trait` then is assumed to have an exponential distribution with parameter μ. We are modeling the lifetimes as an exponential distribution, where the parameter comes from the mortality quadratic. The loop goes through and does this for each of the previously generated 100 data sets. `samp` is made, which is the mcmc list of the `JAGS` model.

```
for(i in 1:b){
  if (i %% 10 == 0) {
    print(i)
  }
  # Model
  sink("lambda.txt")
  cat("model{
  #Priors
  #If mu<0, make it small number
  #And nonzero
  la~dnorm(0, 1/10) #has to be positive exp(1)?
  b.l~dnorm(0, 1/10) #Has to be positive, because function has neg sign
  c~dexp(0.5) #Has to be positive
  epsilon<-0.00001
  #Likelihood
```

```
  for (i in 1:N.obs){
    mu[i] <- (exp(la) * temp[i]^2 - exp(b.l) * temp[i] + c)*
    ((exp(la) * temp[i]^2 - exp(b.l) * temp[i] + c)>epsilon) +
    ((exp(la) * temp[i]^2 - exp(b.l) * temp[i] + c)<=epsilon)*epsilon
    trait[i] ~ dexp(mu[i])
  }
  }",file="lambda.txt")

  # Settings
  parameters <- c("la", "b.l", "c")
  inits<-function(){list(
    la = log(0.001),
    b.l = log(0.05),
    c = 0.6
  )}
  ni <- 60000
  nb <- 30000
  nt <- 8
  nc <- 5
  data.raw<- data.list[[i]]
  data <- data.list[[i]]
  trait <- data$trait
  N.obs <- length(trait)
  temp <- data$T

  # List data
  jag.data<-list(trait = trait, N.obs = N.obs, temp = temp)

  # Run model
  mod.fit <- jags(data=jag.data, inits=inits, parameters.to.save=parameters,
                  model.file="lambda.txt", n.thin=nt, n.chains=nc, n.burnin=nb,
                  n.iter=ni, DIC=T, working.directory=getwd())
  mod.fit.mcmc <- as.mcmc(mod.fit)
  closeAllConnections()
  samp<-as.mcmc.list(mod.fit.mcmc)
}
```

This chunk of code saves the posterior draws. Each model gets it's own 5x4 matrix. For each chain of each model, the mean, median, lower hdi bound, and upper hdi bound are saved into the matrix. This leaves 100 matrices for each parameter, in each component of the list.

```
for(j in 1:nc){
    #la
    param_list123[[1]][[i]][j,1]<-mean(samp[[j]][,4])
    param_list123[[1]][[i]][j,2]<-median(samp[[j]][,4])
    param_list123[[1]][[i]][j,3]<-hdi(samp[[j]][,4])[1]
    param_list123[[1]][[i]][j,4]<-hdi(samp[[j]][,4])[2]
    # b.l
    param_list123[[2]][[i]][j,1]<-mean(samp[[j]][,1])
    param_list123[[2]][[i]][j,2]<-median(samp[[j]][,1])
    param_list123[[2]][[i]][j,3]<-hdi(samp[[j]][,1])[1]
    param_list123[[2]][[i]][j,4]<-hdi(samp[[j]][,1])[2]
    # c
    param_list123[[3]][[i]][j,1]<-mean(samp[[j]][,2])
```

```
    param_list123[[3]][[i]][j,2]<-median(samp[[j]][,2])
    param_list123[[3]][[i]][j,3]<-hdi(samp[[j]][,2])[1]
    param_list123[[3]][[i]][j,4]<-hdi(samp[[j]][,2])[2]
    # dev
    param_list123[[4]][[i]][j,1]<-mean(samp[[j]][,3])
    param_list123[[4]][[i]][j,2]<-median(samp[[j]][,3])
    param_list123[[4]][[i]][j,3]<-hdi(samp[[j]][,3])[1]
    param_list123[[4]][[i]][j,4]<-hdi(samp[[j]][,3])[2]
  }
```

This chunk of code saves the data and summary statistics to show in the upcoming plots. First, I created `df.new` which is the posterior draws from the first chain of `samp`, with every fourth draw chosen. `xseq` is a sequence of temperatures from 10 to 35, with 250 values. `xdf` is a $(250 * nrow(df.new))$x2 dataframe. There is a column of temperatures, and a column of iteration. It is so every row or function of posterior draws can be evaluated at each of those 250 temperatures. The `value.inv` column is created by evaluating the inverse of the quadratic function at each row, then inverting it (this column is not really used). The `value` column is the same, just not inverted. I then us mutate to create a `trunc.eval` column, where the functions are evaluated with the previous indicator function; if the original evaluation is less then 0, it is given a value of epsilon, or 0.00001. `trunc.inv` is this column inverted.

Next, I grouped by point, or the sequence of 250 temperatures, and created the summary statistics, including the mean, median, lower, and upper hdi bounds. I did this using the inverted and noninverted data that has already had the values below 0 changed. I made a function for the true curve, and the inverse of the true curve, and created two more columns in the data frame for the true curve, and the true curve inverted.

```
df.new123 = samp[[1]][seq(1, nrow(samp[[1]]), 4), ]
  xseq<-seq(from=10,to=35,length.out=250)
  xdf123 <- expand.grid(point = xseq, iteration = 1:nrow(df.new123))

  # Apply the quadratic equation for each combination
  xdf123$value.inv <- apply(xdf123, 1, function(row) {
    i <- row["iteration"]
    location <- row["point"]
    1/(exp(df.new123[i, 4]) * location^2 - exp(df.new123[i, 1]) * location +
         df.new123[i, 2])
  })
  xdf123$value <- apply(xdf123, 1, function(row) {
    i <- row["iteration"]
    location <- row["point"]
    exp(df.new123[i, 4]) * location^2 - exp(df.new123[i, 1]) * location + df.new123[i, 2]
  })
  xdf123<- xdf123|> mutate(trunc.eval=ifelse(xdf123$value>0,xdf123$value,epsilon),
                       trunc.inv=1/trunc.eval)
  mean.xdf123<-xdf123 |> group_by(point) |> summarize(avg.value.inv=mean(trunc.inv),
                                                      avg.value=mean(trunc.eval),
                                                      med.value.inv=median(trunc.inv),
                                                      med.value=median(trunc.eval),
                                                      lower.hdi.inv=hdi(trunc.inv)[1],
                                                      upper.hdi.inv=hdi(trunc.inv)[2],
                                                      lower.hdi=hdi(trunc.eval)[1],
                                                      upper.hdi=hdi(trunc.eval)[2])
  true_curve.inv <- function(x) 1/(true.a * x^2 - true.b * x + true.c)
  true_curve <- function(x) (true.a * x^2 - true.b * x + true.c)
  mean.xdf123$true_curve <- true_curve(xseq)
  mean.xdf123$true_curve.inv <- true_curve.inv(xseq)
```

This chunk of code creates the actual plots. There are plots for the true curve and the mean values of the evaluated posterior draws, the true curve and the median values of the evaluated posterior draws, the inverted true curve and the mean values of the evaluated posterior draws inverted, and the inverted true curve and the median values of the evaluated posterior draws inverted. The original data is overlaid onto the flipped lifetime curves. The curves that are concave up are the mortality curves, while the curves that are concave down (inverted) are the lifetime curves.

```
  plot123_mean<-ggplot(mean.xdf123, aes(x = point)) +
    geom_line(aes(y = avg.value), color = "black") +
    geom_ribbon(aes(ymin = lower.hdi, ymax = upper.hdi),
                fill = "coral1", alpha = 0.3)+
    geom_line(aes(y=true_curve),color="deepskyblue2",linetype="dashed")+
    labs(title = "Mean Curve with Interval Bands and True Curve",
         x = "Temperature",
         y = "Response")+
    theme_minimal()
  plot123_med<-ggplot(mean.xdf123, aes(x = point)) +
    geom_line(aes(y = med.value), color = "black") +
    geom_ribbon(aes(ymin = lower.hdi, ymax = upper.hdi),
                fill = "coral1", alpha = 0.3)+
    geom_line(aes(y=true_curve),color="deepskyblue2",linetype="dashed")+
    labs(title = "Median Curve with Interval Bands and True Curve",
         x = "Temperature",
         y = "Response")+
    theme_minimal()
  plot123_mean.inv<-ggplot(mean.xdf123, aes(x = point)) +
    geom_line(aes(y = avg.value.inv), color = "black") +
    geom_ribbon(aes(ymin = lower.hdi.inv, ymax = upper.hdi.inv),
                fill = "coral1", alpha = 0.3)+
    geom_line(aes(y=true_curve.inv),color="deepskyblue2",linetype="dashed")+
    geom_point(aes(x=T,y=trait),data=data.raw)+
    labs(title = "Mean Curve with Interval Bands and True Curve",
         x = "Temperature",
         y = "Response")+
    theme_minimal()
  plot123_med.inv<-ggplot(mean.xdf123, aes(x = point)) +
    geom_line(aes(y = med.value.inv), color = "black") +
    geom_ribbon(aes(ymin = lower.hdi.inv, ymax = upper.hdi.inv),
                fill = "coral1", alpha = 0.3)+
    geom_line(aes(y=true_curve.inv),color="deepskyblue2",linetype="dashed")+
    geom_point(aes(x=T,y=trait),data=data.raw)+
    labs(title = "Median Curve with Interval Bands and True Curve",
         x = "Temperature",
         y = "Response")+
    theme_minimal()

gridExtra::grid.arrange(plot123_mean,plot123_med,nrow=1)
gridExtra::grid.arrange(plot123_mean.inv,plot123_med.inv,nrow=1)
```

This chunk creates histograms of the posterior draws, and overlays the prior distribution onto it. The true value of each parameter is marked as well by the red line.

```
true_values<-c(log(0.001094),log(0.0488),0.609, NA)
hist_list123 <- vector("list", length = 4)

# Loop through indices 1 to 4
for (i in 1:4) {
  # Extract the first column from each matrix in param_list[[i]]
  first_column_list123 <- lapply(param_list123[[i]], function(matrix) matrix[, 1])

  # Combine the vectors into a single vector
  combined_vector123 <- unlist(first_column_list123)
  x<-seq(min(combined_vector123)-50,max(combined_vector123)+1, length=1000)
  # Create a histogram and store it in the list
  hist_list123[[i]] <- hist(combined_vector123,
                            main = paste("Combined Histogram Mean (Parameter", i, ")"),
                          xlab = "Values", col = "lightblue",
                          border = "black", breaks = 10,freq=FALSE)
  abline(v = true_values[i], col = "red", lwd = 2)
  if(i==1){
    lines(x,dnorm(x,0, sqrt(10)), col="green", lty=2, lwd=2)
  }
  if(i==2){
    lines(x, dnorm(x,0, sqrt(10)), col="black", lty=2, lwd=2)
  }
  if(i==3){
    lines(x, dexp(x, 0.5), col="yellow", lty=2, lwd=2)
  }
  if(i==5){
    lines(x, dexp(x, 0.1), col="purple", lty=2, lwd=2)
  }
}
```

This chunk of code calculates the coverage. For each row of the matrices of each parameter's posterior mean, we calculate if the true value is in the hdi bounds of posterior estimates (not the hdi of the posterior mean in the upcoming table). We then take these and average them.

```
proportions_list_a123<- lapply(param_list123[[1]], function(matrix) {
  # Extract columns
  col2 <- matrix[, 1]
  col3 <- matrix[, 3]
  col4 <- matrix[, 4]

  # Calculate proportion
  proportion <- mean(log(true.a) > col3 & log(true.a) < col4)
  return(proportion)
})

# Display the proportions
#print(proportions_list_median)
mean(unlist(proportions_list_a123))

proportions_list_b123<- lapply(param_list123[[2]], function(matrix) {
  # Extract columns
  col2 <- matrix[, 1]
  col3 <- matrix[, 3]
```

```
  col4 <- matrix[, 4]

  # Calculate proportion
  proportion <- mean(log(true.b) > col3 & log(true.b) < col4)
  return(proportion)
})

# Display the proportions
#print(proportions_list_median)
mean(unlist(proportions_list_b123))

proportions_list_c123<- lapply(param_list123[[3]], function(matrix) {
  # Extract columns
  col2 <- matrix[, 1]
  col3 <- matrix[, 3]
  col4 <- matrix[, 4]

  # Calculate proportion
  proportion <- mean(true.c > col3 & true.c < col4)
  return(proportion)
})

mean(unlist(proportions_list_c123))
```

This chunk of code creates the function and uses said function to calculate the RMSE values for each parameter. This is done by taking the observed values, or the posterior means of each chain for each model, and subtracting the true value. That is squared, the mean is taken, and then the square root is taken.

```
calculate_rmse_a_lambda <- function(mat) {
  observed <- mat[, 1]
  predicted <- rep(log(true.a), length(observed))
  # Calculate RMSE
  rmse <- sqrt(mean((observed - predicted)^2))
  return(rmse)
}
calculate_rmse_b_lambda <- function(mat) {
  observed <- mat[, 1]
  predicted <- rep(log(true.b), length(observed))
  # Calculate RMSE
  rmse <- sqrt(mean((observed - predicted)^2))
  return(rmse)
}
calculate_rmse_c_lambda <- function(mat) {
  observed <- mat[, 1]
  predicted <- rep(true.c, length(observed))
  # Calculate RMSE
  rmse <- sqrt(mean((observed - predicted)^2))
  return(rmse)
}


# Apply the function to each element in the list
rmse_values_a_lambda <- sapply(param_list123[[1]], calculate_rmse_a_lambda)
rmse_values_b_lambda <- sapply(param_list123[[2]], calculate_rmse_b_lambda)
```

```
rmse_values_c_lambda <- sapply(param_list123[[3]], calculate_rmse_c_lambda)
```

The last chunk of code creates a table. The rows represent parameters a, b, and c, respectively. The first column is the true value. The second column is the mean of each value in the parameter list. It is the mean of all the means of the posterior draws. The third and fourth columns are the lower and upper hdi bounds of this mean. The fifth column is the coverage, or how many times the true value of the parameter was within the hdi bounds. The last column is the calculated RMSE.

```
tab.lambda<-matrix(0,nrow=3,ncol=6)
row.names(tab.lambda)<-c("a","b","c")
colnames(tab.lambda)<-c("True Value","Mean","Lower HDI of Mean","Upper HDI of Mean","Coverage","RMSE")
tab.lambda[1,1]<-true.a
tab.lambda[2,1]<-true.b
tab.lambda[3,1]<-true.c
tab.lambda[1,2]<-exp(mean(unlist(lapply(param_list1[[1]],
                                        function(matrix) matrix[, 1]))))
tab.lambda[2,2]<-exp(mean(unlist(lapply(param_list1[[2]],
                                        function(matrix) matrix[, 1]))))
tab.lambda[3,2]<-mean(unlist(lapply(param_list1[[3]],
                                    function(matrix) matrix[, 1])))
tab.lambda[1,3]<-exp(hdi(unlist(lapply(param_list1[[1]],
                                       function(matrix) matrix[, 1])))[1])
tab.lambda[2,3]<-exp(hdi(unlist(lapply(param_list1[[2]],
                                       function(matrix) matrix[, 1])))[1])
tab.lambda[3,3]<-hdi(unlist(lapply(param_list1[[3]],
                                   function(matrix) matrix[, 1])))[1]
tab.lambda[1,4]<-exp(hdi(unlist(lapply(param_list1[[1]],
                                       function(matrix) matrix[, 1])))[2])
tab.lambda[2,4]<-exp(hdi(unlist(lapply(param_list1[[2]],
                                       function(matrix) matrix[, 1])))[2])
tab.lambda[3,4]<-hdi(unlist(lapply(param_list1[[3]],
                                   function(matrix) matrix[, 1])))[2]
tab.lambda[1,5]<-mean(unlist(proportions_list_a1))
tab.lambda[2,5]<-mean(unlist(proportions_list_b1))
tab.lambda[3,5]<-mean(unlist(proportions_list_c1))
tab.lambda[1,6]<-mean(rmse_values_a_lambda)
tab.lambda[2,6]<-mean(rmse_values_b_lambda)
tab.lambda[3,6]<-mean(rmse_values_c_lambda)
```

We use lifetime data, generated from an exponential distribution withe parameter a function of temperature, where this function comes from the mortality curve (quadratic). We then fit this `JAGS` model, modeling the data as exponential with mu being from that quadratic function. We get `samp` out, which gives us the posterior draws (estimates) of each parameter, for each iteration in each chain. We use these posterior draws to make histograms of the posterior distributions, overlaying the prior distribution to compare. We calculate the proportion of times that the hdi bounds of the posterior draws contains the true value, in each chain, in each model. We also calculate the RMSE, then put all of this information in a table. The hdi bounds specified in the table are the hdi bounds of the means, not of all of the posterior samples.

The purpose of the plots evaluating each posterior draw as a function is to compare the data that we generated the model from, and the curve we get. When we get the posterior draws to give us values for the mortality curve that data was generated from, we evaluate them at 250 temperatures to see what our posterior draws are like compared to the true function. Once we evaluate these, we flip them to get the lifetime curve. This allows us to compare the data the model was made from to the function it produces. We truncate the model after evaluating the posterior draws, since the mortality curve can not go below zero. However, the plots while using the inverse of the individual data or the mean inverse (or inverse mean), run into issues. Many of

the values get truncated to 0.00001, so when inverting the curve, they have extremely large values, this is why I created an ifelse statement to truncate the max value to 100.

Mean Data:

In the next method of models, we use not the individual lifetime data points, but averages. For each 100 individual points at each temperature, the lifetimes are in groups of 10, and the mean of those 10 is used. This gives us 10 points at each of the 5 temperatures. This is done because sometimes in the field, scientists will not record each individual data point. The only thing that is changed from the individual lambda method to the mean method, is this.

The code chunk below is what splits the data into groups of 10 and calculates the means.

```
group_means <- tapply(data.list[[i]]$trait, data.list[[i]]$T, function(x) {
    split_values <- split(x, rep(1:10, each = 10))
    means <- sapply(split_values, mean)
    return(means)
  })
  df.mean<- data.frame(
    T=c(rep(10,10),rep(17,10),rep(21,10),rep(25,10),rep(32,10)),
    trait = unlist(group_means)
  )
```

Inverse Individual Data:

In the third method of models, we use the individual data points, but inverted. For an exponential distribution, inverting the mean lifetime gives the rate. By inverting the data points, we are trying to estimate rate directly by fitting a curve to $\frac{1}{Lifetime}$. In the model, we put a prior on sigma, and square and invert that to use tau. We are now modeling the trait as normal with mean μ from the quadratic mortality function, and precision tau. When we make the plots evaluating each posterior draw as a function, we can put this inverted data on the non-inverted function to show the posterior mortality curves and the inverted data that we used.

```
  #Model
  sink("inv_lambda.txt")
  cat("model{
  #Priors
  #If mu<0, make it small number
  #And nonzero
  la~dnorm(0, 1/10) #has to be positive exp(1)?
  b.l~dnorm(0, 1/10) #Has to be positive, because function has neg sign
  c~dexp(0.5) #Has to be positive
  sig~dexp(4000)
  sig2<-sig^2
  tau<-1/sig2
  epsilon<-0.0001
  #Likelihood
  for (i in 1:N.obs){
    mu[i] <- (exp(la) * temp[i]^2 - exp(b.l) * temp[i] + c)*
    ((exp(la) * temp[i]^2 - exp(b.l) * temp[i] + c)>epsilon) +
    ((exp(la) * temp[i]^2 - exp(b.l) * temp[i] + c)<=epsilon)*epsilon
    trait[i] ~ dnorm(mu[i],tau) T(0,)
  }
}",file="inv_lambda.txt")
```

Mean Inverse Data:

In the fourth method of models, we take the mean of the 10 groups of 10 at each temperature like previously, then invert the points. The specified model is the same as using the individual inverted data.

```
data.raw <- data.list[[i]]
  group_means <- tapply(data.raw$trait, data.raw$T, function(x) {
    split_values <- split(x, rep(1:10, each = 10))
    means <- sapply(split_values, mean)
    return(means)
  })
  df.mean<- data.frame(
    T=c(rep(10,10),rep(17,10),rep(21,10),rep(25,10),rep(32,10)),
    trait = unlist(group_means)
  )
  data<-df.mean
  trait <- 1/(data$trait)
  N.obs <- length(trait)
  temp <- data$T
```

Inverse Mean Data:

In the fifth method of models, we invert the individual lifetime data points, then take the mean of the 10 groups of 10 at each temperature like previously. The specified model is the same as using the individual inverted data.

Weibull

The next step is to simulate the lifetime data from a Weibull distribution and run the same models, as well as a Weibull model. This will allow us to see if the current modeling technique in the field is adequate, even if lifetime data follows a Weibull distribution rather than an exponential distribution. To do this, we need to simulate Weibull data, using the median as the scale parameter, and a reasonable shape parameter (currently 3). The problem I'm currently having is the different parameterizations between `R` and `JAGS`, where `R` uses lambda and `JAGS` uses the generalized gamma with inverse lambda.

The following parameterizations are:

$$R: \ (\frac{a}{b})(\frac{x}{b})^{a-1}e^{-(\frac{x}{b})^a} \tag{1}$$

$$Wikipedia: \ (\frac{k}{\lambda})(\frac{x}{\lambda})^{k-1}e^{-(\frac{x}{\lambda})^k} \tag{2}$$

$$JAGS: \ \frac{b\lambda^{br}y^{br-1}e^{-(\lambda y)^b}}{\Gamma(r)} \tag{3}$$

Where Equation 1 is the `R` parameterization, Equation 2 is the Wikipedia parameterization, and Equation 3 is the `JAGS` parameterization. If we let $r = 1$ in Equation 3, $y = x$, and $\lambda = \frac{1}{\lambda}$, we see we can rewrite it as:

$$JAGS: \ (\frac{b}{\lambda})(\frac{1}{\lambda})^{b-1}x^{b-1}e^{-(\frac{x}{\lambda})^b} \tag{4}$$

From these, we can see the following relationships:

$$R - Wikipedia: \ a = k, \ b = \lambda$$

$$JAGS - R: \ y = x, \ \lambda = \frac{1}{b}, \ b = a$$

$$JAGS - Wikipedia: \ y = x, \lambda = \frac{1}{\lambda}, b = k$$

When generating data from the Weibull distribution, we exponentiated everything so that nothing was less than 0. Our estimated shape parameter was 3, based on how we thought the data should look. With `f.exp` being our exponentiated quadratic, a concave up function, we are representing the inverse of the median lifetime. This needs to be flipped in order to get a concave down function, which is the purpose of `my.inv.lambda`. This function is then used as the scale parameter in simulating data. `my.inv.lambda` is a function to calculate lambda using the median ($M(T) = \lambda(ln2)^{1/k}$, so $\lambda = \frac{M(T)}{(ln2)^{1/k}}$). This function calculates $\frac{1}{\lambda}$.

```
true.exp.a<- 0.00826
true.exp.b<- 0.37
true.exp.c<- 1.406
# Making quadratic function
f.exp<-function(x,a=0.00826,b=0.37,c=1.406){
  exp(-a*x^2+b*x-c)
}
# Set reasonable shape parameter
sh.est<-3
# Inverse lambda function
my.inv.lambda<-function(x,a=0.00826,b=0.37,c=1.406,sh=3){
  (log(2))^(1/sh.est)/f.exp(x,a,b,c) ## fx0 is the median here
}
# Generate data from 5 temperatures and create data frame
T1W<-rweibull(n=100,scale=my.inv.lambda(10),shape=sh.est)
T2W<-rweibull(n=100,scale=my.inv.lambda(17),shape=sh.est)
T3W<-rweibull(n=100,scale=my.inv.lambda(21),shape=sh.est)
T4W<-rweibull(n=100,scale=my.inv.lambda(25),shape=sh.est)
T5W<-rweibull(n=100,scale=my.inv.lambda(32),shape=sh.est)
dfW<-data.frame(
  T=c(rep(10,100),rep(17,100),rep(21,100),rep(25,100),rep(32,100)),
  trait=c(T1W,T2W,T3W,T4W,T5W)
)
```

This shows the `JAGS` model, where `med[i]` is the median lifetime, using our quadratic function, `lambda[i]` is the scale parameter, and `trait[i]` follows a generalized gamma distribution with r set equal to 1 in order to make it equivalent to the Weibull distribution, scale parameter `1/lambda[i]`, to represent the inverse of the lambda that was used to generate the data.

```
sink("weibull.model.base.txt")
cat("model{
  #Priors
  #If mu<0, make it small number
  #And nonzero
  la~dnorm(0, 1/10) #has to be positive exp(1)?
  b.l~dexp(0.5) #Has to be positive, because function has neg sign
  c~dexp(0.5) #Has to be positive
  shape~dexp(0.001)
  #Likelihood
  for (i in 1:N.obs){
    trait[i] ~ dgen.gamma(1, 1/lambda[i], shape) #
    lambda[i] <- (log(2))^(1/shape)/(med[i])
    med[i] <- exp((exp(la) * temp[i]^2 - b.l * temp[i] + c))
  }
```

```
}",file="weibull.model.base.txt")
```

However, for simplicity, we decided to model the median lifetime directly rather than the inverse of the median. This involved making our quadratic function to be a representation of the median, $\frac{1}{exp(ax^2-bx+c)}$, so that it is now concave down. `my.lambda` is the concave down function that represents the scale parameter, λ. This is used to simulate the data from a Weibull distribution. $\frac{1}{exp(ax^2-bx+c)}$ or $exp(-(ax^2-bx+c))$ is the median lifetime since it is concave down. We then model lambda as $\frac{M(T)}{ln2^{1/k}}$. This is the scale parameter to generate data. The original `f.exp` function represents the inverse of the median because it is concave up. If it's too hot or too cold, organisms will not live as long, so the representation of the median is based on the shape and orientation of the curve. In the exponential distribution, the parameter that we use is the rate, which is $\frac{1}{time}$, whereas the parameter that we want here is just *time*.

```
# Making quadratic function
my.med<-function(x,a=0.00826,b=0.37,c=1.406){
  exp(-(a*x^2-b*x+c))
}
#Because it's concave up, need to flip ito be concave down
# That function is 1/median
# TO make median function, do 1/e^-()
# Set reasonable shape parameter
sh.est<-3
my.lambda<-function(x,a=0.00826,b=0.37,c=1.406,sh=3){
  my.med(x,a,b,c)/(log(2))^(1/sh.est) ## my.med is the median here
}
# Generate data from 5 temperatures and create data frame
T1W<-rweibull(n=100,scale=my.lambda(10),shape=sh.est)
T2W<-rweibull(n=100,scale=my.lambda(17),shape=sh.est)
T3W<-rweibull(n=100,scale=my.lambda(21),shape=sh.est)
T4W<-rweibull(n=100,scale=my.lambda(25),shape=sh.est)
T5W<-rweibull(n=100,scale=my.lambda(32),shape=sh.est)
dfW<-data.frame(
  T=c(rep(10,100),rep(17,100),rep(21,100),rep(25,100),rep(32,100)),
  trait=c(T1W,T2W,T3W,T4W,T5W)
)
```

Since we are using λ to generate the data, this means that the scale parameter in the `JAGS` model needs to be the inverse of that. The median and λ are the same.

```
sink("weibull.model.base.txt")
cat("model{
  #Priors
  #If mu<0, make it small number
  #And nonzero
  la~dnorm(0, 1/10) #has to be positive exp(1)?
  b.l~dexp(0.5) #Has to be positive, because function has neg sign
  c~dexp(0.5) #Has to be positive
  shape~dexp(0.001)
  #Likelihood
  for (i in 1:N.obs){
    trait[i] ~ dgen.gamma(1, 1/lambda[i], shape) #
    lambda[i] <- (med[i])/(log(2))^(1/shape)
    med[i] <- exp(-(exp(la) * temp[i]^2 - b.l * temp[i] + c))
  }
}",file="weibull.model.base.txt")
```