Code tutorial: applying TGP modeling to telemetry data

This pdf walks through code for fitting Bayesian treed Gaussian process models to animal telemetry data and obtaining derived quantities describing movement. This tutorial works through application to one individual animal, and must be repeated for multiple individuals or separate treatments.

As a demonstrative example, we use 100 consecutive GPS points from one lesser prairie-chicken in our dataset. After using this example to work through the code we hope that you will replace it with your own data and research questions/derived quantities. Data for these 100 GPS data points, as well as code that can be copied to follow this pdf tutorial, can be found in the accompanying .R document.

We rely on Robert Gramacy's tgp package (version 2.4) to fit movement models.

```
#install.package("tgp")
library(tgp)
```

Step 1) Read in data

The dataframe we use requires **X, Y,** and **t.obs columns**, which may require some manipulation of your GPS data. Data should already be cleaned and checked for location error outliers, as described in Appendix B.

You also must choose to model in lat-long or UTM. If modelling in lat-long, you will have to convert to UTM to perform derived quantity computations such as distance between points. However, UTM may pose problems for large datasets that span multiple zones.
The lesser prairie-chicken dataset utilizes UTM.

Pre-formatted example toy lesser prairie-chicken data can be read in the .Rmd document:

```
individual<- cbind(timestamp,individual)
names(individual) <- c("timestamp","X", "Y")
head(individual)
```

```
##             timestamp        X       Y
## 1 2018-04-15 00:00:00 495544.2 4137085
## 2 2018-04-15 02:00:00 495544.2 4137066
## 3 2018-04-15 04:00:00 495529.2 4137066
## 4 2018-04-15 06:00:00 495544.2 4137066
## 5 2018-04-15 08:00:00 495514.1 4137085
## 6 2018-04-15 10:00:00 495514.2 4137122
```

Add t.obs column:

t.obs provides a standardized time measure for the model to intake. Units will depend on your time scale, but counting in hours is common. Hour "0" can relate to any time within or outside the dataset, and won't affect the modeling, as you'll transform back to the natural time scale for your results.

We used t.obs = hours since first recorded GPS point.

```
individual$t.obs <- (as.numeric(individual$timestamp- min(individual$timestamp)))/3600
```

Step 2) Fit tgp Model

Fit the model to the x and y data separately:

Bayesian treed Gaussian process with jumps to the limiting linear model (within the tgp package) Note that for large datasets and without any parallelization this may be a time consuming step.

```
model_x <- btgpllm(X=individual$t.obs, Z= individual$X ,bprior="b0", verb=0, pred.n = F)
model_y <- btgpllm(X=individual$t.obs, Z= individual$Y ,bprior="b0", verb=0, pred.n = F)
```

Step 3) Predict paths

Set MCMC hyperparameters:

Prediction under the treed Gaussian Process requires MCMC sampling. In the tgp package this requires the tuning parameter of BTE = c(burn in, total samples, thinning value). Concepts for this are explained in Appendix B, and we also recommend Hobbs and Hooten (2015) or Hooten and Hefley (2019) for explanations of these concepts.

We recommend prototyping your data with small BTE settings first, for example `c(400,500,2)`. This translates to an MCMC sample size of 50. Once you have prototyped and code is running without error, increase the BTE and MCMC sample size. `bte <- c(2000, 12000, 10)` is a good starting place for an appropriately large sample size, but this selection is discussed further in Appendix B. Options to evaluate the MCMC sample are given at the end of this section.

```
bte <- c(2000,12000,10) #EDIT this line to prototype
MCMC_sample_size <- (bte[2] - bte[1])/bte[3] #do not edit this line
```

Set Δt

Choice of Δt is described in depth in Appendix B. A Δt of 1 hour is reasonable for many applications.

```
delta.t <-  1 # 1 hour. This may be edited.
```

Use fitted model to predict:

This chunk does not require editing. It will sample from the posterior predictive distribution of locations at each Δt time point.

Note that for large datasets and without any parallelization this may be a time consuming step and should be prototyped on a small dataset and small BTE first.

```
season.end <- max(individual$t.obs)
XX <- seq(0, season.end, by = delta.t) #points we want prediction at

#### build storage table
all_paths <-  data.frame(matrix(ncol = 1+ 2*MCMC_sample_size, nrow = length(XX)) )
names(all_paths)[1] <- "t"
all_paths$t <- XX
seq.x <- seq(2,ncol(all_paths), by = 2) #sequence of even numbers
names(all_paths)[seq.x] <- paste0('x', 1:MCMC_sample_size)
seq.y <- seq(3,ncol(all_paths), by = 2) #sequence of odd numbers not 1
names(all_paths)[seq.y] <- paste0('y', 1:MCMC_sample_size)

#### make your transformation constants from the data once
z.x <-individual$X
```

```
z.x.ranged <- -0.5 +(z.x - min(z.x))/(max(z.x) - min(z.x))
z.y <-individual$Y
z.y.ranged <- -0.5 +(z.y - min(z.y))/(max(z.y) - min(z.y))

#### compute predictions
predicted_x <- predict(model_x, XX, pred.n=FALSE,  BTE=bte,  MAP=FALSE, trace=T)
posterior_predictives_x_ <- predicted_x$trace$preds$ZZ
predicted_y <-  predict(model_y, XX, pred.n=FALSE, BTE=bte, MAP=FALSE, trace=T)
posterior_predictives_y <- predicted_y$trace$preds$ZZ
#Setting MAP = FALSE when making prediction means that the MCMC sampling will start at
#the MAP (maximum a posteriori probability) tree, but then continue in Bayesian form
#and sample across the full posterior distribution of trees. This means different MCMC
#samples will often be from different possible trees (different partitioning solutions).

# do their transformations:
posterior_predictives_x <- t(posterior_predictives_x_ + mean(z.x.ranged) + 0.5)*
  (max(z.x) - min(z.x)) + min(z.x);
posterior_predictives_y <- t(posterior_predictives_y + mean(z.y.ranged) + 0.5)*
  (max(z.y) - min(z.y)) + min(z.y);
# take transpose of matrix so it's set up with times are rows and samples are columns
#instead of vice versa

### seperate these into the table
all_paths[,seq.x] <- posterior_predictives_x ; #fill in all the xs
all_paths[,seq.y] <- posterior_predictives_y #fill in all the ys
```

After all_paths is outputted, this would be a good time to export it and save it on your device if your dataset is large.

The full set of MCMC samples obtained from the trajectory's distribution is large and not insignificant for storage, with a size of time period/Δt * MCMC sample size. Though it may be tempting to elimininate this storage by predicting the paths and computing the derived quantities simultaneously, saving the full MCMC sampled distribution allows the flexibility to change scale or derived quantities as research questions change or are added. Because predicting these full trajectories is the most time consuming step, we recommend saving the full predicted trajectory sample outside of R for future use.

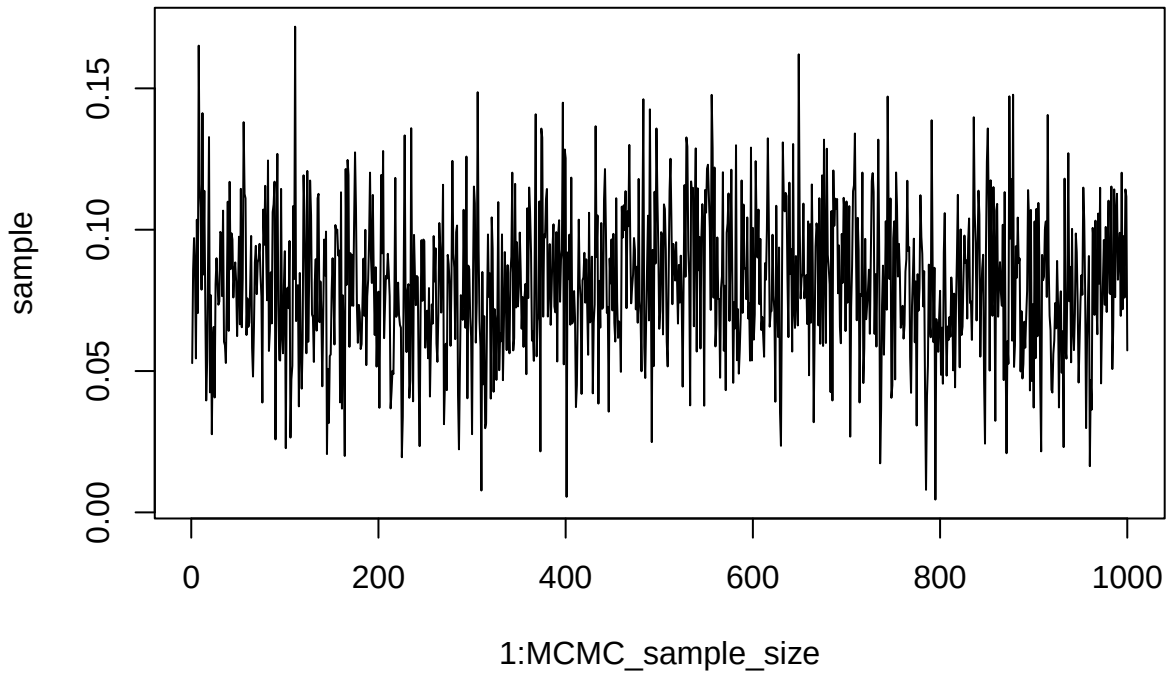
If you wish to check the effective sample size of your MCMC samples, run the following code after you've sampled from the posterior predictive distribution:

```
#install.packages("coda")
library(coda)
sample <- as.numeric(posterior_predictives_x_[,1])
effectiveSize(sample)
```

```
##     var1
## 852.5559
```

If you wish to examine a trace plot of your MCMC samples, run the following after you've sampled from the posterior predictive distribution:

```
plot(1:MCMC_sample_size, sample,type="l")
```



Step 4) Compute derived quantity at Δt scale

We obtain derived quantities from our MCMC sample of trajectories, which is a large table of dimensions: T x (MCMC sample size x 2). Where T = (end time - start time)/delta_t

```
dim(all_paths)
```

```
## [1]  229 2001
```

```
all_paths[1:5,1:5]
```

```
##   t       x1      y1       x2      y2
## 1 0 495489.6 4137246 495539.1 4136904
## 2 1 495519.1 4136873 495530.6 4137068
## 3 2 495526.8 4137041 495529.8 4137081
## 4 3 495529.3 4137113 495592.9 4137139
## 5 4 495488.5 4137119 495477.6 4137275
```

First we compute the derived quantity at the scale of Δt (e.g., hourly). This is the scale the above table is already at, and then allows us to then transform to any larger desired scale (e.g., daily, weekly).

It may be helpful to follow along with the displacement example in Appendix C as well, and we've noted how each code step aligns with each step in Appendix C.

Your derived quantity:

You will have to edit this section depending on your derived quantity. As the options for derived quantities are infinite, this may take some creativity. Refer to Table 1 for basic derived quantity functions that can be built off of.

You will have to write your derived quantity function here:

A displacement function is provided as an example.

```
my_DQ_fun <- function(i){
  #a function of one delta t instant, represented by i (the row in the dataframe)
  return(sqrt((all_paths[i,colStart ] - all_paths[i+1, colStart])^2  +
                (all_paths[i, colStart+1 ] - all_paths[i+1,colStart+1])^2  ))
}
# this is the distance formula calling (x1,y1) and (x2,y2) from the all_displacements data frame,
#indexed at time i
```

The following step produces a table of dimension (time points - 1) x (MCMC samples)

[arrow 1 in Appendix C]

AKA it takes a sample from the distribution of the derived quantity at each time point.

```
all_DQs <-  data.frame(matrix(ncol = 1+ MCMC_sample_size, nrow = length(XX)-1) )
names(all_DQs)[1] <- "t"
all_DQs$t <- XX[-1]
names(all_DQs)[-1] <- paste0('d', 1:MCMC_sample_size)

for(j in 1:(ncol(all_DQs)-1) ){  #loop through all the MCMC samples
  colStart <- 2*j
  #one sample of DQs:
  for(i in 1:nrow(all_DQs)){  # loop through all the times (fill out one column)
    DQ <- my_DQ_fun(i)
    all_DQs[i, j+1] <- DQ
  }
}
dim(all_DQs)
```

```
## [1]  228 1001
```

```
all_DQs[1:5,1:5]
```

```
##   t        d1        d2        d3       d4
## 1 1 374.20260 164.64964 148.46503 122.8898
## 2 2 167.91838  13.17845  75.80742 283.8824
## 3 3  72.01423  85.40967 193.79285 118.9141
## 4 4  41.28666 178.37181 124.79405 142.0678
## 5 5 190.85457 318.48758 205.04023 126.1891
```

We now have a data frame of samples of the derived quantity at each Δt point, with d1-d1000 indexing the 1000 samples.

Step 5) Output inference at desired scale

The above 252,000 samples of different hourly displacements is not very interpretable or useful, so we must transform it into our desired value: average daily total distance traveled over the study period.

Though it may be tempting to just average all values in the table, we must follow the basic MCMC principles. See Appendix 3 for further walking through of this example.

These steps will vary based on your transformations and desired ending derived quantity

In our example, if we want to transform to **average daily total distance traveled for the season**, we first take daily sums.

[arrow 2 in Appendix C]

(we used the dplyr and lubridate packages to make this easier)

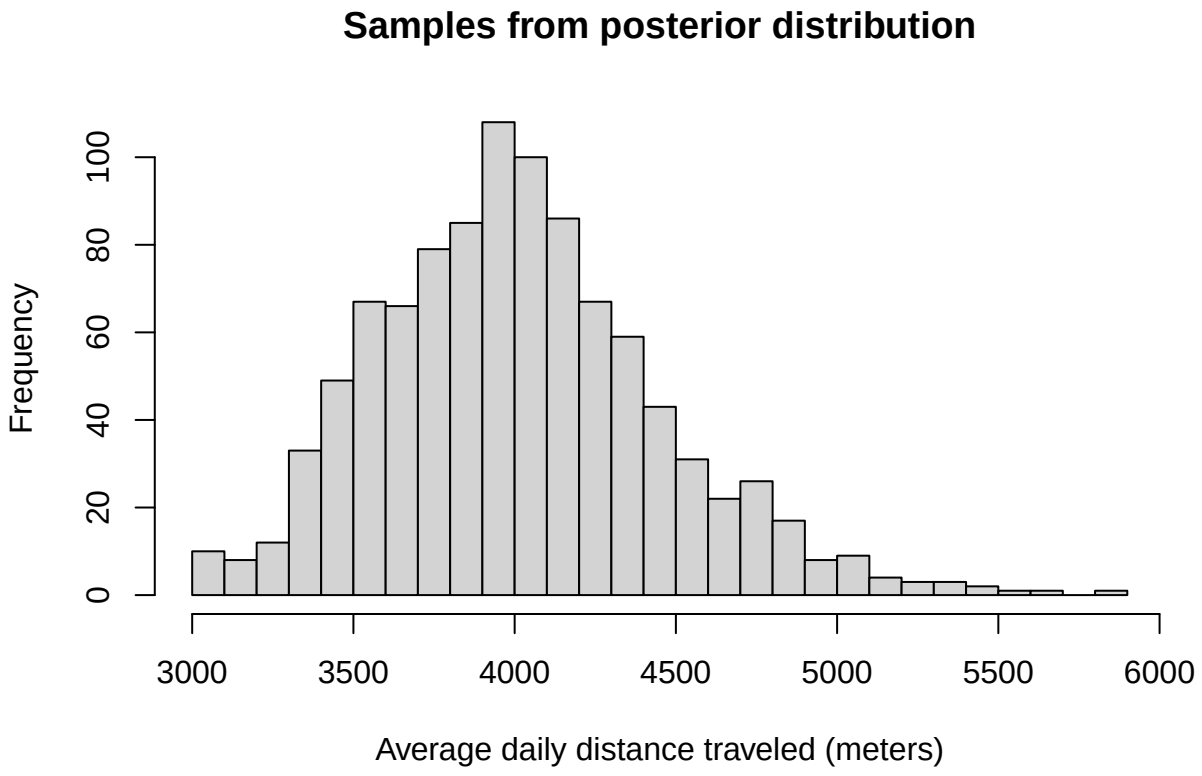
```r
#All derived quantities values are in t.obs, and we need to transform them back to natural
#time to get dailies:

library(lubridate) #Installing the package lubridate is useful for this.

# we will use the original timestamps now so that we can get dailies:
time_start <- min(individual$timestamp) + 60*60  #adding one hour since this was hour 0
#and displacement starts at hour 1
time_end <- time_start  + (max(all_DQs$t)-1)*60*60  #adding all the hours in the DQ
#predictions to get end time

time <- seq(time_start , time_end  , by = 'hours')

daily <- cbind(time, all_DQs)
day <- floor_date(daily$time, "day")
daily <- cbind(day, all_DQs)

library(dplyr)
daily_DQs <- daily %>%
  group_by(day) %>%
  summarise(across(d1:d1000, sum))
```

This gives a distribution (1000 samples) of daily total distance traveled.

To get 1000 samples of *average* daily distance traveled, we take column averages.

[arrow 3 in Appendix C]

```r
average_daily_DQs <- as.vector(colMeans(daily_DQs[,-1]))
hist(average_daily_DQs, breaks = 30, xlab = "Average daily distance traveled (meters)",
     main = "Samples from posterior distribution")
```



We can summarize this sample of average daily distance traveled using the mean and 95% credible region:

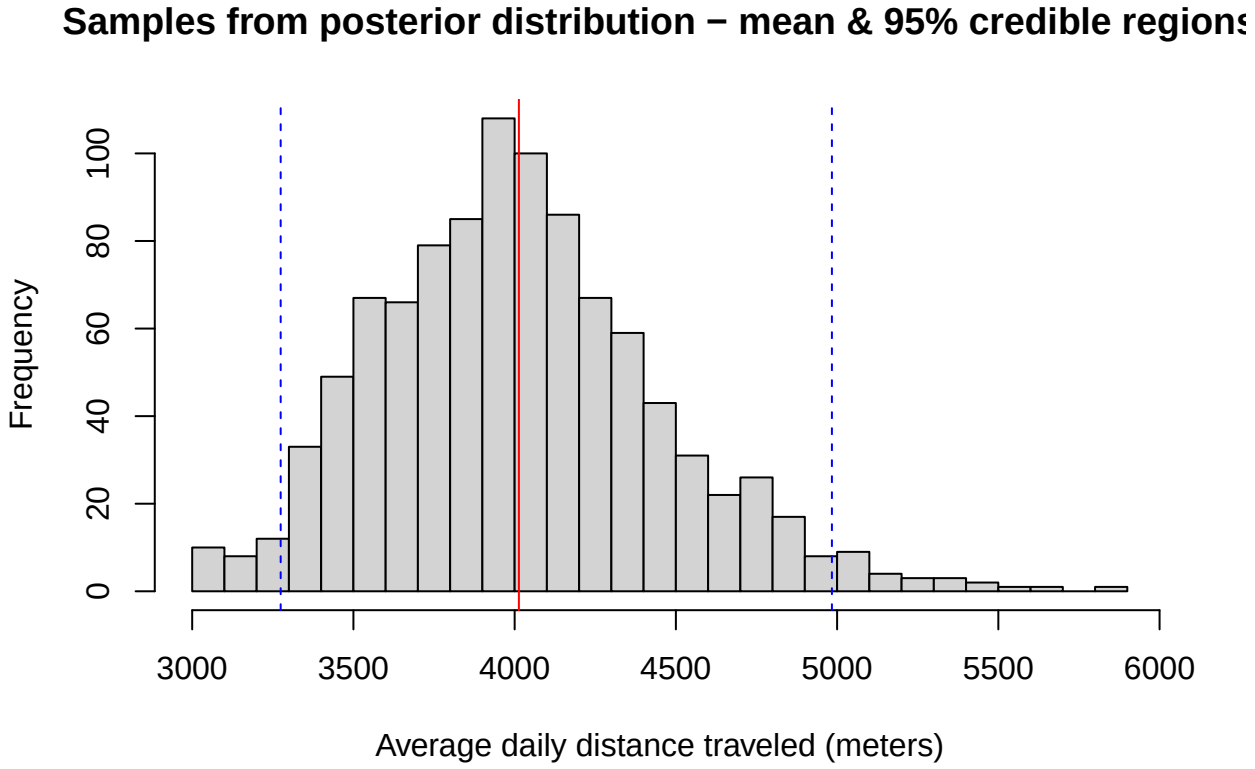
```
mean(average_daily_DQs)
```

```
## [1] 4013.151
```

```
c(quantile(average_daily_DQs, 0.025), quantile(average_daily_DQs, 0.975))
```

```
##     2.5%    97.5%
## 3274.443 4983.902
```

```
hist(average_daily_DQs, breaks = 30, xlab = "Average daily distance traveled (meters)",
     main = "Samples from posterior distribution - mean & 95% credible regions")
abline(v = c( mean(average_daily_DQs), quantile(average_daily_DQs, 0.025),
              quantile(average_daily_DQs, 0.975)), col = c("red", "blue", "blue"), lty=c(1,2,2) )
```



For inference at a different scale, the above chunks must be edited.

Step 6) Repeat for individuals and study periods

We leave it to the practitioner to repeat this across individuals and chosen study periods, and to store the all_paths tables, and their desired derived quantity point estimate and variance estimates.

Speed up options

There are multiple options to increase run speeds.

Option A uses simple parallelization to fit both X and Y models simultaneously on your machine by using separate cores simultaneously. This can expect a modest ~50% speed up for only the model fitting step.

Option B offers the largest speed up (up to 10x), but requires slight reconfiguration of the computer and will depend slightly on the computer, making it more complicated.

Option C offers 2x or 3x speed up and is the most complicated to install, and requires reconfiguring the tgp package.

If speed is an important goal, combining option B and option C will have the best outcome.

Option A)

Simple parallelization of fitting the X and Y models. (This simple parallelization cannot be applied to prediction.)

```
library(parallel)
```

Replace fitModels chunk with:

```
numCores <- detectCores() #the number of cores on your computer

#rewrite the tgp function as a function of either X or Y:
fit <- function(z) { btgpllm(X=individual$t.obs, Z= z,  bprior="b0", verb=0, pred.n = F) }
x_and_y <- list(individual$X, individual$Y)

## split into cores:
models <- mclapply(x_and_y, fit, mc.cores = numCores) #runs x and y fitting separately

names(models) <- c("X", "Y")
models$X #lets you grab the x_model
## In predict chunk, replace model_x with models$X and replace model_y with models$y
```

Option B)

Option B is to use a more efficient matrix algebra library within R. This will greatly speed up computations done within both stages of tgp. These changes must be made on the computer outside of R, but once this change is made then tgp will automatically run faster in R every time, without any changes to code.

This can be done in the Accelerate framework on a mac, following installation instructions provided by Berkley statistics.

For windows, Intel's Math Kernel Library (MKL) can be used.

More details on Accelerate and MKL can be found in Robert Gramacy's Surrogates (2020).

Option C)

Option C requires uninstalling and reinstalling tgp with Pthreads. This will make tgp prediction automatically utilize parallelization. Details for this and reinstillation can be found in the tgp documentation appendix C.2.

References

Gramacy, R. B. (2007). tgp: an R package for Bayesian nonstationary, semiparametric nonlinear regression and design by treed Gaussian process models. Journal of Statistical Software, 19, 1-46. pdf

Gramacy, R. B., & Lee, H. K. H. (2008). Bayesian treed Gaussian process models with an application to computer modeling. Journal of the American Statistical Association, 103(483), 1119-1130. pdf

Gramacy, R. B. (2020). Surrogates: Gaussian process modeling, design, and optimization for the applied sciences. Chapman and Hall/CRC. book

Hooten, M. B., & Hefley, T. J. (2019). Bringing Bayesian models to life. CRC Press.

Hobbs, N. T., & Hooten, M. B. (2015). Bayesian models. Princeton University Press.