TrackReconstruction R package Vignette

Brian Battaile

November 13, 2024

1 Introduction

Dealing with the quantities of data from tags that collect magnetometer and accelerometer data is perhaps the greatest difficulty involved in track reconstruction. The raw data files from the loggers are large and as human readable text are often too large to import into free text editors let alone then manipulate. You will want a computer with large amounts of memory and 64bit R. The more memory you have, the larger the data files you can process without having to split them up. There are two tasks that may require data subsetting, the first is eliminating periods outside the times that you want to do the track reconstruction, such as when a seal is on land. The second is using chunks of data that are small enough for the TrackReconstruction package to handle. I have written R code to deal with both of these tasks, but depending on the length of time data are collected and the sampling rate, and your computers abilities, data sets may be too large for R to import or then manipulate for either of these tasks. In case you are unable to import your data into R and remove unwanted sections or subset it, I wrote some PERL code that I used for both of these tasks that can handle most any sized data set efficiently. I have put the PERL code in the last section of this document as an Appendix.

My desktop is 64bit with 16GB of RAM. The data I use is triaxial magnetometer and accelerometer and DateTime sampled at 16Hz with included depth information. R 3.0.2 was able to import a 3GB raw data file (46 million lines, about 1 month) for trimming on my machine and I was successfully able to process a 1.2GB file that consists of 18 million lines (a 13 day trip) using the TrackReconstruction package in a single chunk. I have however also successfully reconstructed elephant seal trips that last over 3 months but with considerable subsetting with PERL. I also recommend a solid state drive large enough to handle the many data files, as reading and writing many files of this size can be time consuming.

For better or for worse, I work on a Windows machine, I'm sure most of the concepts here are adaptable to Mac and Linux machines, but the coding and interfaces are different enough to make this more of a guide as opposed to a copy and paste vignette when it comes to DOS and R file management code.

1.1 Date and Time in R and TrackReconstruction

Dealing with date and time data in R has been one of THE most frustrating things I have ever done. One of the main issues is the many formats date and time data come in. The functions in TrackReconstruction compare the date and time data (merged as a DateTime column) to split data sets and georeference the tracks to force them to go through GPS points. I have chosen to do this using the DateTime as a string of characters and matching as opposed to using the DateTime as an actual date and time. Perhaps this is the wrong way to do things but it's how I have done them, I'm open to suggestion. What this means is that the date and time files from different tags must be in the same format by the time you use the data in TrackReconstruction functions. Here is some R code I recommend.

This is something I put at the top of most of my R scripts, when R imports data, it defaults to importing data that is not numeric as factors. This imports data as a character string. Really, the only time I ever want data as a factor is when I'm using it to build statistical models.

```
options(StringsAsFactors=FALSE)
```

If you don't want this as a global change to your options, you can always put the *StringsAsFactors=False* as an option in your *read.table* for importing the data into R.

The following code should start you off on changing the format of date and time data if it is not already the same.

```
DT<-paste(as.character(TagFile$Date),as.character(TagFile$Time))
DateTime<-strptime(DT,format="%Y-%m-%d %H:%M:%S") #default code for format argument
DateTime<-as.character(DateTime)
```

You shouldn't need the *as.character* function if you used the *StringsAsFactor=FALSE*, but in my experience.....it never hurts.

Spend some time with the *strptime* help page as that will help you translate the code "

```
options(digits.secs=4)
DT=c("Jul/12/2009 05:45:35.0625","Jul/12/2009 05:45:35.1250","Jul/12/2009 05:45:35.1875")
DateTime1<-strptime(DT,format="%b/%d/%Y %H:%M:%S")
DateTime1
DateTime2<-strptime(DT,format="%b/%d/%Y %H:%M:%OS")
DateTime2
```

1.2 Subsetting the data

The first step is to determine when you want to begin and end the track reconstruction. For my marine animals it began and ended when the animal entered and exited the water, it is important to have accurate location (GPS) information for these times as well. These two times and locations bookend the analysis. The other times that most people will hopefully have are times when known locations (GPS fixes) were taken between the begin and end times to periodically georeference the track. Why this is important will be discussed later.

Prior to running this first bit of code, you should understand the principles laid out in the following article

Shepard E.L.C., Wilson, R.P., Halsey, L.G., Quintana, F., Laich, A.G., Gleiss, A.C., Liebsch, N., Myers, A.E., Norman, B. (2008) Derivation of body motion via appropriate smoothing of acceleration data. Aquatic Biology 4:235-241

as it describes how to determine the Running Mean Length (*RmL*) variable. While *RmL* is part of the *DeadReckoning* function code, you need to understand what it does now because this tells us how much data we need to keep before the beginning and after the end of the dataset so the running mean function can calculate a number starting with the first data point when the animal enters the water and the last data point before the animal leaves the water. Here, it is set to the sampling frequency, and that may be a place to start but it also may be inappropriate for your animal. This same consideration must be made for any subsetting of the data prior to applying the *DeadReckoning* function, for example, if subsetting data between GPS locations.

This first set of R code takes a raw text data file and subsets it by begin and end times, which can be a trip or a portion of a trip such as between GPS locations or however you want to split it. There is a function in the TrackReconstruction package called *Splitter* that does this. This example uses a data set from a Wildlife Computers Daily Diary tag. Their instrument helper program exports data ready for importation into R. Simply export the entire data file with the data channels of interest. This code will write subsetted data files to the working directory. The *Trip* and *Section* arguments are for naming the files.

```
options(digits=15,digits.secs=4) #To visualize the fractions of seconds %H:%M:%OS3
setwd("G:\\Bog 2009 Toughbook\\2009 Data\\Daily Diary\\Bogoslof\\Cu09BG03")
FullFile=read.table("09A0572.tab", header=T)
colnames(FullFile)=c("DateTime","Depth","MagSurge","MagSway","MagHeave","AccSurge",
"AccSway","AccHeave","Wet_Dry")
#Times when the animal entered and exited the water
Begin<-as.POSIXct(c("2009-07-16 18:26:14","2009-07-23 19:08:24","2009-07-29 22:53:43"
,"2009-08-06 04:26:13"),tz="GMT")
End<-as.POSIXct(c("2009-07-22 06:21:11","2009-07-28 16:15:49","2009-08-04 14:36:26",
```

```
"2009-08-11 01:22:16"),tz="GMT")
RmL=2
Splitter(FullFile, Begin, End, RmL,Hz, Animal01,Trip="Trip")
```

It is important to have the DateTime in the same format for the biologged files and the Begin and End files. If the time and date are in different columns or the formats are different you may need to add some code that looks like this. Or just format the Begin and End files to match the DateTime on the FullFile, whichever is easiest.

```
DT<-paste(as.character(FullFile$Date),as.character(FullFile$Time))
tt<-strptime(DT,format="%Y/%m/%d %H:%M:%S")
DateTime<-as.POSIXct(tt,tz="GMT")
```

1.3 File management in R

This subsection is really designed for those instances where you end up with many data files, this could be subsets of single animal tracks or if you have many animals. For those of you that can process your entire tracks without breaking them up and don't have many animals, or just want to do a test run, you can probably just skip or skim this section. However, if you have more than a handful of files to deal with, this short section may be worth your while.

HELPFUL HINT–it is best to number files with 001.csv instead of just 1.csv as R will order things correctly if you have leading 0's in your file numbers. e.g., if you have 100's of segments, 001.csv, if you have 10's of segments, 01.csv will be fine. You should label all folders this way as well. Here is a snippet that I use to name trip section data tables created in R.

```
num=ifelse(num<10,paste("0",num,sep=""),num)
num=ifelse(num<100,paste("0",num,sep=""),num)
num=ifelse(num<1000,paste("0",num,sep=""),num) # etc.
filename=paste("xyz-",num,".csv",sep="")
write.table(xyz, file = filename, sep="\t",row.names=FALSE)
```

That should get you files that can now be used in TrackReconstruction. But now that you have ALL these data files, you need some R code that automates this for you so you can go home at night while the computer works for you. The following is an example of setting up and importing all the data files to do the pseudotrack calculation using the *DeadReckoning* function in TrackReconstruction.

This is an example of the path to one of my raw between-GPS-points files

```
C:\filepath\Bogoslof\Cu09BG01\trip 1\01-232.csv
```

Animals in separate folders, the code is Callorhinus Ursinus 2009 BoGoslof animal 01

```
C:\filepath\Bogoslof\Cu09BG01
```

Animal trips are in separate folders

```
C:\filepath\Bogoslof\Cu09BG01\trip 1
```

and between GPS segments are separate files

```
C:\filepath\Bogoslof\Cu09BG01\trip 1\01-232.csv
```

This first line of code uses regex wildcards to find all possible file paths corresponding to all the trips taken by all my animals. Fill in the "filepath" with the rest of your filepath to your data.

```
dirpath=Sys.glob(file.path("C:\\filepath\\Bogoslof","*","trip*"))
```

If I had 10 animals that each went on 2 trips this would result in a vector of 20 file paths that can be accessed individually with a loop through dirpath[j] where j=1-20. A similar structure will get us the path to any data that we need, such as the coefficients for normalizing the accelerometer and magnetometer sensors for each tag, which I named "betas" (this file is also made with the *Standardize* function which is described later).

```
dirpathbetas=Sys.glob(file.path("C:\\filepath\\Bogoslof","*","trip*","betas*"))
```

A similar structure will get us the path to the declination and inclination data for each trip.

```
dirpathdecinc=Sys.glob(file.path("C:\\filepath\\Bogoslof","*","trip*","decinc*"))
```

Looping through each of these.

```
for(j in 1:length(dirpath))
{
    #load declination, inclination and tag standardization data
  decinc=read.table(dirpathdecinc[j], fill=TRUE, sep="\t", header=TRUE)
  betas=read.table(dirpathbetas[j], fill=TRUE, sep="\t", header=TRUE)
    #We use a similar structure to determine paths for each between-gps segment,
    #for my BG_01_01 animal example, this will be a vector of length 233, this
    #vector will be in numerical order IF you put 0s in front of your numbers.
  dirpath2=Sys.glob(file.path(dirpath[j],"0*-*.csv"))
   setwd(dirpath[j])
   for(i in 1:length(dirpath2))
   {
       #load the file into R, the "sep" indicates how your columns are delimited
       #\t is regex for tab, use sep="," for comma seperated files.
     rawdata=read.table(dirpath2[i], fill=TRUE, sep="\t", header=TRUE)
       #save the number of the file
     num=as.numeric(strsplit(strsplit(dirpath2[i],split="-")[[1]][2],split=".csv")
            [[1]][1])
       #If you did not originally take my helpful advice, this code will add leading 0s
     num=ifelse(num<10,paste("0",num,sep=""),num);num=ifelse(num<100,paste
            ("0",num,sep=""),num)
       # note that RmL units are seconds in the DeadReckoning function
     xyz<-DeadReckoning(rawdata, betas, decinc,Hz=16, RmL=2, DepthHz=1, SpdCalc=1,
            MaxSpd=NULL)
       #Create a file name
     filename=paste("xyz-",num,".csv",sep="")
     filename2=paste("xyz-",num,".RData",sep="")
       #Write the file in .csv or .txt
     write.table(xyz, file = filename, sep="\t",row.names=FALSE)
       #Write the file in *.Rdata, this is good for compressed files but you will
       #not be able to view as human readable in a text editor, only save one of
       #*.csv or *.RData, no point in saving both.
     #save(xyz,file=filename2) # saves in *.Rdata format
   }
 print(j) #This just lets me know where the program is
 }
```

So, without all the comments it looks like this

```
dirpath=Sys.glob(file.path("C:\\filepath\\Bogoslof","*","GPS segments","trip*"))
dirpathbetas=Sys.glob(file.path("C:\\filepath\\Bogoslof","*","GPS segments","trip*","betas*"))
dirpathdecinc=Sys.glob(file.path("C:\\filepath\\Bogoslof","*","GPS segments","trip*","decinc*"))
for(j in 1:length(dirpath))
{
  decinc=read.table(dirpathdecinc[j], fill=TRUE, sep="\t", header=TRUE)
  betas=read.table(dirpathbetas[j], fill=TRUE, sep="\t", header=TRUE)
  dirpath2=Sys.glob(file.path(dirpath[j],"0*-*.csv"))
  setwd(dirpath[j])
  for(i in 1:length(dirpath2))
  {
```

```
    rawdata=read.table(dirpath2[i], fill=TRUE, sep="\t", header=TRUE)
    num=as.numeric(strsplit(strsplit(dirpath2[i],split="-")[[1]][2],split=".csv")[[1]][1])
    num=ifelse(num<10,paste("0",num,sep=""),num);num=ifelse(num<100,paste("0",num,sep=""),num)
    xyz<-DeadReckoningR(rawdata, betas, decinc,Hz=16, RmL=2, DepthHz=1, SpdCalc=1, MaxSpd=NULL)
    filename=paste("xyz-",num,".csv",sep="")
    #filename2=paste("xyz-",num,".RData",sep="")
    write.table(xyz, file = filename, sep="\t",row.names=FALSE)
    #save(xyz,file=filename2)
  }
print(j)
}
```

I recommend setting up something like that for the *DeadReckoning* and *GeoReference* and possibly the *Standardize* and *GPStable* functions as well.

If you want to reconstitute your track into a single file at either the deadreckoning or georeferencing phase, insert code that looks something like this where the write.table line (or save) is in the above code. The *DeadReckoning* function trims off the extra bits of data from each end that was required to calculate the running means so they *SHOULD* match together nicely

```
if(i==1)
{
        write.table(xyz, file = filename, sep="\t",row.names=FALSE)
        write.table(xyz, file = "xyzmaster.csv", sep="\t",row.names=FALSE)
}else{
        write.table(xyz, file = filename, sep="\t",row.names=FALSE)
        write.table(xyz, file = "xyzmaster.csv", sep="\t",row.names=FALSE,col.names=F,append=T)
}
```

1.4 Thinning the data

If you want to reduce (thin) the size of your master file this can be trivial and it can also be difficult. If you have missing data, things become more difficult, if you don't have missing data, the following R code works well.

The following is some R code to reduce files by every xth number of rows, the "by=x" controls this, so in this case I am taking every 8th row of data.

```
dataout<-datain[seq(1,nrow(datain),by=8),]
```

You could incorporate this into the " batch" files using *DeadReckoning* or *GeoReference* to save smaller files.

2 Using the TrackReconstruction package

2.1 Standardize function

So what we have done so far is get the data ready for import into the TrackReconstruction package. To start with the TrackReconstruction package, the first thing we need to do is normalize the magnetometer and accelerometer data between -1 and 1 with the *Standardize* function. The first part of this is to determine the orientation of your sensors and they should follow the right hand rule. The right hand rule is as follows, hold your hand in front of you and point your thumb straight up. Next point your index finger away from you and finally point your middle finger to the left so that you now have your fingers as vectors pointing at right angles to one another. These fingers indicate the top, front and left side of the tags. When these sides of the tags are pointing towards the earth, you accelerometers should be at a maximal positive static reading, which is gravity. When they are pointing away from the earth, they should be at their minimal reading. The same goes for the magnetometers but they will be maximal (when in the northern hemisphere) when they

are pointing north and at the angle of inclination of the magnetic field (the angle it enters the earths surface) which is different all around the globe. I am not sure what the readings should be in the southern hemisphere (pointing south and towards the earth at the angel of inclination or towards the north and at 180 degrees opposite to the angel of inclination?). You need to determine what the maximal and minimal readings for the tags are where the animals will be. This can be done by rotating the tags in all three dimensions while facing magnetic north. Optimally this is done with a tool that allows you to stop every 10 degrees or so and hold the tag steady for 5 seconds. This is tedious but will pay off if you need a better model of your sensors than the simple linear model the *Standardize* function provides (such as a logistic). If your animal moves far enough for the angle of inclination to change significantly on their feeding trip, you will want to do multiple standardizations for the magnetometers. You can tell this occurs by looking at your full magnetometer data set and if it gradually gets larger or smaller as the animal moves away from the original place of tagging, and then returns to the same magnitude as the animal returns home. Elephant seals and albatross will likely do this. If your animal tends to move enough so that the tag will rotate around to most positions, you can subset your data into sections where the magnetometer max and min do not noticeably change and simply take the max and min for that section. Do this for each section. These max and min you will put into the Standardization file.

As an example, this is the standardization data required for the tag that took the rawdataXX.Rdata files supplied with the TrackReconstruction package. The first six arguments indicate that all but the sway magnetometer (side or middle finger on the right had rule) on my tag was oriented according to the right hand rule. See the help file for the Standardize function for details on the parameter orientation.

```
> library(TrackReconstruction)
> betas<-Standardize(1,1,-1,1,1,1,-57.8,68.76,-61.8,64.2,-70.16,58.08,-10.1,9.55,
+ -9.75,9.72,-9.91,9.43)
> betas
```

```
               MagSurge    MagHeave     MagSway   AccSurge    AccHeave
B0 Intercept -0.08659924 -0.01904762 -0.09419838 0.02798982 0.001540832
B1 Slope      0.01580278  0.01587302 -0.01559576 0.10178117 0.102722137
               AccSway
B0 Intercept 0.02481903
B1 Slope     0.10341262
```

As you can see, this outputs the slope and intercept of the line that normalizes the millivolt output of the tags to between -1 and +1.

2.2 GapFinder function

Often, you want to see if there are gaps in your data and where they are, if there are significant sized or lots of gaps the track reconstruction becomes less reliable. The GapFinder function finds all the gaps in your data greater than your desired window, though it defaults to 1 second. I recommend again setting up a function that will go through all your files and using some code such as that below to write a file if a gap has been found in that data file.

```
> data(rawdatagap)
> gaps<-GapFinder(rawdatagap, timediff = 1, timeformat = "%d-%b-%Y %H:%M:%S")
> if(gaps[1,1]!=0) write.table(gaps, file = "gaps-1.csv", sep=",",row.names=FALSE)
> head(gaps)
```

```
   Line TimeDiffinSec               Time1               Time2
1 10801         24600 2009-07-19 01:32:23 2009-07-19 08:22:23
2 10802         24563 2009-07-19 08:22:23 2009-07-19 01:33:00
3     0             0                   0                   0
```

So this tells you what line of the data file that gap occurs and how large they are. In this example, there was a 37 second gap and an aberrant time of 8:22:23 was inserted in the gap....tags can and will do strange things.

2.3 GPStable function

The *GPStable* function formats your GPS latitude and longitude data file into something directly importable into the *GeoReference* function and calculates the distance between points, bearing from one point to the next in degrees and radians and converts the latitude and longitude degrees into radians. All navigation functions use radians. The *CalcBearing*, and *CalcDistance* functions are called in the *GPStable* function.

```
> data(gpsdata02)
> gpsformat<-GPStable(gpsdata02)
> head(gpsformat)

              DateTime Latitude Longitude    LatRad   LongRad  BearingRad
1 21-Jul-2009 09:30:00 53.93111 -168.0349 0.9412755 -2.932762  0.09180597
2 22-Jul-2009 01:23:39 53.93306 -168.0346 0.9413094 -2.932757 -0.47828271
3 22-Jul-2009 01:45:09 53.94334 -168.0436 0.9414889 -2.932915 -1.46424471
4 22-Jul-2009 02:07:13 53.94441 -168.0607 0.9415076 -2.933213 -1.36558342
5 22-Jul-2009 02:36:46 53.94956 -168.1028 0.9415975 -2.933948 -1.88773642
6 22-Jul-2009 02:56:31 53.94757 -168.1132 0.9415627 -2.934128 -1.47080487
  BearingDeg DistanceKm
1   5.260095  0.0000000
2 332.596419  0.2170772
3 276.104958  1.2876086
4 281.757833  1.1241930
5 251.840670  2.8147816
6 275.729088  0.7123356
```

2.4 DeadReckoning function

The *DeadReckoning* function is one of two primary functions in the TrackReconstruction package. See the help page for detailed information on the *DeadReckoning* function including the *Hz*, *RmL*, *DepthHz* and *SpdCalc* parameters. In short the function tries to deal with the different sampling rates for the acceleration, magnetometer, depth and speed channel, or if there is no speed channel, the various ways speed may be estimated. In addition to the raw data file, we also need the Declination and Inclination for the study area which can be gotten at the time of writing from here
`http://www.geomag.bgs.ac.uk/data_service/models_compass/wmm_calc.html`

```
> #get declination and inclination data for study area
> decinc<-c(10.228,65.918)
> #example data set with start and end times corresponding to the first and seventh GPS
> #fixes from the gpsdata02 data set, plus additional rows=Hz*RmL*0.5 on each end.
> data(rawdata)
> DRoutput<-DeadReckoning(rawdata, betas, decinc, Hz = 16, RmL = 2, DepthHz = 1, SpdCalc=3,
+ MaxSpd=3.5)
```

and a simple plotting call gives us figure 1

```
plot(DRoutput$Ydim,DRoutput$Xdim)
```

2.5 GeoReference function

The GeoReference function forces the DeadReckoning track to go through the GPS, ARGOS, or other types of supplemental location measures (such a focal follows) that you might have. If you have none, you could simply use the start and finish locations to bound the track. There are only two data sets required for the *GeoReference* function and they were created by the *DeadReckoning* and *GPStable* functions so we can just put in the direct results of those functions.

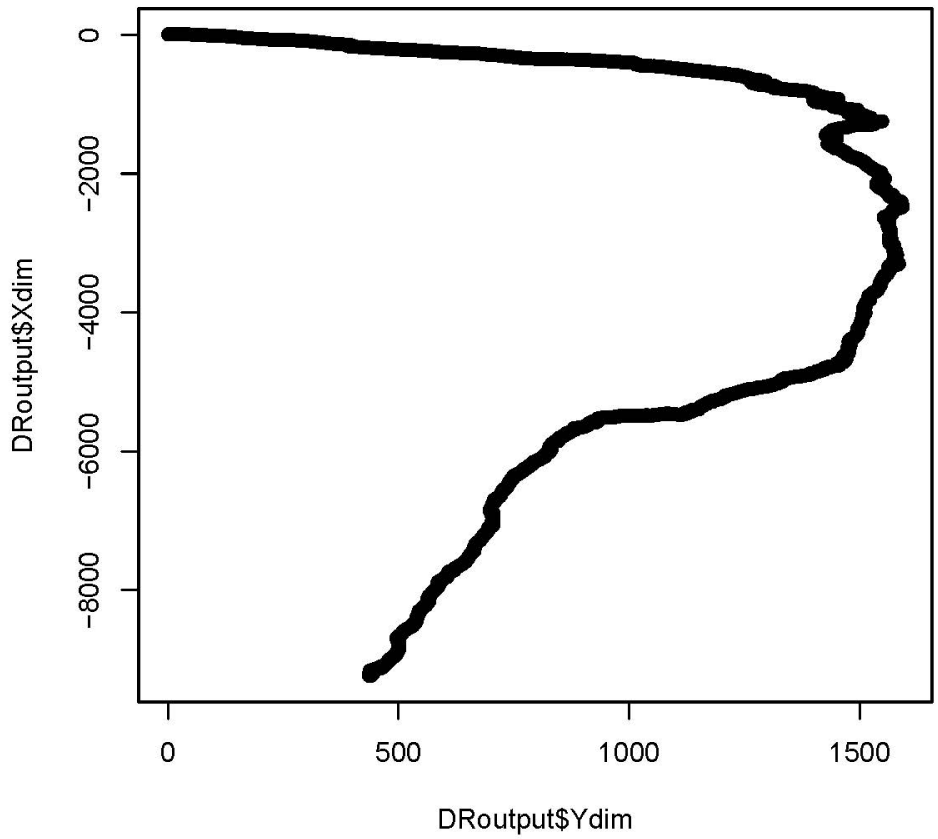

Figure 1: Raw Dead Reckoning plot

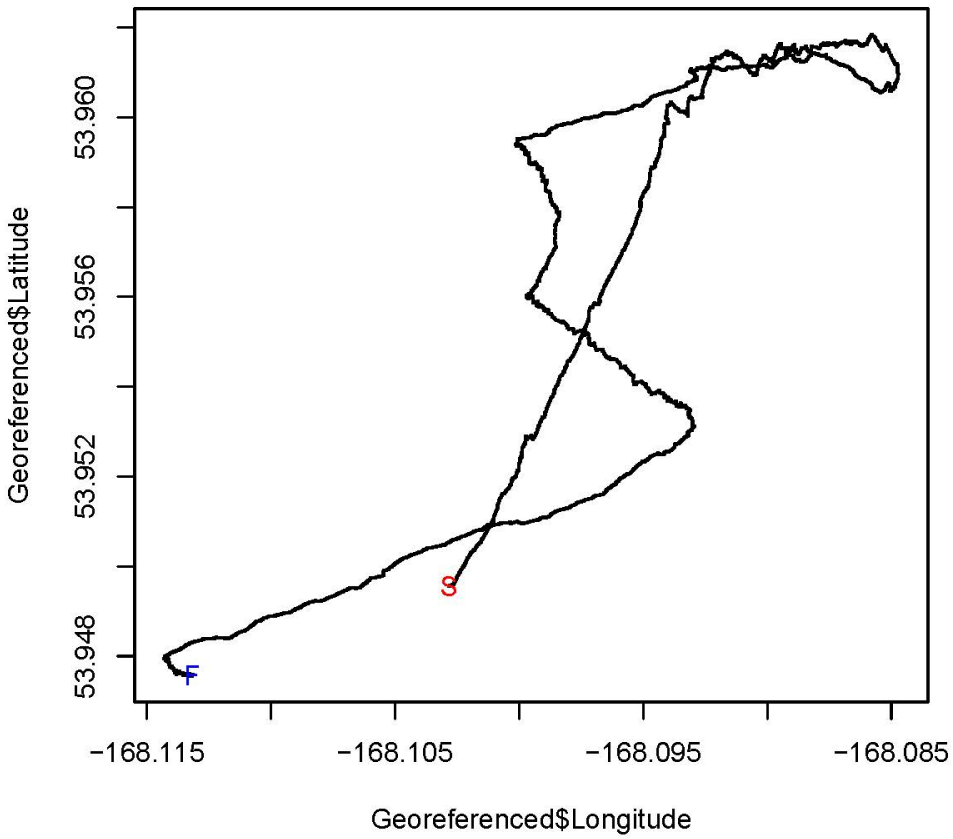

Figure 2: GeoReferenced plot

```
> Georeferenced<-GeoReference(DRoutput,gpsformat[5:6,])
```

and another simple plotting call gives us figure 2

```
plot(Georeferenced$Longitude,Georeferenced$Latitude,pch=".")
points(gpsformat$Longitude[5],gpsformat$Latitude[5],pch="S",col="Red") #Start
points(gpsformat$Longitude[6],gpsformat$Latitude[6],pch="F",col="Blue") #Finish
```

2.6 GeoRef function

The *GeoRef* function is a wrapper for the *GeoReference* function that forces your deadreckoning track through multiple known locations in series, basically it just loops the GeoReference function through all of your dead reckoning data and GPS locations.

3 Graphing

3.1 Color Graphing

If you want to do color graphing of the tracks within R I have provided a function to do this but you will need to get data from the General Bathymetric Chart of the Oceans (GEBCO), or some other raster bathymetry/elevation data at an appropriate resolution for your project, to create the background map. http://www.gebco.net.

```
> require(scatterplot3d)
> require(onion)
> require(RColorBrewer)
> require(lattice)
> library(plotrix)
> library(fields)
> require(rgl)
> library(TrackReconstruction)
> #Import data
> #setwd("G:\\filepath\\gebco_08")
> #bathymetry<- read.table("Gebco1.asc",sep=",",header=TRUE)
> #or get the example data from the package
> data(bathymetry)
> col=gray(0:200/200)
> #format data for graphing
> image.xyz=tapply(bathymetry$Depth, list(bathymetry$Long, bathymetry$Lat), unique)
> #create palatte for depth colors
> Bathymetry.palatte<-colorRampPalette(brewer.pal(9, "Blues"),bias=3)

>          image.plot(x=as.numeric(dimnames(image.xyz)[[1]]),
+          y=as.numeric(dimnames(image.xyz)[[2]]),
+          z=image.xyz,
+         col=c(rev(Bathymetry.palatte(100)),#gray(0:20/20),
+         terrain.colors(100)),
+         breaks=round(c(seq(from=min(image.xyz),to=0,length.out=101),
+         seq(from=max(image.xyz)/101,to=max(image.xyz),length.out=100))),
+         ylab="",
+          xlab=""
+         #,smallplot=2 #plots legend off x axis
+          )
```

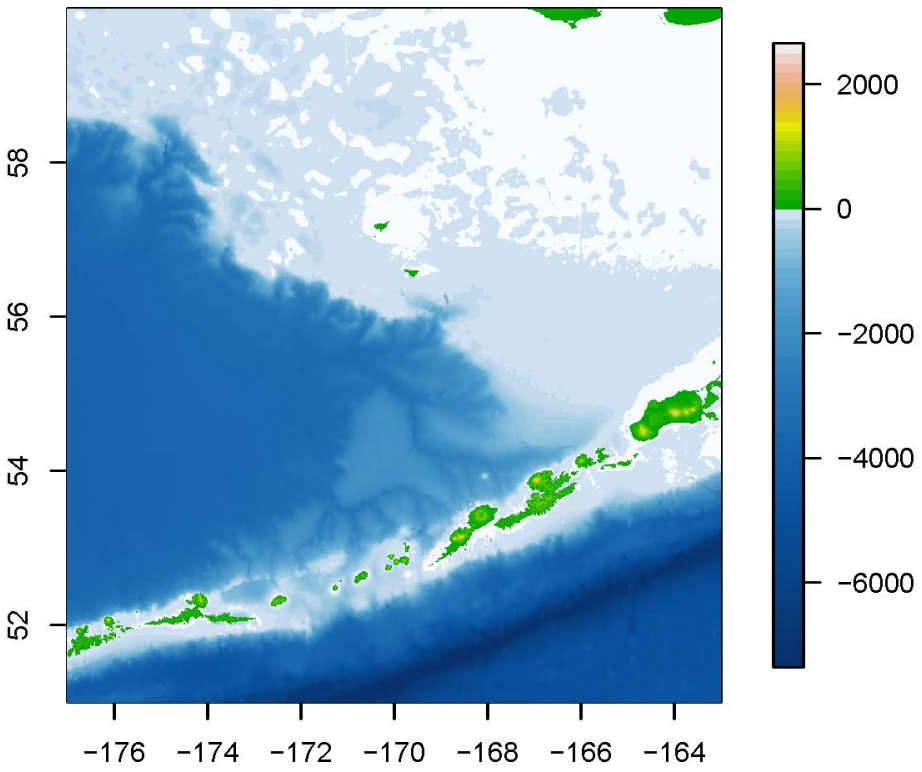

Figure 3: Eastern Bering Sea Bathymetry

Often, I am interested in zooming in on a portion of the graph to look at just a section of the track. Depending on how zoomed in you get, the base graph can look pixelated. To help make the graph look better, there are some interpolating functions in the library(fields) that smooth this pixelated look. This has been incorporated into the *Mapper* function. If you are concerned with accuracy, this may not be good for you, but for presentations and visualizing the data where aesthetics matter more than accuracy, this will look better. See the help file for the *Mapper* function and try out the example.

3.2 3D Graphing

If you are interested in doing a 3D graph here is a function to get you started. I have not yet incorporated this as a function into the TrackReconstruction package as it is simply from the scatterplot3d package. The *GraphLimits* function is in the TrackReconstruction package and automatically calculates limits of the graph in order to minimize the distortion that would occur if R imposed it's default graph limits given the actual limits of the data.

```
>  data(georef1min01)
>  limits<-GraphLimits(georef1min01)
>          Sminlat=limits$miny
>          Smaxlat=limits$maxy
>          Sminlong=limits$minx
>          Smaxlong=limits$maxx

>  scatterplot3d(georef1min01$Longitude,georef1min01$Latitude,(georef1min01$Depth*-1),
+          color="black",#ifelse(georef1min01$SunTimes==1,"red","black"), shades night
+                  #and day if you have the data
+          type="l",
+          lwd=1,
+          #pch=".",
+          highlight.3d=F,
+          angle=55,
+          xlim=c(Sminlong,Smaxlong),
+          ylim=c(Sminlat,Smaxlat),
+          zlim=c(0,-80),
+          zlab="Depth",
+          ylab="Latitude",
+          xlab="Longitude",
+          #x.ticklabs=round(seq(from=Sminlong,to=Smaxlong, by=(Smaxlong-Sminlong)/4),digits=2),
+          #y.ticklabs=round(seq(from=Sminlat,to=Smaxlat, by=(Smaxlat-Sminlat)/4),digits=2),
+          #z.ticklabs=c(-80,-60,-40,-20,0),
+          cex.lab=1,
+          cex.axis=1,
+          cex.symbols=1,
+          #lab=c(3, 4),
+          lab.z=5
+  )
```

That chunk of code resulted in Figure 4.

Next is a cool moveable 3D plotter function from the rgl package. This is not directly embeddable into a presentation as far as I know but it's good to get an idea of what your data look like. This doesn't make a Sweave embeddable figure for this document but the code works (at the time of writing).

```
 library(rgl)
 #par3d(FOV=160)

 plot3d(georef1min01$Longitude,georef1min01$Latitude,georef1min01$Depth*-1,
```

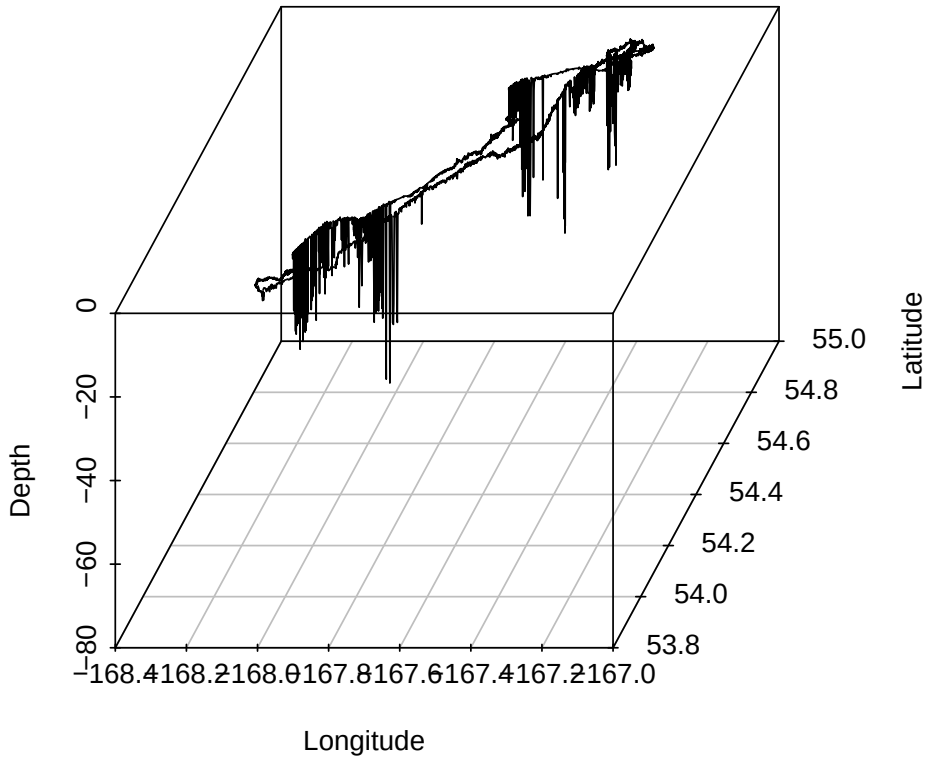

Figure 4: 3Dplot

```
        #col=ifelse(georef1min01$SunTimes==1,"red","black"), #daytime related color
        #col=terrain.colors(100)[rescale(georef1min01$Speed,c(0,100))] #speed related color
        size=.75,
        xlim=c(Sminlong,Smaxlong),
        ylim=c(Sminlat,Smaxlat),
        xlab="Longitude",
        ylab="Latitude",
        zlab="Depth m"
        )
points3d(-168.035,53.931,0,#Bogoslof
#points3d(-170.294,57.107,0,#St.Paul
        size=4,
        color=c("blue"),
        point_antialias=TRUE
        )
#play3d(spin3d(axis=c(1,0,0), rpm=6), duration=10)
```

4 Worm Movie!!!

This is a rather extensive undertaking and requires the animation package which itself requires considerable effort to understand. The animation package can make HTML, GIF, Flash and PDF animations as well as .mp4, .avi and .wmv videos. Only the HTML animations do not require downloading other free programs off the internet so that is what I am going to use as an example here and I will also show how to stitch a video together. What the animation package does is make a series of graphs that will be put together like an old school Disney animated movie. Updates of the animation package has caused previously working code to stop working for me. New versions making old code break is one (of many) VERY frustrating features of R that makes me scream at my computer and pull my hair. However, I was able to make the code below work with no problems using R 3.0.2 64bit with animation package version 2.2. The first obstacle to tackle is creating the data base.

4.1 Making the 3D data base array

The first thing we need to determine is the time over which the movie will run and a column of DateTime in the same format as your track relocation data. Looking at the *head()* and *tail()* of the three georef1min0X data sets that come with the package indicate that the data start on July 14, 2009 and end July 28, 2009.

This following code creates a vector of times from midnight July 14 to midnight July 28, the dates from which the georef1min01-03 data sets are included. The format function converts to timedate format of the track relocation data.

```
>  DateData<-seq(ISOdatetime(2009,07,14,00,00,00, tz="GMT"),ISOdatetime(2009,07,28,00,00,00,
+          tz="GMT"), by="min")
>  head(DateData)

[1] "2009-07-14 00:00:00 GMT" "2009-07-14 00:01:00 GMT"
[3] "2009-07-14 00:02:00 GMT" "2009-07-14 00:03:00 GMT"
[5] "2009-07-14 00:04:00 GMT" "2009-07-14 00:05:00 GMT"
```

Next we need to make an array to hold the relocation data for the three tracks we want to plot. The data are 15 days long and relocations are by the minute so

```
numrows<-15*24*60
numtracks<-3
numcols<-2 #Lat and Long data
#The 0 is what will fill the matrix temporarily
Bogswormdata<-array(0,c(numrows,numcols,numtracks))
```

```
data(georef1min01)
data(georef1min02)
data(georef1min03)
datafiles=list(georef1min01,georef1min02,georef1min03)
for(i in 1:3)
{
        rawdata=as.data.frame(datafiles[i])
        #this line determines the row(date and time) of DateData that the georef1min01
                #track start on
        matcher=1+as.double(abs(difftime(DateData[1],round(strptime(rawdata[1,1],format=
                "%d-%b-%Y %H:%M:%S",tz="GMT"), units = c("mins")), tz="GMT",units="mins")))
        nrows=nrow(rawdata)+matcher-1
        for(j in 1:2)
        {
                #Inserts Lat and Long data into the data array
                Bogswormdata[matcher:nrows,j,i]<-(rawdata[,(j+4)])
        }
}
```

4.2 Function to make the graphs

The following is code to make the individual plots that are to be stitched together given the example data above. It makes 360 individual plots. If you just want to see if it works, you can make fewer plots try *for (j in 100:150)* or whatever you have time/patience for.

```
require(RColorBrewer)
data(bathymetry)
image.xyz=tapply(bathymetry$Depth, list(bathymetry$Long, bathymetry$Lat), unique)
Bathymetry.palatte<-colorRampPalette(brewer.pal(9, "Blues"),bias=3)
plotter=function()
{
        for(j in 1:360)#360 is the number of hours
        {
                par(xpd=T,mar=c(5,5,4,2),bg="black")
                image(x=as.numeric(dimnames(image.xyz)[[1]]),y=as.numeric(dimnames
                        (image.xyz)[[2]]), z=image.xyz,
                        col=c(rev(Bathymetry.palatte(100)),terrain.colors(100)),
                        breaks=round(c(seq(from=min(image.xyz),to=0,length.out=101)
                                ,seq(from=max(image.xyz)/101,to=max(image.xyz),
                                length.out=100))),
                        axes=T,
                        xlab="Longitude",
                        ylab="Latitude",
                        cex.axis=2,
                        cex.lab=2,
                        col.ticks="white",
                        col.axis="white",
                        col.lab="white"
                        #xlim=c(-171.30,-171),
                        #ylim=c(56.0,56.30)
                )
                #Add date and time to graph
                text(x=-165.5,y=51.5, DateData[(j*60+241)], col="white", cex=2)
                        points(Bogswormdata[(j*60):(j*60+480),2,],Bogswormdata[
                                (j*60):(j*60+480),1,],
```

```
                        pch=".",
                       cex=6
                )
                print(j)
        }
}
```

The following is code that I used to color code the worms according to behavior and with the addition of tracks from a second island, a legend at the top and slight differences in background shading to indicate night and day (which actually turned into a rather annoying flashing effect but I was too proud of the accomplishment to admit that it actually made the graph worse). This example is just to show that there is a lot you can add to these graphs, just add another column to each matrix in the array for the data that you want.

```
#Code to shade for night
shade <- rgb(0, 0, 0, alpha=80, maxColorValue=255)
plotter=function()
{
        for(j in 1:360)#360 is the number of hours
        {
                #plot the base map
                par(xpd=T,mar=c(5,5,4,2),bg="black")
                image(x=as.numeric(dimnames(image.xyz)[[1]]),y=as.numeric(dimnames
                                (image.xyz)[[2]]), z=image.xyz,
                        col=c(rev(Bathymetry.palatte(100)),terrain.colors(100)),
                        breaks=round(c(seq(from=min(image.xyz),to=0,length.out=101),
                                seq(from=max(image.xyz)/101,to=max(image.xyz),
                                length.out=100))),
                        axes=T,
                        xlab="Longitude",
                        ylab="Latitude",
                        cex.axis=2,
                        cex.lab=2,
                        col.ticks="white",
                        col.axis="white",
                        col.lab="white"
                        #xlim=c(-171.30,-171),
                        #ylim=c(56.0,56.30)
                )
                legend(-178, 60.5,
                        legend=c("Dive", "Rest", "Shake", "Spin","W-Spin"),
                        col = c("red","yellow","purple","orange","brown"),
                        text.col = c("red","yellow","purple","orange","brown"),
                        lty = c(1,1,1,1,1,1),
                        horiz=T,
                        bty="o",
                        box.col="white",
                        box.lty=0,
                        cex=2
                       )
        #Night and Day shading
                if(DateData[(j*60+240),2]==2)
                        {
                        rect(xleft=-177,xright=-163,ybottom=51,ytop=60,col=shade,border=NA);
                        }
```

```
        #Add date and time to graph
                text(x=-165.5,y=51.5, DateData[(j*60+241)], col="white", cex=2)
        #plot Pribs tracks
                points(Pribswormdata[(j*60):(j*60+480),2,],Pribswormdata[(j*60):
                                (j*60+480),1,],
                        col=ifelse(Pribswormdata[(j*100):(j*100+1000),4,]==1,"red"
                                ,"black"),
                        col=ifelse(Pribswormdata[(j*60):(j*60+480),5,]==0,"red",ifelse(
                                Pribswormdata[(j*60):(j*60+480),5,]==1,"yellow",ifelse(
                                Pribswormdata[(j*60):(j*60+480),5,]==2,"purple",ifelse(
                                Pribswormdata[(j*60):(j*60+480),5,]==3,"orange",ifelse(
                                Pribswormdata[(j*60):(j*60+480),5,]==4,"brown","black"))))),
                        pch=".",
                        cex=4,
                        xlab="Longitude",
                        ylab="Latitude",
                        xlim=c(-169.11,-168.90),
                        ylim=c(54.0,54.4),
                )
                #plot Bogs tracks
                points(Bogswormdata[(j*60):(j*60+480),2,],Bogswormdata[(j*60):(j*60+480),1,],
                        col=ifelse(Bogswormdata[(j*60):(j*60+480),5,]==0,"red",ifelse(
                                Bogswormdata[(j*60):(j*60+480),5,]==1,"yellow",ifelse(
                                Bogswormdata[(j*60):(j*60+480),5,]==2,"purple",ifelse(
                                Bogswormdata[(j*60):(j*60+480),5,]==3,"orange",ifelse(
                                Bogswormdata[(j*60):(j*60+480),5,]==4,"brown","black"))))),
                        pch=".",
                        cex=4,
                        xlab="Longitude",
                        ylab="Latitude",
                        xlim=c(-169.11,-168.90),
                        ylim=c(54.0,54.4),
                )
                print(j)
        }
}
```

4.3 Dealing with the animation package

The following code creates the plots, saves them in a folder and stitches them together to make an HTML movie. If you want to make a video, see the *saveVideo* function in the animation package. You will need to download ffmpeg.exe which can currently be found at http://ffmpeg.org/ to make a video.

```
require(animation)
ani.options(
        #subfolders will be made in this folder to store various things
        outdir="C:\\filepath to where you want the video\\",
        interval=0.1,
        ani.width=1000,
        ani.height=1000)
saveHTML(plotter())
```

If you happen to have a large number of pictures already, you can still stitch these together using a different function called *im.convert* but you need to download ImageMagik. Read the *im.convert* help page for details. *gm.convert* does the same thing but requires the GraphicsMagik program, one might work better

than the other on your system. The following code is a little ridiculous if you have large numbers of pictures you want to stitch together but sometimes ugly works.

```
setwd("G:\\file path where your images are")
ani.options(interval=0.1,
ani.width=1000,
nmax=500,
ani.height=1000,
convert = shQuote("c:/program files (x86)/imagemagick-6.8.0-Q16/convert.exe")
)
im.convert(c(#"Rplot*.png", output = "bm-animation1.gif")
"Rplot1.png","Rplot2.png","Rplot3.png","Rplot4.png","Rplot5.png",
"Rplot6.png","Rplot7.png","Rplot8.png","Rplot9.png","Rplot10.png",
"Rplot11.png","Rplot12.png","Rplot13.png","Rplot14.png","Rplot15.png",
"Rplot16.png","Rplot17.png","Rplot18.png","Rplot19.png"
)
, output = "bm-animation1.gif")
```

5 Appendix 1 PERL and DOS

One possible free source for PERL is this http://www.activestate.com/activeperl/downloads. Installing should create a Perl folder in the directory at C:/Perl I keep all my Perl programs in this folder. See http://www.anaesthetist.com/mnm/perl/Findex.htm for a starting tutorial on regular expressions in Perl and sites such as http://regexpal.com for testing your code.

This first PERL program subsets between two dates as a rough cut to trim data to make the raw file smaller or in a batch file to cut multiple trips out of a single tag deployment or to cut single trips into multiple segments of a suitable size that the trackreconstruction package can handle. It is designed to do the same thing as the *Splitter* R function in TrackReconstruction.

```
#!C:\Perl64\bin
#Subset data between two dates

# The next four lines are for bachfile use
$srcfile= $ARGV[0]; #input file
$outfile= $ARGV[1]; #output file
$indate= $ARGV[2]; #start date
$outdate= $ARGV[3]; #end date
       #The next four lines are an example for a single file this needs to
       #look like your raw file date and time format, \t is the regex for tab
#$indate="12-Jul-2011\t16:19:49";
#$outdate="12-Jul-2011\t16:34:20";
       #this file is the raw data
#$srcfile="C:\Users\Marine\Desktop\DR check\09A0570square2.csv";
       #this output file is the new one the Program will create
#$outfile="C:\Users\Marine\Desktop\DR check\01.csv";

open(IN, $srcfile) or die "Cant open infile ... ";
open(OUTF, ">$outfile") or die;

#Some thought must be put into this number (16) as it is =RmL*Hz/2
       #in the DeadReckoning function.
$tails=16;
$lineflag=1; # This is the number of rows of header on the original raw file
```

```
$lineflag1=0;
$lineflag2=0;

#This while loop goes line by line looking for the indate and outdate to
       #determine what lines they are on
while (($line = <IN>) && ($lineflag2 ==0))
{
       if($line =~ /$indate/)
       {
              $lineflag1= $lineflag;
       }
       if($line =~ /$outdate/)
       {
              $lineflag2= $lineflag+$tails;
       }
$lineflag += 1;
}

$lineflag1=$lineflag1-$tails;
#These next two lines print line numbers in the command prompt
       #window so you can see things are working.
print "$lineflag1";
print "$lineflag2";

open(IN, $srcfile) or die "Cant open infile ... ";

$lineflag=1;
#This next line is the header and should have these elements with
       #perhaps Date and Time as one and optional Speed
       #and possibly any other data you want to keep.
print OUTF "Date\tTime\tDepth\tMagSurge\tMagSway\tMagHeave\tAccSurge\tAccSway\t
       AccHeave\r";

while (($line = <IN>) && ($lineflag<$lineflag2))
{
# If the line is between the start and end dates and does not
       #have excessive tabs (for my data that ment
       #that the line was blank or had data I was not interested in
if(($lineflag >= $lineflag1) && ($lineflag <= $lineflag2) && ($line !~/.\t\t\t\t\t./))
       {
              print OUTF "$line";
       }
#This section might work to stop the searching after the end date line was
       #passed, speeding up the program.
       if($lineflag>$lineflag2)
       {
              last;
       }
       $lineflag += 1;
}

close IN;
close OUTF;
```

I saved this as `DD_GPS_Date.pl` in the `C:/Perl` folder. Run it by opening the command prompt in

Windows, navigating to the C:/Perl folder. typing "cd\" will get you to the C: drive and then "cd Perl" to get to that folder. Type "DD_GPS_Date.pl" and the file will run and create the new file.

5.1 Batch files and file management

If you have multiple files you want to do this to, we need a batch file. I've created a batch file folder on the C: drive but it doesn't matter where you keep these. Here is an example of what these look like except I have all four arguments for each trip on a single line with a space between each argument.

```
:: DDGPS Date.bat
:: Starts Perl DD_GPS_Date.pl program to cut large files into smaller files
:: between GPS locations
::

CD\
CD Perl64
print(1)
DD_GPS_Date.pl "C:\filepath\inputfilename.csv" "G:\filepath\animal01\trip01\01.csv"
       "16-Jul-2009\t08:56:24" "21-Jul-2009\t00:31:21"
DD_GPS_Date.pl "C:\filepath\inputfilename.csv" "G:\filepath\animal01\trip02\02.csv"
       "22-Jul-2009\t01:18:56" "28-Jul-2009\t09:45:22"
print(2)
DD_GPS_Date.pl "C:\filepath\inputfilename.csv" "G:\filepath\animal02\trip01\01.csv"
       "14-Jul-2009\t22:27:00" "19-Jul-2009\t14:30:01"
DD_GPS_Date.pl "C:\filepath\inputfilename.csv" "G:\filepath\animal02\trip02\02.csv"
       "21-Jul-2009\t04:18:09" "26-Jul-2009\t10:53:28"
DD_GPS_Date.pl "C:\filepath\inputfilename.csv" "G:\filepath\animal02\trip03\03.csv"
       "27-Jul-2009\t04:19:01" "27-Jul-2009\t17:13:33"
DD_GPS_Date.pl "C:\filepath\inputfilename.csv" "G:\filepath\animal02\trip04\04.csv"
       "29-Jul-2009\t05:40:23" "02-Aug-2009\t22:33:45"
DD_GPS_Date.pl "C:\filepath\inputfilename.csv" "G:\filepath\animal02\trip05\05.csv"
       "05-Aug-2009\t05:24:40" "08-Aug-2009\t09:49:49"
DD_GPS_Date.pl "C:\filepath\inputfilename.csv" "G:\filepath\animal02\trip06\06.csv"
       "09-Aug-2009\t05:08:45" "11-Aug-2009\t22:30:52"
print(3)
DD_GPS_Date.pl "C:\filepath\inputfilename.csv" "G:\filepath\animal03\trip01\01.csv"
       "16-Jul-2009\t18:26:14" "22-Jul-2009\t06:21:11"
DD_GPS_Date.pl "C:\filepath\inputfilename.csv" "G:\filepath\animal03\trip02\01.csv"
       "23-Jul-2009\t19:08:24" "28-Jul-2009\t16:15:49"
DD_GPS_Date.pl "C:\filepath\inputfilename.csv" "G:\filepath\animal03\trip03\01.csv"
       "29-Jul-2009\t22:53:43" "04-Aug-2009\t14:36:26"
DD_GPS_Date.pl "C:\filepath\inputfilename.csv" "G:\filepath\animal03\trip04\01.csv"
       "06-Aug-2009\t04:26:13" "11-Aug-2009\t01:22:16"
.
.
.
```

:: is the batch file comment character. These files are saved as *.bat files and are started by simply double clicking with a mouse, be careful, more than once I started these files that way thinking that I was just opening them in a text editor to edit, for that, right click and choose 'open with'. Each line is composed of 4 arguments each of which is surrounded by quotes and a space between. The order of the arguments is the same as that of the $ARGV[0] $ARGV[1] $ARGV[2] $ARGV[3] in the Perl program. "Input file" "output file" "start date" "end date". In this example I had multiple trips for single raw data files. The print(1), print(2) and print(3) are just there to periodically print something on the command prompt screen so I know things are still working and what file the program is working on.

Now that we have single files for an entire trip, we may want to further break these up into between-GPS-fixes sized chunks. The same Perl program can do this but we require many batch files of a similar format to the one above to work through all the trips. Again, the "tails" number (RmL*Hz/2) is important when doing this subsetting. The R programs in section 1.2 above can also do this if your machine can handle it.

```
:: DDGPS Date.bat
:: Starts Perl DD_GPS_Date.pl program to cut contiguous trips into smaller between GPS locations
::

CD\
CD Perl64
DD_GPS_Date.pl "C:\filepath\animal01\trip 01\01.csv" "C:\filepath\animal01\trip 01\01-001.csv"
       "16-Jul-2009\t08:56:24" "16-Jul-2009\t08:57:45"
DD_GPS_Date.pl "C:\filepath\animal01\trip 01\01.csv" "C:\filepath\animal01\trip 01\01-002.csv"
       "16-Jul-2009\t08:57:45" "16-Jul-2009\t09:17:14"
DD_GPS_Date.pl "C:\filepath\animal01\trip 01\01.csv" "C:\filepath\animal01\trip 01\01-003.csv"
       "16-Jul-2009\t09:17:14" "16-Jul-2009\t09:38:12"
DD_GPS_Date.pl "C:\filepath\animal01\trip 01\01.csv" "C:\filepath\animal01\trip 01\01-004.csv"
       "16-Jul-2009\t09:38:12" "16-Jul-2009\t09:53:04"
DD_GPS_Date.pl "C:\filepath\animal01\trip 01\01.csv" "C:\filepath\animal01\trip 01\01-005.csv"
       "16-Jul-2009\t09:53:04" "16-Jul-2009\t10:08:52"
DD_GPS_Date.pl "C:\filepath\animal01\trip 01\01.csv" "C:\filepath\animal01\trip 01\01-006.csv"
       "16-Jul-2009\t10:08:52" "16-Jul-2009\t10:25:04"
DD_GPS_Date.pl "C:\filepath\animal01\trip 01\01.csv" "C:\filepath\animal01\trip 01\01-007.csv"
       "16-Jul-2009\t10:25:04" "16-Jul-2009\t10:42:05"
DD_GPS_Date.pl "C:\filepath\animal01\trip 01\01.csv" "C:\filepath\animal01\trip 01\01-008.csv"
       "16-Jul-2009\t10:42:05" "16-Jul-2009\t11:07:54"
DD_GPS_Date.pl "C:\filepath\animal01\trip 01\01.csv" "C:\filepath\animal01\trip 01\01-009.csv"
       "16-Jul-2009\t11:07:54" "16-Jul-2009\t11:27:00"
DD_GPS_Date.pl "C:\filepath\animal01\trip 01\01.csv" "C:\filepath\animal01\trip 01\01-010.csv"
       "16-Jul-2009\t11:27:00" "16-Jul-2009\t11:53:19"
...
...
...
DD_GPS_Date.pl "C:\filepath\animal01\trip 01\01.csv" "C:\filepath\animal01\trip 01\01-228.csv"
       "20-Jul-2009\t20:12:03" "20-Jul-2009\t20:58:04"
DD_GPS_Date.pl "C:\filepath\animal01\trip 01\01.csv" "C:\filepath\animal01\trip 01\01-229.csv"
       "20-Jul-2009\t20:58:04" "20-Jul-2009\t22:41:49"
DD_GPS_Date.pl "C:\filepath\animal01\trip 01\01.csv" "C:\filepath\animal01\trip 01\01-230.csv"
       "20-Jul-2009\t22:41:49" "20-Jul-2009\t23:46:41"
DD_GPS_Date.pl "C:\filepath\animal01\trip 01\01.csv" "C:\filepath\animal01\trip 01\01-231.csv"
       "20-Jul-2009\t23:46:41" "21-Jul-2009\t00:05:24"
DD_GPS_Date.pl "C:\filepath\animal01\trip 01\01.csv" "C:\filepath\animal01\trip 01\01-232.csv"
       "21-Jul-2009\t00:05:24" "21-Jul-2009\t00:31:21"
CD\
CD Batch
CD GPSBatch
CD Bogoslof
BG_01_02.bat
```

So for trip `BG_01_01` I had 233 GPS locations (including the start and end GPS locations) that created 232 files. It is not necessary to cut your files at every GPS point that you have, in fact I do not recommend it, but that would be the smallest subsetting that you could do and still allow the TrackReconstruction package to georeference your track. The last five lines in this program simply navigate to the next batch file

to get it started, so you just need to double click the first one to get all of them working. If you only want to do one, then just comment out the last line. The real work in all of this is creating these batch files.

5.2 Thinning the data

Sometimes you don't need the high resolution of data your tag collects. If you want to reduce (thin) the size of your master file this can be trivial and it can also be difficult. If you have missing data, things become more difficult, if you don't have missing data, the following Perl script reduces a data set collected at 16 HZ down to 1 Hz.

```
#!C:\Perl64\bin
# Created November 5 2010 to remove all off-second data from Daily Diary tag output

$srcfile=$ARGV[0];
$outfile=$ARGV[1];
#$srcfile="C:\Users\Marine\Desktop\Antarctica\pup 252\GPS segments\trip 1\01-3.csv";
#$outfile="C:\Users\Marine\Desktop\Antarctica\pup 252\GPS segments\trip 1\01-3-2.csv";

open(IN, $srcfile) or die "Cant open infile ... ";
open(OUTF, ">$outfile") or die;

#This should be 1 less than the amount of data you want to thin, i.e. I want every 16th
#row starting with the first row after the header.
$count=15;
$count2=1;

while ($line = <IN>)
{
       #This prints the header
       if ($count2==1)
       {
              print OUTF "$line";
              $count2=2;
       }
       #This prints every 16th line, good for 16Hz data
       if ($count==16)
       {
              print OUTF "$line";
              $count=0;
       }
$count += 1;
}
print uno
close IN;
close OUTF;
```

If your database has missing data then you might want to try something like this that looks for unique strings that occur. My file had a string of 3 tabs that only occurred on the second (not for the 15 fractions of a second in my 16Hz sampling rate), if you are lucky or more clever than me, your data will have a unique string of characters that you can search for.

```
#!C:\Perl64\bin
# Created April 7 2010 to remove all off-second data from Daily Diary tag output

$srcfile= $ARGV[0];
$outfile= $ARGV[1];
```

```
#$srcfile="G:\Bog 2009 Toughbook\2009 Data\Daily Diary\Bogoslof\Cu09BG4\09A0574tab.csv";
#$outfile="G:\Bog 2009 Toughbook\2009 Data\Daily Diary\Bogoslof\Cu09BG4\09A0574sec.csv";

#I do not remember what these next three lines do, probably not needed.
foreach $argnum (0 ..$#ARGV){
       print"@ARGV$[argnum]\n";
       }

open(IN, $srcfile) or die "Cant open infile ... ";
open(OUTF, ">$outfile") or die;

#you can use the following line to test to see if the program works without
       #going through the entire data set.
#while ( ($line = <IN>) && ($count <= 1000) )
while ($line = <IN>)
{
       if ($line !~/.\t\t\t./)
       {
              print OUTF "$line";
       }
}

close IN;
close OUTF;
```

Save one of these as a `DDsec_red.pl` and use this in a batch file that looks something like this to reduce many files while you sleep.

```
:: DDsec_red.bat
:: Starts Perl sec_red.pl program to reduce large DD files to 1 sec samples
::

CD\
CD Perl64
DDsec_red.pl "C:\filepath\Cu09BG01\01.csv" "E:\filepath\Cu09BG01\01_1sec.csv"
DDsec_red.pl "C:\filepath\Cu09BG02\01.csv" "E:\filepath\Cu09BG02\01_1sec.csv"
DDsec_red.pl "C:\filepath\Cu09BG03\01.csv" "E:\filepath\Cu09BG03\01_1sec.csv"
DDsec_red.pl "C:\filepath\Cu09BG04\01.csv" "E:\filepath\Cu09BG04\01_1sec.csv"
DDsec_red.pl "C:\filepath\Cu09BG05\01.csv" "E:\filepath\Cu09BG05\01_1sec.csv"
.
.
.
.
```

GOOD LUCK!!!
THE END