Limpet force meter analysis code

2017-10-27

#################################################################################  
## Function loadCalibFile  
## This function imports a calibration data file (.csv) with 4 columns of data  
## that should be titled 'axis', 'direction', 'mass.grams', 'analogValue'.  
## The input should be the file name and directory of the file  
## The output will be a dataframe holding the data.   
loadCalibFile = function(fname = 'CalibrationFiles\_Apr202016.csv' ,  
 fdir = 'Dropbox/Limpet\_force\_meter/'){  
 # Determine which computer we're working on so we can put the appropriate  
 # file path prefix on the Dropbox file directory  
 platform = .Platform$OS.type  
 if (platform == 'unix'){  
 prefixDrive = '~/'  
 } else if (platform == 'windows'){  
 prefixDrive = 'D:/'  
 }  
   
 fdir = paste0(prefixDrive,fdir)  
   
 # Import the calibration data file  
 calib = read.csv(paste0(fdir,fname))   
}  
#################################################################################  
  
  
################################################################################  
# Function plotSeparateAxesCalib  
# A function to plot the individual positive and negative calibration data for  
# each of the axes (X, Y, Z). This includes the zero values for a particular   
# axis in each plot of the positive or negative direction. Note that the Z   
# axis only has one direction (positive = downwards)  
# Input a data frame imported from the raw calibration data file  
  
plotSeparateAxesCalib = function(calib = calib){  
# Pull apart the separate axes calibration data, and include the zero values  
 xpos = calib[calib$axis == 'X' & (calib$direction == 'zero' |   
 calib$direction == 'positive'),]  
 xpos = droplevels(xpos)  
   
 xneg = calib[calib$axis == 'X' & (calib$direction == 'zero' |   
 calib$direction == 'negative'),]  
 xneg = droplevels(xneg)  
   
 ypos = calib[calib$axis == 'Y' & (calib$direction == 'zero' |   
 calib$direction == 'positive'),]  
 ypos = droplevels(ypos)  
   
 yneg = calib[calib$axis == 'Y' & (calib$direction == 'zero' |   
 calib$direction == 'negative'),]  
 yneg = droplevels(yneg)  
   
 zneg = calib[calib$axis == 'Z' & (calib$direction == 'zero' |   
 calib$direction == 'negative'),]  
 zneg = droplevels(zneg)  
  
# Plot the raw data for each axis  
 par(mfrow = c(3,2))  
 #############################  
# Positive X-axis  
 plot(x = xpos$analogValue, y = xpos$mass.grams, type='p',   
 xlab = 'Analog count',  
 ylab = 'Mass, g',  
 main = 'X axis positive')  
 mod = lm(mass.grams~analogValue, data = xpos)  
 modSum = summary(mod)  
 abline(mod)  
 r2 = modSum$adj.r.squared  
 mylabel = bquote(italic(R)^2 == .(format(r2, digits = 3)))  
 legend('topleft', legend = mylabel, bty = 'n')  
 ###########################  
# Negative X-axis  
 plot(x = xneg$analogValue, y = xneg$mass.grams, type='p',   
 xlab = 'Analog count',  
 ylab = 'Mass, g',  
 main = 'X axis negative')  
 mod = lm(mass.grams~analogValue, data = xneg)  
 modSum = summary(mod)  
 abline(mod)  
 r2 = modSum$adj.r.squared  
 mylabel = bquote(italic(R)^2 == .(format(r2, digits = 3)))  
 legend('topright', legend = mylabel, bty = 'n')  
 #######################  
# Positive Y axis  
 plot(x = ypos$analogValue, y = ypos$mass.grams, type='p',   
 xlab = 'Analog count',  
 ylab = 'Mass, g',  
 main = 'Y axis positive')  
 mod = lm(mass.grams~analogValue, data = ypos)  
 modSum = summary(mod)  
 abline(mod)  
 r2 = modSum$adj.r.squared  
 mylabel = bquote(italic(R)^2 == .(format(r2, digits = 3)))  
 legend('topleft', legend = mylabel, bty = 'n')  
 ###########################  
# Negative Y-axis  
 plot(x = yneg$analogValue, y = yneg$mass.grams, type='p',   
 xlab = 'Analog count',  
 ylab = 'Mass, g',  
 main = 'Y axis negative')  
 mod = lm(mass.grams~analogValue, data = yneg)  
 modSum = summary(mod)  
 abline(mod)  
 r2 = modSum$adj.r.squared  
 mylabel = bquote(italic(R)^2 == .(format(r2, digits = 3)))  
 legend('topright', legend = mylabel, bty = 'n')  
 #######################  
# Z-axis  
 plot(x = zneg$analogValue, y = zneg$mass.grams, type='p',   
 xlab = 'Analog count',  
 ylab = 'Mass, g',  
 main = 'Z axis negative')  
 mod = lm(mass.grams~analogValue, data = zneg)  
 modSum = summary(mod)  
 abline(mod)  
 r2 = modSum$adj.r.squared  
 mylabel = bquote(italic(R)^2 == .(format(r2, digits = 3)))  
 legend('topleft', legend = mylabel, bty = 'n')   
} # end of plotSeparateAxesCalib  
  
#############################################################################  
# Produce a list of calibration coefficients for the three axes.   
# The output will be a list with entries X, Y, Z, each containing a field  
# 'intercept','slope', and 'R2' (R-squared)  
# The regression coefficients intercept and slope will convert an input   
# analogValue into an estimate of Force (Newtons), with a positive or negative  
# value depending on the direction of the force application.   
  
calibCoefficients = function(calib = calib){  
 #########################################  
 xax = calib[calib$axis == 'X',]  
# Convert mass into force (Newtons) by multiplying by gravity acceleration  
 xax$Force.N = (xax$mass.grams/1000) \* 9.8066  
# Make negative-direction values into negative forces  
 xax$Force.N[xax$direction == 'negative'] = -1 \*   
 xax$Force.N[xax$direction=='negative']  
 #########################################  
# Y-axis all data, converted to force in Newtons  
 yax = calib[calib$axis == 'Y',]  
# Convert mass into force (Newtons) by multiplying by gravity acceleration  
 yax$Force.N = (yax$mass.grams/1000) \* 9.8066  
# Make negative-direction values into negative forces  
 yax$Force.N[yax$direction == 'negative'] = -1 \*   
 yax$Force.N[yax$direction=='negative']   
 ############################################  
# Z-axis all data, converted to force in Newtons  
 zax = calib[calib$axis == 'Z',]  
# Convert mass into force (Newtons) by multiplying by gravity acceleration  
 zax$Force.N = (zax$mass.grams/1000) \* 9.8066  
# Make negative-direction values into negative forces  
 zax$Force.N[zax$direction == 'negative'] = -1 \*   
 zax$Force.N[zax$direction=='negative']  
 ################################################  
 # Fit regressions for each axis  
 modX = lm(Force.N~analogValue, data = xax)  
 modXSum = summary(modX)  
 # Extract intercept, slope, R^2 of regression  
 myinterceptX = coef(modXSum)[1,1]  
 myslopeX = coef(modXSum)[2,1]  
 r2X = modXSum$adj.r.squared  
 ################  
 modY = lm(Force.N~analogValue, data = yax)  
 modYSum = summary(modY)  
 # Extract intercept, slope, R^2 of regression  
 myinterceptY = coef(modYSum)[1,1]  
 myslopeY = coef(modYSum)[2,1]  
 r2Y = modYSum$adj.r.squared  
 ################  
 modZ = lm(Force.N~analogValue, data = zax)  
 modZSum = summary(modZ)  
 # Extract intercept, slope, R^2 of regression  
 myinterceptZ = coef(modZSum)[1,1]  
 myslopeZ = coef(modZSum)[2,1]  
 r2Z = modZSum$adj.r.squared  
 ###############################################  
 # Combine data into an output list  
 output = list(X = data.frame(intercept = myinterceptX, slope = myslopeX,   
 R2 = r2X),  
 Y = data.frame(intercept = myinterceptY, slope = myslopeY,R2 = r2Y),  
 Z= data.frame(intercept = myinterceptZ, slope = myslopeZ,R2 = r2Z))  
 output  
}  
################################################################################  
  
  
################################################################################  
# Function plotAxesCalib  
# This function will produce three plots (one per axis X, Y, Z) to show the   
# calibration data and regression fit to those data. The regression will use  
# analogValue as the x-axis, and force in newtons (converted from mass in grams)  
# for the y-axis.   
# Input should be a data frame of calibration data, with a column 'axis',   
# 'direction', 'mass.grams', and 'analogValue', produced by importing the  
# raw calibration data file   
  
plotAxesCalib = function(calib = calib){  
 ######################################################  
 # Combine all data for a single axis into one set, convert to force units,  
 # estimate regression fit.   
 ######################################################  
 # X-axis all data, converted to force in Newtons  
 par(mfrow=c(3,1))  
 xax = calib[calib$axis == 'X',]  
 # Convert mass into force (Newtons) by multiplying by gravity acceleration  
 xax$Force.N = (xax$mass.grams/1000) \* 9.8066  
 # Make negative-direction values into negative forces  
 xax$Force.N[xax$direction == 'negative'] = -1 \*   
 xax$Force.N[xax$direction=='negative']  
   
 plot(x = xax$analogValue, y = xax$Force.N,las = 1,  
 xlab = 'Analog Count',  
 ylab = 'Force, N',  
 main = 'X axis')  
 abline(h = 0, lty = 2, col = 'grey70')  
 # Fit regression  
 modX = lm(Force.N~analogValue, data = xax)  
 modXSum = summary(modX)  
 abline(modX)  
 myinterceptX = coef(modXSum)[1,1]  
 myslopeX = coef(modXSum)[2,1]  
 r2 = modXSum$adj.r.squared  
   
 # Start by making an expression vector to hold the 2 lines of output:  
 rp = vector('expression',2)  
   
 # Write the first line, which will give R-squared and   
 # pull the value from the data frame wt.fits  
 # The double == prints an equal sign when used inside expression()  
 rp[1] = substitute(expression(italic(R)^2 == MYVALUE),  
 list(MYVALUE = format(r2,digits = 4)))[2]  
   
 # Write the 2nd line, which will pull 2 values from data frame wt.fits:  
 rp[2] = substitute(expression(italic(Y) == MYVALUE2 + MYVALUE3\*x),  
 list(MYVALUE2 = format(myinterceptX, digits = 3),  
 MYVALUE3 = format(myslopeX,digits = 3)))[2]  
 # Finally, simply plot with legend() function:  
 legend('topleft', legend = rp, bty = 'n')  
   
 ############################################  
 # Y-axis all data, converted to force in Newtons  
 yax = calib[calib$axis == 'Y',]  
 # Convert mass into force (Newtons) by multiplying by gravity acceleration  
 yax$Force.N = (yax$mass.grams/1000) \* 9.8066  
 # Make negative-direction values into negative forces  
 yax$Force.N[yax$direction == 'negative'] = -1 \*   
 yax$Force.N[yax$direction=='negative']  
   
 plot(x = yax$analogValue, y = yax$Force.N,las = 1,  
 xlab = 'Analog Count',  
 ylab = 'Force, N',  
 main = 'Y axis')  
 abline(h = 0, lty = 2, col = 'grey70')  
 # Fit regression  
 modY = lm(Force.N~analogValue, data = yax)  
 modYSum = summary(modY)  
 abline(modY)  
 myinterceptY = coef(modYSum)[1,1]  
 myslopeY = coef(modYSum)[2,1]  
 r2 = modYSum$adj.r.squared  
   
 # Start by making an expression vector to hold the 2 lines of output:  
 rp = vector('expression',2)  
   
 # Write the first line, which will give R-squared and   
 # pull the value from the data frame wt.fits  
 # The double == prints an equal sign when used inside expression()  
 rp[1] = substitute(expression(italic(R)^2 == MYVALUE),  
 list(MYVALUE = format(r2,digits = 4)))[2]  
   
 # Write the 2nd line, which will pull 2 values from data frame wt.fits:  
 rp[2] = substitute(expression(italic(Y) == MYVALUE2 + MYVALUE3\*x),  
 list(MYVALUE2 = format(myinterceptY, digits = 3),  
 MYVALUE3 = format(myslopeY,digits = 3)))[2]  
 # Finally, simply plot with legend() function:  
 legend('topleft', legend = rp, bty = 'n')  
   
 ############################################  
 # Z-axis all data, converted to force in Newtons  
 zax = calib[calib$axis == 'Z',]  
 # Convert mass into force (Newtons) by multiplying by gravity acceleration  
 zax$Force.N = (zax$mass.grams/1000) \* 9.8066  
 # Make negative-direction values into negative forces  
 zax$Force.N[zax$direction == 'negative'] = -1 \*   
 zax$Force.N[zax$direction=='negative']  
   
 plot(x = zax$analogValue, y = zax$Force.N, las = 1,  
 xlab = 'Analog Count',  
 ylab = 'Force, N',  
 main = 'Z axis')  
 abline(h = 0, lty = 2, col = 'grey70')  
 # Fit regression  
 modZ = lm(Force.N~analogValue, data = zax)  
 modZSum = summary(modZ)  
 abline(modZ)  
 myinterceptZ = coef(modZSum)[1,1]  
 myslopeZ = coef(modZSum)[2,1]  
 r2 = modZSum$adj.r.squared  
   
 # Start by making an expression vector to hold the 2 lines of output:  
 rp = vector('expression',2)  
   
 # Write the first line, which will give R-squared and   
 # pull the value from the data frame wt.fits  
 # The double == prints an equal sign when used inside expression()  
 rp[1] = substitute(expression(italic(R)^2 == MYVALUE),  
 list(MYVALUE = format(r2,digits = 4)))[2]  
   
 # Write the 2nd line, which will pull 2 values from data frame wt.fits:  
 rp[2] = substitute(expression(italic(Y) == MYVALUE2 + MYVALUE3\*x),  
 list(MYVALUE2 = format(myinterceptZ, digits = 3),  
 MYVALUE3 = format(myslopeZ,digits = 3)))[2]  
 # Finally, simply plot with legend() function:  
 legend('topright', legend = rp, bty = 'n')   
}  
###############################################################################

We have the following calibration files and trial files. Note that the April 2016 trial data should not be analyzed, as the z-axis was not working.

if (.Platform$OS.type == 'windows'){  
 fdir = "D:/Dropbox/Force Meter Files with Luke/Force Meter Data and Calibration Files/"  
} else if (.Platform$OS.type == 'unix'){  
 fdir = "~/Dropbox/Force Meter Files with Luke/Force Meter Data and Calibration Files/"  
}  
  
  
fnames = dir(fdir)  
fnames # print file names

## [1] "CalibrationFiles\_Apr202016.csv"   
## [2] "CalibrationFiles\_Dec132016.csv"   
## [3] "CalibrationFiles\_Dec142016.csv"   
## [4] "CalibrationFiles\_Jan42017.csv"   
## [5] "CalibrationFiles\_Jan52017.csv"   
## [6] "Events\_classified\_20171208.csv"   
## [7] "Events\_picker\_output\_20171208.csv"   
## [8] "ForceMeterData\_Apr262016.csv"   
## [9] "ForceMeterData\_Dec142016.csv"   
## [10] "ForceMeterData\_Dec152016.csv"   
## [11] "ForceMeterData\_Jan52017.csv"   
## [12] "ForceMeterData\_Jan62017.csv"   
## [13] "ForceMeterData\_May182016.csv"   
## [14] "ForceMeterData\_May192016.csv"   
## [15] "Pound\_ForceData\_Final.csv"   
## [16] "Pound\_ForceData\_withXYZ\_forces.csv"

# Do not bother with Apr 26 2016 data, the Z-axis wasn't working.   
  
#[1] "CalibrationFiles\_Apr202016.csv"  
#[2] "CalibrationFiles\_Dec132016.csv"  
#[3] "CalibrationFiles\_Dec142016.csv"  
#[4] "CalibrationFiles\_Jan42017.csv"   
#[5] "CalibrationFiles\_Jan52017.csv"   
#[6] "ForceMeterData\_Apr262016.csv"   
#[7] "ForceMeterData\_Dec142016.csv"   
#[8] "ForceMeterData\_Dec152016.csv"   
#[9] "ForceMeterData\_Jan52017.csv"   
#[10] "ForceMeterData\_Jan62017csv.csv"  
#[11] "ForceMeterData\_May182016.csv"   
#[12] "ForceMeterData\_May192016.csv"   
  
# subset names by file type (calibration or force trials)  
calibfiles = fnames[grep(x = fnames, pattern = 'Calibration')]  
forcefiles = fnames[grep(x = fnames, pattern = 'ForceMeterData')]  
eventFile = fnames[grep(x = fnames, pattern = 'Pound\_ForceData\_Final.csv')]  
  
# Load the event file  
events = read.csv(paste0(fdir,eventFile))  
events$Date = as.Date(events$Date, format = '%m/%d/%Y')  
  
# Concatenate all calibration data into one data frame  
for (i in 1:length(calibfiles)){  
 temp = read.csv(paste0(fdir,calibfiles[i]))  
 # Extract date from file name  
 locs = regexpr(text = calibfiles[i],  
 pattern='[[:upper:]][[:lower:]]{2}[[:digit:]]\*.csv')  
 month = substr(calibfiles[i],start = locs[[1]][1],   
 stop = locs[[1]][1] + 2)  
 nums = regexpr(text = calibfiles[i],  
 pattern='[[:digit:]]+')  
 if(attr(nums,'match.length')>5){  
 # 2-digit day value  
 day = substr(calibfiles[i],start = nums[[1]][1],  
 stop = nums[[1]][1]+1)  
 yr = substr(calibfiles[i], start = nums[[1]][1]+2,  
 stop = nums[[1]][1]+5)  
 } else {  
 # 1- digit day value  
 day = substr(calibfiles[i],start = nums[[1]][1],  
 stop = nums[[1]][1])  
 yr = substr(calibfiles[i], start = nums[[1]][1]+1,  
 stop = nums[[1]][1]+4)  
 }  
 # Assemble date  
 temp$Date = as.Date(paste0(month,'-',day,'-',yr),format = "%b-%d-%Y")  
   
 if (i == 1){  
 calibs = temp  
 } else {  
 calibs = rbind(calibs,temp)  
 }  
}  
  
if (exists('forces')) rm(forces)  
  
# Concatenate all force data into one data frame  
for (i in 1:length(forcefiles)){  
 temp = read.csv(paste0(fdir,forcefiles[i]))  
  
 # Extract date from file name  
 locs = regexpr(text = forcefiles[i],  
 pattern='[[:upper:]][[:lower:]]{2}[[:digit:]]\*.csv')  
 month = substr(forcefiles[i],start = locs[[1]][1],   
 stop = locs[[1]][1] + 2)  
 nums = regexpr(text = forcefiles[i],  
 pattern='[[:digit:]]+')  
 if(attr(nums,'match.length')>5){  
 # 2-digit day value  
 day = substr(forcefiles[i],start = nums[[1]][1],  
 stop = nums[[1]][1]+1)  
 yr = substr(forcefiles[i], start = nums[[1]][1]+2,  
 stop = nums[[1]][1]+5)  
 } else {  
 # 1- digit day value  
 day = substr(forcefiles[i],start = nums[[1]][1],  
 stop = nums[[1]][1])  
 yr = substr(forcefiles[i], start = nums[[1]][1]+1,  
 stop = nums[[1]][1]+4)  
 }  
 # Assemble date   
 thisDate = as.Date(paste0(month,'-',day,'-',yr),format = "%b-%d-%Y")  
 if (thisDate == as.Date('2016-04-26')){  
 # skip any file from 2016-04-26  
 } else {  
 # Remove columns that don't match these names:  
 keepCols = c('Time.msec.','JOY\_X\_signal','JOY\_Y\_signal','BEAM\_Z\_signal')  
 temp = temp[,keepCols]  
 names(temp)[1] = 'Time.msec'  
 temp$Date = thisDate  
   
 if (!exists('forces')){  
 forces = temp  
 } else {  
 forces = rbind(forces,temp)  
 }  
 }  
   
}  
  
# Rename the Time.msec. column to Time.msec  
#names(forces)[1] = 'Time.msec'

The relevant calibration file for the May 2016 trials is the Apr202016 file.

During the import process, the calibration files and force trial files are concatenated together into two data frames, calibs and forces, each with a Date column that can be used to separate different days. The data frame events contains the identified peck and push events, with dates and millisecond timestamps.

The goal is to go through each identified event and extract the X-axis and Y-axis forces (which need to be estimated based on the associated calibration values). The events data frame currently only has the net euclidean norm (total for all 3 axes).

# For each date in the 'calibs' data frame, fit regressions to each axis  
# and determine conversions from raw counts to force in Newtons.   
  
  
  
dates = unique(calibs$Date)  
  
# 2016-04-20 calibration goes with May 18 + 19 2016 ForceMeterData  
# 2016-12-13 calibration goes with Dec 14 2016 ForceMeterData  
# 2016-12-14 calibration goes with Dec 15 2016 ForceMeterData  
# 2017-01-04 calibration goes with Jan 5 2017 ForceMeterData  
# 2017-01-05 calibration goes with Jan 6 2017 ForceMeterData  
  
for (i in 1:length(dates)){  
 # subset the calibs by date  
 temp = calibs[calibs$Date == dates[i],]  
   
 tempCalib = calibCoefficients(temp)  
 # Assemble into a list based on date  
 if (i == 1){  
 calibCoeffs = list()  
 calibCoeffs[[1]] = tempCalib   
 calibCoeffs[[i]]$Date = dates[i]  
 # names(calibCoeffs[1]) = dates[i]  
 } else if (i > 1) {  
 calibCoeffs[[i]] = tempCalib  
 calibCoeffs[[i]]$Date = dates[i]  
 }  
}  
  
  
for (j in 1:length(calibCoeffs)){  
 if (j == 1){  
 calibDates = calibCoeffs[[j]]$Date   
 } else {  
 calibDates = c(calibDates,calibCoeffs[[j]]$Date)  
 }  
}

# Go through each events in the events data frame, pull out the relevant time  
# which should be in the Time\_msec column and the   
# associated Date. Then go find the relevant rows in the forces data frame  
# to get the raw JOY\_X\_signal, JOY\_Y\_signal, and BEAM\_Z\_signal for a chunk of   
# just before and after the event. Use the relevant calibration data to convert  
# each channel to forces in Newtons,   
events2 = events # make a copy  
events2$X.N = NA # make empty columns  
events2$Y.N = NA  
events2$Z.N = NA  
for (i in 1:length(events$Event)){  
 dateval = events[i,'Date']  
 msecval = events[i,'Time\_msec']  
# Go into forces and find the time point  
# Start by subsetting by Date  
 temp = forces[forces$Date == dateval,]  
 timeMatch = which.min(abs(temp$Time.msec - msecval))  
# Grab some rows ahead and after the timeMatch  
 temp2 = temp[(timeMatch-100):(timeMatch+100),c('JOY\_X\_signal','JOY\_Y\_signal',  
 'BEAM\_Z\_signal','Time.msec','Date')]  
# Use the initial rows as a local zero offset reading to apply the calibration.  
  
# 2016-04-20 calibration goes with May 18 + 19 2016 ForceMeterData  
# 2016-12-13 calibration goes with Dec 14 2016 ForceMeterData  
# 2016-12-14 calibration goes with Dec 15 2016 ForceMeterData  
# 2017-01-04 calibration goes with Jan 5 2017 ForceMeterData  
# 2017-01-05 calibration goes with Jan 6 2017 ForceMeterData  
  
 # Use the calib file from the day before the force dateval (i.e. the  
 # date closest to dateval-1)  
 tempcalibs = calibCoeffs[[which.min(abs(calibDates - (dateval-1)))]]  
   
 ##############################################  
# Convert raw count data to forces  
 temp2$X.N = (temp2$JOY\_X\_signal \* tempcalibs$X$slope) +   
 tempcalibs$X$intercept  
 temp2$Y.N = (temp2$JOY\_Y\_signal \* tempcalibs$Y$slope) +   
 tempcalibs$Y$intercept  
 temp2$Z.N = (temp2$BEAM\_Z\_signal \* tempcalibs$Z$slope) +   
 tempcalibs$Z$intercept  
 ##################  
# Calculate an offset from 'zero' for each axis using the first few samples  
# where there is presumably no force being exerted  
 xoffset = mean(temp2$X.N[1:50])  
 temp2$X.N = temp2$X.N - xoffset  
 yoffset = mean(temp2$Y.N[1:50])  
 temp2$Y.N = temp2$Y.N - yoffset  
 zoffset = mean(temp2$Z.N[1:50])  
 temp2$Z.N = temp2$Z.N - zoffset   
# Take the focal time match and write it into the new columns of the events2   
 # data frame.  
 events2[i,'X.N'] = temp2$X.N[101]  
 events2[i,'Y.N'] = temp2$Y.N[101]  
 events2[i,'Z.N'] = temp2$Z.N[101]  
   
# Also generate a list containing an entry for each event that will hold the  
 # relevant time points  
 if (i == 1){  
 forceList = list()  
 }   
 forceList[[i]] = temp2  
 forceList[[i]]$Event = events[i,'Event']   
 # Calculate the euclidean norm of all three axes forces  
 xyz = as.matrix(forceList[[i]][,c('X.N','Y.N','Z.N')])  
 for (k in 1:nrow(xyz)){  
 # norm() function computes the norm. Needs to work on one row of data at  
 # a time  
 forceList[[i]]$Norm[k] = norm(xyz[k,],"2")   
 }  
   
}   
# Save output  
# write.csv(events2,file='Pound\_ForceData\_withXYZ\_forces.csv',row.names=FALSE)

## DEFINTIONS

* A given force can only be assigned to one category (cannot be a push and peck)
* Forces discussed below refer to Euclidean forces

### Peck:

* Magnitude: ≥ 2N and MUST BE greater than or equal to two times the intensity of the force data points immediately before AND after a given point.
* Duration: Force MUST BE sustained for ≤ 2 consecutive milliseconds
* Both magnitude and duration conditions must be met in order for a force to be categorized as a peck

### Push:

* Magnitude: ≥ 2N and may or may not be greater than or equal to two times the intensity of the force before and after a given force data point
* Duration: May be sustained for any amount of time
* Forces categorized as pushes do not meet conditions of Peck and Touch

### Touch:

* Magnitude: < 2N (e.g. the bird seems to be playing with the ‘limpet’ mimic rather than actively attempting to remove it)
* We are not considering ‘Touches’ in our force meter analyses

I wrote an interactive function to let the user pick a trial date and go through the raw time series data and attempt to identify potential pecks and pushes. The script has the user first identify a period of at least 10 samples (100 milliseconds) where the baseline signal appears stable, and then the user clicks on each peak above the 2N threshold outlined above. Because the sensors tended to drift and take new baseline values, particularly after big hits by the bird, the user is identifying a new baseline for each peak or group of peaks that happen in quick succession and appear to share a common baseline value. The output of the function peakChooser() is a data frame that contains a row from the original forces data frame for each identified peak. As a result, the raw X, Y, Z values, time in milliseconds, and calibrated forces in the X, Y, and Z axes (plus the euclidean norm of all 3 axes) are available for each peak event, along with the starting time value for the baseline. These data could be used to relocate a chunk of the raw timeseries around each peak event for further analysis.

# Code to run through a data set and have the user identify probable push or   
# peck events.   
# All raw data are in the 'forces' data frame  
  
# Step through rows of 'forces' data frame, plot a chunk of time vs Norm, and  
# have user select any events that look like pecks or pushes.   
  
# Since 'forces' data frame only has raw signals, we need to apply the   
# calibration values to each chunk of data to convert to Newtons and calculate  
# the norm.   
  
library(RANN) # for nn2 nearest neighbor search function  
  
baseLength = 8  
# Inputs  
# trialDate - A Date object giving the date of the bird trial  
# forces - a dataframe containing all of the trial data from each day, imported  
# earlier  
# calibs - a list object containing the calibration slopes associated from   
# each trial date. Created earlier.  
# eventsdf - an optional data frame created from the output of this function. If  
# supplied, this will be used to plot markers on already-identified peaks in the  
# chosen day's timeseries.   
# baseLength - the number of samples to be averaged together to form the baseline  
# values for X, Y, and Z axes prior to an identified peak. 8 samples at 10ms   
# interval = 80 milliseconds  
   
peakChooser = function(trialDate, forces = forces, calibs = calibCoeffs,   
 eventsdf = NULL, baseLength=8){  
 require(RANN)  
 if (!is.null(eventsdf)){  
 peakPushEvents = eventsdf  
 }  
   
 # Extract force data for the current date  
# dat = forces[forces$Date == myDates[dayDate],]  
 dat = forces[forces$Date == trialDate,]  
 nsamps = nrow(dat)  
   
 baseLength = baseLength - 1 # take off 1 since we always add to baseLocation  
 # Get the relevant calibration data from calibCoeffs list  
  
   
# tempcalibs = calibCoeffs[[which.min(abs(calibDates-(myDates[dayDate]-1)))]]  
 tempcalibs = calibCoeffs[[which.min(abs(calibDates-(trialDate-1)))]]  
   
 # Generate estimates of force (N) on each axis using the calibration data.   
 # Note however that these force values will have an offset due to the drift  
 # in the baseline value of the transducers. So for each chosen peak, we will  
 # first identify a preceding baseline set of "0" values and correct these  
 # force values by that baseline offset.   
 dat$X.N = (dat$JOY\_X\_signal \* tempcalibs$X$slope) +   
 tempcalibs$X$intercept  
 dat$Y.N = (dat$JOY\_Y\_signal \* tempcalibs$Y$slope) +   
 tempcalibs$Y$intercept  
 dat$Z.N = (dat$BEAM\_Z\_signal \* tempcalibs$Z$slope) +   
 tempcalibs$Z$intercept  
  
 # Make a plot of raw sample data to have user select a baseline time period  
 repeatLoop = TRUE  
 for (j in seq(1,nsamps, by = 1000)){  
 while(repeatLoop == TRUE){  
 # Create the initial wide-angle plot showing 1000 samples  
 # Calculate an offset for JOY\_Y signal so that it's close to   
 # JOY\_X data on the y-axis for plotting purposes   
 offY = (dat[1,'JOY\_X\_signal']-dat[1,'JOY\_Y\_signal'])  
 ylims = range(dat[,'JOY\_X\_signal'])  
 plot(dat$Time.msec[j:(j+999)],y = dat[j:(j+999),'JOY\_X\_signal'],  
 type='l',  
 ylim = ylims, xlab = 'Time, ms', yaxs='i',   
 ylab = paste0('Raw axis signal'),   
 main = trialDate)  
 # Add the offset Y-axis values to the plot  
 lines(x = dat$Time.msec[j:(j+999)],  
 y = dat[j:(j+999),'JOY\_Y\_signal']+offY,  
 col = 'blue')  
 # Check and see if peakPushEvents data frame exists  
 if (exists('peakPushEvents')){  
 # If it exists, find any events that might already be   
 # identified in the current plot  
 peakPushSub = peakPushEvents[which(peakPushEvents$Date ==   
 unique(dat$Date)),]  
 # Subset only rows that are within the current time range  
 peakPushSub = peakPushSub[which( (peakPushSub$Time.msec >   
 dat$Time.msec[j]) &   
 (peakPushSub$Time.msec <   
 dat$Time.msec[j+999])),]  
 # Plot previously identified peaks on the current plot  
 points(x=peakPushSub$Time.msec,   
 y = peakPushSub[,'JOY\_X\_signal'],  
 pch = 20, col = 'red')  
 # Plot previously identified peaks on the current plot  
 points(x=peakPushSub$Time.msec,   
 y = peakPushSub[,'JOY\_Y\_signal']+offY,  
 pch = 20, col = 'skyblue')  
 }  
 par(xpd = TRUE)  
 # Draw button for skipping ahead to next section  
 dist = (par()$usr[2]-par()$usr[1]) \* 0.2  
 bluelocsx = c(par()$usr[2]-dist, par()$usr[2],par()$usr[2],  
 par()$usr[2]-dist)  
 bluelocsy = c(ylims[2],ylims[2],ylims[2]+100,ylims[2]+100)  
 polygon(bluelocsx,bluelocsy,col='lightblue')  
 text(x = par()$usr[2],y = par()$usr[4], labels='skip ahead',  
 adj = c(1.5,-1.5))  
 # Put message on plot  
 text(x = par()$usr[1], y = ylims[2],  
 labels = 'Click on peak to zoom in\n or click blue button\n to skip ahead',  
 adj = c(0,-0.5))  
 # Have the user choose either a button or peak to zoom in on  
 clickLocation = locator(n = 1)  
 # Handle the click  
 if (clickLocation$x > bluelocsx[1] &   
 clickLocation$x < bluelocsx[2] &  
 clickLocation$y > bluelocsy[1] &   
 clickLocation$y < bluelocsy[3]) {  
 # User clicked in the skip ahead box  
 print("skip ahead")  
 skip = TRUE  
 zoom = FALSE  
 } else if (clickLocation$x > par()$usr[1] &  
 clickLocation$x < par()$usr[2] &   
 clickLocation$y > par()$usr[3] &   
 clickLocation$y < par()$usr[4]) {  
 # User clicked somewhere else in the figure, decide if it was   
 # a value location to click  
 print('click in plot')  
 points(x = clickLocation$x, y = clickLocation$y, col = 'red',  
 pch = 20, cex = 1.5)  
 zoom = TRUE  
 skip = FALSE  
 } else {  
 # User clicked outside the plot or buttons, so trigger a re-plot  
 print('click not in field')  
 clickLocation  
 zoom = FALSE  
 skip = FALSE   
 }  
   
 if (zoom == FALSE & skip == TRUE){  
 # cycle main loop again to move to next chunk of time   
 break # break out of while loop  
 } else if (zoom == TRUE & skip == FALSE) {   
 ##################################################  
 # Create the 2nd plot (the initial zoomed-in plot)  
 # Zoom in for the next phase of peak choosing  
 # Find the row of dat closest to the click location  
 clickRow = which.min(abs(clickLocation$x - dat$Time.msec))  
 # subset a new data frame centered around the clickRow  
 temp = dat[(clickRow-100):(clickRow+100),]  
   
 # Now the user will choose a time point that represents the  
 # baseline force value for upcoming peaks  
 ylims = range(c(temp[,'JOY\_X\_signal'],  
 temp[,'JOY\_Y\_signal']+offY))  
 xlims = range(temp$Time.msec)  
 plot(temp$Time.msec,  
 y = temp[,'JOY\_X\_signal'],  
 type = 'l',   
 ylab = paste0('Raw axis signal'),   
 xlab = 'Time, ms',  
 ylim = ylims)  
 lines(temp$Time.msec,  
 y = temp[,'JOY\_Y\_signal']+offY, col = 'blue')  
 points(temp$Time.msec,  
 y = temp[,'JOY\_X\_signal'], col = 'black',  
 pch = 20, cex = 0.8)  
 points(temp$Time.msec,  
 y = temp[,'JOY\_Y\_signal']+offY, col = 'blue',  
 pch = 20, cex = 0.8)  
 if (exists('peakPushEvents')){  
 # If it exists, find any events that might already be   
 # identified in the current plot  
 peakPushSub = peakPushEvents[which(peakPushEvents$Date ==   
 unique(dat$Date)),]  
 # Subset only rows that are within the current time range  
 peakPushSub = peakPushSub[which( (peakPushSub$Time.msec >   
 temp$Time.msec[1]) &   
 (peakPushSub$Time.msec <   
 temp$Time.msec[nrow(temp)])),]  
 # Plot previously identified peaks on the current plot  
 points(x=peakPushSub$Time.msec,   
 y = peakPushSub[,'JOY\_X\_signal'],  
 pch = 20, col = 'red')  
 points(x=peakPushSub$Time.msec,   
 y = peakPushSub[,'JOY\_Y\_signal']+offY,  
 pch = 20, col = 'magenta')  
 }  
 par(xpd=TRUE)  
 # Plot a button for user to skip ahead  
 dist = (par()$usr[2]-par()$usr[1]) \* 0.2  
 bluelocsx = c(par()$usr[2]-dist, par()$usr[2],par()$usr[2],  
 par()$usr[2]-dist)  
 bluelocsy = c(par()$usr[4],par()$usr[4],ylims[2]+100,ylims[2]+100)  
 polygon(bluelocsx,bluelocsy,col='lightblue')  
 text(x = par()$usr[2],y = par()$usr[4],   
 labels='Skip to next section', adj = c(-0.1,-1))  
 text(x=par()$usr[1], y = par()$usr[4],  
 labels='Click a baseline point or hit skip',   
 adj = c(0,-0.3))  
 # The user should click on a baseline point now.   
 clickLocation = locator(n = 1)  
 if (clickLocation$x > bluelocsx[1] &   
 clickLocation$x < bluelocsx[2] &  
 clickLocation$y > bluelocsy[1] &   
 clickLocation$y < bluelocsy[3]) {  
 # User clicked in box to skip ahead  
 # Set zoom and skip so that the wide-angle plot will still  
 # show the same range of time as the previous wide plot  
 zoom = FALSE  
 skip = FALSE  
 break  
 } else if (clickLocation$x > par()$usr[1] &  
 clickLocation$x < par()$usr[2] &   
 clickLocation$y > par()$usr[3] &   
 clickLocation$y < par()$usr[4]) {  
 # User clicked on baseline plot, find nearest row  
 baseLocRow = which.min(abs(clickLocation$x -   
 temp$Time.msec))  
 # Use the chosen row value to work out a baseline value  
 # to use with the calibrations  
 }  
   
 ##########################  
 # With a baseline location in hand, calculate calibrated force  
 # values and replot the norm'd forces  
 # Calculate an offset from 'zero' for each axis using the first   
 # few samples where there is presumably no force being exerted  
 xoffset = mean(temp$X.N[baseLocRow:(baseLocRow+baseLength)])  
 temp$X.N.off = temp$X.N - xoffset  
 yoffset = mean(temp$Y.N[baseLocRow:(baseLocRow+baseLength)])  
 temp$Y.N.off = temp$Y.N - yoffset  
 zoffset = mean(temp$Z.N[baseLocRow:(baseLocRow+baseLength)])  
 temp$Z.N.off = temp$Z.N - zoffset   
 # Calculate euclidean norm force  
 for (r in 1:nrow(temp)){  
 xyz = as.matrix(temp[r,c('X.N.off','Y.N.off','Z.N.off')])  
 temp$Norm[r] = norm(xyz,'2')  
 }  
 ################################################################  
 # Create the 3rd plot, showing Norm'd force  
 # Have user identify peaks here to be classified as peak/push  
 xlims = range(temp$Time.msec)  
 ylims = range(temp$Norm)  
 plot(x = temp$Time.msec, y = temp$Norm,type='b',   
 xlab = 'Time, msec', ylab = 'Norm, Newtons', las = 1)  
 if (exists('peakPushEvents')){  
 # If it exists, find any events that might already be   
 # identified in the current plot  
 peakPushSub = peakPushEvents[which(peakPushEvents$Date ==   
 unique(dat$Date)),]  
 # Subset only rows that are within the current time range  
 peakPushSub = peakPushSub[which( (peakPushSub$Time.msec >   
 temp$Time.msec[1]) &   
 (peakPushSub$Time.msec <   
 temp$Time.msec[nrow(temp)])),]  
 # Plot previously identified peaks on the current plot  
 points(x=peakPushSub$Time.msec, y = peakPushSub$Norm,  
 pch = 20, col = 'red')  
 }  
 # Plot the chosen baseline points  
 points(x = temp$Time.msec[baseLocRow:(baseLocRow+baseLength)],   
 y = temp$Norm[baseLocRow:(baseLocRow+baseLength)],   
 pch = 20, col = 'orange')  
 text(x = temp$Time.msec[baseLocRow],   
 y = temp$Norm[baseLocRow]+1, labels = 'Baseline',  
 adj=c(0,0.7))  
 par(xpd=FALSE)  
 grid(col = 'grey70',lty = 2)  
 abline(h = 2, lty = 2, col = 'red')  
 par(xpd=TRUE)  
 # Add a box to let the user declare they are done choosing  
 dist = (par()$usr[2]-par()$usr[1]) \* 0.2  
 bluelocsx = c(par()$usr[2]-dist, par()$usr[2],par()$usr[2],  
 par()$usr[2]-dist)  
 bluelocsy = c(par()$usr[4],par()$usr[4],ylims[2]+100,ylims[2]+100)  
 polygon(bluelocsx,bluelocsy,col='lightblue')  
 text(x = par()$usr[2],y = par()$usr[4],   
 labels='Done choosing', adj = c(1.3,-1))  
 text(x = par()$usr[1],y = par()$usr[4],   
 labels = 'Click on each peak',  
 adj = c(0,-0.5))  
 keepChoosing = TRUE  
 while(keepChoosing == TRUE){  
 clickLocation = locator(n = 1) # have user select a point  
 # Draw where the user clicked  
 points(x=clickLocation$x,y=clickLocation$y,col='blue',  
 pch = 20)  
 if (clickLocation$x > bluelocsx[1] &   
 clickLocation$x < bluelocsx[2] &  
 clickLocation$y > bluelocsy[1] &   
 clickLocation$y < bluelocsy[3]) {  
 # User clicked in box to say they are done choosing  
 break # kill the while(keepChoosing == TRUE) loop  
 } else if (clickLocation$x > par()$usr[1] &  
 clickLocation$x < par()$usr[2] &   
 clickLocation$y > par()$usr[3] &   
 clickLocation$y < par()$usr[4]) {  
 # User clicked on plot, find nearest point  
 # by using a nearest neighbor search algorithm using  
 # the function nn2 from the package 'RANN'  
 # Prepare the data matrix  
 mymatrix = cbind(temp$Time.msec,temp$Norm)  
 # Rescale the x-axis data, since the results of nn2  
 # will be thrown off if the scale of x data is very   
 # different from the scale of y data  
 mymatrix[,1] = mymatrix[,1]/1000  
 mypoint = cbind(clickLocation$x,clickLocation$y)  
 # Also rescale the clickLocation x-axis value  
 mypoint[,1] = mypoint[,1]/1000  
 # Run the nearest neighbor search. The k=1 argument  
 # should return the 1st-closest nearest neighbor  
 rowIndex = RANN::nn2(data=mymatrix,query=mypoint,  
 k = 1)$nn.idx[1,1]  
 # Add the identified point to the plot  
 points(temp$Time.msec[rowIndex],y=temp$Norm[rowIndex],  
 pch = 20, col = 'royalblue')  
 # Use the chosen row value to work out a baseline value  
 # to use with the calibrations  
 keepChoosing = TRUE  
 if(!(exists('peakPushEvents'))){  
 # Get the appropriate row of info out of the  
 # original data frame dat  
 peakPushEvents = temp[rowIndex,]  
 # Also add on the Time.msec value for the baseline  
 # point  
 peakPushEvents$BaselineTime.msec = temp$Time.msec[baseLocRow]  
 } else {  
 # If peakPushEvents exists, add onto it  
 tempVals = temp[rowIndex,] # extract the peak row  
 # Add on the baseline row Time.msec value  
 tempVals$BaselineTime.msec = temp$Time.msec[baseLocRow]  
 # rbind the new data onto peakPushEvents  
 peakPushEvents = rbind(peakPushEvents,  
 tempVals)  
 }  
 keepChoosing = TRUE # allow while loop to repeat  
 } # end of if (clickLocation$x > bluelocsx[1]... section  
 } # end of while(keepChoosing == TRUE)  
  
 } else if (zoom == FALSE & skip == FALSE ){  
 # let the main plot loop repeat (plots a wide-angle view)  
 repeatLoop = TRUE   
 }  
 } # end of while(repeatLoop == TRUE)   
 }  
 cat('Finished\n')  
 # Return peakPushEvents data frame  
 peakPushEvents  
   
} # end of function

# After defining the peak choosing function in the previous chunk of code,   
# now use it to identify potential peak/push events on each trial date.   
# Run the code in this chunk manually, since it requires a lot of user   
# interaction.   
  
# Available trial dates:  
# 2016-05-18  
# 2016-05-19  
# 2016-12-14  
# 2016-12-15 # Relatively noisy signal, is the calibration reasonable?  
# 2017-01-05   
# 2017-01-06  
  
mydate = as.Date('2017-01-06')  
  
eventdf = peakChooser(trialDate = mydate, forces = forces, calibs = calibCoeffs,  
 eventsdf = NULL, baseLength = 8)  
  
# events20160518 = eventdf  
# events20160519 = eventdf  
# events20161214 = eventdf  
# events20161215 = eventdf  
# events20170105 = eventdf  
 events20170106 = eventdf  
 output = rbind(events20161214,events20161215,events20160518,  
 events20160519,events20170105,events20170106)  
  
output[,6:12] = round(output[,6:12], digits = 2)  
output= cbind(seq(1,nrow(output)),output)  
names(output)[1]= 'EventNumber'  
 write.csv(output, file = 'Events\_picker\_output\_20171211.csv',row.names=FALSE)

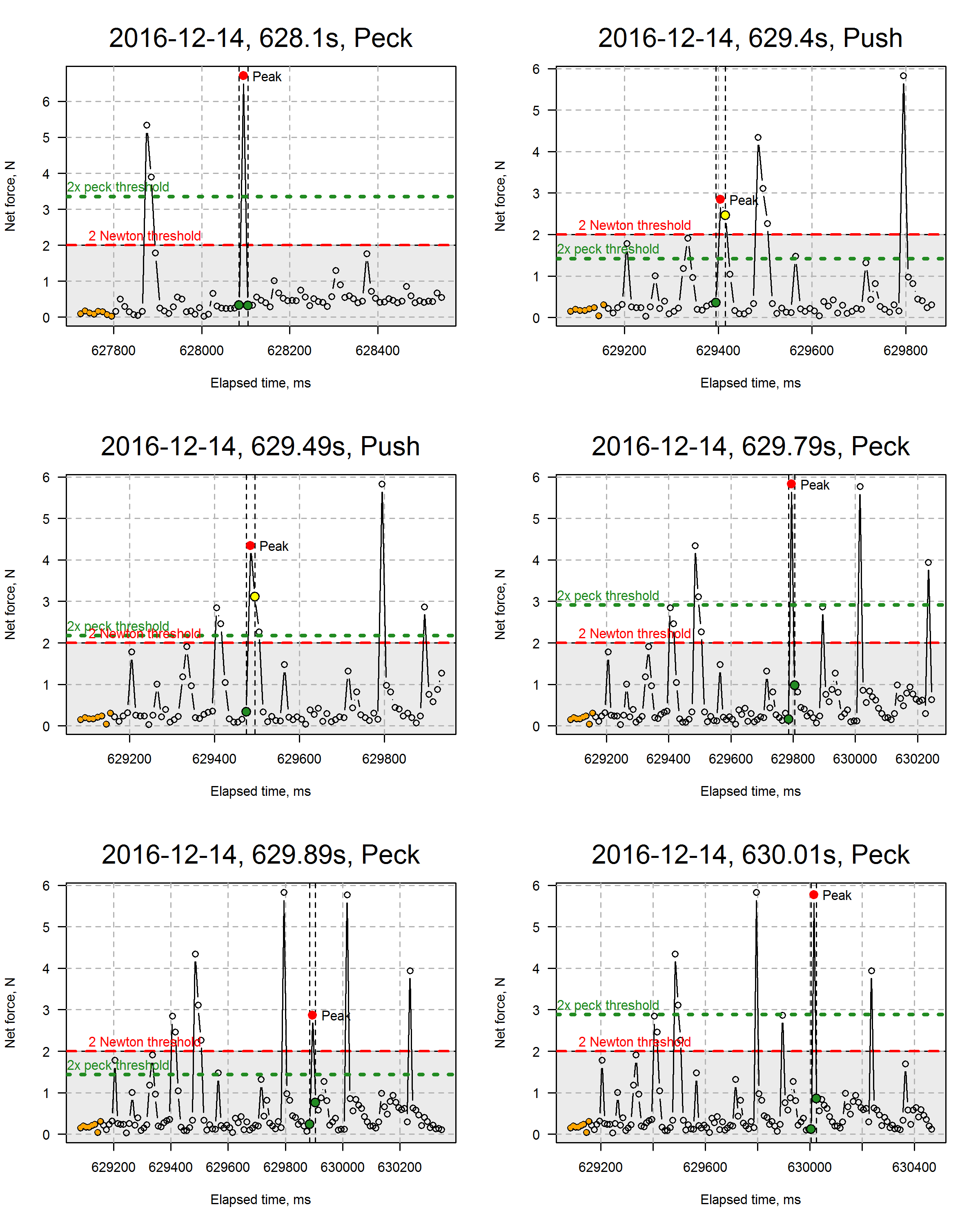
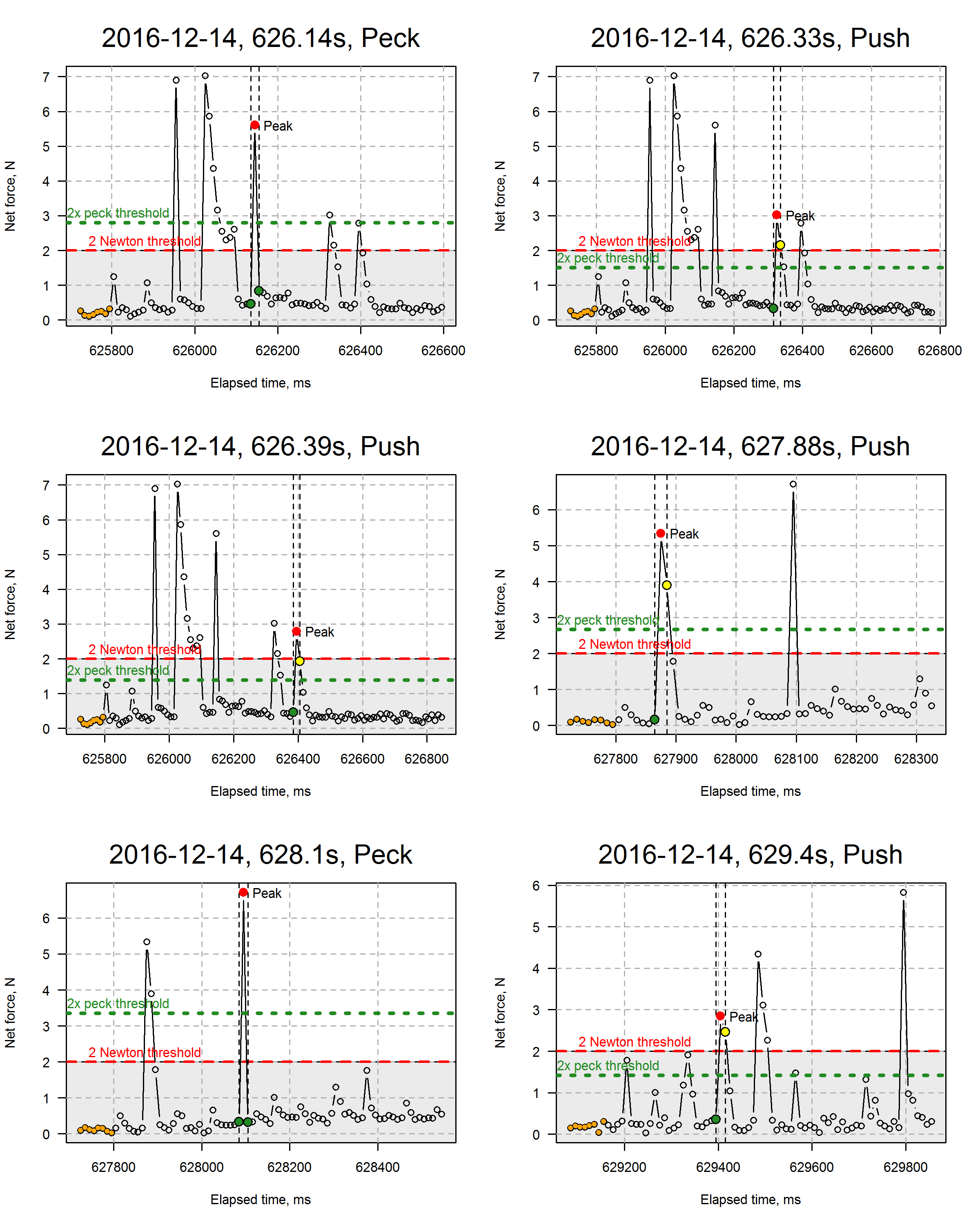
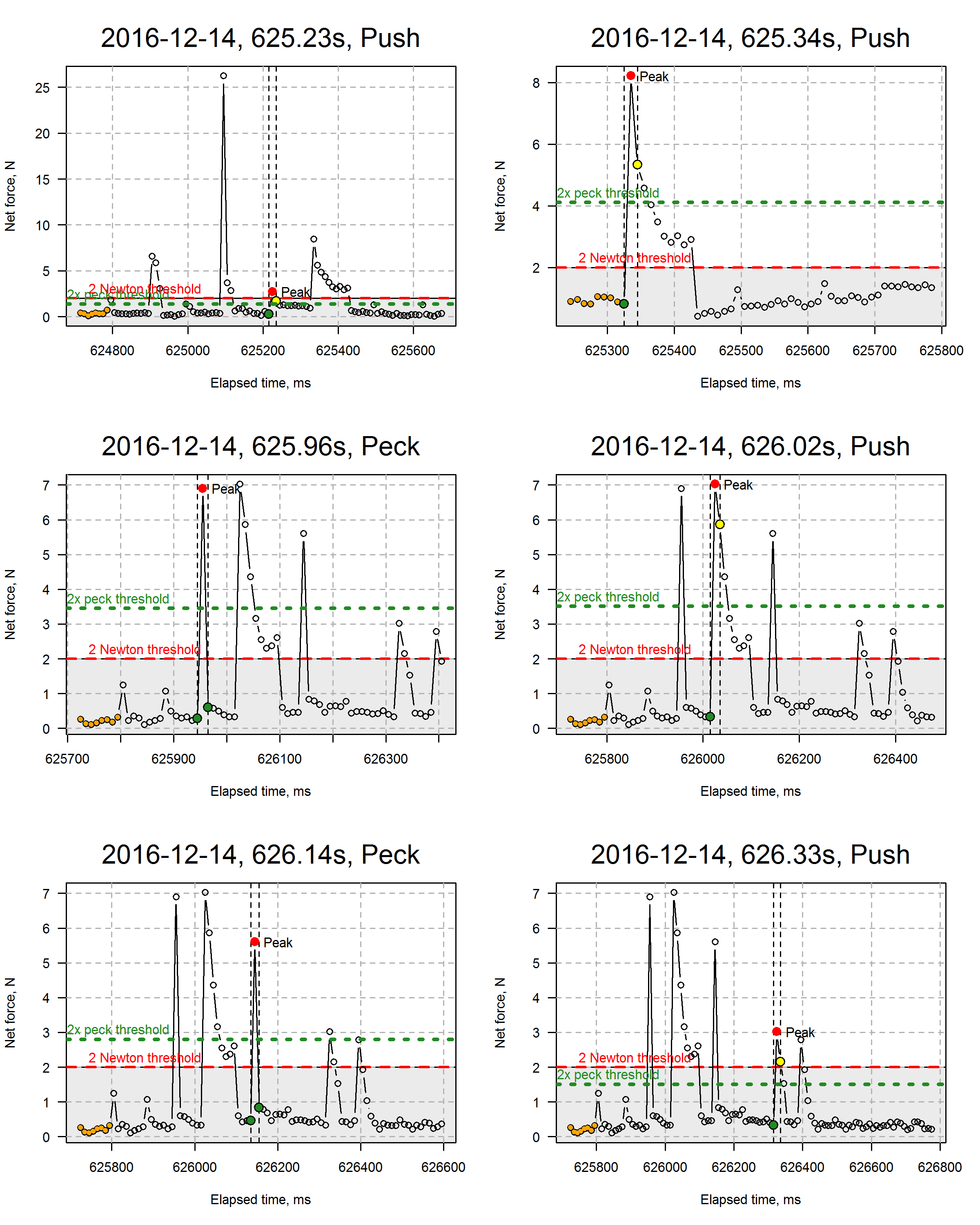
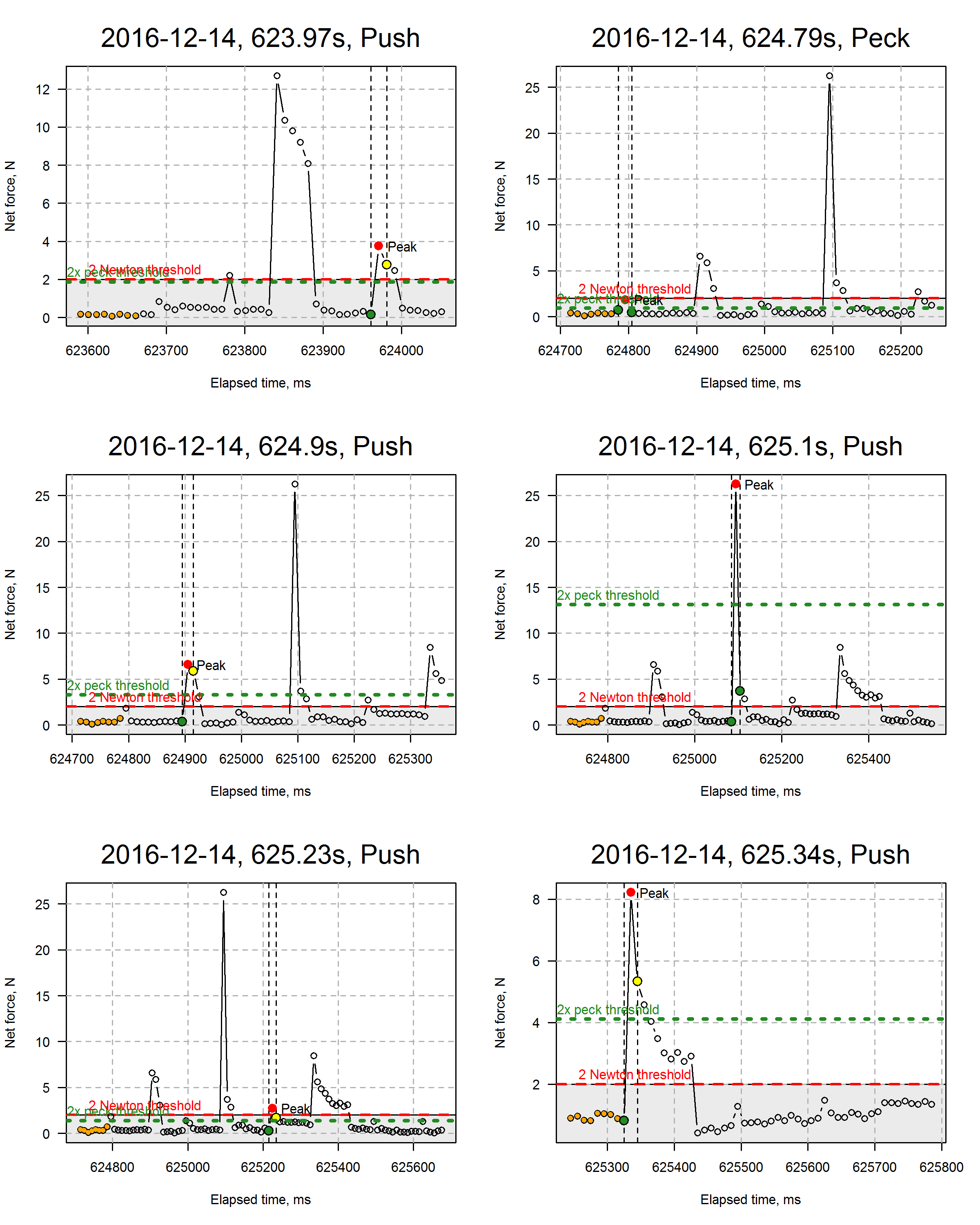
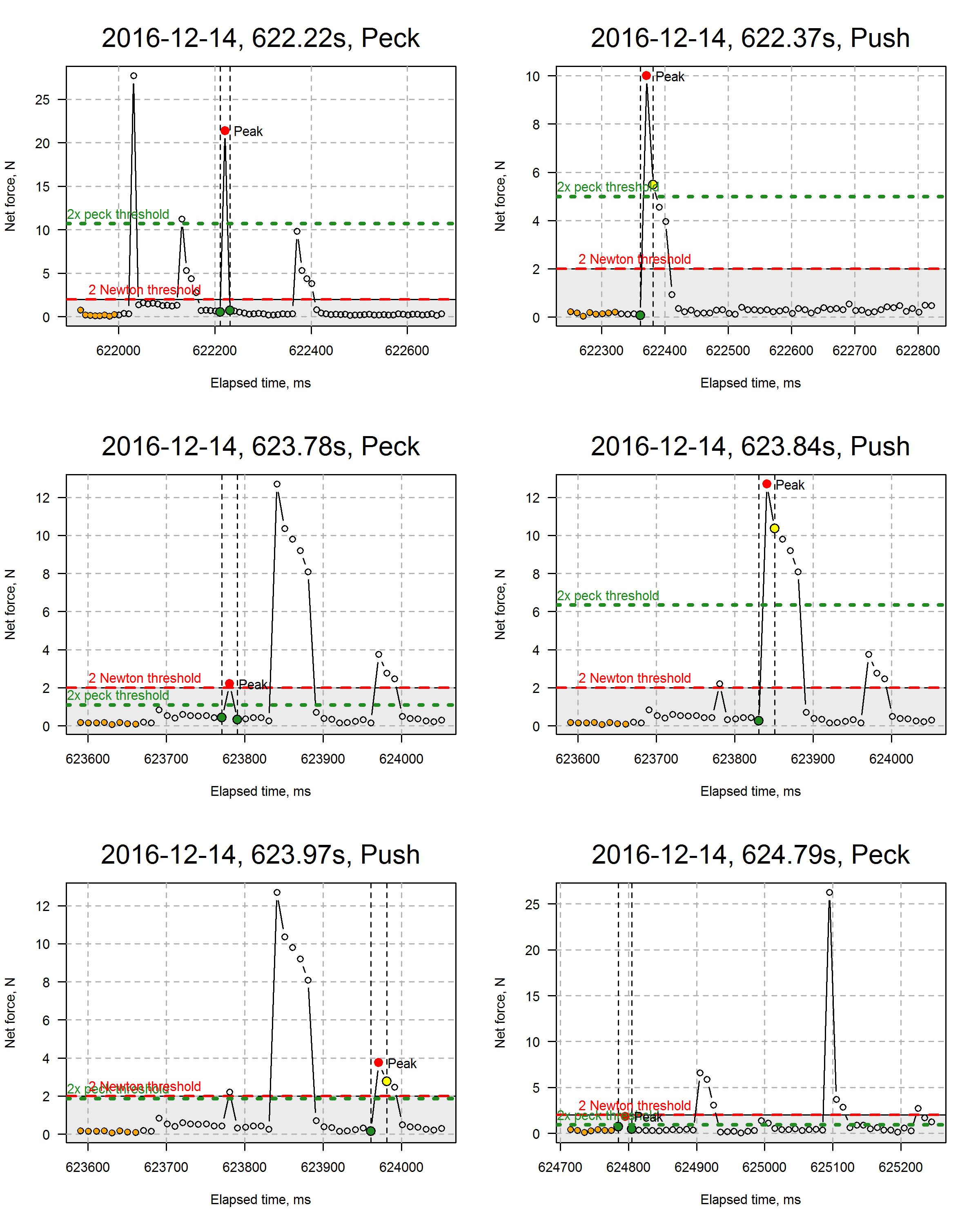
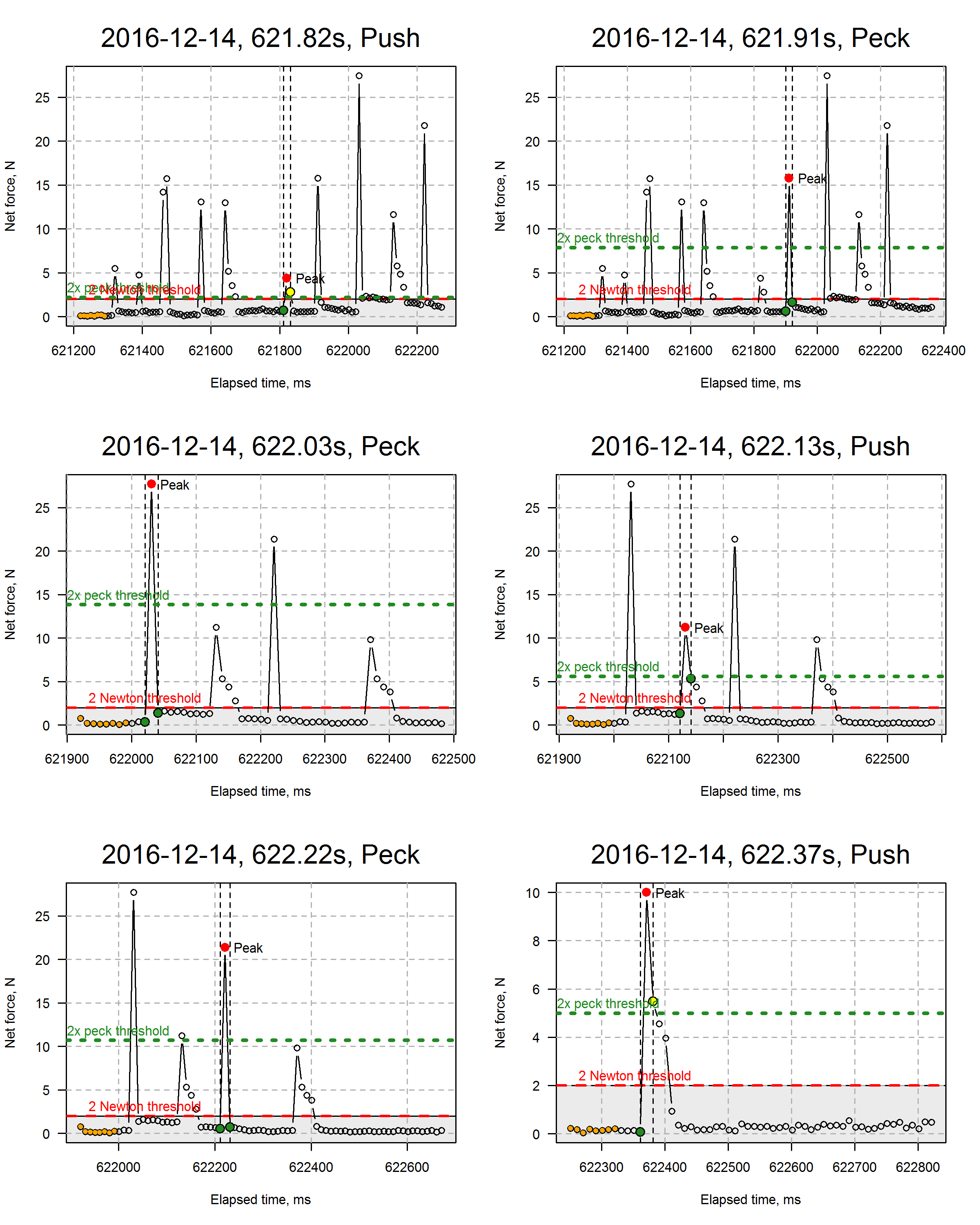
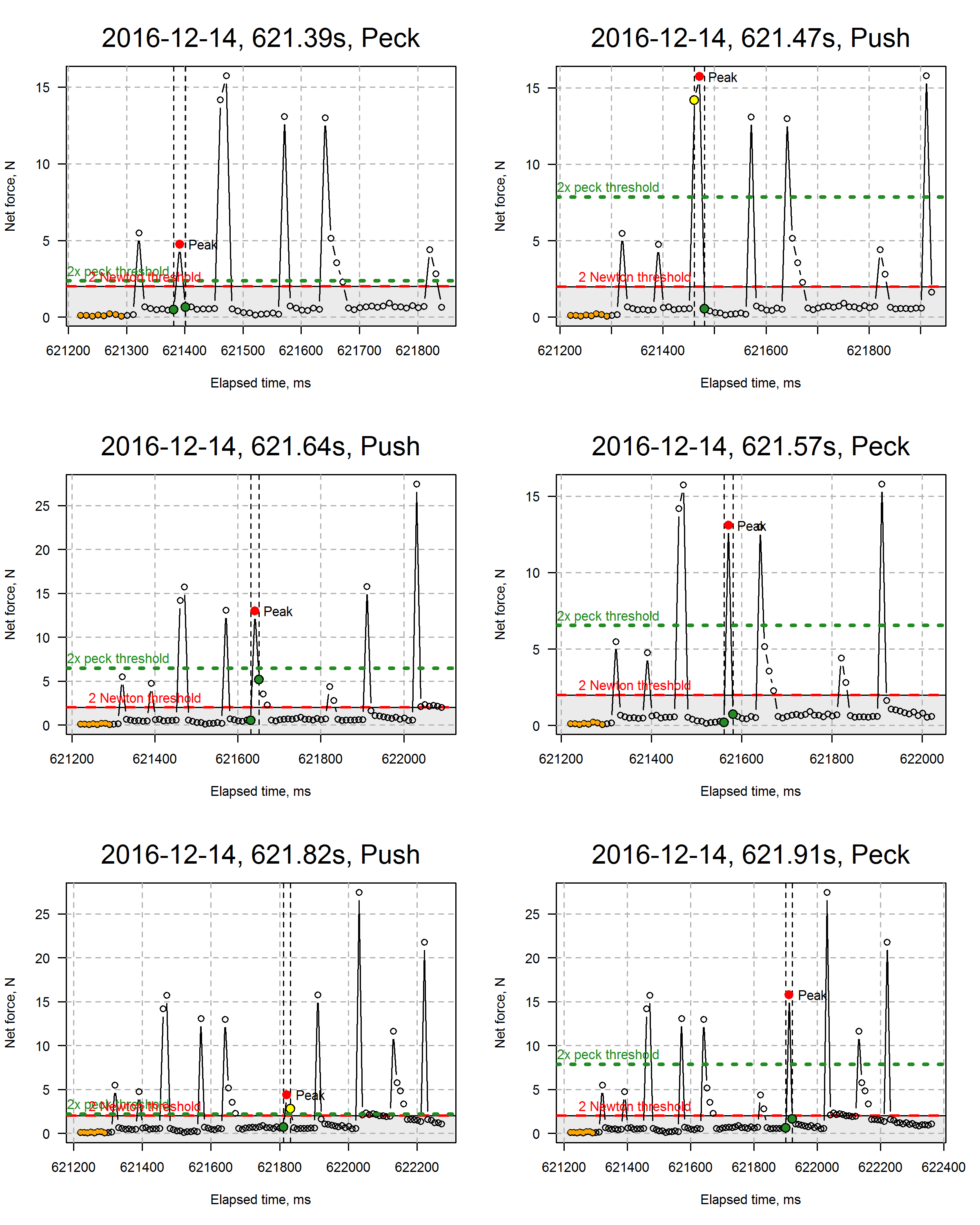
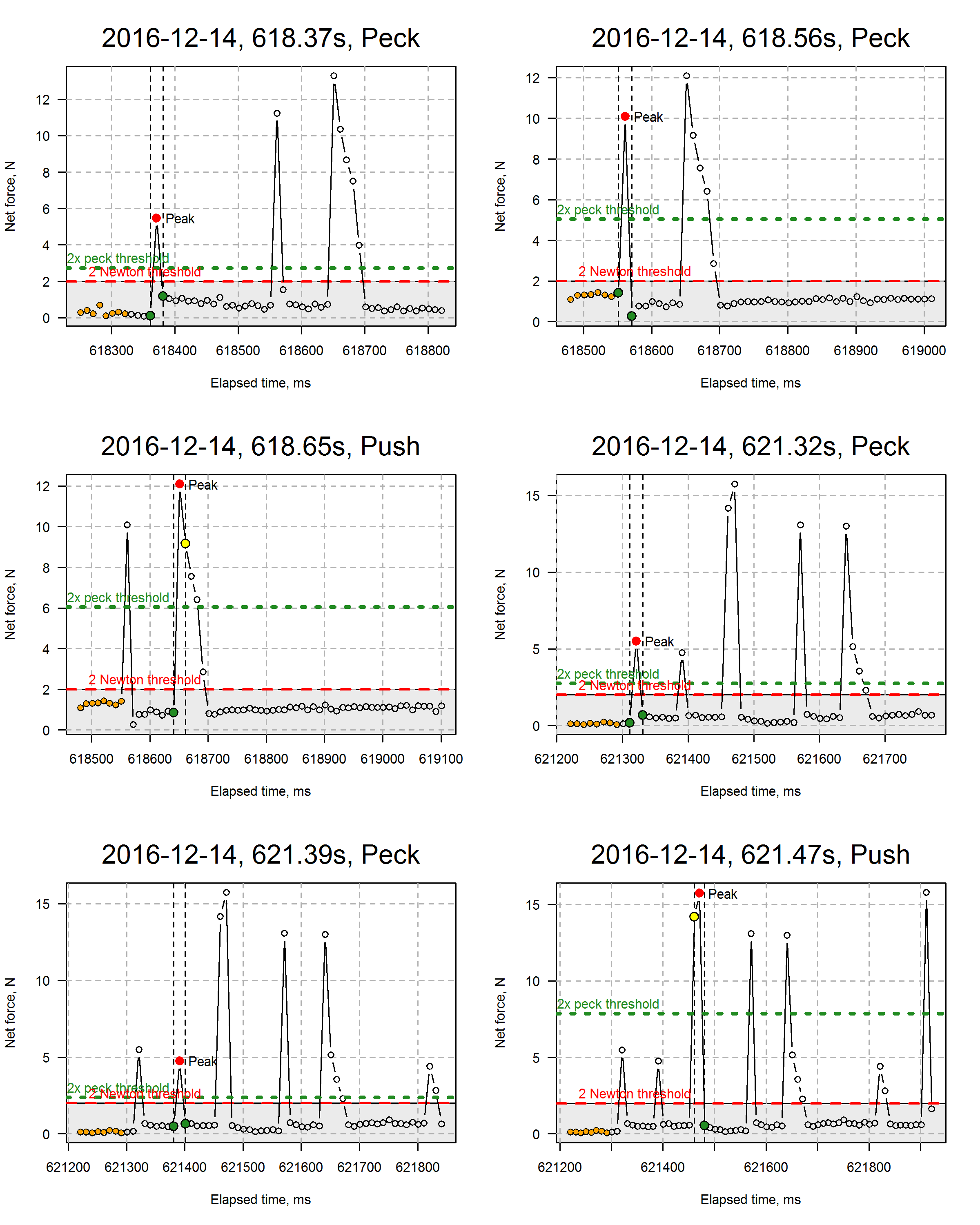
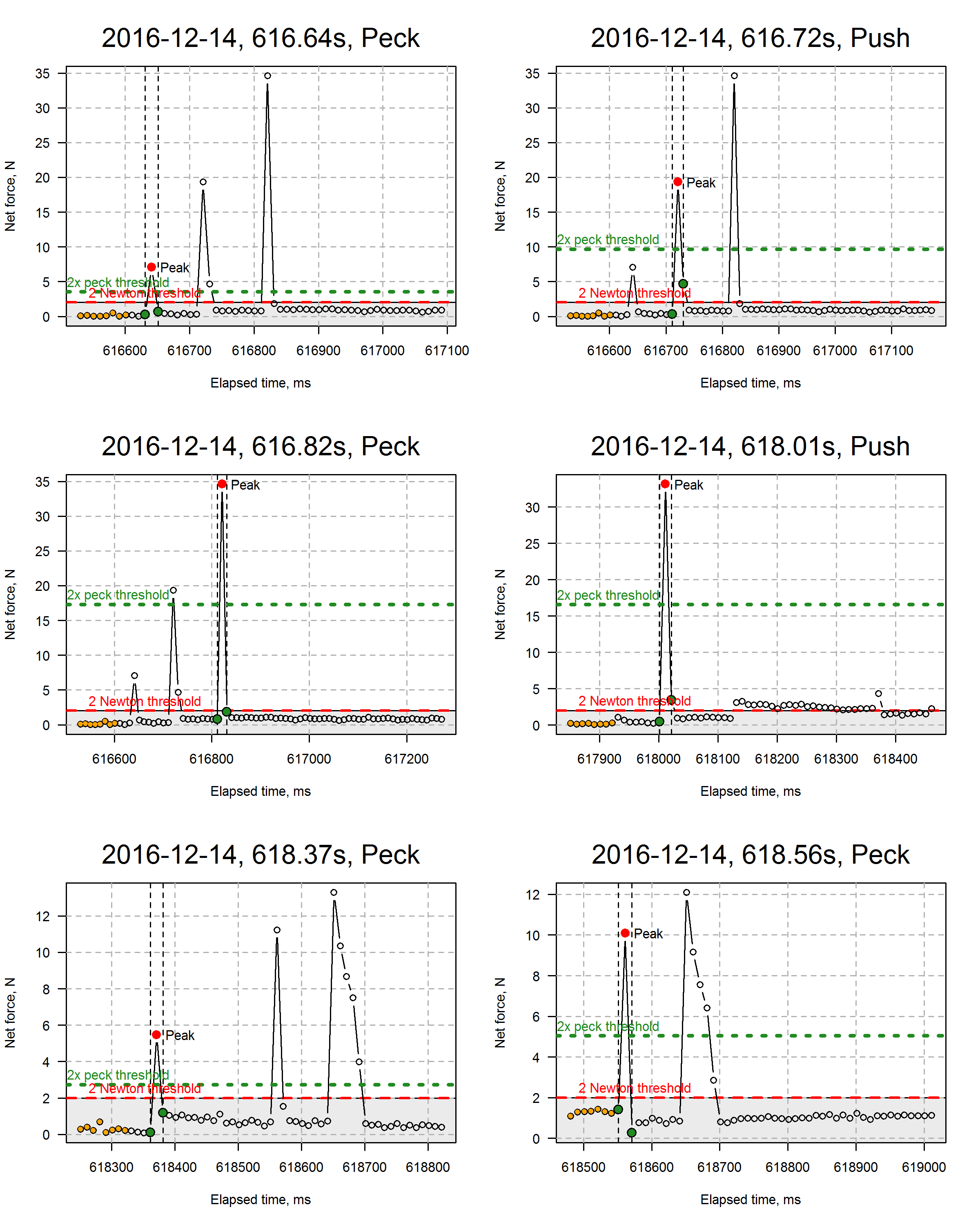
The output from the peakChooser function was saved to a csv file called Events\_picker\_output\_20171208.csv. The data in that file next need to be processed to determine if each event was a peck or a push according to the criteria outlined above.

#################################  
# Function peckPushAlgo  
#' Classify an event as a peck or push  
#' @param Norm A vector of forces (euclidean norm = 3-axis net force), units of   
#' Newtons   
#' @param peakIndex A row index into the Norm vector that marks the peak to be  
#' classified as a peck or push  
#' @param forceThreshold Numeric value specifying the minimum force (Newtons)   
#' needed for the peak to be considered a peck or push. Default = 2 Newtsons.   
#' @return A 1-element character vector containing 'Peak', 'Push', or NA, based  
#' on the results of the classification algorithm.   
  
peckPushAlgo <- function(Norm, peakIndex, forceThreshold = 2){  
# Determine if this was a peck or push  
 nearestN = max(c(Norm[peakIndex-1],Norm[peakIndex+1]))  
 if ( (Norm[peakIndex] / nearestN) > forceThreshold) {  
 # If the ratio of the peak force to its nearest neighbors is greater  
 # than 2, it meets the threshold for magnitude of a peck  
 peckThreshMagnitude = TRUE  
 } else {  
 peckThreshMagnitude = FALSE  
 }  
 if (Norm[peakIndex-1] < forceThreshold &   
 Norm[peakIndex+1] < forceThreshold) {  
 # If the time points immediately before and after the peak are   
 # both below forceThreshold (in Newtons), the peak meets the criteria   
 # for peck duration because the time above forceThreshold must not \  
 # exceed 20 ms (= 2 sample intervals)  
 peckThreshTime = TRUE  
 } else {  
 peckThreshTime = FALSE  
 }  
   
 if (peckThreshMagnitude == TRUE & peckThreshTime == TRUE){  
 peckPush = 'Peck'  
 } else if (peckThreshMagnitude != TRUE | peckThreshTime != TRUE){  
 peckPush = 'Push'  
 }  
 peckPush # return peck/push status  
}  
  
  
################################################################################  
# Go through each row in events2, and pull the relevant data from the   
# forces data frame to generate a plot of the time just before and after an   
# identified peak  
  
#' A function to go through all events and classify them as pecks or pushes  
#' @param events A data frame where each row represents an identified peak that  
#' needs to be classified as a peck or push. Contains columns 'Date',   
#' 'BaselineTime.msec', and 'Time.msec'.  
#' @param forces A data frame containing raw timeseries of transducer values  
#' @param calibs A list containing calibration data for each X, Y, Z transducer  
#' axis for each date.   
#' @param forceThreshold A numeric value denoting the minimum force necessary to   
#' be classified as a peck or push event (anything below this threshold is   
#' classified NA).   
#' @param baseLength A numeric value denoting the number of samples from the   
#' baseline timepoint to average for the calculation of the baseline force   
#' offset.   
#' @return A version of the input 'events' data frame with a column 'PeckPush'  
#' containing a classification for each event (row).   
  
peckpushClassify <- function(events, forces, calibs, forceThreshold = 2,   
 baseLength = 8){  
 # Extract the calibration dates  
 for (j in 1:length(calibCoeffs)){   
 if (j == 1){  
 calibDates = calibCoeffs[[j]]$Date   
 } else {  
 calibDates = c(calibDates,calibCoeffs[[j]]$Date)  
 }  
 }  
 # Step through each row of 'events' data frame to classify the event  
 # as a peck or push  
 for (i in 1:nrow(events)){  
 # Get date and time for this event  
 trialDate = events$Date[i]  
 basetime = events$BaselineTime.msec[i]  
 peaktime = events$Time.msec[i]  
 # Get the relevant calibration coefficients  
 tempcalibs = calibCoeffs[[which.min(abs(calibDates-(trialDate-1)))]]  
 # Get data from the forces data frame starting at basetime and extending  
 # past peaktime  
 temp = forces[forces$Date == trialDate,]  
 # Grab the relevant chunk of data  
 chunk = temp[ (temp$Time.msec >= basetime) &   
 (temp$Time.msec < (peaktime+500)),]  
 peakIndex = which.min(abs(chunk$Time.msec - peaktime))  
 # Convert to force, Newtons  
 chunk$X.N = (chunk$JOY\_X\_signal \* tempcalibs$X$slope) +   
 tempcalibs$X$intercept  
 chunk$Y.N = (chunk$JOY\_Y\_signal \* tempcalibs$Y$slope) +   
 tempcalibs$Y$intercept  
 chunk$Z.N = (chunk$BEAM\_Z\_signal \* tempcalibs$Z$slope) +   
 tempcalibs$Z$intercept  
 # Apply offset based on the average of the baseline values (8 samples)  
 xoffset = mean(chunk$X.N[1:baseLength])  
 chunk$X.N.off = chunk$X.N - xoffset  
 yoffset = mean(chunk$Y.N[1:baseLength])  
 chunk$Y.N.off = chunk$Y.N - yoffset  
 zoffset = mean(chunk$Z.N[1:baseLength])  
 chunk$Z.N.off = chunk$Z.N - zoffset   
 # Calculate euclidean norm force  
 for (r in 1:nrow(chunk)){  
 xyz = as.matrix(chunk[r,c('X.N.off','Y.N.off','Z.N.off')])  
 chunk$Norm[r] = norm(xyz,'2')  
 }  
 # Call the peckPushAlgo() function to classify the event  
 events$PeckPush[i] = peckPushAlgo(chunk$Norm, peakIndex = peakIndex,  
 forceThreshold = forceThreshold)  
   
 # Estimate the duration of the event (time with force above 2N thresh)  
 # A single sample peak above the peakThreshold will end up with a   
 # minimum duration of 20 milliseconds, since that is the interval between  
 # two successive sample intervals   
 # (i.e. going from 0N to >2N back to 0N = at least 20 ms, when the sample  
# interval was 10 ms. )  
 if (!is.na(events$PeckPush[i])){  
 stillHigh = TRUE  
 move = 1  
 duration = 0  
 # Check values before peak  
 while (stillHigh){  
 if( (chunk$Norm[peakIndex-move] > peakThreshold) ){  
 duration = duration + 10  
 move = move + 1  
 } else {  
 duration = duration + 10 # add 10 ms for sample interval  
 startIndex = peakIndex-move # start index of this event  
 break  
 }   
 }  
 move = 1 # reset for next check  
 # Check values after peak  
 while (stillHigh){  
 if( (chunk$Norm[peakIndex+move] > peakThreshold) ){  
 duration = duration + 10  
 move = move + 1  
 } else {  
 duration = duration + 10 # add 10 ms for last sample interval  
 endIndex = peakIndex+move # end index of this event  
 break  
 }  
 }  
 # Write results to columns in output data frame  
 events$Duration.msec[i] = duration  
 events$StartPeak.msec[i] = chunk$Time.msec[startIndex]  
 events$EndPeak.msec[i] = chunk$Time.msec[endIndex]  
 } else {  
 # If the event was not a peck or push (was NA instead)  
 events$Duration.msec[i] = NA  
 events$StartPeak.msec[i] = NA  
 events$EndPeak.msec[i] = NA  
 }  
 }   
 events # return the events data frame as output  
}  
  
  
# Function to plot a given event  
plotpeckpush = function(eventNum, events, forces, calibs, forceThreshold = 2,  
 baseLength = 8){  
 for (j in 1:length(calibCoeffs)){  
 if (j == 1){  
 calibDates = calibCoeffs[[j]]$Date   
 } else {  
 calibDates = c(calibDates,calibCoeffs[[j]]$Date)  
 }  
 }  
   
 i = eventNum  
 # Get date and time for this event  
 trialDate = events$Date[i]  
 basetime = events$BaselineTime.msec[i]  
 peaktime = events$Time.msec[i]  
 # Get the relevant calibration coefficients  
 tempcalibs = calibCoeffs[[which.min(abs(calibDates-(trialDate-1)))]]  
 # Get data from the forces data frame starting at basetime and extending  
 # past peaktime  
 temp = forces[forces$Date == trialDate,]  
 # Grab the relevant chunk of data  
 chunk = temp[ (temp$Time.msec >= basetime) &   
 (temp$Time.msec < (peaktime+460)),]  
 peakIndex = which.min(abs(chunk$Time.msec - peaktime))  
 # Convert to force, Newtons  
 chunk$X.N = (chunk$JOY\_X\_signal \* tempcalibs$X$slope) +   
 tempcalibs$X$intercept  
 chunk$Y.N = (chunk$JOY\_Y\_signal \* tempcalibs$Y$slope) +   
 tempcalibs$Y$intercept  
 chunk$Z.N = (chunk$BEAM\_Z\_signal \* tempcalibs$Z$slope) +   
 tempcalibs$Z$intercept  
 # Apply offset based on the average of the baseline values (10 samples)  
 xoffset = mean(chunk$X.N[1:10])  
 chunk$X.N.off = chunk$X.N - xoffset  
 yoffset = mean(chunk$Y.N[1:10])  
 chunk$Y.N.off = chunk$Y.N - yoffset  
 zoffset = mean(chunk$Z.N[1:10])   
 chunk$Z.N.off = chunk$Z.N - zoffset   
 # Calculate euclidean norm force  
 for (r in 1:nrow(chunk)){  
 xyz = as.matrix(chunk[r,c('X.N.off','Y.N.off','Z.N.off')])  
 chunk$Norm[r] = norm(xyz,'2')  
 }  
   
 # Generate a plot of the data  
 plot(x = chunk$Time.msec, y = chunk$Norm,   
 type = 'b',  
 xlab = 'Elapsed time, ms', ylab = 'Net force, N', las = 1)  
 par(xpd=FALSE)  
   
 rect(par()$usr[1],par()$usr[3],par()$usr[2],2, col = rgb(.9,.9,.9,.8))  
 grid(lty = 2, col = 'grey70')  
 abline(h = 2, lty = 2, col = 'red', lwd = 2)  
 # replot the force  
 lines(x = chunk$Time.msec, y = chunk$Norm, type = 'b')  
 # Draw 2 vertical lines on either side of the peak  
 abline(v = chunk$Time.msec[peakIndex-1], lty = 2, col = 'black')  
 abline(v = chunk$Time.msec[peakIndex+1], lty = 2, col = 'black')  
 # Plot the baseline points  
 points(x = chunk$Time.msec[1:baseLength], y = chunk$Norm[1:baseLength],   
 pch = 20,  
 col = 'orange')  
# text(x = chunk$Time.msec[1], y = chunk$Norm[1],   
# labels = 'Baseline', adj=c(0,-2))  
 # Plot a point at the identified peak  
 points(x = peaktime, y = chunk$Norm[peakIndex], pch = 20, cex = 2,   
 col='red')  
 # Calculate the threshold value that is 1/2 of the peak value  
 halfThresh = 0.5 \* chunk$Norm[peakIndex]  
 # Plot points at the neighboring 2 samples (before and after)  
 points(x = chunk$Time.msec[peakIndex-1], y = chunk$Norm[peakIndex-1],  
 pch = 21, bg = ifelse(chunk$Norm[peakIndex-1] > halfThresh,  
 'yellow','forestgreen'), cex = 1.5)  
 points(x = chunk$Time.msec[peakIndex+1], y = chunk$Norm[peakIndex+1],  
 pch = 21, bg = ifelse(chunk$Norm[peakIndex+1] > halfThresh,  
 'yellow','forestgreen'), cex = 1.5)  
 # Draw a line at half the height of the peak  
 abline(h = halfThresh, col = 'forestgreen', lty = 3, lwd = 3)  
# text(x = par()$usr[1], y = halfThresh,   
# labels = '2x peck threshold, neighboring points must be below this line',  
# adj = c(-0.01,-0.5), col = 'forestgreen')  
 text(x = par()$usr[1], y = halfThresh,   
 labels = '2x peck threshold',  
 adj = c(-0.01,-0.5), col = 'forestgreen')  
 text(x = par()$usr[1], y = 2,   
 labels = '2 Newton threshold', col = 'red',  
 adj = c(-0.2,-0.5))  
 text(x = chunk$Time.msec[peakIndex], y = chunk$Norm[peakIndex],  
 labels = 'Peak', adj = -0.3)  
 # Determine if this was a peck or push  
 peckPush = peckPushAlgo(chunk$Norm, peakIndex = peakIndex,  
 forceThreshold = forceThreshold)  
 # Put an informative title on the plot  
 mtext(side = 3, text = paste0(trialDate, ', ',   
 round(peaktime/1000,digits=2),'s, ', peckPush),   
 cex= 1.3, line = 1)  
   
}  
  
#plotpeckpush(eventNum = 9, events = events2, forces = forces, calibs = calibCoeffs)

The code defines a function peakPushAlgo to classify each peak event as either a push or peck, when an sample force greater than 2 Newtons was identified. The function peckpushClassify then cycles through all identified events and classifies them. The code in peakpushClassify also determines the duration of each event, based on the time the force spent above the 2 Newton threshold during that event. Because the raw data were sampled at 10 millisecond intervals, a single peak force above 2 Newtons (peck) will have a duration of 20 ms. That is the time it takes to go from ~0 N to >2 N and back to ~0 N. For push events, the duration should end up longer than 20 ms. The output of peakpushClassify is a data frame similar to the input events data frame, but with four new columns added on, PeckPush, Duration.msec,StartPeak.msec, and EndPeak.msec. The latter two columns list the time stamp in milliseconds where the algorithm declared the start and end of a given event (the force exceeding the 2 Newton threshold). If an event in the events data frame does not meet the critera for a peck or a push, these new columns will contain NA.

# Re-load the events file from the output of the peakChooser function above  
  
events2 = read.csv(paste0(fdir,'Events\_picker\_output\_20171211.csv'))  
events2$Date = as.Date(events2$Date)   
events = events2  
events3 = peckpushClassify(events = events2, forces = forces,   
 calibs = calibCoeffs, forceThreshold = 2, baseLength = baseLength)  
# Save the data frame with peck/push classifications as a new csv file.   
#write.csv(events3, file = 'Events\_classified\_20171211.csv',row.names=FALSE)

# Make some example plots  
for (i in seq(1,36,by = 4)){   
 par(mfrow = c(3,2) )  
 for (j in i:(i+5)){  
 plotpeckpush(eventNum = j, events = events2, forces = forces,   
 calibs = calibCoeffs, forceThreshold = 2, baseLength=baseLength)   
 }  
   
}



par(mfrow=c(1,1))