G54MRT Coursework 2 Final Report

Title: Exploring the Effect of Different Belaying Techniques on a Falling Climber Using an Accelerometer

Student ID: 4252754
Date: 12/05/2019

Summary

This project demonstrates the importance of dynamic belaying whilst lead climbing, to ensure a safe and soft catch for a falling climber.
For this project, a system was built for a climber to wear whilst ascending a wall to measure the acceleration and deceleration of a fall. Using these values, we can calculate the jounce (m/s/s/s/s, m/s^4) of the climber, where the higher the jounce value, the harder the fall. By performing a series of test falls, both large and small with static and dynamic belaying, at the University's climbing gym, we can determine a threshold jounce value signalling a good or bad catch.

Background and motivation

The motivation for this project came from my passion for climbing, and determination to disprove common and potentially dangerous belaying mistakes.
Lead climbing is a technique used by climbers both inside and outdoors. A climber ascends the wall whilst attached to a belayer via a rope, attaching the rope to protection points located on the wall. The belayer provides the climber with rope as they ascend and catches the climber if they fall.
A belayer may employ one of two belaying styles; static belaying, where the belayer doesn't move to slow down the falling climber, and dynamic belaying, where the belayer actively moves to slow down the falling climber.
The distance a climber will fall depends on the height above the last piece of protection they are and how much slack is in the system. E.g. if there is 3ft of slack in the system and the climber is 4ft above their last protection point, they will fall approximately 12ft; 4ft to the anchor, 4ft past the anchor, 3ft of slack and around 1ft of rope stretch.
Big falls could potentially injure a climber if they abruptly stop, hence it is critical for the belayer to provide a soft dynamic catch.
Whilst climbing I have been offered belaying advice, most commonly "more slack in the system will result in a softer fall". This statement is untrue, as more slack will give the climber more time to accelerate, and when the slack has gone the climber will stop with a much greater force. Despite this, people still believe the statement, hence why this project is aimed at disproving it and helping people understand dynamic belaying.

Related work

Fall detection has been implemented in a few other products on the market, such as the Apple watch and other smart phones and watches [1]. These fall detection products are predominantly aimed at seniors to aid them when they fall and are not used in sport to

enhance performance. When looking at the design of Apple watches, I noticed that a gyroscope was needed to determine if the user has fallen down and not moved forward [3]. From this I learned I would not need a gyroscope, as the climber will only move vertically. There are currently no products on the market to help climbers learn belaying techniques by evaluating the impact of a climber during a fall, therefore there is very little related work for this project.

Design

Although the system evaluates the performance of the belayer, it is the climber that must wear the system to get acceleration readings during a fall. When designing, the biggest problem was determining how to begin data logging. To start logging, either the belayer must remotely start the program via a laptop or the climber must start it. It would be dangerous for the belayer to divert attention away from the climber so it was clear that the climber must start the program themselves, starting and stopping logging on the wall as necessary. Climbing with a backpack is common practice when climbing outdoors therefore it made sense to fix the system within a backpack on the climber. This design is displayed in Figure 1.

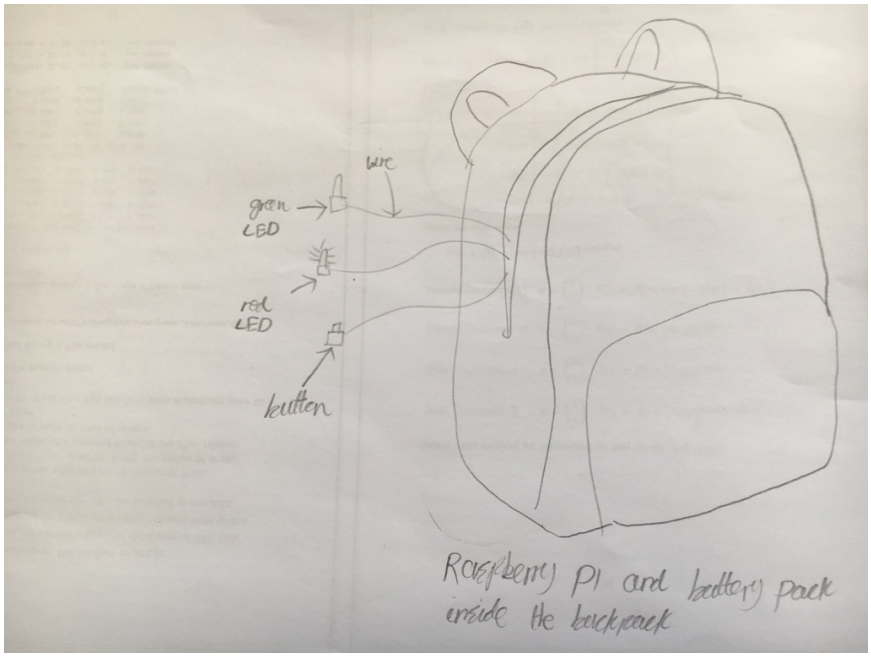

Figure 1 Diagram of System Iteration

To allow a climber to initiate data logging, a design of two LEDs and a button was used. With the main body of the system secured in the backpack, a button along with a red and green LED would stick out of the zip on the side of the bag using extra-long cables. The program will begin when the climber is at the bottom of the climb, but no logging will take place and only the red LED will be on. When the climber is ready to fall they can press the button; the red LED will turn off and the green LED will turn on to indicate data is being logged.

To log the fall data an accelerometer was added to the design, providing x, y and z acceleration values. When designing, it was clear the accelerometer would not sit perfectly flat within the backpack so just taking the z-axis readings would not be sufficient and all the

data streams would have to be merged. As mentioned in my related work, I knew that this method is sufficient, without a gyroscope.
Within the design I also planned for an NFC chip reader to allow the belayer to scan their student card to keep track of their belaying performance.
A diagram of the entire system design is displayed in Figure 2.

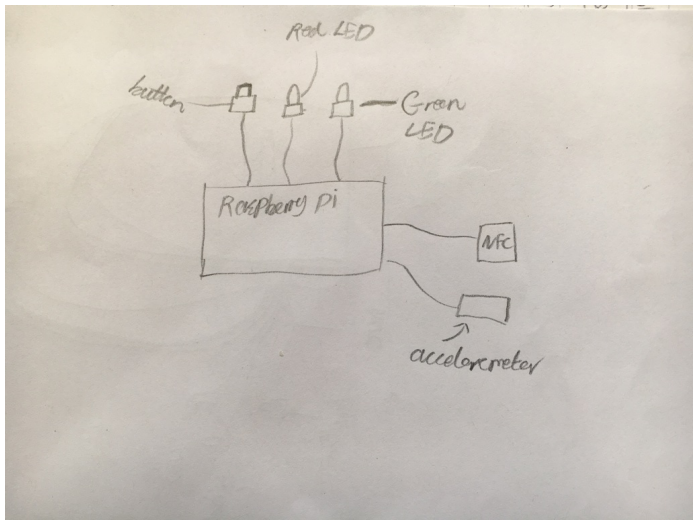

Figure 2 Diagram of System Hardware Design

Implementation

As described above, a red and green LED was used along with a button to start the data logging. An accelerometer was used to continuously take data readings whilst logging. Figure 3 displays the system in its final state.

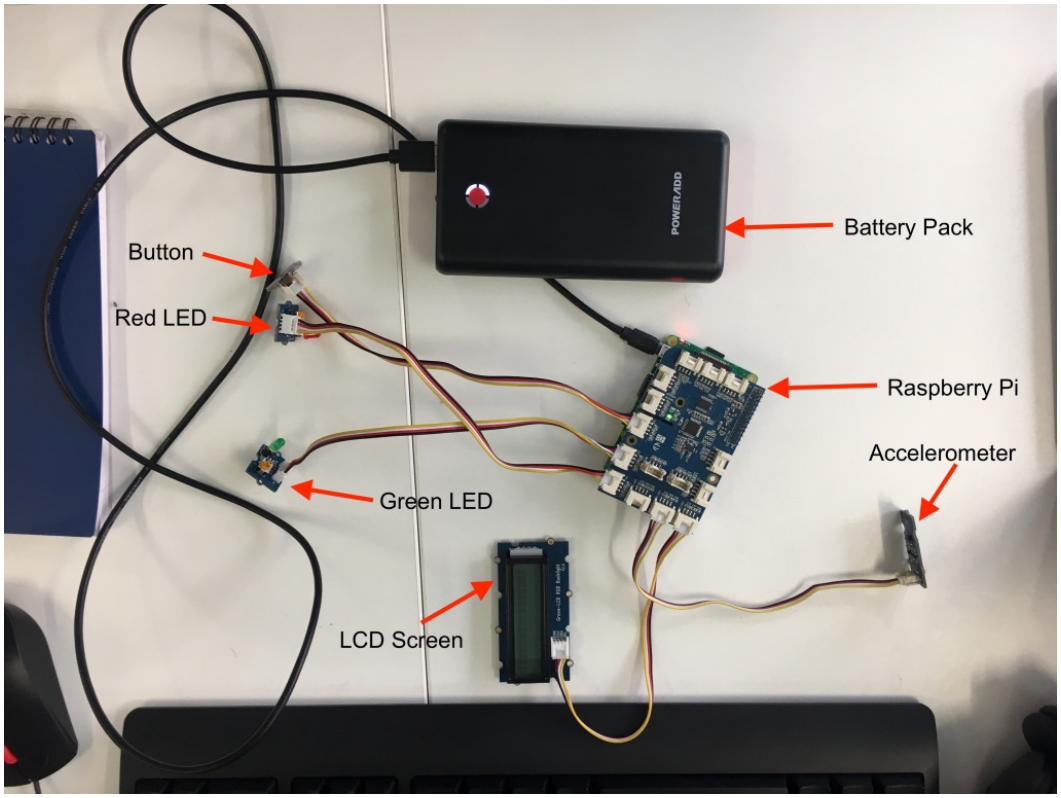

Figure 3 The system fully built

Although the x, y and z acceleration data points had to be merged into a single value, I decided that I would not do this whilst logging, as the acceleration data points can be smoothed and merged during the pre-processing stage of the data evaluation. I also decided that I would not sleep the logging, making the data logging stage easier to implement, faster to run on the Raspberry Pi and give more data points for evaluation. The section of Python code responsible for logging the acceleration values and time elapsed can be seen in Figure 4.

```
firstVals = list(grove6axis.getAccel())
startTime = time.time()
print("%f, %f, %f, %f"%(firstVals[0], firstVals[1], firstVals[2], 0.0))
index = 1
while True:
    # End logging
    if grovepi.digitalRead(2) == 1:
        # Turn red light on and green off
        swapLight(4, 3)
        logging = False
        break

    # Get the acceleration Values
    nextVals = list(grove6axis.getAccel())
    # Print values - with piping it'll go to csv
    print("%f, %f, %f, %f"%(nextVals[0], nextVals[1], nextVals[2], time.time() - startTime))
```

Figure 4 Logging code

Once fall (or simulated fall) data had been collected, a csv of raw data needed evaluation. All data manipulation was performed in R due to its capability of dealing with datasets. Plotting the x, y and z axis in scatter graphs displays noisy but promising results, as displayed in figures 5-7. The plots below show real climbing fall data. All plots were created in R using the library 'ggplot2'.

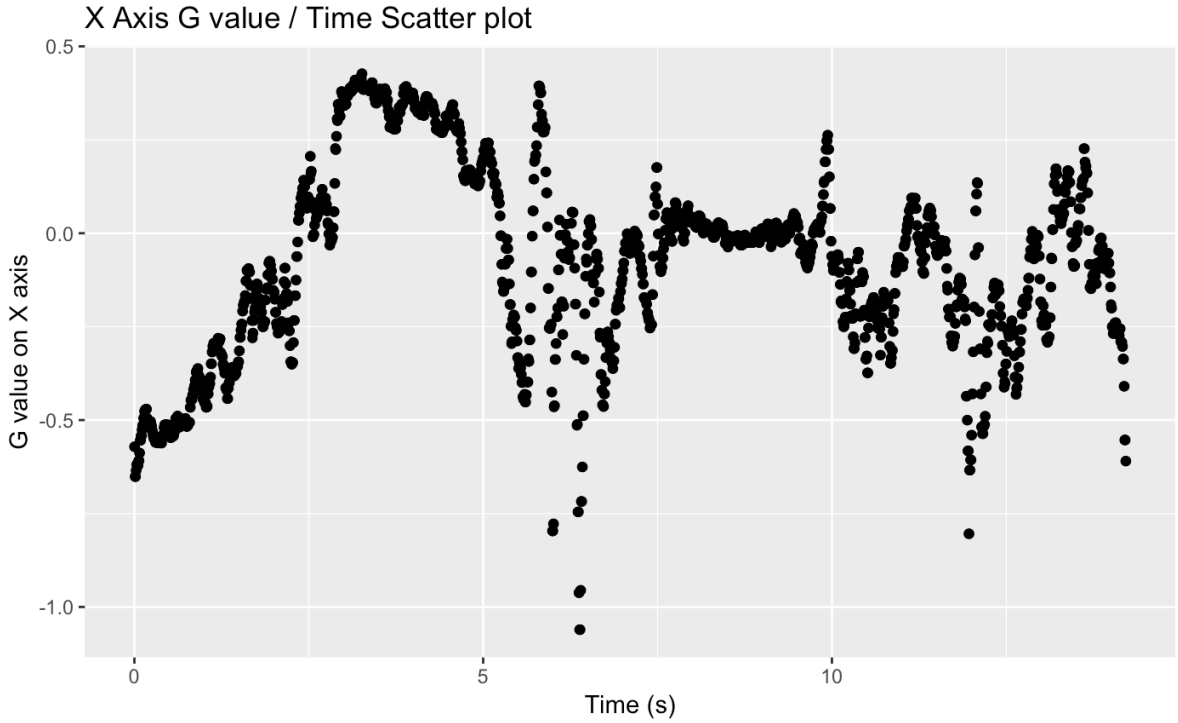

Figure 5 X-axis fall data

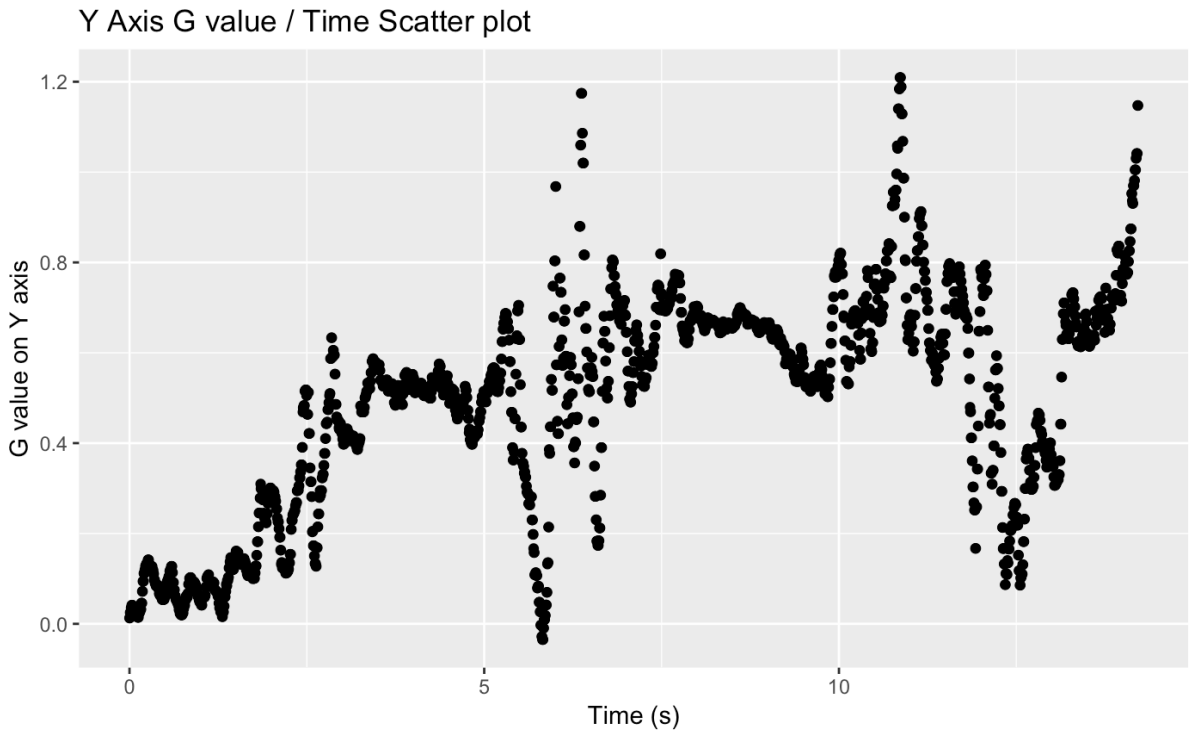

Figure 6 Y-axis fall data

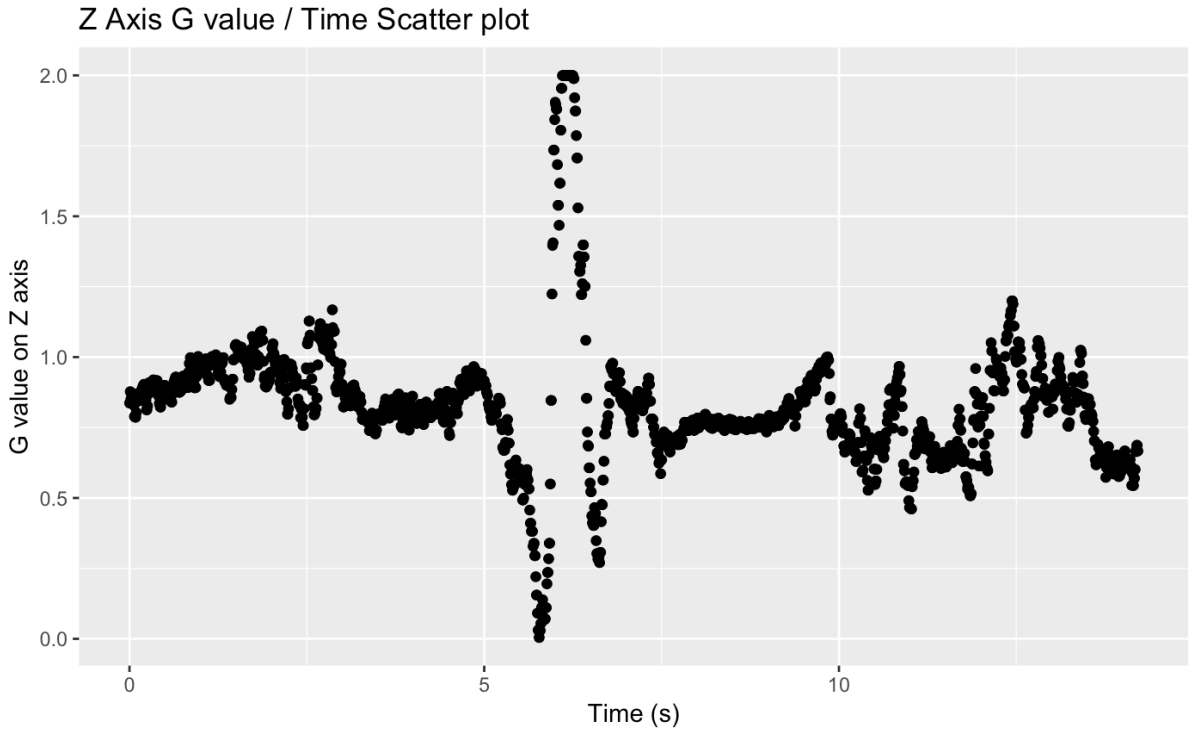

Figure 7 Z-axis fall data

As seen above the z-axis displays the least amount of noise, as I tried to position the accelerometer as flat as possible when securing it within the backpack.
To smooth the data a rolling average was taken based on the size of the dataset. To calculate how many data instances are averaged at a time, the following formula was used; $\lfloor(n \div 24)+3\rfloor$ where n is the number of rows in the dataset. This was obtained by making careful adjustments over many datasets, which would have not been possible if we tried smoothing

during data collection. After smoothing, we have workable data as displayed in Figures 8-10 and the code used to smooth can be seen in Figure 11.

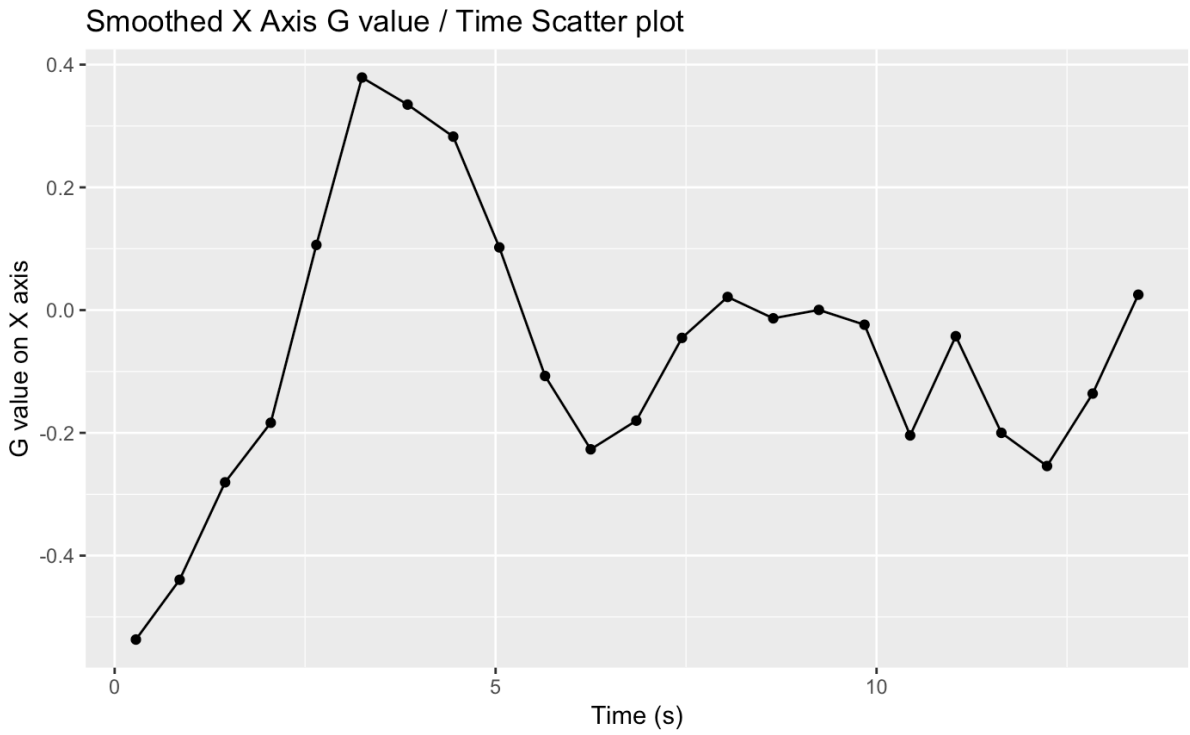

Figure 8 X-axis cleaned fall data

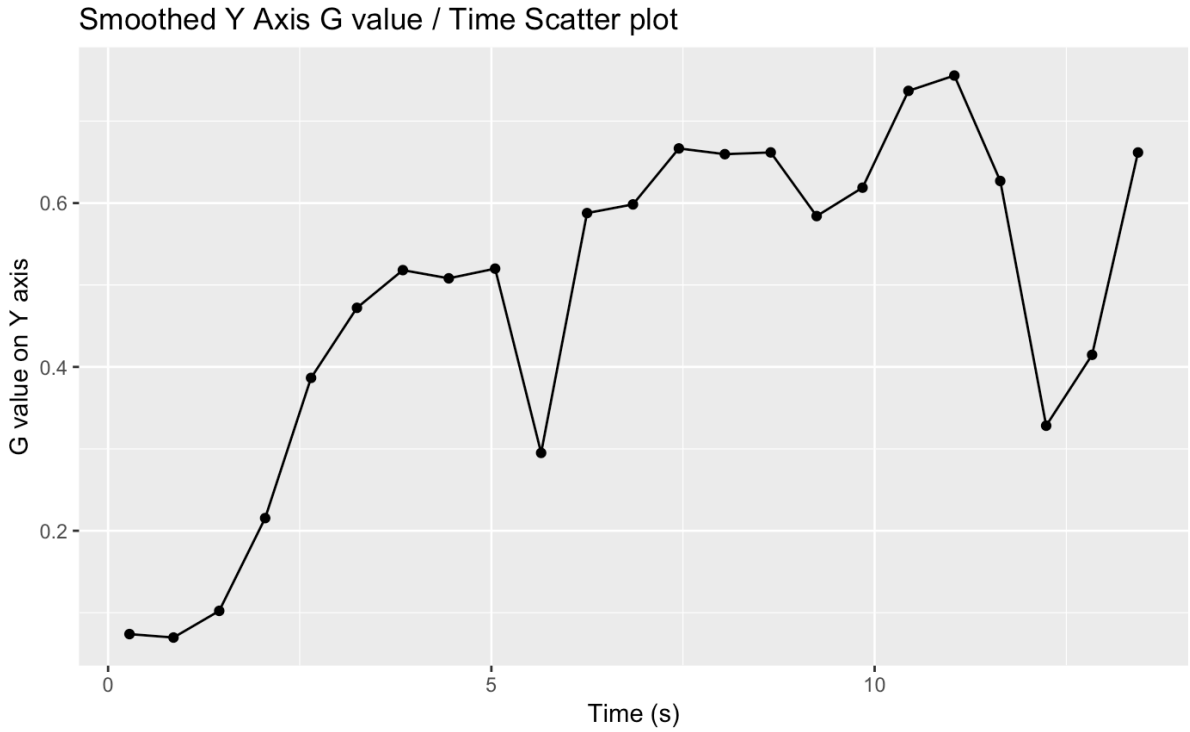

Figure 9 Y-axis cleaned fall data

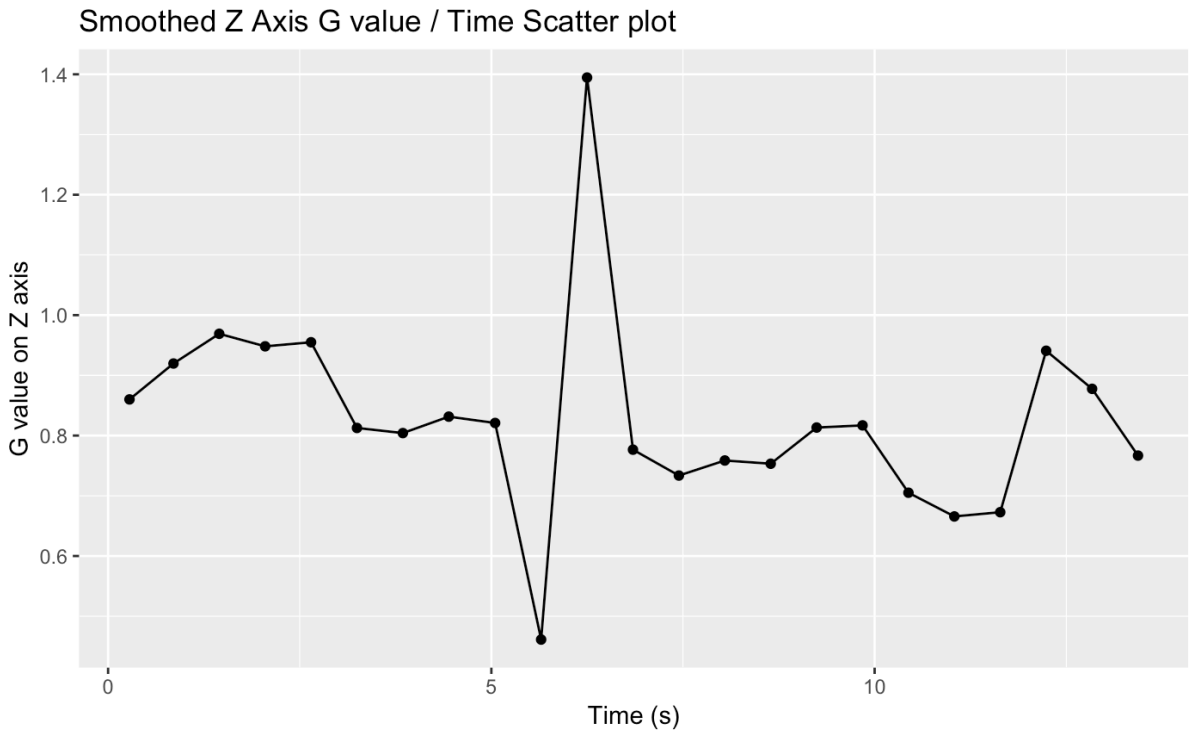

Figure 10 Z-axis cleaned fall data

```
# Calculate rolling average number
avgNum <- floor(nrow(data) / 24) + 3
index <- 1
count <- 1
# How many loops we need to average the whole dataset
loop <- nrow(data) - (nrow(data) %% avgNum)
# Create vectors and fill with 0
avg.x <- replicate(loop / avgNum, 0)
avg.y <- replicate(loop / avgNum, 0)
avg.z <- replicate(loop / avgNum, 0)
avg.time <- replicate(loop / avgNum, 0)
for(i in 1:loop){
  # If the rolling agerage rumber is reached divide the sum to get the average
  if(count == avgNum){
    count <- 1
    avg.x[index] <- avg.x[index] / avgNum
    avg.y[index] <- avg.y[index] / avgNum
    avg.z[index] <- avg.z[index] / avgNum
    avg.time[index] <- avg.time[index] / avgNum
    index = index + 1
  }else{ # Sum the values
    avg.x[index] <- avg.x[index] + data$x[i]
    avg.y[index] <- avg.y[index] + data$y[i]
    avg.z[index] <- avg.z[index] + data$z[i]
    avg.time[index] <- avg.time[index] + data$time[i]
    count <- count + 1
  }
}
```

Figure 11 Data cleaning R code

Axis data was merged by summing the squared value for each point and square rooting it to give a single gravitation pull value. We can subtract 1 and multiply by 9.81 to give the acceleration of the climber as they are falling. The graph in Figure 12 shows the final acceleration values and the stages a climber goes through when falling, from accelerating downwards, to decelerating when the rope catches them and finally a bounce at the end due to rope stretch.

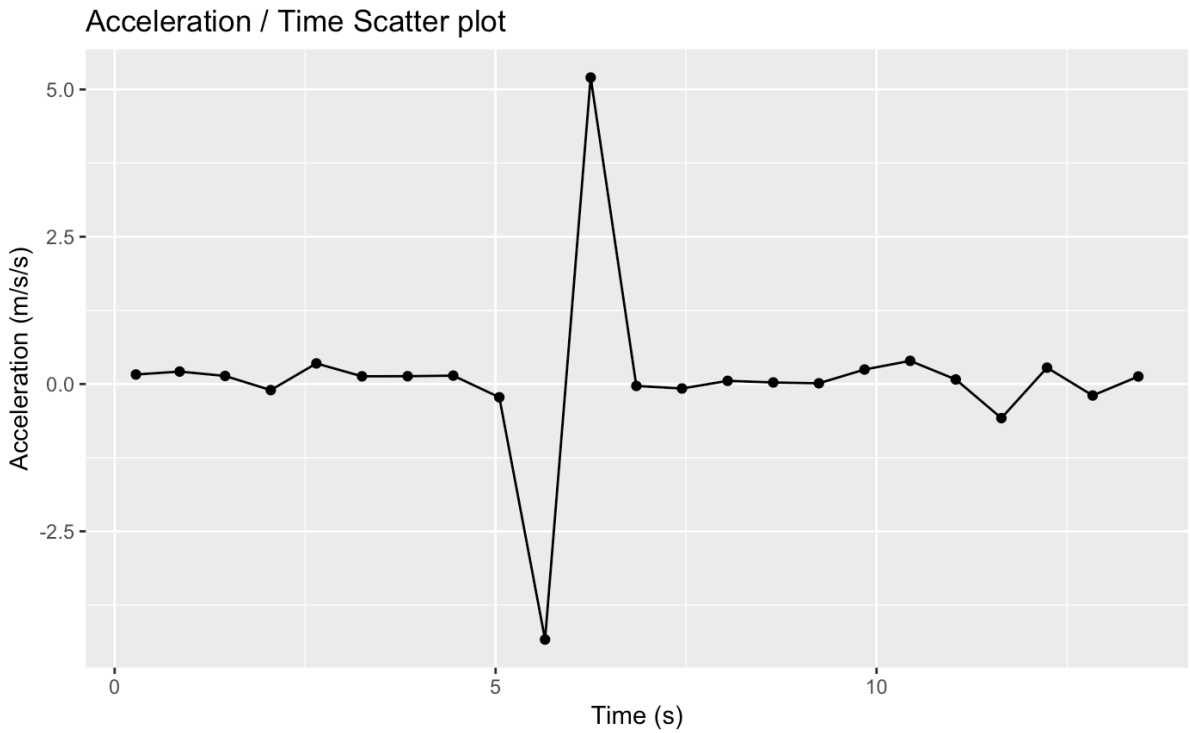

Figure 12 Acceleration time graph

With the data cleaned and merged into a single axis, I could calculate the jounce [2] (also known as snap) of the fall to determine if the fall was soft or hard. To do this we want the change in acceleration before and after the climber is caught by the rope, these points are marked on the graph in Figure 13. To calculate the change in acceleration (jerk) of the falling climber, we get the acceleration at the time the rope catches them (the bottom point on the graph) and the acceleration at the time before that. The formula $\Delta a \div \Delta t$ is used, giving a value of -5.78. The same can be done after the rope has caught the climber for a jerk value of 15.38.

Calculating the change in these jerk values give the jounce ($\Delta j \div \Delta t$) resulting in a value of 17.64. The relevance of this value is explained in the testing section of this report. A code sample displaying how the jounce value is calculated is displayed in Figure 14.

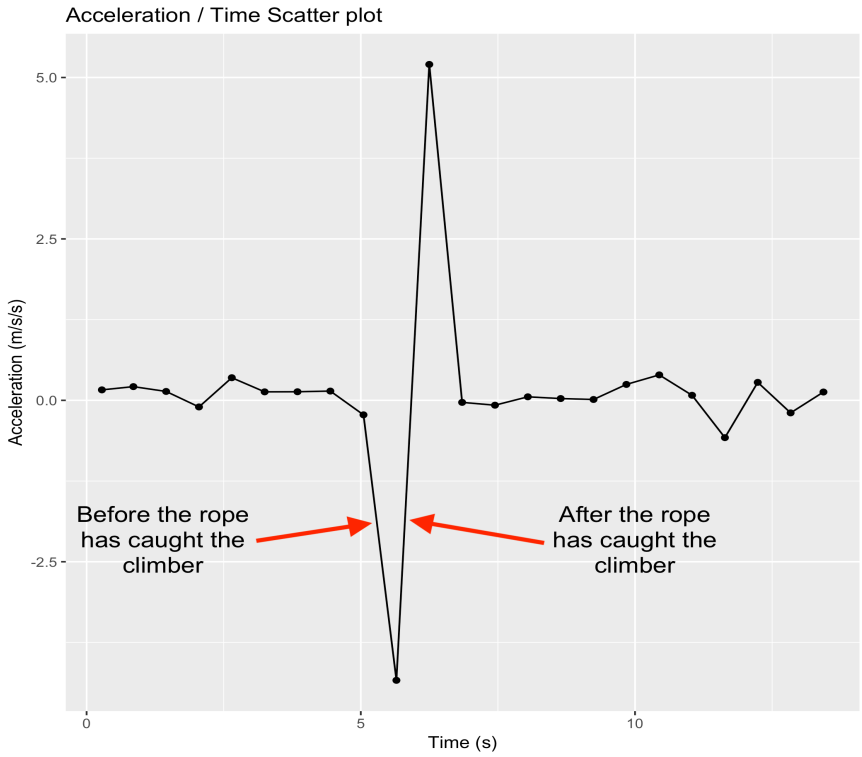

Figure 13 Labelled Acceleration Time Graph

```
calculateJounce <- function(avg.df){
  # Get the min acceleration value
  minMag <- min(avg.df[,5])
  # Get the time at the min acceleration
  minTime <- avg.df[avg.df$mag == minMag,]$time
  # Get max acceleration
  maxMag <- max(avg.df[avg.df$time > minTime,]$mag)
  # Get the time fror the max acceleration
  maxTime <- avg.df[avg.df$mag == maxMag,]$time

  # Get a subset of the data points between the min and max time
  lowerPoints <- avg.df[avg.df$time <= maxTime,]
  interPoints <- lowerPoints[lowerPoints$time >= minTime,]

  # Calculate Jerk after caught by the rope
  jerkPoints <- c()
  for(i in 1:nrow(interPoints)-1){
    t <- interPoints[i + 1,]$time - interPoints[i,]$time
    a2 <- interPoints[i + 1,]$mag
    a1 <- interPoints[i,]$mag
    jerkPoints[i] <- (a2-a1)/t
  }
  jerkUp <- sum(jerkPoints)

  # Calculate Jerk before the rope catches
  bf <- avg.df[avg.df$time <= minTime,]
  i <- nrow(bf) -1
  bfT <- bf[i,]$time
  t <- bf[i + 1,]$time - bfT
  a2 <- bf[i + 1,]$mag
  a1 <- bf[i,]$mag
  jerkDown <- (a2-a1)/t

  # Calculate the Jounce
  jounce <- (jerkUp - jerkDown) / (maxTime - bfT)
  return(jounce)
}
```

Figure 14 Code to calculate jounce

Testing

Testing the system comprised of two parts; lab testing and testing at a climbing wall. Whilst in labs I created datasets of simulated falls and discovered my first challenge; cleaning the noisy data into clear graphs. A rolling average had to be taken, however a fixed value for the number of rows to be averaged didn't work well for different dataset sizes. Some datasets would be stripped back too far and others not enough, so it was clear the average had to be taken respective of the dataset size. The rolling average value was determined using trial and error over 6 datasets, simulating 3 hard and 3 soft falls. Finally, the end formula was derived, which works across all datasets.
Simulated falls were created as 'soft' and 'hard' to try and determine a clear separation in jounce values, which was accomplished. An example of a cleaned simulated hard fall is shown in figure 15.

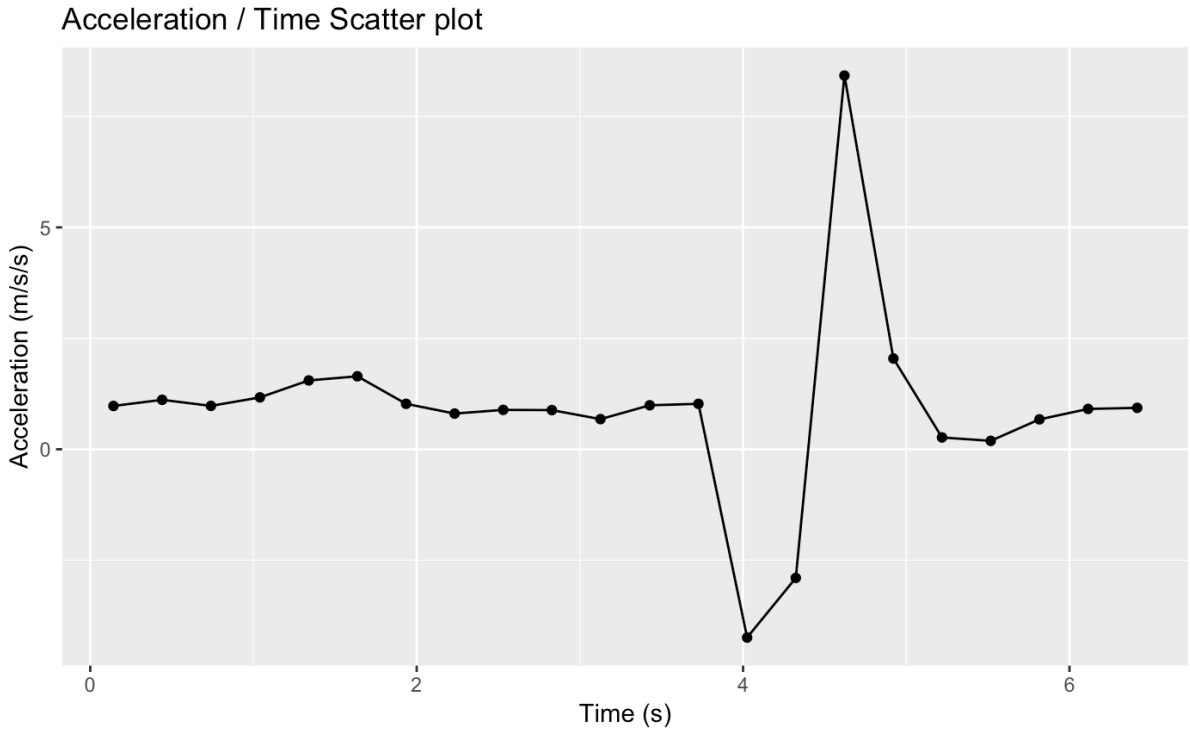

Figure 15 Acceleration time graph for simulated data

With the dataset cleaning and jounce calculations working well, data collection at the climbing wall could begin. 6 falls were taken by myself with my climbing partner belaying from 3 different heights each with a static and dynamic belaying technique used. Figure 16 shows a table of the results of these falls and whether it was a soft catch or not. To determine if the fall was a soft catch, a threshold of 17 was used for the jounce value, obtained using the data from these datasets. One should note that a bigger fall was made by adding slack into the system. We tried to get the same amount of slack into the rope system for each fall of the same height, but this is hard in practice. We can see that dynamic belaying provides a softer catch on all fall heights making it safer for the climber.

Fall Height <fctr>	Belaying Style <fctr>	Jounce Value <fctr>	Soft Catch? <fctr>
Small	Static	24.29	No
Small	Dynamic	12.11	Yes
Medium	Static	25.11	No
Medium	Dynamic	7.26	Yes
Large	Static	18.99	No
Large	Dynamic	11.63	Yes

Figure 16 Real Fall Data Results

During implementation I decided that an NFC chip wasn't necessary to log the belayer's progress as one can name the dataset with their student ID when running the Python program, by piping the outputted data to a csv where you also name the file.

Critical reflection

The design of the system worked well when put into practice, detecting a soft catch compared to a hard catch. The design is effective when testing the belaying techniques and can help beginner lead climbers learn how to belay dynamically and safely. The design of the system helped overcome the issue of how data logging will begin, and the solution implemented works well. A weakness of the design and implementation, however, is logging can only begin just before a fall. It can be awkward for the climber to start the logging process if they are already tired from climbing, but if not, it is relatively easy.
Consistently providing a dynamic catch may be difficult for some, so a higher dynamic belaying jounce value may not necessarily indicate that their dynamic belaying didn't work, but that comparatively to previous attempts it wasn't as good. Testing some belaying techniques such as adding more slack can be dangerous if the belayer gives too much, so care needs to be taken when testing out such a hypothesis. For safety reasons I couldn't get other people to test the system on the climbing wall, however I did ask my belayer if they could simulate a fall using the system. She was able to start logging the data, simulate the fall and end logging without much instruction.

References

1 - Casilari-Pérez, Eduardo & A. Oviedo-Jiménez, Miguel. (2015). Automatic Fall Detection System Based on the Combined Use of a Smartphone and a Smartwatch. PLOS ONE. 10. e0140929. 10.1371/journal.pone.0140929.

2 - Eager, David and Pendrill, Ann Marie and Reistad, Nina. (2016). Beyond velocity and acc eleration: Jerk, snap and higher derivatives. IOP Publishing, European Journal of Physics. htt p://dx.doi.org/10.1088/0143-0807/37/6/065008

3 - Verger, R. (2018). The Apple Watch learned to detect falls using data from real human mishaps | Popular Science. [online] Popsci.com. Available at: https://www.popsci.com/apple-watch-fall-detection [Accessed 12 May 2019].

Appendix A – Instructions

To set up the system you'll need the raspberry pi with the grove pi attached, a green and red LED, a button and an accelerometer with compass (not gyroscope). You need to connect the accelerometer to any I2C port on the grove pi board. The button needs connecting to the second digital pin D2, the green LED needs connecting to the third digital pin D3 and finally the red LED needs connecting to the 4th digital pin D4.

To run the data logging system, you will need to connect a laptop to the raspberry pi. From the command line you can run the script and pipe the output to a csv using the command 'python FallDetect.py > climbingFall.csv'. This will start the script but not the logging, to start logging data just before a fall you need to press the button and when the red LED turns off and the green LED turns on you are ready to take the fall. To stop logging you need to press the button again, the green LED will turn off and the red LED will turn back on.
You should now have a CSV file with the fall data. Make sure this file is in the same directory as the R script and your R working directory is set to the same file path. At the bottom of the R script are 2 variables you can change. The first variable is called 'filePath', you need to set in to the name of the CSV containing the fall data. The second variable is called 'saveGraphsToPDF', this is a Boolean and if you set it to true all the axis graphs and acceleration graphs will be saved to a PDF file where you can view them.
You can then run the script and the output in the R console will tell you if the catch was hard or soft.
To run the script, you can highlight all the code with 'ctrl + A' and run it by pressing 'ctrl + enter'.