Development of a Chair-Based Measurement System to Supplement Get-Up-and-Go Tests

A senior team project report
submitted in partial fulfillment of the requirement for the degree of
Bachelor of Science in Physics,
Engineering Physics and Applied Design
from the College of William and Mary in Virginia,

by

Ruth A. Beaver
Chang Lu
Matthew M. Velasco

Mentor: Prof. William E. Cooke

Prof. Jeffrey K. Nelson

Williamsburg, Virginia
April 30 2020

Acknowledgements

We would like to thank Dr. Cooke for his guidance and support, without which this project could not have been completed.

We would like to thank Dr. Burnet for answering our questions and allowing us to observe her test procedures.

We would like to thank Dr. Frey and the makerspace engineers for their assistance and advice.

Contents

List of Figures

List of Tables

Listings

1 Introduction

A major health risk for the elderly is falling. Not only does falling cause injury, but after the first fall elderly people experience a decline in health and an increased risk of subsequent falls. There are ongoing research efforts to find reliable testing methods to identify elderly people who are at risk for falling before they fall so that preventative measures can be taken to preserve their health, such as prescribing exercises for specific muscle groups. Dr. Burnet and her team at the William and Mary Department of Health Sciences are researching possible tests at the Williamsburg Landing retirement community. Dr. Burnet's group uses various tests to gauge the strength and balance abilities of elderly test subjects. They hope to find tests that can reliably predict fall risk in seniors, which can then be used to identify at-risk patients for preventative treatment (for example: an exercise plan). However, most of the tests they use are very qualitative and labor intensive, involving subjective ratings and measurements taken and processed by hand. Her team wants us to automate the different tasks as well as give her team the ability to take more quantitative data.

We formulated two possible systems to aid in Dr. Burnet's efforts: a chair-based system for measurement of force exerted on a chair during sitting and an accelerometer-based system for measurement of balance recovery. The chair-based system would provide information on both the balance abilities and the strength of subjects while the accelerometer-based system would provide information on only the balance abilities of subjects. After some development of both of these systems in parallel, the decision was made to direct our focus to the chair-based force sensor system. In the second chapter of this thesis, we discuss the initial literature research and design of both systems. We also outline the progress made on the accelerometer system before its downselection. In the third chapter of this thesis we discuss the development of the chair-based force sensor system.

2 Design and Planning

In this section the rationale behind each system is discussed as well as the initial plan for the designs of the systems. The chair system was designed chiefly to measure control during sitting using the initial peak force in the sitting event. The accelerometer system was designed to provide quantitative data on a subject's balance abilities by measuring balance recovery as a time series. The progress made on the accelerometer system before its downselection is also discussed in this section.

2.1 Chair-Based Force Sensor Measurements

In this section the initial design and literature research for the chair-based force sensor system is discussed.

2.1.1 Rationale

A test used by Dr. Burnet's team, the "get-up-and-go" test, involves the patient trying to stand up and sit down as fast as possible for five repetitions. This test is timed, and the student researcher rates the difficulty of the action for the test subject. While this test is already somewhat quantitative (ie. it is timed) we hoped to provide more information on the strength of the test subject by measuring the force exerted on the chair during the test using force sensors. We hypothesize that subjects with less strength will have less control during sitting and thus sit down harder on the chair, which we believe could be measured by comparing the height of an initial impulse peak to their weight on the chair. The data generated by the chair-based force sensor system could also be used to automatically generate the timing data that the group already uses. Finally, there is a possibility that the chair-based system could provide information on the balance of elderly patients by measuring their weight distribution during sitting events. Thus, this system would aid in Dr. Burnet's efforts to quantify both balance and strength.

2.1.2 Literature Research

There has been considerable research done on using chair-based sensing techniques to get information on the chair occupant's activities. For example, a group has developed a method to determine posture and detect gestures by processing images generated by a sensing mat placed on the chair.[1] Another group used handmade, resistance-based load cells placed under the legs of a chair to detect user activity.[2] They were able to both measure total weight and distribution of weight on the chair, and use fluctuations in these measurements to detect activities such as typing. The simplicity and effectiveness of this setup suggested it would be a good way to approach our measurement problem. To get more information on force measurements, we researched how electronic bathroom scales are constructed. The key component in an electronic bathroom scale is a strain gauge load cell, which changes resistance when pressure is applied to it.[3]

More information was needed, however, on the variables that would be calculated by the system. Previous kinesiology studies have been done using floor-based force sensors to measure power exerted when standing up.[4] Other studies have used body-based sensors to measure variables such as trunk angle and acceleration, and used these measurements to calculate quantities such as power exerted or time to sit.[5,6] Other studies were done based on a strategy of measuring many quantities and then later using statistics to determine which quantities were useful.[6] However, no kinesiology studies have been done with chair-based force sensors. Thus, there are no established quantities for chair-based sensors to calculate, and no established requirements for accuracy or time resolution of measurements.

Our client has expressed interest in measuring "control during descent" but not force distribution. Since it is fairly common in the literature for studies to be done measuring novel variables and then determining the statistical significance of those variables later, we decided to create a device that could measure directly the force exerted on the chair over time. The data from this device will be used to determine several physically relevant variables: the force a person exerts on the chair when they initially sit down (do they drop onto the chair or sit down with control?), the time between sitting and standing, and distribution of weight during sitting. We decided to include distribution

measurement capability despite our client's lack of interest in it in case the measurement becomes of more interest later on. The device will be used in tandem with the "get-up-and-go" test, where a subject is asked to stand up and sit down five times as fast as they can. The time it takes the subject to complete this task is measured, and they are given a rating on how easily they can perform the task. Quantitative data from our device with strengthen this testing method since it is more objective than the current rating system for measuring how "easily" a person completes the task.

2.1.3 Proposed System

Our proposed system would measure the force exerted on the chair during a get-up-and-go test. We believe that each sit event will begin with an initial force peak corresponding to the impulse of the person as they sit on the chair. This should be smaller for patients who sit with control and larger for patients who fall into the chair. The force should then reach a steady state representing the subject's weight. Then force will go to zero (or the weight of the chair) as the subject stands again. We propose to compare the impulse peak to the steady state weight to give a measure of the subject's control during sitting. We are also interested in automatically replicating the time data that is currently being measured by Burnet's team.

2.2 Accelerometer Balance Recovery Measurements

In this section, the initial design and planning of the accelerometer system is discussed as well as the initial progress made on the system before its downselection.

2.2.1 Rationale

One of the tests used by Dr. Burnet's team, the "push test," involves a push applied by a member of Burnet's team on the patient. The patient's ability to regain balance is measured on a qualitative scale from 1-5 where 3+ is considered a significant fall risk. The rating system used here is a very qualitative process, which relies on the subjective judgements of student researchers. We came up with the idea of a body-mounted accelerometer to measure the process of balance recovery, which we could then

quantify using a time series. We believe the time series data from the accelerometer would allow quantification of balance abilities of elderly patients, thus aiding in Dr. Burnet's efforts to quantify balance.

2.2.2 Literature Research

Several systems have been developed to quantatively assess human postural balance. Of these systems, force plates are the most commonly used.[7] Force plates measure the center of pressure (CoP) of a person standing on them. CoP is defined as the point of application of the ground reaction force (force exerted by the ground on a body in contact with it).[8] Force plates usually consist of a board in which four force sensors are distributed to measure the three force components, F_x, F_y and F_z and the three components of the moment of force, M_x, M_y, and M_z, acting on the plate.[9] The CoP positions are calculated using these six measurements. When CoP is measured of a patient regaining balance, many variables can be derived from the raw CoP data which can provide information about the balance recovery mechanism. Some commonly used variables are mean coordinates, ellipse area, path length, amplitude of displacement, velocity and standard deviation.[7,9]

Body-worn accelerometers have been recently explored as a portable, low-cost alternative to the force plates. In one study,[10] an accelerometer was positioned at the subjects' pelvis to measure center of mass acceleration. The study was able to yield measurements that have good test-retest reliability and are correlated well with the CoP variables. There are also studies where the accelerometer was attached to different body parts such as the chest or posterior trunk.[11,12] However, the effect of the position the sensor on the subject's body on the measurements is unknown. Another factor that might have an impact on the measuring results is the differences between individual standing postures. An effective method to minimize this possible effect should be taken into account when we develop our own balance system.

No commonly agreed cutoff value was found that differentiates the balance performance of potential fallers from non-fallers for either CoP or accelerometer based variables, since the values are largely dependent on the adopted test method and signal processing

algorithm.[13] However, it is possible to calculate our own cutoff value using a clinical cutoff score formula[14] for a chosen test method once we acquire a large pool of data.

Among the studies trying to compare CoP or accelerometer variables for elderly (aged 60 or over) fallers versus non-fallers during quiet standing, a large amount of them come to the conclusion that their chosen variables are significantly different between fallers and non-fallers, based on a statistical analysis method called ANOVA.[14–16] However, ANOVA test allows researchers to claim two measurements to be significantly different even if their mean values are less than one standard deviation apart. Therefore, it can still be technically difficult to differentiate fallers versus non-fallers when we develop our own balance devices.

While there have been many previous studies on the use of an accelerometer as a motion tracker, there have not been any pertaining specifically to measurements of balance recovery to predict fall risk in seniors. Several of the papers talk about using accelerometers as well as gyroscopes for use in motion tracking. They use the accelerometer to detect acceleration due to gravity. From there, they use arctans of the different axes of acceleration in order to get the angle that the accelerometer is tilted at. In one of the articles, accelerometers are used to track the motion of the entire body rather than just the sway of the upper body as we are doing in our tests. Researchers were able to track and model the movement of a body onto a virtual model with a good degree of accuracy. It was necessary to researchers to calibrate multiple sensors relative to each other; however, this will not be necessary in our proposed system since it will only consist of a single sensor. In another study, the issue of non-gravitational terms is assessed. When an accelerometer is in motion, it is no longer only measuring the acceleration due to gravity, which can lead to problems in determining tilt angles. In this study the non-gravitational term was isolated mathematically to clean up the data.[17]

2.2.3 Proposed System

Our proposed accelerometer system would be attached to the trunk of a subject. During a push test the accelerometer would measure wobbling of the subject during balance recovery along multiple axes. This would generate a time series, which could be analyzed

Required Specifications for the Accelerometer System	
Required Measurement Rate	15 Hz
Required Accuracy	1°
Other	Ease of use, comfortable mount

Table 1: Estimated required specifications for the accelerometer system.

to determine the balance ability of the subject. We hypothesize that subjects who recover their balance with fewer, smaller oscillations will have a better balance ability. Based on the literature search, we believe this is a novel measurement which could possibly improve upon previous data. The data produced by this system would need to be correlated will fall risk by Dr. Burnet's study.

2.2.4 Required Specifications

In order for the accelerometer system to be useful, it would need to attain a measurement rate of 20 Hz. Experiments with the push test using group members suggest that patients will regain balance within one second and sway up to three times. Estimating that about five points per sway event are needed to accurately detect the event gives a necessary measurement rate of 15 Hz.

We estimate that the accelerometer system would need to be accurate to within one degree in order to provide useful measurements. The angles of sway when people are pushed are small, estimated to be well within 20 degrees from the vertical. Thus, the accelerometer system would need to have a high measurement accuracy to measure swaying behavior. Finally, the accelerometer system must be both easy for the kinesiology researchers to use and comfortable for the test subjects to wear. Kinesiology researchers specifically requested that we avoid excess wires in our systems. Thus, the system must either be able to log data via bluetooth or a usb and be able to run without being attached to an external computer.

2.2.5 Progress on Accelerometer System

The accelerometer selected was a SensorTag cc2650. The sensor is connected via bluetooth to a RaspberryPi 3. The accelerometer has to be able to effectively capture the motion to a resolution that provides recognizable data. While the Sensortag specifications say that the period between measurements is 10ms (which translates to a frequency of 100Hz), testing has shown the actual measurement rate to be 8 Hz. This is a major hindrance to the system, since it is well below the required measurement rate of 15 Hz. Multiple attempts to increase the measurement rate of the Sensortag failed.

The system measures the angle away from the horizontal. The measurements were off by about 2° on average with a max deviation of 3.8°. The deviations could be due to the table that the measurements were being conducted. The experiment was conducted by tilting the accelerometer next to a protractor and seeing if the measurement done by the accelerometer was accurately portraying the angle.

Experimentally, we have seen that accuracy in angle measurements is reduced when the accelerometer is in motion. This occurs because the current code assumes that all acceleration is due to gravity; thus, if any other accelerations are present the angle calculations are thrown off. In our literature search, a previous experiment was able to account for the horizontal motion with a formula that eliminates the cross term that is the horizontal motion.

2.2.6 Downselection Decision

We decided to focus on the force sensor system over the accelerometer system. While the accelerometer system was promising and likely to be very useful to the kinesiology study, its progress was impeded by issues with time resolution of measurements. The decision to downselect to the force sensor system was based off of the fact that the force sensor system became more clearly feasible before the accelerometer system did. Additionally, we were motivated to downselect by the amount of work that the force sensor system began to require once a proof-of-concept prototype was completed. The force sensor system also seemed to be able to contribute more to the kinesiology study, since it is theoretically capable of quantifying both balance and strength, while the accelerometer

Required Specifications for the Force Sensor System	
Required Measurement Rate	20 Hz
Required Accuracy	0.2 kg
Other	Ease of use, reliability, and/or ease of calibration

Table 2: Estimated required specifications for the force sensor system.

system could only measure balance.

3 Development of a Chair-Based Force Sensor System

In this section we will discuss the development of a chair-based force sensor system. We begin with an outline of requirements for a successful system. We then discuss the development of the hardware used in the system, followed by a discussion of testing done to validate the sensor data and theoretical calibration procedures to further validate the system. We then discuss the software developed both to run on our system and to process generated data. We conclude with a discussion of improvements that should be made in future iterations of this product.

3.1 Required Specifications

The force sensor system must be able to take measurements at a sufficient rate to fully characterize the force exerted on a chair during a sitting event. Estimating that it takes about half a second for someone to sit down, and assuming we need about ten points per event to characterize it fully, this leads to a necessary measurement rate of 20 Hz. We believe the measurement rate of 80 Hz provided by our current hardware is more than sufficient based on its tested ability to fully characterize various sitting behaviors.

The force sensor system also needs sufficient measurement accuracy to repeatably and reliably measure force on the chair. Given that the average person in North America weighs 80.7 kg,[18] and estimating that weight will be distributed evenly among the four

chair legs, this means that each force sensor needs to be able to give measurements between zero and about 20 kg (neglecting the weight of the chair). Accuracy of 1% would give measurements within 0.2 kg, which seems a reasonable and attainable goal for our system. The force sensors also need to maintain accuracy, meaning that they provide accurate results on multiple measurements separated by large time spans, such as days. To ensure our system meets this criteria, testing should be done to determine whether there is a "drift" in the measurements. If this is the case, we have developed a simple calibration procedure that can be done before each measurement to account for the drift (see the Future Directions section).

Finally, the force sensor system needs to be user-friendly. The system should be easy to set up and adjust. Furthermore, the software to run the system should be simple to run and provide automatic processing of data. Ideally, we want a system that only requires a user to press "go" and then will take measurements and return processed values with little or no additional input from the user.

3.2 Hardware Development

In this section development of the electronic components of the system is discussed, including testing used to validate sensor data as well as theoretical procedures for calibration of the final system. This is followed by a discussion of the development of mounts for the sensors.

3.2.1 Electronics

A bathroom scale force sensor kit was purchased, containing four strain gauge load cells and a HX711 breakout board. Each strain gauge load cell contains two resistors (see Figure 1), one of which has a fixed resistance, and one of which has a resistance which varies with force exerted on the sensor. Force can be measured by applying a voltage across two of the leads and measuring the voltage from the middle lead (labelled "tap" in Figure 1). However, the change in resistance of the variable resistor (and thus the change in measured voltage at the tap) when force is applied to the sensor is very small. Thus, it is impractical to directly measure the output voltage from a sensor to determine

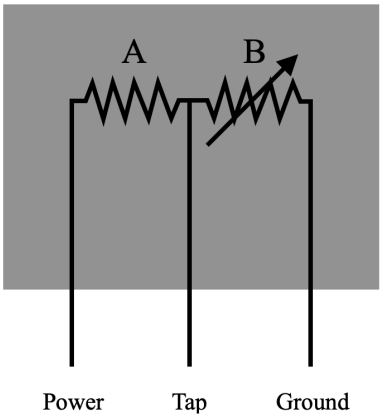

Figure 1: A simple diagram of the strain gauge load cells used.

force applied to the sensor. Initially, this problem was resolved using a wheatstone bridge formation (see Figure 2). This arranges the load cells such that two voltage dividers are formed, each with two variable resistors and two fixed resistors. In one of the voltage dividers, the voltage over the fixed resistors is measured, and in the other the voltage over the variable resistors is measured. The ratio of these two measurements can be used to determine the change in resistance over the variable resistors. In our system, we compared the two outputs of the bridge using the HX711 breakout board which came with the bathroom scale force sensor kit. This breakout board was able to compare two pairs of input voltages, amplify these differences, and convert the differences to digital signals which could be read by an arduino. This breakout board also served to amplify the difference to a measurable amount. Initial measurements of this system were made using an arduino, and we were able to produce live plots of the combined force exerted on the sensors over time.

However, the wheatstone bridge system was not ideal. A wheatstone bridge produces a single combined measurement from all the sensors, thus preventing the measurement of distribution of weight over the sensors. Furthermore, the system would give anomalous results if the weight distribution was not even across the sensors, sometimes reading negative weights.

To fix these issues, a system which could measure variation of force from each sensor individually was developed. Each sensor was treated as an individual voltage

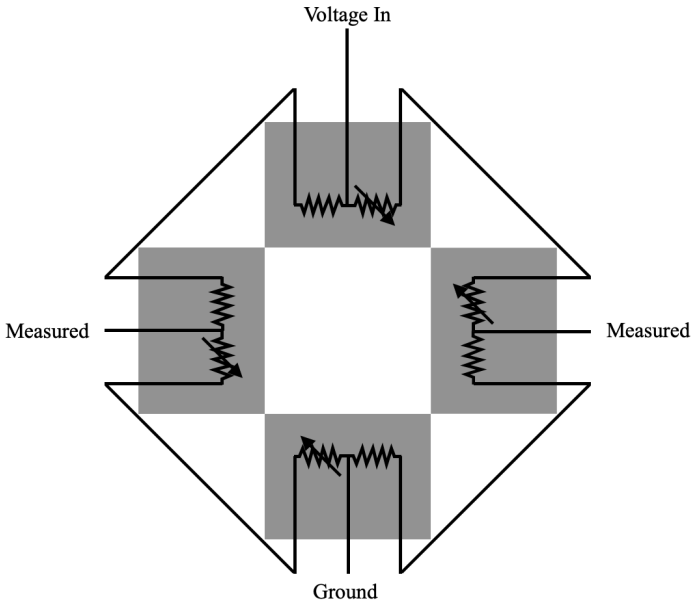

Figure 2: The wheatstone bridge configuration of the strain gauge load cells.

divider, and the output from each tap was compared to a reference voltage produced by a variable resistor. This reference voltage was tuned to be as close as possible to the output voltage of the sensors when no force was applied in order to maximize the measurement range. A separate HX711 breakout board was used for each sensor to compare its output to the reference voltage, amplify the difference, and convert the signal to a digital measurement. We successfully connected four sensors set up this way to a raspberry pi, and were able to produce essentially simultaneous measurements of all four sensors.

3.2.2 Testing and Calibration

We set up an op amp to subtract a reference voltage from the sensor voltage and to amplify that voltage by a factor of 105. We were able to plot force on the sensor in real-time using an arduino. Using this op-amp system, we verified the linearity of the force sensor response by measuring the output voltage when pre-measured weights were loaded onto a sensor. This showed us that the sensors responded linearly to forces up to 12 kg. We were not able to accurately measure forces above 12 kg due to instabilities in our testing system.

We also used the op-amp system to measure the time response of the force sensors. We used an oscilloscope to measure the output voltage from the sensor when a metal cylinder was bounced on it. The high resolution characterization of the bouncing (which had behavior occurring on the order of microseconds) suggests that the force sensor response can be regarded as happening essentially instantaneously. The only limit to time resolution is in our processing of the force sensor response, for example, in the conversion of the signal from analog to digital.

The current system we have is based on the assumptions that each of the four sensors has the same scaling factor (the ratio of weight in kg loaded on sensor to sensor output) and that the scaling factor keeps constant over time. The value of the scaling factor currently does not affect the result of our data processing since the parameters we currently choose to measure are not affected by the scale of the sensor output. However, in the future, if we want our system output to reflect the real magnitude of the loaded weight so that scale-related parameters can be measured (e.g. peak value, the ratio between the weights on the front and rear legs of the chair), knowing the exact value of the scaling factor for each sensor becomes crucial. Therefore, we propose a series of calibration procedures for the future to obtain the value of the scaling factor for each sensor:

Firstly, we want to check if the scaling factors for the sensors are constant over time. This can be achieved by comparing the overall scaling factor (the ratio of total weight in kg loaded on all four sensors to total output of all four sensors) over time. We propose to measure the overall scaling factor by placing pre-measured weights on the chair with four sensors attached below and linearly fitting the plot of the total loaded weights versus the total sensor outputs. The overall scaling factor is equal to the slope of the linear fit. We plan to measure the overall scaling factor on five different days and compare the values to see if they are constant over time.

If the overall scaling factor varies over time, we plan to develop a calibration program that automatically returns the overall scaling factor each time before the data collecting process. If the overall scaling factor does not vary such a procedure would not be necessary and the scaling factor can be hard-coded into the processing software.

We then propose to check whether each sensor has the same value of scaling factor. To achieve this, we plan to place a cylinder with uniform density on the chair. Then we will move the object around the seat to find a location where the outputs of all four sensors are equal to each other (after taring to account for different offsets due to hardware calibration and uneven weight distribution of the chair). At this location, mark the center of the bottom surface of the cylinder. If the center of the cylinder is located at the center of the chair seat, it shows that each sensor has the same value of scaling factor. We would repeat the procedures on five different days before making any conclusion.

If each sensor does have the same scaling factor, this will be equal to the overall scaling factor, and thus easy to measure. If each scaling factor is not the same, we plan to measure each scaling factor by mounting a test stand vertically on individual sensors and adding pre-measured weights to the stand to plot loaded weight versus individual sensor output. If the overall scaling factor of the system keeps constant, we only need to measure the scaling factor for each sensor once and use it to convert sensor output for every data collecting process. However, if the overall scaling factor varies over time, we have to measure each scaling factor before every data collecting process and may need to consider switching to other types of force sensors.

3.2.3 Mounts

We also developed a mount for the force sensor which fit a chair leg. Each sensor had to be elevated by its rim only with no support at its center in order for the sensor to deform correctly to detect force. A mount was designed which supported the sensors at the rim and provided support to hold a chair leg on the sensor. Four of the of these mounts were manufactured using 3D printing, which allowed for initial measurements of sitting behaviors in a chair.

We discovered that our setup was not always level, depending on its location. We were concerned that rocking motion of a slightly unbalanced chair could interfere with measurements, so we decided to design an adjustable mount. In order to level a chair with four legs, in principle only one of the mounts needs to be adjustable. We planned to

make four mounts that would be nearly identical, with one capable of being adjusted. We decided to manufacture the mounts out of aluminum for durability and so that a screw could be added easily.

Initially we designed a mount that would ultimately replace the initial 3D printed non-adjustable mounts that currently held the sensors. However, the original mounts required wiring to be threaded through a hole in the mount, so replacing the mounts would require rewiring the sensors. Due to the complexity of the wiring we made the decision to redesign the adjustable mounts to accommodate the original mounts.

However, this raised a new issue. Using the Tormach 770 (a CNC mill in the William and Mary Makerspace) and an aluminum stock of dimensions 76mm x 154mm, the mill bit would come too close to the mounts and would risk damage to the machine. This was not an issue in the previous iteration since the standalone mount was small enough to properly fit within the stock and the tool path did not come close to damaging the machine. This issue was fixed by milling two separate pockets on one blank and cutting the stock between them creating two separate mounts. We then drilled the holes for the screws using a drill press and hand tapped them to fit an M3 screw on one of the mounts. This design was both easy to manufacture and efficiently uses resources. Unfortunately, due to COVID-19-related closures, we were not able to test the efficacy of the adjustable mounts.

3.3 Software Development

In this section we first discuss software designed to run measurements on our system and then discuss software developed to automatically process data generated by the system. Code discussed in these sections can be found in the Appendix.

3.3.1 Control

Software was developed to allow a Raspberry Pi to communicate efficiently with the force sensors via the HX711 breakout boards. Communication with the boards was done along two lines, DOUT and CLK. The DOUT line is read by the Raspberry Pi. When the DOUT line is high, the HX711 breakout board cannot be read. When the DOUT line

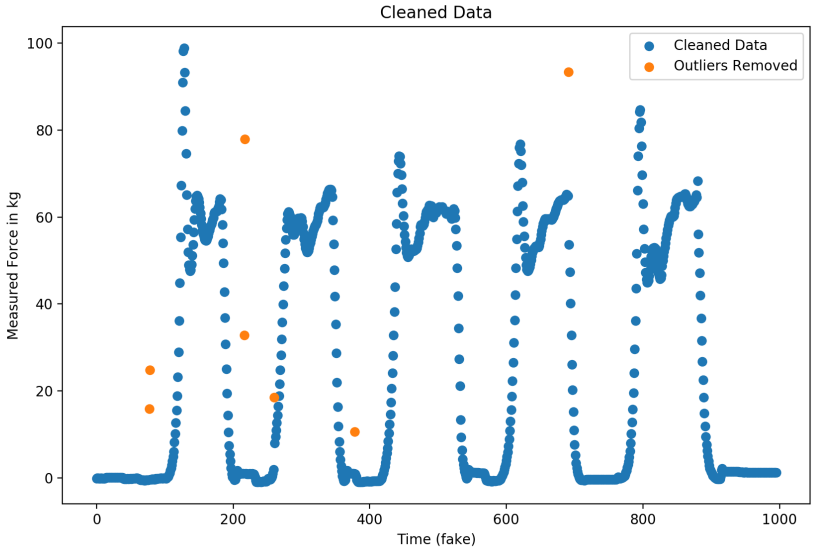

Figure 3: A get-up-and-go measurement containing several outlier points, highlighted in orange. These points were identified and removed using the data filter algorithm. The time vector used for this measurement was artificially generated, and thus the time axis is labeled "fake."

goes low, data can be transferred from the breakout board to the Pi. This is done by applying 25 pulses along the CLK line. After each pulse on the CLK line, the breakout board will shift a bit of the digitized data on the DOUT line, which can be read by the Pi. After the last pulse, the DOUT line will go high until the next measurement is made, thus preventing the same data point from being documented twice. The breakout board digitizes approximately 80 measurements per second, and takes approximately 86 μs to digitize a measurement.

Software was developed based on the HX711 library for Raspberry Pi developed by underdoeg[19] to take measurements from all sensors simultaneously. The code was altered to work on an interrupt schedule, so that sensors would be triggered as data became available. Measurements are stored in a *measurement* object, and are manipulated using functions of the object. This simplifies the code the user needs to interact with to simple commands, such as "r.get_data()". After measurements are taken, data is exported to a .csv file with five columns, one for each sensor and one for time. (See Appendix for code.)

3.3.2 Analysis

In this section we discuss code developed to automatically process data generated by the system. The first section deals with code designed to remove noise from data. The second section deals with code developed to quantify the initial impulse peak of a sit event. The third section discusses code developed to calculate time parameters from get-up-and-go data.

Outlier Removal The sensors, while usually reliable, occasionally generated noise points (see Figure 3). These points needed to be removed from the data before automatic processing of the data. An algorithm was developed to remove noise points from the data (Listing 2 in the Appendix). The algorithm works by taking the differences between subsequent points in each sensor dataset. These differences are then statistically analyzed to generate a mean and a standard deviation. The code then looks through the differences to identify differences which are greater than two standard deviations from the mean. If a difference is found that meets these parameters, the code checks to see if the surrounding points follow a specific pattern: a difference within two standard deviations from the mean, a difference greater than two standard deviations from the mean, followed later by a difference greater than two standard deviations from the mean in the opposite direction, then a difference within two standard deviations from the mean. If this pattern is found, the algorithm deletes all points within the range from all sensor datasets and the time dataset. Since noise points typically occur in only one sensor at a time, this does result in some loss of data; however, it simplifies processing of combined sensor data. Furthermore, the impact of the data loss is small, since the noise points are relatively infrequent, occurring no more than five times per second and often not appearing in measurements. This pattern successfully differentiates noise points from large peaks in sit events, and is able to identify noise points even when multiple noise points occur in sequence. Figure 3 demonstrates a successful run of this algorithm.

However, the data filter algorithm does not work in all cases. If there is an unusually large initial peak in the sit events, there will be a large standard deviation in the differences between subsequent points. This can lead to noise points being ignored,

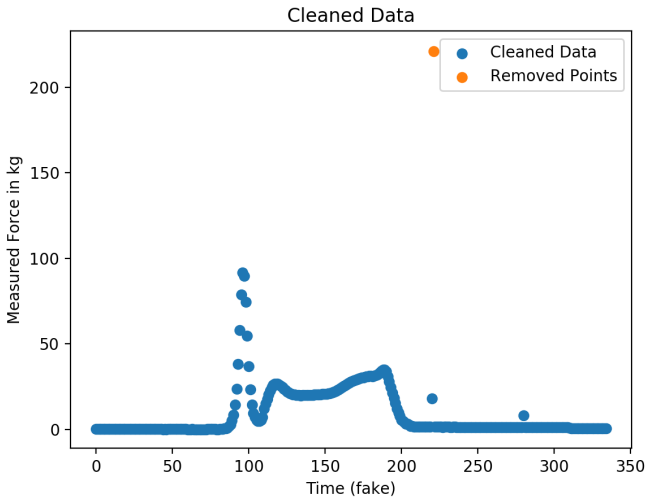

Figure 4: A sit-stand measurement with an unusually large initial peak (generated by Dr. Cooke purposely sitting down unusually hard in the chair). Noise points removed by the data filter are shown in orange. The data filter algorithm is no longer able to remove all of the noise points. The time vector used for this measurement was artificially generated, and thus the time axis is labeled "fake."

because they may fall within two standard deviations from the mean differences when the large differences found in unusually large peaks are included. This type of case can be seen in Figure 4. This could be a major issue if these large peaks are typical of the behavior of the elderly subjects of kinesiology studies. We hypothesize that subjects with less control over sitting will have larger initial peaks. If the peaks caused by poor sitting control are large enough to cause problems with outlier identification, the data filter will need to be re-designed. However more data is needed to determine if this is the case.

Impulse Quantification Python code was developed to automatically process the data, generating relevant parameters. An ideal sit-stand event is shown in Figure 5. As the subject sits down, there is an initial peak, which then levels to a steady state force which is approximately the weight of the subject. We were interested in measuring timing of sit-stand events during a get-up-and-go-test (both overall timing and the length of each event), the ratio of the initial peak force of a sit event to the steady state force of the event, and average steady state force for each sensor.

Code was developed to measure the steady state force and the ratio of the initial peak to the steady state force for individual sit events (see Listing 3 in the Appendix).

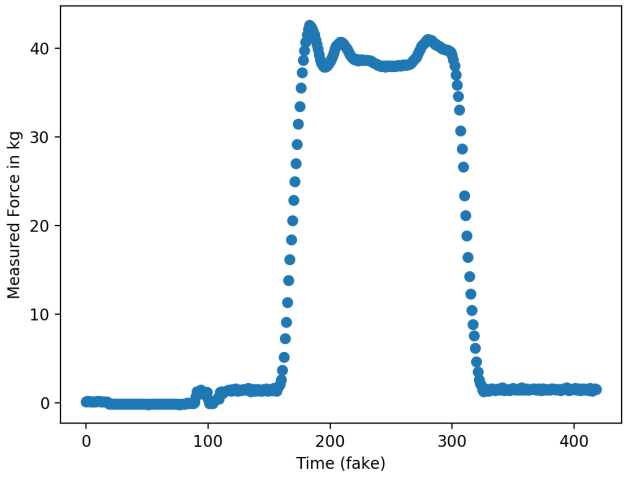

Figure 5: An ideal sit-stand measurement. Initially there is a peak, which levels out to a steady state force. The time vector used for this measurement was artificially generated, and thus the time axis is labeled "fake."

The algorithm identifies the initial peak by taking the maximum value of the dataset. The steady state region is then identified by looking at differences between the averages of subsequent groups of five points, starting from five points after the maximum. When the difference between two groups is larger than $\frac{1}{3}$ of the maximum value, the current position is used as the end of the steady state region. The steady state average force is obtained by taking the average of the points within the steady state region. The ratio is then obtained by dividing the value of the initial peak by the average steady state value.

This algorithm works well for ideal sit events (see Figure 6). However, not all datasets are ideal. The algorithm does not work well for datasets where the initial peak is not the max of the dataset. Furthermore, some datasets do not have a well-defined "steady-state" region, instead fluctuating wildly throughout the sit event. This makes the "steady state average force" number somewhat useless, as there is no real "steady state" in the data to average. These behaviors typically show up on single sensor datasets, not datasets with combined data from all sensors. This may indicate that the combined data is more suitable for ratio processing. It is possible that these fluctuations are caused by problems with leveling of the chair, leading to rocking of the chair during the sit events which causes variations in single sensor data but averages out in combined data. A full

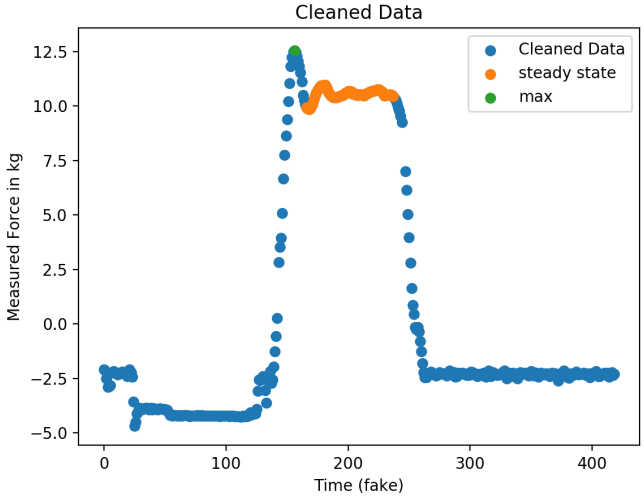

Figure 6: Processing of an ideal sit event. The steady state region is well-defined and the inital peak is the maximum of the dataset, leading to easy processing. The time vector used for this measurement was artificially generated, and thus the time axis is labeled "fake."

dataset processed using this algorithm can be found in the Appendix.

Edge Detection While the developed code works well for single sit-stand events, we were interested in developing code that would process get-up-and-go tests, which involve five sit-stand events. Additionally, we needed code which would calculate the total time of a get-up-and-go test as well as the time for each sit-stand event in the test. Code was developed which would identify the beginning and end of each sit-stand event in a get-up-and-go test. This information can be used to calculate timing parameters and to segment the get-up-and-go test into single sit-stand events to be processed by the algorithm discussed previously. The code works as an edge detector which marks the starting point of any sharp rising edge (indicates the start of an event) and the end point of any sharp falling edge (indicates the end of an event). The algorithm works well for the current data we have (see Figure 7).

To achieve edge detecting, the algorithm firstly captures all the data points on any rising or falling edge of the dataset. What the algorithm specifically does is that it calculates the force differences between any two consecutive data points and compares all the differences to a threshold value (0.1kg). Only when the differences of a point itself and three points before and after it are all larger than the threshold value, the

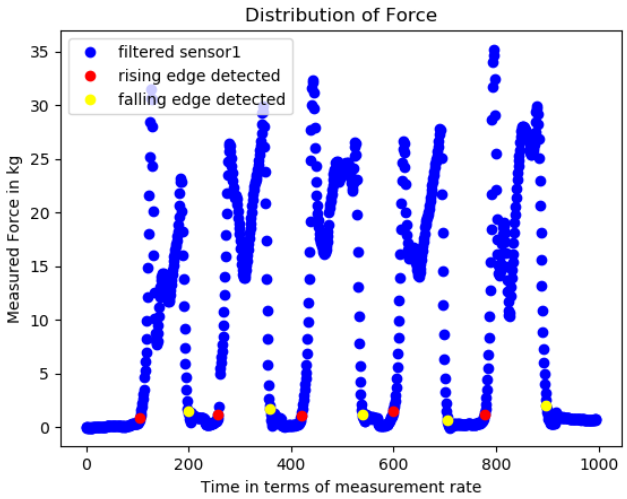

Figure 7: Edge detecting for a typical dataset involving multiple sit events. The algorithm successfully identifies the starting point and end point of each sit event.

algorithm identifies the point as a rising point. Similarly, when the differences of a point itself and its nearby points are all smaller than the negative threshold value (-0.1kg), the algorithm identifies the point as a falling point. (see Figure 8) The algorithm then defines a local minimum function which takes in an array of data points and identifies any point in the array whose neighboring points (one point before and after) are both larger than the point itself. The algorithm uses the local minimum function to identify the minimum point of each rising and falling edge (see Figure 9). The algorithm then runs the local minimum function again to remove any minimum points (red or yellow points) that are relatively large within each sit segment. The algorithm will keep running the local minimum function until the number of points remained matches the number of sit events (which is known to be five in a get-up-and-go test). Finally, time duration is calculated by taking the time difference between each remained point.

3.4 Future Directions

One of the most important improvements the current system needs is more rugged hardware design. The wiring of the current system is both fragile and complicated. If connections could be incorporated into a printed circuit board (pcb), components could be directly attached to the board. This would simplify troubleshooting procedures and make

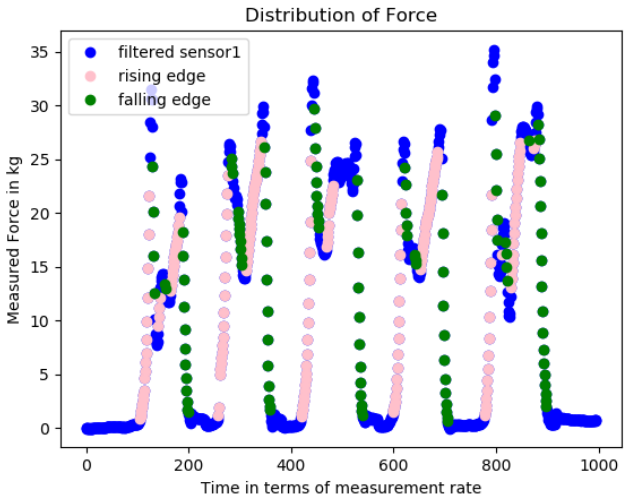

Figure 8: All the rising points (marked in pink) and falling points (marked in green) identified by the edge detecting algorithm for a typical dataset involving multiple sit events

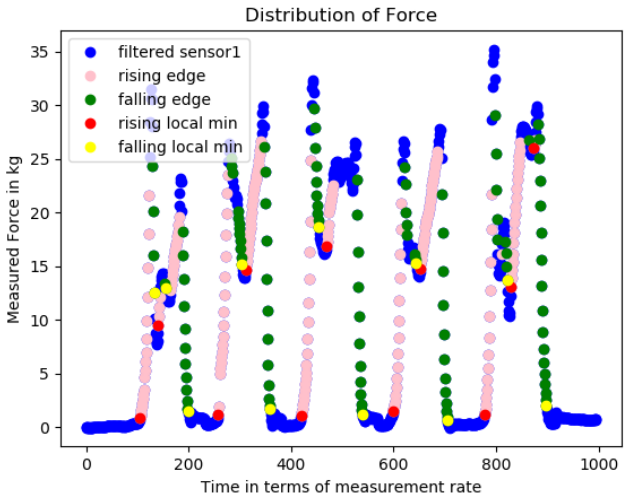

Figure 9: The minimum point of every rising and falling edge (marked in red and yellow respectively) for a typical dataset involving multiple sit events.

the system more durable.

Furthermore, the user interface for the code should be improved. While effort has been made to simplify the scripts necessary to run measurements through a class structure, the system could be further simplified. Ideally, all measurements would be run from a single, interactive script. This script would prompt the user to input necessary values (how long to measure, what output files should be named, etc.), then run the experiment and process and export values. Then the processing algorithms could be run on the output data, generating variables of interest and exporting them to a csv file. This type of system would require very little input or effort from the user.

Finally, while much progress was made on the processing scripts, further improvements need to be made. The single sit-stand processing algorithm needs to be integrated into the event-detecting algorithm so that it can be applied to get-up-and-go tests. Additionally, the single sit-stand processing algorithm needs to be re-designed so that it can more accurately identify relevant regions of sit-stand events. These algorithms should also be exhaustively tested on many types of get-up-and-go tests done by many different people, ensuring that they can handle all relevant measurement conditions.

Unfortunately, progress on this project was hindered by COVID-19-related closures. This prevented us from making improvements to our hardware. Furthermore, this somewhat limited our ability to collect additional data. While we had some ability to do tests by proxy through Dr. Cooke, the system was damaged in transport and virtual troubleshooting was unable to resolve the issues. We decided to focus on data processing and code improvements, since these were the easiest to implement remotely.

4 References

[1] H. Tan, L. Slivovksy, and A. Pentland, "A sensing chair using pressure distribution sensors," *IEEE/ASME Trransactions on Mechatronics*, vol. 6, pp. 261–268, 2001.

[2] *Smart Chair: What Can Simple Pressure Sensors under the Chairs' Legs Tell Us about User Activity?* UBICOMM 2013: The Seventh International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies, 2013.

[3] R. Bradley, "Electronic bathroom scale," 1989, uS Patent 4880069A.

[4] B. Fleming, D. Wilson, and D. Pendergast, "A portable, easily performed muscle power test and its association with falls by elderly personas," *Archives of Physical Medicine and Rehabilitation*, vol. 72, pp. 886–889, 1991.

[5] H. Tan, L. Slivovksy, and A. Pentland, "Measurement of stand-sit and sit-stand transitions using a miniature gyroscope and its application in fall risk evaluation in the elderly," *IEEE Transactions on Biomedical Engineering*, vol. 49, pp. 843–851, 2002.

[6] W. Zijlstra, R. Bisseling, S. Schlumbohm, and H. Baldus, "A body-fixed-sensor-based analysis of power during sit-to-stand movements," *Gait & Posture*, vol. 31, pp. 272–278, 2009.

[7] T. Paillard and F. Noé, "Techniques and methods for testing the postural function in healthy and pathological subjects," *BioMed Research International*, pp. 1–15, 2015.

[8] "Center of pressure (terrestrial locomotion)," accessed: November 2019. [Online]. Available: https://en.wikipedia.org/wiki/Center_of_pressure_%28terrestrial_locomotion%29

[9] M. Duarte and S. Freitas, "Revision of posturography based on force plate for balance evaluation," *Revista Brasiliera de Fisioterapia*, vol. 14, pp. 183–192, 2010.

[10] S. L. Whitney, J. L. Roche, G. F. Marchetti, C.-C. Lin, D. P. Steed, G. R. Furman, M. C. Musolino, and M. S. Redfern, "Revision of posturography based on force plate for balance evaluation," *Gait & Posture*, vol. 33, pp. 594–599, 2011.

[11] C. Curtuss, "System and method for assessing postural sway and human motion," 2012, uS Patent 9451916 B2.

[12] M. Mancini, A. Salarian, P. Carlson-Kuhta, C. Zampieri, L. King, L. Chiari, and F. B. Horak, "Isway: a sensitive, valid, and reliable measure of postural control," *Journal of NeuroEngineering and Rehabilitation*, vol. 9, p. 59, 2012.

[13] F. Scoppa, R. Capra, M. Gallamini, and R. Shiffer, "Clinical stabilometry standardization," *Gait & Posture*, vol. 37, pp. 290–292, 2013.

[14] J. Howcroft, E. Lemaire, J. Kofman, and W. McIlroy, "Elderly fall risk prediction using state posturography," *PLOS ONE*, vol. 33, pp. 602–607, 2017.

[15] I. Melser, "Postural stability in the elderly: a comparison between fallers and non-fallers," *Age and Ageing*, vol. 33, pp. 602–607, 2004.

[16] *Displacement of centre of mass during quite standing assessed using accelerometry in older fallers and non-fallers.* 2012 Annual International Conference of the IEEE Engineering in Medicine and Biology Society, 2012.

[17] L. Jihong and I. Ha, "Real-time motion capture for a human body using accelerometers," *Robotica*, vol. 19, pp. 601–610, 2000.

[18] "Human body weight," accessed: November 2019. [Online]. Available: https://en.wikipedia.org/wiki/Human_body_weight

[19] "Python port for raspberrypi to the hx711 breakout board." [Online]. Available: https://gist.github.com/underdoeg/98a38b54f889fce2b237

5 Appendix

5.1 A Processed Dataset

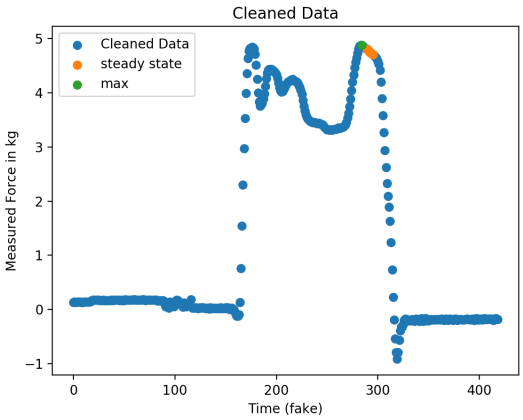

Figure 10: Processed data from sensor 1 during the measurement. This data was not processed well because the maximum was not the initial peak. Furthermore, there is no steady state behavior in this measurement. The time vector used for this measurement was artificially generated, and thus the time axis is labeled "fake."

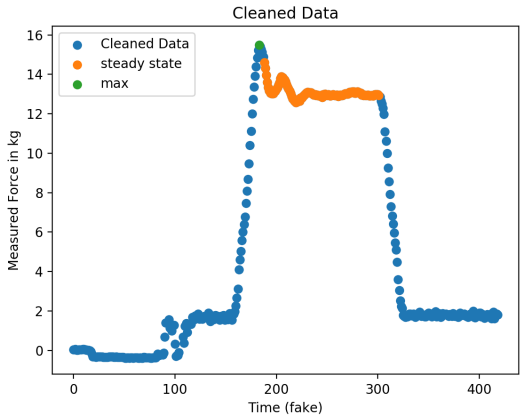

Figure 11: Processed data from sensor 2 during the measurement. This data closely resembles an ideal measurement and was processed well. However, more of the initial peak was included in the steady state behavior than should have been. The time vector used for this measurement was artificially generated, and thus the time axis is labeled "fake."

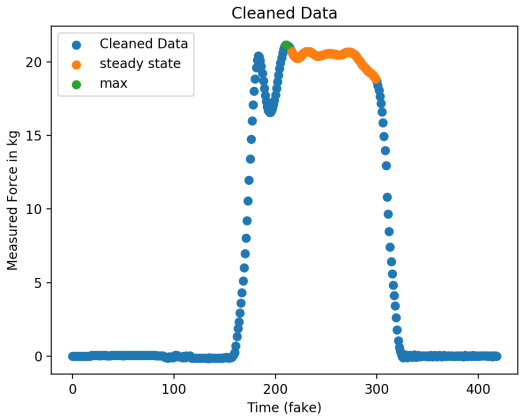

Figure 12: Processed data from sensor 3 during the measurement. While this dataset has both a well-defined peak and a well-defined steady state behavior, it was not processed well since the peak was not the maximum of the dataset. The time vector used for this measurement was artificially generated, and thus the time axis is labeled "fake."

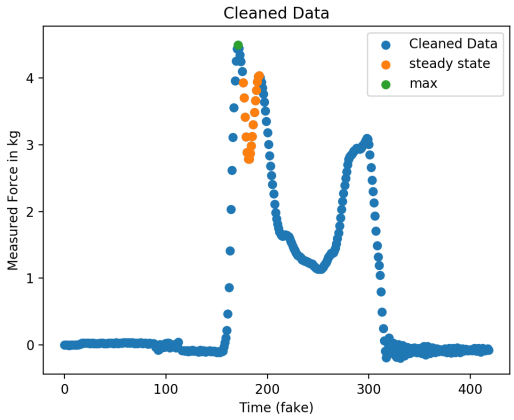

Figure 13: Processed data from sensor 4 during the measurement. The peak was correctly identified here; however, this dataset has no steady state behavior. The time vector used for this measurement was artificially generated, and thus the time axis is labeled "fake."

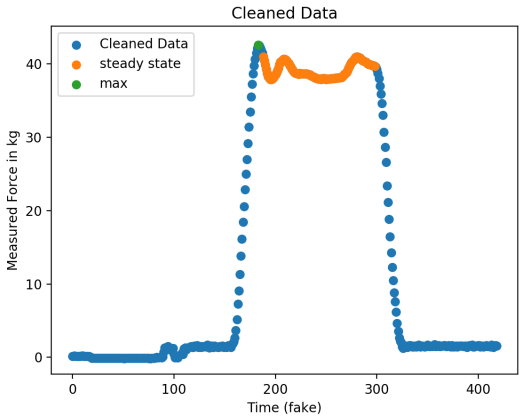

Figure 14: Processed data from the combined sensor data during the measurement. Notice that even though some of the single-sensor data in this system had nonideal behavior, the total combined behavior of the system is close to an ideal sit event. The time vector used for this measurement was artificially generated, and thus the time axis is labeled "fake."

Sensor	Ratio	Steady Average (kg)
1	1.02	4.76
2	1.18	13.08
3	1.04	20.30
4	1.32	3.40
Combined	1.09	39.11
		sum: 41.54

Table 3: Values generated from the processing of this dataset. The ratios vary over a rather large range due to processing errors on several sensors. However, the steady state averages appear to qualitatively give a good measure of the distribution over the sensors, and their sum has a rather close agreement to the steady state combined average.

5.2 Software

Listing 1: The code developed to run measurements on the raspberry pi.

```
import sys
import RPi.GPIO as GPIO
from hx711 import HX711
import matplotlib.pyplot as plt
import time
#import numpy as np
class sensors:
    def __init__(self,sens1= [14,15],sens2=[9,11],sens3=[23,24],sens4=[13,19]):

        self.hx_1 = HX711(sens1[0], sens1[1])
        self.hx_2 = HX711(sens2[0], sens2[1])
        self.hx_3 = HX711(sens3[0], sens3[1])
        self.hx_4 = HX711(sens4[0], sens4[1])

        self.hx_1.set_reading_format("MSB", "MSB")
        self.hx_2.set_reading_format("MSB", "MSB")
        self.hx_3.set_reading_format("MSB", "MSB")
        self.hx_4.set_reading_format("MSB", "MSB")

        self.hx_1.reset()
        self.hx_2.reset()
        self.hx_3.reset()
        self.hx_4.reset()

```

```
25          self.sens1_offset = 0
26          self.sens1_slope = 1
27          self.sens2_offset = 0
28          self.sens2_slope = 1
29          self.sens3_offset = 0
30          self.sens3_slope = 1
31          self.sens4_offset = 0
32          self.sens4_slope = 1
33
34          self.sens_1_values = []
35          self.sens_2_values = []
36          self.sens_3_values = []
37          self.sens_4_values = []
38
39          self.load_storage = []
40
41          self.times = []
42          self.sens_1_measured = False
43          self.sens_2_measured = False
44          self.sens_3_measured = False
45          self.sens_4_measured = False
46
47      def measure_sens_1(self):
48          self.sens_1_values = self.sens_1_values + [self.hx_1.get_weight(1)]
49          self.sens_1_measured = True
50
51      def measure_sens_2(self):
52          self.sens_2_values = self.sens_2_values + [self.hx_2.get_weight(1)]
53          self.sens_2_measured = True
54
55      def measure_sens_3(self):
56          self.sens_3_values = self.sens_3_values + [self.hx_3.get_weight(1)]
57          self.sens_3_measured = True
58
59      def measure_sens_4(self):
60          self.sens_4_values = self.sens_4_values + [self.hx_4.get_weight(1)]
61          self.sens_4_measured = True
62
63      def measure(self):
64          GPIO.add_event_detect(sens_1[0],GPIO.FALLING, callback = measure_sens_1())
65          GPIO.add_event_detect(sens_2[0],GPIO.FALLING, callback = measure_sens_2())
66          GPIO.add_event_detect(sens_3[0],GPIO.FALLING, callback = measure_sens_3())
67          GPIO.add_event_detect(sens_4[0],GPIO.FALLING, callback = measure_sens_4())
68          while (not self.sens_1_measured) or (not self.sens_2_measured) or (not self.
                sens_3_measured) or (not self.sens_4_measured):
69               pass
```

```
70          GPIO.remove_event_detect(sens_1[0])
71          GPIO.remove_event_detect(sens_2[0])
72          GPIO.remove_event_detect(sens_3[0])
73          GPIO.remove_event_detect(sens_4[0])
74          self.sens_1_measured = False
75          self.sens_1_measured = False
76          self.sens_1_measured = False
77          self.sens_1_measured = False
78
79      def grab_time(self,start_time):
80          t = time.time()
81          self.times = self.times + [t - start_time]
82
83      def measure_cali(self):
84          self.sens_1_values = self.sens_1_values + [self.hx_1.get_weight(5)]
85          self.sens_2_values = self.sens_2_values + [self.hx_2.get_weight(5)]
86          self.sens_3_values = self.sens_3_values + [self.hx_3.get_weight(5)]
87          self.sens_4_values = self.sens_4_values + [self.hx_4.get_weight(5)]
88
89
90      def calibrate(self,sensor,slope,offset = 0):
91          if sensor == 1:
92              self.sens1_slope = slope
93              self.sens1_offset = offset
94          elif sensor == 2:
95              self.sens2_slope = slope
96              self.sens2_offset = offset
97          elif sensor == 3:
98              self.sens3_slope = slope
99              self.sens3_offset = offset
100         elif sensor == 4:
101             self.sens4_slope = slope
102             self.sens4_offset = offset
103         else:
104             raise Exception('Problem␣with␣sensor␣input')
105
106     def convert_values(self):
107         for i in range(len(self.sens_1_values)):
108             self.sens_1_values[i] = self.sens_1_values[i]*self.sens1_slope - self.
                    sens1_offset
109         for i in range(len(self.sens_2_values)):
110             self.sens_2_values[i] = self.sens_2_values[i]*self.sens2_slope - self.
                    sens2_offset
111         for i in range(len(self.sens_3_values)):
112             self.sens_3_values[i] = self.sens_3_values[i]*self.sens3_slope - self.
                    sens3_offset
```

```
113         for i in range(len(self.sens_4_values)):
114             self.sens_4_values[i] = self.sens_4_values[i]*self.sens4_slope - self.
                    sens4_offset
115 
116     def tare(self):
117         self.sens1_offset = self.hx_1.get_weight(5)*self.sens1_slope
118         self.sens2_offset = self.hx_2.get_weight(5)*self.sens2_slope
119         self.sens3_offset = self.hx_3.get_weight(5)*self.sens3_slope
120         self.sens4_offset = self.hx_4.get_weight(5)*self.sens4_slope
121 
122 
123     def write_values(self,filename):
124         with open(filename+".csv", "w") as f:
125             for i in range(0,len(self.sens_1_values)):
126                 f.write(str(self.sens_1_values[i]) + ","+ str(self.sens_2_values[i]) + "
                        ,"+str(self.sens_3_values[i]) + ","+str(self.sens_4_values[i]) + ","
                         + str(self.times[i])"\n")
127 
128 ##         with open(filename+str(1)+".txt", "w") as f:
129 ##             for i in self.sens_1_values:
130 ##                 f.write(str(i) + "\n")
131 ##         with open(filename+str(2)+".txt", "w") as f:
132 ##             for i in self.sens_2_values:
133 ##                 f.write(str(i) + "\n")
134 ##         with open(filename+str(3)+".txt", "w") as f:
135 ##             for i in self.sens_3_values:
136 ##                 f.write(str(i) + "\n")
137 ##         with open(filename+str(4)+".txt", "w") as f:
138 ##             for i in self.sens_4_values:
139 ##                 f.write(str(i) + "\n")
140 
141 
142     def calibration_program(self,filename):
143         for i in range(1,7):
144             self.load_storage = self.load_storage + [float(input("Enter␣total␣weight␣
                    number␣"+str(i)+":"))]
145             raw_input("Load␣weight␣and␣press␣enter␣when␣ready")
146             print("Measuring␣...")
147             self.measure_cali() #averaging five measurements
148             print("Measurement␣complete!")
149 
150         print("Calibration␣Measurements␣done.␣Scaling...")
151         self.convert_values()
152 
153         with open(filename+".txt", "w") as f:
```

```
154             f.write("total_loaded,␣␣measured_1,␣␣measured_2,␣␣measured_3,␣␣measured_4␣\n
                    ")
155             for i in range(0,len(self.load_storage)):
156                 f.write(str(self.load_storage[i]) + ",␣"+str(self.sens_1_values[i]) + ",
                        ␣"+str(self.sens_2_values[i]) + ",␣"+str(self.sens_3_values[i]) + ",
                        ␣"+str(self.sens_4_values[i]) + "\n")
157         self.load_storage = []
158
159
160
161     def get_values(self):
162         print("sensor1:",str(self.sens_1_values))
163         print("sensor2:",str(self.sens_2_values))
164         print("sensor3:",str(self.sens_3_values))
165         print("sensor4:",str(self.sens_4_values))
166         print("total:", str(self.sens_1_values[0]+self.sens_2_values[0]+self.
                sens_3_values[0]+self.sens_4_values[0]))
167         print("times:", str(self.times))
168
169     def clear_values(self):
170         self.sens_1_values = []
171         self.sens_2_values = []
172         self.sens_3_values = []
173         self.sens_4_values = []
174         self.times = []
175
176     def plot_values(self):
177         plt.figure(1)
178         plt.plot(self.times,self.sens_1_values, label = "Sensor␣1")
179         plt.xlabel("Time␣in␣terms␣of␣measurement␣rate")
180         plt.ylabel("Measured␣Force␣in␣kg")
181         plt.title("Distribution␣of␣Force")
182         plt.plot(self.times,self.sens_2_values, label = "Sensor␣2")
183         plt.plot(self.times,self.sens_3_values, label = "Sensor␣3")
184         plt.plot(self.times,self.sens_4_values, label = "Sensor␣4")
185         plt.legend()
186         plt.show()
187         total = []
188         for i in range(len(self.sens_1_values)):
189             total = total + [self.sens_1_values[i] + self.sens_2_values[i] + self.
                    sens_3_values[i] + self.sens_4_values[i]]
190         plt.figure(2)
191         plt.plot(self.times,total)
192         plt.xlabel("Time␣in␣terms␣of␣measurement␣rate")
193         plt.ylabel("Measured␣Force␣in␣kg")
194         plt.title("Combined␣measurements")
```

```
195         plt.show()
196
197     def end_measurements(self):
198         GPIO.cleanup()
199         print("Done!")
200         sys.exit()
```

Listing 2: The code developed to removed outlier noise points from the data. This was designed to be imported into the software used to calculate ratios

```
import csv
import statistics
import matplotlib.pyplot as plt

def data_filter(data,time):

    def get_differences(array_in):
        array_out = []
        for i in range(len(array_in)-1):
            array_out = array_out + [array_in[i+1] - array_in[i]]
        return array_out

    def check(high,low,array,i):
        first_check = array[i] < low or array[i] > high
        return first_check

    # get differences between data points
    differences = get_differences(data)

    #find means of differences
    mean = sum(differences)/len(differences)

    #find standard deviations of differences
    dev = statistics.pstdev(differences)

    #generate comparison values
    low = mean - 2*dev
    high = mean + 2*dev

    #put checks for first and last here
    outliers_c = []
    outliers_t = []
    indexes = []
    i = 0
    while i < len(differences):
        n = i
        A = differences[i] < low
        B = differences[i] > high
        if A or B:
            for x in range(1,4):
                y = i + x

                if A and differences[y] > high:
```

```
                    C = differences[i-1] > low and differences[i-1] < high
                    D = differences[y+1] > low and differences[y+1] < high
                    if C and D:
                        n = y
                if B and differences[y] < low:
                    C = differences[i-1] > low and differences[i-1] < high
                    D = differences[y+1] > low and differences[y+1] < high
                    if C and D:
                        n = y
        if n != i:
            outliers_c = outliers_c + data[(i+1):(n+1)]
            outliers_t = outliers_t + time[(i+1):(n+1)]
            for x in range(i,n):
                indexes = indexes + [x]
            del data[(i+1):(n+1)]
            del time[(i+1):(n+1)]
            del differences[i:n]
            i = n+1
        else:
            i = i+1
    return [data, time, outliers_c, outliers_t, indexes]
```

Listing 3: The code developed to calculate ratios and distributions from a given dataset.

```
from data_filter import data_filter
import matplotlib.pyplot as plt
import statistics
import csv

def open_txt(filename_array):
    [filename1,filename2,filename3,filename4] = filename_array
    sensor_1 = []
    sensor_2 = []
    sensor_3 = []
    sensor_4 = []
    with open(filename1) as textfile:
        for row in textfile:
            sensor_1 = sensor_1 + [float(row.strip())]
    with open(filename2) as textfile:
        for row in textfile:
            sensor_2 = sensor_2 + [float(row.strip())]
    with open(filename3) as textfile:
        for row in textfile:
            sensor_3 = sensor_3 + [float(row.strip())]
    with open(filename4) as textfile:
        for row in textfile:
            sensor_4 = sensor_4 + [float(row.strip())]
    time = list(range(len(sensor_1)))

    return [sensor_1, sensor_2, sensor_3,sensor_4, time]

def open_csv(filename):
    sensor_1 = []
    sensor_2 = []
    sensor_3 = []
    sensor_4 = []
    time = []
    with open(filename) as csvfile:
        reader = csv.reader(csvfile)
        for row in reader:
            sensor_1 = sensor_1 + [float(row[0])]
            sensor_2 = sensor_2 + [float(row[1])]
            sensor_3 = sensor_3 + [float(row[2])]
            sensor_4 = sensor_4 + [float(row[3])]
            #time = time + [float(row[4])]
    time = list(range(len(sensor_1)))
    return [sensor_1, sensor_2, sensor_3,sensor_4, time]

def outlier_plot(time,data,outliers_t,outliers_c):
```

```
46      plt.figure(1)
47      plt.scatter(time,data, label = "Cleaned Data")
48      plt.scatter(outliers_t,outliers_t,label = "Removed Points")
49      plt.xlabel("Time (fake)")
50      plt.ylabel("Measured Force in kg")
51      plt.title("Cleaned Data")
52      plt.legend()
53      plt.show()
54
55  def filter_data(sensor_1,sensor_2,sensor_3,sensor_4,time):
56      [sensor_1, time, outliers_c, outliers_t, indexes] = data_filter(sensor_1,time)
57
58      for i in indexes:
59          del sensor_2[i]
60          del sensor_3[i]
61          del sensor_4[i]
62      outlier_plot(time,sensor_1,outliers_t,outliers_c)
63
64      return [sensor_1,sensor_2,sensor_3,sensor_4,time]
65
66  def clean_data(sensor_1,sensor_2,sensor_3,sensor_4,time):
67      [sensor_1, time, outliers_c, outliers_t, indexes] = data_filter(sensor_1,time)
68
69      for i in indexes:
70          del sensor_2[i]
71          del sensor_3[i]
72          del sensor_4[i]
73      outlier_plot(time,sensor_1,outliers_t,outliers_c)
74
75      [sensor_2, time, outliers_c, outliers_t, indexes] = data_filter(sensor_2,time)
76      for i in indexes:
77          del sensor_1[i]
78          del sensor_3[i]
79          del sensor_4[i]
80
81      outlier_plot(time,sensor_2,outliers_t,outliers_c)
82
83      [sensor_3, time, outliers_c, outliers_t, indexes] = data_filter(sensor_3,time)
84      for i in indexes:
85          del sensor_1[i]
86          del sensor_2[i]
87          del sensor_4[i]
88
89      outlier_plot(time,sensor_3,outliers_t,outliers_c)
90
91      [sensor_4, time, outliers_c, outliers_t, indexes] = data_filter(sensor_4,time)
```

```
92      for i in indexes:
93          del sensor_1[i]
94          del sensor_2[i]
95          del sensor_3[i]
96
97      outlier_plot(time,sensor_4,outliers_t,outliers_c)
98
99      combined = []
100     for i in range(len(sensor_1)):
101         combined = combined + [sensor_1[i] + sensor_2[i] + sensor_3[i] + sensor_4[i]]
102
103     return [sensor_1, sensor_2, sensor_3, sensor_4, combined, time]
104
105 def process(data,time):
106     peak = max(data)
107     index = data.index(peak)
108     diff = peak/3
109
110
111     number = 5
112     i = index + number
113     n = 0
114
115     while i < len(data) and n == 0:
116         avg1 = sum(data[i:i+6])/5
117         avg2 = sum(data[i+6:i+11])/5
118         if avg1-avg2 > diff:
119             n = i
120         i = i+1
121     if n != 0:
122         i = index + number
123         steady = sum(data[i:n])/len(data[i:n])
124         ratio = peak/steady
125     else:
126         i = 0
127         index = 0
128         ratio = 0
129         steady = 0
130     return [data, steady, ratio, i, n, index]
131
132 def calculate_ratios(filename,txt):
133     #read in files
134     if txt == 1:
135         [sensor_1, sensor_2, sensor_3,sensor_4, time] = open_txt(filename)
136     else:
137         [sensor_1, sensor_2, sensor_3,sensor_4, time] = open_csv(filename)
```

```
138
139     #remove noise points
140     [sensor_1, sensor_2, sensor_3,sensor_4, combined, time] = clean_data(sensor_1,
            sensor_2, sensor_3,sensor_4, time)
141
142     sensor_1_out = process(sensor_1,time)
143     print("one done")
144     sensor_2_out = process(sensor_2,time)
145     print("two done")
146     sensor_3_out = process(sensor_3,time)
147     print("three done")
148     sensor_4_out = process(sensor_4,time)
149     print("four done")
150     combined_out = process(combined,time)
151     print("five done")
152
153     return [sensor_1_out, sensor_2_out, sensor_3_out, sensor_4_out, combined_out,time]
154 def ratio_plot(time,data,i,n,index):
155     plt.figure(1)
156     plt.scatter(time,data, label = "Cleaned Data")
157     plt.scatter(time[i:n],data[i:n],label = "steady state")
158     plt.scatter(time[index],data[index],label = "max")
159     plt.xlabel("Time (fake)")
160     plt.ylabel("Measured Force in kg")
161     plt.title("Cleaned Data")
162     plt.legend()
163     plt.show()
164
165 if __name__ == "__main__":
166     files = ['18Feb1.txt','18Feb2.txt','18Feb3.txt','18Feb4.txt']
167     [sensor_1_out, sensor_2_out, sensor_3_out, sensor_4_out, combined_out,time] =
            calculate_ratios(files,1)
168     [sensor_1, steady1, ratio1, i1, n1, index1] = sensor_1_out
169     [sensor_2, steady2, ratio2, i2, n2, index2] = sensor_2_out
170     [sensor_3, steady3, ratio3, i3, n3, index3] = sensor_3_out
171     [sensor_4, steady4, ratio4, i4, n4, index4] = sensor_4_out
172     [combined, steady5, ratio5, i5, n5, index5] = combined_out
173
174     ratio_plot(time,sensor_1,i1,n1,index1)
175     ratio_plot(time,sensor_2,i2,n2,index2)
176     ratio_plot(time,sensor_3,i3,n3,index3)
177     ratio_plot(time,sensor_4,i4,n4,index4)
178     ratio_plot(time,combined,i5,n5,index5)
179
180     print("sensor_1 ratio:", ratio1, "sensor_1 steady average:", steady1)
181     print("sensor_2 ratio:", ratio2, "sensor_2 steady average:", steady2)
```

```
182     print("sensor_3 ratio:", ratio3, "sensor_3 steady average:", steady3)
183     print("sensor_4 ratio:", ratio4, "sensor_4 steady average:", steady4)
184     print("combined ratio:", ratio5, "combined steady average:", steady5)
```

Listing 4: The code developed to calculate time duration within and between each sit event from a given dataset involving multiple sit events.

```
import matplotlib.pyplot as plt
import numpy as np
import csv
from data_filter import data_filter

open_filename = input('Enter␣the␣name␣of␣the␣file␣you␣want␣to␣import:␣')
event_num = float(input('Enter␣the␣number␣of␣sit␣events␣this␣dataset␣has:␣'))
print('')

def open_csv(filename):
    sensor_1 = []
    sensor_2 = []
    sensor_3 = []
    sensor_4 = []
    time = []
    with open(filename) as csvfile:
        reader = csv.reader(csvfile)
        for row in reader:
            sensor_1 = sensor_1 + [float(row[0])]
            sensor_2 = sensor_2 + [float(row[1])]
            sensor_3 = sensor_3 + [float(row[2])]
            sensor_4 = sensor_4 + [float(row[3])]
    time = list(range(len(sensor_1)))
    return [sensor_1, sensor_2, sensor_3,sensor_4, time]

def clean_data(sensor_1,sensor_2,sensor_3,sensor_4,time):
    [sensor_1, time, outliers_c, outliers_t, indexes] = data_filter(sensor_1,time)
    for i in indexes:
        del sensor_2[i]
        del sensor_3[i]
        del sensor_4[i]
    [sensor_2, time, outliers_c, outliers_t, indexes] = data_filter(sensor_2,time)
    for i in indexes:
        del sensor_1[i]
        del sensor_3[i]
        del sensor_4[i]
    [sensor_3, time, outliers_c, outliers_t, indexes] = data_filter(sensor_3,time)
    for i in indexes:
        del sensor_1[i]
        del sensor_2[i]
        del sensor_4[i]
    [sensor_4, time, outliers_c, outliers_t, indexes] = data_filter(sensor_4,time)
    for i in indexes:
```

```
44          del sensor_1[i]
45          del sensor_2[i]
46          del sensor_3[i]
47      combined = []
48      for i in range(len(sensor_1)):
49          combined = combined + [sensor_1[i] + sensor_2[i] + sensor_3[i] + sensor_4[i]]
50
51      return [sensor_1, sensor_2, sensor_3, sensor_4, combined, time]
52
53  def plot_data(x,y,label_name):
54      plt.plot(x,y,'bo',label = label_name)
55      plt.xlabel("Time in terms of measurement rate")
56      plt.ylabel("Measured Force in kg")
57      plt.title("Distribution of Force")
58
59  #create a list of differences between consecutive points
60  def get_differences(array_in):
61      array_out = []
62      for i in range(len(array_in)-1):
63          array_out = array_out + [array_in[i+1] - array_in[i]]
64      return array_out
65
66  def local_min(array_x, array_y):
67      local_min_index = []
68      local_min_y = []
69      local_min_x = []
70      for i in range(1,len(array_y)-1):
71          if array_y[i] < array_y[i-1] and array_y[i] < array_y[i+1]:
72              local_min_index += [i]
73              local_min_y += [array_y[i]]
74              local_min_x += [array_x[i]]
75      return local_min_x, local_min_y
76
77
78  def get_time_duration(data,time,sensor_num):
79      plt.figure(sensor_num)
80      plot_data(time,data,'filtered sensor'+str(sensor_num))
81      data_diff = get_differences(data)
82
83      #detect rising and falling edges
84      diff_thres = 0.1 #set the threshold value
85      points = 3 #set the number of nearby points
86
87      rising_time = []
88      rising_force = []
89      falling_time = []
```

```
    falling_force = []
    i = points
    while i < len(data_diff)-points:
        data_diff_subset = [data_diff[i-p] for p in range(points,0,-1)] + [data_diff[i]]
             + [data_diff[i+p] for p in range(1,points+1)]
        if all(item > diff_thres for item in data_diff_subset):
            rising_time += [time[i]]
            rising_force += [data[i]]
        if all(item < -diff_thres for item in data_diff_subset):
            falling_time += [time[i]]
            falling_force += [data[i]]
        i = i+1

    #find local min of risng and falling edges
    try:
        rising_time_lm, rising_force_lm = local_min(rising_time,rising_force)
        while len(rising_time_lm) > event_num-1:
            rising_time_lm, rising_force_lm = local_min(rising_time_lm,rising_force_lm)
        rising_time_lm = [rising_time[0]] + rising_time_lm
        rising_force_lm = [rising_force[0]] + rising_force_lm
        plt.plot(rising_time_lm,rising_force_lm,'o',color = 'red', label = 'rising edge
            detected')

        falling_time_lm, falling_force_lm = local_min(falling_time,falling_force)
        while len(falling_time_lm) > event_num-1:
            falling_time_lm, falling_force_lm = local_min(falling_time_lm,
                falling_force_lm)
        falling_time_lm =  falling_time_lm +[falling_time[-1]]
        falling_force_lm = falling_force_lm +[falling_force[-1]]
        plt.plot(falling_time_lm,falling_force_lm,'o',color = 'yellow', label = 'falling
            edge detected')
        plt.legend()

        total_time = falling_time_lm[-1] - rising_time_lm[0]
        time_within = [falling_time_lm[i] - rising_time_lm[i] for i in range(0,len(
            falling_time_lm))]
        time_between = [rising_time_lm[i+1] - falling_time_lm[i] for i in range(0,len(
            falling_time_lm)-1)]
        print('total time duration for sensor'+ str(sensor_num) + ': ' + str(total_time)
            )
        print('time duration for each sit event for sensor'+ str(sensor_num) + ': ',
            time_within)
        print('time between each sit event for sensor'+ str(sensor_num) + ': ',
            time_between)
        print('list of every rising time for sensor'+ str(sensor_num) + ': ',
            rising_time_lm)
```

```
126         print('list␣of␣every␣falling␣time␣for␣sensor'+ str(sensor_num) + ':␣',
                falling_time_lm)
127         print('')
128     except:
129         print('sensor'+ str(sensor_num)+'␣error')
130         print('')
131
132
133 if __name__ == "__main__":
134     [sensor_1, sensor_2, sensor_3, sensor_4, time] = open_csv(open_filename)
135     [sensor_1_fil, sensor_2_fil, sensor_3_fil, sensor_4_fil, combined_fil, time] =
            clean_data(sensor_1,sensor_2,sensor_3,sensor_4,time)
136
137     get_time_duration(sensor_1_fil, time,1)
138     get_time_duration(sensor_2_fil, time,2)
139     get_time_duration(sensor_3_fil, time,3)
140     get_time_duration(sensor_4_fil, time,4)
141
142     plt.show()
```