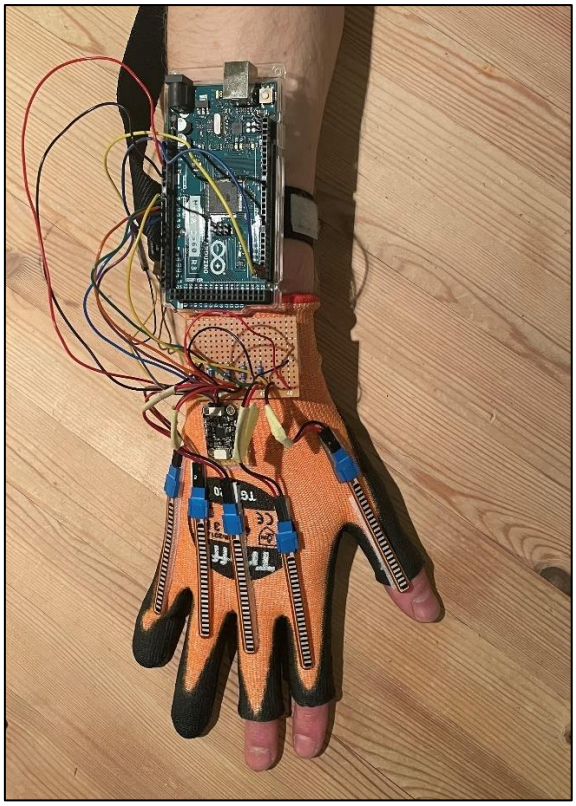
University of Bath | Department of Electrical and Electronic Engineering

MiGlove – Multimodal Glove and Machine Learning for Hand Gesture and Action Recognition

MEng Final Year Project Report

Report Word Count: 13183
Abstract Word Count: 281



Student Name: Joseph Hitchen
Student Number: 199147441
MEng(Hons) Integrated Mechanical and Electrical Engineering with Yearlong Work Placement
Academic Year: 5th

Supervisor: Dr Uriel Martinez Hernandez
Assessor: Dr Chanel Fallon

An ethics document for the MiGlove (Project ID 3517) was approved on 6th February 2024.

ABSTRACT

In the manufacturing industry, achieving seamless collaboration between workers and robots emerges as a critical imperative. Traditional automation, while successfully streamlining processes, has also raised concerns regarding job displacement, task flexibility, and worker safety. Cobots, offering advanced safety features and heightened adaptability, present a favourable approach for addressing these challenges. However, their transformative potential is restricted by safety regulations related to proximity to workers and limited task comprehension. Effective collaboration between humans and cobots is crucial for optimising manufacturing operations. Through the integration of advanced Human Gesture Recognition (HGR) into collaborative robotic systems, not only can worker safety be ensured, but also visual comprehension of assembly tasks can be fostered, thereby maximising efficiency and accelerating workflow.

This project addresses the underutilisation of collaborative robots (cobots) in the manufacturing industry. A comprehensive literature review reveals the effectiveness of wearable, multimodal sensing gloves equipped with IMUs and flex sensors in enhancing collaborative robotics. Machine learning techniques, particularly Long Short-Term Memory (LSTM) models, have shown promising results in gesture recognition using such gloves. Subsequently, this report details the design process for creating a multimodal glove capable of capturing wielder gesture data, covering key decisions, challenges, and design iterations. Four distinct gestures—hammering, sawing, screwdriving, and 'no gesture'—were recorded and used to train an LSTM model created in Python. The model achieved 100% accuracy on the testing dataset and 85% accuracy with live gesture data. Simulation experiments conducted in CoppeliaSim validate the efficacy of gesture recognition, with the robot mimicking the recognised gestures. The MiGlove can be considered a cost-effective yet accurate solution for gesture recognition, with the potential to revolutionise collaborative robotics in manufacturing and ultimately improving productivity and safety in industrial settings.

1 TABLE OF CONTENTS

2 TABLES OF INFORMATION

2.1 TABLE OF ACRONYMS

ACRONYM	DEFINITION
MSD	Musculoskeletal disorders
ISO	International Organization for Standardization
HGR	Human gesture recognition
SME	Small and Medium-sized Enterprises
CNN	Convolutional Neural Network
IMU	Inertial Measurement Unit
SVM	Support Vector Machine
ANN	Artificial Neural Network
LSTM	Long Short-Term Memory
R-CNN	Region-based Convolutional Neural Network
ASL	American Sign Language
CSL	Chinese Sign Language
DFFN	Deep Feature Fusion Network
DCNN	Deep Convolutional Neural Network
DTW	Dynamic Time Warping
SEMG	Surface Electromyography
IDE	Integrated Development Environment
API	Application Programming Interface
MCU	Microcontroller
I2C	Inter-Integrated Circuit
MCP	Metacarpophalangeal
IP	Interphalangeal
PIP	Proximal Interphalangeal
IO	Inputs/Output
UART	Universal Asynchronous Receiver-Transmitter
USB	Universal Serial Bus
BPS	Bits per second
GPIO	General-Purpose Input/Output
SPI	Serial Peripheral Interface
ADC	Analogue-to-Digital Converter
COM	Communication
CSV	Comma-Separated Values
UID	Unique Identifier
RNN	Recurrent Neural Network

2.2 TABLE OF FIGURES

3 INTRODUCTION AND CONTEXT

3.1 ROBOTS AND COBOTS: OPTIMISING THE MANUFACTURING INDUSTRY

The evolution of industry has seen major shifts that have transformed production methods and economies worldwide. It began with the First Industrial Revolution in the 18th century, driven by the need to meet the increasing demands of a growing global population. The mechanisation of production led to volumes skyrocketing by up to eight times [1]. Again, in the 19th century, a Second Industrial Revolution brought electricity and assembly line production techniques which greatly reduced manufacturing durations and costs [1]. The Third Industrial Revolution came with the introduction of robots, capable of automating entire production processes [2]. This particularly benefitted industries with repetitive tasks like automotive and electronics [3].

With the focus on maximising throughput, large and powerful robots were heavily utilised to perform these repetitive tasks. However, market demands shifted towards flexibility and customisation, rendering traditional automation methods less effective. This shift established the foundation for the "Industry 4.0 paradigm," [4] emphasising enhanced human-machine interaction. Smart factories

became prevalent, where production systems, components and people communicate via a network and production is nearly entirely autonomous [1].

The number of robots per 1000 workers has steadily increased over the past 22 years. Figure 1 shows evidence of this in two of the world's leading manufacturing nations: USA and Germany [5]. Evidently, more manufacturers are prepared to invest in robots to improve efficiency and remain at the forefront of the manufacturing industry.

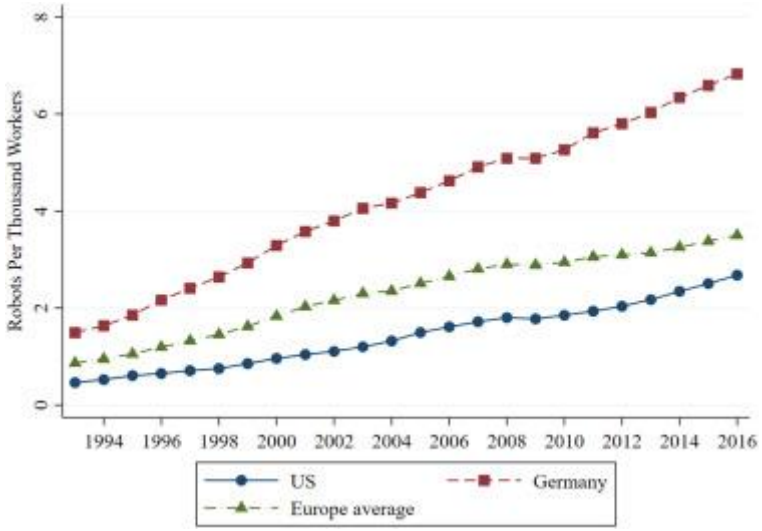


Figure 1: Trends in Robot Adoption in USA, Germany, and European average from 1993 to 2016 [6].

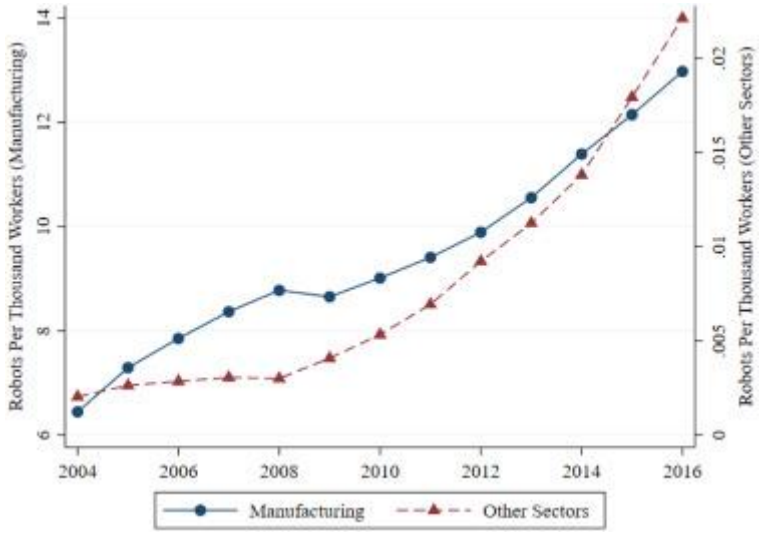


Figure 2: Evolution of Industrial Robots in the US, Manufacturing vs Other Sectors [6].

Figure 2 compares robot inclusion in the manufacturing industry against other sectors in the USA. Far more robots are being used in manufacturing compared to other sectors – between 2014 and 2016, the rate of increase in number of robots per 1000 workers was about 0.6 per year, whereas in other sectors the rate of increase was much lower at about 0.005 per year. Even if growth in other industries continues at the same rate, the number of robots used in manufacturing will vastly outnumber other sectors for the foreseeable future, warranting its investigation.

Conversely, robots do have high up-front investment and operational costs. Furthermore, transforming a workplace to accommodate robots has various operational and social implications: some workers are

apprehensive about robotic automation, with almost 40% believing that their jobs are 50% likely to be replaced by machinery [7]. However, results from Germany indicate the contrary – with no disruptive job losses, robotisation has contributed to improvements in workplace safety and reduced worker requirements to perform physically demanding tasks, without adversely affecting workers' physical and mental health [6]. Furthermore, Amazon say that introducing robots has created over a million human jobs since 2012 [8]. Nevertheless, the fear of job displacement prevents some larger companies with stronger workers' union influence from introducing robots, causing their efficiency and performance to suffer in comparison to those that have successfully automated. A study conducted by Smids et al. [9] examined the ethics of robotisation and offered solutions for how to – and how not to – implement them in the workplace. Rather than having workers compete against robots, the study implies that human-robot collaboration could increase accessibility to certain jobs like surgery by simplifying the role, creating more opportunities for workers who lack the necessary physical abilities. Collaborative robots, or "cobots", can alleviate concerns regarding job displacements and enable workers to acquire new skills through collaboration, leading to a notable increase in manufacturing efficiency without compromising workforce morale.

According to Keshvarparast et al. [4], cobots provide enhanced safety when working alongside humans. Cobots operate at lower speeds and forces, reducing the risk of harm in case of impact and making them safer to work with than conventional robots. Additionally, cobots are seen as complementary rather than replacement technology for human workers, as they can collaborate with humans to improve productivity without jeopardising job security. This collaborative approach not only enhances throughput but also preserves workers' livelihoods. Furthermore, cobots are easier to program and can be quickly located and relocated to different tasks or workstations. This versatility makes them well-suited to dynamic manufacturing environments where tasks may vary, and adaptability is crucial for efficiency.

Additionally, cobots do not possess biological cells, which must regenerate, grow and adapt. Instead, they can be optimised for performance of a single function [10]: they can be made stronger and faster by switching to larger, more powerful actuators; reach further by using longer links; or last longer with larger batteries. Albeit each upgrade makes them larger and/or heavier, limiting portability, dexterity and versatility [10], hence a compromise is typically required.

Cobots can perform more dangerous tasks which typically have higher risk of worker injury, including the lifting of heavy objects. They can also perform repetitive actions which may normally cause musculoskeletal disorders (MSDs) in humans over time, such as bending and picking objects up. Other tasks include the handling of hazardous materials, welding, cutting and handling of sharp objects and working at height, each of which help to limit worker exposure to harm.

As such, cobots are becoming increasingly more popular than regular robots. Figure 3 shows that the revenue growth from cobots is projected to grow by almost 20% across the next two years, indicating that many manufacturing companies are choosing to target cobots over industrial robots.

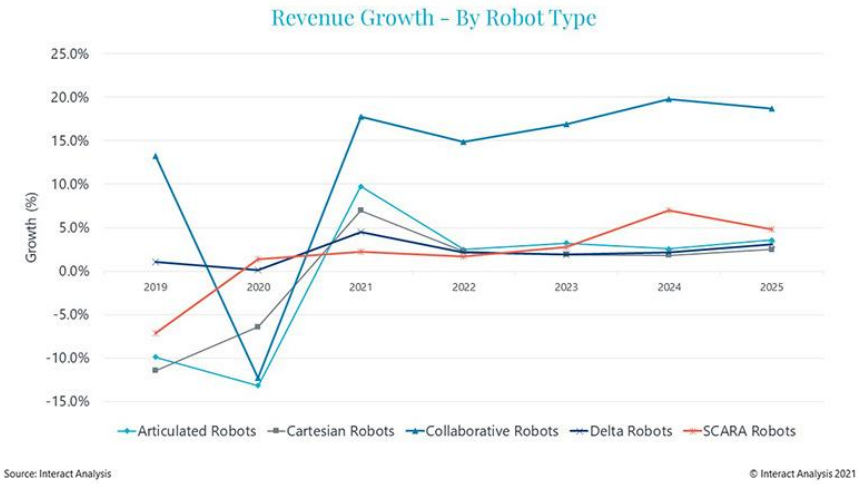


Figure 3: Graph to show revenue growth by robot type [11].

3.2 WHAT IS COLLABORATION?

There are different levels of human-robot interaction, as depicted in Figure 4.

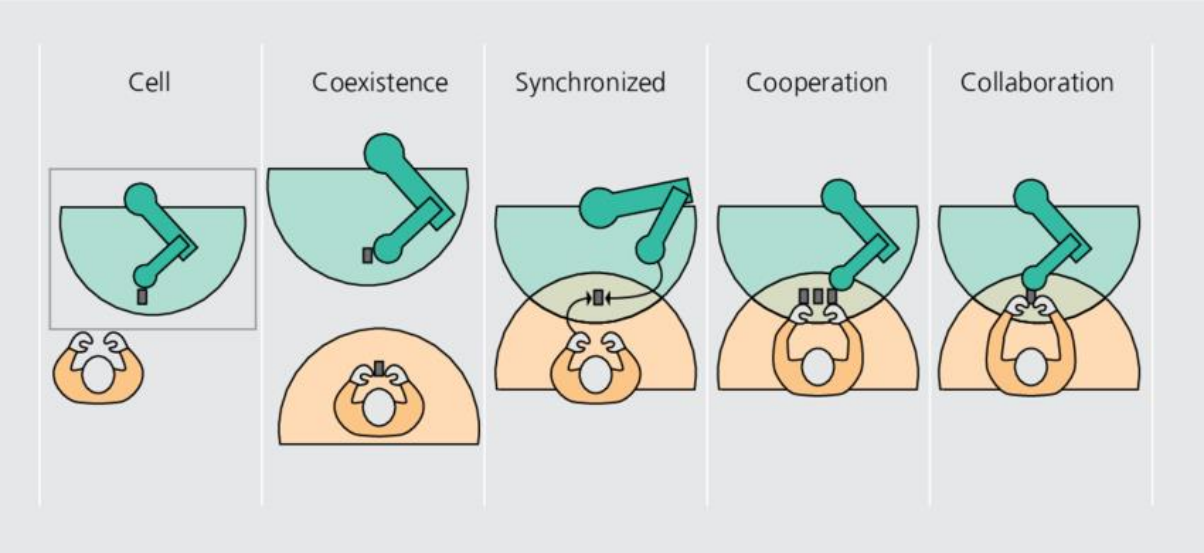


Figure 4: Types of human-robot interaction [12].

Cell represents a robot operating autonomously within its own designated workspace, with no direct interaction with human workers. Whilst coexistent and synchronised tasks do involve humans and robots occupying the same workspace, there is no mutual assistance – the human and robot either work on separate parts, or the involvement of one succeeds the other. These situations are very safe due to limited interaction between the two, but processing speed can be slow and lack flexibility, since the robot will follow predetermined routines and cannot respond effectively to unexpected scenarios, i.e. error handling. Therefore, quality assurance without human intervention is limited. Furthermore, robots do not develop an intuitive sense of physics [13]. Robots must be programmed and calibrated to manage forces and perform certain actions in advance. Very elaborate machine learning would be required for them to achieve experimental learning the way humans do. Ultimately, this means for tasks which humans can perform through generalised understanding and improvisation, the same level of

performance may not be achievable by robots due to their lack of adaptability (e.g. in emergency situations).

Conversely, cooperative and collaborative interaction leverages the human worker's problem-solving and creativity while simultaneously delegating tasks to the robot to speed up assembly. Robots can easily detect patterns in large amounts of data, run complex manufacturing processes, and quickly perform computationally intensive calculations; these tasks typically take humans much longer to complete. Combining both skill sets can greatly optimise performance while reducing risk of harm to human workers. It also enables intuitive error handling and flexibility during manufacturing and assembly.

3.3 EXISTING COLLABORATIVE TECHNOLOGY IN INDUSTRY

Amazon have deployed warehouse robots which bring items to human handlers to be packaged and labelled for dispatch. They can move entire shelving units while ensuring they do not collide with and injure any human workers. However, they are currently restricted to working in designated areas, which offer limited access to human workers, thereby safeguard against workplace accidents. That said, trials for a new model called "Bert", able to safely navigate the entire factory floor, are currently underway [8]. Symbio Robotics provide robots for tasks ranging from welding and spray painting to intricate tasks like component installation and quality testing, primarily used by car manufacturers like Ford and Toyota [8]. Automating these tasks is incredibly complex and only recently became available. However, these robots are not collaborative, missing out on aforementioned benefits.

ISO (International Organization for Standardization) has guidelines and standards which influence human-cobot interaction. ISO/TS 15066 discusses safe design and operation of cobots; it also outlines safety requirements and risk assessment methodologies for human-cobot interactions. These include force and pressure limits, speed and separation monitoring, and collaborative workspace design [14]. Safety regarding general robot-human interaction is outlined in ISO 13849-1 [15]. Currently, the types for collaborative tasks available may be limited due to these regulations. Enhanced human gesture recognition (HGR) could ease regulations and constraints, allowing manufacturers to explore new collaborative ventures and capitalise on opportunities for profit.

The adoption of cobots presents many challenges for Small and Medium-sized Enterprises (SMEs). These include the need to make cobots financially viable for small-scale production and ensuring a satisfactory long-term return on investment [16]. Additionally, identifying suitable tasks to allocate to them for maximising value added may be difficult due to strict regulations limiting their utilisation. Moreover, maintaining product quality is crucial; cobots may struggle to recognise defects on the production line, affecting output quality. This could disproportionately impact the profit margins of SMEs.

There are relatively few instances where cobots are being utilised effectively to enhance the manufacturing industry. Evidently, a variety of factors are responsible. Equipping collaborative robotic systems with advanced HGR would not only allow them to remain aware of the workers actions from a safety standpoint but would also enable visual comprehension of the assembly task sequence, optimising task allocation and accelerating workflow.

4 LITERATURE REVIEW

This literature review explores the application of hand gesture recognition in enhancing collaborative robotics. Whilst the primary focus is on assembly applications, methodologies with broader relevance to the overarching goal are also considered. The review will gauge the effectiveness of different gesture classification approaches, as well as evaluate the choice of sensors and machine learning models used

in these studies, aiming to identify their suitability and potential contribution to improving human-robot interaction within assembly contexts.

4.1 Review of Gesture Recognition Methods and Technologies

One of the most significant studies, conducted by Male and Hernandez, combined a vision system with human gesture recognition (HGR) to predict the current assembly state of a manufactured part [17]. A Kinect V1 vision sensor was mounted above the part to capture images showing build progression. These images were converted to greyscale and classified using a Convolutional Neural Network (CNN) to determine the current state of the part. Simultaneously, the user wore 3 Shimmer3 Inertial Measurement Units (IMUs) which streamed 3-axis acceleration and gyro data from the user. These were placed around the user's dominant hand: on top of the hand, the wrist and upper arm. The IMU data was used by another CNN to classify the user's current action. The study included data from 2 users. The recognition accuracies were 77% and 81% respectively. The outputs of these models were combined and tested against various fusion methods: mean, weighted average, Support Vector Machine (SVM), Naïve Bayesian, Artificial Neural Network (ANN) and Long Short-Term Memory (LSTM). Their architecture is shown in Figure 5.

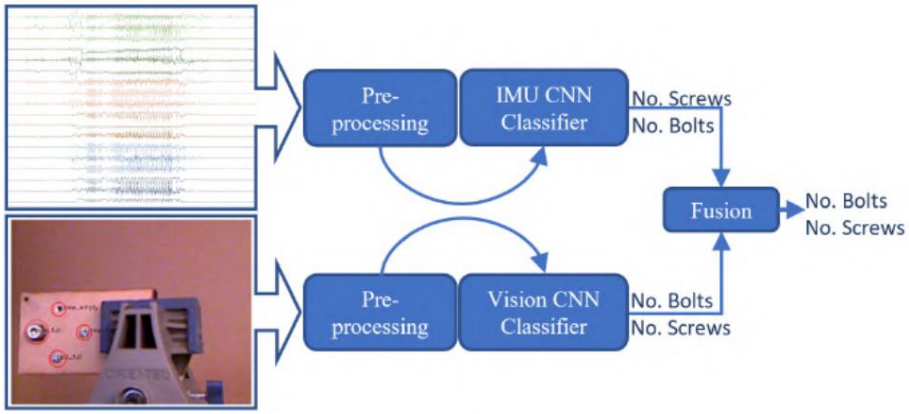


Figure 5: Overall fusion system structure [17].

The LSTM produced the highest accuracy of 93%, followed by the Bayesian (87%) and SVM (86%) methods. The classification results were presented as confusion matrices in Figure 6, with the LSTM producing the most favourable results.

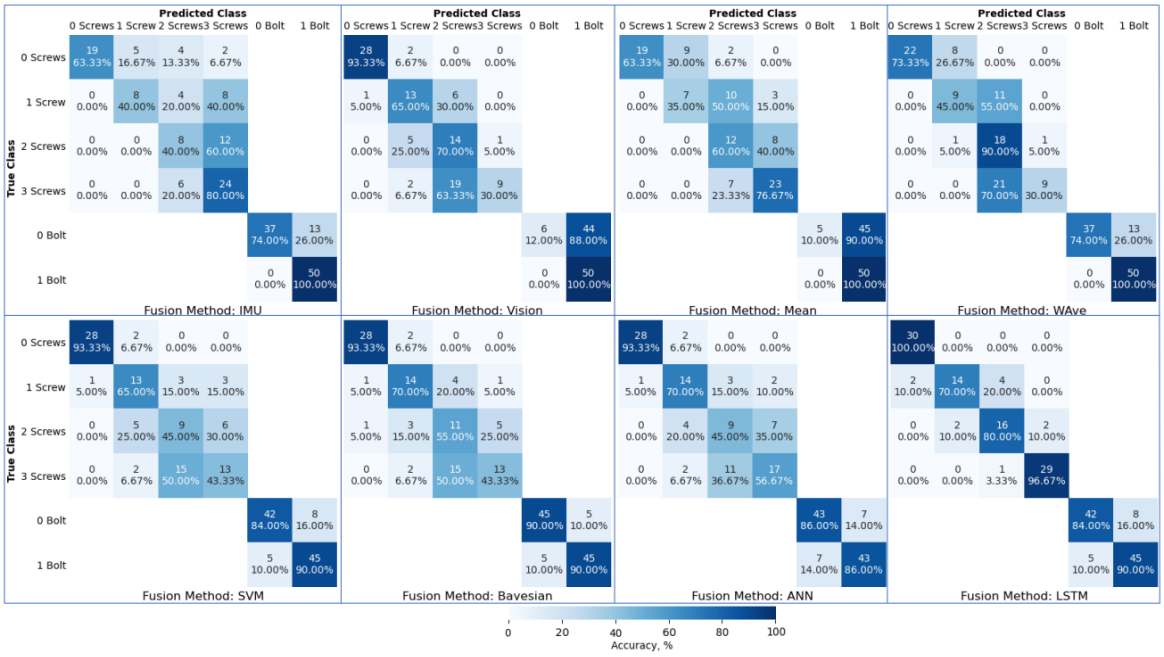


Figure 6: Confusion matrices for fusion methods [17].

Like Male and Hernandez, Nuzzi et al. [18] and Lin et al. [19] also combined the XBOX Kinect camera with CNN classifiers. Nuzzi et al. developed a gesture recognition system in MATLAB, using a Faster Region-based CNN (R-CNN) object detector to recognise different positions of both hands (see Figure 7), which were used to control a cobot [18]. Conversely, Lin et al. focused on recognising 7 different single-handed gestures (shown in Figure 8), with the CNN model in Figure 9.

Figure 7: Use of gestures to stop, start and signal a collaborative robot to move [18].

Figure 8: Lin et al.'s selected hand gestures to be recognised [19].

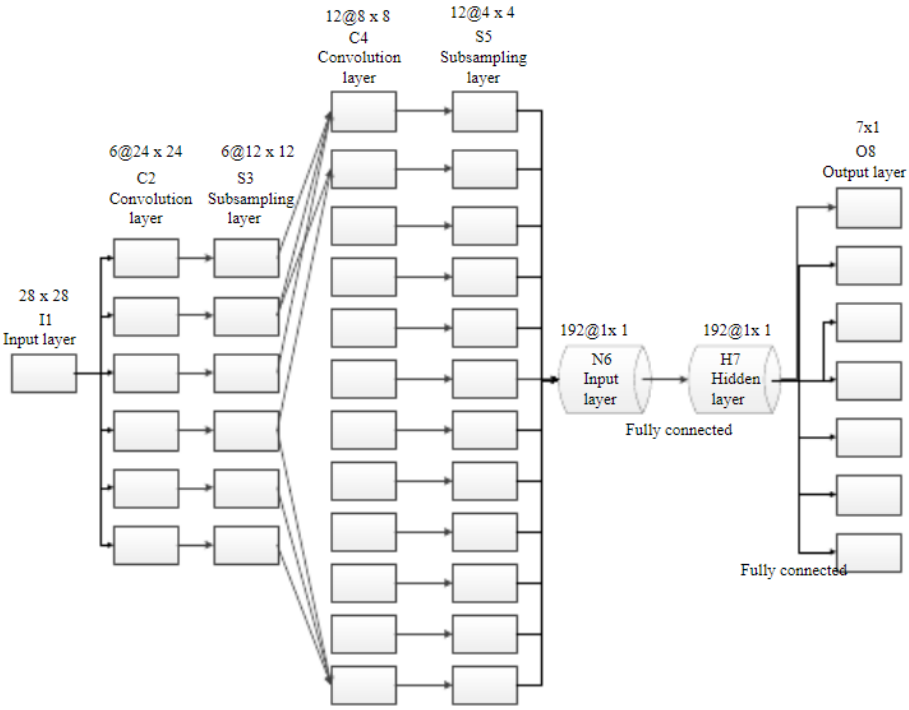


Figure 9: Lin et al.'s CNN control architecture [19].

Nuzzi et al. captured 750 images from 15 different operators and trained the model on 80%. They used a variety of validation statistics including F1-score, selectivity and specificity, and achieved a recognition accuracy of 88.5%. The confusion matrix in Figure 10 is testament to the high gesture recognition accuracy, with a strong leading diagonal. Their model had an inference time of 0.23s, making it suitable for live applications. It also only accepts prediction scores above 90%, increasing model specificity by ignoring classifications with low prediction scores.

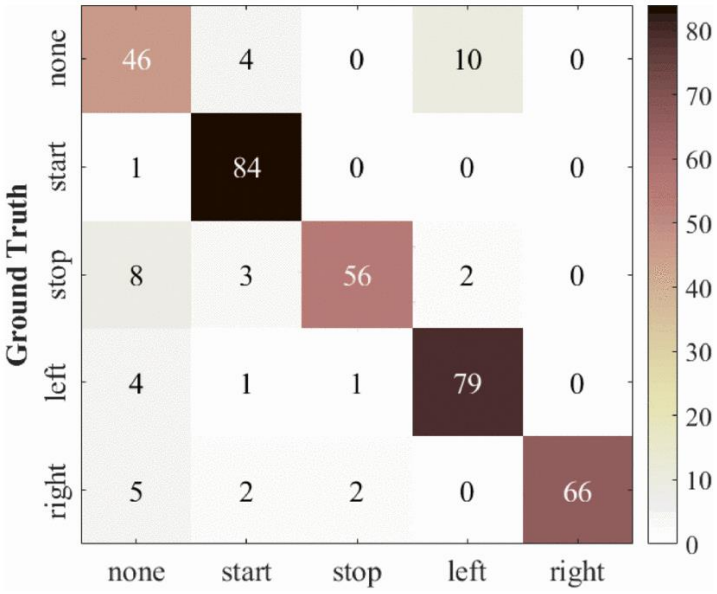


Figure 10: Confusion matrix of the complete test dataset without face detection [18].

Lin et al. trained their model using 600 images of the gestures in Figure 8, before testing on 200 to achieve a static accuracy of 99%. Subsequently, the model was trained and tested using 7 different subjects, achieving a dynamic accuracy of 95.96%. They found that grey-scaling images was far more effective than binarizing since the latter made certain gestures less distinguishable from one another.

Dumoulin et al.'s study also used vision sensors in the form of a computer webcam to record videos of 10 gestures in an industrial context for cobot control [20]. The pixels were grey-scaled and normalised to between 0 and 1 before being passed to a 3-block hybrid model. The first 2 include 3D convolution layers for spatial feature extraction, dense layers for classification, dropout layers for regularisation, max pooling 3D layers for spatial down-sampling and batch normalisation layers. The final block introduces 2 ConvLSTM2D layers (specialised for video analysis by combining convolutions with LSTMs in 2D space), an LSTM layer and an additional dense layer for classification. The LSTM architectures were significant because the model needed to differentiate between gestures moving left-to-right and those moving right-to-left, i.e. the data had time dependence. The model was trained using 1500 videos and tested with more challenging examples. The confusion matrix for their results is presented in Figure 11; the clear leading diagonal indicates that the model correctly classified 8/10 gestures it could detect. The overall accuracy of the model was 84%, with up to 90% for 8/10 gestures. Like Nuzzi et al., they added thresholds to reject prediction scores below 60% to increase model specificity.

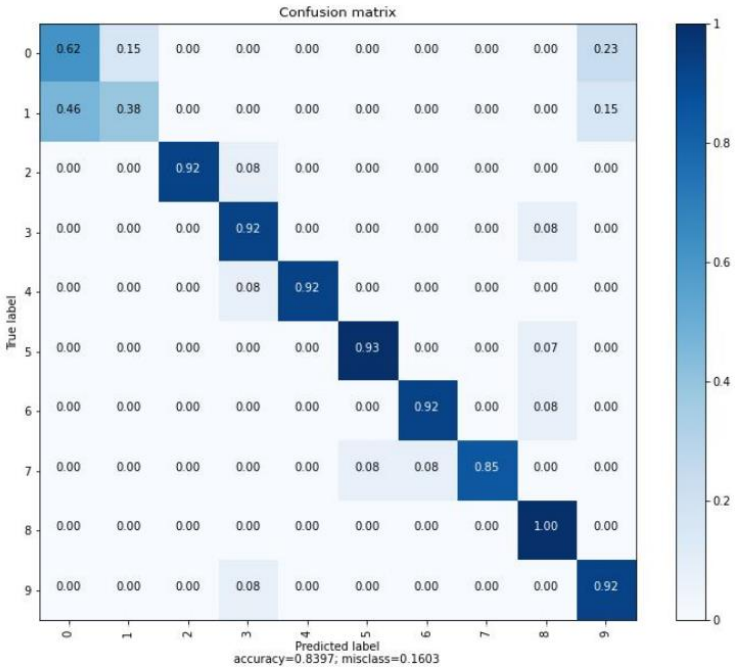


Figure 11: Confusion matrix of CNN-LSTM hybrid model [20].

Yuan et al.'s research shares similarities with that of Male and Hernandez: they too used fusion methods and IMUs to classify hand gestures, although their work focused on classifying gestures from American Sign Language (ASL) and complex Chinese Sign Language (CSL) [21]. While not oriented around assembly, CSL includes gestures used in daily life which parallel the complexity of some assembly actions, hence this gesture recognition model could be transferred to assembly applications. They developed a multimodal glove which incorporated IMUs to provide 3-axis acceleration and gyro data of the hand. However, they also included flex sensors to obtain the bending angle of the fingers. The glove recorded 24 ASL and 72 CSL gestures, which were passed through a Deep Feature Fusion Network (DFFN) model, composed of a Deep CNN (DCNN) and an LSTM (see Figure 12).

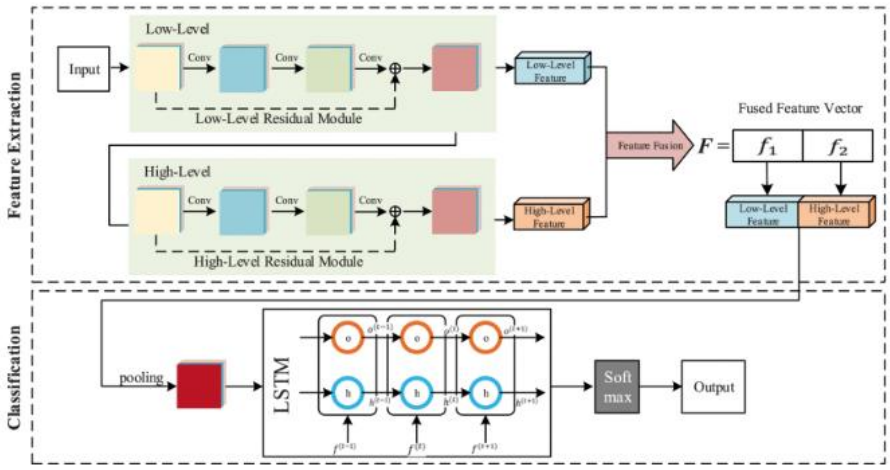


Figure 12: Architecture of DFFN [21].

The model was trained with 70% of data. Their model reached a precision of 99.93% on ASL data and 96.1% on CSL data. The LSTM was significant in classifying more complex gestures.

Other studies classifying sign language used similar sensors: Ajay et al. used an MPU6050 with an Arduino to obtain 6-axis acceleration and gyro data for each gesture in ASL. However, they classified using an SVM; a supervised machine learning algorithm used to classify data by calculating the optimal path or hyperplane (depending on the number of features in the input data), which maximises the distance in an N-dimensional space [22]. This segregates the data by class (see Figure 13). While multiple hyperplanes may exist within a single dataset, SVMs calculate that which best separates the classes. The distance between classes is called the margin, and data points which lie on the margin are called the supporting vectors. SVMs are most used in text and image classification since they perform well with high-dimensional data.

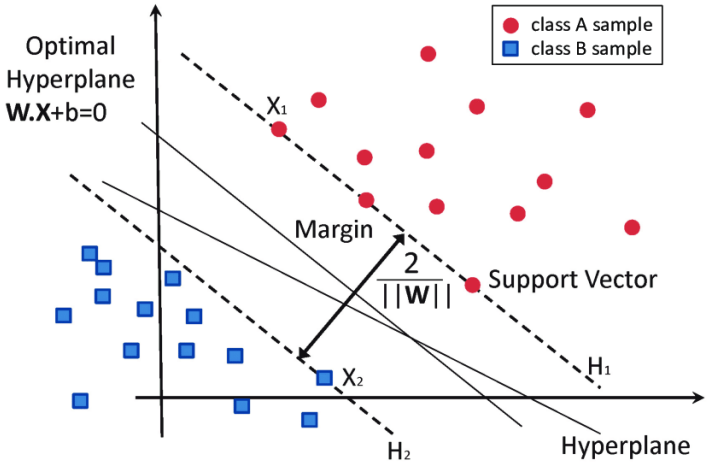


Figure 13: Classification of data by support vector machine (SVM) [23].

Their algorithm used a winner-takes-all strategy; this means that the neurons in a layer compete for activation and the classification is determined by the neuron with the largest value. Their model achieved a recognition accuracy of 98%. The leading diagonal of the confusion matrix in Figure 14 indicates that the model rarely misclassified sign gestures.

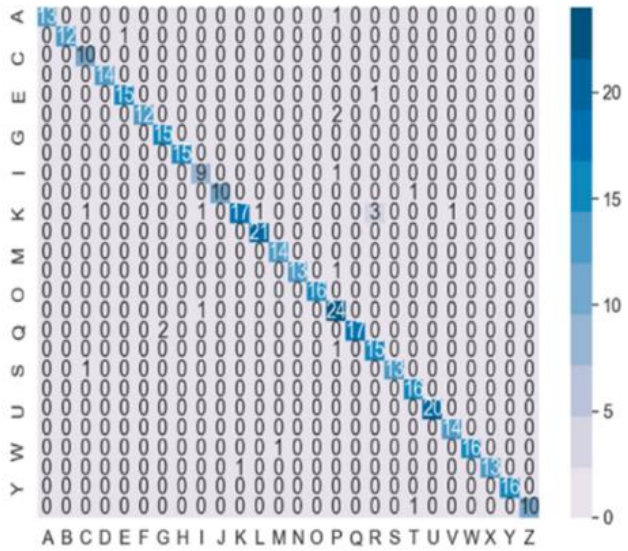


Figure 14: Confusion matrix for SVM model [23].

Mace et al. employed more unique gesture recognition methods: they collected hand acceleration data from a TI eZ430-Chronos Watch, which contains a VTI-CMA3000 3-axis accelerometer, with a measurement range of 2g, 8-bit resolution, and 100Hz sampling rate [24]. This data was classified using two different machine learning algorithms: Naïve Bayesian and Dynamic Time Warping (DTW).

The Naïve Bayesian Classification makes accurate predictions using statistical measures to calculate membership probabilities. Mace et al. used 20 features extracted from their acceleration data, including interquartile range, average energy, maximum of absolute value and standard deviation [24]. The normal distribution determines proximity to the average trained feature measure of each gesture type. Proximity values are multiplied by feature weights assigned during training. The system assigns the gesture label with the highest score based on the sum of resulting values from each feature.

Mace et al. also investigated template matching using DTW, which calculates the similarity between two time-series data sets [24]. To find the time-independent similarity between a gesture and template, the i^{th} point of the gesture can be aligned (or warped) to the j^{th} point of the template [25]. It calculates a matrix where each element represents the distance between data points at different times. Minimal distances suggest a match between gesture and template. DTW finds a path in the matrix where the sum of distances is minimised. This method determines similarity between sample and template datasets. It's repeated for all sample-template pairs, assigning a gesture match to the pair with the smallest sum.

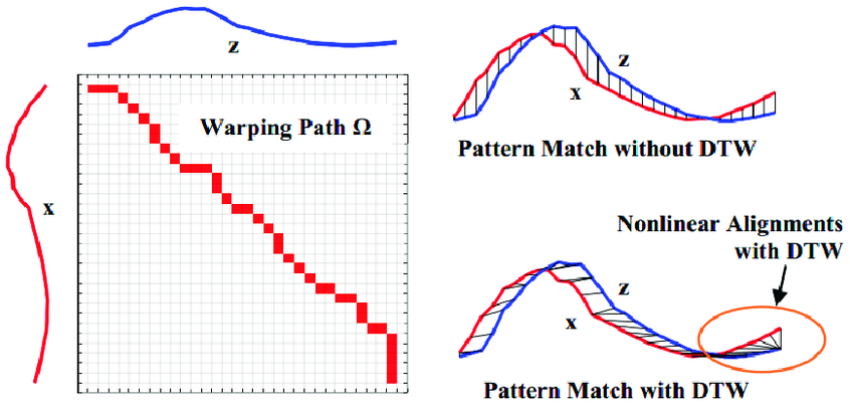


Figure 15: Illustration of Dynamic Time Warping [26].

Both methods were tested using 5 gesture samples of 4 types, including circle, figure of eight, square and star) from 5 different people. The feature separability weighted Bayesian Classifier achieved an average accuracy of 97%, whilst the DTW achieved 95% [24]. Both algorithms have comparable accuracy to the renowned Markov Models and k-mean algorithms and run faster on larger datasets, as well as requiring fewer training samples [24].

One final multimodal glove design to be examined was used to capture and store gesture data on Ninapro [27], a database containing various gesture data from numerous subjects. Dataset 1 contains 52 gestures from 27 subjects, varying from basic movements of fingers and wrists to grasping and functional movements. The data was recorded using surface Electromyographic (sEMG) and kinematic modalities: 10 Otto Bock MyoBock 13E200 electrodes placed along the forearm, and a kinematic Cyberglove containing 22 other sensors (see Figure 16). This produced a total of 32 feature columns of sensor data.

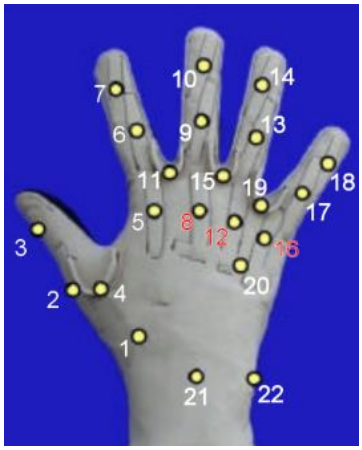


Figure 16: Cyberglove sensor locations [27].

This database does not report machine learning to classify gestures itself but serves as a fine example for recording and storing gesture data, especially in preparation for machine learning. This glove design uses considerably more sensors and was tested on many more subjects than any other study.

4.2 Discussion on Relevant Literature

4.2.1 Sensing Technologies

Sensing methods involved either using cameras to visually detect user gestures, or a multimodal glove to capture data related to arm motion, whether as acceleration and gyro data or using sEMG. A comparison of accuracies and costs are shown in Figures 17 and 18 respectively.

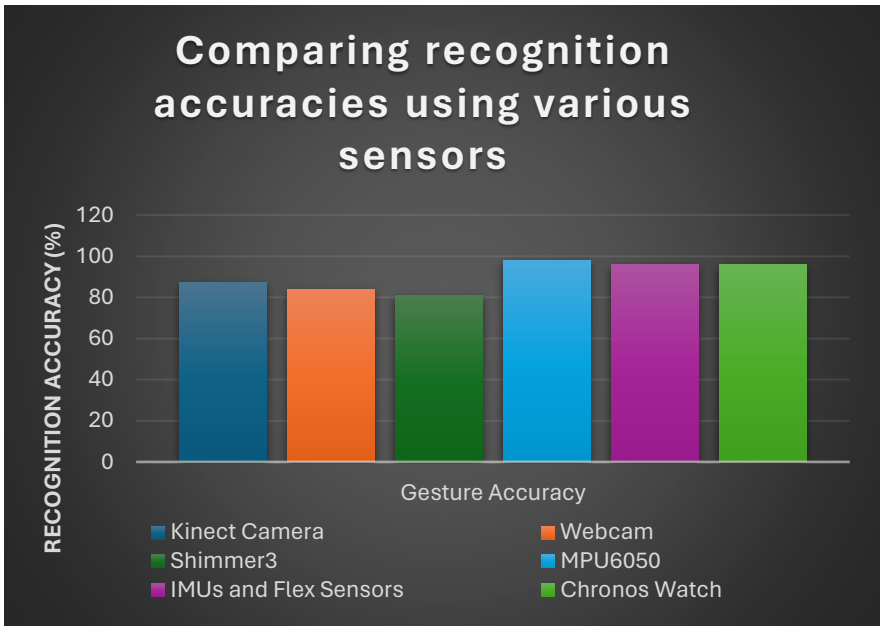


Figure 17: Sensor comparison based on final recognition rates achieved.

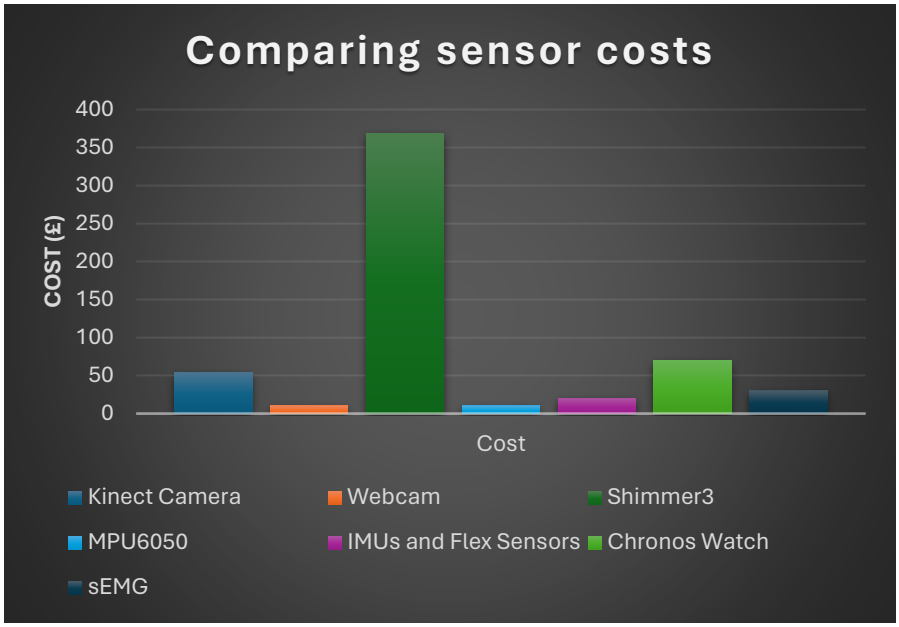


Figure 18: Comparison of average sensor costs from university-approved suppliers.

The Kinect Camera was particularly popular since it also provides depth information; however, it required mounting in a fixed position above the user, where vision was occasionally obstructed mid-gesture. Additionally, the captured images required significant pre-processing such as grey-scaling and pixel normalisation, which is computationally burdening. Video capture uses sequential gesture data rather than just static images, which is more useful for assembly applications since it provides task progression with respect to time. However, this is even more computationally intensive and requires huge amounts of data to process.

The sensing glove proved to be a more portable, accurate solution, providing data which requires less processing. These incorporated IMU sensors or sEMG sensors, which provided 1D data streams as opposed to more complex 2D images or videos. sEMG sensors are more expensive than IMUs and multiple are typically required to obtain sufficient data. Therefore, IMUs are more suitable for developing accurate yet cost-efficient devices. Mace et al. used a discontinued, more expensive IMU device which helped to provide recognition accuracies between 95-97%. Alternatively, Ajay et al. used only 1 MPU6050 and achieved comparable accuracy at a greatly reduced cost. The expensive Shimmer3 sensor did not noticeably improve recognition accuracy, whereas the IMU-flex sensor combination showed potential with its low cost and high-accuracy results.

When considering a cost-effective gesture capture system, IMUs stood out. In particular, the MPU6050 offers a cheap yet effective solution to capturing gesture data. Whilst one sensor is insufficient for classifying complex gestures used in manufacturing, multiple IMUs and flex sensors could be combined to produce a multimodal glove capable of capturing such gestures. This solution could be worn by workers without impeding tasks, and data flow left uninterrupted due to visibility restrictions.

4.2.2 Machine Learning Methods and Model Performance

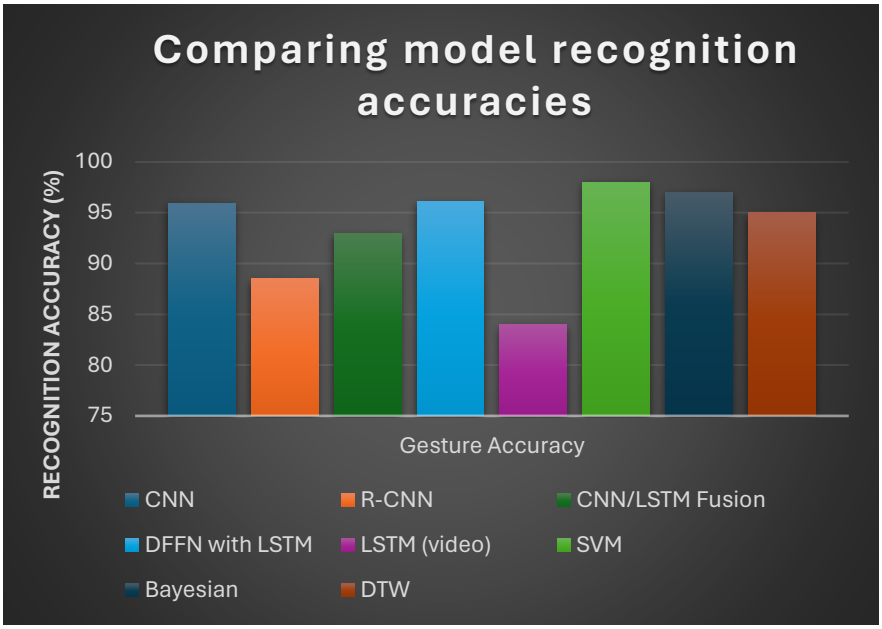


Figure 19: Comparison of model recognition accuracies.

According to Figure 19, the SVM achieved the best recognition accuracy. However, this was only tested with basic sign language images; recognition results may not translate well to assembly gestures which can be complex and include more sequential nuances. The CNN classifiers all achieved good recognition results; CNNs pair well with vision systems by accurately classifying spatial data like images or 2D arrays. However, multimodal gloves with IMUs were deemed better suited to assembly use cases. From the literature, CNNs seemed less accurate when not classifying image data, reducing their applicability.

Models featuring LSTM layers performed well, apart from video classification since this likely required convolutional layers to assist spatial trend recognition. The DFFN and CNN/LSTM fusion models both achieved above 90% accuracy; this is because LSTM layers are well-suited for classifying sequential gesture data. The DFFN model utilised IMU and flex sensor data to recognise gestures of the gloved hand, whereas the CNN/LSTM model required a combination of vision and expensive IMU data. Combining the former's better performance with the preference against vision systems and CNNs, LSTMs with IMUs present the ideal approach. The Bayesian and DTW classifiers performed noticeably well on Mace et al.'s simple gesture shapes; however, Male and Hernandez' results indicated that LSTMs outperformed them in the context of assembly task progression. Additionally, these methods have fewer resources available whereas LSTM model resources are available from TensorFlow [28].

4.2.3 Concluding the Review

In conclusion, this review indicated that developing a multimodal glove using IMU sensors like the affordable MPU6050 (potentially combined with flex sensors) was an appropriate plan in creating a gesture capture system. The most suitable model to classify such gestures was an LSTM sequential model since it displays excellent sequential data classification performance. Potential testing of an SVM model could be worthwhile, considering its excellent performance with the MPU6050. CNNs and vision systems were deemed less suitable for this project due to high costs and computational requirements. More complex fusion models and uncommon Bayesians/DTWs were less favourable due to limited support availability and time constraints of this project.

5 PROJECT AIMS AND OBJECTIVES

After considering the background and literature research, the project scope was adjusted (see Table 1):

Table 1: Project scope considerations adapted from Planning Document [29].

In Scope	Out-of-Scope
▪ Producing a wearable device for humans which can assist robotic coordination with human workers in collaborative tasks. ▪ Implementation of different sensing modalities and machine learning methods for fast and accurate hand gesture recognition processes. ▪ Enabling robots to cooperate effectively with the user in real time. ▪ Consideration given for ergonomics and user experience in the design of the wearable device and overall interaction process. ▪ Using the output from the recognition process to control a robot in a simulated environment to demonstrate collaborative use in assembly tasks.	▪ Testing the collaborative process with a physical robot due to complexity and training requirements. ▪ Capturing movement of the elbow / upper arm / full body. ▪ Physical design of collaborative robots which would cooperate with the glove wielder. ▪ Considerations for mass production – production will be prototype-specific. ▪ Applications outside of assembly. ▪ Use of soft robotics – outside of project budget so more affordable sensors will be mounted onto the device instead. ▪ Programming the robot to accurately mimic each manufacturing task.

The resultant aim of this project was to design a wearable system which will allow humans to interact collaboratively with robots in assembly tasks. This was separated into 5 key objectives:

Objective 1 – Develop a Bench Prototype with 5-Sensor Motion Capture

By 25th March 2024, design, build, and validate a bench prototype featuring 6 sensors for motion capture. Ensure that each sensor produces distinct signals which are observable and recordable.

Objective 2 - Develop the Physical Glove Prototype

By 4th April 2024, design, fabricate and validate a functional, wearable glove prototype incorporating the 6 sensors to capture motion of the wielder's hand, including wrist and fingers. The prototype should be ready for machine learning application.

Objective 3 –Collect Systematic Data for Gesture Recognition

By 8th April 2024, collect systematic data from at least 4 datasets representing different hand gestures to facilitate machine learning methods.

Objective 4 – Achieve Gesture Recognition with Machine Learning

By 22nd April 2024, explore and implement various machine learning methods to achieve a minimum accuracy of 80% in gesture recognition across at least 3 gestures.

Objective 5 – Teleoperate Robot Simulation for Unique Motions

By 30th April 2024, demonstrate teleoperation or collaboration with robot simulation software, showcasing unique motions corresponding to at least 3 different gestures achieved in Objective 4.

6 BACKGROUND THEORY, ANALYSIS TECHNIQUES AND DESIGN TOOLS

6.1 HAND ANATOMY

Figure 20 shows the joints in a human right hand. These are referenced using their acronyms when discussing sensor locations on the glove.

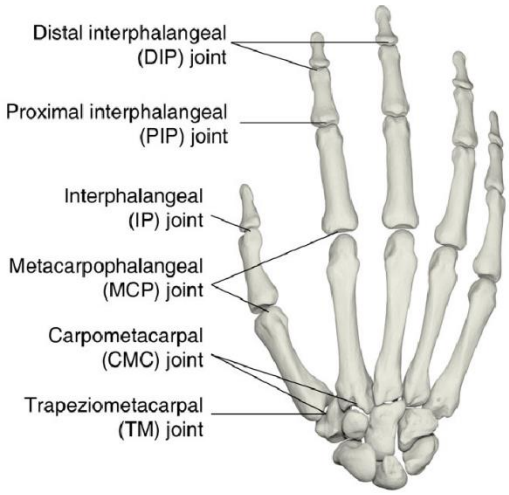


Figure 20: Dorsal (back) view of joints in a human (right) hand [30].

6.2 MACHINE LEARNING THEORY AND ANALYSIS TECHNIQUES

6.2.1 LSTM Structure and Modelling

LSTMs provide short-term memory for Recurrent Neural Networks (RNNs) which can last thousands of timesteps [31]. Their structure includes a cell and 3 gates: input, output, and forget. These gates regulate information flow to and from the cell (see Figure 21).

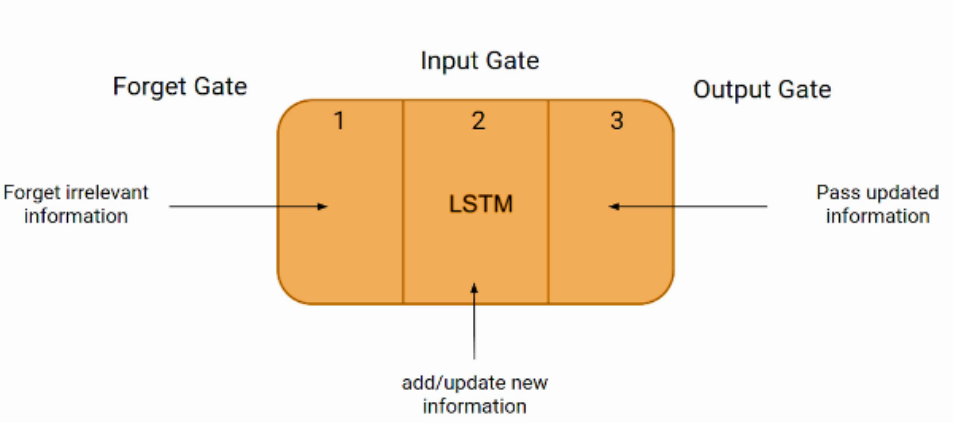


Figure 21: Basic structure of LSTM cell [32].

The gates use the SoftMax function σ, which is the sigmoid function scaled up for multi-class problems (see Figure 22).

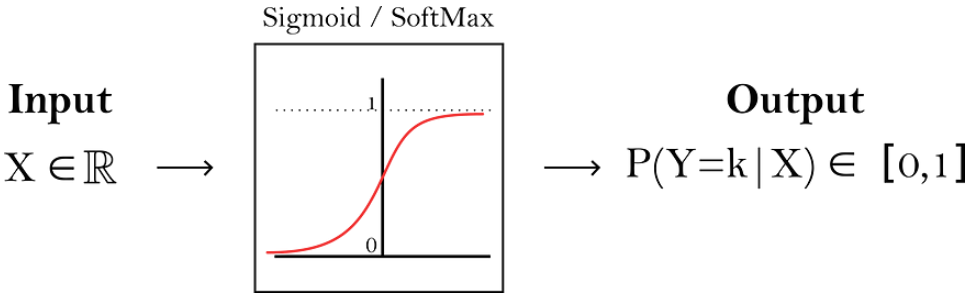


Figure 22: Illustration of the Sigmoid and SoftMax function [32].

They output values from 0 (for blocking) to 1 (everything passes through). Their respective equations are shown in Figure 23:

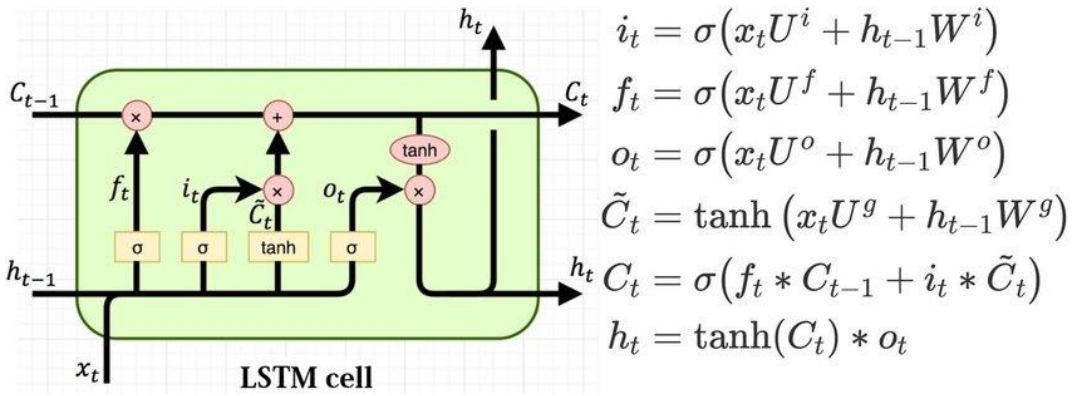


Figure 23: Structure of LSTM cell and equations that describe its gates [33].

In timestep *'t'*: the input gate (i_t) determines what new information will be stored in the cell; the forget gate (f_t) discards certain information from the cell; finally, the output gate (o_t) provides activation to the final output of the LSTM [34].

A candidate cell state is calculated ($\tilde{C}_t$); it forgets data from the previous state and considers inputs from the current timestep to get the updated cell state (C_t) [34].

The updated cell state is filtered before being passed through the output's SoftMax function to get the hidden state vector (h_t). This generates predictions for the current time step by passing through a SoftMax layer to get the final class predictions.

In Keras, these processes are handled internally by its LSTM layer: the 'epochs' hyperparameter specifies the number of times the entire training dataset is passed forward and backward through the neural network during training. Meanwhile, the 'batch size' hyperparameter determines the number of training examples used in each iteration of gradient descent, influencing the model's optimisation process. Final class predictions are generated by passing hidden states through a Dense layer and applying SoftMax activation [28].

6.2.2 Precision and Recall

Generalised metrics used to evaluate the performance of machine learning algorithms are defined by Evidently AI [35] as:

True Positives (TP): Number of correctly identified positive cases.

True Negative (TN): Number of correctly identified negative cases.

False Positives (FP): Number of incorrectly predicted positive cases (or false alarms).

False Negatives (FN): Number of incorrectly predicted negative cases (or missed cases).

Precision measures the proportion of true positive predictions among all positive predictions made by the model (see Equation 1).

$$Precision = \frac{TP}{TP + FP} \quad (1)$$

Recall (or sensitivity) measures the proportion of true positive instances that were correctly identified by the model (see Equation 2). This is more useful in applications where false negatives are costly and undesirable.

$$Recall = \frac{TP}{TP + FN} \quad (2)$$

Combined, precision and recall offer a balanced assessment of the model's effectiveness, especially in scenarios with imbalanced classes or varying costs associated with different types of errors. This helps to provide a comprehensive understanding of a model's strengths and weaknesses and helps to direct model selection and optimisation.

6.2.3 Confusion Matrices

Among the various model validation methods explored in the literature review, the most prevalent were the confusion matrices; these are tables which sum up the performance of a classification model [35]. An example is shown in Figure 24, which depicts a binary classification problem.

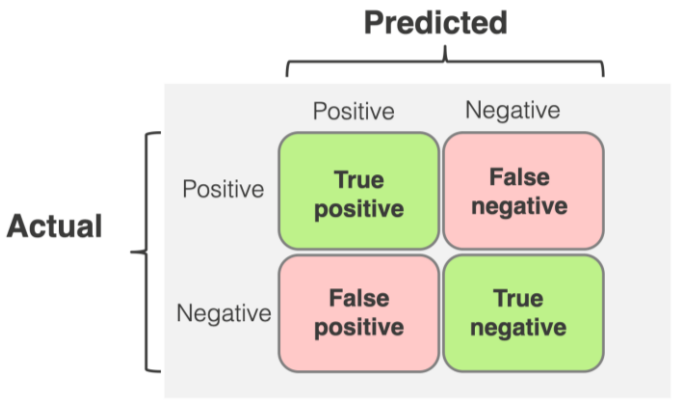


Figure 24: Confusion matrix example [35].

This can be scaled to multi-class classification. For N classes being classified, there exists an N x N confusion matrix. These are useful for visualising the number of correct predictions and errors of each class simultaneously.

6.3 Software Design Tools

6.3.1 The Arduino IDE

The Arduino Integrated Development Environment (IDE) is an ideal choice for sensor software design due to its simplicity, accessibility, and focus on embedded systems development. It offers a user-friendly environment, extensive libraries, and a collaborative community, making it ideal for rapid prototyping and experimentation.

6.3.2 Python

Python has many useful libraries like NumPy, Pandas, and Scikit-Learn which assist data manipulation, analysis, and task modelling. Its intuitive syntax and readability make it easy to write and maintain code. Moreover, Python's flexibility allows for seamless integration with popular deep learning frameworks like TensorFlow, making it a popular choice for machine learning applications.

6.3.3 CoppeliaSim for Robot Motion

Remote Application Programming Interface (API) with Python enables users to program CoppeliaSim robot simulations. Through a Python script, a connection to CoppeliaSim is established; this connection facilitates tasks like robot movement using forward or inverse kinematics, sensor data reading, and object property manipulation. This software was used to validate the application of the final design on a live, cobot system.

7 METHODS AND RESULTS

7.1 HARDWARE AND COMPONENT SELECTION FOR THE MIGLOVE

The following section presents the chosen sensor components of the MiGlove, as well as the selection process for the most ideal microcontroller (MCU).

7.1.1 The MPU6050 IMU Sensor

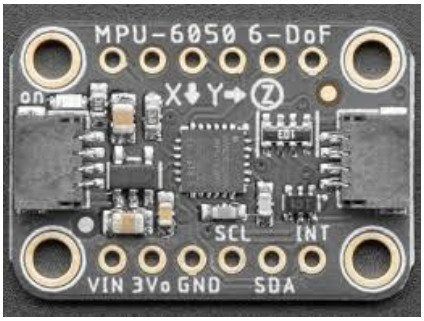


Figure 25: Image of Adafruit MPU6050 IMU sensor [36].

The MPU6050 has long been a popular choice as an IMU sensor, due to its triple-axis accelerometer and gyroscope, with the accelerometer capable of measuring gravity and therefore finding its current orientation. Adafruit's breakout board (shown in Figure 25) combines the sensing capabilities of the MPU6050 with Inter-Integrated Circuit (I2C) communication for quick and easy prototyping. The board operates at 3.3V, but an on-board voltage regulator allows it to accept 5V as well, making it versatile for various MCUs. Finally, it includes Sparkfun's qwiic compatible STEMMA QT connector for the I2C bus, further improving its versatility when prototyping. Adafruit's Arduino drivers enable use of the Arduino IDE, alongside useful examples to assist with familiarisation of the device, receiving data and detecting motion [37].

This sensor was used to obtain acceleration and gyro data of the user's wrist. Additional IMU sensors could have been deployed onto the finger joints, however, these sensors were bulkier and more expensive compared to what was ultimately chosen to gather finger data.

7.1.2 Introducing Flex Sensors

Flex sensors are variable resistors where their resistance is dependent on deflection. One side consists of a polymer ink with conductive particles embedded within it (shown by the dotted face in Figures 26 and 27). As the sensor bends, the conductive particles are forced further apart from each other, which increases its resistance from 30kΩ to 70kΩ [38].

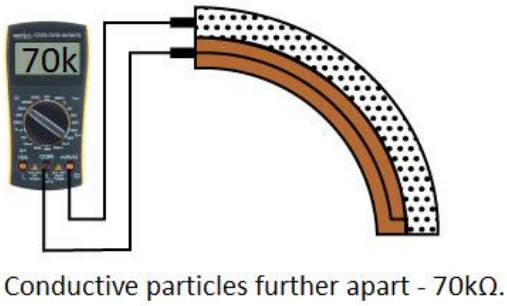


Figure 26: Diagram showing the piezoresistive nature of the flex sensor – with deflection, the resistance increases [1].

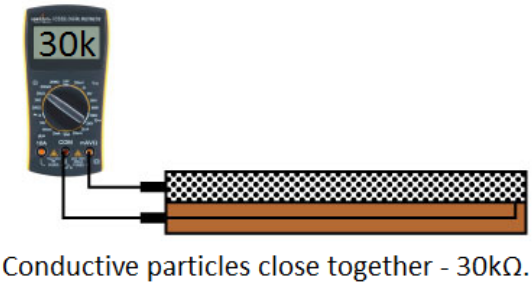


Figure 27: At 0° deflection, the resistance is minimum [1].

Figure 28 shows one of two variations of Sparkfun's flex sensors. This design is 2.2 inches long (2.2", or 0.05588m), whilst the other is 4.5" long. The longer sensors are capable of measuring flexion along the metacarpophalangeal (MCP) joints, as well as the interphalangeal (IP) and proximal interphalangeal (PIP) joints (refer to Figure 20) but are twice as expensive.

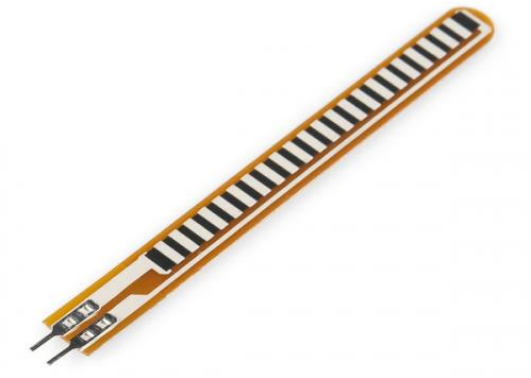


Figure 28: Image of a 2.2" flex sensor [1].

Most bending will be exhibited over the MCP joints, by clenching a fist around a tool. This indicates that the 2.2" sensors would be a cheap yet sufficient solution to monitor the flexion and extension of the fingers during a gesture. These type of flex sensors were also used in the Nintendo Power Glove [38], which serves as a testament to their potential in handheld sensing devices.

7.1.3 The Microcontroller

Once the desired sensors had been chosen, the next task was finding a suitable MCU to manage them. Various considerations are presented in Table 2. Each were scored against relevant weighted criteria in Table 3 and ranked by mean score. The scores are shown in Table 4.

The Arduino Mega 2560 Rev3 (shown in Figure 30) attained the highest weighted score, signifying its appropriateness for this application. Based on the ATmega2560, this board includes 15 digital inputs/outputs (IOs), 16 analogue inputs, 4 Universal Asynchronous Receiver-Transmitters (UARTs) and a Universal Serial Bus (USB) connector. Like the Arduino Uno, it operates at 5V and provides 20mA of current to each IO pin. However, it has 256kB of flash memory, making it capable of running larger programs than the Uno. It uses the user-friendly Arduino IDE, making it a popular choice for easy prototyping. Despite its size, its potential to facilitate and drive multiple sensors on the glove across various communication protocols arguably increases its suitability. MCU familiarity was heavily prioritised to accelerate prototype development. The ESP32-S3-DevKit-1 offers a compact yet versatile board package, and in hindsight would likely be chosen over the Arduino Mega 2560 Rev 3 for revised prototype iterations, provided sufficient time was available to become familiar with the board. Once the fundamental component selection was complete, development of a MiGlove bench prototype could begin.

Table 2: MCU options for the MiGlove.

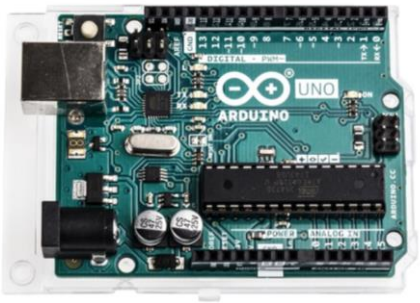


Figure 29: Arduino Uno Rev 3 [39].

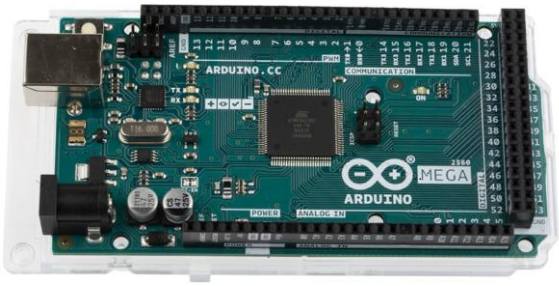


Figure 30: Arduino Mega 2560 Rev 3 board [40].

Figure 31: STM32 Nucleo-F030R8 board [41].

Figure 32: ESP32-S3-DevKit-1 board [42].

Table 3: Selection criteria for MiGlove MCU.

Criterion	Weighing	Reasoning
Peripheral interfaces	0.3	Glove requires sufficient communication interfaces to accommodate multiple sensors and computer access e.g. Serial ports, General-Purpose IO (GPIO) pins, UART, Serial Peripheral Interface (SPI), I2C etc.
Integrated features	0.15	Glove requires standard, built-in features such as Analogue-to-Digital Converters (ADCs), timers and potentially Wi-Fi/Bluetooth/Ethernet communication
Software support and familiarity	0.35	Availability of software libraries, IDE support, and documentation, as well as general familiarity with software and hardware may speed up development.
Size and ergonomics	0.1	Physical size and packaging of the MCU may compromise attachment to glove and user comfort while equipped.
Cost	0.1	Limited budget available for project.

Table 4: MCU scores against criteria.

MCU	Arduino Uno	Arduino Mega	STM32 Nucleo-F030R8	ESP32-S3-DevKit-1
Peripheral interfaces	4	10	8	9
Integrated features	6	9	9	10
Software support and familiarity	10	10	4	6
Size and ergonomics	5	2	3	8
Cost	6	2	10	9
Total weighted score	6.7	8.25	6.45	8.00

7.2 Development of the Bench Prototype

7.2.1 The Wrist Sensor

5V was supplied to the board via the USB connection with the computer running the Arduino IDE. The Arduino baud rate was initially set to 9600 bits per second (bps), primarily because this is a standard baud rate used with many MCU-related devices and can deliver 10 lines of data per second. Adafruit's MPU6050 captured the wrist's acceleration and gyro data and transmitted it to the Arduino via Sparkfun's qwiic compatible STEMMA QT connector for the I2C (Inter-Integrated Circuit) bus. This heavily simplified the sensor's integration. The sensor was powered by the 5V line on the Arduino, utilising its built-in voltage regulator to step the voltage down to 3.3V. Figure 33 shows the wiring diagram for the sensor and the orientation of the IMU axes.

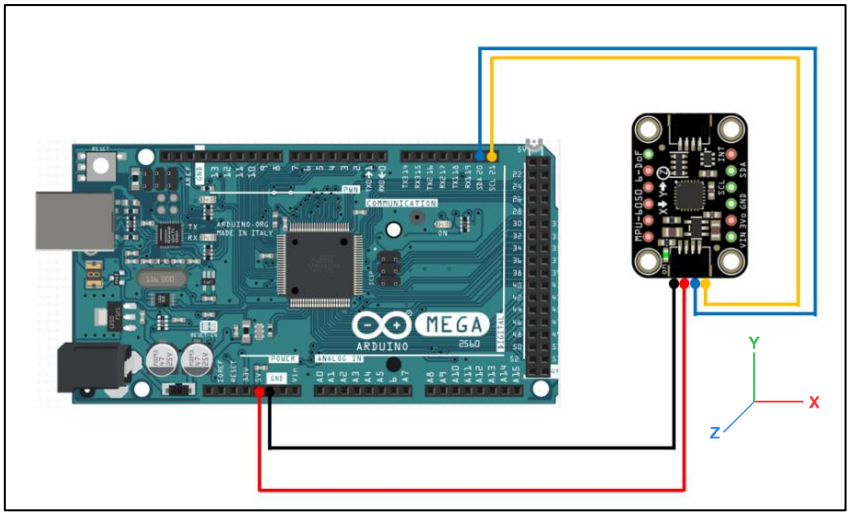


Figure 33: Circuit Diagram of Adafruit MPU6050 connected to Arduino using qwiic (STEMMA) connector.

Adafruit's 'basic_readings' example demonstrated how the sensor could be correctly set up and calibrated, before transmitting continuous 3-axis acceleration and gyro data to the serial monitor. The MPU6050 library includes functions to change the sensitivity of the accelerometer and the gyro, as well as set the filter bandwidth of the internal low-pass filter:

Using Adafruit's example for inspiration, the MPU6050 software was developed in the Arduino IDE. The accelerometer range was set to ±8g, where g is the gravitational constant of 9.81ms^{-2}. Lower ranges of ±2g and ±4g are better suited for subtle movements which must be detected with high precision; gestures such as hammering and sawing can be performed at variable frequencies, where higher frequencies correlate to larger accelerations. In observations, the hammering gesture can reach above ±40ms^{-2}, which would be saturated at these settings. The largest range was ±16g, which is ideal for aerospace and sports performance where incredibly large accelerations are present. However, increasing the range lowers the resolution of the sensor, since the same number of bits is used to represent larger values. Using a wider range can also increase the noise floor of the sensor, making it more difficult to distinguish true acceleration readings from noise. For this purpose, 8g was sufficient to measure faster gesture movements without compromising accuracy.

The same reasoning was adopted when setting the gyro range to 500deg s^{-1}, as opposed to 250deg s^{-1} or over 1000deg s^{-1}. During assembly tasks, the fingers are typically flexed to hold tools; frequent, rapid flexion and extension is not required for such tasks.

Finally, the filter bandwidth was set to 94Hz to reduce noise and aliasing in the sensor readings. Noisy MPU6050 readings would make gesture recognition difficult, particularly for gestures which use similar hand grips, since the flex sensor data would not provide enough data to differentiate between them. However, overly aggressive noise reduction can affect response time, which is undesirable for obtaining precise gesture data. Therefore, medium noise reduction was deemed sufficient.

Acceleration and gyro data across the X, Y and Z axes were printed on the serial monitor. The sensor was moved around and the values on the monitor were observed, confirming that motion data was successfully being captured. A script was written in Python which could open and close the serial port between the computer and the Arduino. This meant that once the basic sensor code was uploaded from the Arduino IDE to the Arduino Mega, the sensor software would run continuously on the Arduino, but only transmit data to the computer while the serial port was open. This script was edited to include manual interrupts; this enabled the user to choose how many data points were returned by the sensor before the interrupt closed the serial port. The serial data from the Arduino would appear in the Python terminal as shown in Figure 34.

```
PS C:\Users\joehi\OneDrive\Documents\University\Year 5\FYP\MPU> & C:/Users/joehi/anaconda3/python.exe "c:/Users/joehi/OneDrive/Documents/University/Year 5/FYP/MPU/MPU sensor code.py"
Arduino says: MPU6050 Found!

AccelX: -7.49, AccelY: 5.4, AccelZ: 2.32, GyroX: -1.0, GyroY:q -0.59, GyroZ: 0.9
AccelX: -6.83, AccelY: 4.16, AccelZ: 2.83, GyroX: -0.7, GyroY:q -1.18, GyroZ: 0.85
AccelX: -6.53, AccelY: 4.15, AccelZ: 4.21, GyroX: 0.05, GyroY:q -0.46, GyroZ: -0.43
AccelX: -8.07, AccelY: 3.21, AccelZ: 1.53, GyroX: 2.99, GyroY:q -1.8, GyroZ: -2.19
AccelX: -5.54, AccelY: 2.95, AccelZ: 4.03, GyroX: 1.75, GyroY:q -1.21, GyroZ: -1.27
AccelX: -5.93, AccelY: 4.36, AccelZ: 6.16, GyroX: 0.8, GyroY:q -0.42, GyroZ: -0.5
AccelX: -5.31, AccelY: 4.13, AccelZ: 5.75, GyroX: 0.12, GyroY:q 0.09, GyroZ: 0.16
AccelX: -4.97, AccelY: 3.74, AccelZ: 5.28, GyroX: -0.03, GyroY:q -0.3, GyroZ: 0.2
AccelX: -4.54, AccelY: 3.65, AccelZ: 5.62, GyroX: -0.12, GyroY:q -0.04, GyroZ: 0.44
AccelX: -4.56, AccelY: 3.58, AccelZ: 5.45, GyroX: -0.17, GyroY:q -0.5, GyroZ: 0.07
AccelX: -4.28, AccelY: 3.88, AccelZ: 6.18, GyroX: -0.56, GyroY:q -0.95, GyroZ: 0.47
AccelX: -3.49, AccelY: 3.76, AccelZ: 6.92, GyroX: -0.14, GyroY:q -0.33, GyroZ: 0.02
AccelX: -3.72, AccelY: 3.66, AccelZ: 6.74, GyroX: -0.13, GyroY:q -0.29, GyroZ: 0.27
AccelX: -2.84, AccelY: 3.44, AccelZ: 7.45, GyroX: -0.23, GyroY:q -0.02, GyroZ: 0.52
AccelX: -3.05, AccelY: 3.57, AccelZ: 6.77, GyroX: 0.01, GyroY:q 0.25, GyroZ: -0.19
AccelX: -2.49, AccelY: 3.57, AccelZ: 7.03, GyroX: 0.02, GyroY:q -0.12, GyroZ: 0.15
AccelX: -2.11, AccelY: 4.4, AccelZ: 7.79, GyroX: -0.14, GyroY:q -1.4, GyroZ: 0.87
Interrupt detected...
Data points captured: 17
```

Figure 34: Serial data from the Arduino shown in the Python terminal.

Within Python, the 6 variables from the MPU6050 were stored into lists, creating a matrix of (N x 6) where N is the number of data rows.

The MPU6050 also supports motion detection, where datapoints are only acquired during active motion. This could minimise the number of pre-gesture or post-gesture idle points which may be mistaken for inactivity by the recognition program. This feature was initially added to the sensor software but was later removed, as explained upon completion of the bench prototype.

7.2.2 The Finger Sensors

Sparkfun's 2.2" flex sensors were built into voltage divider circuits depicted in Figure 35 – the output voltage signal was dependent on the ratio between the variable resistance of the flex sensor and a fixed resistor.

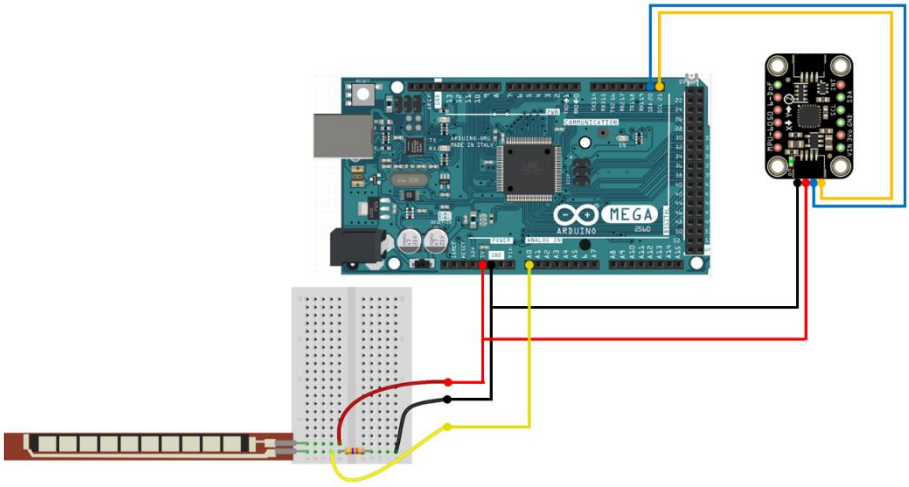


Figure 35: Arduino and MPU6050 circuit with one flex sensor and potential divider added.

The Arduino reads the voltage on an analogue pin with 10-bit precision, assigning a value between 0 and 1023 using its analogue-to-digital (ADC) converter. This maps the voltage to between 0 and 5V. The exact value of deflection can be calculated by calibrating the sensor. Whilst the flex sensor is powered by the 5V line on the Arduino, the actual voltage (V_{CC}) can be measured directly using a voltmeter and is about 4.98V.

The voltage reading over the analogue pin (V_{out}) is:

$$V_{out} = V_{CC} \times \frac{Pin\ value}{1023} \qquad (3)$$

The resistance across the flex sensor (R_{flex}) is derived from the voltage divider equation:

$$V_{out} = V_{CC} \times \frac{R_{res}}{R_{flex} + R_{res}} \quad (4)$$

This becomes:

$$R_{flex} = R_{res} \times (\frac{V_{CC}}{V_{out}} - 1) \quad (5)$$

According to the datasheet, the flex sensor's resistance ranges from 25kΩ ±30% when flat, to a minimum of 2 times greater at 180° deflection. The sensitivity of the voltage divider is higher when using larger static resistors because a larger change in output voltage can be produced for the same change in sensor resistance. Higher resistances also reduce current drain in the circuit, which is ideal for reducing power consumption. Conversely, since Arduino pins have very high impedances of about 100MΩ [43], the divider should have a much lower resistance in comparison; this ensures that the voltage division is not significantly affected by the input impedance – larger resistors increase the load on the sensor, which can hinder sensor performance and accuracy. As a result, the recommended static resistance range for Arduino inputs is between 10kΩ-100kΩ [38]. To achieve a good compromise between flexing sensitivity and voltage reading stability and accuracy, the static resistor value (R_{res}) was set to 47kΩ. This should provide a suitable voltage range at the output of the potential divider, allowing the Arduino to measure the flex sensor's varying resistance effectively.

Within Arduino, a mapping function could convert the flex resistance into a bending angle, using two calibration parameters: flat resistance and bend resistance (resistance displayed at 0° and 90° deflection respectively). These values could be tuned heuristically to provide accurate angle readings. However, in data collection for machine learning preparations, analogue pin and voltage values were deemed sufficient, so this function was ultimately unrequired.

The solder tabs and pins were removed due to poor setting within the breadboard. Instead, 2-pin (male) Amphenol FCI clincher connectors were added (see Figure 36). These connectors pierce the conducting material in each sensor with metal staples, providing a robust interface to attach the sensors to breadboards or MCUs. Female-to-male jumper cables were used to connect the pin attachments to the potential divider circuits mounted on a breadboard (see Figure 37).

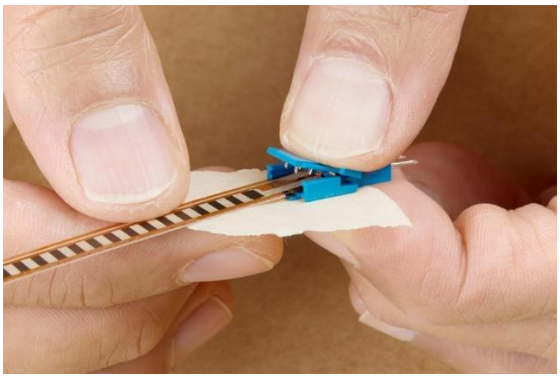


Figure 36: Applying the Amphenol FCI clincher connector to the flex sensor [38].

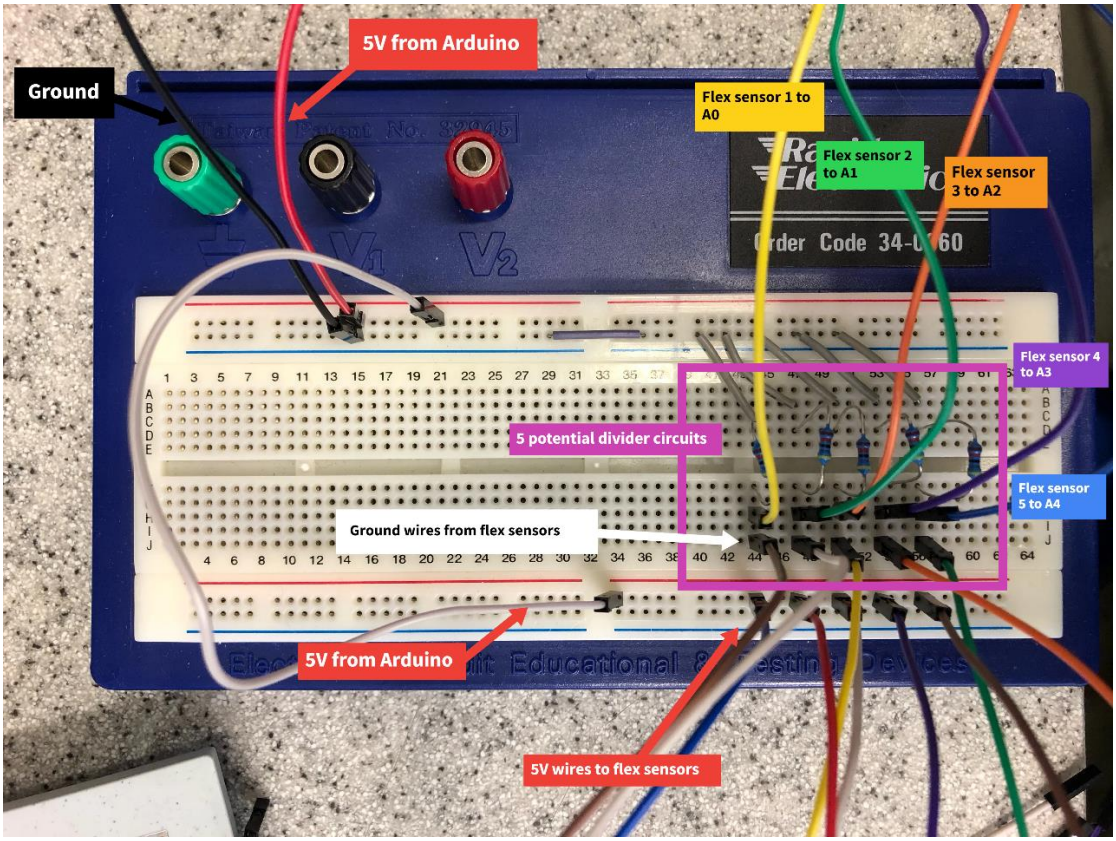


Figure 37: Annotated breadboard circuit with potential divider circuits for flex sensors.

The 5V and ground cables of the flex sensors can be seen in the bottom-right of Figure 37. The ground cables feed into the potential divider circuits, and the voltages calculated in Equation 4 are delivered to various analogue pins on the Arduino Mega.

Sparkfun's example sketch was adapted into a function called 'flex_read', which takes a variable denoting a specific flex sensor from 1 to 5, reads the output voltage across its respective sensor pin (conversion shown in Equation 3) and prints it to the serial monitor. This is repeated in a loop for all 5 flex sensors. The use of this function allows the system to be scaled and modified to include any number of sensor circuits. Each sensor's flexing sensitivity was verified by bending the sensor and observing the voltage variations in the serial monitor. This successful integration motivated progression onto the final build of the bench prototype.

7.2.3 The Completed Bench Prototype

The flex sensors' Arduino script was combined with the existing script for the MPU6050, allowing all sensor data to be acquired simultaneously. 5 additional data lists were added to the Python script controlling the data transmission - one for each flex sensor combined with the existing 6 from the MPU6050. This resulted in an (N x 11) data matrix, where N is the number of data rows and 11 is the number of different sensors collecting data.

Upon adding the flex sensors to the existing circuitry, the motion detection interrupts for the MPU6050 were removed since they would interfere with data receival from the flex sensors. Realistically, some gestures may involve periods of motionlessness, which the interrupt script would not detect. Instead, the Arduino continuously receives data from the MPU6050 and the flex sensors at a rate dependent on the baud rate. At 9600bps, this was about 6.6 lines of data per second.

The flowchart in Figure 38 accurately depicts the Arduino's operation of the sensors. (For program and flowchart source codes, go to Appendix A).

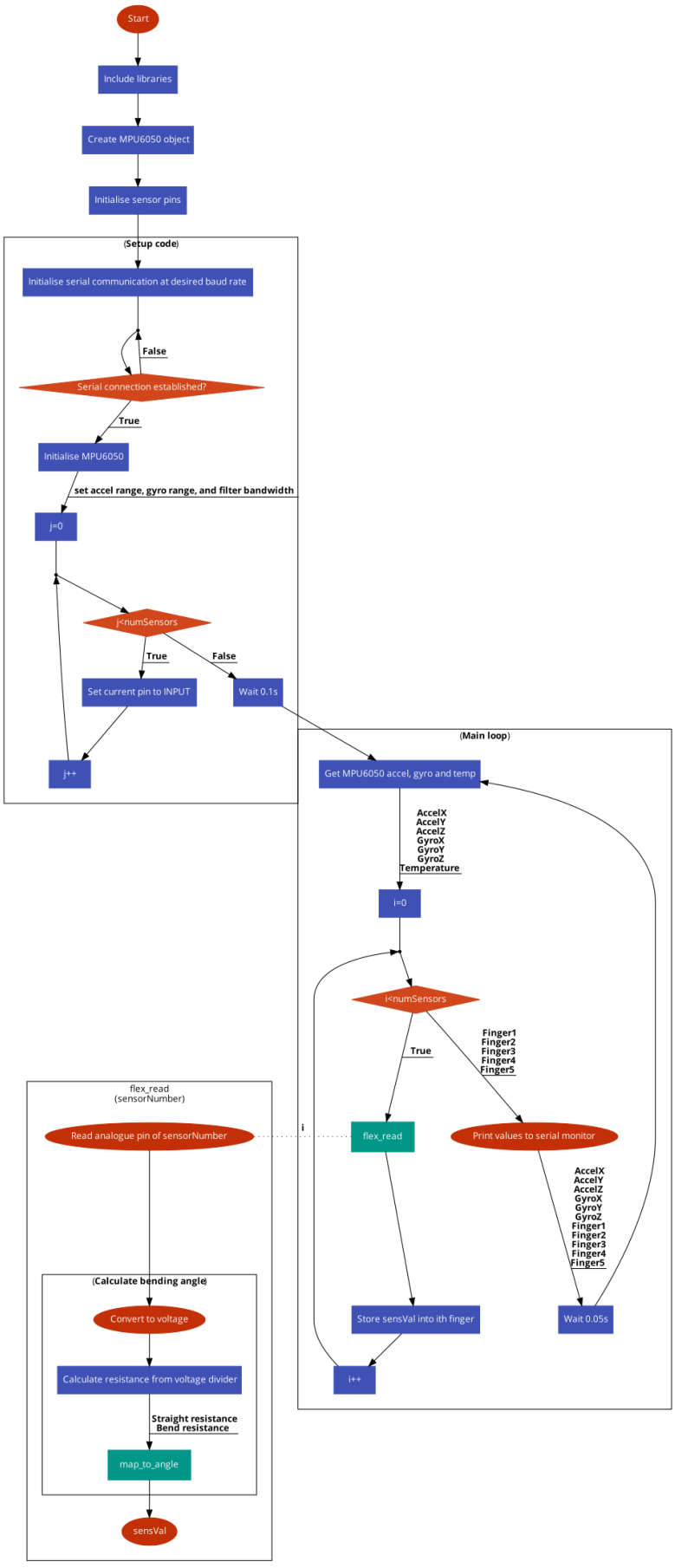


Figure 38: Flowchart for Arduino sensor software (created using code2flow).

7.3 BUILDING THE GLOVE PROTOTYPE

7.3.1 Glove Hardware

Once full integration of the bench prototype was complete, the design was transferred to the physical glove. Polyamide assembly gloves were deemed suitable for the first prototype since these are commonly used in the automotive industry and designed to be ideal for manufacturing and fine product assembly work [44]. The right-handed glove was picked due to user preference, although the same design is transferable to the left hand.

For this prototype, the breadboard in Figure 37 provided insufficient durability, and was replaced with more robust Veroboard as a permanent solution. This prevented wires from becoming disconnected mid-gesture. Additionally, Veroboard facilitates greater customisation, allowing for the compact placement of potential divider circuits by selectively scoring through rows of copper contacts with a cutting tool to separate them. This enables more space-efficient utilisation compared to breadboards. The Veroboard housed the potential divider circuits for the flex sensors, as well as the voltage and ground rails connected to the Arduino. These rails powered the 5 flex sensors and the MPU6050.

A design concept for the glove is presented in Figure 39. The MPU6050's position was switched from the wrist to the central dorsal side of the glove. This was because, although the Veroboard offered superior space efficiency compared to the breadboard, it lacked sufficient clearance with the flex sensors when positioned in the glove's centre. Consequently, it was relocated to the wrist, exchanging positions with the MPU6050 sensor. The performance of the MPU6050 remained unchanged.

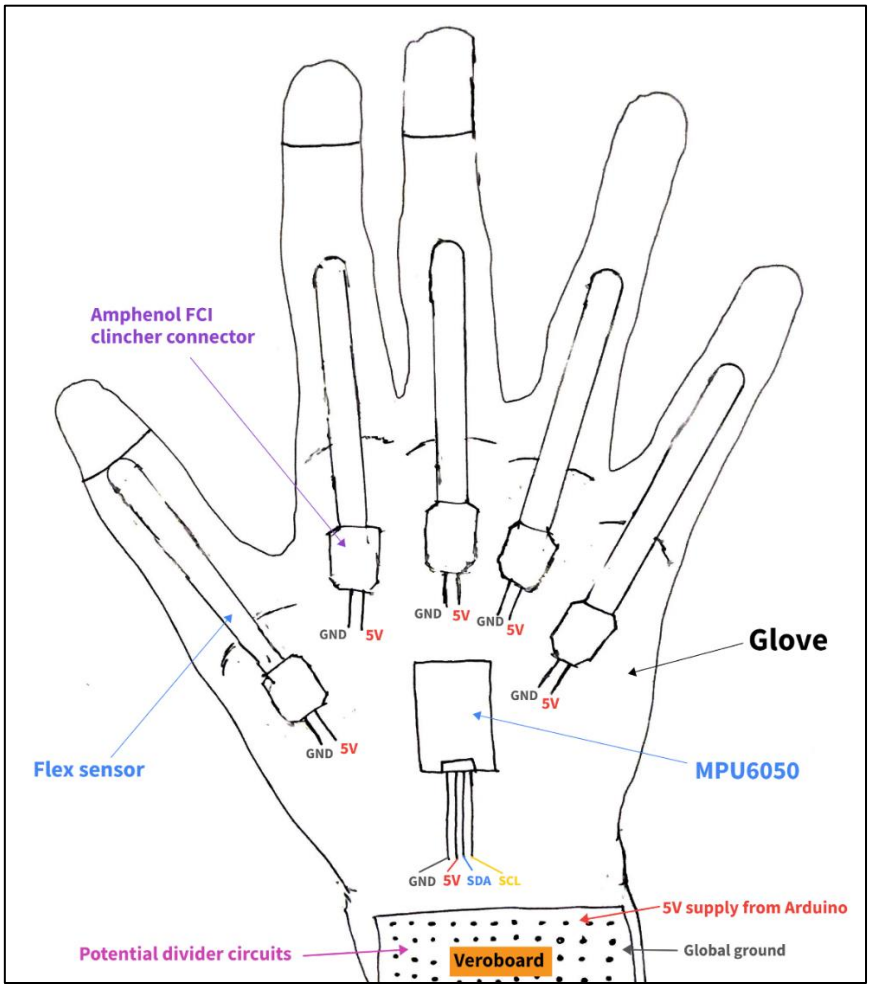


Figure 39: Design concept for hardware placement on glove.

The sensors and Veroboard were attached to the glove using adhesive Velcro hook-and-loop tape. The base of each flex sensor was carefully aligned with its corresponding MCP joint to maximise flexion detection. A wrist strap was attached to the Arduino using adhesive Velcro so that it could be fitted to the forearm. Because of the considerable distance between the Arduino and the flex sensors, the analogue pins assigned to the flex sensors were adjusted to A11 through A15, as these were closer to the wrist. The finished product is shown in Figure 40.

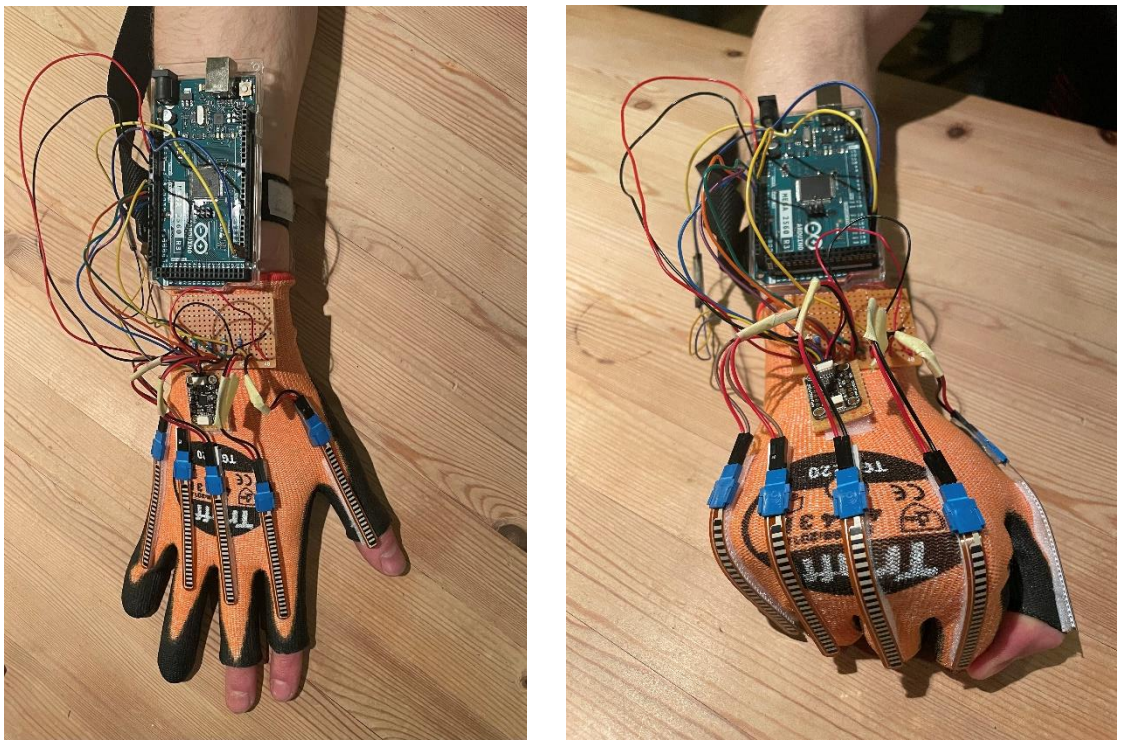


Figure 40: Final glove prototype with flex sensors at zero deflection (left) and maximum deflection (right).

7.3.2 Glove Software

The Python script used to communicate with the Arduino was improved to provide user-friendly gesture recording and storage. A separate file was created for key functions of the MiGlove. These functions include blocks of code which are repeatedly called in the main loop:

1. **clear_input_buffer():** This function clears the input buffer to handle controlled user interrupts. It detects keyboard inputs and flushes the input buffer accordingly. It uses different methods based on the operating system, utilising the 'msvcrt' module for Windows systems and the 'termios' module for Linux/Unix-based systems. This was to accommodate future use of a Raspberry Pi.

2. **connect_to_arduino(arduino):** This function establishes a serial connection with the Arduino device. It first closes any existing connection if 'arduino' is not 'None', waits for a brief period, and then initialises a new serial connection with the Arduino using the specified communication (COM) port (COM3 in this case) and baud rate (115200). It sends a greeting message to the Arduino, reads the response, prints it to the console, and returns the 'arduino' object.

3. **clear_arduino_input_buffer(arduino):** This function clears any data accumulated in the input buffer of the Arduino device. It continuously reads and discards data from the input buffer until it is empty, ensuring that any residual data from previous operations is removed before starting a new data recording session.

4. **wait_for_input(arduino):** This function waits for user input to start recording gesture data. It prompts the user to press the *'Enter'* key to initiate data recording, providing them with full control and time to prepare for the next gesture recording. Upon receiving the input, it prints a message indicating that data recording has started and then calls **clear_arduino_input_buffer(arduino)** to clear the Arduino's input buffer before data acquisition begins.

The complete flowchart in Figure 41 illustrates the glove software and key functions. (For program and flowchart source codes, go to Appendix A).

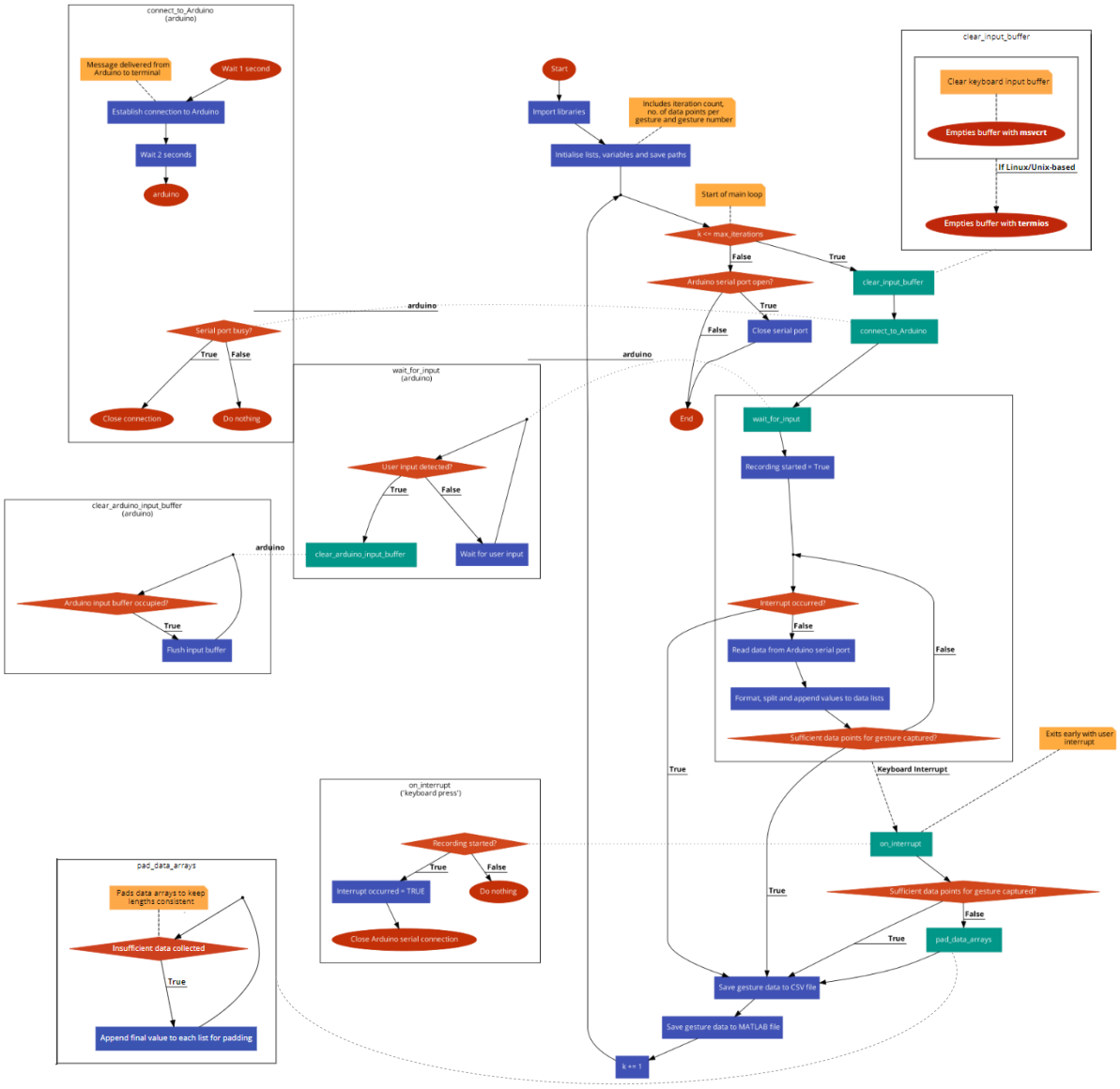


Figure 41: Complete flowchart of MiGlove Software for recording gesture datasets (created using code2flow).

Upon execution, the script initialises several variables, including empty lists to store sensor data and parameters specifying the number of iterations and data points per iteration.

At the beginning of each loop, the input buffer on the computer-side is emptied to flush accumulated input data which has been buffered by the input stream from the keyboard. This ensures that only intentional keyboard inputs are recognised as interrupts, maintaining the integrity of the recording process.

Subsequently, the script closes any existing serial connection to the Arduino device and waits for a user input. This prevents unwanted sensor data being transmitted to the program. Upon user input, the serial connection is re-established, the Arduino's input buffer is emptied to flush accumulated sensor data, and recording is initiated. This ensures that only data following the user input is transmitted to the Python script, isolating the relevant gesture data, and preventing any previously accumulated sensor data from being included in the recording. As a result, the accuracy and integrity of the captured data are maintained.

Once recording begins, the script continuously reads data from the Arduino's serial output and parses it to extract sensor values. These values, representing accelerometer, gyroscope, and flex sensor readings, are then appended to their respective lists for further processing. This is repeated until the required number of data points per iteration is satisfied. Additionally, a data padding function ensures that this is also satisfied following a user input, as this will immediately exit the loop early; the function retains the last value of each list to ensure uniformity in the dataset.

To ensure the robustness of the data acquisition process, the script incorporates exception handling mechanisms. It gracefully handles user inputs to terminate data recording; this means that one incorrect gesture recording does not compromise the entire dataset and can instead be removed in post-processing.

The script saves the captured sensor data to both Comma-Separated Values (CSV) and MATLAB files before iterating to the next loop, facilitating flexible analysis and visualisation. The main loop iterates by a preset iteration number before ending. The gesture number represents the chosen gesture label, which can be manually changed when recording different gesture datasets. Once the final iteration is complete, the script closes the serial connection to the Arduino for the final time.

Thus, the wearable prototype was complete, and the glove could be used to generate gesture datasets to train a machine learning model.

7.4 Collecting and Storing Gesture Data

7.4.1 Selecting Gestures to Test

Ninapro was sourced to obtain 4 hand gestures used in assembly context [27]. These included: turning a screwdriver, small diameter grasp (for hand tools), and resting. Two of the most common hand tools used in manufacturing are saws and hammers [45]; it was important to be able to differentiate between sawing and hammering given their similar grips. The 'resting' gesture represented periods where no tool was in use, i.e. no gesture. The collection of gestures are shown in Table 5.

Table 5: Collection of gestures captured with the MiGlove.

Figure 42: User with MiGlove performing hammering.

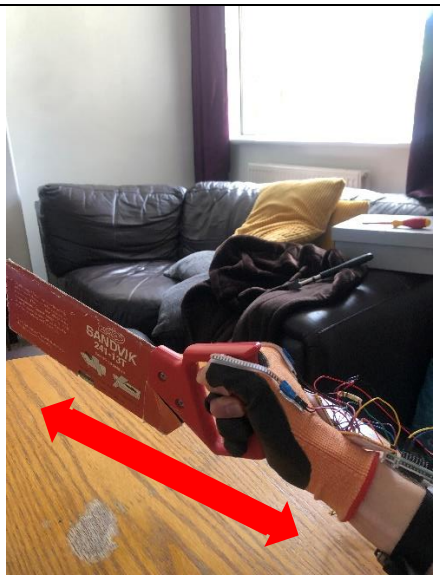


Figure 43: User with MiGlove performing sawing.

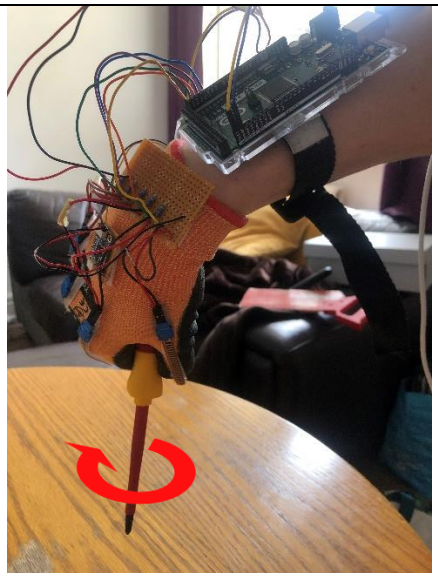


Figure 44: User with MiGlove performing screwdriving.

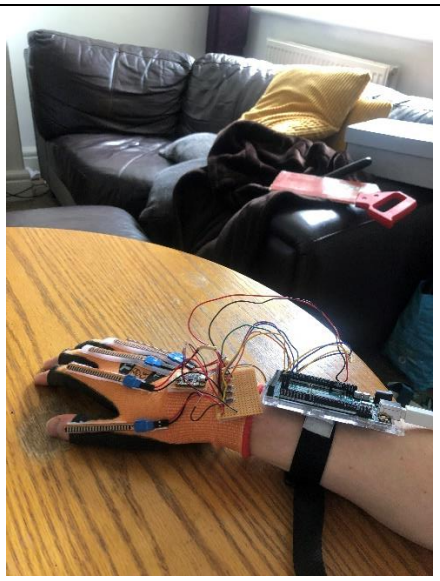


Figure 45: User with MiGlove performing no gesture.

7.4.2 Creating and Formatting the Datasets

Initially, the 4 different gestures were recorded into separate CSV files using the MiGlove. The size of the CSV file was determined by user-defined parameters. (MATLAB formatting was also included to accommodate different methodologies).

The number of data points per iteration was set to 50, so that one iteration included 50 datapoints (50 rows) of the 11 sensor features and their individual indices (totalling to 12 columns). This facilitated the recording of sufficient data to accurately represent each gesture.

The iteration number was set to 30, meaning that one gesture dataset contained 30 iterations of the given gesture; this was added as an additional 13^{th} column to the dataset. 30 iterations should effectively capture the range of nuances present in each gesture whilst reducing overfitting, allowing the model to better generalise patterns rather than memorise repeated examples in the training data.

The gesture number was set individually according to the unique identifiers (UIDs) of the desired gesture, as shown below:

Hammering	Sawing	Screwdriving	No gesture
1	2	3	4

These would form the gesture label outputs used to train and test the machine learning model and were assigned to each data point. This increased the total number of columns to 14.

In summary, one gesture CSV file compiled 30 iterations, resulting in 1500 rows by 14 columns. Once each of the 4 gesture datasets had been successfully recorded, they were combined into 1 CSV file containing all 120 gesture recordings.

7.5 ACHIEVING GESTURE RECOGNITION

7.5.1 Creating, Training and Testing the LSTM Model with Static Data

The LSTM model was constructed within a Sequential framework, recommended for organising a stack of layers where each layer has one input tensor (gesture data) and one output tensor (gesture prediction). The first layer added to the model was an LSTM layer. This was configured with a specified number of units, determining the dimensionality of the output space, and an input shape defined by the number of time steps in the sequence and the number of features at each time step. Next, a dense layer was added to the model, consisting of a fully connected neural network with units corresponding to the number of gestures to be classified. The 'SoftMax' activation function was applied to the dense layer, allowing the model to output probability distributions over the classes in multi-class classification problems. Additionally, callbacks such as 'ReduceLROnPlateau' and 'EarlyStopping' were incorporated to dynamically adjust the learning rate and terminate training if the validation loss failed to improve, respectively. Finally, the model was compiled with the efficient Adam optimiser and sparse categorical cross-entropy loss function [46], with accuracy as the evaluation metric for monitoring training progress.

This model needed input data to adhere to a specific format. The required input shape was 1 x N x M (or one gesture set consisting of N timesteps and M features). The number of timesteps was determined by the length of each gesture recording (in this case 50), while the feature number was set to the number of columns in the dataset which would be used to train the model (initially 13).

The complete CSV file of 120 gestures needed reshaping to be compatible with the model; hence it was converted into a 3D data tensor with dimensions 120 x 50 x 14. This represented 120 gesture tensors to be used as inputs in the LSTM model (denoted as 'x'). Their respective gesture classifications were separated and stored as 'y_labels'. The shape of 'x' was 120 x 50 x 13, and the shape of 'y_labels' was 120 x 50 x 1.

The 3D tensor of 120 gestures was split into training and testing sets using the **train_test_split()** function from Scikit-Learn [47]. The training to testing ratio was set to 9:1 to maximise the model's ability to generalise to unseen data and accurately evaluate its performance on new data. Whilst this does reduce the amount of static testing data available, the model would ultimately be tested on live, unseen data, rendering the testing set useless beyond the training phase. Exposing the model to as much variety as possible was prioritised. The static testing set still included 12 gestures, which was deemed sufficient to evaluate the model's performance.

Hyperparameters were picked arbitrarily - the number of epochs and batch size were set to 1000 and 32 respectively. Normally, these would be tuned heuristically to optimise the model. Moreover, a larger number of epochs risks overfitting the model to the dataset, affecting its ability to generalise with unseen data; however, the model managed to classify 100% of unseen gestures accurately, giving a test loss of about 0.0511. Given the remarkable performance from the outset, the necessity for hyperparameter

tuning was removed and development proceeded to the next step. Optimising the model for speed was less important for this proof-of-concept.

Initially, the model was tested without normalised data. However, using normalised data helps to stabilise and accelerate the training process, particularly for models like LSTMs, which are sensitive to the scale of the input features [48]. It prevents larger features from dominating the learning process simply due to their scale, which could trap the model in local minima. Subsequently, data was normalised using 'MinMaxScaler' from Scikit-Learn [47]. This scaled each feature in the input data to between 0 and 1 before training the model. It works by subtracting the minimum value of a feature from all data points and then dividing by the range (the difference between the maximum and minimum values). This ensures that the minimum value becomes 0 and the maximum value becomes 1, with other values scaled proportionally in between. The test results using normalised data showed 100% accuracy with a notable reduction of fivefold in test loss, measured at 0.0110.

It was noted that the index and iteration columns were being passed through the LSTM unnecessarily, since these values did not provide any meaningful information about each gesture. Thus, these columns were discarded, reducing the feature dimensionality from 13 to 11. The resulting 3D tensor was of shape 120 x 50 x 11. Retraining the model yielded an accuracy of 100% but with a loss of 0.0266. Interestingly, this was slightly higher than before. This could be attributed to the fact that the model previously relied on the index and iteration values to predict the gesture label, especially considering that neighbouring iterations belonged to the same gesture class due to batch recording. Removing these features meant the model needed to identify hidden nuances between different gestures to classify them, leading to higher computational burden and a slight increase in test loss.

However, after debugging, a small error was found to be affecting the data normalisation. Upon fixing this error, the model was retrained once more, giving a test accuracy of 100% and a test loss of 0.000138. This successful result indicated that the model was well-prepared to attempt classifying live, unseen gesture data.

The software program written to achieve this is visualised in the flowchart in Figure 46. (For program and flowchart source codes, go to Appendix A).

Ideally, the SVM model would have been tested and compared alongside the LSTM model. However, this project also involved electronics and prototype design, testing, and simulation to present a complete and thorough solution to the problem within strict time constraints, rather than only focusing on the optimisation of the machine learning model. Outcomes from the literature review strongly indicated that the LSTM was the most suitable for classifying sequential data in gestures, whereas the feasibility of using SVMs on large sets of time-variant data was unclear. Additionally, the LSTM's performance on static gesture data was excellent, hence testing with an SVM model was unnecessary to progress.

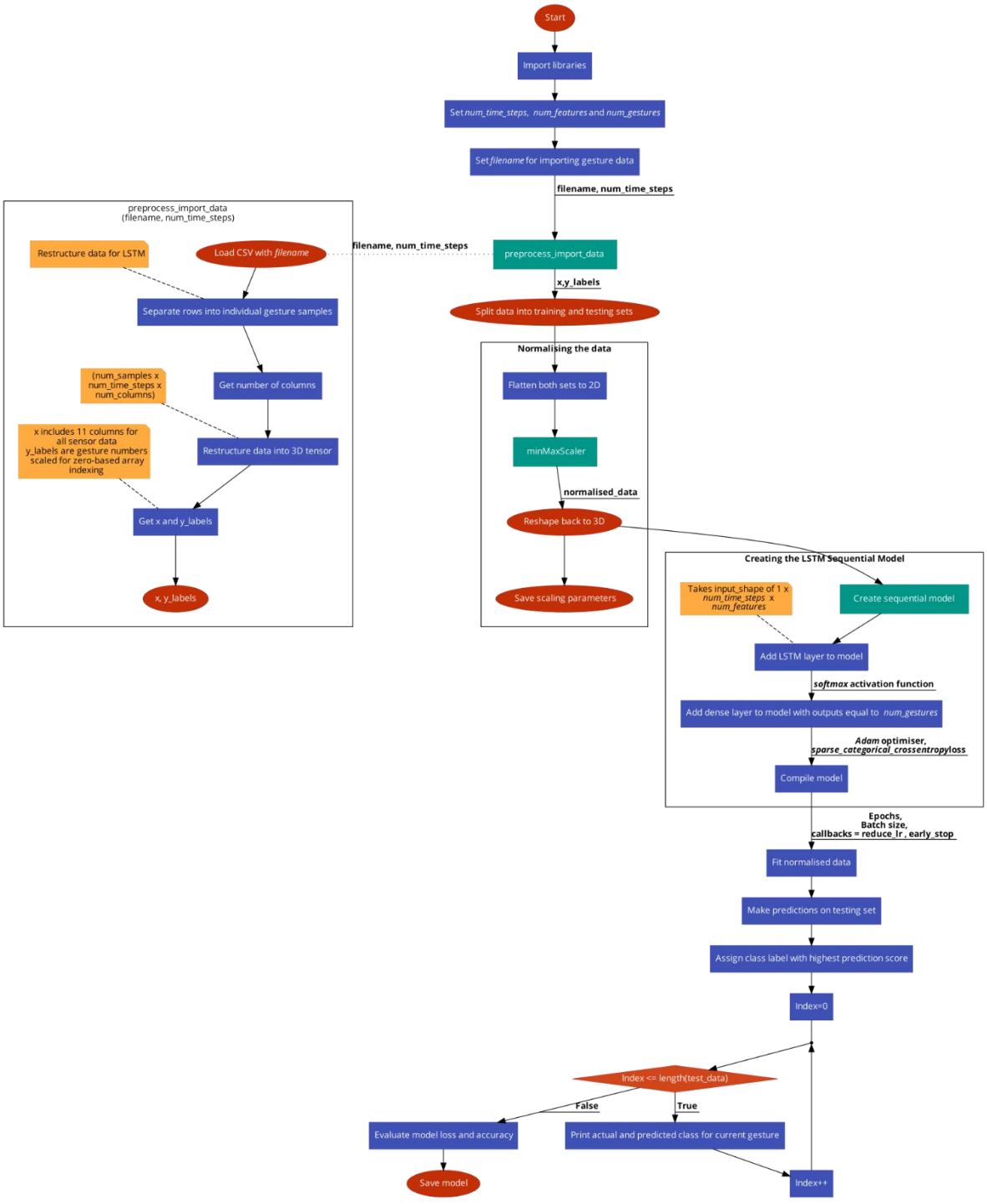


Figure 46: Flowchart depicting the generation, training, and testing of the LSTM model (created using code2flow).

7.5.2 Dynamic (Live) Gesture Recognition with the MiGlove

The workflow of dynamic (live) gesture recognition is detailed below:

Firstly, the trained model is imported, having been trained with 90% of the static dataset. Next, a 10-second recording of live gesture data is acquired using the MiGlove, mirroring the process for obtaining training data. Then, this is formatted into a 2D array, where the data is normalised using the scaling parameters from the training script. The live data is split into windows of 50 data points (equal to the number of timesteps in one gesture). Each window is reshaped into a 3D tensor before being passed

individually through the sequential model. Gesture predictions are calculated, and the window is assigned a gesture label based on the class with the highest prediction score. This is repeated for every possible window in the input data. At 9600bps, one recording contains 66 data points, or 17 windows of data. Each window is predicted by the model, producing 17 gesture predictions per recording. A winner-takes-all strategy is employed, setting the gesture label based on the most popular prediction.

The first attempt at recognising a live gesture was unsuccessful. Upon examining a live recording of 'no gesture', it was found that a disparity in scaling parameters between the two datasets was causing the model to misclassify gestures. In the training dataset, the range of features varied significantly: the ranges in X, Y and Z accelerations alone were 90.65, 118.59, and 48.32 respectively. Conversely, the live data for 'no gesture' exhibited markedly different acceleration ranges: 0.12, 0.10 and 0.31 respectively. Since the scalers had been applied separately, both sets of ranges were normalised to between 0 and 1 irrespective of each other, effectively amplifying the smaller accelerations in the live data to match the larger values from the static data. These inaccurate transformations of the live data by the scaler were impacting the model's performance. To amend this, the scaling parameters from the training set were exported, saved and imported into the live testing program. This ensured that the live data was always scaled with respect to the training data. The flowchart in Figure 47 visualises the complete live gesture capture and recognition process. (For program and flowchart source codes, go to Appendix A).

For the live gesture recognition test, the selection of gestures to be tested sequentially was primarily randomised. However, to ensure an equal occurrence of each gesture, uniform distribution was enforced to obtain sufficient classification results for all gestures. Additionally, it was skewed to account for all possible neighbouring gestures. This approach was taken to accommodate potential challenges in classification, since there was concern that certain gestures may affect the recognition of others when performed in succession. Every gesture from the list was performed before revisiting the Python terminal to record the results for each gesture. The results of the test are presented in Table 6.

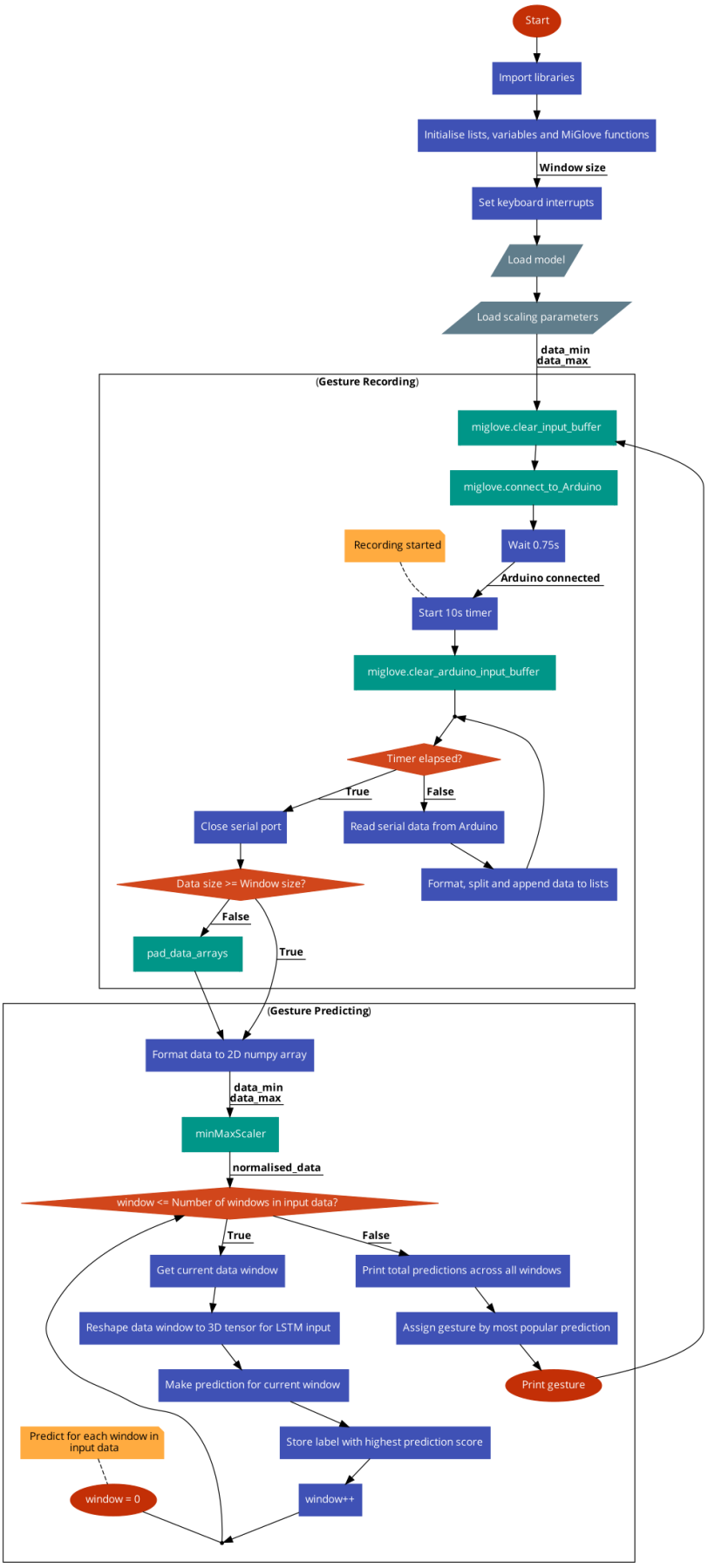


Figure 47: Flowchart depicting live gesture capture and recognition (created using code2flow).

Table 6: Results from first live gesture recognition test at 9600bps.

Index	Correct Gesture	Gesture Guesses				Predicted Gesture
		Hammering	Sawing	Screwdriving	No Gesture	
1	No gesture	0	0	0	16	No gesture
2	Hammering	0	12	0	0	Sawing
3	No gesture	0	0	0	15	No gesture
4	Sawing	2	11	0	2	Sawing
5	No gesture	0	0	3	12	No gesture
6	Screwdriving	0	0	10	1	Screwdriving
7	Hammering	0	9	0	2	Sawing
8	No gesture	0	0	0	9	No gesture
9	Sawing	0	14	0	0	Sawing
10	Screwdriving	3	0	0	10	No gesture
11	No gesture	0	0	2	10	No gesture
12	Sawing	0	16	0	0	Sawing
13	Hammering	2	10	3	0	Sawing
14	No gesture	0	0	0	13	No gesture
15	No gesture	0	0	0	9	No gesture
16	Hammering	1	3	0	9	No gesture
17	Hammering	0	9	0	0	Sawing
18	No gesture	0	0	1	15	No gesture
19	Sawing	2	7	0	0	Sawing
20	Sawing	0	15	0	0	Sawing
21	Screwdriving	0	0	11	3	Screwdriving
22	Screwdriving	14	0	0	0	Hammering

The definitions for the classification metrics in the context of this project are listed below:

True Positives (TP): The model correctly identifies the gesture in the input data.

True Negative (TN): The model correctly identifies the absence of any gesture when no gesture is present. In this case, however, “no gesture” is a classification type, so true negatives are irrelevant.

False Positives (FP): Incorrect classification of the input gesture.

False Negatives (FN): Failure to identify any gesture in the input data. Not relevant in this model since a gesture classification is always returned.

From the live gesture recognition test, 15 out of 22 gestures were correctly identified, giving a precision of 0.682. In this iteration of the model, a gesture classification is always returned. Therefore, there were no negatives, and the recall score was 1. The winner-takes-all strategy, whilst simplifying implementation and enhancing system responsiveness, compromises reliability. In this approach, gestures with lower prediction scores may still be classified, even when the model is uncertain. This challenge is revisited later in the report where potential enhancements in classification reliability are suggested through the adoption of more sophisticated prediction logic. However, for an initial iteration, the simplicity of the winner-takes-all strategy enabled rapid development and provided quick validation of the overall system concept.

		Predicted			
Actual	0	4	0	1
	0	5	0	0
	1	0	2	1
	0	0	0	8

Figure 48: Confusion matrix for dynamic performance of the first test.

The confusion matrix in Figure 48 show that sawing and no gesture classifications were successful. However, screwdriving was only classified correctly 50% of the time, and hammering could not be classified correctly at all.

7.5.3 Troubleshooting the Hammering Classification Problem

To investigate the issues with classifying the hammering gesture, a closer examination of the MPU6050 data was conducted. The raw data streams (shown by the blue lines in Figure 49) were noticeably unrefined and constantly fluctuating. Since hammering involves periodic strokes, these strokes could be misinterpreted as random fluctuations which exist in the training data of other gestures. A moving average filter was applied to the MPU6050's acceleration and gyro data to observe the resolution of the data and create a smooth motion path. These modifications were tested by recording a simple gesture and comparing the original and filtered data at both baud rates. The gesture performed was an extension and flexion of the right elbow in the transverse plane. The results at 9600bps and 115200bps are shown in Figures 49 and 50 respectively.

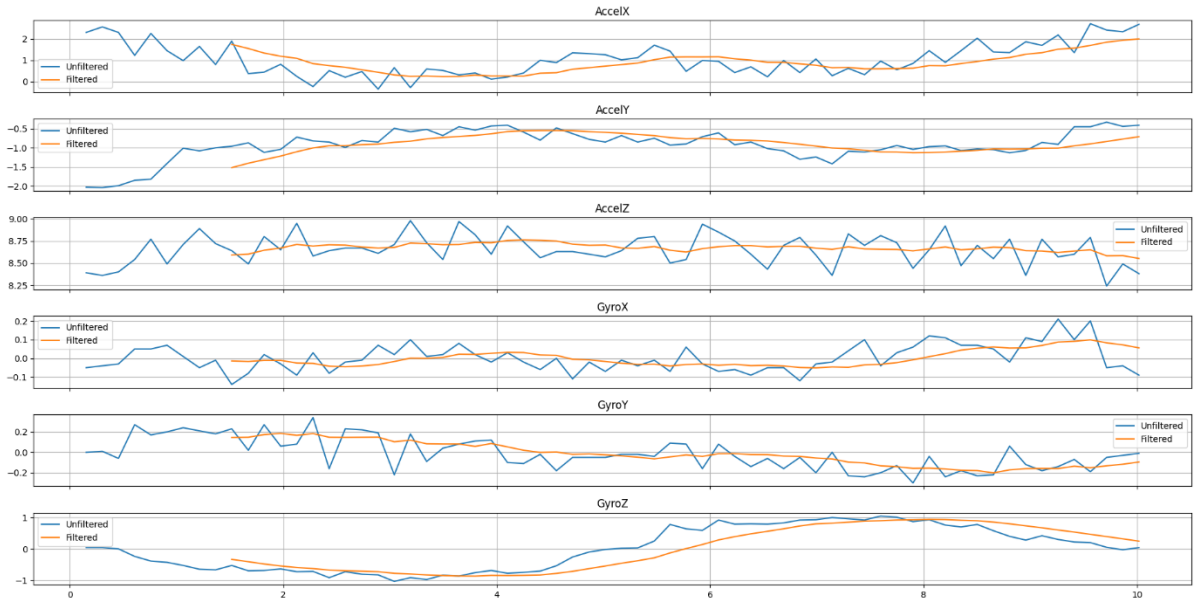


Figure 49: MPU6050 sensor values against time during full elbow extension-flexion using a baud rate of 9600.

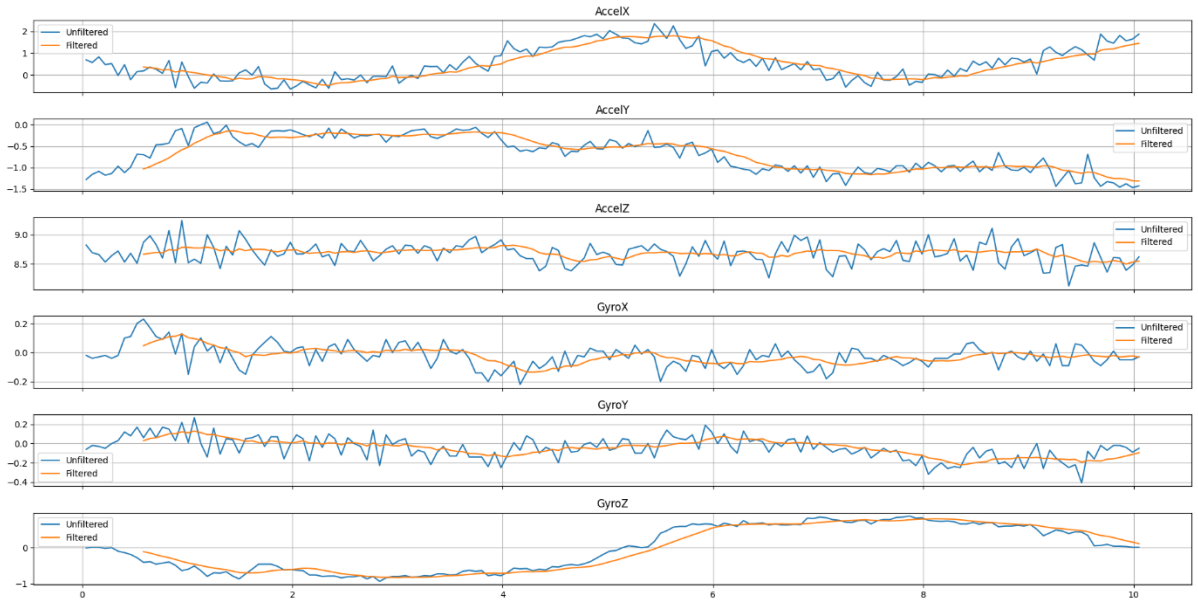


Figure 50: MPU6050 sensor values against time during full elbow extension-flexion using a baud rate of 115200.

Evidently, the moving average filter greatly reduces the fluctuations in the motion data from the MPU6050 (shown by the yellow lines in Figures 49 and 50). At 9600bps, the rate of data acquisition was 6.6/s, whilst at 115200bps, the rate was 16.6/s. The faster baud rate provided greater resolution, allowing it to capture nuances in the motion patterns which could not be detected at 9600bps. However, it was noted that increasing the baud rate would require updating the software to manage much more data per gesture (due to the faster acquisition rate). This would require a new training dataset which would be more time-consuming to generate, as opposed to simply adding the moving average filter to the existing dataset at the lower baud rate.

Unfortunately, whilst the moving average filter does present smooth data capture in Figures 49 and 50, gestures such as hammering can be performed at a variable frequency depending on the user. This means the sensor's resolution at this lower baud rate is potentially too low for the sensor to successfully capture each hammer stroke. To determine whether this was true, the filter and baud rate variations were subsequently applied to the hammering gesture. The results are shown in Figures 51 and 52.

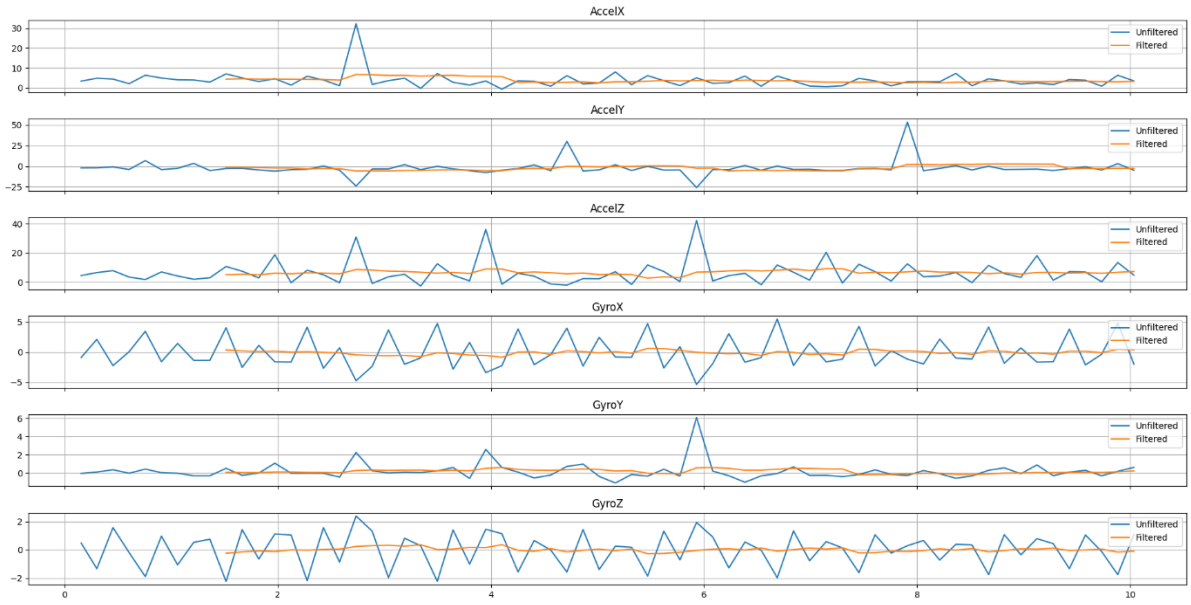


Figure 51: MPU6050 sensor values against time while hammering using a baud rate of 9600.

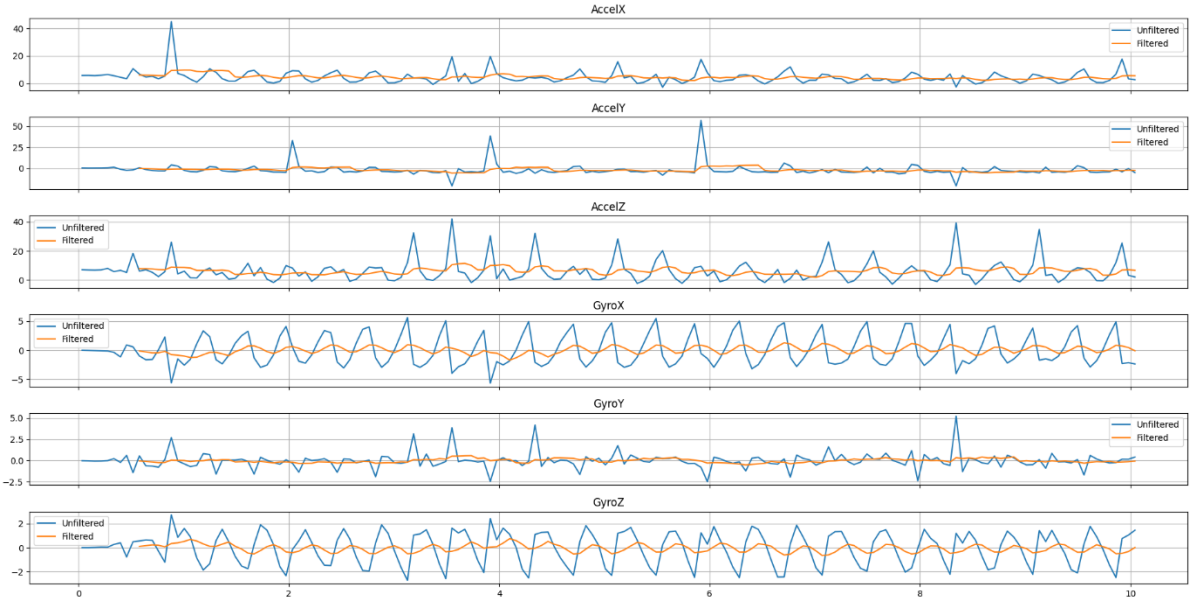


Figure 52: MPU6050 sensor values against time while hammering using a baud rate of 115200.

A total of 25 hammer strokes were performed in a 10 second window. These 25 peaks are clearly visible in Figure 52 by the positive peaks in GyroX and negative peaks in GyroZ. The gyro data best represents the gesture because the hammer acts as a lever, using its own weight to move in an arc. This means the hand (acting as the pivot), rotates rather than moving vertically, so whilst AccelZ does capture some motion, it is less representative of the gesture. With the filter applied, there are still noticeable fluctuations in the yellow filtered line, making the gesture easier to identify manually. However, in Figure 51, whilst GyroX and GyroZ do show these same 25 fluctuations, the resolution is too low, such that the filtered data shows too little variation to identify individual hammer strokes manually. Although the LSTM model may be able to detect less obvious trends in this data, the comparisons between Figures 51 and 52 suggested that switching to a higher baud rate could improve the recognition of the hammering gesture. Additionally, the results indicated that adding a filter would ultimately provide little improvement and may remove other features of the data which may be useful for classification, especially at the lower baud rate where the filtered data appears to flatline due to the lack of data.

Another observation is that at 9600bps, the hammering gesture rarely shows AccelZ peaks with amplitude between $30ms^{-2}$ and $50ms^{-2}$, which the model may rely upon when recognising hammering gestures. If these peaks are present in the training set but not in the live data during a hammering gesture, (such as if the live user is more controlled or moves slower), the model may struggle to identify the correct gesture. At 115200bps, these peaks are more frequent and more closely relate to the periodic hammer strokes.

7.5.4 Solving Hardware Issues

Before updating the software to a higher baud rate, an additional observation was made. Extensive testing at 9600bps had revealed that the flex sensor connections were very fragile (see Figure 53); the multi-core wires in the potential divider circuits had been tinned (i.e. the ends of their individual threads soldered together into single-core) to improve their robustness; however, these would occasionally snap mid-gesture, particularly with more aggressive motion such as hammering. It was theorised that by undergoing tensile stress and deformation as the glove material (and consequently the attached sensor wires) stretched mid-gesture, some noise was being introduced into the recording from the poor electrical connections; in worst cases the connections would snap completely. This seemed to be affecting the gesture recognition performance.

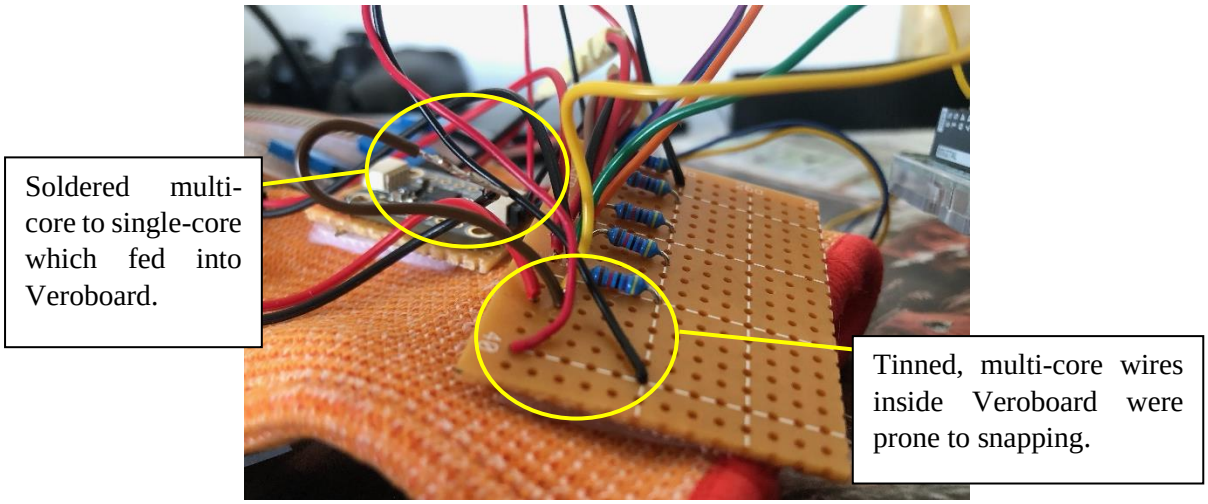


Figure 53: Hardware issues with wiring into Veroboard.

To determine whether this was true, the glove hardware was revised: velour tape was replaced to refresh the adhesive on the glove and minimise sensor drift across the glove's surface; also, frayed multi-core wires were replaced with stronger, single-core wires to improve electrical contact (shown in Figure 53). The revised design was again tested at a baud rate of 9600bps. The results are shown in Table 7, with correct classifications in green, and incorrect in red.

Table 7: Results from second live gesture recognition test at 9600bps with updated hardware.

Index	Correct Gesture	Gesture Guesses				Predicted Gesture
		Hammering	Sawing	Screwdriving	No Gesture	
1	No gesture	0	0	0	16	No gesture
2	Hammering	12	0	0	0	Hammering
3	No gesture	0	0	0	16	No gesture
4	Sawing	3	13	0	0	Sawing
5	No gesture	0	0	0	9	No gesture
6	Screwdriving	0	0	12	0	Screwdriving
7	Hammering	15	0	0	1	Hammering

8	No gesture	0	0	0	15	No gesture
Break						
9	Sawing	16	0	0	0	Hammering
10	Screwdriving	3	0	0	10	No gesture
11	No gesture	0	0	0	10	No gesture
12	Sawing	0	16	0	0	Sawing
13	Hammering	16	0	0	0	Hammering
14	No gesture	0	0	0	14	No gesture
15	No gesture	0	0	0	15	No gesture
16	Hammering	8	0	0	0	Hammering
17	Hammering	12	0	0	0	Hammering
18	No gesture	0	0	0	14	No gesture
19	Sawing	0	16	0	0	Sawing
20	Sawing	0	6	0	0	Sawing
21	Screwdriving	11	0	0	3	Hammering
22	Screwdriving	14	0	0	0	Hammering

18 out of 22 gestures were correctly identified, giving a precision of 0.818. However, the confusion matrix in Figure 54 shows that screwdriving was greatly misclassified. Conversely, the issue with classifying the hammering gesture had been resolved.

		Predicted			
Actual	5	0	0	0
	1	4	0	0
	2	0	1	1
	0	0	0	8

Figure 54: Confusion matrix for dynamic performance of the second test.

7.5.5 Examining the Difficulties in Differentiating Screwdriving from Hammering

Revising the glove hardware had improved the overall accuracy of the program by about 13.4% without increasing the baud rate. However, whilst issues with recognising the hammering gesture had been fixed, they had been replaced with several misclassifications of the screwdriving gesture. It is possible that there was an existing issue with recognising screwdriving, and that it had either been disguised by the poor electrical connections, or the action had been performed slightly differently between the two tests. Before resorting to using the higher baud rate, and having revised the electrical connections, more experimentation was conducted to understand why the screwdriving detection accuracy had fallen. Further testing with the screwdriving gesture is presented in Table 8:

Table 8: Results from third live gesture recognition test focused on screwdriving at 9600bps.

Index	**Correct Gesture**	**Gesture Guesses**				**Predicted Gesture**
		Hammering	**Sawing**	**Screwdriving**	**No Gesture**	
1	Screwdriving	12	0	0	4	Hammering
2	Screwdriving	7	0	0	8	No gesture
Break						
3	Screwdriving	0	0	10	0	Screwdriving
4	Screwdriving	0	0	14	0	Screwdriving
5	Screwdriving	0	0	10	0	Screwdriving
6	Screwdriving	0	0	8	0	Screwdriving
7	Screwdriving	0	0	12	0	Screwdriving
8	Screwdriving	0	0	15	0	Screwdriving

The prototype was visually monitored during the gesture recording. It was noticed that the Arduino mega was periodically bumping the Veroboard mounted to the glove, possibly generating sharp

disturbances in the MPU6050 readings with periods matching that of the screwdriving motion. Since the hammering gesture also exhibits periodic shockwaves per gesture, these sharp peaks could be mistaken for hammer strokes. The grip of the hammer and screwdriver was also very similar so the flex sensor readings would be relatively indiscernible, thus limiting data variation to assist with classification. During a short interval, the Arduino was manoeuvred around the wrist to prevent unwanted contact with the glove's Veroboard mid-recording. The following tests showed that screwdriving could then be identified correctly.

Another aspect affecting the screwdriving gesture was the non-uniform force applied down the screwdriver onto the applied surface; if the user applied more downward force when rotating, the gesture data probably resembled the hammering gesture. To confirm this, the MPU6050 data for the screwdriving gesture was observed.

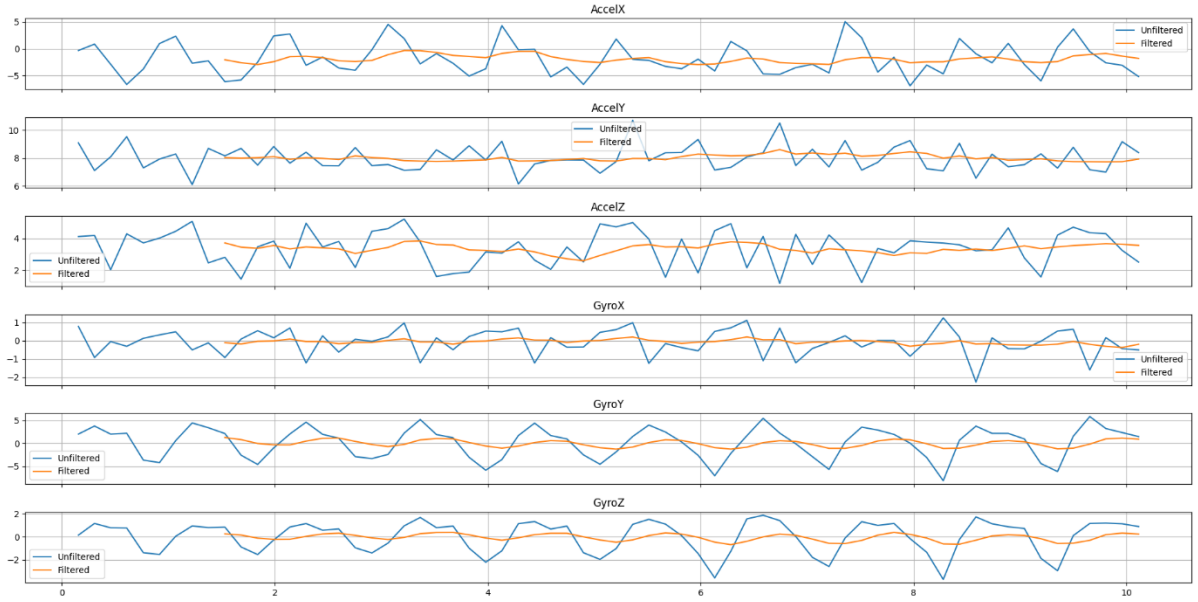


Figure 55: MPU6050 sensor values against time while screwdriving onto desk using a baud rate of 9600.

Figure 55 shows that there is a somewhat oscillatory nature in AccelX and each of the Gyro plots during the screwdriving gesture, not unlike the hammering gesture in Figure 51 (which also exhibited oscillatory behaviour in gyro data).

The fewer pronounced peaks in the AccelZ values in Figure 56 compared to Figure 55 stem from the absence of a reaction force generated when the screwdriver hits the desk. In Figure 55, these spikes in AccelZ occur due to the screwdriver striking the desk mid-gesture, whilst in Figure 56, the gesture is performed in the air, eliminating this impact and resulting in fewer sharp peaks. Using this result, subsequent screwdriving gestures could be better differentiated from the hammering gesture by minimising the downward force onto the desk.

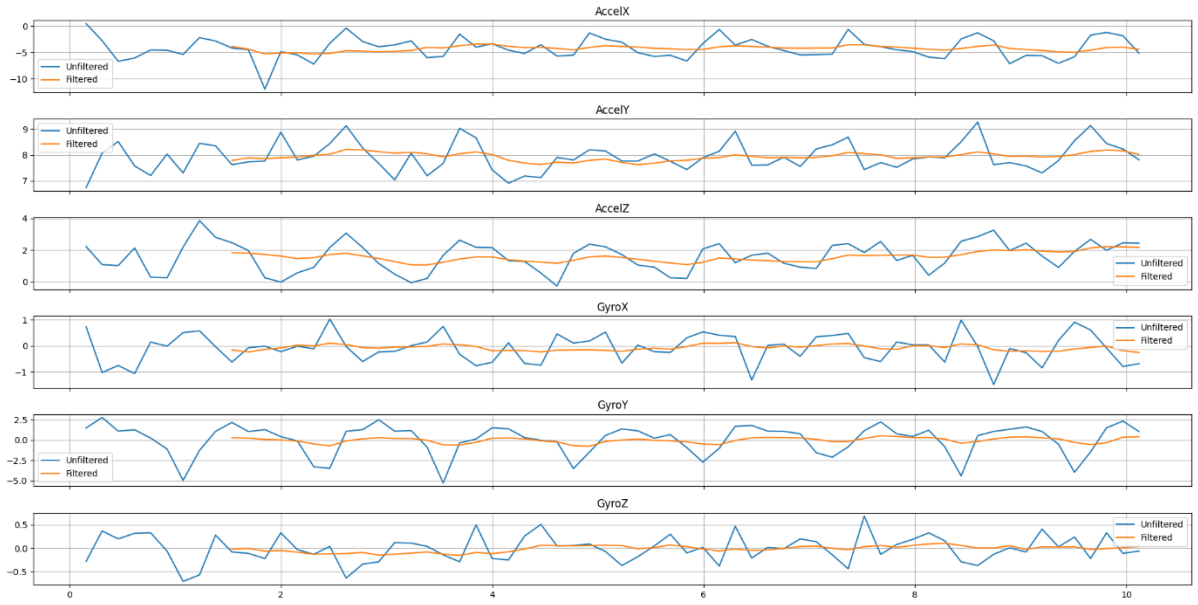


Figure 56: MPU6050 sensor values against time while screwdriving in air using a baud rate of 9600.

Performing gestures in specific ways is not very practical, since most users will exhibit nuances in their movement and gestures that may differ greatly from the training data. However, this problem is relatively insignificant, because realistic deployment of this project would involve training the model with multiple different users to capture these nuances. Ultimately, providing the model with as much user data as possible would significantly improve gesture recognition results. However, to prove the concept, testing extreme versions of each gesture to mimic this user variability was not required.

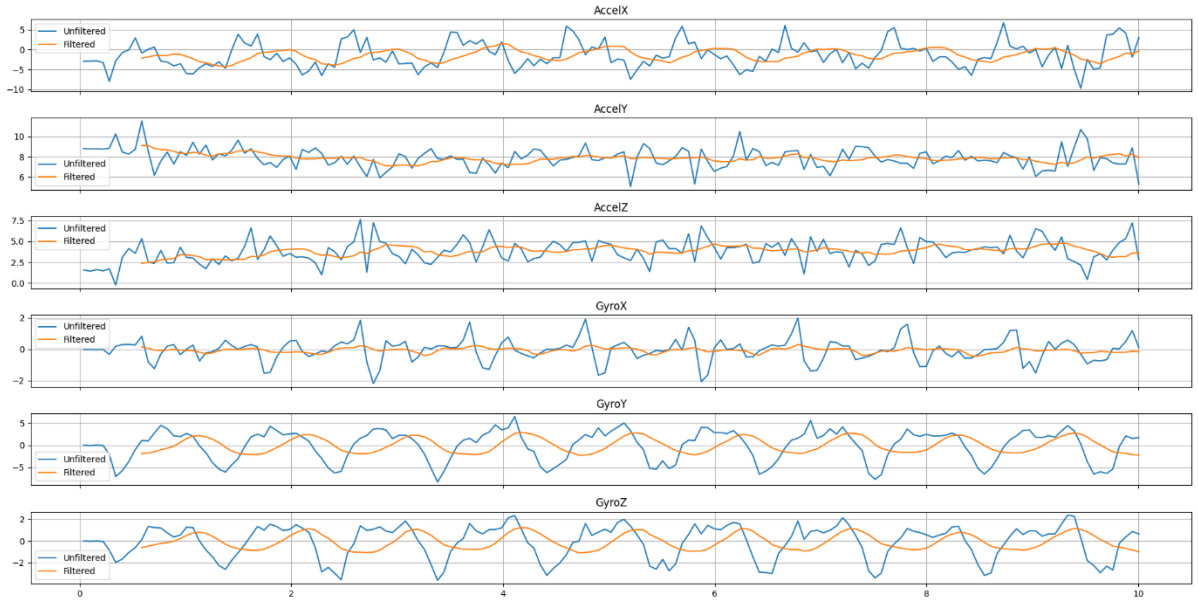
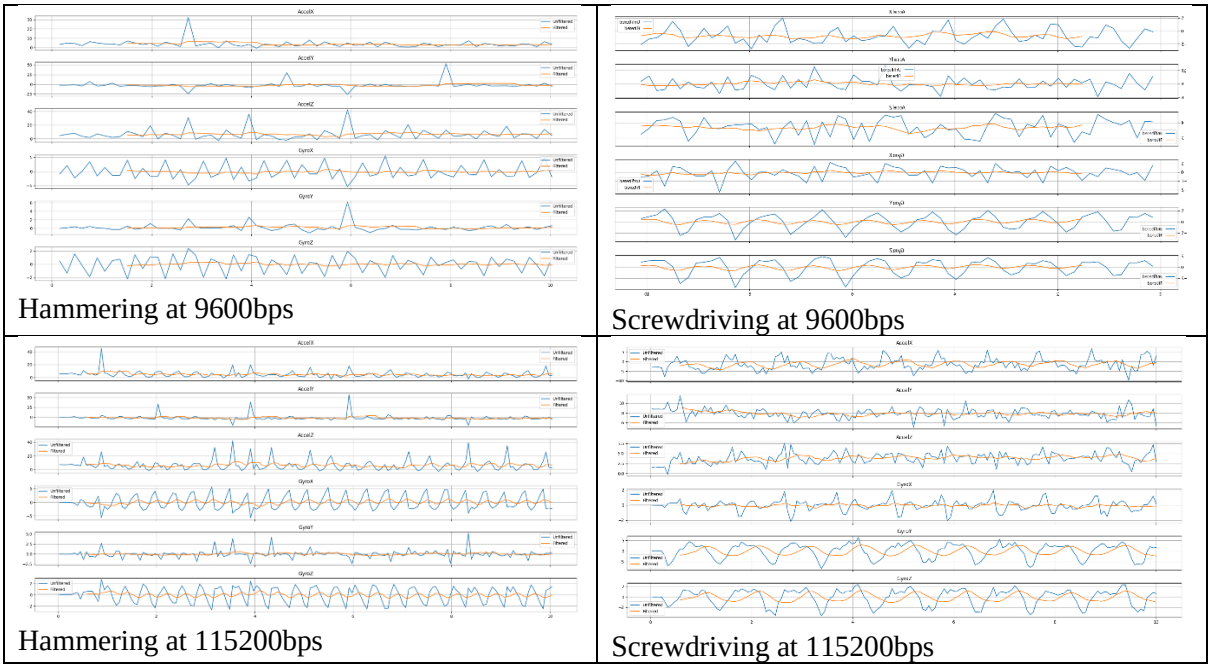


Figure 57: MPU6050 sensor values against time while screwdriving using a baud rate of 115200.

In Figure 55, AccelX again shows periodic motion, however it is much more noticeable at the higher baud rate than in Figure 57. The AccelY and AccelZ values are less useful in identifying this gesture due to their more sporadic natures. Gyro data shows the greatest variation as expected, since screwdriving typically involves rotating the wrist rather than moving the hand around in 3D space.



Hammering at 9600bps

Screwdriving at 9600bps

Hammering at 115200bps

Screwdriving at 115200bps

Note: This is a collection of Figures 51, 52, 55 and 57. Please check original Figures for details.

Upon comparing the hammering and screwdriving gestures at both baud rates, it seemed that the two gestures were slightly more discernible at 115200bps than at 9600bps. It is possible that the GyroX and GyroZ data of the two gestures were not dissimilar enough to reliably determine one over the other. Furthermore, the shape of the screwdriving gyro peaks at 9600bps could potentially be mistaken for a slower hammering gesture; the accelerometer readings also appear more random at this resolution, which were probably impeding recognition.

7.5.6 New Data Collection using Updated Baud Rate and Filter

Overall, the hammering and screwdriving analyses both indicated that a higher accuracy was potentially attainable by using a higher baud rate. As a result, the entire MiGlove software was updated to use the higher baud rate of 115200bps. This meant that recording a new gesture dataset was required to include more data points per gesture; in a 10-second window, 166 data points could be captured as opposed to only 66 at 9600bps. The number of data points per gesture was increased from 50 to 150. An additional 30 recordings were produced for the 4 gestures (hammering, sawing, screwdriving and no gesture). The live data feed window was also increased up to 150 to accommodate the larger data arrays.

7.5.7 Static Performance of New Model

The new model was trained using 90% of 120 newly recorded gestures. Again, the training sample was selected by stratifying the y-labels to remove class bias. All 12 of the gestures in the testing set were identified correctly, giving a test accuracy of 100%, with less than 0.00176 loss. The confusion matrix for this performance is shown in Figure 58:

Actual	Predicted			
	3	0	0	0
	0	3	0	0
	0	0	3	0
	0	0	0	3

Figure 58: Confusion matrix for static performance of the new model.

Figure 58's leading diagonal shows that the static performance of the new model was very good. The model was subsequently tested with live input data to check its dynamic performance.

7.5.8 Dynamic Performance with Live Data at 115200bps

Breaks were introduced between sets of 10 gestures. The gesture list was generated randomly yet in such a way that each gesture appeared 10 times, providing a total of 40 gestures to test. This was done so that each gesture could be tested equally. Results are shown in Table 9.

Table 9: Results from fourth live gesture recognition test at 115200bps with updated hardware.

Index	Correct Gesture	Gesture Guesses				Predicted Gesture
		Hammering	Sawing	Screwdriving	No Gesture	
1	Sawing	0	15	0	0	Sawing
2	Screwdriving	0	0	16	0	Screwdriving
3	Sawing	0	15	0	0	Sawing
4	Hammering	15	0	0	0	Hammering
5	Screwdriving	0	0	16	0	Screwdriving
6	Hammering	15	0	0	0	Hammering
7	No gesture	0	0	0	15	No gesture
8	Screwdriving	0	0	16	0	Screwdriving
9	Sawing	0	15	0	0	Sawing
10	Hammering	15	0	0	0	Hammering
Break						
11	No gesture	0	0	0	15	No gesture
12	Sawing	0	4	1	10	No gesture
13	Screwdriving	0	0	16	0	Screwdriving
14	No gesture	0	0	0	15	No gesture
15	Screwdriving	0	0	16	0	Screwdriving
16	Hammering	7	8	0	0	Sawing
17	No gesture	0	0	0	16	No gesture
18	Screwdriving	0	0	16	0	Screwdriving
19	Sawing	0	11	0	4	Sawing
20	No gesture	0	0	0	16	No gesture
Break						
21	Hammering	0	15	0	0	Sawing
22	Sawing	0	6	3	6	No gesture
23	Hammering	0	7	6	2	Sawing
24	Screwdriving	0	0	16	0	Screwdriving
25	No gesture	0	0	0	16	No gesture
26	Hammering	15	0	0	0	Hammering
27	Sawing	0	11	0	4	Sawing
28	Screwdriving	0	0	16	0	Screwdriving
29	No gesture	0	0	0	16	No gesture

30	Sawing	0	9	0	6	Sawing
Break						
31	Screwdriving	0	0	16	0	Screwdriving
32	No gesture	0	0	0	16	No gesture
33	Sawing	0	15	0	0	Sawing
34	Hammering	0	15	0	0	Sawing
35	No gesture	0	0	0	16	No gesture
36	Hammering	12	3	0	0	Hammering
37	Screwdriving	0	0	16	0	Screwdriving
38	No gesture	0	0	0	16	No gesture
39	Hammering	8	7	0	0	Hammering
40	Sawing	0	15	0	0	Sawing

34 out of 40 gestures were correctly classified, giving a precision of 0.85. The confusion matrix for these results is shown in Figure 59. With a much stronger leading diagonal than Figures 48 and 54, this matrix indicates that this iteration on the model was a significant improvement and provided more accurate, dynamic gesture recognition.

Actual \ Predicted				
	6	4	0	0
	0	8	0	2
	0	0	10	0
	0	0	0	10

Figure 59: Confusion matrix for dynamic performance of the new model.

Unfortunately, despite recognising most gestures correctly, the model was again somewhat struggling to recognise hammering. It should be noted that more flex sensor wires had snapped by the second break which needed replacing. Sensor locations may have shifted slightly during maintenance, which may have affected subsequent readings.

Before the break, gesture recognition seemed to be performing well, with only two mishaps: 'sawing' (index 12) and 'hammering' (index 16). Upon examining the prediction outputs for the 'sawing' classification, the highest class probability was only 0.71866 for 'no gesture'. Similar outputs were noticed in the misclassified hammering gesture. Since these outcomes are relatively low compared to other successful classifications with class probabilities above 0.9, this could indicate that a "winner-takes-all" strategy may not be sufficient to reliably determine a gesture. This could usefully be explored in future research.

The initial objective for this model was to achieve 80% accuracy with gesture recognition. Given that it achieved 85% accuracy, the objective was successfully met.

7.6 Achieving Robotic Collaboration in Simulation

The most recently identified gesture was passed to a simulation of a UR10 cobot in CoppeliaSim via Remote API with Python. A sequence of joint angles was set specifically for each gesture class, before performing forward kinematics to mimic the recognised gesture.

The flowchart in Figure 60 visualises the added robot simulation functionality to the live gesture recognition script. (For program and flowchart source codes, go to Appendix A).

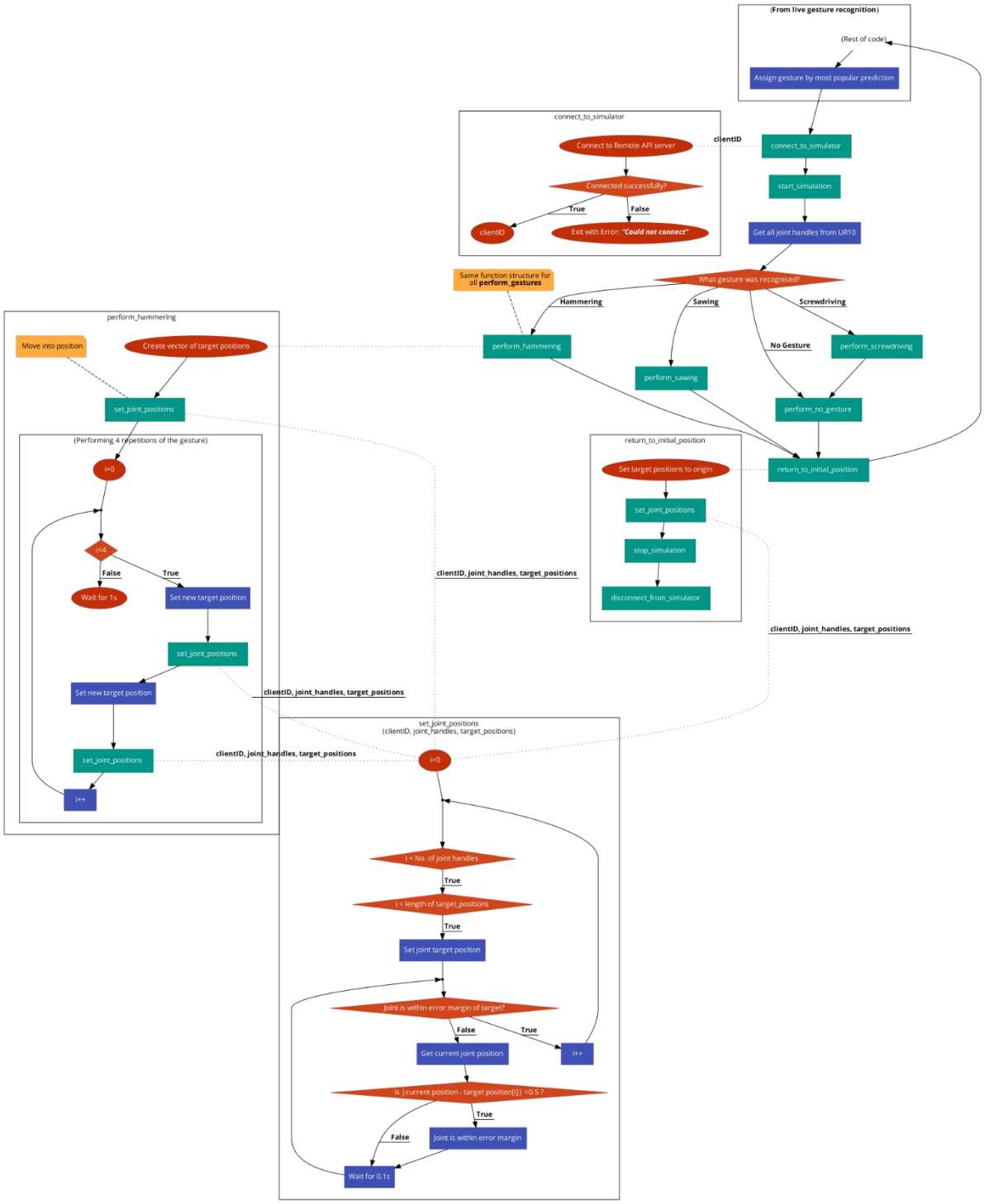
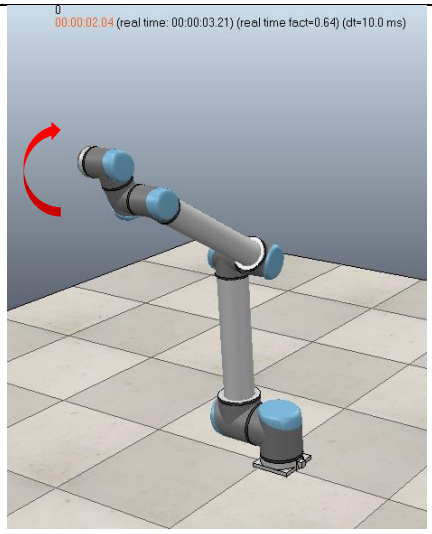
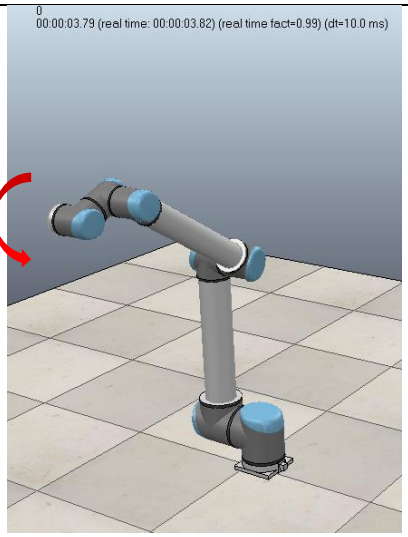
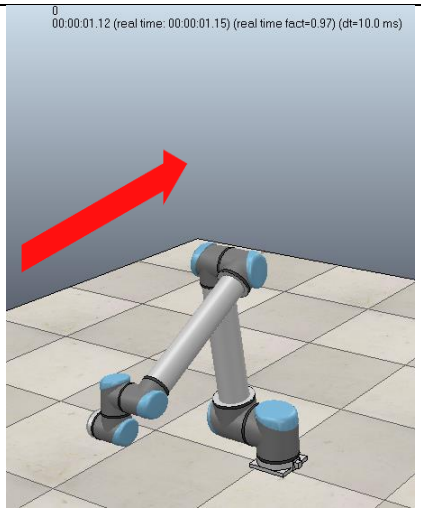
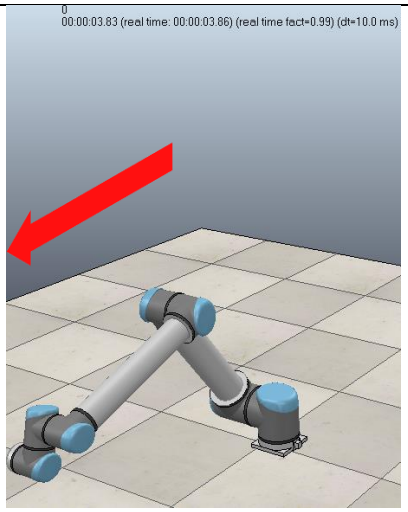
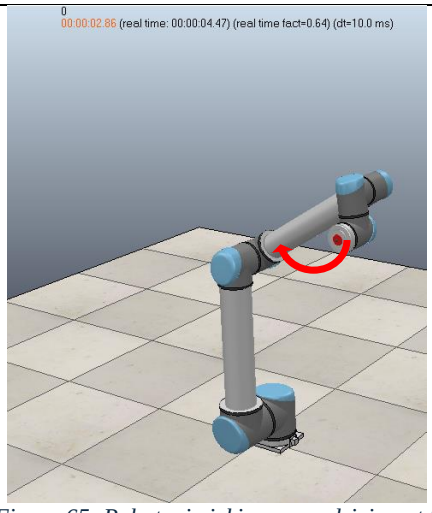
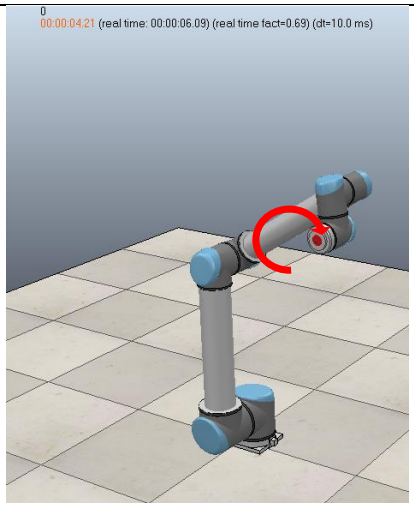
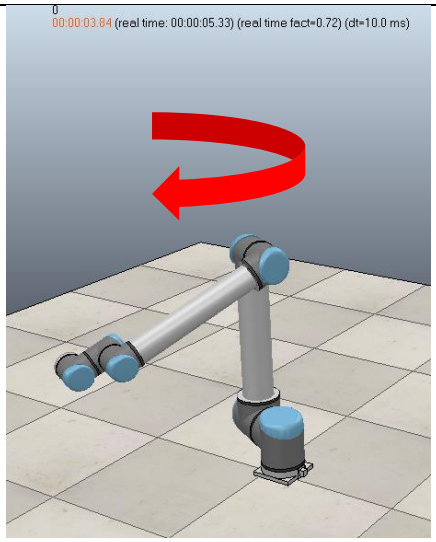
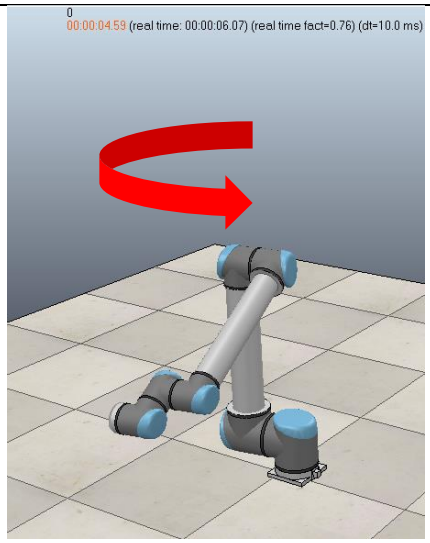


Figure 60: Flowchart depicting simulation of a UR10 robot mimicking recognised gestures using CoppeliaSim and Python (created using code2flow).

Table 10 displays UR10 cobot motion in response to MiGlove wearer gestures.

Table 10: Collection of CoppeliaSim UR10 cobot movements in response to specific gestures.

Gesture	Motion in Phase 1	Motion in Phase 2
Hammering	 *Figure 61: Robot mimicking hammering pt.1*	 *Figure 62: Robot mimicking hammering pt.2*
Sawing	 *Figure 63: Robot mimicking sawing pt.1*	 *Figure 64: Robot mimicking sawing pt.2*
Screwdriving	 *Figure 65: Robot mimicking screwdriving pt.1*	 *Figure 66: Robot mimicking screwdriving pt.2*

No Gesture	 *Figure 67: Robot shaking head in response to 'no gesture'pt.1*	 *Figure 68: Robot shaking head in response to 'no gesture' pt.2*

Cobot motion presented in Figures 61-68 could be substituted for tasks specific to the assembly procedure. This successfully demonstrates that, given a worker's classified gesture, the cobot will enact predefined instructions to best suit the worker's immediate needs.

8 DISCUSSION, CONCLUSIONS AND FUTURE WORK

The final design incorporated MPU6050 IMUs with flex sensors to create a multimodal glove capable of capturing and classifying live gesture data from the user. The glove transmitted data at 115200bps to an LSTM Sequential model, where it was reformatted and normalised into the correct input shape. The data was then windowed into sets of 150 points; each window was classified by the LSTM individually, storing each prediction before applying a winner-takes-all strategy – the class with the most window classifications became the overall predicted gesture. This model achieved a static recognition accuracy of 100% on 12 unseen gestures within the training set, and a dynamic recognition accuracy of 85%. Compared to other studies where classification using IMUs reached 98% [23] and 96.1% [21], this appears lower. However, this was likely due to hardware failures; recognition results were accurate across all 4 gestures before glove durability issues ensued. Simulations successfully demonstrated how HGR can be used to control cobots, facilitating improved task comprehension and workflow efficiency.

Other efforts using vision systems and CNNs achieved higher dynamic accuracies between 93% and 96%. However, there is a trade-off: their use of fixed cameras greatly limits their potential in assembly applications, as worker movements may obscure vision, or the system may focus on someone else in-frame. Additionally, cameras are typically much more expensive. Conversely, the MiGlove offers an affordable yet relatively accurate solution to gesture recognition; the flex sensors were particularly successful in capturing finger flexion data, which was crucial in recognising certain gestures. Sensor connections can be updated with negligible costs to raise the accuracy above 90% (shown by block 1 of Table 9), making it comparable to more expensive systems.

Other studies also obtained data from different users, or imported datasets where multiple participants were involved in data collection, increasing model flexibility and reusability. Due to the limited timeframe of this project and prioritising swiftness in development, training data was only obtained from one individual – this greatly reduces variety, as gestures can be performed with various styles across different individuals. Therefore, the model may not generalise well to new users or contexts. Future work would involve expanding the dataset to include 120 gestures from up to 5 different

individuals. This may reduce overall accuracy slightly, but will assist the model's ability to generalise, enhancing application in assembly sites where workers may switch places (e.g. between shifts).

This study demonstrates groundbreaking work by attempting to differentiate between hammering, sawing, screwdriving and no gesture. The relevant literature for hand gesture recognition typically involved sign language or basic gesture shapes with little relevance to proper manufacturing tasks. Classifying these tasks with 85% accuracy despite hardware malfunctions signifies a considerable leap forward in utilising HGR for collaborative manufacturing and assembly tasks. Future testing would add more manufacturing gestures to increase the range of available collaborative tasks.

The aim of this project was to design a wearable system which would allow humans to interact collaboratively with robots in assembly tasks. The achievement of this aim can be inferred from the objectives reached:

1. A bench prototype featuring 6 sensors for motion capture was designed, built and validated, with sensor values presented through graphs and screenshots.

2. A functional, wearable glove prototype was successfully built and tested, incorporating the 6 sensors to capture motion of the wielder's hand, including wrist and fingers. Sensor functionality was confirmed in situ.

3. 120 iterations of data representing 4 different hand gestures was systematically collected using the MiGlove rather than online datasets, due to the glove's unique data acquisition.

4. A single, LSTM model achieved 85% accuracy in live gesture recognition across 4 gestures. Since 100% static accuracy was achieved post-training and considering the time constraints, testing additional models was deemed unnecessary.

5. Demonstrated teleoperation or collaboration with robot simulation software, showcasing unique motions corresponding to all 4 gestures achieved in Objective 4.

The datasets and software were stored into a GitHub, serving as deliverable proof of achieving these objectives. These all indicate that the project aim was successfully achieved.

Previously, filtering MPU6050 data had been avoided due to concerns about potentially removing valuable features essential for classification. However, adding a moving average filter effectively removed large fluctuations in motion data; the significance of this on recognition rate should be explored in future studies.

Additionally, the model could be adapted to only accept prediction scores above a threshold of 90%. Examples of such thresholds in the relevant literature increased model specificity. Whilst the model may not classify a gesture for every window, it would remove less confident classifications and improve the reliability of the program. The priority for gesture classification reliability may vary based on the specific demands of the technology's intended use. For example, in scenarios where a worker is engaged in a complex task requiring concentration, it might be preferable to ensure reliable gesture recognition, even if it means delaying the next collaborative task caused by a less reliable gesture. Instead, the robot could wait for a more confident gesture classification before acting. This approach compromises the robot's reaction speed to prevent potential interruptions or safety concerns caused by incorrect collaborative tasks.

9 PROJECT REVIEW

9.1 SUCCESSES AND ACHIEVEMENTS

A thorough project plan enabled effective time management and provided a comprehensive understanding of the project's critical path – this helped to prioritise tasks to meet key objectives. Achieving these objectives indicated clear progression and provided encouragement to continue development. Additionally, regular meetings with the project supervisor streamlined development by quickly obtaining solutions challenges. Meeting agendas were prepared in advance to avoid digression, and minutes were well-documented to record decision-making and troubleshooting ideas. Critical application of engineering knowledge combined with the use of observation and analysis tools enabled swift diagnosis of challenges associated with recognition performance.

9.2 SETBACKS AND IMPROVEMENTS

The final glove build was too fragile for extensive testing, especially when hammering. Frequent wiring issues interfered with dataset collection and live testing. If repeated, a revamped glove design would be produced, recreating the entire Veroboard setup, as well as soldering wires directly to flex sensor tabs rather than fitting clincher connectors. This would greatly increase durability and improve overall recognition accuracy, as well as streamline glove design adjustments, allocating additional time for optimising the LSTM model or testing an SVM model. Additional time consolidating the Keras model library would allow for more complex models such as CNN/LSTMs to be developed.

Some components with longer lead times were scheduled to arrive during Easter break, potentially delaying prototype completion. To avoid this, part orders were amended. However, confusion over order cancellations led to a slight budget overrun. Consequently, costs were reduced elsewhere to address this issue, though simply informing the unit convenor to extend the budget would have sufficed.

10 ACKNOWLEDGEMENTS

I would like to express my gratitude to my project supervisor, Dr. Uriel Martinez Hernandez, for his insightful feedback and invaluable guidance. His support and genuine interest in my work helped me to achieve more ambitious targets and ultimately create a product of which I can be proud.

I extend my thanks to Jack Coombes and Chun Fai Wong from the EEE storerooms for providing the necessary lab materials, equipment and helpful advice essential for prototyping.

I would also like to acknowledge my peer, William Fox, for his guidance in using Keras for machine learning models. Will's assistance streamlined the model generation process, allowing me to devote more time towards testing and refining the design.

Lastly, I would like to thank my family and peers who have supported me throughout my degree. Their encouragement, advice, and unwavering belief in my abilities have been indispensable in overcoming challenges and achieving milestones throughout my studies.

To streamline software development, ChatGPT 3.5 was utilised solely for code debugging purposes. Creating the gesture datasets required certain CSV export functions and formatting. Addressing data format errors during debugging was not viewed as value-adding to the project, hence ChatGPT 3.5 was utilised to expedite this process.

11 REFERENCES

[1] Desoutter, "Industrial Revolution - From Industry 1.0 to Industry 4.0," Desoutter, [Online]. Available: https://uk.desouttertools.com/your-industry/news/1242/industrial-revolution-from-industry-1-0-to-industry-4-0-2#:~:text=3rd%20Industrial%20Revolution,production%20process%20%2D%20without%20human%20assistance.. [Accessed 2 May 2024].

[2] G. Avalle, F. De Pace, C. Fornaro, F. Manuri and A. Sanna, "An Augmented Reality System to Support Fault Visualization in Industrial Robotic Tasks," *IEEE Access,* vol. 7, no. n/a, pp. 132343-132359, 2019.

[3] W. Xu, J. Cui, B. Liu, J. Liu, B. Yao and Z. Zhou, "Human-robot collaborative disassembly line balancing considering the safe strategy in remanufacturing," *Journal of Cleaner Production,* vol. 324, no. n/a, p. n/a, 2021.

[4] A. Keshvarparast, D. Battini, B. Olga and A. Pirayesh, "Collaborative robots in manufacturing and assembly systems: literature review and future research agenda," *Journal of Intelligent Manufacturing,* vol. 35, no. 4, p. n/a, 2024.

[5] D. M. West and C. Lansang, "Global manufacturing scorecard: How the US compares to 18 other nations," Brookings, [online], 2018.

[6] R. Gihleb, O. Giuntella, L. Stella and T. Wang, "Industrial robots, Workers'safety, and health," *Labour Economics,* vol. 78, no. n/a, p. n/a, 2022.

[7] M. Golin and C. Rauh, "Workers' responses to the threat of automation," CEPR, [online], 2023.

[8] B. Marr, "The Best Examples of Human and Robot Collaboration," Forbes, [online], 2022.

[9] J. Smids, S. Nyholm and H. Berkers, "Robots in the Workplace: a Threat to—or Opportunity for—Meaningful Work?," *Philosophy & Technology,* vol. 33, no. n/a, pp. 503-522, 2020.

[10] R. Riener, L. Rabezzana and Y. Zimmermann, "Do robots outperform humans in human-centered domains?," Frontiers in Robotic AI, [online], 2023.

[11] T. Dawson, "Why industrial robot component manufacturers should target cobots," The Robot Report, [online], 2021.

[12] B. Wilhelm, B. Manfred, M. Braun, P. Rally and O. Scholtz, "Lightweight robots in manual assembly – best to start simply! Examining companies' initial experiences with lightweight robots," Fraunhofer IAO, [online], 2016.

[13] H. J. Wilson and P. R. Daugherty, "Robots Need Us More Than We Need Them," Harvard Business Review, [online], 2022.

[14] M. Lazarte, "Robots and humans can work together with new ISO guidance," ISO, 8 March 2016. [Online]. Available: https://www.iso.org/news/2016/03/Ref2057.html#:~:text=Factors%20to%20be%20considered%20in,monitoring%3B%20power%20and%20force%20limiting.. [Accessed 29 April 2024].

[15] ISO, “ISO 13849-1:2023 | Safety of Machinery,” ISO, 26 April 2023. [Online]. Available: https://www.iso.org/standard/73481.html. [Accessed 29 April 2024].

[16] M. Schnell and M. Holm, “Challenges for Manufacturing SMEs in the Introduction of Collaborative Robots,” IOS Press, Skovde, 2022.

[17] J. Male and U. M. Hernanez, “Recognition of human activity and the state of an assembly task,” in *22nd Conference on Industrial Technology (ICIT)*, [online], 2021.

[18] C. Nuzzi, S. Pasinetti, M. Lancini, F. Docchio and G. Sansoni, “Deep Learning Based Machine Vision: First Steps Towards a Hand Gesture Recognition Set Up for Collaborative Robots,” *2018 Workshop on Metrology for Industry 4.0 and IoT*, vol. n/a, no. n/a, pp. 28-33, 2018.

[19] H.-I. Lin, M.-H. Hsu and W.-K. Chen, “Human hand gesture recognition using a convolution neural network,” in *IEEE International Conference on Automation Science and Engineering (CASE)*, New Taipei, IEEE, 2014, pp. 1038-1043.

[20] W. Dumoulin, N. Thry and R. Slama, “Real Time Hand Gesture Recognition in Industry,” in *Proceedings of the 2021 3rd International Conference on Video, Signal and Image Processing*, New York, 2022.

[21] G. Yuan, X. Liu, Q. Yan, S. Qiao, Z. Wang and L. Yuan, “Hand Gesture Recognition Using Deep Feature Fusion Network Based on Wearable Sensors,” *IEEE Sensors Journal*, vol. 21, no. 1, pp. 539-547, 2021.

[22] IBM, “What are support vector machines (SVMs)?,” IBM, 27 December 2023. [Online]. Available: https://www.ibm.com/topics/support-vector-machine. [Accessed 9 April 2024].

[23] B. Ajay, S. Aditya, A. Adarsha, N. Deekshitha and K. Harshitha, “Hand Gesture Recognition System Using IoT and Machine Learning,” *Intelligent Learning for Computer Vision*, vol. 61, pp. 393-404, 2021.

[24] D. Mace, W. Gao and A. K. Coskun, “Accelerometer-Based Hand Gesture Recognition using Feature Weighted Naive Bayesian Classifiers and Dynamic Time Warping,” in *IUI'13 Companion*, Santa Monica, 2013.

[25] E. Keogh and A. Ratanamahatana, “Exact Indexing of Dynamic Time Warping,” in *Proceedings of International Conference on Very Large Data Bases (VLDB)*, Riverside, 2002.

[26] C.-Y. Yang, P.-Y. Chen, T.-J. Wen and G. E. Jan, “IMU Consensus Exception Detection with Dynamic Time Warping—A Comparative Approach,” *Sensors*, vol. 19, p. 2237, 2019.

[27] M. Atzori, A. Gijsberts, I. Kuzborskij, S. Heynen, A.-G. M. Hager, O. Deriaz, C. Castellini, H. Muller and B. Caputo, “Characterization of a Benchmark Database for Myoelectric Movement Classification,” *Transactions on Neural Systems and Rehabilitation Engineering*, vol. 23, no. 1, pp. 73-83, 2015.

[28] Hochreiter, “tf.keras.layers.LSTM,” Tensorflow, [online], 1997.

[29] J. Hitchen, “Project Scoping and Planning - MiGlove,” University of Bath, Bath, 2024.

[30] I. M. Bullock and J. Borras, “Assessing assumptions in kinematic hand models: A review,” in *Biomedical Robotics and Biomechatronics (BioRob)*, [online], 2012.

[31] S. Hochreiter and J. Schmidhuber, “Long Short-term Memory,” PubMed, [online], 1997.

[32] G. Furnieles, “Sigmoid and SoftMax Functions in 5 minutes,” Towards Data Science, [online], 2022.

[33] S. Varsamopoulos, K. Bertels and C. G. Almudever, “Designing neural network based decoders for surface codes,” ResearchGate, Delft, 2018.

[34] D. Thakur, “LSTM and its equations,” Medium, [online], 2018.

[35] Evidently AI Team, “How to interpret a confusion matrix for a machine learning model,” Evidently AI, [Online]. Available: https://www.evidentlyai.com/classification-metrics/confusion-matrix. [Accessed 3 May 2024].

[36] B. Siepert, “MPU6050 6-axis Accelerometer and Gyro Datasheet,” Adafruit, [online], 2019.

[37] T. Gundry, “Adafruit MPU6050 GIthub Repository,” Github, [online], 2023.

[38] Sparkfun, “Flex Sensor Hookup Guide,” Sparkfun, [Online]. Available: https://learn.sparkfun.com/tutorials/flex-sensor-hookup-guide. [Accessed 25 03 2024].

[39] RS Components, “Arduino Uno Rev 3,” RS Components, [Online]. Available: https://uk.rs-online.com/web/p/arduino/7154081. [Accessed 23 April 2024].

[40] RS Components, “Arduino Mega 2560 Rev 3,” RS Components, [Online]. Available: https://uk.rs-online.com/web/p/arduino/7154084. [Accessed 23 April 2024].

[41] RS Components, “STMicroelectronics STM32 Nucleo-64 MCU Development Board NUCLEO-L452RE,” RS Components, [Online]. Available: https://uk.rs-online.com/web/p/microcontroller-development-tools/1261775. [Accessed 23 April 2024].

[42] Espressif, “ESP32-S3-DevKitC-1 v1.1,” Espressif, [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/latest/esp32s3/hw-reference/esp32s3/user-guide-devkitc-1.html. [Accessed 23 April 2024].

[43] Arduino, “Digital Pins,” Arduino, 25 July 2023. [Online]. Available: https://docs.arduino.cc/learn/microcontrollers/digital-pins/. [Accessed 26 April 2024].

[44] Traffi, “TG3220 Datasheet,” Traffi, [online], 2024.

[45] D. Duddy, “16 Different Types of Hand Tools and Their Uses,” Protrade, 6 August 2022. [Online]. Available: https://www.protrade.co.uk/blog/16-different-types-of-hand-tools-and-their-uses/. [Accessed 7 May 2024].

[46] J. Brownlee, “Sequence Classification with LSTM Recurrent Neural Networks in Python with Keras,” Machine Learning Mastery, [online], 2022.

[47] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot and E. Duchesnay, “Scikit-Learn: Machine Learning in Python,” *Journal of Machine Learning Research,* vol. 12, no. n/a, pp. 2825-2830, 2011.

[48] S. Jaiswal, “What is Normalization in Machine Learning? A Comprehensive Guide to Data Rescaling,” Radar, January 2024. [Online]. Available:

https://www.datacamp.com/tutorial/normalization-in-machine-learning. [Accessed 5 May 2024].

[49] M. e. a. Fiali, "Sensor based (American) Sign Language Recognition," Kaggle, [online], 2024.

[50] C. Conti and A. S. Varde, "Robot Action Planning by Commonsense Knowledge in Human-Robot Collaborative Tasks," in *IEEE IEMTRONICS 2020*, Montclair, 2020.

[51] C. Zhen, Interviewee, *Cobotic Sensors Challenge.* [Interview]. August 2023.

[52] W. P. Neumann, S. Winkelhaus, E. H. Grosse and C. H. Glock, "Industry 4.0 and the human factor – A systems framework and analysis methodology for successful development," *International Journal of Production Economics*, vol. 233, no. n/a, p. n/a, 2021.

[53] S. Saxena, "What is LSTM? Introduction to Long Short-term Memory," Analytics Vidhya, [online], 2024.

12 APPENDICES

12.1 APPENDIX A: PROJECT GITHUB

Python and Arduino programs, scripts created to make flowcharts in code2flow, and gesture datasets can be found in the MiGlove GitHub: https://github.com/jmh91/MiGlove.

12.2 APPENDIX B: FINAL COSTING OF PROJECT

Component	Cost per unit (inc. VAT)	Amount required	Total cost	Source	Stock No.	Other info
Raspberry Pi 3 B+	£32.78	1	£32.78	rapidonline.	75-1005	Limited stock across all sites
Arduino Mega board	£35.99	1	£35.99	RS	715-4084	In stock
Adafruit MPU-6050 6-DoF Accel and Gyro Sensor	£12.90	3	£38.70	Pimoroni	ADA3886	Various available sensors around price.
Flex sensors 2.2"	£11.75	1	£11.75	Robot Shop	SEN-10264	
Flex sensors 2.2"	£8.34	7	£58.38	Mouser	744-FS-L-0055-253-ST	
Velcro hook and loop tape	£15.30	1	£15.30	RS	423-9555	
Glove	£4.78	1	£4.78	RS	256-5191	Exposed thumb, 2nd and 3rd fingertips for improved dexterity
Misc. electronics and connectors	(Provided)	-	-	University of Bath	-	Available from lab
Breadboard jumper wires - male to female	£1.19	2	£2.38	RS	204-8242	
Breadboard jumper wires - male to male	£2.55	2	£5.10	RS	204-8241	
Amphenol FCI Clincher Connectors (male)	£1.67	5	£8.35	Robosavvy	S-COM-14195	
Flexible Qwiic cable to female jumper pins	£1.50	1	£1.50	Sparkfun	CAB-17261	
Estimated total:			£215.01			