Imperial College London
Department of Electrical and Electronic Engineering

WHEELS Technical Report

Group 6

Supervisor: Prof. Adria Junyent-Ferre

Jamie-lee Thompson	01187862
Rohan Tangri	01198239
Zheyuan Li	01181358
Shiwei Liu	01192758
Zengrui Huang	01204572
Kosidinna Umeigbo	01201479

Table of Contents

Abstract

This report aims to summarize the engineering principles involved in creating the WHEELS project; an open source controller box designed to be taken and fitted onto any existing wheelchair. Overall, there are three main themes to the detailed design. The first details the control algorithm design, the second concentrates on the Android app and the final section looks into the design of an example powered wheelchair to demonstrate the control software. Furthermore, the paper includes details on the design process, team management and marketing as well as testing and simulation results. This project has been undertaken as part of a contribution to the charity ReMap Bristol in order to help people motorize their manual wheelchairs at an affordable price; and as such, all resources referenced to will be available in public repositories to be easily accessed and modified.

1. Introduction

In the UK alone, 1.2 million people use a wheelchair. However, 26% of wheelchair users were denied funding for a powered wheelchair. This is a problem since powered wheelchairs can be wildly expensive, generally ranging from £1,500 to £15,000 and even higher. For this reason, only 50% of powered wheelchair users feel their purchase is good value for money.

It is clear that many would prefer to use a powered wheelchair over a manual one but are limited by their finances. Due to this, these people miss out on the many advantages a powered wheelchair could provide, potentially dramatically increasing quality of life. For example, a manual wheelchair requires significant upper body strength to manoeuvre, especially if travelling outdoors for long periods of time or in hilly terrain. A motorized wheelchair would allow the user to enjoy a more comfortable ride since no significant work is required from the user to accelerate themselves. This also allows for shorter journey times as well as a smoother ride since there is a constant torque being applied to the wheels instead of infrequent pushes.

2. Brief

2.1 Client brief

The purpose of this project is to develop an open source controller box for electric wheelchairs. This should be able to read input from the user (e.g. via joystick) and generate appropriate control signals for the motor drivers to produce the desired motion. This project should form a basis for a low cost open source design that can be expanded and reused to meet different user requirements around the world.

2.2 Specifications

The control box to be designed must meet the following criteria in order to satisfy the client brief:

1. The code created must be easy to understand, hack and modify.
2. There must be acceleration and jerk limiter software in place, both for safety and comfort.
3. The controller software should be able to adjust the input joystick variables (e.g. non-linearity, sensitivity and dead zone area).
4. Low cost of controller design is essential.

Additional features to be included are:

1. An Electronic Auto-Braking System (EABS) should be included.
2. A secure Android App should be developed as a potential input device as well as for easily getting diagnostics data from the wheelchair.
3. Cruise control functionality.

3. Market analysis

In 2018, the global wheelchair industry was valued at $6.8 billion with an average growth rate of 2.5% per annum.

This project will mainly target the manual wheelchair market, since already powered wheelchairs will generally have little need for the controller software.

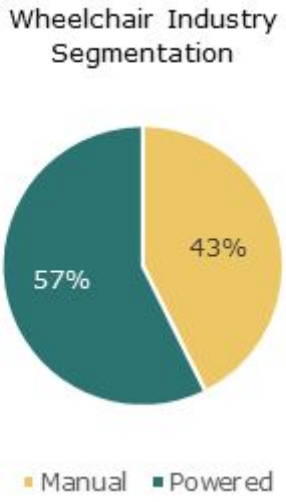


Figure 3.1: Wheelchair Industry Segmentation

As shown in Figure 3.1, 43% of the wheelchair market is made up of manual wheelchairs. Therefore, the approximate global value of the manual wheelchair industry in 2018 was $2.92 billion. Following on from this, the predicted five year manual wheelchair market size from 2019 can be extrapolated assuming the same growth rate as the total industry:

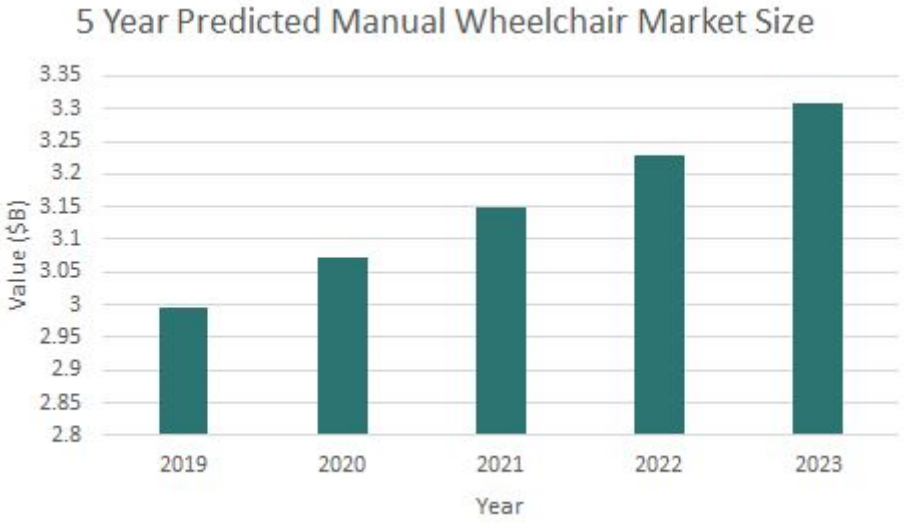


Figure 3.2: 5 year predicted global manual wheelchair market size model from 2019.

4. High level design

The client brief specifies designing of the control algorithm for an electric wheelchair. There is therefore a focus in this project on the design of the algorithms and any inputs and outputs needed. In order to demonstrate the design however there is also a need to build a prototype / modify an existing wheelchair and so the demonstration design will also be documented in this document.

4.1 Client design.

In order to meet the specifications at the start of this document, the control algorithm will contain a way to interface from both mechanical joystick and joystick on an application. The joystick input will be mapped to an output that would control some motor drivers, including modes for normal control and cruise control and containing the essential jerk limiting design. The app should contain all the necessary buttons/ controls to completely replace any mechanical input to the controller algorithm if needed whilst also providing a diagnostic page to display information from the wheelchair, that can easily be edited by future engineers working on this project.

The control algorithm will be based on an arduino.

4.1.1 Arduino design

The mechanism of controlling a two-wheel driving vehicle is straightforward and intuitive, and it is better to start thinking of the kinematic behaviour before deducing the kinematic formulae.

Figure 4.1.1A: Dash Express Wheelchair

Figure 4.1.1A shows the wheelchair that is used for making the prototype, which represents an ordinary wheelchair with essential functionalities. The mechanically easiest way to steer this wheelchair is motorizing the two back wheels and keeping the front universal wheels unchanged.

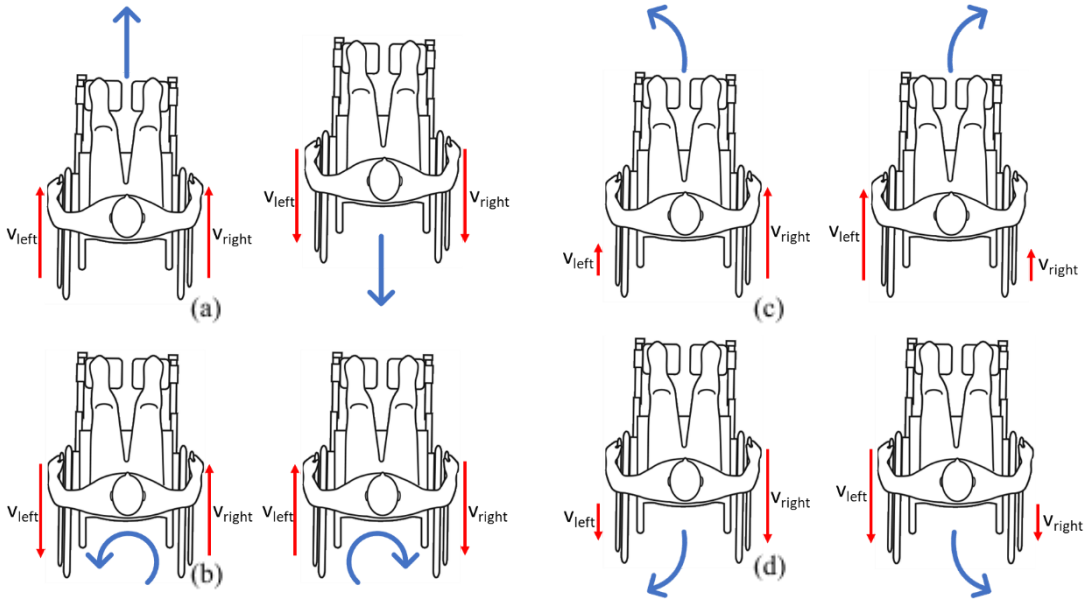


Figure 4.1.1B: Top-down view visualizations of kinematic behaviour.
red arrows = tangential velocity of wheels, blue arrows = displacement prediction of wheelchair

To move forwards/backwards, both wheels spin at the same speed in the same direction as shown in Figure 4.1.1B (a); To rotate with zero linear displacement, both wheels spin at the same speed but in the opposite direction as shown in Figure 4.1.1B (b); To turn left/right while moving forwards/backwards, the wheels spin at different speeds as shown in Figure 4.1.1B (c) and (d).

From Figure 4.1.1B it can be concluded that this control mechanism enables a wheelchair to move freely in two axes. It can either move in forward/backward direction or rotate in the vertical axis. The wheels are assumed to have contact with the ground all the time and not slip, i.e. no sideways drifting.

However, there is no direct relationship between the tangential velocity of two wheels and these two axes of motions. Thus, a conversion is needed:

$$v_{sync} = \frac{v_{left}+v_{right}}{2}$$

$$v_{diff} = \frac{v_{left}-v_{right}}{2}$$

Formula 4.1.1A

where v_{left} is the tangential velocity of left wheel and v_{right} is the tangential velocity of right wheel. v_{sync} is the **synchronized velocity** which represents the linear velocity in forward/backward direction, and it is simply the average of the tangential velocity of two wheels. v_{diff} is the **differential velocity** which is half of the difference in velocities between two wheels. The v_{diff} is also the difference between left/right wheel velocity and synchronized velocity, which could also be expressed as:

$$v_{diff} = v_{left} - v_{sync} = v_{sync} - v_{right} \ .$$

From formula (4.1.1A), we can easily derive the linear velocity and angular velocity of the wheelchair for any moment:

$$v = v_{sync} = v_s$$

$$\omega = \frac{v_{diff}}{d} = \frac{v_d}{d}$$

Formula 4.1.1B

where v is the linear velocity (in m/s), ω is the angular velocity (in rad/s) and d is the distance (in m) between two wheels.

From formula (4.1.1A), we can also express v_{left} and v_{right} by v_{sync} and v_{diff}:

$$v_{left} = v_{sync} + v_{diff}$$

$$v_{right} = v_{sync} - v_{diff}$$

Formula 4.1.1C

The example in Figure 4.1.1C helps understand the synchronized and differential velocities conversion. The tangential velocities of left and right wheel are 1.3m/s and 0.7m/s respectively. Using the conversion formula (4.1.1A), the synchronized velocity is 1m/s, so the wheelchair is now moving forward at 1m/s which is the linear velocity of the wheelchair; the differential velocity is 0.3m/s, assuming the distance between two wheels is 0.6m, the angular velocity would be 0.5rad/s.

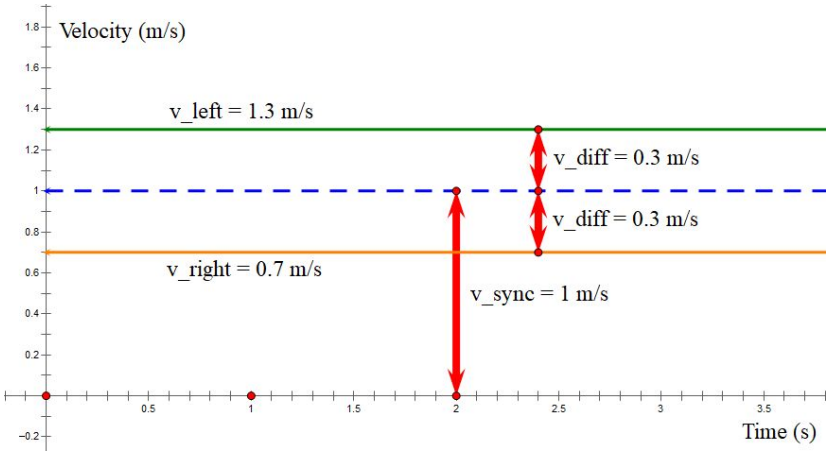


Figure 4.1.1C: Synchronized and differential velocities

It is possible to map these two axes of motion, linear and angular velocities, into the 2 axes of joystick inputs. Joystick y-axis controls the synchronized velocity or linear velocity, x-axis controls the differential velocity or angular velocity. Figure 4.1.1D visualizes the relationship between the joystick input and the expected movements of the wheelchair. When moving the joystick forwards/backwards, the wheelchair should move forwards/backwards accordingly. When moving the joystick left/right, the wheelchair should rotate anti-clockwise/ clockwise with zero linear velocity. Figure 4.1.1E shows how the wheelchair would behave if the joystick is pushed forward and left/right at the same time. It would move forwards while

turning left/right by differentiating the velocities of left and right wheels and vice versa for moving backwards.

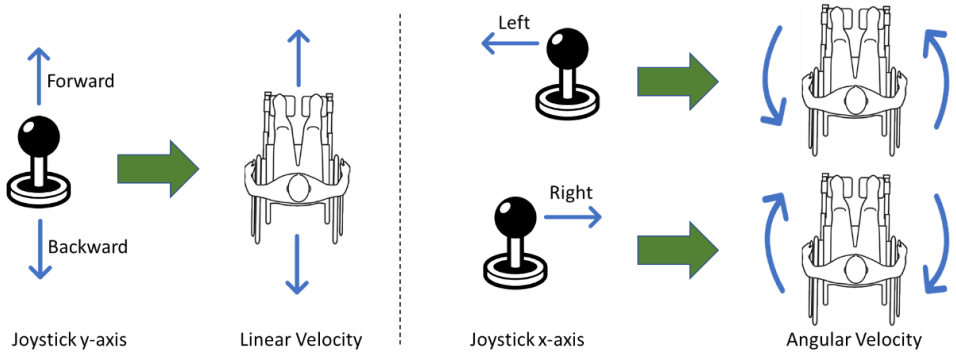


Figure 4.1.1D: Kinematic behaviour for each axis of joystick inputs

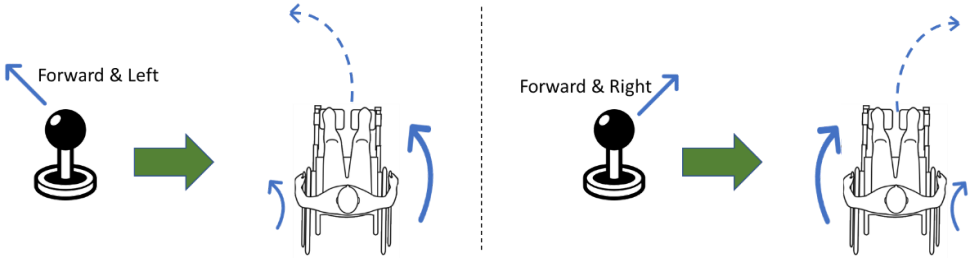


Figure 4.1.1E: Kinematic behaviour for two axes of joystick inputs

In reality, it is very difficult to use joystick to control velocity directly as it requires speed sensor and feedback loops. Thus, like other modern vehicles, the outputs of the program in the microcontroller should be the output power from the motors instead of velocities of wheels. Therefore, in the control program, the variable "ttl" (throttle) is used instead of velocity to avoid misunderstanding.

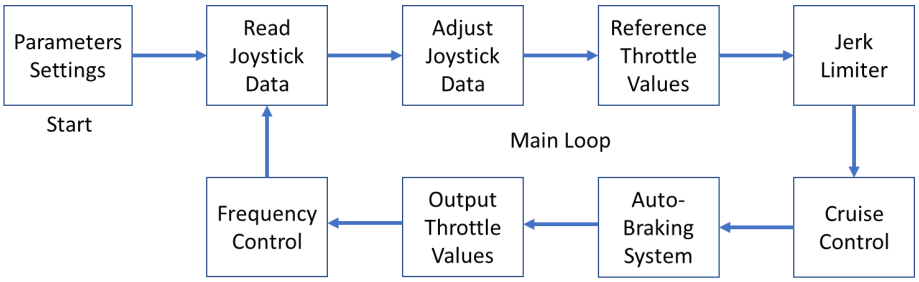


Figure 4.1.1F: Flow chart of the control program

The goal of this control program is to use some algorithms to convert the joystick data into output power/throttle values (0% ~ 100%) for left and right motors with some additional features. Figure 4.1.1F shows the main procedures in the control program.

4.1.2 App design

The Android App works using a Bluetooth LE connection service to communicate with the arduino on the wheelchair.

To make this work, the following components were needed:

1. An Android phone with a Bluetooth adapter.
2. A Bluetooth LE module able to connect to the arduino.

The app is coded in Java using the Android Studio IDE. The following diagram illustrates how the different classes involved interact with each other:

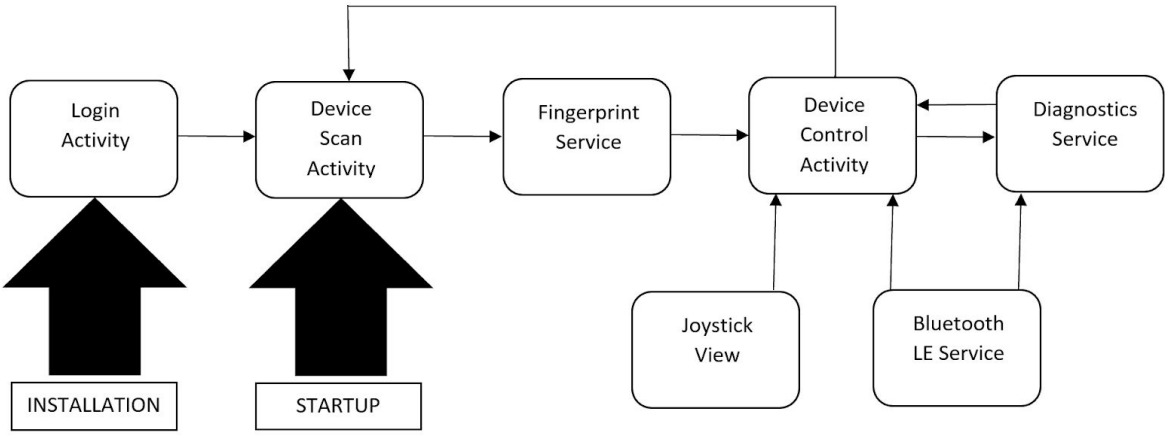


Figure 4.1.2A: App HLD

For more detail, see the detailed design in section 6.2.

As an overview; first, the phone used must scan the surrounding area for nearby bluetooth devices. For this to work, the external device must have a GATT server which broadcasts it's services. Once the relevant device for the wheelchair has been detected, the app must retrieve all the potential services that the external device can provide; each identified by a Universally Unique Identifier (UUID). Each service also has a multitude of characteristics, again being identified with separate UUIDs. Luckily, for both reading and writing data, the same service is used so only one service UUID is needed; however, within this, two characteristic UUIDs are required. Each characteristic can be thought of as an address containing data, whose value can be described by a descripto
r. To obtain this data, the app simply retrieves the information stored on the characteristic sent.c; and to send data, it simply overwrites the characteristic address with the new information to b

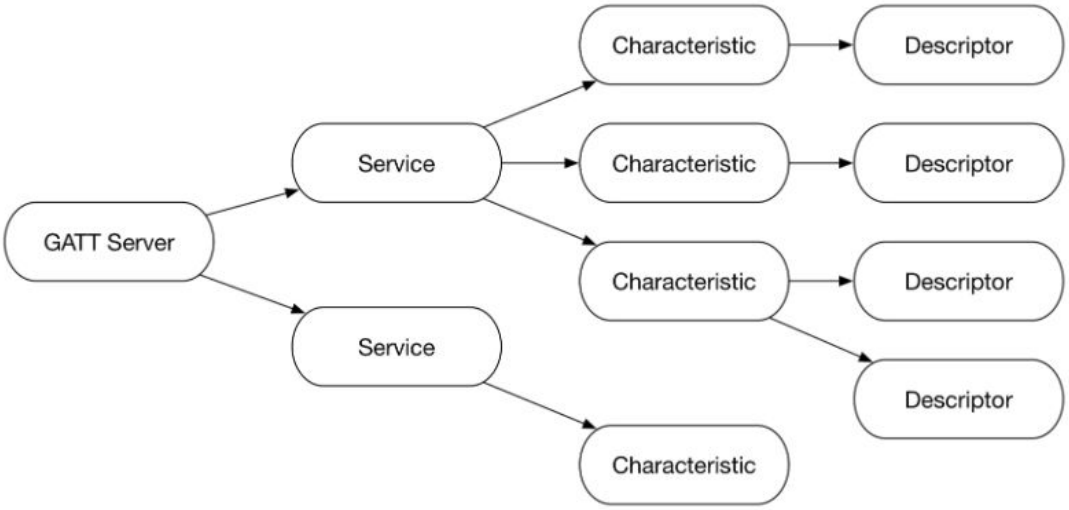


Figure 4.1.2B: Bluetooth LE structure

4.2 Demonstration design

There main modules of the demonstration design will be:

- Control box.
- App (INPUT and OUTPUT)
- Mechanical inputs (joystick and buttons)
- Motor drivers
- Motors.
- Power.
- LED outputs.

The motors are mounted to an existing non electrical wheelchair and they directly drive the wheels. This minimises the need for any mechanical design in this project, however this is purely for the demonstration and it is not advised to do this when modifying a wheelchair for commercial use.

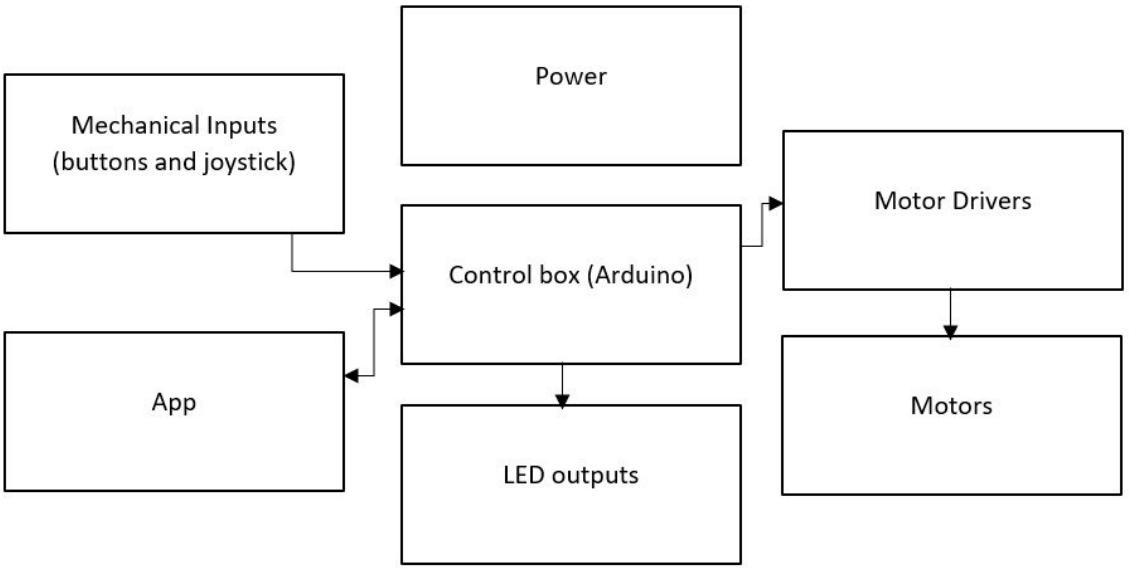


Figure 4.2A: Demonstration HLD

POWER

A large long lasting battery is used to power the motors and the arduino. The motors are rated at 12V and the arduino can also take an input of 12V. Power for the app will be supplied from whatever phone is running the app.

CONTROL BOX

See section 4.1.1 for more information on this design. The control box is an arduino, it sends control signals to the motor drivers via PWM. It is also connected to a bluetooth low energy module which connects to the app. The control box receives joystick and other input signals from the app and also sends out other information (diagnostic signals). The control box is also connected to the mechanical inputs, in case the phone app isn't available.

The control box software has been modified to also include outputs to LEDS to show when cruise control, EABS, jerk limiter and “brake” have been enabled.

APP

See section 4.1.2 for more information on this design. The app connects to the arduino control software via the phone’s bluetooth device and sends control information from the user (joystick position and button clicked). It also receives and displays the diagnostic information from the control box.

MECHANICAL INPUTS

The mechanical inputs provide the ability to control the wheelchair when the app is not available or cannot be used for some reason. A potentiometer joystick is used with x and y axis and so and up/down signal and left/right signal of 0-5Vs is sent to the arduino.
There are also 3 buttons provided
-to enable/disable cruise control
-to enable/disable the jerk limiter
-to enable/disable a “brake”

LED OUTPUTS

There are 4 LEDS in the demonstration design. They are to demonstrate and show the audience when cruise control, EABS, jerk limiter and “brake” have been enabled

MOTOR DRIVERS

The motor drivers take the output control signal from the arduino and map it to a power the motors at the correct voltage.
The motor drivers require an input voltage of 12V and they take a PWM control signal. Another input is also utilised to determine the direction of wheel spin.
There is one motor driver for each motor.
The motor drivers already contain H bridge circuits and so allow the motors to turn either way depending on the control signals.

MOTORS

The motors used are two input 60W 12V DC permanent magnet motors. They will drive the wheelchair.

5. Process

5.1 Concept generation

To begin generating concepts it was important to start with the key features that were important to the client. To fulfil the requirements whatever was created had to be cheap and open source, meaning hardware would have to be easily sourced, and any code easy to understand and modify.

5.1.1 Controller

DEVICE

Two suitable device were looked at for deciding where to implement the controller code, the first being an arduino and the second being a raspberry pi.

A decision matrix was created to choose the best device.

Device	Arduino	Raspberry Pi
Cost (cheap)	+	-
Battery Support	+	-
Complexity	+	-
Inbuilt ADC	+	-
Connectivity	-	+
Language Support	-	+
In built PWM	+	-

From the decision matrix, we chose to use an arduino
- it is cheaper
- it is more easily connected to batteries which is ideal for this project as a wheelchair cannot be connected to mains
- it has inbuilt ADCs which is more desirable as analogue signals will be inputted from the mechanical joystick

However the downsides of the Arduino are that it has limited language support and would require external hardware in order to connect to bluetooth.

The inbuilt PWM of the arduino was very desirable at the end of the project for the demonstration as we initially intended to use a CAN bus modifier which would be available for both devices, but we

had to use PWM due to administrative issues of the original motor driver not being available, and so a last minute motor driver change.

ALGORITHMS

Two concepts were generated for the control algorithm of the wheelchair. The first concept was supplied by Adria Junyent Ferre and is available with this document. The second concept was generated as part of this work and is explained in detail in section 6.

The main difference between the two is what happens to the wheelchair when there is 0 forward component from the joystick, Concept 1 can begin to move in either forward or backwards direction, which the user might not want, whereas concept two will turn on its axis by driving each wheel in opposite directions

To decide between the two a decision matrix was created.

Algorithms	Concept 1	Concept 2
Ease of understanding*	-	+
Effect at 0 forward input component.	-	+
Requiring work.	+	-

* *entirely subjective*.

Concept 2 was chosen as it was found to be more intuitive and easier for the open source nature of the project to be modified afterwards. The effect at 0 forward input also meant the wheelchair would rotate on its axis in concept 2 which was also thought to be more desirable. The down side of concept 2 is simulations would need to be done to prove that the algorithm would behave as desired before moving forward.

5.1.2 App

The app was created in order to provide a greater scope for the future by allowing future developers to add their own features. It began with the purpose of providing a screen with which the user could interface with the wheelchair in a smarter way.

The first development on the app was the joystick and buttons which was initially intended to replace and eradicate the mechanical inputs. However the capability for both the mechanical and the app input remains as it was realised that some users may not have the dexterity to control an app and for safety if the bluetooth from the app was to disconnect, a mechanical control would still be required. It is also important that as this is an open source project, both input types should be in the code as the choice of which or both to use should be left to the developers accessing the code in order to demonstrate the ability to do both.

The second development on the app, but the most important for the future, is the ability of the app to display information from the wheelchair. This is displayed on the diagnostic page of the app. More can be read on this in the future work section.

5.1.2 Demonstration hardware

The demonstration hardware design concept generation began with deciding how best to demonstrate our design.

The demonstration design would need to be capable of demonstrating the controller code, "wheelchair" movement and bluetooth app connectivity.

Several concepts were generated:

1. Create a small scale two wheeled buggy style robot.
2. Create a full sized wheelchair.
3. Drive wheels and measure the speed of each wheel to simulate wheelchair movement in real time on a computer.

There were several advantages and disadvantages of the concepts, which are discussed below:

Concept 1.

Advantages

This concept would require a very small amount of design, having already previously all designed small buggys in our first year projects.

This would be cheap. Low cost of batteries, motors and motor drivers.

Disadvantages

This concept would be hard to demonstrate the jerk limiting software on, with the difference not easy to see/ feel.

Does not necessarily prove the software will be as smooth or work on an actual wheelchair project.

Concept 2.

Advantages

Able to demonstrate full controller capability.

Proof the controller code would work on a real wheelchair project.

Disadvantages

Expensive motors and motor controllers would be needed. Other expenses of buying and modifying an existing wheelchair.

Some mechanical knowledge needed to mount motors.

Concept 3.

Advantages

No need to design full wheelchair.

Cheaper, less powerful motors required.

Disadvantages

Simulating wheelchair movement from the real time speed of wheels may be slow.

Software needs to be written.

Less demonstrability.

Having considered these three concepts and weighing up the advantages and disadvantages, it was decided to design a full scale wheelchair system as this would be the best for demonstrating our work. The cost of this system would be a problem but a pair of motors were gifted, so the use of these in the demonstration would not be a cost.

Wheelchair concept

The next stage in this design was to decide how to modify the existing an existing wheelchair.

An initial idea was to have a retrofittable wheel and pulley system, with gears so that the wheelchair could be driven manually as well. After consultation with the mechanical engineering department it was deemed that this idea was too complex for our time frame and also not completely necessary as the brief was to design and demonstrate the controller code, not create a retrofittable kit. This concept however has been moved to the future work section as it can be developed in the future.

It was then decided, after consultation with the mechanical specialist Phil Jones in the EEE department that it would be possible to mount motors to an existing wheelchair and directly drive the wheels.

This was chosen as it was ideal for demonstrating the work.

The official client brief said to dismiss battery management systems and other parts that would be necessary in a real electric wheelchair and to concentrate on controller code. It was for this reason it was decided to only build with parts absolutely necessary for the demonstration and not for commercial use.

VESC PLAN

To drive the motors, a VESC type motor controller was originally discussed with the supervisor. The team originally began planning to use this type of motor controller and interface between these and the arduinos using a CAN bus, so that information could be sent back and forth between the controller and drivers. However these motor controllers never arrived and due to time constraints, basic motor controllers had to be used.

MOTORS

The power of the motors needed to be able to drive a wheelchair was calculated and will be discussed in section 6 of detailed design. However, as mentioned before, some motors were gifted to us to reduce the cost of the demonstration. These motors were not as powerful as first planned but suitable enough to drive the wheelchair without the added weight of a person.

6. Detailed design

6.1 Control algorithm

The control algorithm consists of all the arduino code used to control the wheelchair movement and added features.

6.1.1 Read Joystick Input

Different platforms may have different ways to read the joystick data, and the results may vary. A joystick value in the range from -1 to 1 for both axes is needed in order to proceed in the control program. If using MATLAB in PC to read joystick data (Code 6.1.1A), the joystick values are already in this range. If using analogue input pins on an Arduino (Code 6.1.1B), the joystick values would be and integer from 0 to 1023, so a conversion is needed into the range -1 ~ 1.

```
% create a joystick handle
ID = 1;
joystick=vrjoystick(ID);
…
% read joystick input
jx = x_inverted*axis(joystick, x_channel);  % joystick x-axis
jy = y_inverted*axis(joystick, y_channel);  % joystick y-axis
but1 = button(joystick, button1);       % button 1
but2 = button(joystick, button2);       % button 2
but3 = button(joystick, button3);       % button 3
```

Code 6.1.1A: MATLAB code - Read Joystick Input

```
#define x_channel A1        // input analogue pin for reading joystick input
#define y_channel A0        // input analogue pin for reading joystick input
#define button1 A2          // button 1 pin number (Cruise Control)
#define button2 A3          // button 2 pin number (EABS)
#define button3 A4          // button 3 pin number (Jerk Limiter)
…
// Read joystick data
jx = analogRead(x_channel);
jy = analogRead(y_channel);

// read button values, use analogue input pins for buttons to save digital pins
but1 = analogRead(button1);
but2 = analogRead(button2);
but3 = analogRead(button3);

// convert jx and jy into the range -1~1
jx = x_inverted*(jx-511)/511;
jy = y_inverted*(jy-511)/511;

// convert analog value into digital value
but1 = (but1 > 511) ? 1 : 0;
but2 = (but2 > 511) ? 1 : 0;
but3 = (but3 > 511) ? 1 : 0;
```

Code 6.1.1B: Arduino code - Read Joystick Input

6.1.2 Joystick Adjustment

To maximize the flexibility of the control program, joystick adjustment is necessary. There are 4 adjustments in total: inversion, deadzone, linearity and sensitivity. The inversion setting is either 1 or -1, which is multiplied directly by the joystick value (Code 6.1.1A and Code6.1.1B) and enables users to invert any one of the two joystick axes.

The dead zone parameter is in the range 0 ~ 1. Due to the inaccuracy of joystick manufacturing process, the reading of the joystick is not exact zero when the joystick returns to central position (when no external force is acting on it). Taking the joystick y-axis as an example, the deadzone setting ignores the joystick input if the raw data from joystick is within the specific range by using an if statement and a formula:

$$jy_{adj}^{ddz} = \begin{cases} 0 & if \quad abs(jy_{raw}) < y_{deadzone} \\ \frac{|jy_{raw}| - y_{deadzone}}{1 - y_{deadzone}} & otherwise \end{cases}$$

Formula 6.1.2A

where jy_{raw} is the raw joystick y-axis data, jy_{adj}^{ddz} is the partially adjusted joystick data (after deadzone compensation). The jy_{adj}^{ddz} is now only positive because the absolute value for jy_{raw} is taken, but the sign for jy_{raw} will be retrieved later. For example, if the deadzone is set to be 0.2, the joystick data is zero when the reading is from -0.2 to 0.2 as illustrated in Figure 6.1.2A.

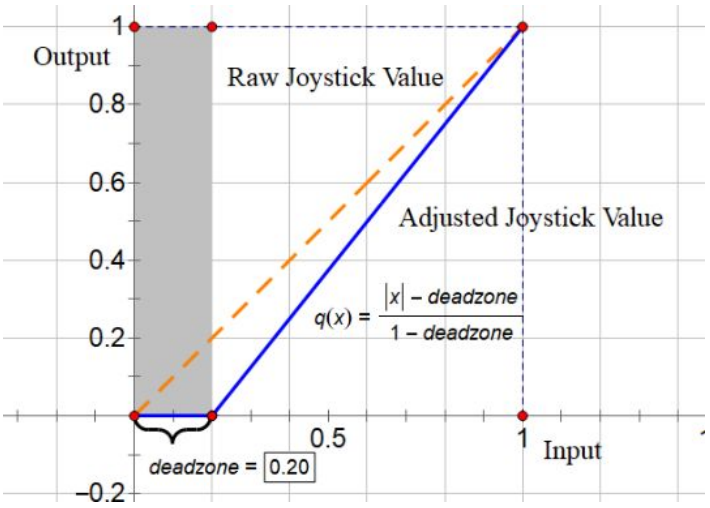


Figure 6.1.2A: Deadzone

Linearity should be both a positive number and have a default value of 1, meaning that it is linear. Linearity is the exponent of the joystick value after compensating for the dead zone. A linearity larger than 1 would let the joystick be more sensitive in the region near the maximum absolute value than near the central position and vice versa for a linearity less than 1.

Sensitivity should also be a positive number with a default value of 1. Sensitivity can be treated as the overall gain which determines how far the joystick needs to travel to give a 100% joystick value. A sensitivity larger than 1 would give a 100% joystick value before the joystick reaches maximum position while a sensitivity smaller than 1 would never give 100% value even the joystick reaches the maximum position. Then the partially adjusted joystick value $\widehat{jy}_{adj}$ (after linearity and sensitivity calculation) can be expressed as:

$$\widehat{jy}_{adj} = sensitivity \times sgn(jy_{adj}^{ddz}) \times jy_{adj}^{ddz\ linearity}$$

Formula 6.1.2B

where *sgn*() is the sign/signum function which extracts the sign of a value. Thus, the sign of the raw joystick input can be retrieved now which makes this adjustment work for negative joystick inputs as well.

The final step for the joystick adjustment is capping the joystick because if the sensitivity is larger than 1 (Figure 8.1.9 left), the $\widehat{jy}_{adj}$ may be greater than 1 which is an invalid result. We can use maximum and minimum operators to make sure the adjusted joystick value jy_{adj} is definitely in the range -1 ~ 1:

$$jy_{adj} = max(min(\widehat{jy}_{adj}, 1), -1)$$

Formula 6.1.2C

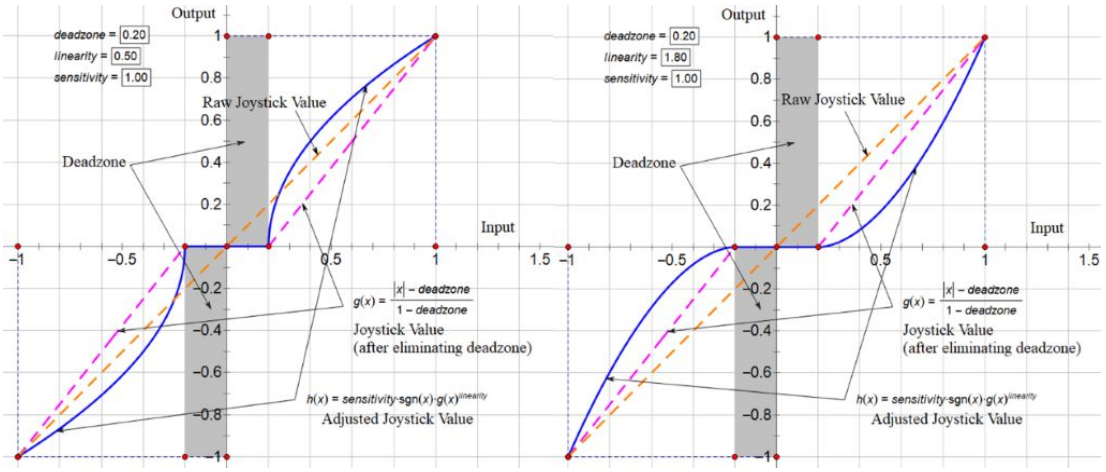


Figure 6.1.2B: Joystick characteristic curve for linearity = 1.8 (left) and 0.5 (right), deadzone = 0.2, sensitivity = 1

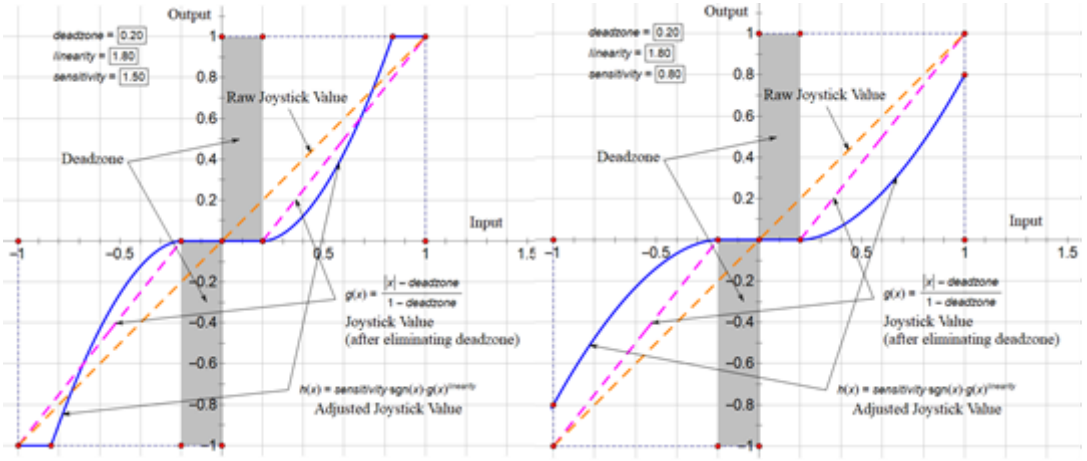


Figure 6.1.2C: Joystick characteristic curve for linearity = 1.8, deadzone = 0.2, sensitivity = 1.5 (left) and 0.8 (right)

Figure 6.1.2B and Figure 6.1.2C compare the joystick characteristic curves under different linearity and sensitivity parameter settings.
Directly translate the joystick adjustment formulae into MATLAB code for y-axis as an example:

```
if abs(jy) < y_deadzone
       jy_adj = 0;     % ignore the joystick input
else    % y-axis adjustments
    jy_adj = (abs(jy)-y_deadzone)/(1-y_deadzone);
    jy_adj = y_sensitivity*jy_sign*jy_adj^y_linearity;
       jy_adj = max(min(jy_adj, 1), -1);   % limit jy value within -1~1
end
```

Code 6.1.2A: MATLAB code - Joystick Adjustment

And the same algorithm is used to calculate adjusted x-axis joystick value jx_{adj}.

6.1.3 Reference Throttle Values

Firstly, some maximum values need to be calculated. *ttlmax* is the maximum throttle value. It limits the maximum possible throttle value for the left and right motors. It could be less than 1 when the motor is too powerful for a wheelchair. *ttldratio* is the ratio of differential throttle to *ttlmax*. Using this, we can calculate the maximum synchronized ($ttls_{max}$) throttle and maximum differential throttle ($ttld_{max}$):

$$ttls_{max} = ttl_{max} \times (1 - ttld_{ratio})$$
$$ttld_{max} = ttl_{max} \times ttld_{ratio}$$

Formula 6.1.3A

Then it can be ensured that:

$$ttls_{max} + ttld_{max} = ttl_{max} = ttl^{left}_{max} = ttl^{right}_{max}$$

Formula 6.1.3B

Now there are fully adjusted joystick values which are in the range -1 ~ 1, so formula (4.1.1C) can be followed to calculate the reference left and right motor power/throttle values (ttl^{left}_{ref} and ttl^{right}_{ref}) by multiplying them with $ttls_{max}$ and $ttld_{max}$:

$$ttl^{left}_{ref} = jy_{adj} \times ttls_{max} + jx_{adj} \times ttld_{max}$$
$$ttl^{right}_{ref} = jy_{adj} \times ttls_{max} - jx_{adj} \times ttld_{max}$$

Formula 6.1.3C

Convert these to reference synchronized throttle (*ttlsref*) and reference differential throttle (*ttldref*) using the following equation (6.1.2D):

$$ttls_{ref} = \frac{ttl^{left}_{ref} + ttl^{right}_{ref}}{2}$$
$$ttld_{ref} = \frac{ttl^{left}_{ref} - ttl^{right}_{ref}}{2}$$

Formula 6.1.2D

From the equations above, it can be seen that *ttlsmax* must be smaller than *ttlmax*. In other words, if the wheelchair is moving straight forward at maximum synchronized power ($ttls_{max}$), the left and right motors are not running at full power (ttl^{left}_{max} or ttl^{right}_{max}) and they are actually at 70% maximum power if $dttl_{ratio}$ is 0.3.

6.1.4 Jerk Limiter

The **jerk (j)** is the derivative of acceleration and the second derivative of velocity. It is not obvious from the third-person perspective, but it is directly related to comfort for the driver. In a linear motion, the jerk can be described mathematically as:

$$jerk = \frac{da}{dt} = \frac{d^2v}{dt^2} = \frac{d^3s}{dt^3}$$

Formula 6.1.4A

where a is the acceleration, v is the velocity, s is the displacement and t is time. In the control program, the dt is not infinitesimal and it is the time consumed for each loop.

The synchronized throttle value ($ttls_{ref}$) we used is assumed to be the percentage of maximum driving force (F/F_{max}) acting on the wheels and the driving force (F) is directly proportional to the acceleration (a):

$$ttls_{ref} = \frac{F}{F_{max}}$$
$$F = ma$$
$$\Longrightarrow a = \frac{F_{max}}{m} \times ttls_{ref}$$

Formula 6.1.4B

where m is the mass of the wheelchair. The ratio F_{max}/m is set by parameter "accel_gain" in the parameter settings section in the MATLAB code. Formula 6.1.2B is used to convert throttle values to acceleration which is important for simulation (Section 6.1.8). The jerk would be:

$$linear\ jerk = \frac{da}{dt} = \frac{F_{max}}{m} \times \frac{d(ttls_{ref})}{dt}$$

Formula 6.1.4C

From equation (6.1.4B), the acceleration is limited by the maximum synchronized throttle value $ttls_{max}$. From equation (6.1.4C), it can be seen that the linear jerk is controlled by the rate of change in synchronized throttle value $ttls_{ref}$. Both $ttls_{ref}$ and $ttld_{ref}$ calculated in Section 6.1.3 are the instantaneous translations for the current joystick values without any jerk limiting or filtering. Therefore, taking the linear motion (controlled by joystick y-axis) as an example, we need to limit the rate of change in $ttls_{ref}$ by its maximum value $dttls_{max}$ as shown in Figure 6.1.4A:

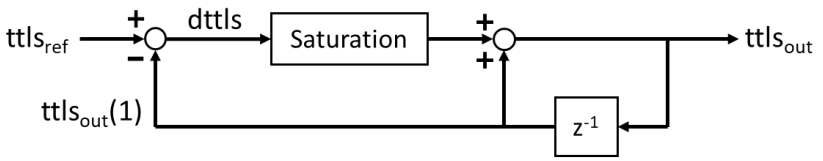


Figure 6.1.4A: Jerk Limiter flow chart

Translate this idea into MATLAB code:

```
dttls_max = js_limit*dt;    % maximum rate of change in synchronized throttle
...
dttls = ttls-ttls_out;      % change in sync throttle
dttls = max(min(dttls, dttls_max), -dttls_max);     % saturation
ttls_out = ttls_out+dttls;  % output sync throttle
```

Code 6.1.4A: MATLAB code jerk limiter

The variable $dttls_{max}$ is controlled by a parameter "js_limit" in the parameter setting section. The js_limit is the maximum rate of change in output synchronized throttle value $ttls_{out}$ per second. The smaller the js_limit value is, the slower the wheelchair accelerates and decelerates and vice versa.

The same algorithm is used to calculate output differential throttle value $ttld_{out}$ which limits the angular jerk (joystick x-axis). The Jerk limiter is on as default and can be turned off by pressing the "Jerk Limiter" button in the control interface shown in Figure 6.1.6A.

6.1.5 Cruise Control

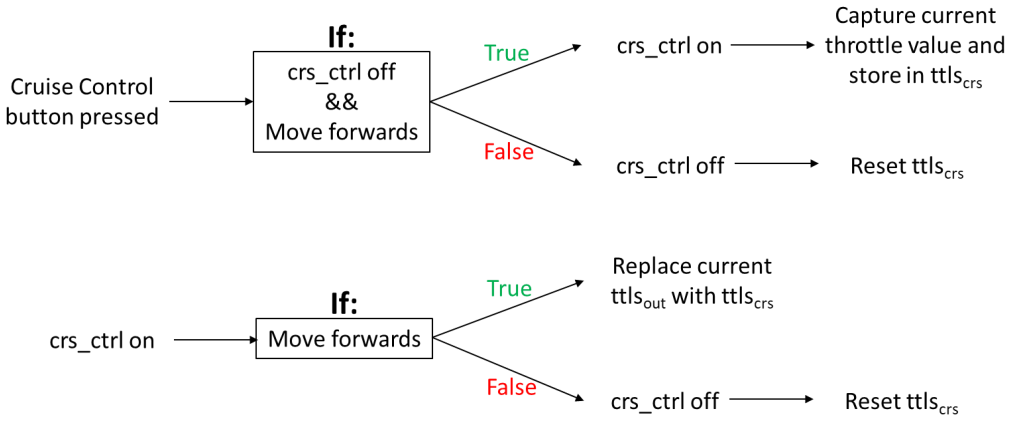


Figure 6.1.5A: Cruise Control flow chart

Figure 6.1.5A illustrates the procedures in the Cruise Control (crs_ctrl) system. The Cruise Control system can be activated by pressing the "Cruise Control" button.

If the cruise control is previously deactivated, when the wheelchair is moving forward at a certain speed and the Cruise Control button is pressed, the cruise control is activated and the program stores current synchronized throttle value ($ttls_{out}$) in variable $ttls_{crs}$. From that moment, the output throttle value $ttls_{out}$ will be replaced by the stored throttle value $ttls_{crs}$ every time until the cruise control is deactivated again. The cruise control can only be activated when the wheelchair is moving forwards ($ttls_{ref}>0$).

There are two independent ways to deactivate the cruise control. One is by pressing the Cruise Control button again, the other is by pulling the joystick backwards ($ttls_{ref} < 0$).

The MATLAB code for the Cruise Control system is shown below:

```
    if but1 > but1_1
        if (crs_ctrl == 0) && (ttls_out > 0)
            crs_ctrl = 1;           % off -> on
            ttls_crs = ttls_out;    % get current sync ttl value
        else
            crs_ctrl = 0;           % on -> off
            ttls_crs = 0;           % clear sync ttl value
        end
    end

    if crs_ctrl == 1
        if ttls_ref >= 0            % only when moving forward
            ttls_out = ttls_crs;    % replace output sync ttl value
        else                        % cancel if moving backward
            crs_ctrl = 0;           % cancel cruise control
            ttls_crs = 0;           % clear sync ttl value
        end
    end
```

Code 6.1.5A: MATLAB code - Cruise Control

6.1.6 Electronic Auto-Braking System (EABS)

The Electronic Auto-Braking System is an additional safety feature for the wheelchair. It is a system that would engage the brakes on two wheels when the wheelchair is stationary. If the wheelchair is stationary, the variables $ttls_{out}$ and $ttld_{out}$ must be zero, i.e. all the output throttle values must be zero. Thus, whether the wheelchair is stationary can be told by these two variables. However, due to inertia, the wheelchair would not stop immediately if output throttles change suddenly to zero. Therefore, a delay is necessary. In the parameter settings in the code, the parameter "t_stat_max" determines how long the system should wait after no output throttles in second. LEDs are used to simulate the state of brakes:

$$Brake\ Light\ LED\{ON = brakes\ engaged,\ OFF = brakes\ disengaged$$

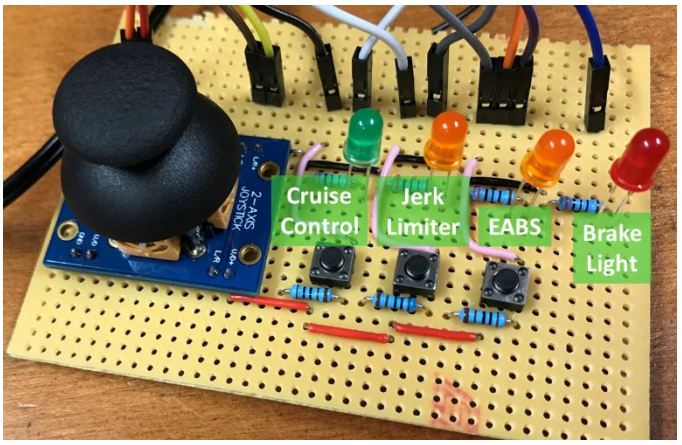


Figure 6.1.6A Control interface

Figure 6.1.6A shows the control interface of the wheelchair. The orange LED labelled "EABS" is the indicator of the EABS system. The red LED is the brake light which simulates the braking mechanism.

The Electronic Auto-Braking System has not been implemented in the prototype during our demonstration. The motors we use for the prototype cannot spin when there is no voltage applied to their terminals, which is similar to what we would expect if adding an EABS system. However, the EABS is too mechanically sophisticated for us to design and build. The pin connected to the brake light can be connected to an actuator to engage/disengage the brakes on wheels.

6.1.7 Output Throttle Values

The joystick data has been converted into output synchronized and differential throttle values $ttls_{out}$ and $ttld_{out}$. Finally, they are converted into left and right motor throttle values using formula (4.1.1C):

$$\{ttl_{out}^{left} = (ttls_{out} + ttld_{out}) \times ttl_{gain}^{left}$$

$$ttl_{out}^{right} = (ttls_{out} - ttld_{out}) \times ttl_{gain}^{right}$$

Formula 6.1.7A

The ttl_{out}^{left} and ttl_{out}^{right} now are from -1 to 1 and need to be converted into signals that the motor controllers understand. There are two important information for motors: rotational speed/torque and direction of rotation. The torque can be determined by the magnitude of ttl_{out}^{left} and ttl_{out}^{right} and the spinning direction can determined by the sign of ttl_{out}^{left} and ttl_{out}^{right}.

The Arduino reads the duty cycle in the program and writes PWM signal to PWM-enabled pins. The duty cycle in Arduino code is an integer from 0 (always off) to 255 (always on). The motor controllers used can reverse the motors' spinning direction by setting the "FW" terminal to HIGH (5V). Therefore, ttl_{out}^{left} and ttl_{out}^{right} can be translated into two PWM signals and two digital signals:

```
#define PWM_left 5          // output digital pin (PWM) for controlling left motor
#define PWM_right 6         // output digital pin (PWM) for controlling left motor
#define rev_left 7          // pin number for reverse (FW) of left motor
#define rev_right 8         // pin number for reverse (FW) of right motor
…
// decide directions of motor rotations (send to FW pins in motor controllers)
if(ttl_left >= 0)   digitalWrite(rev_left, LOW);
else                digitalWrite(rev_left, HIGH);
if(ttl_right >= 0)  digitalWrite(rev_right, LOW);
else                digitalWrite(rev_right, HIGH);

// convert throttle values to PWM signals and send to pins
analogWrite(PWM_left, abs(ttl_left)*255);     // PWM of left wheel
analogWrite(PWM_right, abs(ttl_right)*255);   // PWM of right wheel
```

Code 6.1.7A: Arduino Code - Output Throttle Values

6.1.8 MATLAB Simulation

To test the control theory, we set up a wheelchair simulation in MATLAB. From equation (6.1.4B), we know that the acceleration is controlled by the output throttle values:

$$a = \frac{F_{max}}{m} \times ttls_{out}$$

$$\alpha = \frac{F_{max}}{m} \times \frac{ttld_{out}}{d}$$

Formula 6.1.8A

In the parameter settings section in the MATLAB code, the parameter "accel_gain" is the ratio F_{max}/m.

The joystick is set to 100% forward and 80% right (jy = 1, jx = 0.8) at t=1s, and is returned to central position at t=5s. Theoretically, the wheelchair would move forwards and turn right at the same time. If F_{max}/m is 2, $ttld_{ratio}$ is 0.3 (see Section 6.1.3) distance between two wheels d is 0.6m and taking into account the 15% deadzone for both axes, the linear acceleration a and angular acceleration α for the wheelchair would be:

$$a = \frac{F_{max}}{m} \times ttls_{max} \times \frac{jy - y_{deadzone}}{1 - y_{deadzone}} = 2 \times 0.7 \times \frac{1 - 0.15}{1 - 0.15} = 1.4\ m/s$$

$$\alpha = \frac{F_{max}}{m} \bullet \frac{ttld_{max}}{d} \bullet \frac{jx - x_{deadzone}}{1 - x_{deadzone}} = 2 \times 0.3/0.6 \times \frac{0.8 - 0.15}{1 - 0.15} = 0.765\ rad/s$$

Formula 6.1.8B

The results match with the plots in Figure 6.1.8B and Figure 6.1.8C. Then, for both linear and angular, we can simulate the velocity by taking the integral of acceleration, and the displacement by taking the integral of velocity over time period τ:

$$v = \int_0^{\tau} a dt = v_1 + a \cdot dt$$

$$\omega = \int_0^{\tau} \alpha dt = \omega_1 + \alpha \cdot dt$$

Formula 6.1.8C

where v is linear velocity, ω is angular velocity, v_1 and ω_1 are the linear and angular velocity calculated in the previous loop.

The heading θ of the wheelchair can be determined by taking the integral of angular velocity over period τ :

$$\theta = -\int_0^{\tau} \omega dt = \theta_1 - \omega \cdot dt$$

Formula 6.1.8D

where θ_1 is the heading calculated in the previous loop. The negative sign is taken for the integral because conventionally an anticlockwise rotating angle is positive but v_d is positive if wheelchair rotates clockwise ($v^{left} > v^{right}$). And the coordinate (x,y) for the new wheelchair position would be:

$$x = \int_0^{\tau} v \cos cos\ (\theta)\ dt = x_1 + v \cos cos\ (\theta)\ dt$$

$$y = \int_0^{\tau} v \sin sin\ (\theta)\ dt = y_1 + v \sin sin\ (\theta)\ dt$$

Formula 6.1.8E

where x_1 and y_1 are the coordinates calculated in the previous loop. The MATLAB code includes calculations for other indicating points to refer wheels and direction, so that the wheelchair can be visualised from a top-down view in real-time in MATLAB.

Figure 6.1.8A, Figure 6.1.8B and Figure 6.1.8C are the results for an ideal simulation and a non-ideal simulation:

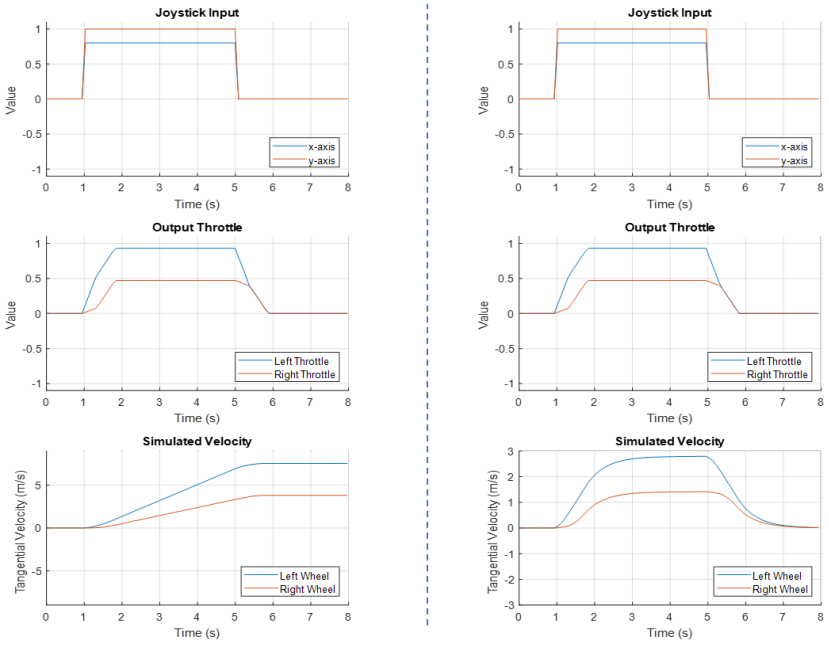


*Figure 6.1.8A: Joystick inputs, output throttles and simulated tangential velocity for **ideal (left)** and **non-ideal (right)** simulations*

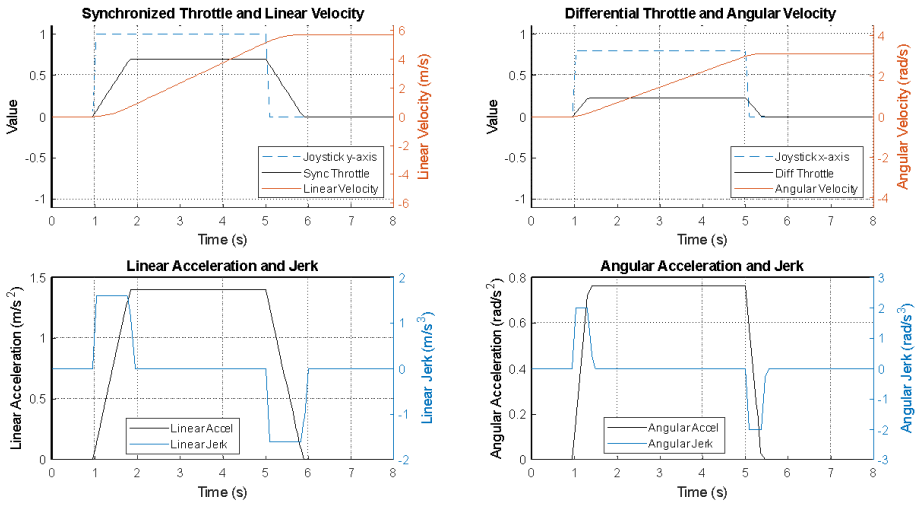


*Figure6.1.8B: Linear and angular analysis in **ideal** simulation*

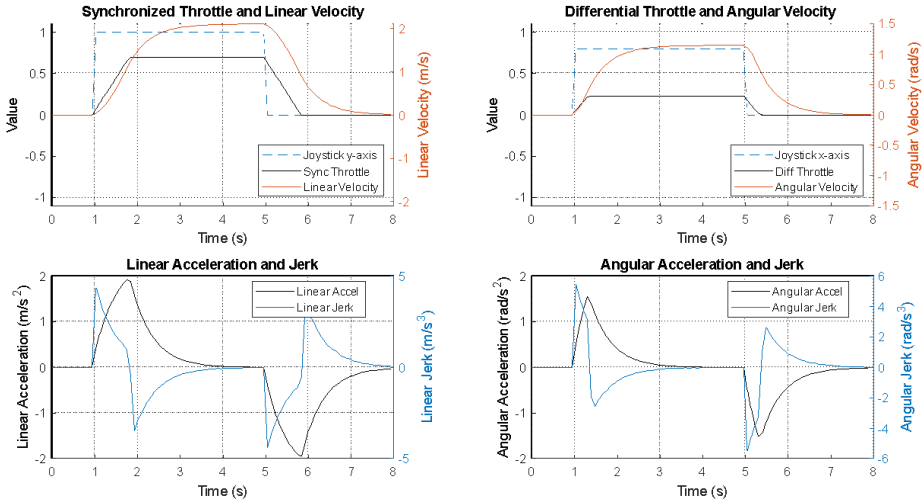


*Figure 6.1.8C: Linear and angular analysis in **non-ideal** simulation*

From Figure 6.1.8A, in an ideal situation, there is no friction or air resistance so the velocity would continue increasing and keep constant when the joystick is returned to central position; however, in a non-ideal simulation, a negative exponential decay is used to simulate the reduction of velocity due to friction. It can be seen that although the joystick changes abruptly, the output throttles have finite gradients and are smoothed out by the jerk limiter. The curvature of the simulated left and right wheel tangential velocities also shows that the acceleration and deceleration are very gentle.

However, in Figure 6.1.8A, the linear and angular velocity, acceleration and jerk cannot be directly seen from the tangential velocities of left and right wheels. Figure 6.1.8B and Figure 6.1.8C are the linear and angular analysis (velocity, acceleration and jerk) for an ideal and a non-ideal simulation. From Figure 6.1.8B and Figure 6.1.8C, it can be concluded that our control algorithm works within our expectation. The jerk is limited effectively for both ideal and non-ideal situations, providing a very smooth and comfortable user experience.

6.2 Wheels app

All code screenshots can be found in the appendix and the app software itself is in a public Github repository called "WHEELS".

The app is designed to connect to the arduino on the wheelchair via Bluetooth LE in order to receive data for diagnostics as well as send data to control the wheelchair itself.

6.2.1 Login Activity

Purpose - To protect from external threats trying to access the app software.

Rationale - On installation and initial startup, the user is required to go through a login authentication for one time only. This ensures that no one can download the app and control another user's wheelchair.

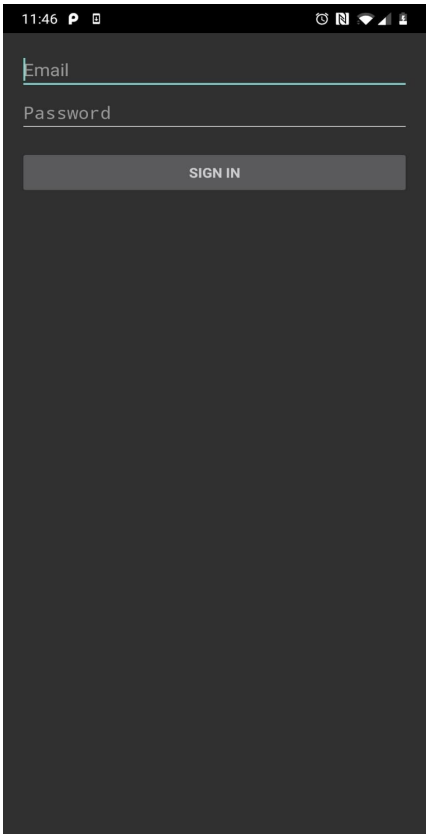


Figure 6.2.1A: Empty Login Activity

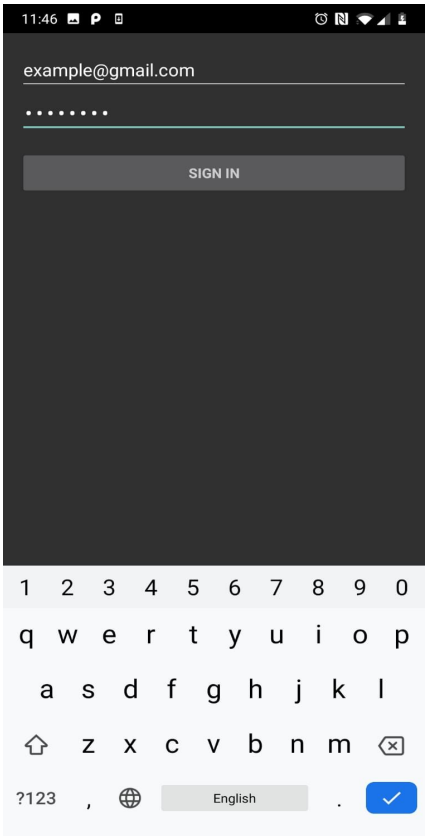


Figure 6.2.1B: Login Activity with credentials

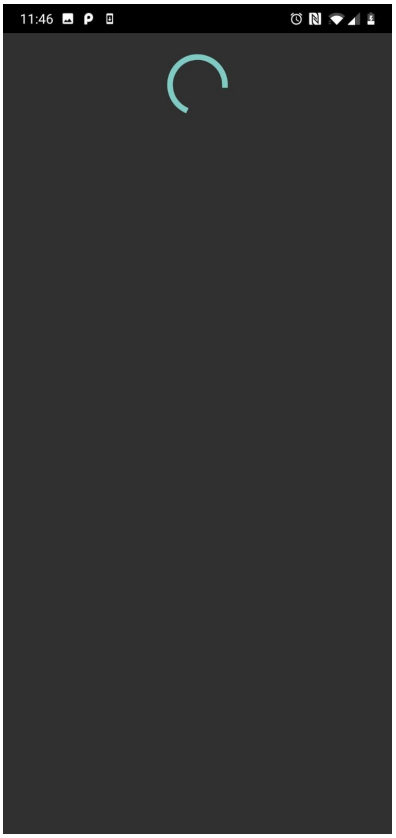


Figure 6.2.1C: Background Login View

As shown in Figure 6.2.1A, the layout consists of two text fields, one for email and one for password, as well as a sign in button. The user can input their credentials as shown in Figure 6.2.1B in order to gain access to the rest of the app software.

Currently, the project has no access to any external servers so for now the app uses a hash map called "DUMMY_CREDENTIALS" that acts to simulate valid login data. Each key is an email address and is linked to a password value. A hash map is used because each stored key has a unique assigned value.

The “attemptLogin” function checks for any user input errors before moving onto the real login background task. The main error handling is done by the boolean function “isEmailValid” which can be modified to whatever the user desires, but for now simply checks if the string inputted contains an @ symbol. This function also checks if the user left either email or password field empty, returning with prompts to fill in these fields.

The “UserLoginTask” class extends AsyncTask in order to override a background process, shown in Figure 6.2.1C, which checks if the user inputs match any one of the valid credentials for login. Currently the network access to get the valid credentials is simulated since for now there is no external server to access and the DUMMY_CREDENTIALS parameter is used. A simple for loop is used to check through the hash map for the correct email and password combination. The “onPostExecute” override class handles success and failure cases. If successful, the Device Scan Activity is opened (see section 6.2.3).

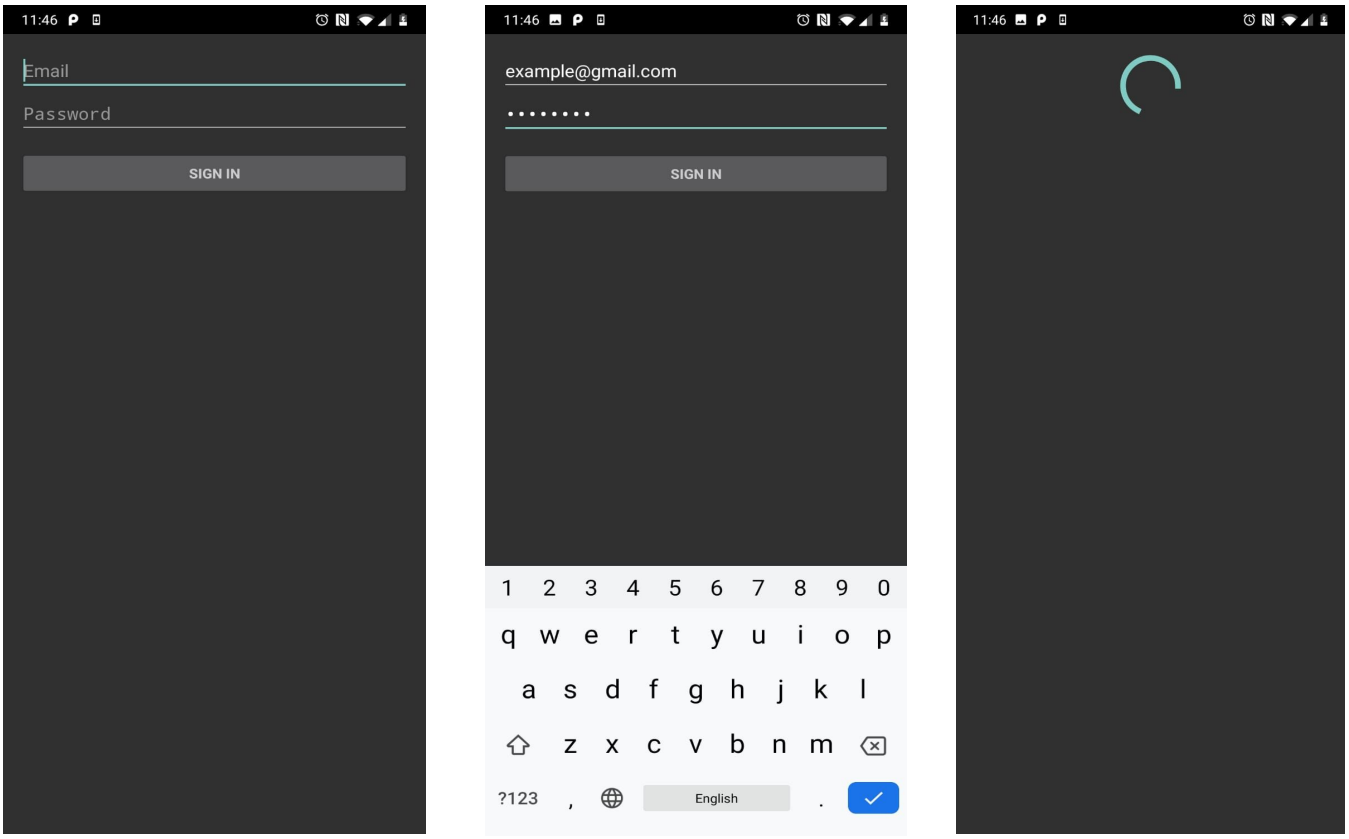
Ideally, the user should only have to login once on installation of the app. In order to achieve this, a shared preference data type is used. Once successfully logged in, the boolean “isLoggedIn” is stored as true in the shared preference. On the next startup, the value of isLoggedIn is checked, and if true, skips this activity altogether.

6.2.2 Bluetooth LE Service

Purpose - To manage the bluetooth connection, including all connection states.

Rationale - The user should be able to send and receive data from the wheelchair via Bluetooth LE.

The “initialize” boolean function checks if the phone that the app is installed on is able to handle bluetooth communication by setting up the BluetoothManager and BluetoothAdapter.



All state changes and service discovery updates are handled by a callback function “mGattCallback”. This contains two override functions; the first is called upon a state change and send a broadcast message to be read by any broadcast receivers (specifically in the DeviceControlActivity class). The second override function is called when a bluetooth service is found and sends another broadcast update.

The “connect” and “disconnect” functions simply connect or disconnect the phone from an external bluetooth device. In the connection case, there is a check is a reconnection service for previously connected devices. This works since a global variable containing the device address is updates once a connection is made so this can be checked to see if it matches the address attempting to be connected.

The function “readCustomCharacteristic” is a string that returns data read from an external bluetooth device that has been connected. The relevant UUIDs were obtained from the Adafruit website. Data is transmitted in byte format so this is converted to a string if data is successfully obtained. The “writeCustomCharacteristic” function takes in a string to be sent to an external bluetooth device. Again, the required UUIDs were obtained from the Adafruit website. Since data is transmitted in byte form, the message is converted from a string to a byte array.

6.2.3 Device Scan Activity

Purpose - To scan for nearby bluetooth devices and show them to the user in a list.

Rationale - The user should be able to decide which device they want to attempt a connection with.

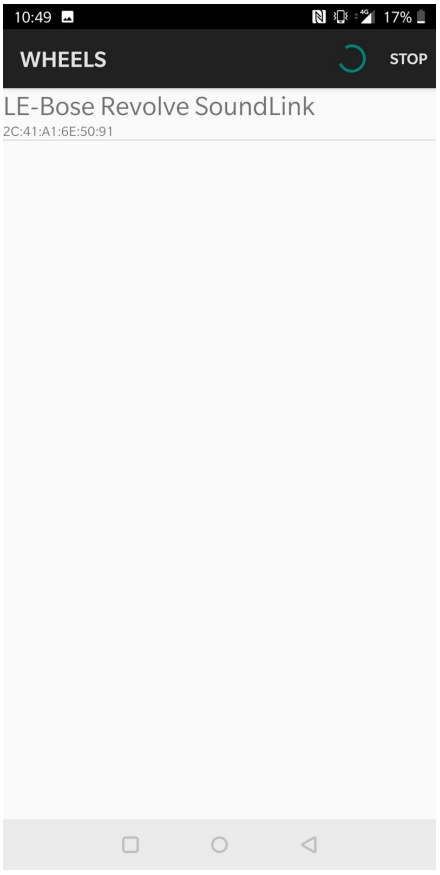


Figure 6.2.3A: Device scanning view

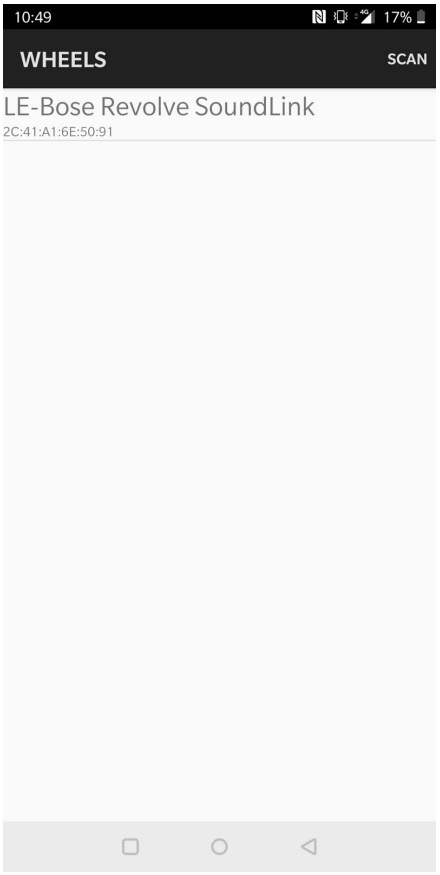


Figure 6.2.3B: Device stop scanning view

The layout changes between scanning and not scanning are handled by the “onCreateOptionsMenu” override function which uses a global boolean “mScanning” to set certain UI features visible depending on whether the phone is scanning or not.

Scanning will start for 10 seconds whenever the “SCAN” button is pressed. The override function “onOptionsItemsSelected” starts the scan by calling the function, “scanLeDevice” which sets mScanning to be true to update the UI before starting the scan callback function “mLeScanCallback”. In this, whenever a device is detected, it is added to a ListView Adapter held in the class “LeDeviceListAdapter”. This has a BluetoothDevice array containing all the devices found and puts them in an indexed View, showing the device name and address.

The “STOP” button will stop the scan. This works again through the scanLeDevice function which sets mScanning to be false to update the UI and cancels the callback function mLeScanCallback. However, the ListView is not cleared so all found devices are still shown.

When a device is selected to connect with, it is obtained from the BluetoothDevice array using the index number given to it in the ListView. The name and address of the device is then retrieved and the “FingerprintService” Activity is started (see section 6.2.4).

6.2.4 Fingerprint Service

Purpose - To protect from external threats trying to access the app software.

Rationale - Standard login is secure but passwords are easy to forget. A better solution is to make the user only login once with email and password and then be able to use fingerprint authentication to verify user identity.

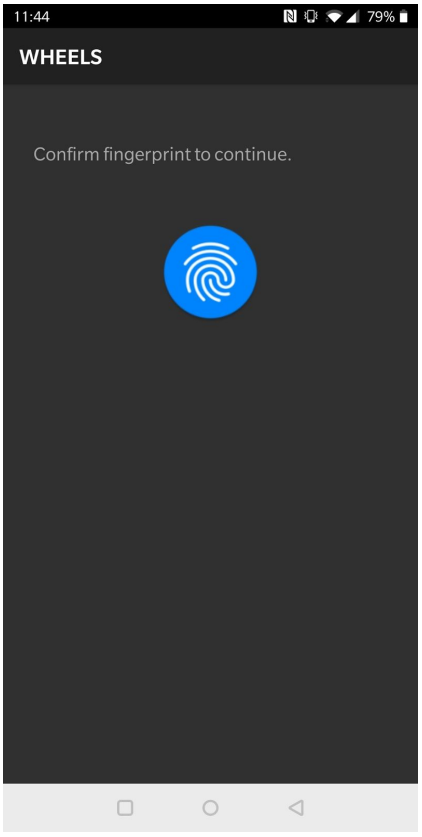


Figure 6.2.4A: Fingerprint startup view

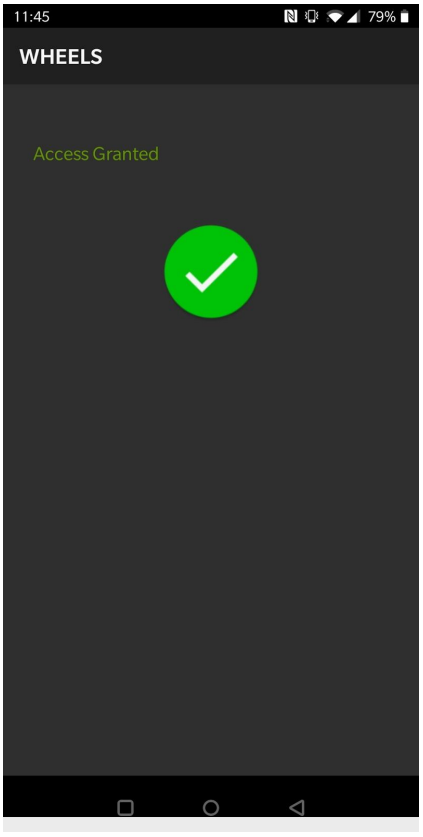


Figure 6.2.4B: Fingerprint verified view

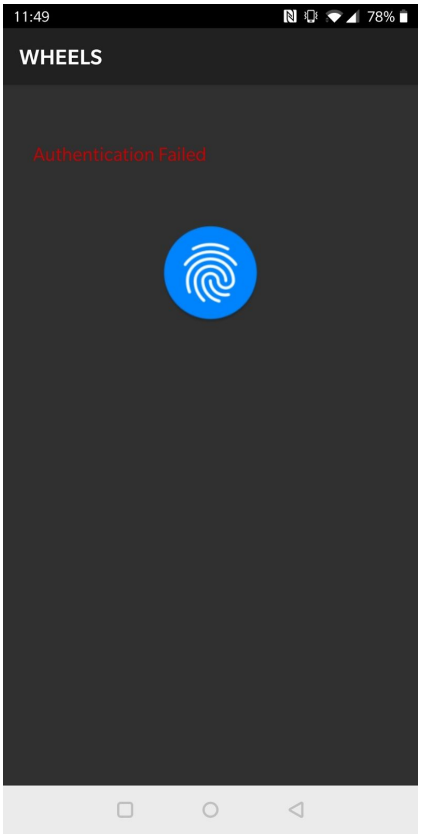


Figure 6.2.4C: Fingerprint unauthorized view

Firstly, the app must check if the phone can handle fingerprint verification. This is done under the boolean function “checkPermissions”. If this returns true, then the fingerprint authentication can begin in the class “FingerprintHandler”.

The actual fingerprint checking is handled by the “AuthenticationCallback” function which is called in the “startAuth” function. There are four possible returns, each handled by an override function. One is the success case and the other three are different failure cases, one for an unauthorized fingerprint, one for fatal errors and one for non-fatal errors. In each case, a string and a boolean are sent to the function “update”. Here, the UI is updated depending on whether the boolean is true (success) or false (error). Furthermore, if successful, the “DeviceControlActivity” is started (see section 6.2.6).

6.2.5 Joystick View

Purpose - To draw and get correct data from the software joystick used to control the wheelchair.

Rationale - The user should be able to use their phone to control the wheelchair since almost everyone has an easily accessible phone.

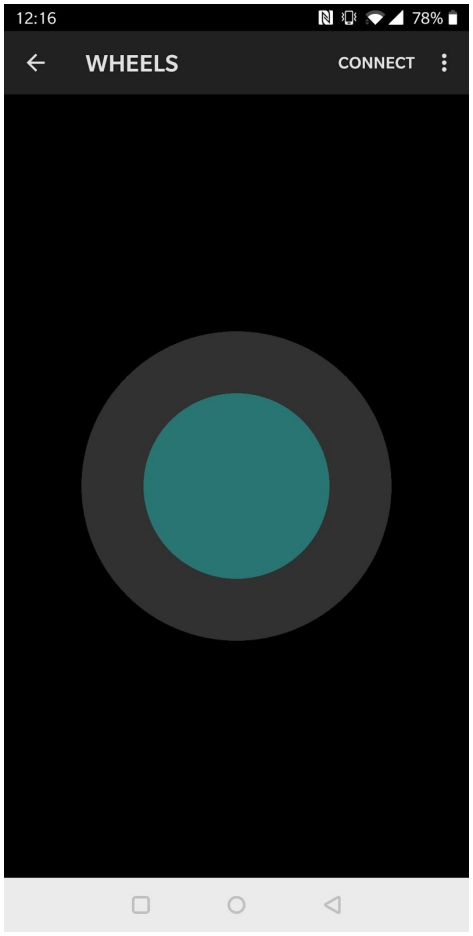


Figure 6.2.5A: Joystick stationar

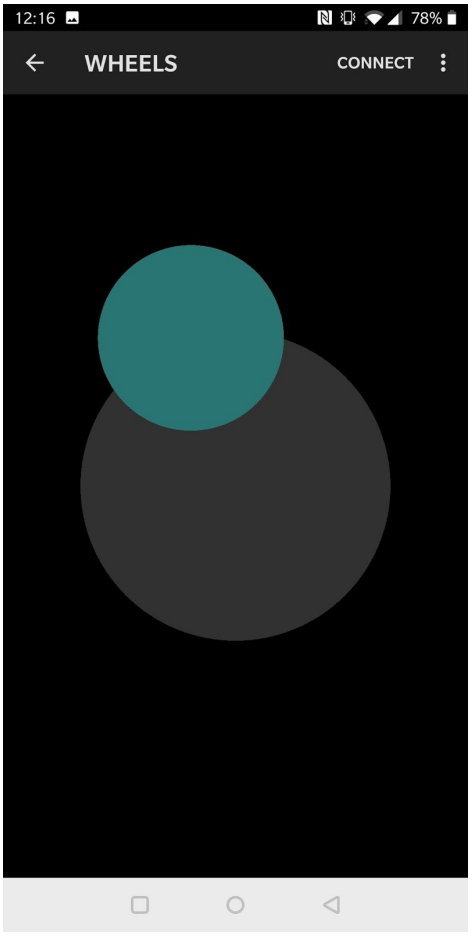


Figure 6.2.5B: Joystick moving

The joystick should be centred around the middle of the screen. However, when drawing in Java, the origin is at the top right hand corner of the screen. This needs to be reset to the centre by getting the middle pixel. This is done in the function “setupDimensions” which also get the radius of the actual joystick circle (the blue/green one shown in Figure 6.2.5A and Figure 6.2.5B) and the base radius (the dark grey circle shown in Figure 6.2.5A and Figure 6.2.5B). Next, the joystick and base can now be actually drawn, taking in x and y positions. This is done using a new Canvas data type through which a circle can be drawn using the parameters obtained through setupDimensions. The colours of both circles are also set here.

The “onTouch” function is where the joystick physics is programmed and is called every time the screen is touched; drawing the joystick circle given the x and y coordinates of where the user touches the screen. Pythagorean theorem can be used here to get the position of where the user has touched the screen. This is stored in a float variable called “displacement”, which gets the total displacement from the centre of the screen through the square root of the x displacement added to the square root of the y displacement. However, the joystick circle should never leave the base, otherwise it looks unrealistic, so the displacement needs constraining if the user taps the edges of the screen. To do this, simply

check if the user has tapped outside of the base radius. If not, no constraining is required. However, if they have, then the parallel triangles identity is used to constrain the joystick movement. This states that if there are two triangles with two parallel sides and one shared angle, the ratios of all three sides will be the same. In this case, when clicking outside the base radius, the angle of the displacement should still be the same, giving the shared angle, and both the desired constrained and unconstrained triangles will be right angled triangles so at least two of the sides will be parallel; allowing the identity to take form. This is shown below where dY and dYc are parallel and dX and dXc are parallel:

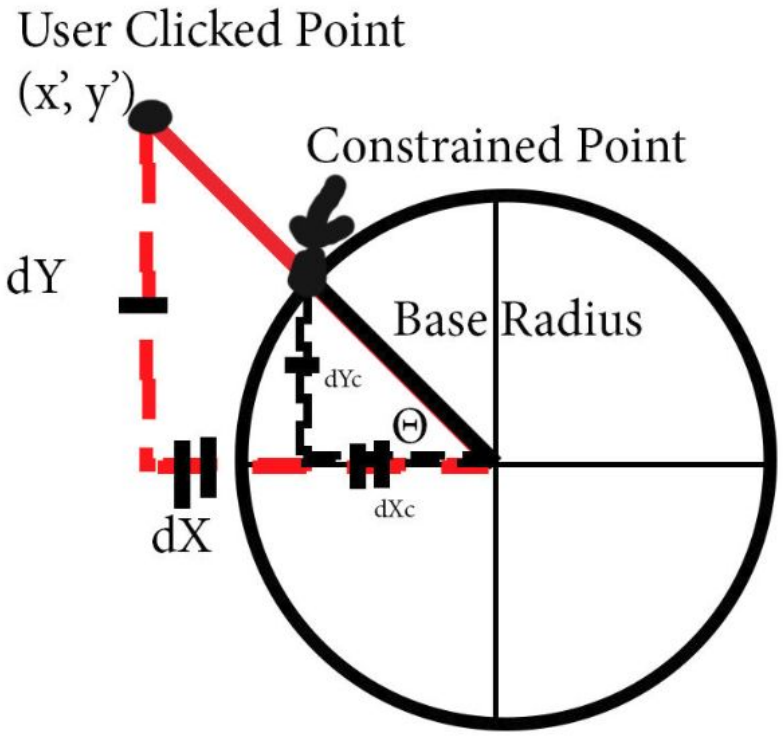


Figure 6.2.5C: Parallel triangles identity

Using the identity, the ratio of the base radius to the user clicked point is the same as the x displacement to the constrained x value and the y displacement to the constrained y value. Therefore, the variable "ratio" get the base radius and divides it by the clicked displacement, and this value is multiplied by the x and y displacements to give the constrained coordinates to be drawn.

Now, real values need to be obtained from the joystick position. The arduino works using data from -1 to 1 in both the x and y directions; however, for now the joystick should output values from -100 to 100 (see section 6.2.8.3 for more detail). To do this, an interface "JoystickListener" is created, taking two floats representing the x and y displacement of the joystick. This is called as a callback function in the onTouch function so that whenever the screen is touched, new values are created. Whenever the screen is let go, the value (0,0) is passed since the joystick returns to the centre of the screen. Otherwise, a scalar of 3.58 is used to mostly scale the output down to between -100 and 100. However, due to the nature of the float data it is almost impossible to exactly scale the output, so if the magnitude ever goes slightly above 100, it automatically gets reduced down to 100 before being sign converted if necessary.

6.2.6 Device Control Activity

Purpose - To send real data to the arduino on the wheelchair.

Rationale - Data must be sent to the arduino to control the wheelchair, with functionality including movement control as well as toggling cruise control, the jerk limiter and the EABS.

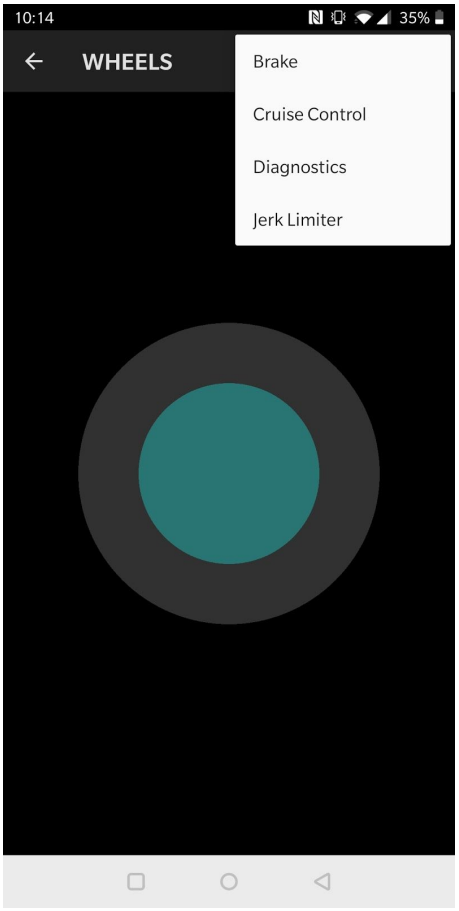


Figure 6.2.6A: All available buttons

By this point, the user has already selected a device to connect to, so this class must initialize and maintain the desired connection. To do this, it calls the "BluetoothLeService" class (see section 6.2.2) using an intent, passing through the "mServiceConnection" function which contains override instructions for connections and disconnection states. This class also contains a broadcast receiver, "mGattUpdateReceiver", which can read different connection states. For now, it simply displays a Toast message to the user when they have connected or disconnected their phone.

The arduino works at 20Hz. Therefore, it was found that to reduce latency but still send data as quickly as possible, the app should send movement data at 10Hz (see section 6.2.8.3). To do this, the class implements the JoystickListener interface (see section 6.2.5) and overrides the "onJoystickMoved" function which is called whenever the user touches the screen (again see section 6.2.5). This takes both x and y float coordinates of the user input (ranging from -100 to 100) and rounds them to the nearest integer. An integer array, "data", of size two then continually updates with these rounded integer values. A timer task is now described, taking each data value and converting it

into a string before making a message containing the x coordinate followed by a comma followed by the y coordinate (e.g. “25,-46”). This message is finally written to the write characteristic to be read and decoded by the arduino. A timer is also created to schedule the timer task with a period of 100 milliseconds and an initial delay of 1 second. The delay is necessary in order to ensure the connection is set up and stable before trying to send data.

A range of buttons are also available for extra control over the wheelchair, illustrated in Figure 6.2.6A. A switch statement is used in the “onOptionsItemSelected” function to handle each button event separately, and in each case, a different integer code is sent to the “toggleControl” function. Here, depending on the code, a different single letter string will be sent to the arduino to decode and a specific Toast message will be shown to the user on the app (e.g. “Jerk Limiter: Enabled”). The only exception is when the Diagnostics button is clicked, in which case the diagnostics menu is opened (see section 6.2.7) and the data writing process is cancelled.

6.2.7 Diagnostics Service

Purpose - To read data from the arduino on the wheelchair.

Rationale - By reading data, diagnostics information can be easily obtained by future engineers or users trying to modify their wheelchair.

Figure 6.2.7A: Diagnostics Menu

The diagnostics menu is deliberately not complete to allow other users to take the source code and program in exactly what diagnostics information they would like to see; appealing to the open source nature of the project. For now, this menu stands as a demo which reads data from the arduino and puts it in the Battery TextView, shown in Figure 6.2.7A. However, currently, the arduino does not send any information so the space appears blank. The code is also built to be user friendly with instructions on where to manipulate variables.

This class uses the same code as in section 6.2.6 to manage the bluetooth connection and call the "BluetoothLeService" class (see section 6.2.2).

Another timer task is used which reads data available from the connected device and puts it in a TextView. A timer is used to schedule the timer task with a period of 100 milliseconds and an initial delay of 500 milliseconds. Although these parameters can easily be changed.

6.2.8 Testing

This section will explain the different tests done to ensure the app works as desired.

6.2.8.1 Software Joystick

It is important to ensure the software joystick works and is correctly calibrated. To do this, follow the instructions below:

1. Uncomment line 140 in the “DeviceControlActivity” class.
2. Run the app.
3. Check the run dialog.

SUCCESS CASE:
Data should appear on the run dialog in the form “number,number” with each number between -100 and 100. The numbers should change when the joystick is moved and return to “0,0” when released.

FAILURE CASE:
No data appears or the range of data is in the wrong range.
Check the “scalar” value in line 95 of the “JoystickView” class and tweak it to calibrate for the correct range.

6.2.8.2 Sending Data

The bluetooth module bought contains a library to be used with the arduino that has a file called “echoDemo” which, when compiled and run, prints any data received to the serial port. This file was used to test whether the app could send data via bluetooth. The steps to be taken for this test are illustrated below:

1. Connect to the arduino with the bluetooth module attached.
2. Open the arduino serial monitor and the app run dialog.
3. Move the software joystick.
4. Observe app and arduino outputs.

SUCCESS CASE:
Data appears on the arduino serial monitor in the form “number,number”.

FAILURE CASE:

Data does not appear on the arduino serial monitor.
Check the app run dialog, if there is a message saying “Failed to write characteristic”, then the app is the source of the problem, otherwise, it is likely the problem is server side. Apply the software joystick test again (details in section 6.2.8.1).

6.2.8.3 Latency

Ideally, there should be no latency between the user input via the software joystick and the output from the motors. However, in the real world, there has to be at least some latency in data transmission; and in this case, the largest delay arises from the transmission of data via the bluetooth link. As a compromise, this delay must simply be so small that it is unnoticeable. In the demonstration case, the arduino runs at 20Hz. For the app to reliably send data without output delay, it was found that the maximum data transmission rate is approximately 10Hz. Furthermore, the size of each data packet was reduced from over 20 bytes per message to a maximum of 6 bytes per message. This was achieved by converting the data from float to integer form. However, if this was done in the data

range between (-1,1), then only the three values, -1, 0 and 1, would be possible, whereas a higher discrimination of data is necessary for accurate movement control. In order to fix this, the range of data on the app side was increased to (-100,100) to be divided by 100 on the arduino side. To test this, the following steps were taken:

1. Run both the app and arduino code.
2. Open the arduino serial monitor.
3. Move the software joystick in any direction for 10 seconds and observe the serial monitor output.
4. Release the joystick.
5. Observe if there is any noticeable delay for the serial monitor output to show “0,0”.

SUCCESS CASE:
There is no noticeable delay for the serial monitor output to change to “0,0”.

FAILURE CASE:
There is a noticeable delay for the serial monitor output to change to “0,0”.
Try reducing the frequency at which the app transmits data since it is possible for the data to become backlogged.

6.2.8.3 Reading Data

The app must be able to read data in order to service the diagnostics activity. To test this, follow the instructions illustrated below:

1. Run the app and arduino code.
2. Open the diagnostics menu.
3. Check the text view next to the “Battery” label.

SUCCESS CASE:
The number in the text view changes to “1”.

FAILURE CASE:
The number in the text view is blank or “0”.
Check the run dialog and arduino serial monitor. If the message “Failed to read characteristic” appears on the app dialog and the sending message command appears on the arduino serial monitor then the arduino has successfully sent data but the app is failing to read the data.

6.3 Demonstration design

The demonstration high level design can be found in section 4.2.

The motors are mounted underneath the wheelchair as can be partially seen in Figure 6.3A. arduino, motor drivers, and battery are all held on the seat of the wheelchair as the wheelchair can not be sat on anyway.

6.3.1 Motors

To calculate the power of the motors required for a wheelchair some calculations were done.

The below table of variable is used, the 2nd column is the maximum requirement and the 4th column is the minimum operating requirement. The 3rd column explains where the numbers are from

Weight of vehicle and user	150KG	Weight of chair 20KG Person 130KG	103.6KG using 83.6KG as average male in UK
Required speed	$2ms^{-1}$	Walking speed of $1.4ms^{-1}$ + $0.6ms^{-1}$ extra	$1.4ms^{-1}$ Average walking speed

Efficiency of motor	70%	Unknown but this should be a reasonable lower bound	100%
Rolling resistance	0.0055 BMX	WIKIPEDIA ESTIMATION	0.0055
Coefficient of drag	1.4	NASA PAPER (DRAG WITH MANNEQUIN)	1.4
Area of vehicle	$0.5m^2$	Estimation from NASA paper	
Density of air	$1.2KGm^{-3}$		
Wheel radius	0.305m	Standard wheelchair	

See aerodynamic characteristics of wheelchairs from NASA.
https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19800004744.pdf

1. **Calculate RPM**
   a. Circumference of wheel
      = 2 * pi * r
      = 1.9163715m
   b. RPM
      = (v / (a))*60
      = 62.618338

2. **Calculate Power**
   a. At max speed
      Power = F*v
      = Resistive forces * velocity
      =(Rolling resistance + air drag) * v
      = $(M * g * R_R) * v + (\text{air density} * C_d * \text{Area} * v^2) * v$
      = $(150 * 9.81 * 0.0055) * 2 + (1.2 * 1.4 * 0.5 * 2^2) * 2$
      = 16.1865W + 6.72W
      =22.9W (3sf)

   b. For initial acceleration
      Power for acceleration $\sim= 0.5 * m * a * v$ (velocity increases from 0 to max as it accelerates)
      = 0.5 * 150 * 0.5 * 2
      = 75W

   c. Total max power.

At the initial acceleration v = 0 so there are no dragging forces, but after this there are and initial acceleration force reduces.
We would have to simulate to find max force needed at any point. But we can produce an upper bound, using the resistive force at max velocity and the force for initial acceleration.
75W +22.9W = 97.9W
This should be an overestimation as it is just an upper-bound using power at v =0 and v=max.

3. **Calculate Torque**
   a. Using efficiency of 70% we need a power in of 97.9W/0.7 = 140W
   b. W = 2* p* RPM /60 = 6.56
   c. Torque = 140/6.56 = 21.34Nm

This is the total torque required max (half for each motor).

This is a good rough approximation.

However at a high speed on a slope the 150kg person + chair being lifted at the vertical projection of that speed the power gives 150 x 9.81 x 1.9 x sin 10º = 485 W. This is a large amount of power for slopes but could be reduced by moving at a lower speed during slopes. Also in reality the person + wheelchair is much less than 150KG.

Motors of this capability are very expensive and not within the budget for the demonstration. Two DC motors were however gifted to this project of 60W at 12V.

Figure 6.3A Motors used.

These motors were deemed good enough for the demonstration but would probably fail with the added weight of a person on top of the wheelchair.

With these motors all capabilities of the control box could be demonstrated.

6.3.2 Motor Drivers

The motor drivers were the most expensive part of the project. EM-24xA DC-Motor Controllers were used, they are full bridge DC-motor starters. These motor drivers required the EM-236 interface unit to program.

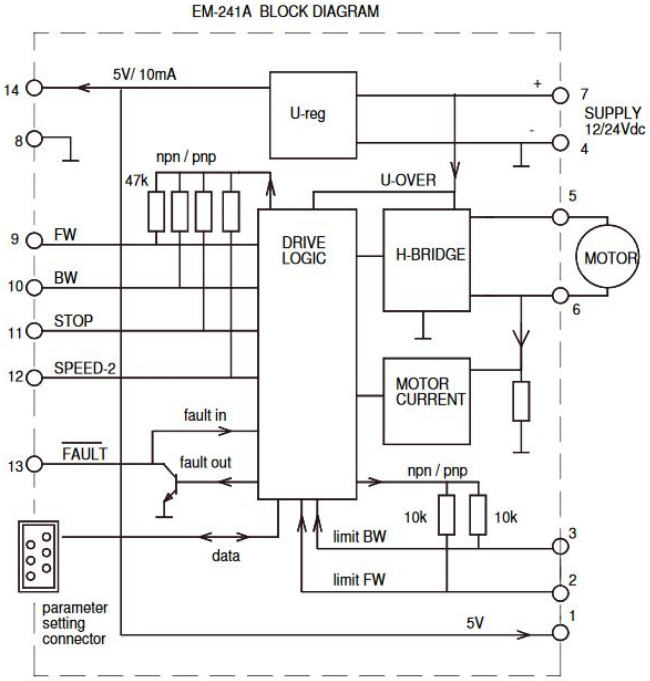


Figure 6.3.2A: Internal diagram of motor driver.

These drivers had the capability of taking in a PWM signal to drive the motors at the correct speed. Parameter 5 was set to 0 so that the SPEED-2 input could be used as the PWM control. In this mode FW pin logic controls the direction of the motor.

6.3.3 Battery choice

A battery was required to power the motor drivers/ motors at 12V and the arduino and mechanical inputs. A 12V 24AH battery was chosen in order to power wheelchair for the duration of the demonstration.

7. Final Prototype and Results

The demonstration took place on the 13/6/19. The demonstration included the wheelchair prototype and the matlab simulations of the wheelchair movement.

To prove the success of the project, the designs were compared to the specifications set out in section 2.2.

Specification	Success?	Rationale
The code created must be easy to understand, hack and modify.	YES	To demonstrate this, the control software was open for viewing. It has been made easy to understand using comments in the code and the explanations in the document. There are points in the code, such as in the wheels app diagnostic page, where space has been explicitly left for future engineers to add their features and this has been pointed out in the code. To make sure the code is easy to modify, all of the controller and app code is placed in a public repository with the link in the appendix.

There must be acceleration and jerk limiter software in place, both for safety and comfort.	YES	The jerk limiter software was demonstrated firstly in the matlab simulations, where it could be turned on or off and the difference between simulated velocity profiles and simulated wheelchair positioning were shown. It was then demonstrated on the physical wheelchair, with the wheelchair moving much smoother when the jerk limiter was on, and there were no sudden movements if the user suddenly switched direction on the wheelchair.
The controller software should be able to adjust the input joystick variables (e.g. non-linearity, sensitivity and dead zone area).	YES	This was demonstrated on the matlab code, as can be seen in section 6.12.
Low cost of controller design is essential.	YES	The design of the controller is low cost as the project is simply open source code, based on cheap arduinos. The demonstration is the only expense as described in section 8.
An Electronic Auto-Braking System (EABS) should be included.	YES	Demonstrated on wheelchair via LED light.
A secure Android App should be developed as a potential input device as well as for easily getting diagnostics data from the wheelchair.	YES	Demonstrated by controlling the wheelchair with WHEELS app.
Cruise control functionality.	YES	Demonstrated on both matlab code and wheelchair.

As shown by the table above, the final controller code met all of the specifications that were set out at the beginning of the project.

8. Budget and Costings

The cost of the product is entirely free due to the open source nature of the product, however in demonstrating the code and building a prototype wheelchair, several costs were incurred. The budget of the programme is £500.

Supplier	Product Description	Unit Price (£)	Quantity	Total Price (£)	Quantity Used In Demo	Total Price On Demo (£)
RS	Arduino Uno	19.73	2	39.46	1	19.73
Supervisor	Motors	FREE	2	0	2	0
RS	EM-24xA DC-Motor Controller	105.50	2	211	2	211
RS	EM-236 interface unit	98.03	1	98.03	1	98.03
RS	Arduino DC power plug	1.52	1	1.52	1	1.52
RS	Lead acid 12V 24AH Battery	80.28	1	80.28	1	80.28
RS	15A Cartridge Fuses	0.286	10	2.86	2	0.572
Stores	Push Button Switch	0.24	3	0.72	3	0.72
RS	LEDS	0.378	5	1.89	4	1.512

Therefore the total amount spent throughout the project was £435.76 and of this cost, the cost of the demonstration was £413.364.

9. Sustainability and ethics report

There is a strong commitment to acting responsibly in all project activities to maintain the trust of the users, developers and other stakeholders. The need for transparency is more acute in an open source application which collects information from the users. For example, the users' fingerprints are required for the phone application, and this might cause a potential privacy issue. Every ethical issue cannot be entirely predicted, but the main principle is to ensure that stakeholders know that the responsible option will always be preferred.

The strategy to facilitate modifying manual wheelchairs aims to rapidly expand the use of affordable powered wheelchairs. In realising this ambition, safeguarding customers' security is increasingly important. The users are recommended to undergo a full assessment for Dementia, eyesight and mobility requirements from an occupational therapist. Users' compliance with the speed limit and rules about using lights, indicators and horns listed in The Highway Code are also monitored as far as possible.

As it is widely known, in order to adapt to different indoor and outdoor environmental requirements, electric wheelchairs must be developed and designed with comprehensive coordination of many factors, such as body weight, vehicle length, vehicle width, wheelbase and seat height. Considering these factors, when the speed is too fast, there might exist some potential safety hazards, such as rolling over.

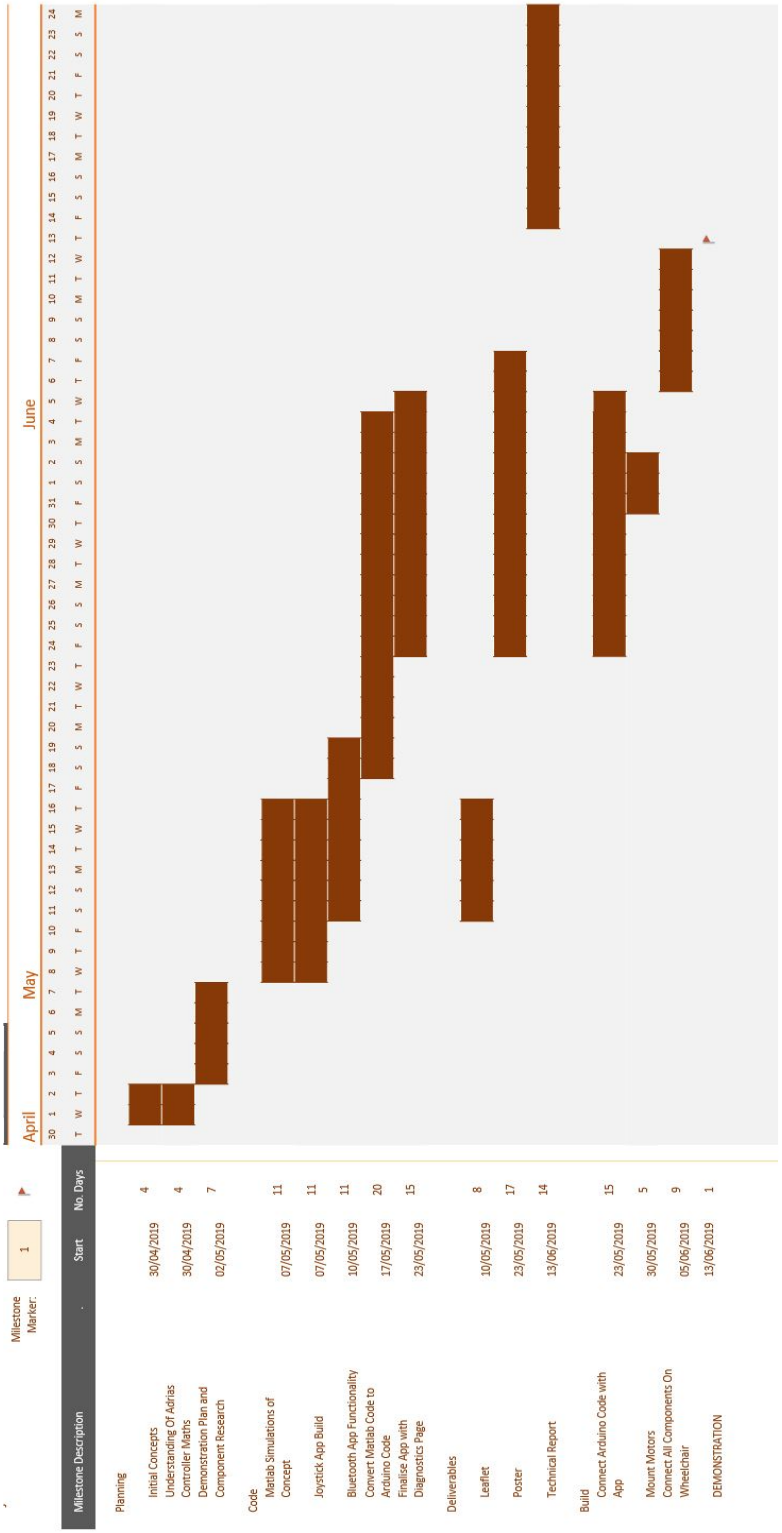
There is also a responsibility to provide information and assistance to the users, industry bodies, and developers to set standards and create guidelines that can help build privacy into the design of new mobile applications, products and services –however they may evolve in the future.

Managing sustainability issues in the supply chain could be another challenge in defining how far the project responsibility extends. Improvements should be driven, for example, by evaluations from our users and developers. But because no one can monitor every tier of the supply chain, this approach should work predominantly with developers and users – with whom there is a direct relationship – to help them improve their performance, and work with them to help implement their own wheelchairs.

Another aim is to reduce environmental impacts across the supply chain and to deliver an innovative, open source product that contributes to the development of a low carbon economy. Using our controller box, there is no need to buy or build an expensive powered wheelchair, engineers could instead easily modify a manual wheelchair into an electric wheelchair. This enables the customers to improve the efficiency of their operations and reduce carbon emissions coming from the factory where electric wheelchairs are manufactured. One way in which the carbon footprint can be reduced is by working with the developers to innovate in order to improve the energy efficiency of wheelchair modifying – bringing additional cost savings and operational efficiencies.

10. PROJECT MANAGEMENT

10.1 GANTT chart



10.2 Meetings

During the course of the project, various meetings were held to discuss and propose ideas moving forward and ways to better optimize the resources and time given. In this section, the contents of the meeting minutes will be outlined as they are.

30th April, 2019

First meeting

The following were to be discussed

- Deliverables
- Setting up a meeting with Dyson building professor
- Study the maths theory
- Discuss project timeline
- Meeting with Adria on Thursday
- Team building workshop on Tuesday

The group decided that by Thursday everyone in the group should have a good understanding of the maths, discussing any flaws and what the inputs and outputs of our control system might be.
Split into two groups before Thursday, each one tasked with coming up with an overview of a solution (basic and non technical) to discuss with Adria on Thursday.
In addition, each member should also look through the mark schemes to check what deliverables need to be done.

2nd May, 2019

Had a meeting with Adria.
Everyone was present.
The Mechanical design for the prototype was discussed.
Adria gave the group a CAN adapter for arduino and some ideas about safety (brake system).
He and the group explained the flaw in the maths (sudden jolts).
A new meeting was also set with him for next Friday.

7th May, 2019

The group was split into two teams for each of the following

- Implementing joystick on raspberry pi
- Matlab improvement

The Arduino was also ordered

The group was struggling to find suitable vesc motor driver, too expensive if they were not ordered on Amazon. The motor drivers from the RS supplier have limited functionality so weren't really suitable.

10th May, 2019

This was a meeting with Adria. The group talked about completing leaflet and the prototype.
The possibility of PCB for prototype was brought up but it was decided best to just use breadboard.
The group also discussed the need to leave room for future development of this product, e.g gyroscopes and encoder.
The group discussed how to implement brakes again which could possibly be mechanical.

The draft leaflet was to be produced this weekend.
The motors were also tested in the lab and looked to be working correctly.

17th May, 2019
This was a meeting with Adria.
He suggested that the leaflet needed to contain more information on the Open Source nature of the control box.
He also suggested that those working on the braking system should email him to check circuit diagrams and ask for help.
He was really impressed with Tom's Matlab simulation and cruise control idea. He also liked the Joystick application capability.
He also gave us another CAN bus shield and also suggested we order the wheelchair now.
Unfortunately, the VESC controllers had not yet arrived.
It was then decided that Tom continue with control algorithms, Rohan with app, Shiwei and Zhengrui with braking system. Also the leaflet needed to be adjusted including changes to mechanical design diagram, Open Source nature and costings.
The Bluetooth arduino module was also ordered.

23rd May, 2019
The group discussed the wheelchair and proposed some mechanical design we could do.
The group also talked about using bigger wheels for the wheelchair and a bar underneath the chair of the wheelchair to support and hold the motors and arduino.
The wheelchair needed to be ordered and the VESC's also hadn't arrived.
The poster was also started on.

30th May, 2019
The group had a meeting with a Mechanical Engineering Lab Technician named Andrew in order to discuss some ideas on how to implement the attachment of the motors to the wheelchair. However, the meeting wasn't as successful as hoped as the most feasible of his proposed solutions wasn't feasible for us as it would take a really long time and need outside contractors.
The group then had a meeting next with Phillip from EEE Mechanical Workshop who helped us come up with a quick solution that would help the group in the short run. He then helped to attach the motors to the back wheels using a bar underneath the chair.
Work on the poster continued throughout the day.

5th June, 2019
The group had to arrange a meeting before the demonstration with the REMAP staff, Simon.

For the demonstration on Thursday, 13/06, it was expected the wheelchair would be able to move and some presentation materials had to be prepared in case the wheelchair didn't work.

A mobile phone application would be suitable to use because it would be inconvenient to have a screen on the wheelchair. Also, the application could show the status of the battery but this is for future work.

The lead acid battery also needed to be purchased for the wheelchair.

The VESC motor controllers will have arrived by this Friday.

The group also discussed the Serial/CAN communication.

The contents for the documentation were to start being prepared.

The group also proposed the idea of placing some items like the joystick and its surrounding components into a box which could either be bought from the EEE store or be 3D printed.

The group also proposed a button to act as a brake.

Call with Simon from REMAP

The group had a video call with Simon from REMAP where our prototype was discussed and what had been able to be achieved. The main idea of the mobile phone application was also discussed with Simon including the Bluetooth connectivity and controlling the wheelchair using the app. Future work using the app was also discussed with Simon. The jerk limiter was also discussed with Simon and how it worked. The cruise control button was also discussed with him. He seemed to be really impressed with what had been done and the demonstration that week was discussed.

11. Future Work

This project is designed to be flexible and open source, given the ability to be further modified and customized to suit the needs of the users. A number of recommendations below are the design choices we envisioned that might be implemented in the future.

11.1 Diagnostics Menu

Since it is inconvenient, expensive and takes up a lot of space to install a display screen on a wheelchair, we chose to transfer various information to our mobile app through the Arduino instead. This will allow the users to monitor the status of the wheelchair in near real-time.The wheels app has been developed with a diagnostics menu which is especially useful for the future developers to decide what data are chosen to be displayed.

11.2 Battery Management System

Battery Management System, commonly called BMS, is a battery monitoring and control system, which is mainly for intelligent management and maintenance of each battery unit, to prevent the battery from overcharging and over-discharging, to extend the life of the battery, monitoring the state of the battery. BMS unit includes management system, the control module and wireless communication module which connects the system with the server. BMS could feed the collected battery information to the user in real time and adjusts the parameters according to the collected information to give full play to the battery performance. The working performance, battery aging,

service life and other information of the battery pack need to be obtained after repeated debugging and experiments, which is tedious and time-consuming.

11.3 Braking System

Although the Electronic Auto-Braking System (EABS) , which is described in section 6.1.6,was not included in the prototype during our demonstration, it is still necessary to be implemented in future practical application. In our initial vision, bicycle-style brakes on the manual wheelchair will be modified and connected to two actuators. When the control system gives an instruction to apply the brake, the actuators will drive the brake blocks to stop the driving wheels.

11.4 Mechanical Design

There are several mechanical design can be implemented in the future to improve user experience.

A container or pallet could be built underneath the seat to hold the battery and the motors.

It will be very helpful to enable the wheelchair to switch between the powered mode and manual mode, and when the users turn off the motor they could switch to the manual mode and let others push the wheelchair at the back. The direct connection between wheels and motors cause the wheelchair unable to move in manual mode, because the motors do not turn when not powered. However a clutch system or gear system could be produced to allow for a manual mode control.

A damping system or a suspension system could be designed to reduce vibrations on the vertical axis to provide a more smoothing user experience.

An anti-tipping device could be designed and installed at the back bottom of the wheelchair to prevent the wheelchair from tipping, which is important for keeping balance.

12. Conclusion

In summary, this report has described the entire design process of the WHEELS open source controller project and has left much scope for the future.

The controller meets the specifications of the client and more, with this paper also exploring the sustainability and ethics of the designed product. The project has tried to go beyond the brief to demonstrate what could be done in the future using this basis.

From the start of the project the work done during the project time, described in this paper, was never meant to be the entire extent of this product, however this paper has enough detail for the reader to pick up where this project has left off and further expand the open source controller with their own ideas or from some of the suggestions from the future work section.

13. Bibliography

Section 3 - market analysis
https://kdsmartchair.com/blogs/news/18706123-wheelchair-facts-numbers-and-figures-infographic

Section 1 - Introduction
https://www.karmamobility.co.uk/powered-wheelchairs-vs-manual-wheelchairs-pros-and-cons/
https://www.ridc.org.uk/sites/default/files/documents/pdfs/research-consultancy/Rica%20powered%20wheelchair%20user%20survey%20report.pdf

Figure 4.1.2B
https://www.bignerdranch.com/blog/bluetooth-low-energy-part-1/

Figure 6.2.5C
https://www.instructables.com/id/A-Simple-Android-UI-Joystick/

Section 9 - highway code
https://www.gov.uk/guidance/the-highway-code/rules-for-users-of-powered-wheelchairs-and-mobility-scooters-36-to-46/

14. Appendix

A- App Repository

https://github.com/09tangriro/WHEELS

B- Matlab Code

Download source code:
http://www.lizheyuan.com/archive/Remap%20Bristol%20Wheelchair%20source%20code/

```
%%%%%%%%%%%%%%%%% MATLAB Code for EE3 Remap Group Project %%%%%%%%%%%%%%%%%
%{
This code simulate the basic control theory and behaviour of a two-wheel
 drived wheelchair using a two-axis joystick input.

If not using a joystick to run the program:
Change parameters in "Parameters for Testing " to change the preset
values of joystick

If using a joystick to run the program:
```

```
In order to use a joystick to successfully run this code:
1. the "Simulink 3D Animation" toolbox need to be installed
2. A joystick device must be plugged into your PC and turned on

In this code:
"Throttle" stands for the command that is given to motor controllers
"Synchronized Throttle" means the average of left and right throttle
"Differential Throttle" means the deviation from the Synchronized Throttle
This also applies to Synchronized/Differential Velocity, Acceleration and
Jerk etc.

The speed of this program may be quite slow (the refresh frequency of
figures may be low) depending on your pc hardware. In reality, there will
be no figure plots and it should be much faster on a microcontroller.

This code was tested and worked fine in MATLAB version R2018b, running in
earlier versions may have problems displaying the figures.

To kill the program, move your cursor into the command window below and
click and press Ctrl+C
%}


% clear command window, workspace and figure windows
clc; close all; clear

%%%%%%%%%%%%%%%%%%%%%%% Joustick and Buttons Input %%%%%%%%%%%%%%%%%%%%%%%
x_channel = 1;          % x-axis of joystick
y_channel = 2;          % y-axis of joystick
button1 = 1;            % button 1
button2 = 2;            % button 2
button3 = 3;            % button 3


%%%%%%%%%%%%%%%%%%%%%%% Parameters Settings for Tuning %%%%%%%%%%%%%%%%%%%%%%%
ttl_max = 1;        % (0~1) maximum standardized throttle for each motor
ttld_ratio = 0.3;   % (0~1) ratio of differential to max throttle
ttl_left_gain = 1;  % (>0) additional gain for left motor, default = 1
ttl_right_gain = 1; % (>0) additional gain for right motor, default = 1
t_stat_max = 1;     % (>=0) delay of braking after stationary (in second)

x_inverted = 1;         % (-1 or 1) -1 = inverted, 1 = non-inverted
y_inverted = -1;        % (-1 or 1) -1 = inverted, 1 = non-inverted

x_linearity = 1;        % (>0) linearity, default = 1 (linear)
x_sensitivity = 1;      % (>0) sensitivity/overall gain, default = 1
x_deadzone = 0.15;      % (0~1) due to non-zero inaccuracy of the joystick

y_linearity = 1;        % (>0) linearity, default = 1 (linear)
y_sensitivity = 1;      % (>0) sensitivity/overall gain, default = 1
y_deadzone = 0.15;      % (0~1) due to non-zero inaccuracy of the joystick

js_limit = 0.8;   % (>0) limit of change per second in ttls (sync jerk)
jd_limit = 0.6;   % (>0) limit of change per second in ttld (diff jerk)


%%%%%%%%%%%%%%%%%%%%%% Parameters for Velocity Simulation %%%%%%%%%%%%%%%%%%%%%%
% assign certain parameters
accel_gain = 2;         % (>0) throttle to acceleration gain
d = 0.6;                % (>0 m) perpendicular distance between two wheels
v_max = 9;              % (>0 m/s) maximum tangential speed of wheels
ideal_sim = 1;          % (1 = on, 0 = off) whether use ideal simulation
time_constant = 0.5;    % (>0) larger = slower to get to the expected value
display_data = 0;       % (1 = on, 0 = off) whether display data on figures

% initialization of certain parameters
```

```
vs = 0;         % synchronized speed
vd = 0;         % differential speed
vs_1 = 0;       % synchronized speed (delayed)
vd_1 = 0;       % differential speed (delayed)
as = 0;         % synchronized acceleration
ad = 0;         % differential accelertaion
as_1 = 0;       % synchronized acceleration (delayed)
ad_1 = 0;       % differential acceleration (delayed)

sys = tf(1, [time_constant 1]);      % system model for velocity decay


%%%%%%%%%%%%%%%%%%%%%% Parameters for Wheelchair Plot %%%%%%%%%%%%%%%%%%%%%%
wchair_plot = 1;        % (0,1) whether plot the wheelchair as simulation
wchair_plot_size = 10;  % (>0) the size of the wheelchair in simulation

x = 0;                  % initial x-coordinate of wheelchair
y = 0;                  % initial y-coordinate of wheelchair
theta = pi/2;           % initial direction/heading of wheelchair (North)

dir_ind_size = wchair_plot_size*0.3;     % size of indicator in front
Lwl_ind_size = wchair_plot_size*0.15;    % size of left wheel indicator
Rwl_ind_size = wchair_plot_size*0.15;    % size of right wheel indicator


%%%%%%%%%%%%%%%%%%%%%%%%%% Parameters for Testing %%%%%%%%%%%%%%%%%%%%%%%%%%
% whether use joystick to control wheelchair in real time
% or use preset joystick values to control
js_control = 0;     % (1 = on, 0 = off), default = 0;
t_test = 8;        % (>0) time for the testing
t_sum = 0;          % sum of dt or simply a stopwatch start from 0

if js_control == 1
    % create a joystick handle
    ID = 1;
    joystick=vrjoystick(ID);
else
    t1 = 1;         % first change of joystick values (in second)
    t2 = 5;         % second change of joystick values (in second)

    % Edit here to change preset values
    jx0 = 0;        % initial joystick x-axis value
    jy0 = 0;        % initial joystick y-axis value
    jx1 = 0.8;      % first joystick x-axis value
    jy1 = 1;        % first joystick y-axis value
    jx2 = 0;        % second joystick x-axis value
    jy2 = 0;        % second joystick y-axis value

    i = 1;          % main loop iteration number
end


%%%%%%%%%%%%%%%% Other Parameters (Preferably Not to Change) %%%%%%%%%%%%%%%%
ttls_max = ttl_max*(1-ttld_ratio);  % maximum synchronized throttle
ttld_max = ttl_max*ttld_ratio;      % maximum differential throttle

vs_max = v_max*(1-ttld_ratio);      % maximum synchronized velocity
vd_max = v_max*ttld_ratio;          % maximum differential velocity

ttls_out = 0;       % output synchronized throttle
ttld_out = 0;       % output differential throttle

brakes = 1;         % (1 = on, 0 = off), brakes, default = 1
t_stat = 0;         % stopwatch in EABS, time since stationary
stat = 0;           % stationary flag (0 = moving, 1 = stationary)
```

```
but1 = 0;           % (0,1) initial button 1 value
but1_1 = 0;         % button 1 delayed by one iteration
but2 = 0;           % (0,1) initial button 2 value
but2_1 = 0;         % button 2 delayed by one iteration
but3 = 0;           % (0,1) initial button 3 value
but3_1 = 0;         % button 3 delayed by one iteration

jlmt_ctrl = 1;      % (1 = on, 0 = off), Jerk Limiter switch, default = 1
crs_ctrl = 0;       % (1 = on, 0 = off), cruise control switch, default = 0
ttls_crs = 0;       % cruise control throttle value
eabs_ctrl = 1;      % (1 = on, 0 = off), EABS switch, default = 1

% edge coordinates of deadzone square
x_ddz = [-x_deadzone x_deadzone x_deadzone -x_deadzone -x_deadzone];
y_ddz = [y_deadzone y_deadzone -y_deadzone -y_deadzone y_deadzone];


%%%%%%%%%%%% Calculate and Plot the Joystick Adjustment Curve %%%%%%%%%%%%
k = 1;      % iteration count for the loop below
for j = 0:0.01:1
    % very similar to the deadzone elimination section in the main loop
    j_sign = sign(j);

    if abs(j) < x_deadzone
        jx_plot(k)=0;
    else
        jx_plot(k)=(abs(j)-x_deadzone)/(1-x_deadzone);
        jx_plot(k)=x_sensitivity*j_sign*jx_plot(k)^x_linearity;
        jx_plot(k) = max(min(jx_plot(k), 1), -1);   % limit jx whithin -1~1
    end

    if abs(j) < y_deadzone
        jy_plot(k)=0;
    else
        jy_plot(k)=(abs(j)-y_deadzone)/(1-y_deadzone);
        jy_plot(k)=y_sensitivity*j_sign*jy_plot(k)^y_linearity;
        jy_plot(k) = max(min(jy_plot(k), 1), -1);   % limit jy whithin -1~1
    end

    h(k) = j;
    k = k+1;
end

% Plot the joystick control curve
f1 = figure(1);
f1.Name = 'Joystick Characteristic Curve';
f1.Units = 'normalized';
f1.OuterPosition = [0.2 0.65 0.3 0.3];  % Adjust figure size and position

subplot(1,2,1)
hold on
plot(h,h,'--')
plot(h,jx_plot)
hold off
axis square
title('x-axis')
xlabel('Input')
ylabel('Output')
legend('Before Adj','After Adj','location','northwest')
grid on

subplot(1,2,2)
hold on
plot(h,h,'--')
plot(h,jy_plot)
hold off
```

```
axis square
title('y-axis')
xlabel('Input')
ylabel('Output')
legend('Before Adj','After Adj','location','northwest')
grid on

% create a figure window and specify its relative size
f2 = figure(2);
f2.Name = 'Joystick Display and Wheelchair Motion Plot';
f2.Units = 'normalized';
f2.OuterPosition = [0.5 0.5 0.5 0.5];


%%%%%%%%%%%%%%%%%%%%% Main loop (loop until ctrl+c) %%%%%%%%%%%%%%%%%%%%%
tic     % need this for initializing the stopwatch
while t_sum < t_test
    % Store previous button values
    but1_1 = but1;
    but2_1 = but2;
    but3_1 = but3;

    if js_control == 1
        % read joystick input
        jx = x_inverted*axis(joystick, x_channel);  % joystick x-axis
        jy = y_inverted*axis(joystick, y_channel);  % joystick y-axis
        but1 = button(joystick, button1);       % button 1
        but2 = button(joystick, button2);       % button 2
        but3 = button(joystick, button3);       % button 3
    else
        % set joystick input to preset values
        if t_sum > t1           % first change
            if t_sum > t2       % second change
                jx = jx2;
                jy = jy2;
            else
                jx = jx1;
                jy = jy1;
            end
        else
            jx = jx0;
            jy = jy0;
        end
    end


    %%%%%%%%%%%%%%%%%%%%%%%% Adjust Joystick Data %%%%%%%%%%%%%%%%%%%%%%%%
    % extract the sign of joystick data
    jx_sign = sign(jx);
    jy_sign = sign(jy);

    if abs(jx) < x_deadzone
        jx_adj = 0;     % ignore the joystick input
    else    % x-axis adjustments
        jx_adj = (abs(jx)-x_deadzone)/(1-x_deadzone);
        jx_adj = x_sensitivity*jx_sign*jx_adj^x_linearity;
        jx_adj = max(min(jx_adj, 1), -1);   % limit jx value whithin -1~1
    end

    if abs(jy) < y_deadzone
        jy_adj = 0;     % ignore the joystick input
    else    % y-axis adjustments
        jy_adj = (abs(jy)-y_deadzone)/(1-y_deadzone);
        jy_adj = y_sensitivity*jy_sign*jy_adj^y_linearity;
        jy_adj = max(min(jy_adj, 1), -1);   % limit jy value whithin -1~1
    end
```

```
%%%%%%%%%%%%%%%% Calculate Reference Throttle Values %%%%%%%%%%%%%%%%%%
% throttle of left wheel motor before smoothing:
ttl_left = jy_adj*ttls_max + jx_adj*ttld_max;
% throttle of right wheel motor before smoothing:
ttl_right = jy_adj*ttls_max - jx_adj*ttld_max;

% convert to synchronized throttle
ttls_ref = (ttl_left+ttl_right)/2;
% convert to differential throttle
ttld_ref = (ttl_left-ttl_right)/2;

dt = toc;   % record the time for the previous iteration
tic         % restart the stopwatch


%%%%%%%%%%%%%%%%%%%%%%%%%%%% Jerk Limiter %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
dttls_max = js_limit*dt;    % maximum change in synchronized throttle
dttld_max = jd_limit*dt;    % maximum change in differential throttle

if but2 > but2_1            % when Jerk Limiter button is pressed
    jlmt_ctrl = 1 - jlmt_ctrl;  % toggle Jerk Limiter switch
end

if jlmt_ctrl == 1
    dttld = ttld_ref-ttld_out;      % change in diff throttle
    dttld = max(min(dttld, dttld_max), -dttld_max);     % capping
    ttld_out = ttld_out+dttld;  % output diff throttle

    dttls = ttls_ref-ttls_out;      % change in sync throttle
    dttls = max(min(dttls, dttls_max), -dttls_max);     % capping
    ttls_out = ttls_out+dttls;  % output sync throttle
else
    % no change
    ttld_out = ttld_ref;
    ttls_out = ttls_ref;
end


%%%%%%%%%%%%%%%%%%%%%%%% Cruise Control System %%%%%%%%%%%%%%%%%%%%%%%%%%
% detect whether cruise control button is pressed and toggle if so
if but1 > but1_1
    if (crs_ctrl == 0) && (ttls_out > 0)
        crs_ctrl = 1;           % off -> on
        ttls_crs = ttls_out;    % get current sync ttl value
    else
        crs_ctrl = 0;           % on -> off
        ttls_crs = 0;           % clear sync ttl value
    end
end

if crs_ctrl == 1
    if ttls_ref >= 0            % only when moving forward
        ttls_out = ttls_crs;    % replace output sync ttl value
    else                        % cancel if moving backward
        crs_ctrl = 0;           % cancel cruise control
        ttls_crs = 0;           % clear sync ttl value
    end
end


%%%%%%%%%%%%%%%%%% Electronic Auto-Braking System %%%%%%%%%%%%%%%%%%%%%%
if but3 > but3_1            % when EABS button is pressed
    eabs_ctrl = 1 - eabs_ctrl;      % toggle switch
    if eabs_ctrl == 0
```

```
        % brakes disengaged
        brakes = 0;
    end
end

% time since stationary
if (ttls_out == 0) && (ttld_out == 0)
    if stat == 0
        t_stat = t_stat + dt;   % start stopwatch and accumulate dt
        if t_stat >= t_stat_max
            stat = 1;   % is t_stat seconds after stationary
        end
    end
else
    stat = 0;       % reset flag
    t_stat = 0;     % reset stopwatch
end

if eabs_ctrl == 1
    % Brakes are on after t_stat_max seconds since no output throttles
    if stat == 1
        % brakes engaged
        brakes = 1;
    else
        % brakes disengaged
        brakes = 0;
    end
end


%%%%%%%%%%%% Output Joystick Values and Output Throttles %%%%%%%%%%%%
% visualize ttld_out and ttls_out as joystick values (not necessary)
jx_out = ttld_out/ttld_max;
jy_out = ttls_out/ttls_max;

% update throttle of left wheel motor after smoothing:
ttl_left = (ttls_out + ttld_out)*ttl_left_gain;
% update throttle of right wheel motor after smoothing:
ttl_right = (ttls_out - ttld_out)*ttl_right_gain;


%%%%%%%%%%%%%%%%%%%%%%%% Simulate Velovity %%%%%%%%%%%%%%%%%%%%%%%%
if ideal_sim == 1
    % limit vs and vd by their corresponding maximum values
    % vs and vd will remain constant if output js value is zero
    vs = max(min(vs+ttls_out*accel_gain*dt, vs_max), -vs_max);
    vd = max(min(vd+ttld_out*accel_gain*dt, vd_max), -vd_max);
else
    syz = c2d(sys,dt);  % convert continuous system to discrete system
    [num,den] = tfdata(syz,'v'); % extract numerators and denominators

    % vs and vd are direct decaying functions of output js values
    % in order to simulate the realistic velocity decrease
    vs = round(num(2)*jy_out*vs_max-den(2)*vs_1, 4);
    vd = round(num(2)*jx_out*vd_max-den(2)*vd_1, 4);
end

% update the tangential speed of left and right wheels
v_left = vs + vd;
v_right = vs - vd;

as = (vs-vs_1)/dt;       % synchronized acceleration
ad = (vd-vd_1)/dt;       % differential acceleration
js = (as-as_1)/dt;       % synchronized jerk
jd = (ad-ad_1)/dt;       % differential jerk
```

```
% Delay by one iteration
vs_1 = vs;
vd_1 = vd;
as_1 = as;
ad_1 = ad;


%%%%%%%%%%%%%%%%%%% Plotting Points and Display Data %%%%%%%%%%%%%%%%%%
figure(2)
if wchair_plot == 1     % whether start the subplot
    subplot(1,2,1)
end

% plot deadzone circle and origin
plot(x_ddz,y_ddz,'r')
hold on
plot(0,0,'+')

% plot the position of the joysticks before adjustments
p1 = plot(jx, jy,'.r', 'MarkerSize', 50);

% plot the position of the joysticks after adjustments
p2 = plot(jx_out, jy_out,'.b', 'MarkerSize', 30);

% plot the position of simulated velocities
p3 = plot(vd/vd_max, vs/vs_max,'.g', 'MarkerSize', 20);

str={['dt = ' num2str(dt,'%.4f')]};
text(-0.9, 0, str)

if display_data == 1
    title('Original js (red), Adjusted js (blue) and Velocity (green)')
    xlabel('Position/Velocity x')
    ylabel('Position/Velocity y')

    %{'Raw js','Adjusted js','velocity'},'Location','northwest')
    % Display the js value, output throttle
    str={['jx adj = ' num2str(jx_adj,'%.2f')], ...
        ['jy adj = ' num2str(jy_adj,'%.2f')], ...
        ['ttl left = ' num2str(ttl_left,'%.2f')], ...
        ['ttl right = ' num2str(ttl_right,'%.2f')], ...
        ['t stat = ' num2str(t_stat,'%.2f')], ...
        ['brakes = ' num2str(brakes)]};
    text(0.4, 0.6, str)

    % Display the button, cruise ttl and dt
    str={['button 1 = ' num2str(but1)], ...
        ['jlmt ctrl= ' num2str(jlmt_ctrl)], ...
        ['crs ctrl= ' num2str(crs_ctrl)], ...
        ['ttls crs= ' num2str(ttls_crs,'%.2f')], ...
        ['eabs ctrl= ' num2str(eabs_ctrl)]};
    text(-0.9, 0.6, str)

    % Display simulated velocities
    str={['v left = ' num2str(v_left,'%.2f')], ...
        ['v right = ' num2str(v_right,'%.2f')], ...
        ['vs = ' num2str(vs,'%.2f')], ...
        ['vd = ' num2str(vd,'%.2f')], ...
        ['w = ' num2str(vd/d,'%.2f')]};
    text(0.4, -0.6, str)

    % Display simulated accelerations and jerks
    str={['as = ' num2str(as,'%.2f')], ...
        ['ad = ' num2str(ad,'%.2f')], ...
        ['alpha = ' num2str(ad/d,'%.2f')], ...
        ['js = ' num2str(js,'%.2f')], ...
```

```
        ['jd = ' num2str(jd,'%.2f')]};
    text(-0.9, -0.6, str)
end

hold off
axis([-1 1 -1 1])   % assign the axis limits
axis square         % axes aspect ration is 1:1


%%%%%%%%%%%%%%%%%%%%%% Wheelchair Motion Plot %%%%%%%%%%%%%%%%%%%%%%
if wchair_plot == 1
    % calculate the position of wheelchair
    theta = theta - vd/d*dt;        % new direction
    x = x + vs*cos(theta)*dt;       % new x-coordinate
    y = y + vs*sin(theta)*dt;       % new y-coordinate

    % calculate the coordinates of indicators
    xa = x + dir_ind_size*cos(theta);
    ya = y + dir_ind_size*sin(theta);
    xb = x + Lwl_ind_size*cos(theta+pi/2);
    yb = y + Lwl_ind_size*sin(theta+pi/2);
    xc = x + Rwl_ind_size*cos(theta-pi/2);
    yc = y + Rwl_ind_size*sin(theta-pi/2);

    % plot the vehicle
    subplot(1,2,2)
    plot(x, y,'.r', 'MarkerSize', 100)
    hold on
    plot(xa, ya,'.b', 'MarkerSize', 50)
    plot(xb, yb,'.g', 'MarkerSize', 40)
    plot(xc, yc,'.g', 'MarkerSize', 40)
    hold off
    axis([-wchair_plot_size wchair_plot_size ...
        -wchair_plot_size wchair_plot_size])
    axis square         % axes aspect ration is 1:1

    if display_data == 1
        title('Wheelchair Motion Plot (Blue point = front)')
        xlabel('Position x')
        ylabel('Position y')
    end
end


%%%%%%%%%%%%%%%%%%%%%%%%%%%% Store Data %%%%%%%%%%%%%%%%%%%%%%%%%%%%
if js_control ~= 1
    jx_test(i) = jx; %#ok<*SAGROW>
    jy_test(i) = jy;

    ttls_out_test(i) = ttls_out;
    ttld_out_test(i) = ttld_out;
    ttl_left_test(i) = ttl_left;
    ttl_right_test(i) = ttl_right;

    vs_test(i) = vs;
    vd_test(i) = vd;
    v_left_test(i) = v_left;
    v_right_test(i) = v_right;

    as_test(i) = as;
    ad_test(i) = ad;
    js_test(i) = js;
    jd_test(i) = jd;

    t(i) = t_sum;
    t_sum = t_sum+dt;
```

```
        i = i+1;
    end

    % should not pause when the time for each iteration is non-zero
    %pause(t)
end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Plot Test Data %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% The below will not be executed unless js_control = 0 (off)
close(f2)

f3 = figure(3);
f3.Name = 'Left Wheel and Right Wheel';
f3.Units = 'normalized';
f3.OuterPosition = [0.65 0.05 0.25 0.9];

subplot(3,1,1)
hold on
plot(t, jx_test)
plot(t, jy_test)
ylim([-1.1 1.1])
hold off
title('Joystick Input')
xlabel('Time (s)')
ylabel('Value')
legend('x-axis','y-axis','location','Southeast')
grid on

subplot(3,1,2)
hold on
plot(t, ttl_left_test)
plot(t, ttl_right_test)
ylim([-1.1 1.1])
hold off
title('Output Throttle')
xlabel('Time (s)')
ylabel('Value')
legend('Left Throttle','Right Throttle','location','Southeast')
grid on

subplot(3,1,3)
hold on
plot(t, v_left_test)
plot(t, v_right_test)
ylim([-v_max v_max])
hold off
title('Simulated Velocity')
xlabel('Time (s)')
ylabel('Tangential Velocity (m/s)')
legend('Left Wheel','Right Wheel','location','Southeast')
grid on


f4 = figure(4);
f4.Name = 'Synchronized and Differential Velocity, Acceleration and Jerk';
f4.Units = 'normalized';
f4.OuterPosition = [0.05 0.05 0.6 0.6];

subplot(2,2,1)
hold on
plot(t, jy_test, '--')
plot(t, ttls_out_test, 'k')
ylim([-1.1 1.1])
xlabel('Time (s)')
ylabel('Value')
```

```
yyaxis right
plot(t, vs_test)
ylim([-vs_max vs_max])
hold off
title('Synchronized Throttle and Linear Velocity')
ylabel('Linear Velocity (m/s)')
legend('Joystick y-axis','Sync Throttle','Linear Velocity', ...
    'location','Southeast')
grid on

subplot(2,2,2)
hold on
plot(t, jx_test, '--')
plot(t, ttld_out_test, 'k')
ylim([-1.1 1.1])
xlabel('Time (s)')
ylabel('Value')
yyaxis right
plot(t, vd_test/d)
ylim([-vd_max/d vd_max/d])
hold off
title('Differential Throttle and Angular Velocity')
ylabel('Angular Velocity (rad/s)')
legend('Joystick x-axis','Diff Throttle','Angular Velocity', ...
    'location','Southeast')
grid on

subplot(2,2,3)
hold on
plot(t, as_test, 'k')
xlabel('Time (s)')
ylabel('Linear Acceleration (m/s^2)')
yyaxis right
plot(t, js_test)
hold off
title('Linear Acceleration and Jerk')
ylabel('Linear Jerk (m/s^3)')
legend('Linear Accel','Linear Jerk','location','Best')
grid on

subplot(2,2,4)
hold on
plot(t, ad_test/d, 'k')
xlabel('Time (s)')
ylabel('Angular Acceleration (rad/s^2)')
yyaxis right
plot(t, jd_test/d)
hold off
title('Angular Acceleration and Jerk')
ylabel('Angular Jerk (rad/s^3)')
legend('Angular Accel','Angular Jerk','location','Best')
grid on


%%%%%%%%%%%%% Thank you for scrolling all the way down here! %%%%%%%%%%%%%
```

C- Arduino Code

Download source code:
http://www.lizheyuan.com/archive/Remap%20Bristol%20Wheelchair%20source%20code/

```
/****************************** EE3 Remap Group Project ******************************/
/* This open-source code realizes basic control theory to a two-wheel drived
 * wheelchair using a two-axis joystick input, enabling several advanced features.
 *
 ************************************* NOTICE *************************************
 * WHEN COMPILING AND UPLOADING THE PROGRAM TO ARDUINO, ANY CONNECTIONS TO DIGITAL
 * PIN 1(RX) SHOULD BE REMOVED TO PREVENT THE UPLOADING PROGRESS BEING STUCKED.
 *
 * This program initially runs at Arduino Uno, due to lack of pins, the three pins for
 * buttons are assigned to be analog pins instead of digital pins due to lack of digital
 * I/O pins on the board.
 *
 * If "serial_display" is turned on, the digital pin 0 and 1 would be unusable.
 */

// These libraries are necessary when using nRF8001 Bluetooth module
#include <SPI.h>
#include "Adafruit_BLE_UART.h"
// The link to download "Adafruit_BLE_UART.h" library is:
// https://github.com/adafruit/Adafruit_nRF8001/archive/master.zip


/********************************** Pin Assignment ***********************************/
#define x_channel A1        // input analogue pin for reading joystick input
#define y_channel A0        // input analogue pin for reading joystick input
#define button1 A2          // button 1 pin number (Cruise Control)
#define button2 A3          // button 2 pin number (EABS)
#define button3 A4          // button 3 pin number (Jerk Limiter)
#define led1 0              // LED indicator pin number (Cruise Control)
#define led2 1              // LED indicator pin number (Jerk Limiter)
#define led3 3              // LED indicator pin number (EABS)
#define led4 4              // LED indicator pin number (Brakes)
#define PWM_left 5          // output digital pin (PWM) for controlling left motor
#define PWM_right 6         // output digital pin (PWM) for controlling left motor
#define rev_left 7          // pin number for reverse (FW) of left motor
#define rev_right 8         // pin number for reverse (FW) of right motor


/************************** Parameters Settings for Tuning ***************************/
#define ttl_max 1           // (0~1) maximum standardized throttle for each motor
#define ttld_ratio 0.3      // (0~1) ratio of differential to max throttle
#define ttl_left_gain 1     // (>0) additional gain for left motor, default = 1
#define ttl_right_gain 1    // (>0) additional gain for right motor, default = 1
#define t_stat_max 1        // (>=0) delay of braking after stationary (in second)
#define update_freq 20      // (Hz) frequency that the program updates PWM and reads input
#define serial_display 0    // (1 = on, 0 = off) whether display info through serial port
                            // default = 0, since digital pins 0 and 1 are used

#define x_inverted 1        // (-1 or 1) -1 = inverted, 1 = non-inverted
#define y_inverted 1        // (-1 or 1) -1 = inverted, 1 = non-inverted

#define x_linearity 1       // (>0) linearity, default = 1 (linear)
#define x_sensitivity 1     // (>0) sensitivity/overall gain, default = 1
#define x_deadzone 0.1      // (0~1) due to non-zero inaccuracy of the joystick

#define y_linearity 1       // (>0) linearity, default = 1 (linear)
#define y_sensitivity 1     // (>0) sensitivity/overall gain, default = 1
```

```
#define y_deadzone 0.1      // (0~1) due to non-zero inaccuracy of the joystick

#define js_limit 0.4        // limit of change per second in ttls (sync jerk)
#define jd_limit 0.5        // limit of change per second in ttld (diff jerk)


/***************** Other Parameters Settings and Variables Declarations ***************/
#define ADAFRUITBLE_RDY 2   // This should be an interrupt pin, on Uno thats #2 or #3
#define ADAFRUITBLE_RST 9   // connects to SPI Chip Select pin, can be changed to any pin
#define ADAFRUITBLE_REQ 10  // resets the board when start up, can be changed to any pin

double ttls_max = ttl_max*(1-ttld_ratio);   // maximum synchronized throttle
double ttld_max = ttl_max*ttld_ratio;       // maximum differential throttle

double ttls_out = 0;        // output synchronized throttle
double ttld_out = 0;        // output differential throttle
double t_stat = 0;          // stopwatch in EABS, time since stationary
int stat = 0;               // stationary flag (0 = moving, 1 = stationary)
int brakes = 1;             // (1 = on, 0 = off), brakes for both wheels, default = 1

int but1 = 0, but1_1 = 0;   // (0,1) initial button 1 value and delayed value
int but2 = 0, but2_1 = 0;   // (0,1) initial button 2 value and delayed value
int but3 = 0, but3_1 = 0;   // (0,1) initial button 3 value and delayed value

int jlmt_ctrl = 1;          // (1 = on, 0 = off), Jerk Limiter switch, default = 1
int crs_ctrl = 0;           // (1 = on, 0 = off), cruise control switch, default = 0
int eabs_ctrl = 1;          // (1 = on, 0 = off), EABS switch, default = 1
double ttls_crs = 0;        // cruise control throttle value

int jx_sign, jy_sign;
double jx, jy, jx_adj, jy_adj;
double ttls_ref, ttld_ref, dttls, dttld, ttl_left, ttl_right, dttls_max, dttld_max;
double T = 1000/update_freq;    // (in ms) convert update frequency into time period
double dt, t_tmp;
unsigned long t_start, t_end;   // start and stop time for the stopwatch

String message_x = "";
String message_y = "";
boolean comma = false;
boolean inputData = false;
boolean reading = true;

// Extract the sign of a value
template <typename T> int sign(T val) {
    return (val>T(0))-(val<T(0));
}

Adafruit_BLE_UART BTLEserial = Adafruit_BLE_UART(ADAFRUITBLE_REQ, ADAFRUITBLE_RDY,
ADAFRUITBLE_RST);

void display_info();

void setup() {
    // assign left and right motor direction and digital PWM pins to be output pins
    pinMode(rev_left, OUTPUT);
    pinMode(rev_right, OUTPUT);
    pinMode(PWM_left, OUTPUT);
    pinMode(PWM_right, OUTPUT);

    // assign digital pins for LED indicators
    pinMode(led1, OUTPUT);
    pinMode(led2, OUTPUT);
    pinMode(led3, OUTPUT);
    pinMode(led4, OUTPUT);

    if(serial_display == 1){
```

```
        // Inizialize Serial
        Serial.begin(9600);
    }

    BTLEserial.begin();
}
aci_evt_opcode_t laststatus = ACI_EVT_DISCONNECTED;


/************************************ Main Loop ************************************/
void loop(){
    // measure the time consumed for current literation
    t_end = millis();
    dt = t_end-t_start;
    t_tmp = T-dt;
    if(t_tmp > 0){
        delay(t_tmp);
        dt = T/1000;
    }else dt = dt/1000;
    t_start = millis();

    // Store previous button values
    but1_1 = but1;
    but2_1 = but2;
    but3_1 = but3;


    /************************** Read Joystick Input Data ****************************/
    // Tell the nRF8001 to do whatever it should be working on
    BTLEserial.pollACI();

    // Ask what is our current status
    aci_evt_opcode_t status = BTLEserial.getState();

    // If the status changed....
    if (status != laststatus){
        if(serial_display == 1){
            // print it out!
            if (status == ACI_EVT_DEVICE_STARTED){
                Serial.println(F("* Advertising started"));
            }
            if (status == ACI_EVT_CONNECTED){
                Serial.println(F("* Connected!"));
            }
            if (status == ACI_EVT_DISCONNECTED){
                Serial.println(F("* Disconnected or advertising timed out"));
            }
        }
        // set the last status change to this one
        laststatus = status;
    }

    // Read joystick and button data from app
    if(status == ACI_EVT_CONNECTED){

            comma = false;
            message_x = "";
            message_y = "";
            while(BTLEserial.available()){
                inputData = true;
                char c = BTLEserial.read();
                //Serial.print(c);
                if(c == ',') comma = true;
                if(c != ',' && comma == false) message_x.concat(c);
                if(c != ',' && comma == true) message_y.concat(c);
                if(c == 'D'){
```

```
                if(serial_display == 1) Serial.println(c);
                reading = false;
            }

            if(c == 'C'){       // Cruise Control
                if(serial_display == 1) Serial.println(c);
                but1 = 1;
                ttls_crs = 0.6; // set cruise ttls 60% if using joystick in app
            }
            else but1 = 0;

            if(c == 'B'){       // EABS
                if(serial_display == 1) Serial.println(c);
                but3 = 1;
            }
            else but3 = 0;

            if(c == 'J'){       // Jerk Limiter
                if(serial_display == 1) Serial.println(c);
                but2 = 1;
            }
            else but2 = 0;
        }

        if(inputData == true){
            jx = message_x.toDouble()/100;
            jy = message_y.toDouble()/100;
            inputData = false;
        }


    if(reading == false){
        //String s = getData();
        //uint8_t sendbuffer[20];
        //s.getBytes(sendbuffer, 20);
        //char sendbuffersize = min(20, s.length());

        if(serial_display == 1){
            Serial.println("Writing!");
            Serial.print(F("\n* Sending -> \""));
            //Serial.print((char *)sendbuffer);
            Serial.println("\"");
        }
        //BTLEserial.write(sendbuffer, sendbuffersize);
    }
}

// Read joystick and button data from physical joystick and buttons
else{
    // Read joystick data
    jx = analogRead(x_channel);
    jy = analogRead(y_channel);

    // read button values, use analog input pins for buttons to save digital pins
    but1 = analogRead(button1);
    but2 = analogRead(button2);
    but3 = analogRead(button3);

    // convert jx an jy into the range -1~1
    jx = x_inverted*(jx-511)/511;
    jy = y_inverted*(jy-511)/511;

    // convert analog value into digital value
    but1 = (but1 > 511) ? 1 : 0;
    but2 = (but2 > 511) ? 1 : 0;
    but3 = (but3 > 511) ? 1 : 0;
```

```
}


/*************************** Adjust Joystick Data ******************************/
// extract the sign of joystick data for both axes
jx_sign = sign(jx);
jy_sign = sign(jy);

if(abs(jx) < x_deadzone) jx_adj = 0;    // eliminate x-axis deadzone
else{
    // x-axis adjustments
    jx_adj = (abs(jx)-x_deadzone)/(1-x_deadzone);
    jx_adj = pow(x_sensitivity*jx_sign*jx_adj, x_linearity);
    jx_adj = max(min(jx_adj, 1), -1);   // limit jx value whithin -1~1
}

if(abs(jy) < y_deadzone) jy_adj = 0;    // eliminate y-axis deadzone
else{
    // y-axis adjustments
    jy_adj = (abs(jy)-jy_sign*y_deadzone)/(1-y_deadzone);
    jy_adj = pow(y_sensitivity*jy_sign*jy_adj, y_linearity);
    jy_adj = max(min(jy_adj, 1), -1);   // limit jy value whithin -1~1
}


/********************** Calculate Reference Throttle Values ***********************/
// throttle of left wheel motor before smoothing:
ttl_left = jy_adj*ttls_max + jx_adj*ttld_max;
// throttle of right wheel motor before smoothing:
ttl_right = jy_adj*ttls_max - jx_adj*ttld_max;

ttls_ref = (ttl_left+ttl_right)/2;    // convert to synchronized throttle
ttld_ref = (ttl_left-ttl_right)/2;    // convert to differential throttle


/******************************** Jerk Limiter *********************************/
dttls_max = js_limit*dt;    // maximum change in synchronized throttle
dttld_max = jd_limit*dt;    // maximum change in differential throttle

if(but2 > but2_1){          // when Jerk Limiter button is pressed
    jlmt_ctrl = 1 - jlmt_ctrl;      // toggle Jerk Limiter switch
}

if(jlmt_ctrl == 1){
    dttld = ttld_ref-ttld_out;    // change in diff throttle
    dttld = max(min(dttld, dttld_max), -dttld_max);   // capping
    ttld_out = ttld_out+dttld;    // output diff throttle

    dttls = ttls_ref-ttls_out;    // change in sync throttle
    dttls = max(min(dttls, dttls_max), -dttls_max);   // capping
    ttls_out = ttls_out+dttls;    // output sync throttle
}
else{
    // no change
    ttld_out = ttld_ref;
    ttls_out = ttls_ref;
}


/************************** Cruise Control System ******************************/
// detect whether cruise control button is pressed and toggle if so
if(but1 > but1_1){
    if((crs_ctrl == 0) && (ttls_out > 0)){
        crs_ctrl = 1;           // off -> on

        // get current sync ttl value if using physical joystick
```

```
        if(ttls_crs == 0) ttls_crs = ttls_out;
    }
    else{
        crs_ctrl = 0;           // on -> off
        ttls_crs = 0;           // clear sync ttl value
    }
}

if(crs_ctrl == 1){
    if(ttls_ref >= 0){          // only when moving forward
        ttls_out = ttls_crs;    // replace output sync ttl value
    }
    else{                       // cancel if moving backward
        crs_ctrl = 0;           // cancel cruise control
        ttls_crs = 0;           // clear sync ttl value
    }
}


/************************ Electronic Auto-Braking System ************************/
if(but3 > but3_1){   // when EABS button is pressed
    eabs_ctrl = 1 - eabs_ctrl;        // toggle switch
    if(eabs_ctrl == 0) brakes = 0;    // brakes disengaged
}

// time since stationary
if((ttls_out == 0) && (ttld_out == 0)){
    if(stat == 0){
        t_stat = t_stat + dt;     // start stopwatch and accumulate dt
        if(t_stat >= t_stat_max) stat = 1;    // is t_stat seconds after stationary
    }
}
else{
    stat = 0;     // reset flag
    t_stat = 0;   // reset stopwatch
}

if(eabs_ctrl == 1){
    // Brakes are on after t_stat_max seconds since no output throttles
    if(stat == 1)   brakes = 1;     // brakes engaged
    else            brakes = 0;     // brakes disengaged
}


/************** Calculate Left and Right Throttle and Output to Pins **************/
// update throttle of left wheel motor after smoothing
ttl_left = (ttls_out + ttld_out)*ttl_left_gain;
// update throttle of right wheel motor after smoothing
ttl_right = (ttls_out - ttld_out)*ttl_right_gain;

// decide directions of motor rotations (send to FW pins in motor controllers)
if(ttl_left >= 0)   digitalWrite(rev_left, LOW);
else                digitalWrite(rev_left, HIGH);
if(ttl_right >= 0)  digitalWrite(rev_right, LOW);
else                digitalWrite(rev_right, HIGH);

// convert throttle values to PWM signals, LED signals and send to pins
analogWrite(PWM_left, abs(ttl_left)*255);     // PWM of left wheel
analogWrite(PWM_right, abs(ttl_right)*255);   // PWM of right wheel
digitalWrite(led1, brakes);         // LED indicator (brakes)
digitalWrite(led2, eabs_ctrl);      // LED indicator (EABS)
digitalWrite(led3, jlmt_ctrl);      // LED indicator (Jerk Limiter)
digitalWrite(led4, crs_ctrl);       // LED indicator (Cruise Control)


// Display Infomation
```

```
    if(serial_display == 1) display_info();
}


void display_info(){
    // Display some information through serial port the more info displayed
    // the slower the program will be, and pin 0 and 1 would be unusable
    String str = "";
    str = String(dt,3) + " "+String(ttl_left) + " "+String(ttl_right);
    str += " "+String(jx) + " "+String(jy);
    Serial.println(str);
}

String getData(){
    return "1";
}
```