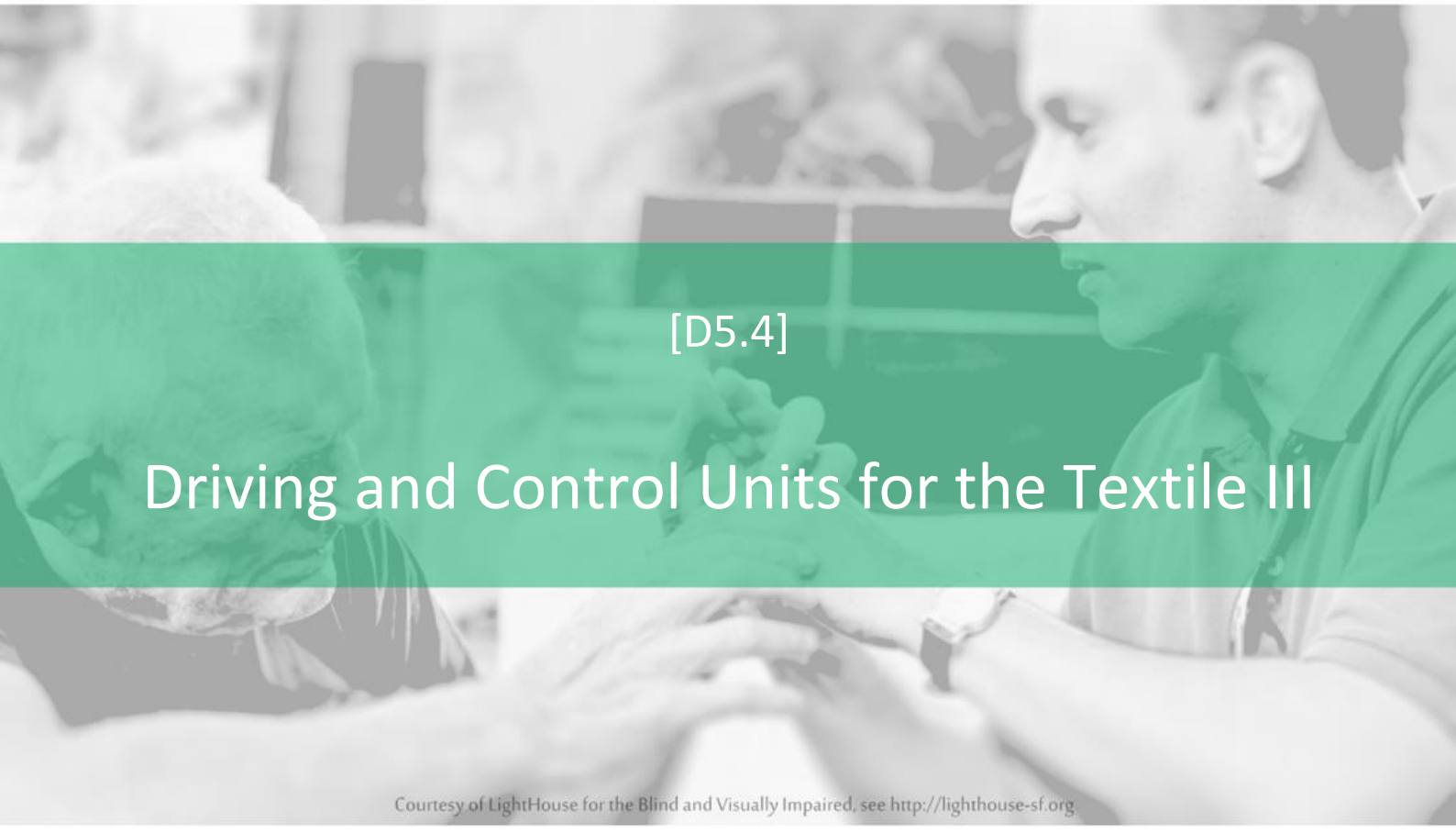
SUITCEYES

1 Jan 2018 - 31 Dec 2020

Smart, User-friendly, Interactive, Tactual, Cognition-Enhancer, that Yields Extended Sensosphere
Appropriating sensor technologies, machine learning, gamification and smart haptic interfaces



[D5.4]

Driving and Control Units for the Textile III

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 780814.

Dissemination level		
PU	PUBLIC, fully open, e.g. web	X
CO	CONFIDENTIAL, restricted under conditions set out in Model Grant Agreement	
CI	CLASSIFIED, information as referred to in Commission Decision 2001/844/EC.	

Deliverable Type		
R	Document, report (excluding the periodic and final reports)	
DEM	Demonstrator, pilot, prototype, plan designs	X
DEC	Websites, patents filing, press & media actions, videos, etc.	
OTHER	Software, technical diagram, etc.	

Deliverable Details	
Deliverable number	D5.4
Part of WP	5
Lead organisation	UNIVLEEDS
Lead member	Raymond Holt

Revision History			
V#	**Date**	**Description / Reason of change**	**Author / Org.**
v0.1	2/5/2019	Structure proposal	Raymond Holt/UNIVLEEDS
v0.2	17/5/2019	Initial Draft for Internal Review	Raymond Holt/UNIVLEEDS
v0.3	6/6/2019	Second Draft for Internal Review	Raymond Holt/UNIVLEEDS
v0.4	20/6/2019	Final Draft for Submission.	Raymond Holt/UNIVLEEDS
V1.0	26/6/2019	Final Draft Submitted to the EU	Thomas Bebis/HB

Authors	
Partner	**Name(s)**
UNIVLEEDS	Raymond Holt

Contributors		
Partner	**Contribution type**	**Name**
CERTH	Reviewer	Efstratios Kontopoulos
HSO	Reviewer	Ruben Gouveia

Glossary	
Abbr./ Acronym	**Meaning**
Actuator	An actuator is a component of a machine that is responsible for moving and controlling a mechanism or system.
Bidirectional Actuator	An actuator whose direction can be reversed – a Peltier module, DC motor or linear actuator.
Boolean direction signal	A digital signal to an H-bridge telling it whether to reverse the direction of the input signal based. Typically, the output is forward if the direction signal is “high” and reversed if the direction signal is “low”.
Breadboard	A solderless construction used for prototyping electronics and circuit design.
Closed Loop Control	Control with feedback – the input signal used to drive the actuator is adjusted based on the behavior of the actuator as detected through a sensor.
Duty Cycle	The proportion of a PWM cycle for which the connection is on. A duty cycle of 1 would be on all the time; a duty cycle of 0 would be off; a duty cycle of 0.5 would be on half the time, and so deliver half the power that the connection is capable of.
H-Bridge	An electronic circuit that can change the direction of an input signal (reversing positive and negative) in response to a control signal so that an actuator (such as a motor) can be driven in different directions.
HIPI	Haptic intelligent personalized interface – the goal of SUITCEYES and built as a textile structure.
Monodirectional Actuator	An actuator whose direction cannot be reversed – a solenoid or vibration motor, for example.
Open Loop Control	Control without feedback – the input signal used to drive an actuator is not adjusted based on the actual of an actuator.

Proportional-Integral-Derivative (PID) Control	Closed loop control in which the input signal used to drive an actuator is an “error signal”, which is determined by three factors: the difference between the desired and actual state of the actuator; the sum of this element over a period of time; and the rate of change of this signal.
Pulse-Width Modulation (PWM)	A method for adjusting the amount of power delivered through a digital connection, by rapidly cycling it on and off. By adjusting the proportion of time for which the connection is on during a given cycle (the duty cycle), the average power delivered can be adjusted, allowing more nuanced control through digital connections rather than a simple “on” or “off”.

Table of Contents

Contents

Executive Summary

This report outlines the third iteration of driving and control units for the SUITCEYES project. This builds upon the versions described in D5.2 and 5.3, to provide a system designed to facilitate prototype testing and psychophysical experiments. As it is intended to support experimentation and prototyping, the controller emphasizes modularity and flexibility, allowing different actuators to be attached to a set of channels. On an Arduino Nano, it can be configured to provide up to 5 PWM-enabled bidirectional open loop channels; up to 10 non-PWM monodirectional open loop channels; or a single closed loop channel with up to 4 open loop channels. In addition, the ability to capture and export sensor feedback and timing data are included, to support verification and testing. This report describes the working principles of the controllers and provides illustrative examples of their use with vibration motors and thermal displays.

1. Introduction and Aims

The purpose of this document is to accompany and describe the third iteration of the controller for Deliverable 5.4, which builds upon the versions described in Deliverables 5.2 and 5.3 and forms the basis of psychophysical experimentation in Work Package 6. Deliverable 5.2 introduced a hardware setup using simple off-the-shelf components for driving vibration motors, based around the use of the Arduino microcontroller. Deliverable 5.3 expanded upon this by introducing an approach to composing more complex signals across multiple actuators as a sequence of "frames", each frame specifying the signal to be sent to each actuator at a given time. Deliverable 5.4 expands upon this by introducing three key features:

- the ability to include a closed loop control channel, which uses sensor feedback to adjust the control signal being sent;
- the ability to set important controller parameters (number of channels, signal lengths, use of feedback) without having to adjust the code on the Arduino controller; and
- the inclusion of a python module to automate interaction with the controller, so that other users can more easily write their own programs to interact with it.

2. Controller Goals

The purpose of the controller is to provide a basis for testing textile prototypes developed in Work Package 5 and enable the carrying out of psychophysical experiments in Work Package 6. Accordingly, the third iteration of the controller has been designed with the following goals:

1) The controller will read a signal from a host computer via a serial that specifies a sequence of frames of varying duration, with each frame specifying an intensity to be displayed on each available channel;
2) The controller will drive an actuator with the requested intensity for the duration specified for each frame;
3) The controller will continue to use the hardware architecture established in Deliverables 5.2 and 5.3, based around an Arduino and the PmodHB3 H-bridge;
4) The controller will permit closed loop control on at least one channel;
5) To facilitate prototype testing and validation, the controller will be capable of capturing and reporting feedback from analogue sensors, and the actual timing of the signals sent to actuators; and
6) It should be possible to amend the number and type of actuators being used, and the haptic signal being displayed without the need to change the program on the Arduino microcontroller.

3. Controller Configuration

The third iteration of the controller continues to use the hardware architecture established in Deliverables 5.2 and 5.3, however, a number of key changes have been introduced:

1) To ensure accurate timing of signals, the controller now listens for the full signal, not just one frame at a time;
2) To increase the flexibility of signals being sent, each frame now specifies a duration as well as intensities for the available channels;
3) Intensities are now specified on a range of -100 to +100, with values less than zero being used for bidirectional control (on a monodirectional display, only the magnitude of the intensity is displayed);
4) Instead of adjusting the number and type of channels by uploading different "personalities" to the Arduino, the user can now specify these during a handshake process when the Arduino is first turned on; this means that users can make adjustments to the behaviour of the controller without needing to write new code to it;
5) Closed loop control has now been introduced, on a single channel, using an existing Arduino PID Library.

The following sections provide an overview of how the third iteration of the controller operates, and the hardware and software used to set it up.

3.1 Basic Principles of the Controller

As in previous iterations, the controller works as an interface between a Central Unit (which could be a desktop or laptop computer), and actuators used to display a haptic signal. Each channel drives one actuator, with a voltage that is varied depending on the requested intensity, by adjusting the duty cycle of a Pulse-Width Modulation (PWM) Signal. A range of actuators that will vary their response to a voltage signal can be driven in this way, allowing a variety of stimuli to be provided by the same controller software (for example vibration motors, solenoids and Peltier elements). In Open Loop control, the intensity represents the duty cycle of the PWM signal; if an actuator is bi-directional, negative intensities can be sent, causing the direction of the voltage applied through the H-bridge to reverse. In closed loop control, the intensity specified represents a target that the controller will try to bring the feedback reading to by adjusting the voltage across the actuator (for example – a target temperature to be achieved by heating or cooling a Peltier element).

To display a signal, a sequence of numbers is sent to the controller over a serial connection (such as a USB cable from a desktop computer), specifying the intensity for each channel for each frame, and the duration of each frame. For example, with a two-channel controller, a row comprising: "100, 0, 40, 0, 0, 80, 0, 80, 40" would lead to the following frames, illustrated in Figure 1:

Frame 1: Channel 1 at 100% duty cycle, Channel 2 at 0% duty cycle, for 40ms

Frame 2: Both Channels at 0% duty cycle for 80ms

Frame 3: Channel 2 at 80% duty cycle for 40ms.

Frame 1			Frame 2			Frame 3		
Channel 1	Channel 2	Duration	Channel 1	Channel 2	Duration	Channel 1	Channel 2	Duration
100	0	40	0	0	80	0	80	40

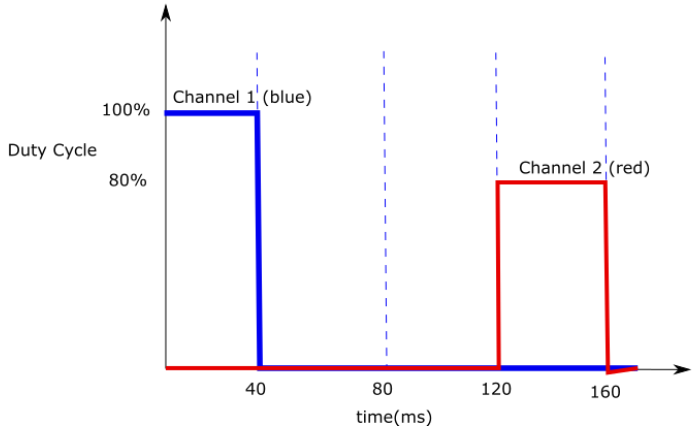


Figure 1: Example of how the controller will interpret "100, 0, 40, 0, 0, 80, 0, 80, 40" with two channels for a Simultaneous Channels signal.

This method is termed a **Simultaneous Channel Signal**, as all channels can be used simultaneously. The Central unit specifies how many frames the signal will be at the start of transmission, and the controller waits until it has received the full signal before displaying it. This can make lengthy signals with large numbers of channels laborious to write and slow to transmit. To address this, the controller also provides two options for faster transmission: **Sequential Channel Signals** (where a single specified channel is active in each frame) or **Single Channel Signals** (where the whole signal is displayed on a specified channel). In a Sequential Channel Signal, three numbers are specified for each frame: the channel to be used in that frame; the intensity to be displayed for that frame; and the duration of the frame. In a Single Channel Signal, the central unit specifies the channel to be used immediately after specifying signal length, and then sends two numbers per frame: intensity and duration. The downside to these approaches is that they allow only a single actuator to be used at once.

3.2 Hardware

As in Deliverables 5.2 and 5.3, the controllers have been designed to be implemented upon the Arduino platform, using a Digilent Pmod Hb3 H-bridge board where bidirectional control is required. The basic setup is illustrated in Figure 2, below. In order to allow bidirectional control, each channel requires two pins on the Arduino: a PWM pin for the control signal, and a second pin for determining direction, as illustrated in Figure 2. The Arduino can also gather data from analog sensors via its analog pins. If closed loop control is used, then a sensor must be attached for Channel 0.

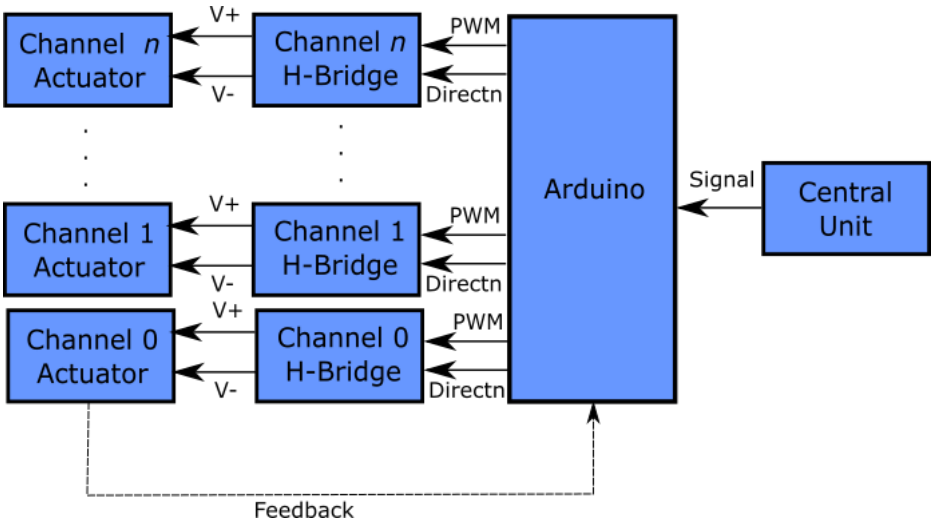


Figure 2: Haptic Display Unit Physical Architecture.

An Arduino Nano is able to provide five such channels and read from up to six sensors. If bidirectional control is not required, then it is possible to use the direction pins to control a further five channels, though these are non-PWM pins, so it is only possible to set the intensity at 0% or 100%, rather than setting it to some intensity in between (intensities of 50% or below are treated as 0%; intensities of 51% or above are treated as 100%). The H-bridges serve two purposes: to allow more current to the actuators than the Arduino could handle directly, and to allow the reversing of current direction for bidirectional display. If bidirectional display is not required, then the h-bridges can be replaced with transistors.

It is possible to use a second Arduino to gain a further five bidirectional channels, or ten monodirectional channels, or a second closed loop channel, in which case one must be set as the Master and the other as a Slave. In this situation, the Slave waits for a Trigger Signal from the Master to begin displaying its Signal, so that the two are synchronised.

3.3 Software

The code on the Arduino runs in four phases, as illustrated in Figure 3, below. In order to synchronize communication between the Arduino and the Central Unit, the Arduino specifies a packet size – the number of characters it expects to be used to communicate. By default, this is 4, to allow values of -100. To allow faster transmission of signals, however, the packet size can be reduced, at the expense of signal resolution. To allow larger values to be transmitted in smaller packet sizes, the Arduino also includes a multiplier which all intensities and durations are multiplied by. This defaults to 1 but can be set to higher values. For example, with a multiplier of 10, a packet size of 3 can transmit values of -100 to +100, but only in increments of 10. A multiplier of 100 would allow intensities of -100 to +100 at a packet size of 2, but only in increments of 100 (intensities of -100, 0, or +100; and durations of up to 9900ms).

Upon starting up, the Arduino attempts to initiate a handshake with the Central Unit over the serial connection. During this handshake, the Arduino tells the Central Unit the packet size it is expecting, and the Central Unit specifies to the Arduino a set of parameters determining how the

controller should behave. At the most basic, this covers how many channels and sensors should be used, and whether closed loop control should be used on Channel 0. In addition, parameters can be set to determine the tuning of closed loop control, and whether feedback or diagnostic data should be provided. Once the handshake is complete, the Arduino will wait to receive a signal. Once a signal has been received, it will display it, then offload any feedback or timing data it has gathered.

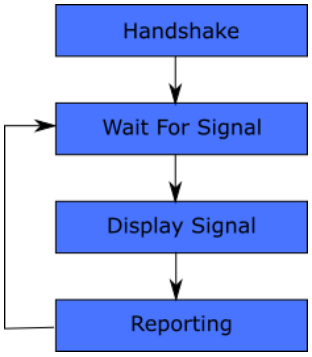


Figure 3: Arduino Process Stages.

A SUITCEYES Python module has been produced that can be imported into Python code and used to manage the communication properly. The Arduino code, the SUITCEYES python module and example code developed in Python for demonstration purposes can be found in the Appendix.

4. Controller Functions

The purpose of the controller is to enable prototype testing and psychophysical experimentation, and this is reflected in the functions it is required to perform. The principle function is to drive an actuator with a specified pattern of intensities. For many actuators, such as vibration motors, this requires open loop control; for others (for example a thermal display, where it would be desirable to control the actual temperature being provided) closed loop control based on feedback is required. Given current plans for experimentation in Work Package 6, the emphasis has been on the use of multiple vibration motors as actuators using open loop control, and a small number of channels for closed loop control, as it is not expected that large numbers of closed loop channels will be required. Finally, it is important for the purpose of conducting psychophysical experiments that the controller is able to report the actual timing with which signals are delivered (in case this deviates from the signal requested) and capture information from any sensors that are being used to monitor the signal delivered, even if these are not being used for closed loop control.

4.1 Open Loop Control

In open loop control, the controller sets the PWM signal and direction signals for each channel based purely upon the requested intensity, without knowledge of what the attached actuator is doing. It is possible to attach up to six sensors and have the Arduino report their readings throughout the signal, in order to monitor what an actuator is doing in practice, but the controller makes no use of this information. The controller defines a set of five “PWM Pins” intended to convey the intensity signal to the actuator through a PWM Signal as explained above, and five corresponding “Direction Pins”, intended to set the direction of the H-bridge based on whether the intensity requested is positive or

negative. If more than five channels are requested during the Handshake, then the controller automatically disables bidirectional control, and uses the Direction Pins as five additional channels, though – as noted above, these do not provide PWM functionality, so can only display intensities of 100% or 0%.

4.2 Closed Loop Control

One channel on the Arduino is capable of providing closed loop control, where the controller attempts to drive the actuator to a target value and calculates the PWM signal and Direction required to achieve this based on the readings from a sensor. This control is provided using the existing Arduino PID Library developed by Brett Beauregard[1], as this is readily available to all partners, free to use, and provides the functionality required without spending time generating a bespoke PID controller. As the controller is intended to be independent of any given actuator and sensor, the user must specify the tunings for the controller and all calculations are based on the Arduino's ADC readings, without any attempt to calibrate these to the actual properties the sensor is measuring. Hence, the user specifies an ADC reading which will be treated as a baseline intensity of zero and intensities that are requested in signals are then taken as being offset from this baseline – hence, if an offset of 510 is requested, then intensity requests of 0, -200, and 125 will result in target readings of 510, 310 and 635 respectively. Notice that in this case, it is possible to specify intensities greater than 100 and lower than -100.

To ensure the safe use of the controller, the user can also specify limits to the acceptable feedback reading – if the sensors reading is outside the specified range, the controller immediately turns off output on all channels, exits the signal and raises an error message to the Central Unit.

4.3 Reporting for Diagnostics and Psychophysical Experiments

To assist with testing the controllers and capturing their behaviour during psychophysical experiments (WP6), the controller records the start time of each Frame, and any feedback it has received. These can be turned off if not required, in order to speed up the time between signals.

5. Examples

This section provides example applications of the controller for ongoing vibrotactile experiments in Work Package 6, and potential future experiments with multiple vibration motors and thermal displays.

5.1 Vibrotactile Display

This section illustrates the setup of the controller for the numerosity and directionality experiments being carried out as part of Work Package 6. Both use a single vibration motor to deliver patterns of vibration, and therefore only require one channel. As vibration motors will provide the same vibration regardless of which direction current is applied to them there is no requirement for an H-bridge to be

[1] https://playground.arduino.cc/Code/PIDLibrary/

used, and in both cases the actuators are either on or off, so only intensities of 100 or 0 are used. The numerosity signals are focussed around grouping of pulses of vibrations, whereas the direction signal presents a continuous vibration with a 100ms gap whose position within the pulse indicates a direction. Full descriptions of these signals can be found in Deliverable 6.2, here the focus is on how they are delivered through the controller. Figure 4 provides an illustration of the voltage delivered by the controller and the vibrations presented As the voltage signal read across the motor is affected by interference from the motor itself and therefore very noisy, the voltage presented on the graphs below was measured without the vibration motor attached to illustrate how the voltage delivered fits with the requested signal. Vibrations were measured using when measured using an ADXL345 accelerometer sampled at 667Hz. It is worth noting that the vibration motor needs time to accelerate once the signal begins, and to decelerate once the signal ends, and so even though the voltage delivered by the controller fits well with the requested signal, the inertia of the actuator attached also affects the signal actually delivered.

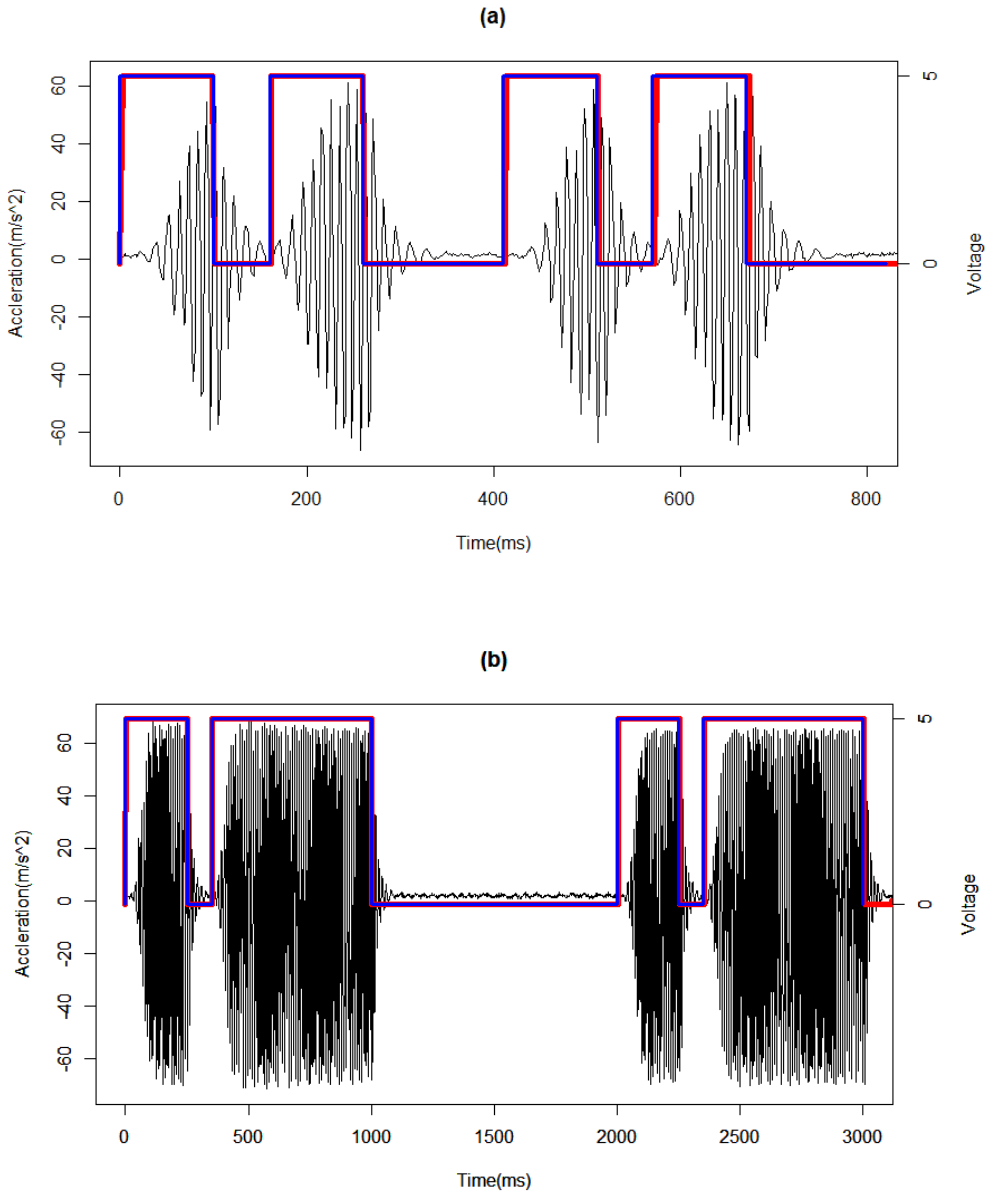


Figure 4: Example Signals for (a) Numerosity Stimuli and (b) Direction Stimuli. Black lines denote acceleration measured from the accelerometer; blue the voltage requested; and red the voltage delivered by the controller pin.

5.2 Thermal Display

Thermal displays are typically built around Peltier Elements, which move heat from one side of the element to the other when current is passed through them. These allow hot or cold stimuli to be delivered by the same element, by reversing the direction of current through it, and a variety of examples of their use in experiments can be found in the literature. It would be unrealistic to expect the generalised controller developed in Work Package 5 to operate on the same level as commercially available thermoelectric controllers, so precise temperature control is not expected. However, presenting basic thermal stimuli is still desirable in terms of exploring thermal perception, and control of a Peltier element presents an interesting case study for this deliverable, because it makes use of both the closed loop control elements that are included in the controller.

For this example, a 1.5A Peltier Element attached to a CPU Heat Sink (Rainbow Components, FHP08) was connected to the closed loop channel on the controller, with a thermistor connected to the first sensor input, and attached to the surface of the Peltier Element to measure its temperature. After some experimentation to establish appropriate tunings the closed loop controller was set with a proportional gain of 2 and a derivative gain of 8. Tuning always involves some trade-off between speed of response, steady-state error, and overshoot, and will inevitably need to be adapted to the specific element being used and the environment in which it is used. In this instance, it was found that introducing even a small amount of Integral gain resulted in problematic oscillations, so only proportional and derivative gain were used.

Figure 5 shows the varying responses of the system to temperature targets 5 °C or 2 °C from a 30°C baseline. Note that the slow response of the Peltier Element means that it can take 30s for the signal to reach its target value, so each example begins with a 30 second period to allow the controller to reach the target baseline. Steady state errors in the range of 0.5°C were encountered, suggesting that precise temperature control would be behind the capability of the controller, but it can deliver simple thermal stimuli.

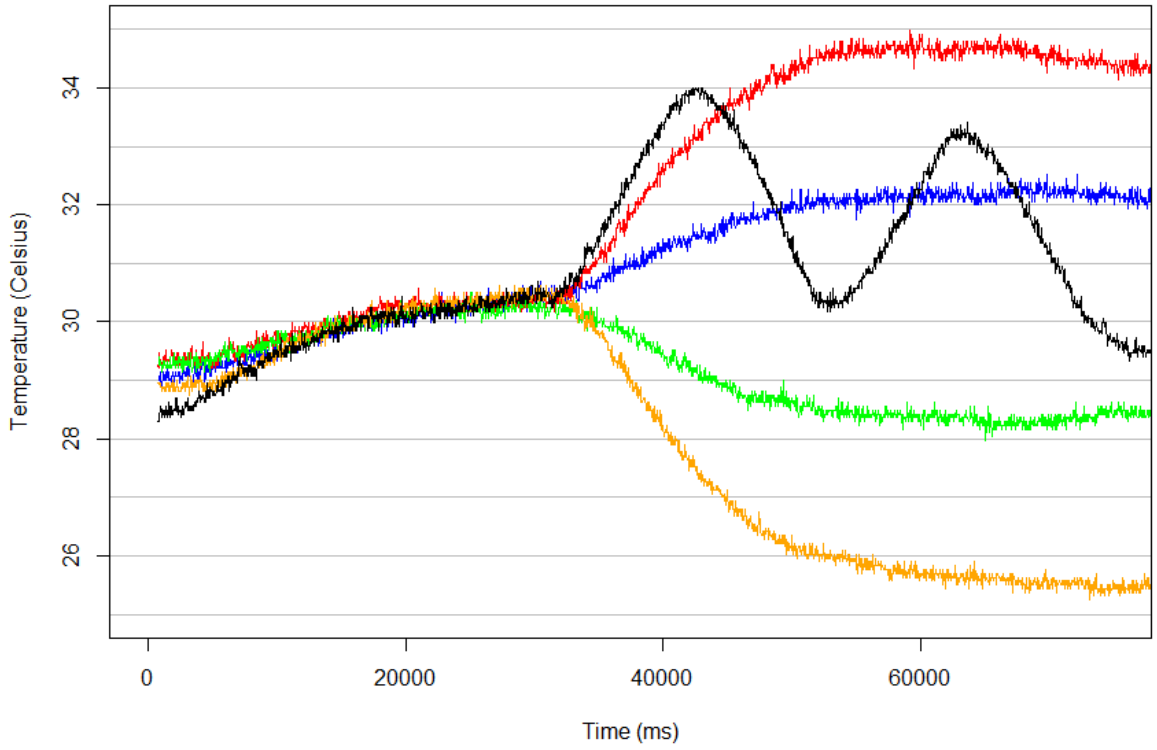


Figure 5: Example thermal responses from a baseline of 30°C aiming for steps of +5°C (red), +2°C (blue), -2°C (green) and -5°C (orange) and an alternating hot/cold signal.

6. Conclusions

This report has outlined the third iteration of the controllers for Work Package 5. These have been used to carry out vibrotactile experiments in Work Package 6 and include the functionality to deliver thermal and multi-point vibrotactile displays. The current design focuses upon the flexibility, adaptability and functionality required for psychophysical experiments, and it has been successfully used in this context.

The next step will be refining the controllers for use in the HIPI itself as part of Deliverable 5.8. This will involve adapting them to work with the sensor system developing in Work Package 4 in readiness for integration as part of Deliverable 4.3; streamlining the design to improve performance with the specific actuators selected at some cost to flexibility; and improving the speed of signal transmission either by serial communication to allow high-speed communication with the rest of the HIPI.

Appendix: Controller Code

This appendix contains code for the Arduino, the Python module developed to facilitate communication with the Arduino, and the illustration code used to generate the examples in Section 5. Both Python and Arduino code are maintained at the GitHub repository: https://github.com/rayjholt-leeds/SUITCEYES-WP5. The text provided here provides a snapshot of the controllers as they are at the time of writing.

A1.1 Arduino Controller Code

```
/*
*SUITCEYES Haptic Display Controller
*by Dr Raymond Holt, University of Leeds
*
*This code was developed as part of the SUITCEYES project (http://suitceyes.eu), and is owned by the SUITCEYES consortium.
*It may not be modified or redistributed without the consortium's express permission.
*
* This code makes use of the Arduino PID Library by Brett Beauregard, which is used under an MIT
*License as follows:
*
*Arduino PID Library v1.2
*
*Copyright (c) 2017 Brett Beauregard
*
*Permission is hereby granted, free of charge, to any person obtaining a copy of this software and
*associated documentation files (the "Software"), to deal in the Software without restriction,
*including without limitation the rights to use, copy, *modify, merge, publish, distribute, sublicense,
*and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do
*so, subject to the following conditions:
*
*The above copyright notice and this permission notice shall be included in all copies or substantial
*portions of the Software.
*
*THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED,
*INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR
*A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR
*COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN
*ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH
*THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
*
*The above Licence applies solely to the Arduino PID Library v1.2.1.
*
*OVERVIEW
```

```
*This sketch controls an Arduino Nano to drive a tactile display as part of the HIPI (Haptic Intelligent
*Personalised Interface) from the SUITCEYES project.
* It can provide up to 5 channels with PWM and bidirectional control; if only an on/off signal is
*required, and no direction, then the direction pins can be repurposed as a further five channels.
* The first channel (Channel 0) can also provide Closed Loop control.
* It uses a serial connection to receive data from the controller, which will send it a signal comprising
*a series of frames.
*
*/
#include <PID_v1.h> //Include the PID Library


//These are Variables are hard coded, and can only be set when running up to
bool Master = true;//records whether this is the master arduino that triggers the others.
byte PWMPins[] = {3,5,6,9,10}; //An array to store output pins for channels - you can change the
number of channels by adding to these, just make sure that you select PWM-enabled pins!
byte TriggerPin = 11; //Define pin to trigger data logging. Make sure this is not one of the pins
defined in PWMPins, so that the trigger doesn't inadvertantly operate an actuator.
byte dirPins[] = {2,4,7,8,12}; //An array to store direction pins for channels. Make sure that these are
NOT among the pins defined in PWMPins. This must be the same length as PWMPins.
byte SensorPins[] = {A0,A1,A2,A3,A4,A5};//Array to define feedback pins.
const byte Packet_Size = 4; //Sets the number of digits available for each intensity and duration
value. This can only be set on the Arduino, to avoid conflicts with the controller.
const byte Max_Signal_Length = 18; //This tells you the most frames that the Arduino can hold in
memory.
byte multiplier = 1; //This determines the multiples of ten used for signals. It allows more
bool DetailedTiming = true;
/*
 * NOTE: DetailedTiming is used for returning the behaviours of the output pins at the start of each
*frame.
* The corresponding arrays take up large amounts of memory, which restricts the maximum number
*of channels and frames you can have.
* This is intended for diagnostic purposes, and takes some effort to set up.
* In order to use it, "Detailed Timing" must be set to True and all instances of DirectionArray and
*OutputArray must be uncommented.
 *
 */


//The variables below can be set during the Handshake
int Signal_Length = Max_Signal_Length; //This tells you how many frames are available for each
signal. All signals must have the same number of frames. Defaults to maximum available
bool Verbose = true; //A variable to set whether the Arduino should return feedback to the
controller.
```

```
bool feedback = true;//Sets whether feedback should be gathered. This automatically enables the cutoff limits set by LowerCutOff and UpperCutOff. Must be enabled for ClosedLoop to work.
bool timing = true;//set whether timing data should be reported.
bool ClosedLoop = false;//Sets whether the controller should run in closed loop mode, using PID control. This is set for each channel.
bool Variable_Offset = false;//If true, this takes the initial reading on ADC0 as the ADC_Offset, and the value to return to.
bool Temperature_Reset = false;//If true, the system will try to get the signal to ADC_Offset between tests.
bool bidirectional = true;//If true, the system will use direction pins to set direction of the display. Defaults to on, unless more channels are requested than are available in PWM Pins.
bool CutOffActive = false;//
int LowerCutOff = 0;// Sets the ADC reading below which power will cut out and a warning light will be activated. Set to zero if no cutoff required. Not currently used.
int UpperCutOff = 1023;// Sets the ADC reading above which power will cut out and a warning light will be activated. Set to 1023 if no cutoff required. Not currently used.
int ADC_Offset = 550; //Set ADC reading equivalent to zero.
int Temp_threshold = 10; //Set how near to ADC_offset a reading needs to be before Reset_Temperature() is considered complete.
double Kp = 16;
double Ki = 0;
double Kd = 0;
const int PWMCount = (sizeof(PWMPins)/sizeof(byte)); //Identify number of PWMPins.
const int SensorCount = (sizeof(SensorPins)/sizeof(byte)); //Identify number of pins from SensorPins Array
const int dirPinCount = (sizeof(dirPins)/sizeof(byte));//Identify number of direction pins.
const int PinCount = PWMCount + dirPinCount; //The maximum number of channels is equal to the sum of PWM and direction pins - use this to reserve memory for the intensity signal.

int LEDPin =13;// defines the pin to which the LED is attached - 13 is the built-in LED on an Arduino Nano.
int8_t Intensity_Signal[PinCount][Max_Signal_Length]={};// 2D array of intensities for each channel for each frame - passed by controller.
int Duration_Signal[Max_Signal_Length]={}; //1D array of durations for each frame in ms - passed by controller from Haptic Library CSV. Note that all channels must have the same duration.
byte Current_Frame; //Variable to store current frame index
char Separator[] = ","; //variable to set separator character for feedback from Arduino.
String SignalEnd = "C,,,";
int Packet_Interval = ((PinCount+1)*(Packet_Size+1)); //Calcultes number of bytes required by each frame, based on Packet_Size.
long int TimeArray[Max_Signal_Length+1] = {}; //Array to store timing data from each signal.
//int OutputArray[][Max_Signal_Length+1] = {}; //Array to store displayed data from each signal. Must be uncommented for DetailedTiming.
```

```
//int DirectionArray[][Max_Signal_Length+1] = {};//Array to store displayed direction from each signal, or in Boolean mode, to store values displayed on non-PWM channels. Must be uncommented for DetailedTiming.
long int SignalStart; //Variable to store the start of signals for timing purposes.
int LastDirection[PinCount]; //Array to store the previous direction from each frame.
double Setpoint, Input, Output; //Initialise variables required by PID controller.
bool CutOff = false; //variable to store when cutoff has been activated.
int channels = PWMCount;//stores the number of Channels - defaults to number of PWM pins available.
int sensors = SensorCount;//Stores the number of Sensors.
PID PID1(&Input, &Output, &Setpoint,Kp, Ki, Kd, DIRECT);//Initialise PID.

/*
 * FUNCTIONS
 */

void ResetSignal(){//Fills the Intensity and Duration Signal Arrays with zeros, ready to receive next signal
  Serial.println("Resetting the old signal...");
  for (int Frame = 0; Frame<Max_Signal_Length; Frame++){
    Duration_Signal[Frame] = 0;
    for (int Pin = 0; Pin<PinCount; Pin++){
      Intensity_Signal[Pin][Frame] = 0;
    }
  }
}

int GetData(){
  while (Serial.available()<Packet_Size){//Do nothing while there is less serial data available than the Packet Size.
}
String Value = Serial.readStringUntil(',');
int number = Value.toInt();
if (Verbose){
Serial.print("Got: ");
Serial.println(number);
}
Serial.println(SignalEnd);
return number;
}

int GetPin(int PinSpecified){//Identify which Pin to use, based on Pin Specified, and the PWMPin and dirPin Arrays.
   int PinIdentified;
```

```
  if (PinSpecified < PWMCount){//If PinSpecified is less than the number of pins identified in the PWMPins array, look up the relevant value in PWMPins.
    PinIdentified = PWMPins[PinSpecified];
  }
  else {//If PinSpecified is larger than the number of pins identified, then we need to look the value up in the dirPins Array.
    PinIdentified = dirPins[(PinSpecified - PWMCount)];
  }
   return PinIdentified;
}

void Handshake(){
//Handshake with controller - send Packet Size required for serial communication, then read data in packets.
Serial.print("X:");
Serial.println(multiplier);
Serial.print("P:");
Serial.println(Packet_Size);
//controller now knows Packet_Size, so data should be sent in packet size chunks.
int VerboseWanted = GetData();
if (VerboseWanted == 1){Verbose = true;}
   else {Verbose = false;}
channels = GetData();//Get number of channels from Controller.
sensors = GetData();//Get number of sensors from Controller.
Packet_Interval = ((channels+1)*(Packet_Size+1));
int FeedbackWanted = GetData();
if (FeedbackWanted == 1){feedback = true;}
   else {feedback = false;}
int TimingWanted = GetData();
if (TimingWanted == 1){timing = true;}
   else {timing = false;}
int ClosedLoopWanted = GetData();
if (ClosedLoopWanted == 1){ClosedLoop = true;}
   else {ClosedLoop = false;}
Kp = GetData();
Ki = GetData();
Kd = GetData();
PID1.SetTunings(Kp, Ki, Kd);
ADC_Offset = GetData();
int VariableOffsetWanted = GetData();
if(VariableOffsetWanted == 1){Variable_Offset = true;}
   else{Variable_Offset = false;}
int CutOffWanted = GetData();
if(CutOffWanted == 1){CutOffActive = true;}
```

```
   else{CutOffActive = false;}
int LowerCutOff = GetData();// Sets the ADC reading below which power will cut out and a warning light will be activated. Set to zero if no cutoff required.
int UpperCutOff = GetData();// Sets the ADC reading above which power will cut out and a warning light will be activated. Set to 1023 if no cutoff required.
int Temp_threshold = GetData(); //Set how near to ADC_offset a reading needs to be before Reset_Temperature() is considered complete.
int TempResetWanted = GetData();
if (TempResetWanted ==1){Temperature_Reset = true;}
  else{Temperature_Reset = false;}
if (ClosedLoop){
      feedback = true;//forces feedback to be on if Closed Loop control is being used on at least one pin, even if it was declared "off" above.
      PID1.SetOutputLimits(-255,255);//Scale the output so it goes from -255 to 255.
      PID1.SetMode(AUTOMATIC);}//Start the PID controller, as closed loop feedback is being used.
if (channels > PinCount){
 Serial.print("E1");
 Serial.println(Separator);
 channels = PinCount;
}
Serial.print("Ch");
Serial.print(Separator);
Serial.println(channels); //tell Central Unit how many channels are available.
if (channels>PWMCount){//If more channels are requested than are available in PWMPins, switch Bidirectional mode off, so that direction pins can be used for intensity signals.
 bidirectional = false;
 Serial.print("E2");
 Serial.println(Separator);
}
Serial.print("OP");
Serial.print(Separator);
for (int Pin = 0; Pin < channels; Pin++) {
   int PinToUse = GetPin(Pin);
   Serial.print(PinToUse); //Iterate through each declared pin, and print its number.
   Serial.print(Separator);
   }
Serial.println();
if (bidirectional){
   Serial.print("DP");
   Serial.print(Separator);
   for (int Pin = 0; Pin < channels; Pin++) {
      Serial.print(dirPins[Pin]); //Iterate through each declared pin, and print its number.
      Serial.print(", ");
      }
```

```
   Serial.println();

   }
Serial.print("Se");
Serial.print(Separator);
Serial.println(sensors);
Serial.print("SP");
Serial.print(Separator);
for (int Sensor = 0; Sensor< sensors; Sensor++) {
   Serial.print(SensorPins[Sensor]); //Iterate through each declared pin, and print its number.
   Serial.print(Separator);;
   }
Serial.println();
Serial.print("PI");
Serial.println(Packet_Interval);
Serial.print("MaxSL");
Serial.println(Max_Signal_Length);
if (Verbose){Serial.print("V");}
   else {Serial.print("NV");}
Serial.println(Separator);
if (Master){Serial.print("M");}
   else {Serial.print("S");}
Serial.println(Separator);
if (feedback){Serial.print("FB");}
   else {Serial.print("NFB");}
Serial.println(Separator);
if (timing){Serial.print("Ti");}
   else {Serial.print("NT");}
Serial.println(Separator);
if (ClosedLoop){Serial.print("CL");}
Serial.println(Separator);
if (Variable_Offset){
  Serial.print("GADC");
  Serial.println(Separator);
  int CurrentValue = analogRead(0);
  for (int i=0; i<9; i++){ //Take 10 readings at 100ms intervals, and average to get ADC Offset
     CurrentValue = CurrentValue+analogRead(0);
  }
  float AverageValue = CurrentValue/10;
  ADC_Offset = int(AverageValue);
  Serial.print("ADCO");
  Serial.print(Separator);
  Serial.println(ADC_Offset);
}
```

```
if (ClosedLoop){//If closed loop control is active on Channel 1.
if(Temperature_Reset){
Reset_Temperature();}
}

Serial.println(SignalEnd);//Sent signal to end handshake.

}

int GetDirectChannel(){
  int ChannelToUse = GetData();
  SignalStart = millis();//Start Timer.
  return ChannelToUse;
}

void GetSignal(int ChannelToUse){
    bool Direct = true;
    bool Multichannel = false;
    bool DisplayNothing = false;
    if (ChannelToUse <=0){Direct = false;}//If the specified Channel is 0 or more, this is a direct signal.
    if (ChannelToUse == -2){//If the specified Channel is -2, this is a multichannel signal.
       Multichannel = true;
       //Serial.println("Multi");//If the specified Channel is zero, get the signal in the same way as
direct, but get two
    }
    if (ChannelToUse == -3) {//If the specified channel is -3, the signal should be displayed on another
Arduino, so this should display zero throughout the signal.
       Multichannel = true;
       DisplayNothing =true;
       //Serial.println("Multiblank");
    }
    if (ChannelToUse == -4) {//If the specified channel is -3, the signal should be displayed on another
Arduino, so this should display zero throughout the signal.
       DisplayNothing =true;
       //Serial.println("DirectBlank");
       Direct = true;
    }
    Signal_Length = GetData();//Get Signal length from Controller.
    int ChannelsToUse = channels;
    if (Direct){ChannelsToUse = 1; Serial.println("Setting Channels to Use to 1");}//In direct mode, set
channels to use to 1 - each frame has only an intensity, plus the duration.
    if (Multichannel){ChannelsToUse = 1;}//In multichannel direct mode, set channels to use to 2 -
each frame has a channel and an intensity and a
```

```
  Packet_Interval = ((ChannelsToUse+1)*(Packet_Size+1));
  if (Multichannel){((ChannelsToUse+2)*(Packet_Size+1));}
   while (Current_Frame < Signal_Length){//keep repeating this until all the frames have been read.
    if (Serial.available()<Packet_Interval){//keep coercing temperature while waiting for signal.
      if (ClosedLoop){
      if (Temperature_Reset){
      Reset_Temperature();}}
       }

       if (Serial.available()>= Packet_Interval){ //If there are more than "Packet Interval" characters
waiting, we know that at least one whole number is in the buffer.
       String Value;
       for (int Pin = 0; Pin < ChannelsToUse; Pin++) {
       int PinToUse = Pin;
       if (Direct){PinToUse = ChannelToUse;}
       if (Multichannel){//override the channel by reading the next integer
       Value = Serial.readStringUntil(',');
       PinToUse = Value.toInt();
       Serial.print("Channel: ");
       Serial.println(PinToUse);
       }
       Value = Serial.readStringUntil(','); // Read the next string until the next separator.
       if (DisplayNothing){
       Intensity_Signal[PinToUse][Current_Frame] = 0;
       }
       else{
       Intensity_Signal[PinToUse][Current_Frame] = (Value.toInt())*multiplier; // Get the next integer
from the buffer.
       }
       Serial.print("I");
       Serial.println(Intensity_Signal[PinToUse][Current_Frame]);
       if (Verbose){
        Serial.print("Sending");
        Serial.print(Pin+1);
        Serial.print("/");
        Serial.println(ChannelsToUse);
        }
       }
       Value = Serial.readStringUntil(',');// Get next integer from buffer
       Duration_Signal[Current_Frame] = (Value.toInt())*(multiplier);
       Serial.print("DS");
       Serial.println(Duration_Signal[Current_Frame]) ;

       if (Verbose) {
```

```
        Serial.print("CF");
        Serial.print(Current_Frame);
        Serial.print(", read:");
        for (int Pin = 0; Pin < ChannelsToUse; Pin++) {
          int PinToUse = Pin;
          if(Direct){PinToUse = ChannelToUse;}
          Serial.print(Intensity_Signal[PinToUse][Current_Frame]);
          Serial.print(Separator);
        }
        Serial.println(Duration_Signal[Current_Frame]);
      }
      Current_Frame++;

      if (ClosedLoop){
      if (Temperature_Reset){
      Reset_Temperature();}}//   coerce temperature towards startpoint

      Serial.println(F("C,,,"));
      }
          }
}

void SendFeedback(){
  long int Timer = millis() - SignalStart;
       Serial.print("F"); //use an F to indicate that the line is feedback.
       Serial.print(Separator);
       Serial.print(Timer);
       for (int Sensor = 0; Sensor < sensors; Sensor++){//get sensor data.
          int currentReading = analogRead(SensorPins[Sensor]);
          if (Sensor == 0){
             if (currentReading < LowerCutOff){ActivateCutOff(Sensor);}
             if (currentReading > UpperCutOff){ActivateCutOff(Sensor);}
          }
          Serial.print(Separator);
          Serial.print(currentReading);
          }
       Serial.println();

}

void PrintIntensityArray() {
  Serial.print("Printing IS");
   for (int i=0; i<Max_Signal_Length; i++){
      for (int j=0; j<PinCount; j++){
```

```
      Serial.print(Intensity_Signal[j][i]);
      Serial.print(Separator);}
      Serial.println(" ");
}
}

void PrintDurationArray() {
 Serial.print("Printing DS");
  for (int i=0; i<Max_Signal_Length; i++){
      Serial.print(Duration_Signal[i]);
      Serial.print(Separator);}
      Serial.println(" ");
}

void DisplaySignal(int ChannelToUse) {
 if (Master){digitalWrite(TriggerPin, HIGH);} //If in Master mode, send signal.
 else {
  while(digitalRead(TriggerPin)==LOW){SendFeedback();};//If in Slave mode, wait until the trigger
signal is received.
 }
 for (int Frame = 0; Frame<Signal_Length; Frame++){//iterate through each frame in turn, presenting
data.
       if (Verbose){
       Serial.print("start frame");
       Serial.println(Frame);
       }
       int ChannelsToUse = channels;
       for (int Pin = 0; Pin < ChannelsToUse; Pin++) {
         int PinSelected = Pin;
         int PinToUse = GetPin(PinSelected);
         int DisplayBytes; //A variable to store the bytes for display
         int DisplayDirection;//Variable to store Direction.

         /*
         if (DetailedTiming){//Please note - OutputArray and DirectionArray consume large amounts of
memory, so are best left commented out.
           if (Pin < PWMCount){
       OutputArray[Pin][Frame] = Intensity_Signal[Pin][Frame]; //Before doing anything, store the
requested intensity for timing data.
         }
         else {
          DirectionArray[Pin-PWMCount][Frame] = Intensity_Signal[Pin][Frame];//If this channel is a
dirPin, use Direction Array to store the data.
         }
```

```
        }
        */

        if (ClosedLoop && PinSelected==0){//if ClosedLoop control is active and this is the pin for Channel 1, then we need to do something a bit different.
          double TargetGiven = ((Intensity_Signal[Pin][Frame])/2)+ADC_Offset; //calculate the target setpoint  as a double: doing it as an integer seems to confuse the PID library.
          Setpoint = TargetGiven;//adjust setpoint. It *should* now stay at this level until the next frame, so there is no need to recalculate TargetGiven.
          CalculatePID(PinSelected);//Reads the current ADC value, and updates Output based on the PID.
          DisplayBytes = abs(Output); //It is convenient to record this so that PID Output can be fed back.
          //Serial.println("Time to check direction...");
          DisplayDirection = CalculateDirection(PinSelected, Output); //Set direction based on output, not intensity request.

          /*
          if (DetailedTiming){
          DirectionArray[Pin][Frame] = Setpoint; //If Closed Loop Control is being used, then Direction is meaningless, so SetPoint is returned instead.
          }
          */

      }
      else {//if closed loop isn't being used on this pin, then just scale the Intensity Signal directly and calcualte .
       DisplayBytes = CalculatePWM(Intensity_Signal[Pin][Frame]); //Calculate PWM value from intensity.
       if (bidirectional){//if the bidirectional mode is available, set the directional signal and store the direction data in DirectionArray.
          DisplayDirection = CalculateDirection(Pin,Intensity_Signal[Pin][Frame]);//Calculate Direction from intensity.

         /*
          if (DetailedTiming){
          DirectionArray[Pin][Frame] = DisplayDirection; //If Open Loop control is used, just direction will be returned on direction pins.
          }
          */

          LastDirection[Pin] = DisplayDirection; //Store LastDirection as an Array
          }
       }
```

```
      analogWrite(PinToUse,DisplayBytes); //write requested intensity to relevant output pin.
            }
      long int Timer = millis()-SignalStart;
      TimeArray[Frame] = Timer;
      int FrameStart = millis();
      int Frame_Timer = millis()-FrameStart;
       while (Frame_Timer < Duration_Signal[Frame]){
      Frame_Timer = millis()-FrameStart;//Update Frame Timer
      //Offload feedback to PC.
      if (feedback){
      SendFeedback();
        if (ClosedLoop){//If Closed Loop Control is active, carry out the PID calculation.
            CalculatePID(0);//determines the new Setpoint, and defines the Bytes as the initial error,
so that direction can be determined.
            CalculateDirection(0, Output); //Set direction based on output.
            analogWrite(PWMPins[0],abs(Output));//Set PWM based on Output.
          };

      if (CutOff){break;}//if CutOff has been activated, terminate frame.
      }
      if (CutOff){break;}//if CutOff has been activated, terminate signal.
      }
    }
      for (int Pin = 0; Pin < channels; Pin++) {
      int PinToUse = GetPin(Pin);
      analogWrite(PinToUse,0); //End Signal on each Pin - all signals should now turn off.
      }
      TimeArray[Signal_Length] = millis()-SignalStart;

      /*
      if (DetailedTiming){
          for (int Pin = 0; Pin < PWMCount; Pin++) {
            OutputArray[Pin][Signal_Length] = 0; //Write values of zero for end of Timing Data Array.
            DirectionArray[Pin][Signal_Length] = 0;
            }
      }
      */

    if (Master){digitalWrite(TriggerPin, LOW);}//Turn off Trigger Pin.
    if (Verbose){
    PrintIntensityArray();
    PrintDurationArray();
    }
```

```
      Serial.println("C, Signal Ends"); // - Send "C" to notify controller that signal has ended, and next signal can be displayed.
      ResetSignal();
      if (Verbose){
      PrintIntensityArray();
      PrintDurationArray();
      }
}

int CalculatePWM(int BytetoProcess) {
  int ByteConverted = BytetoProcess*2.55;
  int BytetoReturn = abs(ByteConverted);
  return BytetoReturn;
}

void Reset_Temperature() {//attempts to settle the temperature towards the ADC_Offset value. Only works on first channel at present/
   if (Temperature_Reset){
   Serial.println("Reset CL");
   int Diff; //Variable to store ADC difference.
   Setpoint = ADC_Offset;
   do {//Currently only set to work on the first SensorPin.
    int currentReading(analogRead(SensorPins[0]));
    Diff = abs(currentReading - ADC_Offset);
    CalculatePID(0);
    CalculateDirection(0, Output);//Set Direction.
    if (Verbose){
      Serial.print("Read ");
      Serial.println(currentReading); //Iterate through each declared pin, and print its number.
      Serial.print(", Diff ");
      Serial.println(Diff);
    }
    analogWrite(PWMPins[0],abs(Output));
    if (currentReading < LowerCutOff){ActivateCutOff(0);}//If the ADC reading is outside the acceptable limits,
    if (currentReading > UpperCutOff){ActivateCutOff(0);}
    delay(10);
    if (CutOff){break;}
    } while(Diff > Temp_threshold);
     }
}

void CalculatePID(int PinNumber){
        Input = analogRead(PinNumber);
```

```
        PID1.Compute();
        if (Verbose){//if in Verbose Mode, clarify output used.
        Serial.print(", SP: ");
        Serial.print(Setpoint);
        Serial.print(", OP:");
        Serial.println(Output);
        }
}

int CalculateDirection(int PinNumber, int BytetoProcess){

  int BytetoReturn;//declare local variable.

  //Establish whether signal is positive, negative or zero. If it's zero, then we can leave it as before, and BytetoReturn equals zero. Otherwise, we need to select an appropriate value to return (1 if positive, -1 if negative).
  if (BytetoProcess == 0){//if the Byte to process is zero, BytetoReturn is zero, and you don't need to do anything else.
    BytetoReturn = 0;
  }
  else if (BytetoProcess > 0){//If the byte to process is +ve, then set BytetoReturn to one.
    BytetoReturn = 1;
  }
  else {
     BytetoReturn = -1;//If the byte to process is neither +ve nor zero, it is negative, so set BytetoReturn to minus one.
  }

  if (BytetoReturn > 0){//If Byte to Process is the same as the last one,
    analogWrite(PWMPins[PinNumber], 0);//Write 0 to the current Pin, while things change
    //if (Verbose){Serial.println("Direction High");}
    delay(1);//1ms delay - to avoid shoot through. Not sure if this is necessary.
    digitalWrite(dirPins[PinNumber], HIGH);
  }
  else if (BytetoReturn < 0)  {
    analogWrite(PWMPins[PinNumber], 0);//Write 0 to the current Pin, while things change.
    //if (Verbose){Serial.println("Direction Low");}
    delay(1);//1ms delay - is this necessary? Ensures that PWM has set to zero before switching H-bridge, to avoid shoot-through.
    digitalWrite(dirPins[PinNumber], LOW);
 }

 return BytetoReturn;
}
```

```
void ActivateCutOff(int SensorID){//turns off all channels and exits signal if ADC readings are outside acceptable bounds.
    if (CutOffActive){
    for (int Pin = 0; Pin < channels; Pin++) {
      analogWrite(PWMPins[Pin], 0); //Iterate through each declared pin, and set its output to zero.
    }
    Serial.print("ADC");
    Serial.print(SensorPins[SensorID]);
    Serial.print(" Outside Cutoff.");
    CutOff = true;//Set CutOff to true.
    digitalWrite(LEDPin, HIGH); //turn on the LED
    }
}

void SendTimingData(){
       if (timing){
       for (int Report = -1; Report<Signal_Length+1; Report++){
       Serial.print("T");
       Serial.print(Separator);
       Serial.print(TimeArray[Report]);
       /*
       if (DetailedTiming){
       for (int Pin = 0; Pin < PWMCount; Pin++) {//Report Data Stored for Each Pin in Turn
       Serial.print(Separator);
       Serial.print(OutputArray[Pin][Report]);
       Serial.print(Separator);
       Serial.print(DirectionArray[Pin][Report]);
       }
       }
       */
       Serial.println(" "); //End the Line
      }
       }
      Serial.println("C, Signal Ends");
}
/*
 * MAIN PROGRAM
 */

void setup () {

if (Master){pinMode(TriggerPin,OUTPUT);} else {pinMode(TriggerPin,INPUT);}
pinMode(LEDPin, OUTPUT);
```

```
for (int Pin = 0; Pin < PinCount; Pin++) {
   pinMode(PWMPins[Pin], OUTPUT); //Iterate through each declared pin, and set it as an Output.
   pinMode(dirPins[Pin], OUTPUT);
   digitalWrite(PWMPins[Pin], LOW);//start with a signal of zero.
   digitalWrite(dirPins[Pin], LOW);
}

//Begin Serial Communication
Serial.begin(9600);
while (Serial.available()>0){
 Serial.read(); //Empty the serial buffer before starting.
}
Handshake();
}

void loop () {
//main loop
   digitalWrite(TriggerPin, LOW); //Turn TriggerPin off so that it doesn't trigger data logging until
signal begins.
   Current_Frame = 0;
   Setpoint = ADC_Offset;//Set setpoint to ADC offset, so that PID if active drives towards desired
starting temperature.
   CutOff = false;//reset cutoff signal.
   int DirectChannel = GetDirectChannel();
   GetSignal(DirectChannel);
   digitalWrite(TriggerPin, HIGH);//Send trigger signal to notify any data logger that recording should
start - the trouble is that this disconnects the two timers. They need to begin at the same time.
   //SignalStart = millis(); //Record start time of signal, to co-ordinate timer - comment this line out
for Timing Data to be based on signal request time, but be aware that this will put any other
Arduinos timers out of Sync.
   DisplaySignal(DirectChannel);
   SendTimingData();
   if (ClosedLoop){
      Reset_Temperature();}//   coerce temperature towards startpoint
   }
```

A1.2 SUITCEYES Python Module

```
##############################################################################################
#
# SUITCEYES Python Module v1.0
```

```
# by Dr Raymond Holt, University of Leeds
#
# This code was developed as part of the SUITCEYES project (http://suitceyes.eu), and is owned by
the SUITCEYES consortium.
# It may not be modified or redistributed without the consortium's express permission.
#
# modified 3rd May 2019
# by Raymond Holt
#
# Version History
# v0.1 - Raymond Holt (UNIVLEEDS) - 23rd April 2019 - Initial version created
# v1.0 - Raymond Holt (UNIVLEEDS) - 3rd May 2019 - First release.
# Overview
# This module contains the functions used to communicate with the SUITCEYES controllers
developed in WP5.
#
##############################################################################################
###########

def Handshake(Port, channels, sensors, Verbose = False, Feedback = False, Timing = False,
ClosedLoop = False, Kp=1, Ki = 0, Kd = 0, Offset = 510, VariableOffset = False, CutOff = False,
LowerCutOff = 0, UpperCutOff = 1023, ADCThreshold = 10, ADCReset = False):
   print("Beginning Handshake")
   print("Obtaining Packet Size from Arduino")
   packet_size_received = False
   while packet_size_received == False:
      if Port.inWaiting() > 0: #if data is waiting. First thing to identify is the Packet Size, since this is
needed to send information.
         status = Port.readline() #read the next line.
         print(status)
         character = status.split(":")
         if character[0] == "X":
            print("Signal values will be multiplied by " + str(character[1]))
         if character[0] == "P": #collect Packet Size from Arduino
            Packet_Size = int(character[1])
            packet_size_received = True
   print("Packet Size Received - Packet Size:" + str(Packet_Size))
   print("Setting Verbose:" + str(Verbose))
   if Verbose:
      TransmitValue(Port, Packet_Size, 1, Verbose)
   else:
      TransmitValue(Port, Packet_Size, 0, Verbose)
   print("Setting Channels: " + str(channels))
   TransmitValue(Port, Packet_Size, channels, Verbose)
```

```
print("Setting Sensors: " + str(sensors))
TransmitValue(Port, Packet_Size, sensors, Verbose)
print("Setting Feedback: " + str(Feedback))
if Feedback:
   TransmitValue(Port, Packet_Size, 1, Verbose)
else:
   TransmitValue(Port, Packet_Size, 0, Verbose)
print("Setting Timing: " + str(Timing))
if Timing:
   TransmitValue(Port, Packet_Size, 1, Verbose)
else:
   TransmitValue(Port, Packet_Size, 0, Verbose)
print ("Setting Closed Loop:" + str(ClosedLoop))
if ClosedLoop:
   TransmitValue(Port, Packet_Size, 1, Verbose)
else:
   TransmitValue(Port, Packet_Size, 0, Verbose)
print("Setting Proportional Gain")
TransmitValue(Port, Packet_Size, Kp, Verbose)
print("Setting Integral Gain")
TransmitValue(Port, Packet_Size, Ki, Verbose)
print("Setting Derivative Gain")
TransmitValue(Port, Packet_Size, Kd, Verbose)
print("Setting default ADC Offset")
TransmitValue(Port, Packet_Size, Offset, Verbose)
print("Setting Variable ADC Offset")
if VariableOffset:
   TransmitValue(Port, Packet_Size, 1, Verbose)
else:
   TransmitValue(Port, Packet_Size, 0, Verbose)
print("Setting Cutoff")
if CutOff:
   TransmitValue(Port, Packet_Size, 1, Verbose)
else:
   TransmitValue(Port, Packet_Size, 0, Verbose)
print("Setting LowerCutOff")
TransmitValue(Port, Packet_Size, LowerCutOff, Verbose)
print("Setting UpperCutoff")
TransmitValue(Port, Packet_Size, UpperCutOff, Verbose)
print("Setting ADC Threshold")
TransmitValue(Port, Packet_Size, ADCThreshold, Verbose)
print("Setting ADC Reset")
if ADCReset:
   TransmitValue(Port, Packet_Size, 1, Verbose)
```

```
    else:
        TransmitValue(Port, Packet_Size, 0, Verbose)
    print("Confirming settings from Arduino")
    ListenToController(Port, True)#Let Arduino Offload Summary Information.
    print("Handshake Complete")
    Port.flushInput()
    Port.flushOutput() #Clear Serial Buffer
    return Packet_Size

def MultiHandshake(Port, channels, sensors, Verbose = False, Feedback = False, Timing = False,
ClosedLoop = False):
    Packet_Size = Handshake(Port[0], channels[0], sensors[0], Verbose, Feedback, Timing, ClosedLoop)
#handshake with Arduino - send channels, sensors, etc and receive Packet_Size for future data
transmissions.
    CurrentController = 1
    while CurrentController<len(Port):
        Handshake(Port[CurrentController], channels[CurrentController], sensors[CurrentController],
Verbose, Feedback, Timing, ClosedLoop)
        CurrentController+=1
    return Packet_Size

def ListenToController(Port, Verbose = False):
    #Listen for the controller to confirm receipt of a signal (by writing a line beginning "C")
    #Any feedback from the controller (denoted by a line beginning "F") will be stored as an Array of
timestamps, and an Array of Measurements.

    Feedback_Array = [] #Create a blank array for returning timestamp and feedback data
    confirmed = False
    while confirmed == False: #Until the frame is confirmed, keep checking the serial buffer.
        if Port.inWaiting() > 0: # if data is waiting.
            status = Port.readline()
            if Verbose:
                print(status)
            character = status.split(",")
            if character[0] == "C":         #If the Arduino sends a line beginning C, the Arduino has
completed the task.
                confirmed = True
            if character[0] == "F": #if an 'F' is received, the line is feedback, so append it to the relevant
arrays.
                Sensor_Data = [] #Create blank array for storing the current feedback run
                for i in range(1,len(character),1):
                    Sensor_Data.append(int(character[i])) #Create array of feedback for current timestamp
and readings
                Feedback_Array.append(Sensor_Data)#append feedback array
```

```
        if character[0] == "T": #if an 'T' is received, the line is Timing Data, so append it to the relevant
arrays.
            Timing_Data = [] # create blank array for storing current timing
            for i in range(1,len(character),1):
                Timing_Data.append(int(character[i])) #Create array of feedback for current timestamp
and readings
            Feedback_Array.append(Timing_Data)#append feedback array
        if character[0] == "E1":
            print("Channels requested exceeds maximum for this Arduino. Setting channels to
maximum available.")
        if character[0] == "E2":
            print("Too many channels for bidirectional display - no direction control available, as
direction pins needed for intensity display")
        if character[0] == "E3":
            print("ADC reading on Channel 0 outside cutoff. Terminating signal.")
        if character[0] == "V":
            print("Arduino set to Verbose Mode")
        if character[0] == "NV":
            print("Arduino set to Silent Mode")
        if character[0] == "M":
            print("This Arduino is Master - it will provide a trigger signal to any Slave Arduinos")
        if character[0] == "S":
            print("This Arduino is a Slave - it will wait for a trigger signal before displaying the signal")
        if character[0] == "FB":
            print("Feedback enabled")
        if character[0] == "Ti":
            print("Timing Enabled")
        if character[0] == "CL":
            print("Closed Loop Control enabled on Channel 0")
        if character[0] == "GADC":
            print("Variable ADC offset enabled - getting offset")
        if character[0] == "ADCO":
            print("ADC Offset identified as: "+str(character[1]));
        if Verbose:
            if character[0] == "RT":
                print("Resetting Temperature")
            if character[0] == "RS":
                print("Resetting Signal")

    return Feedback_Array; #return the timestamp and feedback arrays

def TransmitValue(Port, Packet_Size, value, Verbose):
    value_to_send = str(value)#convert value to string
    string_to_send = "" #set the string for communication to be blank
```

```
    if value_to_send[0] == "-":
        value_to_send = value_to_send[1:] #remove the first character, which is the negative size.
        while len(value_to_send) <(Packet_Size-1): #if there are less than Packet Size - 1 characters in
the current signal.
            value_to_send = "0"+value_to_send
        value_to_send = "-" + value_to_send #re-append the negative sign.
    else:
        while len(value_to_send) <Packet_Size: #if there are less than three characters in the current
signal.
            value_to_send = "0"+value_to_send
    string_to_send = string_to_send + value_to_send +"," #append latest figure
    Port.write(string_to_send)
    if Verbose:
        print(string_to_send)
    ListenToController(Port, Verbose) # wait for Arduino to respond.

def MultiDirectSignal(Port, Packet_Size, SignalToSend, Verbose = False):
    TransmitValue(Port, Packet_Size, -2, Verbose) #transmits "-2" to the controller to tell it that this is
a multichannel direct signal.
    Sensor, TimingData = SendSignal(Port, 2, Packet_Size, SignalToSend, Verbose)
    return Sensor, TimingData

def DirectSignal(Port, channel, Packet_Size, SignalToSend, Verbose = False):
    TransmitValue(Port, Packet_Size, channel, Verbose) #transmits channel number to the controller
to tell it that this is a direct signal.
    Sensor, TimingData = SendSignal(Port, 1, Packet_Size, SignalToSend, Verbose)
    return Sensor, TimingData

def BlankDirectSignal(Port, channel, Packet_Size, SignalToSend, Verbose = False):
    TransmitValue(Port, Packet_Size, -4, Verbose) #transmits channel number to the controller to tell
it that this is a direct signal.
    Sensor, TimingData = SendSignal(Port, 1, Packet_Size, SignalToSend, Verbose)
    return Sensor, TimingData

def BlankMultiDirectSignal(Port, Packet_Size, SignalToSend, Verbose = False):
    TransmitValue(Port, Packet_Size, -3, Verbose) #transmits "-2" to the controller to tell it that this is
a multichannel direct signal.
    Sensor, TimingData = SendSignal(Port, 2, Packet_Size, SignalToSend, Verbose)
    return Sensor, TimingData

def TransmitSignal(Port, channels, Packet_Size, SignalToSend, Verbose = False):
    TransmitValue(Port, Packet_Size, -1, Verbose) #transmits "-1" to the controller to tell it that this is
a standard signal.
    Sensor, TimingData = SendSignal(Port, channels, Packet_Size, SignalToSend, Verbose)
```

```
    return Sensor, TimingData

def SendSignal(Port, channels, Packet_Size, SignalToSend, Verbose = False, Direct = False):
    current_frame = 0 #integer to keep track of length of signal
    interval = channels+1 #Set interval for reading from signal library: each frame has one entry per
channel, plus one for duration.
    signal_length = len(SignalToSend)/interval #Calculated number of frames in signal.
    TransmitValue(Port, Packet_Size, signal_length, Verbose)
    while current_frame < signal_length:#for each frame in the signal.
        frame_signal_start = current_frame * interval #calculate the start point of the current frame
        frame_signal_end = frame_signal_start + interval#calculate end of frame - note that python will
return up to, but not including, this.
        current_signal = SignalToSend[frame_signal_start:frame_signal_end] #get the next interval of
readings
        if Verbose:
            print("Signal Length"+str(signal_length))
            print("Current Frame"+str(current_frame))
            print("Time to send the frame")
        string_to_send = "" #set the string for communication to be a blank.
        for channel in range (0, interval, 1):
            value_to_send = str(current_signal[channel])
            if value_to_send[0] == "-":
                value_to_send = value_to_send[1:] #remove the first character, which is the negative size.
                while len(value_to_send) <(Packet_Size-1): #if there are less than Packet Size - 1 characters
in the current signal.
                    value_to_send = "0"+value_to_send
                value_to_send = "-" + value_to_send #re-append the negative sign.
            else:
                while len(value_to_send) <Packet_Size: #if there are less than three characters in the
current signal.
                    value_to_send = "0"+value_to_send
            string_to_send = string_to_send + value_to_send +"," #append latest figure
        Port.write(string_to_send)
        if Verbose:
            print(SignalToSend)
            print(signal_length)
            print("I sent: " + string_to_send)
            print("Frame Transmitted: Waiting for response")

        ListenToController(Port, Verbose) #Wait until Arduino confirms signal.
        current_frame += 1
    Sensor = ListenToController(Port, Verbose)
    TimingData = ListenToController(Port,Verbose)
    return Sensor, TimingData;
```

```
def SendDirectSignal(Port, channels, Packet_Size, SignalToSend, Verbose = False):
    current_frame = 0 #integer to keep track of length of signal
    interval = channels+1 #Set interval for reading from signal library: each frame has one entry per channel, plus one for duration.
    signal_length = len(SignalToSend)/interval #Calculated number of frames in signal.
    TransmitValue(Port, Packet_Size, signal_length, Verbose)
    while current_frame < signal_length:#for each frame in the signal.
        frame_signal_start = current_frame * interval #calculate the start point of the current frame
        frame_signal_end = frame_signal_start + interval#calculate end of frame - note that python will return up to, but not including, this.
        current_signal = SignalToSend[frame_signal_start:frame_signal_end] #get the next interval of readings
        if Verbose:
            print("Signal Length"+str(signal_length))
            print("Current Frame"+str(current_frame))
            print("Time to send the frame")
        string_to_send = "" #set the string for communication to be a blank.
        for channel in range (0, interval, 1):
            value_to_send = str(current_signal[channel])
            if value_to_send[0] == "-":
                value_to_send = value_to_send[1:] #remove the first character, which is the negative size.
                while len(value_to_send) <(Packet_Size-1): #if there are less than Packet Size - 1 characters in the current signal.
                    value_to_send = "0"+value_to_send
                value_to_send = "-" + value_to_send #re-append the negative sign.
            else:
                while len(value_to_send) <Packet_Size: #if there are less than three characters in the current signal.
                    value_to_send = "0"+value_to_send
            string_to_send = string_to_send + value_to_send +"," #append latest figure
        Port.write(string_to_send)
        if Verbose:
            print(SignalToSend)
            print(signal_length)
            print("I sent: " + string_to_send)
            print("Frame Transmitted: Waiting for response")

        ListenToController(Port) #Wait until Arduino confirms signal.
        current_frame += 1
    Sensor = ListenToController(Port, Verbose)
    TimingData = ListenToController(Port,Verbose)
    return Sensor, TimingData;
```

```
def WriteFeedback(FeedbackFile, FeedbackArray, Verbose = False):
   #Takes an array of feedback or timing data provided by the Arduino and writes it to the specified
File with in CSV format
   with open(FeedbackFile, "wb") as feedback:
      columnTitleRow = "Time(s),"
      for i in range(1,(len(FeedbackArray[0])),1):
         columnTitleRow = columnTitleRow + "ADC" + str(i) +"," #use this to set column headings for
the output CSV
      columnTitleRow = columnTitleRow + "\n"
      feedback.write(columnTitleRow) #write column headings to CSV
      working_row = 0 #set row counter to co-ordinate writing to CSV
      while working_row < len(FeedbackArray):
         current_row = ""
         working_array = FeedbackArray[working_row]
         for i in range(0,len(working_array),1):
            current_row = current_row + str(working_array[i]) +"," #Write data from Feedback Array
         current_row = current_row + "\n"
         if Verbose:
            print(current_row)
         feedback.write(current_row) #write to CSV
         working_row += 1 #increment working row.

def WriteDetailedTimingData(FeedbackFile, FeedbackArray, SignalArray, Verbose = False, Graphable
= False):
   #Takes an array of timing data provided by the Arduino and writes the first column to a CSV file.

   with open(FeedbackFile, "wb") as feedback:
      columnTitleRow = "Time(s),"
      columnTitleRow = columnTitleRow + "\n"
      feedback.write(columnTitleRow) #write column headings to CSV
      working_row = 0 #set row counter to co-ordinate writing to CSV

      while working_row < (len(FeedbackArray)-1):
         working_array = FeedbackArray[working_row]
         next_array = FeedbackArray[(working_row+1)]
         current_row = str(working_array[0])+"," # start the current row with the timestamp.
         next_row= str(next_array[0])+"," #start the next row with the timestamp.
         for i in range(1,len(working_array),1):
            current_row = current_row + str(working_array[i]) +"," #Write data from Feedback Array
            next_row = next_row + str(working_array[i]) + ","
         current_row = current_row + "\n"
         if Verbose:
```

```
            print(current_row)
        next_row = next_row + "\n"
        feedback.write(current_row) #write to CSV
        if Graphable:
            feedback.write(next_row)
        working_row += 1 #increment working row.
        if Verbose:
            print(working_array)
            print(working_row)
    working_array = FeedbackArray[working_row]
    current_row = str(working_array[0])+"," # start the current row with the timestamp.
    for i in range(1,len(working_array),1):
        current_row = current_row + str(working_array[i]) +"," #Write data from Feedback Array
    current_row = current_row + "\n"
    if Verbose:
        print(current_row)
    feedback.write(current_row) #write to CSV

def WriteTimingData(FeedbackFile, FeedbackArray, SignalArray, Verbose = False):
    #Takes an array of timing data provided by the Arduino and writes the first column to a CSV file.

    with open(FeedbackFile, "wb") as feedback:
        columnTitleRow = "Time(s),"
        columnTitleRow = columnTitleRow + "\n"
        feedback.write(columnTitleRow) #write column headings to CSV
        working_row = 0 #set row counter to co-ordinate writing to CSV
        while working_row < (len(FeedbackArray)-1):
            working_array = FeedbackArray[working_row]
            current_row = str(working_array[0])+"," # start the current row with the timestamp.
            for i in range(1,len(working_array),1):
                current_row = current_row + str(working_array[i]) +"," #Write data from Feedback Array
            current_row = current_row + "\n"
            if Verbose:
                print(current_row)
            feedback.write(current_row) #write to CSV
            working_row += 1 #increment working row.
            if Verbose:
                print(working_array)
                print(working_row)
        working_array = FeedbackArray[working_row]
        current_row = str(working_array[0])+"," # start the current row with the timestamp.
        for i in range(1,len(working_array),1):
            current_row = current_row + str(working_array[i]) +"," #Write data from Feedback Array
        current_row = current_row + "\n"
```

```
    if Verbose:
        print(current_row)
    feedback.write(current_row) #write to CSV
```

A1.3 SUITCEYES Controller Illustration Python Code

```
################################################################################
########
#
# SUITCEYES Serial Controller Illustration v1.0
# by Dr Raymond Holt, University of Leeds
#
# This code was developed as part of the SUITCEYES project (http://suitceyes.eu) and is owned by
the SUITCEYES consortium.
# It may not be modified or redistributed without the consortium's express permission.
#
# created 3rd May 2019
# by Raymond Holt
#
# Version History
# v1.0 Raymond Holt (UNIVLEEDS) 3rd May 2019 - first version uploaded to Box for illustration of
controller.
#
# OVERVIEW
# This program is intended to illustrate use of the SUITCEYES python module to communicate with
the Arduino Controller developed in Work Package 5.
# It requires that the SUITCEYES.py module is in the working directory and imported.
# This provides three modes for interacting with the controller:
#  1) By specifying the intensity of every channel during each frame - this allows control of multiple
channels simultaneously, but is both slower to transmit and to write;
#  2) By specifying the intensity and duration of each frame for a single channel - this is faster to
transmit, but only permits one motor to be operated during the signal; and
#  3) By specifying a channel, intensity and duration for each frame - this allows only one channel
frame, but is faster than method 1, and allows multiple channels to be used in a single signal.
#
#Program Begins
#
#import required libraries

import SUITCEYES

import sys

import serial

import csv

import time
```

```
################################################################################################################################################
#
# User defined variables - use these variables to customise the behaviour of the code.
#
################################################################################################################################################
Timing_Data_Wanted = True; #Determine whether timing files should be written. True will produce a .csv of timing data for every file.
Feedback_Data_Wanted = True; #Determine whether feedback files should be written. True will produce a .csv of feedbackk data for every file.
Signal_Select = True; #Determine whether to allow users to select which signal to display, or to automatically run through each row of the Haptic Library in turn. True means users will be prompted for the row to use before each trial.
arduinoport = "COM7" #Set the COM port for the arduino
dir_path = "/Users/rayjh/Documents/Work/SuitCEYES/" #Set the Path where the Haptic Library will be read from, and output files will be written to.
haptic_library_title = "Thessaloniki Demos.csv" #title of haptic library file to use.
prompt = "Which direction?" #This customises the prompt for recording answers at the end of the trial.
channels = 2 #defines number of channels.
sensors = 2 #defines number of channels.
Verbose = False;
ClosedLoop = True;
Master = True;

################################################################################################################################################
# End of User defined variables.
################################################################################################################################################


################################################################################
#
#Main Program
#
################################################################################

pp=raw_input("Participant identifier?")

haptic_library_file = dir_path + haptic_library_title

file_title = "Data_"+pp #name of the output file
```

```
download_dir = dir_path + file_title + ".csv"  # define the filename to write to
iteration = 0 #set iteration counter to 0. This is used to give each signal a unique ID for recording purposes.

# Open and read the signal library
with serial.Serial(arduinoport, 9600, timeout=0.5) as ser: # Open the Serial connection to the Arduino
   with open(haptic_library_file, "rb") as HapticLibrary:
      reader = csv.reader(HapticLibrary)
      rows = [r for r in reader]
      signal_length = (len(rows[0])/(channels+1))
      Packet_Size = SUITCEYES.Handshake(ser, channels, sensors, Verbose, Feedback_Data_Wanted, Timing_Data_Wanted, ClosedLoop) #handshake with Arduino - send channels, sensors, etc and receive Packet_Size for future data transmissions.

#main loop
      with open(download_dir, "wb") as output:  # "w" indicates that you're writing strings to the file
         while (len(rows)>iteration) or Signal_Select:
            print("trial number =" + str(iteration))
            if Signal_Select == True:
               select=raw_input("Select the Signal You Want to Display")
            else:
               select=iteration
            if select == "override":
               signalentered = raw_input("Type in the signal you want to display")
               signal = signalentered.split(",")
            elif select == "multichannel":
               signalentered = raw_input("Type in the signal you want to display")
               signal = signalentered.split(",")
            elif select == "direct":
               channel_to_use = int(raw_input("Type in the channel you want to communicate with..."))
               signalentered = raw_input("Type in the signal you want to display")
               signal = signalentered.split(",")
            elif select == "blankdirect":
               channel_to_use = int(raw_input("Type in the channel you want to communicate with..."))
               signalentered = raw_input("Type in the signal you want to display")
               signal = signalentered.split(",")
            elif select == "blankmultichannel":
               signalentered = raw_input("Type in the signal you want to display")
               signal = signalentered.split(",")
            else:
              signal = rows[int(select)]
            raw_input("Press Enter key to start trial")
```

```
            iteration += 1
            if select == "direct":
               print("I'm in direct mode!")
               Sensor, TimingData = SUITCEYES.DirectSignal(ser, channel_to_use, Packet_Size, signal,
Verbose)
            elif select == "multichannel":
               Sensor, TimingData = SUITCEYES.MultiDirectSignal(ser, Packet_Size, signal, Verbose)
            elif select == "blankdirect":
               Sensor, TimingData = SUITCEYES.BlankDirectSignal(ser, channel_to_use, Packet_Size,
signal, Verbose)
            elif select == "blankmultichannel":
               Sensor, TimingData = SUITCEYES.BlankMultiDirectSignal(ser, Packet_Size, signal, Verbose)

            else:
               Sensor, TimingData = SUITCEYES.TransmitSignal(ser, channels, Packet_Size, signal,
Verbose)

            if Feedback_Data_Wanted:
               print("Writing feedback file")
               feedback_dir = dir_path + "Feedback_Data_" + pp + "_"+ str(iteration)+ ".csv"
               SUITCEYES.WriteFeedback(feedback_dir, Sensor, Verbose)

            print("Collecting Data from Arduino")

            timing_dir = dir_path + "Timing_Data_" + pp + "_"+ str(iteration)+ ".csv"

            if Timing_Data_Wanted:
               print("Writing timing data file")
               SUITCEYES.WriteTimingData(timing_dir, TimingData, Verbose)

            print("Data collection complete")
            condition=False
            while condition ==False:
               answer = raw_input(prompt)  # enter response via keyboard
               condition = str.isdigit(answer)
            row_out=signal+answer.split()
            writer = csv.writer(output)
            writer.writerow(row_out)
```