1 Softwear

The software are build as modular as possible, for easy operation of the arm for a user with any programming background. The course are meant to be applied for students with some to none experience in programming and control theory, and at the same time give experienced personnel the possibility to conduct testing of advanced control theory applied. For a less experienced user, the steering library can be used where only the control factor numbers are to be adjusted. A more experienced user may build the control part itself for the implementation of other steering methods. The basic libraries for data collection can be adjusted and modified by an experienced user for optimal control, higher precision and further development of the system.
The software consist of five classes. The AS5600 library is provided by Seed-Studio for easy reading of data from the absolute magnetic encoder. There are three low level classes, the PID, pwmMotor and HapticSensor classes, which are made for data retrieving and control of the haptic arm on a hard code level. The final library, the HapticArm, contains a range of control methods based on control theory using the mentioned classes above for control of the arm.

1.1 AS5600

The AS5600 library is created and distributed by Seed-Studio, which is the producer of the encoder used in this project. The data are retrieved using I2C and memory access and the code are therefore manageable but too complicated to learn if you are not interested in the code side. The library gives the user an easy-to-use interface, retrieving the magnets position. This library needs to be put inside the programming folder or the main Arduino library folder for the program to work.

1.1.1 getRawAngle()

The *getRawAngle* is the only method used of this library. The command return the position of the magnet as a 12 bit signal (0-4095).

1.2 PID

The PID class are made for easy implementation of PID loops. It takes advantage of the object properties of classes for the user to easily be able to create multiple loops with different PID values based on the same class. The PID class have three public method in addition to the constructor. One which gives the possibility of a standard PID loop using Kp, Ki and Kd and one which provide the possibility of a PID controller with back calculation integration as an anti wind-up method. The last one is a complimentary filter which are used as a low pass filter on the sensor data. The PID method uses the same values and can be switch during operation if needed.

1.2.1 PID(float Kp_in, float Ki_in, float Kd_in)

The PID constructor is used to initialize the PID object. The constructor take Kp, Ki and Kd as input and a single PID object are intended to be used for a single PID loop. The values of the PID block cannot be changed during operation.

1.2.2 caluclate(float value, float target)

The *calculate* method is used if a standard PID loop are intended to be used. This method do not have an anti wind-up integrated. The PID equation used are the standard European method, see equation 1.1. The method have the current value intended to be used and the target as input. The aim target may be changed during operation, which is a key feature for the use of control systems.

The loop time is calculated for every loopback by the object for precision loop calculation. This make the object independent of the clock frequency or the loop time. And the code may stagger without the PID failing, taking advantage of the variable loop time calculation.

$$u_d(k) = K_p e(k) + K_i(u_{i-1} + e(k) \cdot dt) + K_d \frac{e(k) - u_{d-1}}{dt} \tag{1.1}$$

u_d is calculated value, $e(k)$ is the error of the system, u_i is calculated integral constant and dt the time step.

1.2.3 backcalc(float value, float target, float backVal, float satutaionMin, float saturationMax)

The *backcalc* method use the same equation as the calculation method, as well as iterate over the same loop time and uses the same Kp, Ki and Kd. These properties make it possible for the user to switch between these two methods. The value are the current value input for the calculation and the target value are the target for the calculation. In addition to these two, the *backVal* input is used for tuning the anti wind-up section. The *saturtionMin* and *saturationMax* inputs are used for the upper and lower limit for the anti wind-up.
The anti wind-up constant could be set to 1 which would work great. This method should be used as the final loop before sending the value to the motor. If this is the case, an upper limit of 255 and lower -255 is advised, corresponding to the upper and lower limit for PWM.
In the backcaluc method, the integral part of the PID calculation are switched with this equation 1.2.

$$u_d(k) = K_p e(k) + K_i(u_{i-1} + (e(k) + \frac{e_p(k) - u_{d-1}}{T}) \cdot dt) + K_d \frac{e(k) - u_{d-1}}{dt} \tag{1.2}$$

e_p is the saturated u_d and T is the back calculation value

1.2.4 compfilter(float in_val, float alpha)

The *compfilter* method is integrated as a way to use the equation for complimentary for values. The complimentary filter may be used as a low-pass filter or high-pass filter, depending on the inn-values. A low-pass filter can be essential for readable sensor-data. The *in_val* are the new data from the sensor used for calculation. A separate PID object should be initialized for every sensor in the system needing a filter. The earlier results are remembered by the object, and the same object cannot be used for different sensor-data. The *alpha* constant is the tuning value for the filter. This decides how much the new values are to be weighted. This value is set to 0.01 as standard and do not need to be given for this value, unless a different number is to be used.
The method return the final calculated value. The equation used is 1.3.

$$u_d = (1 - \alpha) \cdot u_{d-1} + \alpha \cdot e(k) \tag{1.3}$$

1.3 pwmMotor

The *pwmMotor* class are a collection of different method used for control of a PWM motor using a H-brigde style motor controller and collect data from a hall encoder. The class consist of five public methods in addition to two private ones and the constructor. The two private methods are used for the collection of data as interrupt from the hall encoder and are set to ”static void” as all methods interacting with interrupt variables do need to be of this data type. This is also the reason for the “static void” data type of the *reset_hall_val* method.

1.3.1 pwmMotor(int forwardPin_in, int backwardPin_in, int pwm_in, int hallPinOne_in, int hallPinTwo_in)

The constructor of the pwmMotor class have five inputs for correct setup of the connection between the microchip and the motor controller. The first two inputs are the microchip pins, which are to be connected to the directional control of the motor. These pins are used for digital signals. The third is the pin which the PWM signal are sent to. The last two are the pins which the hall sensor pins are connected. Mark: On Arduino Due the only two pins supporting interrupt are 0 and 1.

1.3.2 goToSpeed(int motorSpeed)

This method is the main method used for control of the motor. The method receives the motor speed as an integer. Inside the method, the number will be limited to -255 to 255 to ensure the value do not exceed the capabilities of the microchip. The Arduino Due have an 8 bit DAC and these values are preset for this intent, but may be adjusted if a different chip set is to be used.
If *motorSpeed* is negative the method will spin the motor backward and on the contrary if the value is positive the motor is spun forward. The method also check if the value is equal to the last value asked. The operation of changing direction may be more time conceiving than just adjust the PWM sequence. Checking if the value have changed and immediately break if the value are the same, saves computational time. A value of -1 for backward and one for forward, are additionally commutated and saved for the intent of being able to check the motor direction elsewhere in the code. This is used for the emergency brake.

1.3.3 stop()

The *stop* method immediately set both directional control pin to zero, and then adjust the PWM signal to zero as well. In this configuration, the motor will run free, which is the optimal configuration if the arm have hit something and need to be stopped immediately. This method is used by the emergency break due to endstops, but may also be implemented if the arm is used for obstacle avoidance or as a cobot.

1.3.4 check_rotation()

The *check_rotation* method return the angle of the arms based on the number of trigger events from the hall sensor interrupt routines. While reading the count, the interrupt routine have to be temporarily paused. This means that for every iteration of the method, steps may be missed.

1.3.5 reset_hall_val()

This method is used by the calibration method of the arm. At the end of the calibration sequence, the arm is situated at angle 0. Since the motor encoder is incremental, a fixed start position is necessary. The interrupt will start counting as soon as the motor object is initialized, and the count have to be zeroed at a known location. The calibration routine calls this method in the known zero position, which set both trigger event counts to zero.

1.3.6 return_motor_dir()

The value saved by the *goToSpeed* method is internal, and the *return_motor_dir* method can be called to read the current value. This is used by the end stop emergency stop. The motor direction value is used for the trigger event detection as well.

1.4 HapticSensor

The Haptic Arm have five types of sensors which are integrated into the platform. The motor encoder is implemented in the *pwmMotor* library. The remaining four are the data collected through this library. The AS5600 library is used for the reading of the absolute position encoder. The library

consist of five methods in addition to the constructor. Four are the methods for returning sensor data, and the last one is for configuration of the end positions of the arm for the position sensor.

1.4.1 HapticSesor(int forcePin, int currentPin, int switchPinOne, int switchPinTwo)

The *HapticSensor* constructor takes four inputs, which are the pins which the sensors are connected to. *ForcePin* is the pin for the load cell, which has to be connected through an instrument amplifier and are read as an analogue signal. The *currentPin* is the pin connected to the current sensor, which is read as an analogue signal as well. *switchPinOne* and *switchPinTwo* are the pin which the signal side of the endstops. In the constructor, all the input pins are set as inputs, and the analogue read resolution set to 10 bit.

1.4.2 readForce()

The *readForce* method return the force exposed to the end of the arm. The signal read from the load cell is adjusted due to a graph found through testing with known weights at the connection point on the arm. The force is calculated from the weigh and adjusted for the current angle of the arm for neglecting the weight of the swing arm. The value is adjusted with a low pass filter via the PID library before returned as a float.

1.4.3 readPos()

The *readPos* method use the AS5600 library to retrieve the absolute position of the arm from the magnetic encoder. The belonging angle is calculated and returned as a float. The calculation use the values found by the calibration sequence for the value of the magnetic encoder at the endstops.

1.4.4 readCurrent()

This method read the value provided by the current sensor, which is situated between the motor and the motor controller. The signal is converted from a bit value into volt and adjusted for the signal at zero current, roughly in the middle. The volt signal is then multiplied with the amp per volt value given for the sensor. A low-pass filter using a PID block are used to smooth out the signal. The current is returned as a float.

1.4.5 readSwitch()

The *readSwitch* method update both values in the list containing the state of the two endstop switches connected. And finally returning these as a pointer to the list. Therefore, the data type int* is used. The list need to be initiated as a class object and not initialized in the object, as the list in that case would be saved in the stack (which is temporary) and removed when the method returns.

1.4.6 calibrateEncoder(int newMinVal, int newMaxVal)

The values for the magnetic absolute encoder are saved as object parameters, such that the calibration sequence is not absolutely necessary if the values for a specific configuration are well known. These values are set in the constructor. After the calibration sequence, the found values are used to change these private values by the use of the *calibrateEncoder* method. The method return the current raw bit value of the magnetic encoder for the intent of using it during calibration. The argument are optional, but the values will be set to 0 if no value is provided.

1.5 HapticArm

The *HapticArm* library is a class combining the lower lever control classes of the Haptic Arm with the intent of controlling the arm using control theory. The library consist of four public and four

private methods in addition to the constructor. The of the private method two are calculating the moved length, speed, and acceleration based on the current and part position using numerical mathematics and the same as angles. The mathematics used for the calculation are shown in equation ??. The last two are the emergency protocols witch read data to ensure the arm have a reasonable behaviour and engage the stop routines if not.

1.5.1 HapticArm(int motorSettings[ ], int sensorSettings[ ], int PIDset[ ][3])

The *HapticArm* constructor takes two list and one multidimensional list as input. The first list are the values used to set up the motor object, in total four values. The object for using the other classes are set up inside this class, which means that all necessary values have to be delivered by this object as a middle man. As a way to avoid having 15-20 input for the object, it is initialized with lists. The second list are the values for the sensor class, and need five objects. The constructor do not need to know the size of the list, but enough values have to be given for the constructor to succeed. The third list is a Nx3 matrix, and contains the values Kp, Ki and Kd for all PID objects which are used in the object. The objects are the defined in the constructor as well as other needed constant. One of this is the type of switch used. Original the endstop switches are set to normally closed (NC) corresponding to "1". If the setup support NO, this would be better, but the Due does not support the needed pull-down resistor.

1.5.2 goToPos(float requiredPos)

This method is the basic inner loop of the motor control. It read the position of the arm and adjust the PWM signal sent to the motor, trying to achieve the given required position. This method use the Position PID object, which is the first three values given in the PID matrix. This PID should be the first to be tuned for optimal control of the arm, due to the fact that it is the inner loop for the *goAdmittance* method. For the PID calculation, back calculation is used for saturation of the signal.

1.5.3 calibrateArm()

The *calibrateArm* method are the main calibrate sequence which check the position of the magnet at the end stops and zero the motor encoder at 0 degrees. By reading both switches during the sequence, the switch located at zero degrees and 250 depresses is mapped, and take care of misplacement of these wires. The motor speed will be manual, set to the minimum safe speed which the arm will be able to rotate.
For visual confirmation of started sequence, the method will print "Calibration started" to the serial monitor. After successful calibration, the maximum and minimal magnetic value will be printed at the serial monitor. This value may be used to adjust the preset values in the code.
The calibration sequence may be left out if the preset values are to be trusted and the motor encoder is not in active use.

1.5.4 goImpedance(float massConstant, float damperConstant, float springConstant)

This method enables the user to control the arm using Impedance control, which means reading the movement of the arm and adjusting the output torque of the motor. The private method *movedAngle* are used to find the radial movement of the arm and the current are read by using the sensor object. The movement values are then multiplied with the mass, spring, damper values, which are given by the user as inputs for the method. The equation used is 1.4.

$$\tau = M\ddot{\theta} + D\dot{\theta} + K\theta \tag{1.4}$$

The resulting torque is with the read current used to calculate a new speed and feed to the motor. The emergency test is embedded for safety.

1.5.5 goAdmittance(float massConstant, float damperConstant, float springConstant, float initialPosition)

This method enables the user to control the arm using Admittance control, which means reading the input force and controlling the position of the arm. The system will act as a spring damper system and centre around the given initial position value. The private method *movedLenght* is used to find the moved length since last iteration. The length is based on the given length of the arm and uses trigonometry. It is an estimate based on numerical iteration of a curve divided into triangles, and the frequency of the code are important for the validation of the method.
The current applied force are read by the use of the sensor object and the corresponding offset length are calculated using this equation 1.5.

$$x_{new} = \frac{F - D\dot{x} - M\ddot{x}}{K} \tag{1.5}$$

The calculated length is calculated as angle and applied to the initial angle value. Finally, the required angle is sent to the *goToPosition* method used for the inner loop. The emergency test is embedded in the *goToPosition* method.

1.5.6 energencyCheck()

The method check the current motor direction and if any of the end stop switches are activated. If one of the switches are activated and the motor direction is pushing the arm further into the switch, the emergency brake is activated. If the motor direction are away from the switch, it is assumed the motor are trying to fix the problem and the break are not activated.