Industrial Automation Project 2

Kedar More

Abstract

This project will build on the previous project and use the Continuous time controller to design a Discrete time controller. This will be implemented on the system and check the differences in the output. The end goal is to make the system stable with a Discrete time controller.

1 Introduction

Till now we were working on the continuous-time systems. But it is not always feasible and out of reach of several hardware. Hence it is necessary to discretise our signal so the the hardware will be able to read it. Also there must be some time for the hard to give out some signal to the actuators according to the input. All this cant be done using a continuous feedback loop. In this report we will go over several methods of signal sampling and implement them on Arduino Due. In Matlab there are several inbiult emulation methods of which we will try the Forward Euler's and Tustin's method. Also we will see the on paper derivation of these and try our own controller block. Lastly we will try to disretise the singal using the concept of Interupt Service Routine to give the output in a specific interval of time.

2 Emulation

2.1 What is Emulation

Emulation is basically a method of converting a continuous time controller to a discrete time controller. The term "emulation" comes from the verb "emulate," which means to imitate or reproduce.

2.2 Difference between Emulations

There are 3 types of emulation methods:[3]

1. Forward Euler Method: Also known as Forward-difference approximation, it is possible that a stable continuous-time system is mapped into an unstable discrete-time system.

2. Backward Euler Method: Also known as Backward approximation, a stable continuous-time system will always give a stable discrete-time system.

3. Tustin's or Trapezoidal Method: It has the advantage that the left half s-plane is transformed into the unit disc in the z-plane.

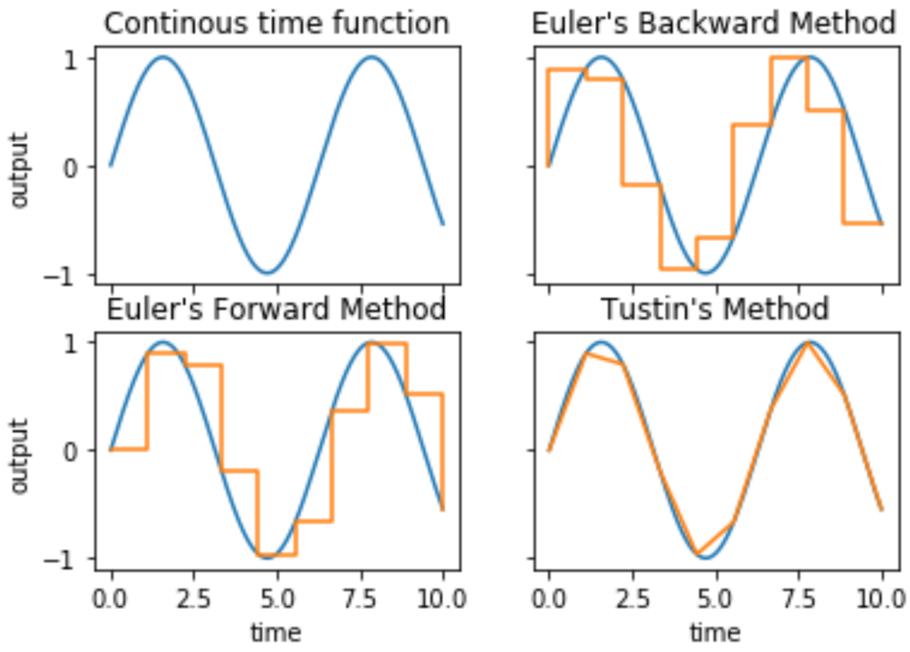


Figure 1: Types of Emulation Methods

2.3 Continuous time controller

The best controller which was developed in the Project 1 was:

$$y = \begin{cases} -70 * (e^{-\frac{u+400}{300}} + 1) + 3 * du & u \geq 0 \\ 50 & u < 0 \end{cases}$$

This is a (non-linear P)D controller where the non linear term was decided by experimenting with the system. A linear P controller would make the system system unstable by an impulse input. Adding D to it would reduce the amplitude of the output but would not damp it as needed. Hence the new P controller was there to input a lesser value than expected for a larger error and vice versa.

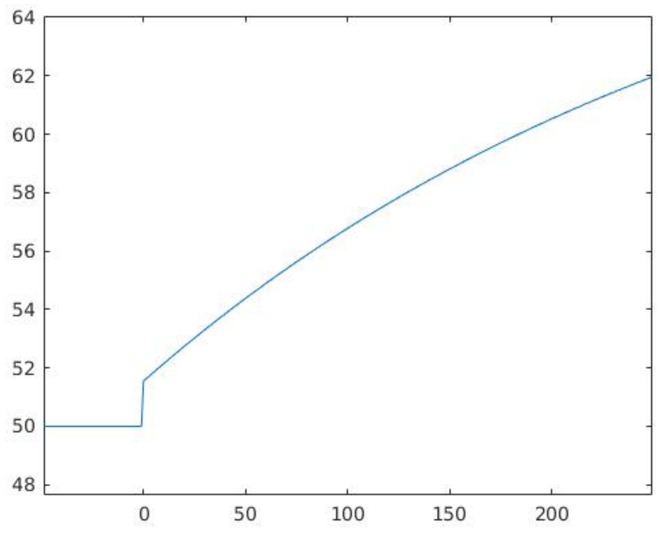


Figure 2: Non-linear Proportional controller

Taking a Laplace transform of the equation:

$$\mathcal{L}(-70 * (e^{-\frac{u+400}{300}} + 1) + 3 * du)$$
$$= \mathcal{L}(-70 * e^{-\frac{u}{300} + \frac{4}{3}} - 70 + 3 * du)$$
$$= -\frac{21000}{(300s + 1)e^{\frac{4}{3}}} - \frac{70}{s} + \frac{3}{s}$$

2.4 Euler's Forward Method

Derivation:

Using z-transform to eliminate the input.

$$z = e^{st}$$

As the sampling frequencies are very small we can approximate using the Taylor series.

$$z = 1 + st$$

$$s = \frac{z-1}{t}$$

Substituting the above result in the controller.

Changing the pieces from $s \geq 0$ and $s < 0$ to $z \geq 1$ and $z < 1$ as t is always positive.

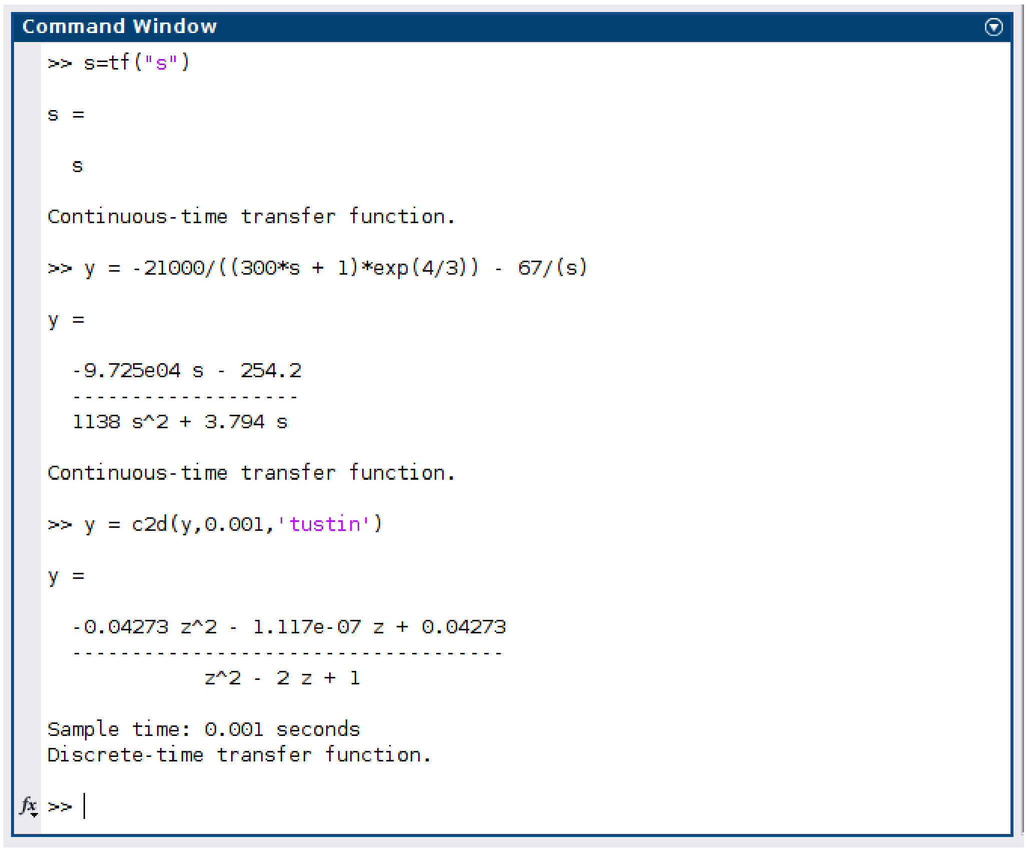
$$y = \begin{cases} -\frac{21000}{(300\frac{z-1}{t}+1)e^{\frac{4}{3}}} - \frac{70}{\frac{z-1}{t}} + \frac{3}{\frac{z-1}{t}} & z \geq 1 \\ 50 & z < 1 \end{cases}$$

$$y = \begin{cases} \frac{-97.25z+97.25}{1138z^2-2276z+1138} & z \geq 1 \\ 50 & z < 1 \end{cases}$$

where t=0.001

2.5 Tustin's Method

We use the matlab function c2d() to emulate the system using Tustin's method.



```
>> s=tf("s")

s =

  s

Continuous-time transfer function.

>> y = -21000/((300*s + 1)*exp(4/3)) - 67/(s)

y =

  -9.725e04 s - 254.2
  -------------------
  1138 s^2 + 3.794 s

Continuous-time transfer function.

>> y = c2d(y,0.001,'tustin')

y =

  -0.04273 z^2 - 1.117e-07 z + 0.04273
  ------------------------------------
             z^2 - 2 z + 1

Sample time: 0.001 seconds
Discrete-time transfer function.

>> 
```

Figure 3: Tustin calculation

The final z transform after the using the tustin method is:

$$y = \begin{cases} \frac{-0.04273z^2 - 1.117*10^{-7}z + 0.04273}{z^2 - 2z + 1} & z \geq 1 \\ 50 & z < 1 \end{cases}$$

3 Simulink Implementation #1

3.1 Solvers

There are 12 types of solvers namely Fixed-Step, Variable-Step, Continuous, Discrete, Explicit, Implicit Continuous, One-Step, Multistep Continuous, Single-Order, Variable-Order Continuous Solvers.[2]

To choose the correct solver for our system we need to incorporate the following criteria:

1. System dynamics
2. Solution stability
3. Computation speed
4. Solver robustness

Computation Step Size Type

Fixed-step solvers, as the name suggests, solve the model using the same step size from the beginning to the end of the simulation. You can specify the step size or let the solver choose it. Generally, decreasing the step size increases the accuracy of the results and the time required to simulate the system.

Variable-step solvers vary the step size during the simulation. These solvers reduce the step size to increase accuracy at certain events during the simulation of the model, such as rapid state changes, zero-crossing events, etc. Also, they increase the step size to avoid taking unnecessary steps when the states of a model change slowly. Computing the step size adds to the computational overhead at each step. However, it can reduce the total number of steps, and hence the simulation time required to maintain a specified level of accuracy for models with zero-crossings, rapidly changing states, and other events requiring extra computation.

Model States

Continuous solvers use numerical integration to compute continuous states of a model at the current time step based on the states at previous time steps and the state derivatives. Continuous solvers rely on individual blocks to compute the values of the discrete states of the model at each time step.

Discrete solvers are primarily used for solving purely discrete models. They compute only the next simulation time step for a model. When they perform this computation, they rely on each block in the model to update its individual discrete state. They do not compute continuous states.

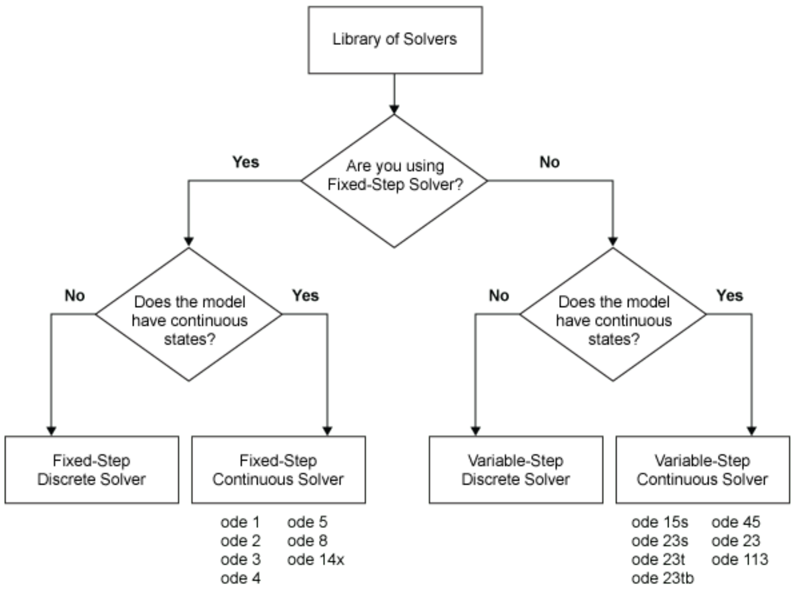

Library of Solvers
Are you using Fixed-Step Solver?
Yes
No
Does the model have continuous states?
No
Yes
Does the model have continuous states?
No
Yes
Fixed-Step Discrete Solver
Fixed-Step Continuous Solver
Variable-Step Discrete Solver
Variable-Step Continuous Solver
ode 1
ode 2
ode 3
ode 4
ode 5
ode 8
ode 14x
ode 15s
ode 23s
ode 23t
ode 23tb
ode 45
ode 23
ode 113

Figure 4: Choosing a solver

3.2 Implementation using "ode1"

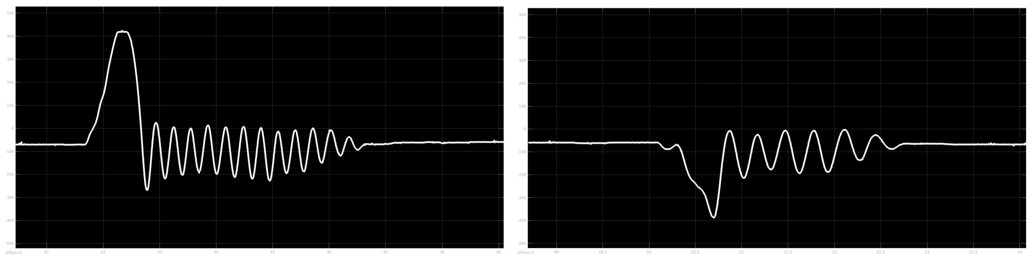


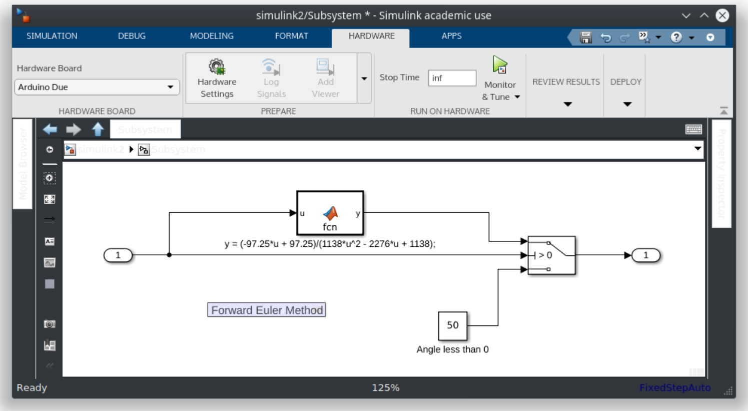
Figure 5: Step response using ode1 solver

As expected the response for the solver "ode1" is much slower than the "auto". The main reason is the approximation till 1st degree differential. Yet, the system is stable for a step response with a delayed response.

4 Simulink Implementation #2

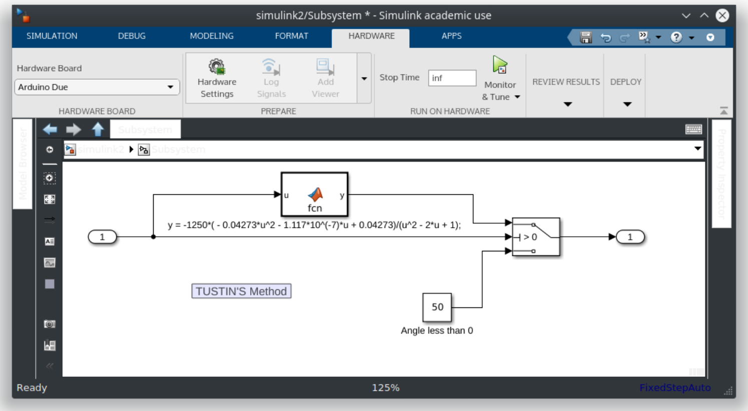
4.1 Transfer Blocks

Here the Matlab functions are used to calculate the transfer function with the given input. The same feedback structure is used as in the previous project but the controller is changed as shown below. Instead of the continuous one I discretised it using two methods of the z transform.



```
y = (-97.25*u + 97.25)/(1138*u^2 - 2276*u + 1138);
```

Figure 6: Euler forward method implemented



```
y = -1250*( - 0.04273*u^2 - 1.117*10^(-7)*u + 0.04273)/(u^2 - 2*u + 1);
```

Figure 7: Tustin's method implemented

4.2 Output

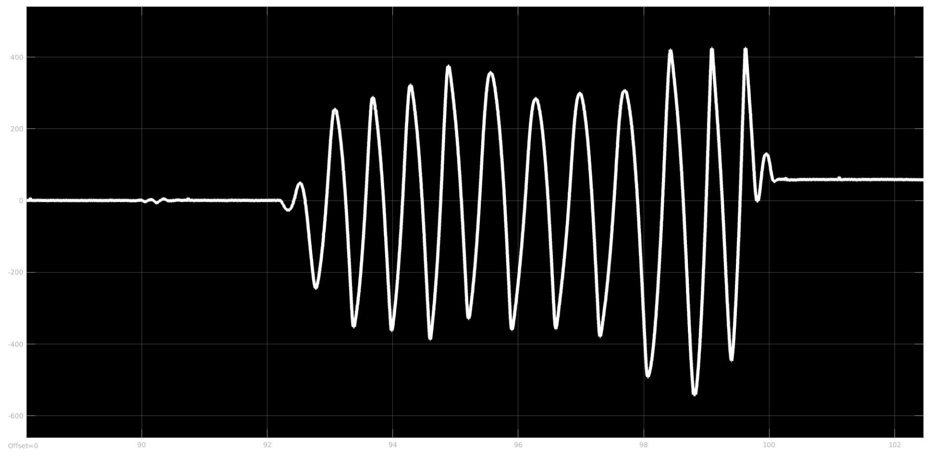


Figure 8: System becoming unstable (forward euler)

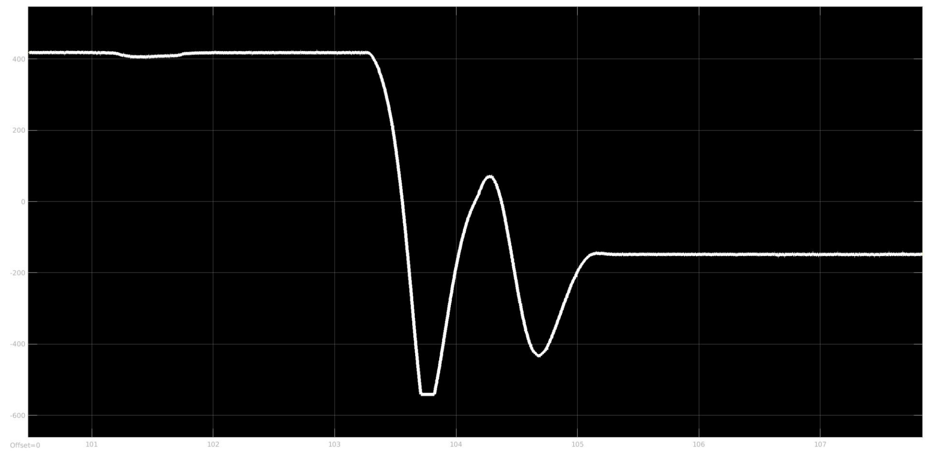


Figure 9: System is somewhat stable (tustin's method)

Euler's forward method made the system unstable with a range of sample times. Tustin's method does a better job at encompassing the actual signal rather than Forward Euler method. the slope of the line between the sample is as close as possible the original signal. This accurately gives the output to the controller without too much change.

5 Arduino Due Implementation

5.1 Feedback using Potentiometer

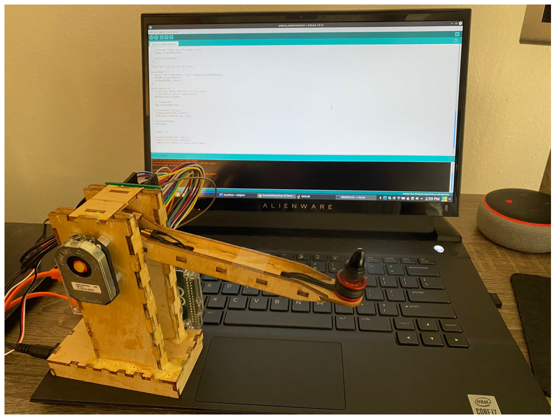


Figure 10: Code uploaded using Arduino

5.1.1 Code

The most important part of the code was to implement an interrupt timer routine to build the required discrete controller. The interupt is a software protocol which is associated to a function. It is used to run the function with a priority. When an interrupt is run all the other implementations are stopped and that function is completed.

An interrupt handler, also known as an interrupt service routine (ISR), is a callback subroutine in an operating system or device driver whose execution is triggered by the reception of an interrupt. Interrupt handlers have a multitude of functions, which vary based on the reason the interrupt was generated and the speed at which the Interrupt Handler completes its task.

Converting the z -transform to code:

$$\frac{y}{u} = \frac{-0.04273z^2 - 1.117 * 10^{-7}z + 0.04273}{z^2 - 2z + 1}$$

Dividing numerator and denominator by z^2

$$\frac{y_i}{u_i} = \frac{-0.04273 - 1.117 * 10^{-7}z^{-1} + 0.04273z^{-2}}{1 - 2z^{-1} + z^{-2}}$$

$$y_i - 2y_{i-1} + y_{i-2} = -0.04273u_i - 1.117 * 10^{-7}_{u-1} + 0.04273_{u-2}$$

5.1.2 Controller

The original non linear controller could not be implemented here as there a limit to which we can stall the interrupt function. If there are many computations in the interrupt the the hardware will be slowed down. This will change the results for that particular sample time. Hence a linear PD controller was implemented with the following constants.

$$kp = 0.6, kd = 0.005$$

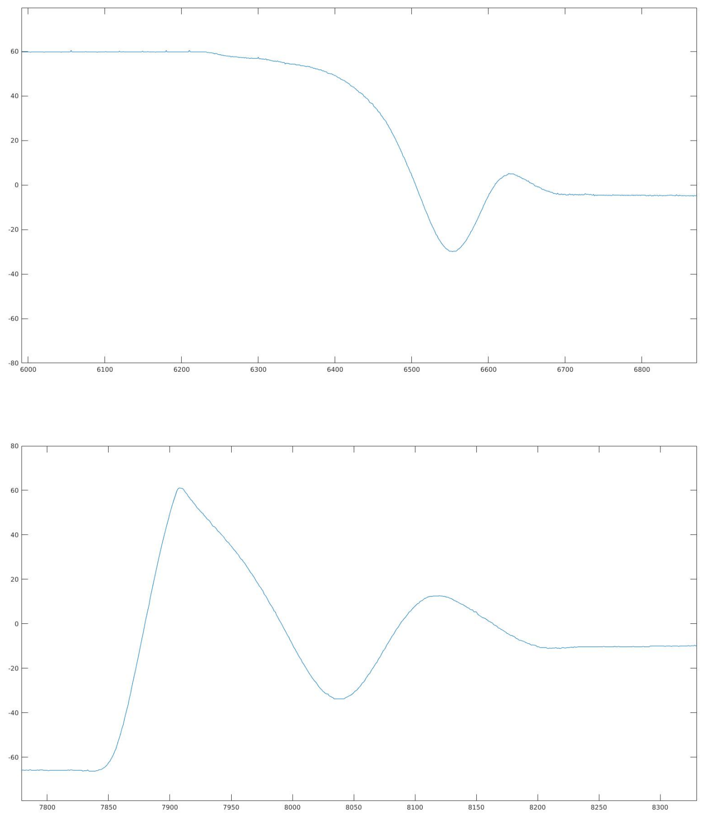


Figure 11: Step response of T-RECS[4] with a feedback from potentiometer

5.2 Feedback using Encoder

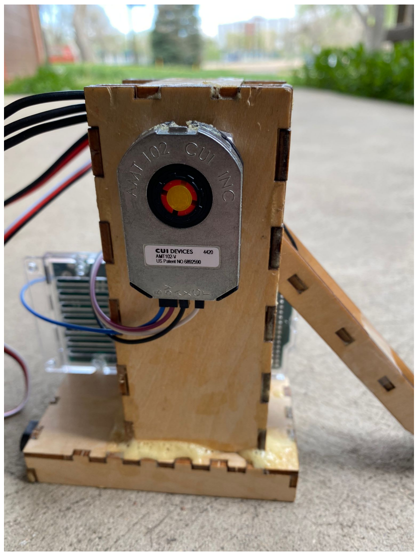


Figure 12: AMT 102 rotary encoder

5.2.1 Connections

4 pins from the encoder are used to take feedback.

Encoder	Arduino
5V	5 V
G	Gnd
A	D2
B	D12

5.2.2 Code

To run the encoder some of the registers in Arduino Due need to be set and reset. The resolution of this encoder is 12 bits which is more than enough for the accuracy we need.[1]

```
REG_PMC_PCER0 = PMC_PCER0_PID27;
REG_TC0_BMR = TC_BMR_QDEN;
REG_TC0_CMR0 = TC_CMR_TCCLKS_XC0;
```

```
REG_TC0_BMR = TC_BMR_QDEN | TC_BMR_POSEN | TC_BMR_EDGPHA;
REG_TC0_CCR0 = TC_CCR_CLKEN | TC_CCR_SWTRG;
```

Rest of teh things are same as the above implementation such as the mapping of angles and the ISR.

5.2.3 Controller

The controller is kept the same and the results are plot with approximately similar disturbance.

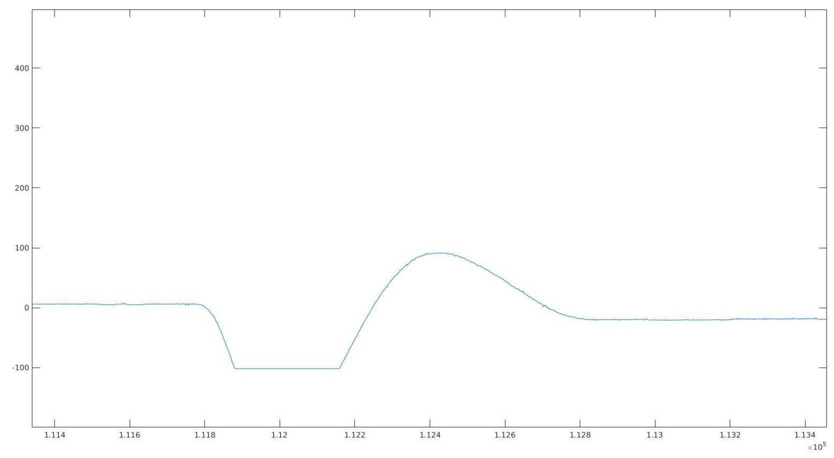


Figure 13: Step response of T-RECS with a feedback from AMT 102 rotary encoder

6 Conclusion

The rotary encoder work better than the potentiometer by a small margin. Visually it seem to be more smooth as apposed to the jagged path taken by the potentiometer. The resolution difference come to play here as the sudden changes are registered. When a huge change occurs PID controller tend to fail as the P term gets comically large and as the system approaches its steady state the velocity becomes to huge to be controlled by the D term. Hence small changes are preferred which is provided by the encoder. Finally it is concluded that using Tustin's method is the best and the most feasible method to discretise the signal for a feedback controller. While using the in built function for the Simulink the AutoFixedStep can be used to get good results.

References

[1] *Arduino Due reads encoder — Details — Hackaday.io.* `https://hackaday.io/project/13084-load-frame-update/log/48234-arduino-due-reads-encoder`. (Accessed on 05/07/2021).

[2] *Choose a Solver - MATLAB & Simulink.* `https://www.mathworks.com/help/simulink/ug/choose-a-solver.html`. (Accessed on 05/07/2021).

[3] *Forward and Backward Euler Methods.* `https://web.mit.edu/10.001/Web/Course_Notes/Differential_Equations_Notes/node3.html`. (Accessed on 05/07/2021).

[4] *T-RECS System — Tangibles That Teach.* `https://www.tangiblesthatteach.com/product-page/t-recs-system`. (Accessed on 05/07/2021).