Software Report

Updated – April 8, 2022

Wenhao Xu

Abstract

This paper is about the software aspect of the project, including PID controller, actuators and C code.

Nomenclature

Joint 1 Shoulder joint of SCARA robot, & all associated components.

Joint 2 Elbow joint of SCARA robot, & all associated components.

1 Controller Design

1.1 RCG

There are two main concerns. First, it should be accurate to be useful. Next, it should be fast to be adapted by factories.

Requirement:

- Must be able to settle within 1.5 seconds

Constraint:

- Must not have overshoot larger than 20%

Goal:

- Minimize settle time
- Minimize rise time
- Minimize overshoot
- Minimize steady state error

1.2 System Models

The model of the control system is as following.

1.2.1 Amplifier Transfer Function

The amplifier circuit introduces some transient. It is modeled as a second order system for simplicity. The picture below shows a step response of the PCB driver circuit in multisim. The switch is flipped at 22ms, and a response is produced.

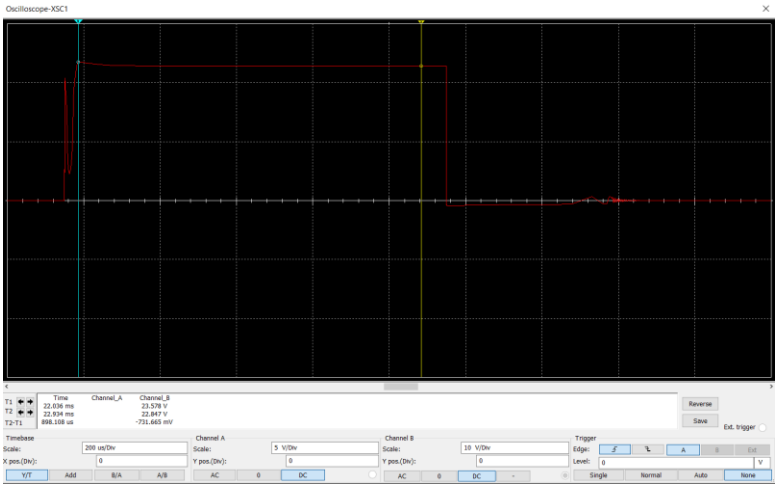

Fig 1. Step Response

From figure 1 it can be identified that the peak occurs at 0.036ms, with peak value 23.578 V. The system settles to about 22.847 V.

So, the overshoot value is 23.578-22.847 = 0.7310 V. The values can be used to estimate a second order system. The transfer function is found to be: (voltage/controller output)

$$\frac{3.829e11}{s^2 + 1.912e5\ s + 1.676e10}$$

Plot the transfer function in Matlab, the graph is as the following.

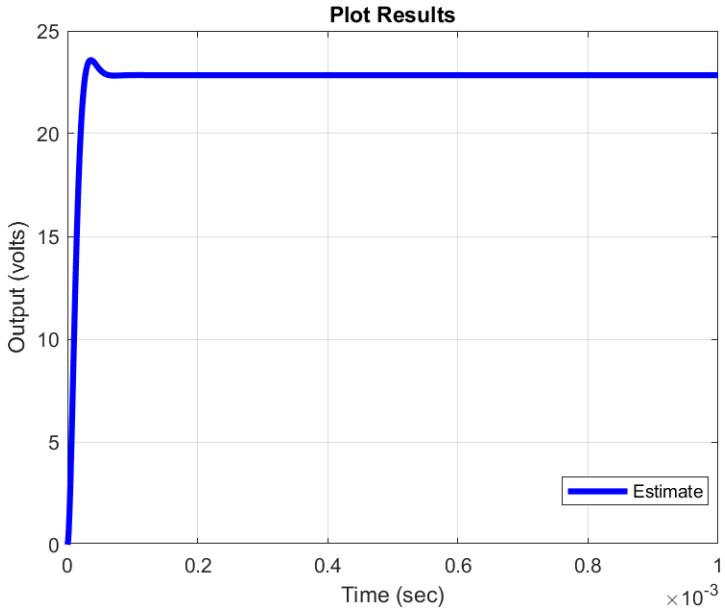

Fig 2. Step Response

The figure shows that there's an overshoot which peaks at about 0.03 msecs, which is a good estimation of the driver circuit.

1.2.2 Motor

For the motor, first it has an electrical subsystem that can be modeled as a first order transfer function. To determine this system, the terminal resistance and terminal inductance are retrieved from the datasheet. [1]

R_w = 2.68 Ω

L_w = 0.514e-3 H

Then the transfer function is:

$$\frac{1}{Lw * s + Rw} = \frac{1}{0.514e-3\ s + 2.68}$$

It is also worth noting that for a motor, there are:

$$\tau = K_m I$$

$$E_a = K_m \omega$$

From the data sheet above, the torque constant K_m is 42.9 mNm/A = 42.9e-3 Nm/A.

1.2.3 Mechanism

For the two joints, two different gears are used.

1.2.3.1 Joint 1 (Shoulder)

The gear ratio used is i = 30. So, the mechanical impedance is

$$\frac{1}{0.7833\ s + 1.965e-3}$$

Combined, the mechanical system is as shown in the figure 3.

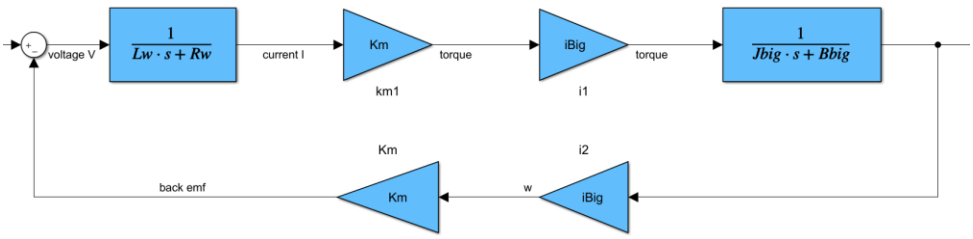

Fig 3. Mechanical System

1.2.3.2 Joint 2 (Elbow)

The gear ratio is i = 20. So, the mechanical impedance is

$$\frac{1}{0.1131\ s + 8.731e - 4}$$

1.2.4 Sensor

Sensor introduces non-linearity. Based on different stage of modeling, there are different identifications.

1.2.4.1 Linear

At this stage, it is modeled as a linear system. So, it simply converts angle to counts, which is converted back to angles by the microcontroller. Thus, it is ignored for now.

1.2.4.2 Non-linear

For non-linear system, the sensor can be modeled as a Simulink subsystem like in the figure.

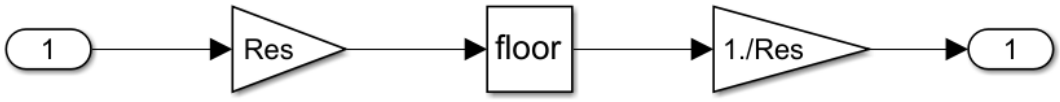

Fig 4. Sensor Model

The sensor is used as a relative sensor. The sensor has max pulses per revolution of 1440 from the data sheet [2]. Therefore, the sensor can see 360/4/1440 = 1/16 degrees of angle at best. The Res gain here first converts the radians to degrees, then multiplies it by 16 (so the least angle it can see is 1 count). That is

$$Res = \frac{180}{\pi} \cdot \frac{1}{\frac{1}{16}}$$

Because the sensor outputs whole number counts, Matlab function floor is used. This introduces non-linearities. The 1./Res gain emulates the behavior of the microcontroller, where the angle in radians is recovered.

1.2.5 Control Frequency and Microcontroller

Control frequency is determined to be around 1300/s. The effect of sampling by the microcontroller is modeled as a transfer function:

$$\frac{2CF}{s + 2CF}$$

1.3 Sensor Filter

We chose the sensor because the sensor has a good resolution, which may not require future filtering.

To see if the sensor requires extra filtering, we investigated the sensor output versus input.

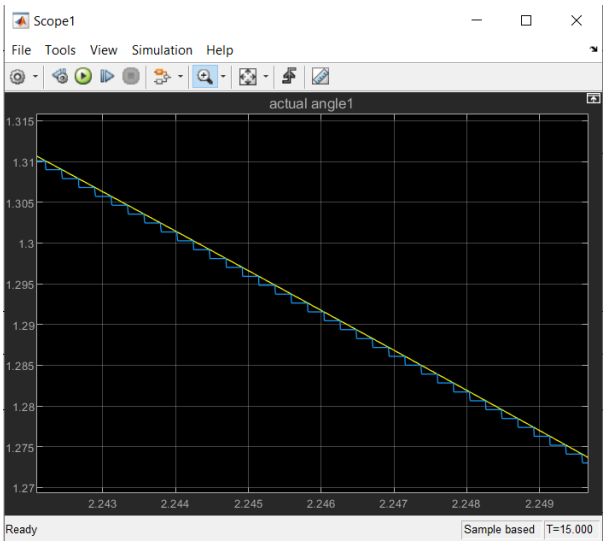

Fig 5. Sensor

In the figure, the blue graph is the output and the yellow one is input. As the figure shows, the sensor has a good resolution.

After evaluating the sensor, we decided that it does not require extra filtering.

1.4 Derivative Filter

The derivative filter is implemented in microcontroller. Thus, the unmodified derivative filter is the same as the microcontroller dynamics. After the tuning, the filter is not changed because the pole given by the filter is good enough.

1.5 10-Step Process

1.5.1 Joint 1

Step 1:

First, calculate the G = G1*G2*1/s and H=Hs*Hc. Hs is approximated as 1 in our system.

G1 is amplifier transfer function.

$Ye = 1/(L_w s + R_w)$ is the electrical transfer function of motor.

$Ym = 1/(J_{big} s + B_{big})$ is the mechanical impedance of the joint.

G2 = Ye*Ym*K_m*i_{big}/(1+Ye*Ym*K_m*i_{big}*Km*i_{big}) – the overall forward branch.

Step 2:

Use the margin () function get Gain & Phase X-over frequencies (GXO & PXO). The margin graph is as below.

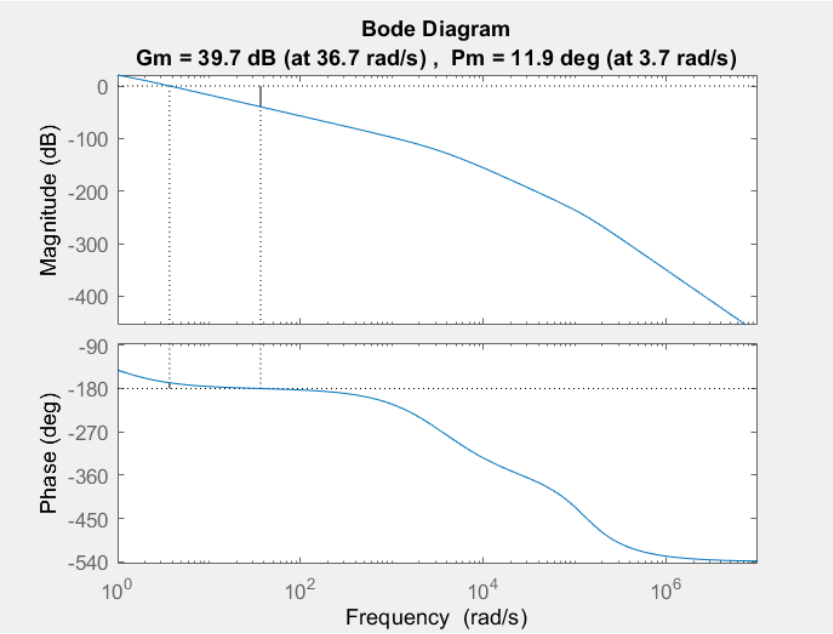

Fig 6. Margin

From the graph, there is

PXO = 36.7

so,

z = 3.67

Step 3:

P = 2*CF =2*1300 = 2600

Kp = 2/z-1/p = 0.5446

Ki =1

Kd = 1/(z*z) - Kp/p = 0.0741

D = Kp + Ki/s + Kd*p*s/(s+p)

Step 4:

Iterate. The final z value is chosen to be z = 0.15.

Step 5:

Zoom in the root locus of the open loop system G*H, around the origin there is

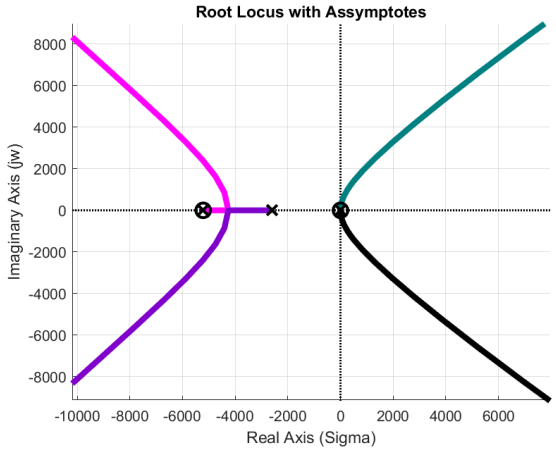

Fig 7. Root Locus

which is unstable. Then plot the root locus of open-loop system D*G*H.

Zoom in, it shows a better gain.

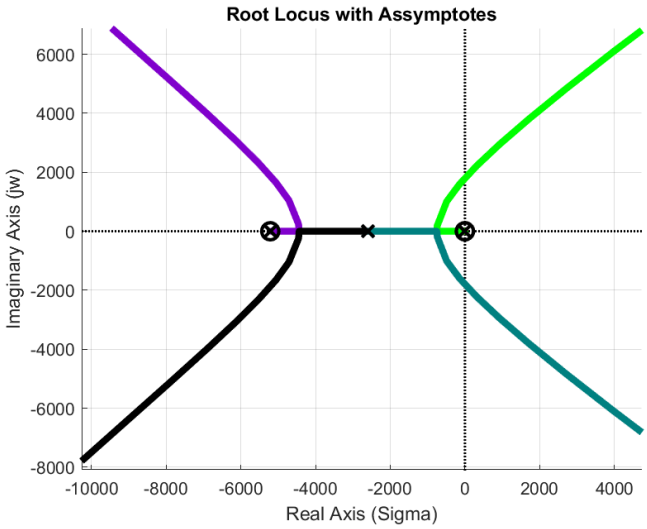

Fig 8. Root Locus Improved

To get the initial gain K, let K = 1/abs(freqresp(D*G*H,z)) = 6.4705e-4.

Step 6:

To check if the K improves the phase margin, plot the Nyquist contour of the open-loop system D*G*H and K*D*G*H.

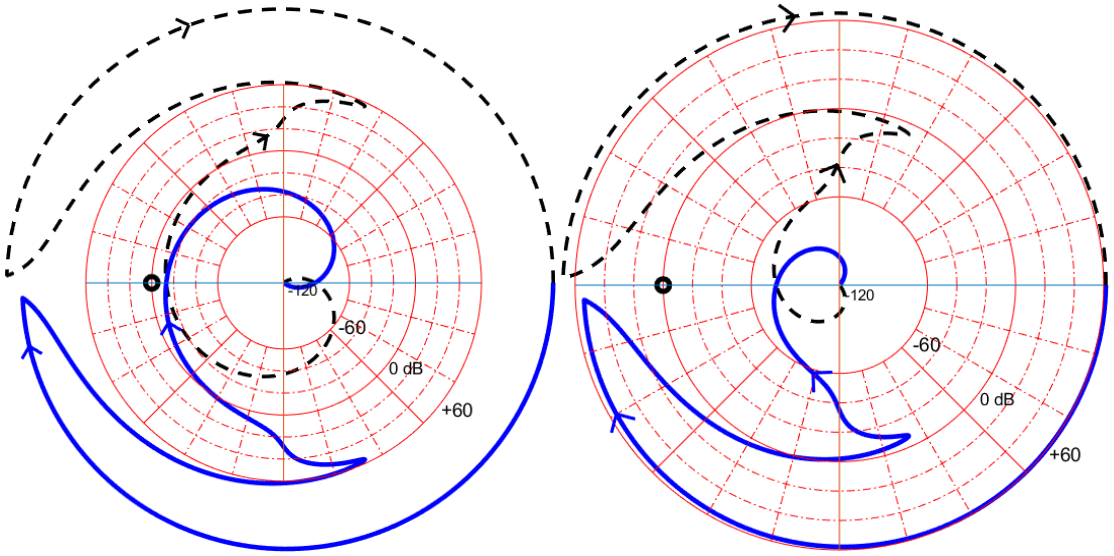

Fig 9. Phase Margin Improved

The left is the one of D*G*H and the right is the one of K*D*G*H. As the graphs show, the phase margin is improved because the -1 point now has more room to move.

Step 7:

Use step response of closed-loop transfer functions to choose a master gain K suitable to our RCG (fast and accurate). The figure below is produced in Matlab for comparison of different Ks.

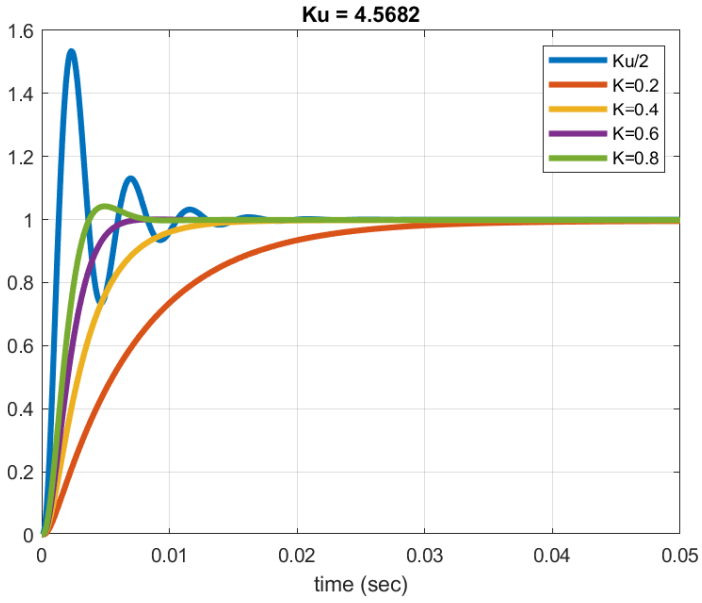

Fig 10. Step Response

All of the graphs settle in less than 0.05 seconds, which is favorable for our RCG. We also want to minimize the overshoot, so K should be at least less than 0.8. We chose K=0.4 for a good compromise between speed, accuracy and stability.

Step 8:

Use Simulink for heuristic tuning.

The systems have been identified in 1.2. Assemble them in Simulink as the figure below and run the model.

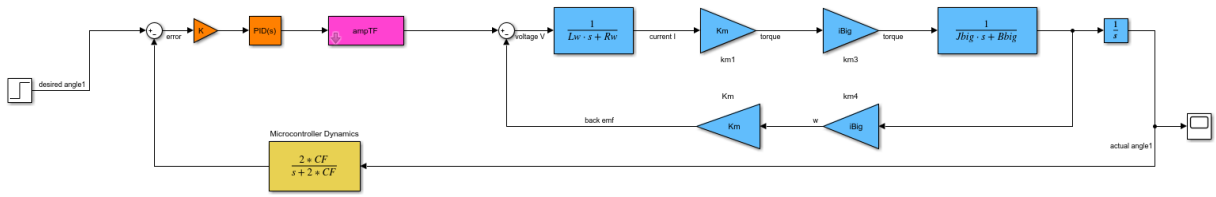

Fig 11. System

The wave we get from step 7 is already chosen to be the desired response. As the figure shows, it settles in 0.02 seconds with no overshoot.

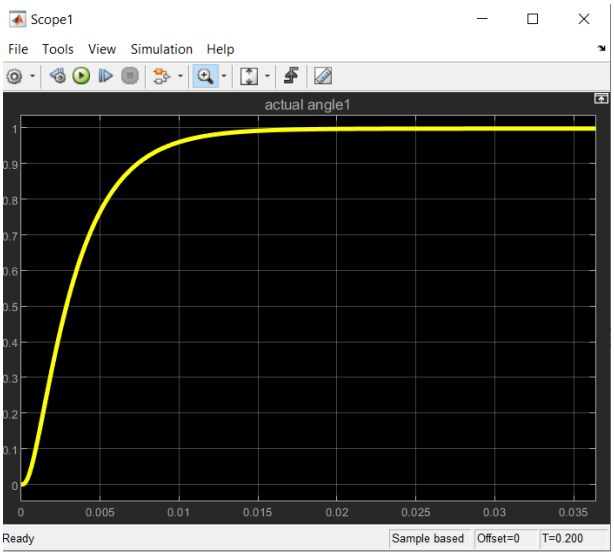

Fig 12. Step Response

The K values are:

K = 0.4, Kp = 13.3, Ki = 1, Kd = 44.44

Step 9:

Now the non-linearities are brought back to the model.

The amplifier circuit has saturation effect on the voltage. It is modeled in Simulink as a saturation block.

The sensor can only output whole number counts, which is modeled as a gain that converts counts back to radians after the floor function after a gain that converts radians to counts.

The dynamic friction is linear time-invariant and is taken into account by the mechanical impedance, but static friction is dependent on past input. Thus, static friction is represented as a Matlab function (appendix).

After adding the non-linearities, the Simulink model is as following.

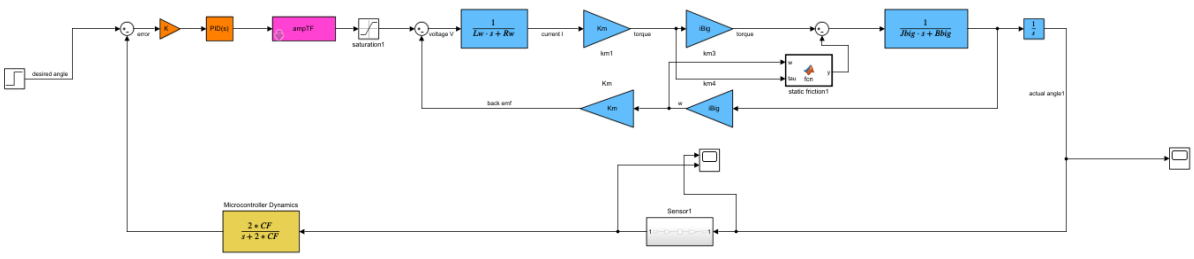

Fig 13. System

Use the system with non-linearities to get step response. The figure is the step response graph, which now has a very long settle time and large steady state error.

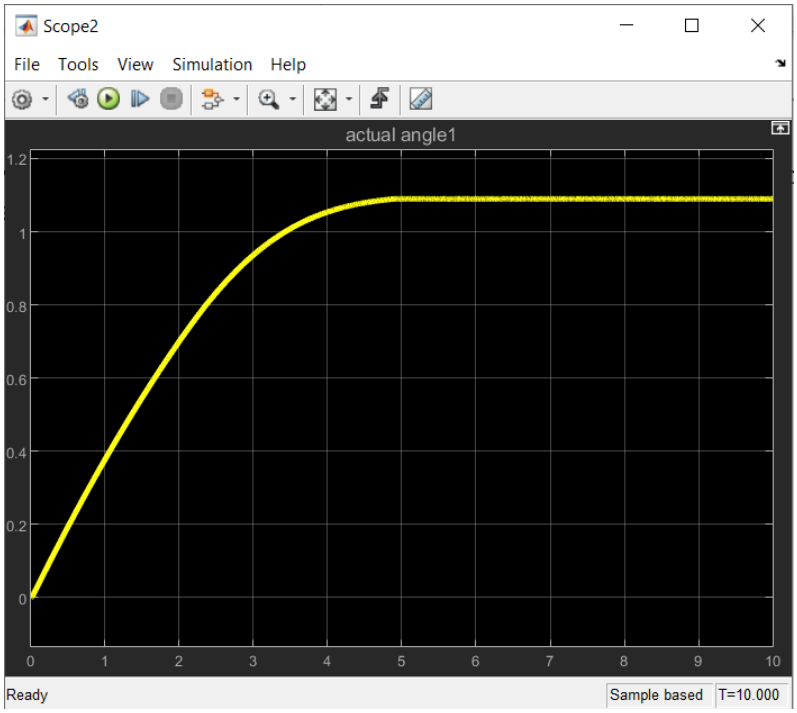

Fig 14. Step

The saturation slows down the rise time (and by extension settle time). The sensor introduces more steady state error.

Do heuristic tuning. The tuning should increase Kp (steady state error and rise time). First, because it is too slow, the master gain K was increased.

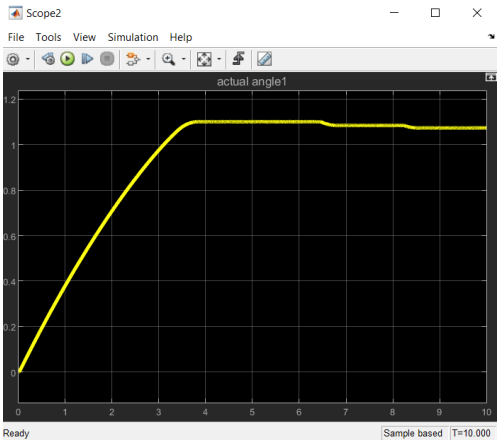

Fig 15. Tuning

Increase K from 0.4 to 3, the figure is the step response. Then Kp should be increased to lower the steady state error.

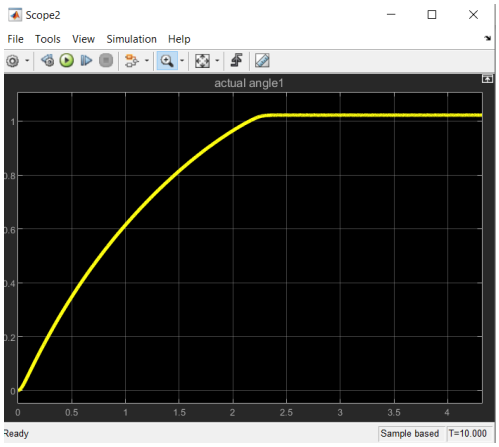

Fig 16. Tuning

When Kp is around 50 the graph is similar to the figure above.

Finally, we adjust Kd.

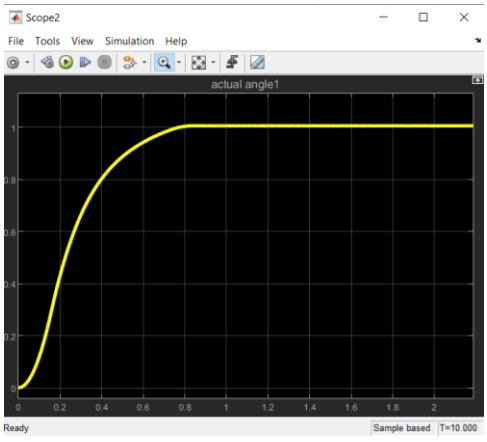

Fig 17. Tuned

After the first heuristic tuning, the step response is as shown in the figure. The response has good settle time and low overshoot, which satisfies our RCG.

The final test is the moving target. It sees if we have a good rise time, which is also what we want. Now, the input is a sinusoidal wave.

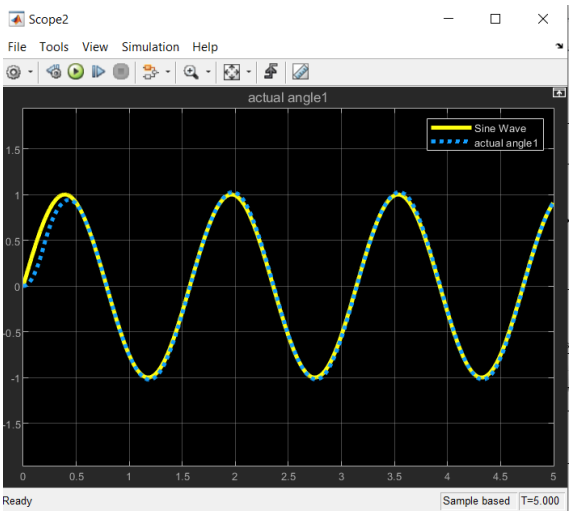

Fig 18. Moving Target

The figure shows the result of the test. After the startup, the output signal only lags the input signal a little. This shows that the system has a fast response, which meets our RCG.

The K values are

K = 3, Kp = 50, Ki = 1, Kd = 10.

1.5.2 Joint 2

The methods used for joint 2 are the same as joint 1.

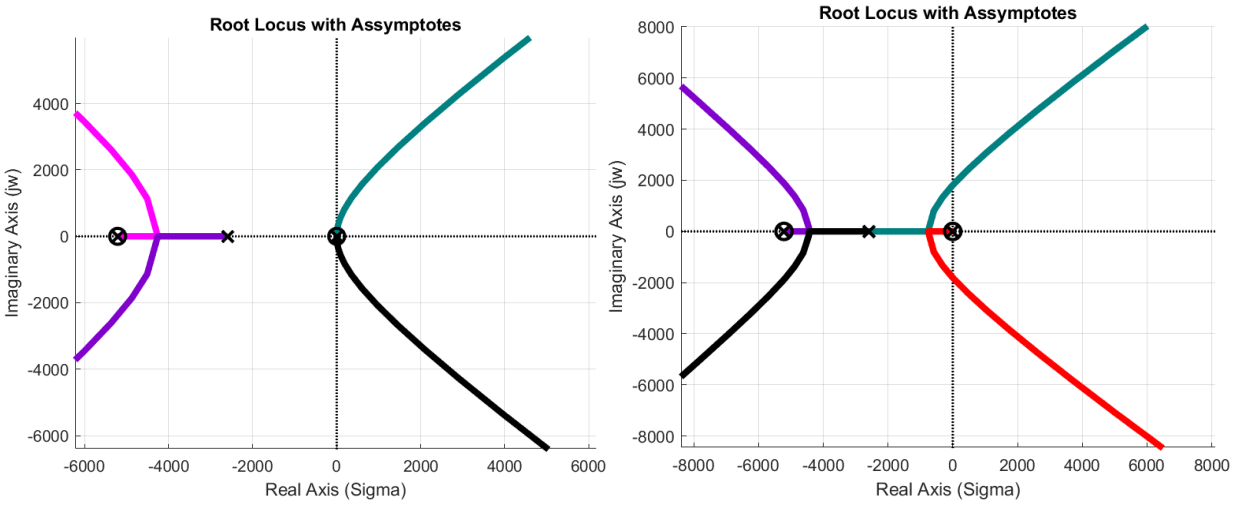

Fig 19. Root Loci

Using z = 1.2, the open-loop root locus of D*G*H (right figure) is improved compared to G*H (left figure).

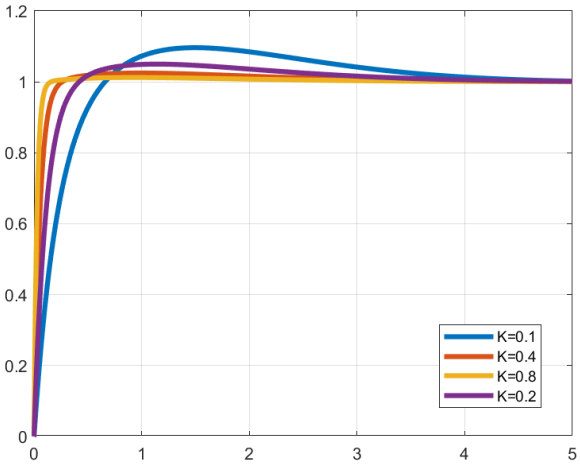

Fig 20. Step

Choose K=0.4 because it has a lower overshoot.

Do heuristic tuning.

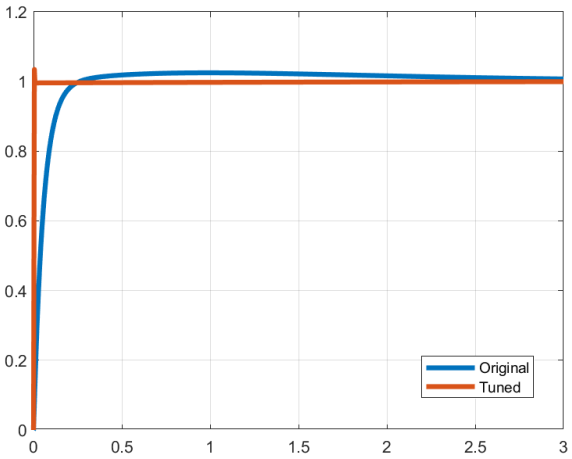

Fig 21. Tuned

Compared to the original, tuned response is faster.

Now, K = 0.4, Kp = 8.7, Ki = 1, Kd = 18.9.

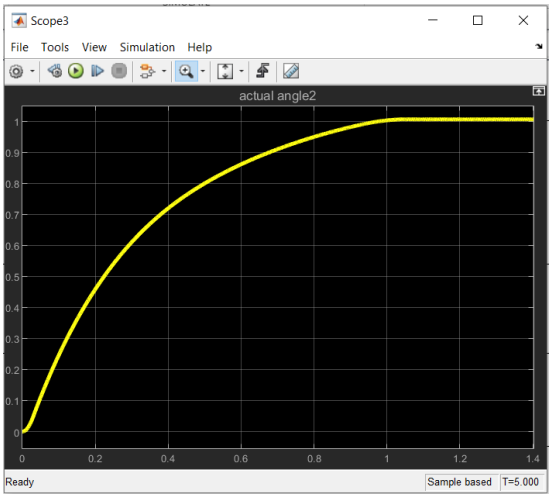

Fig 22. Tuned

This figure shows the tuned response in non-linear system. The settle time is good for our goal. The overshoot is small.

K = 3, Kp = 40 Ki = 1, Kd = 12.

1.6 Heuristic Tuning

The Simulation X system is as the figure.

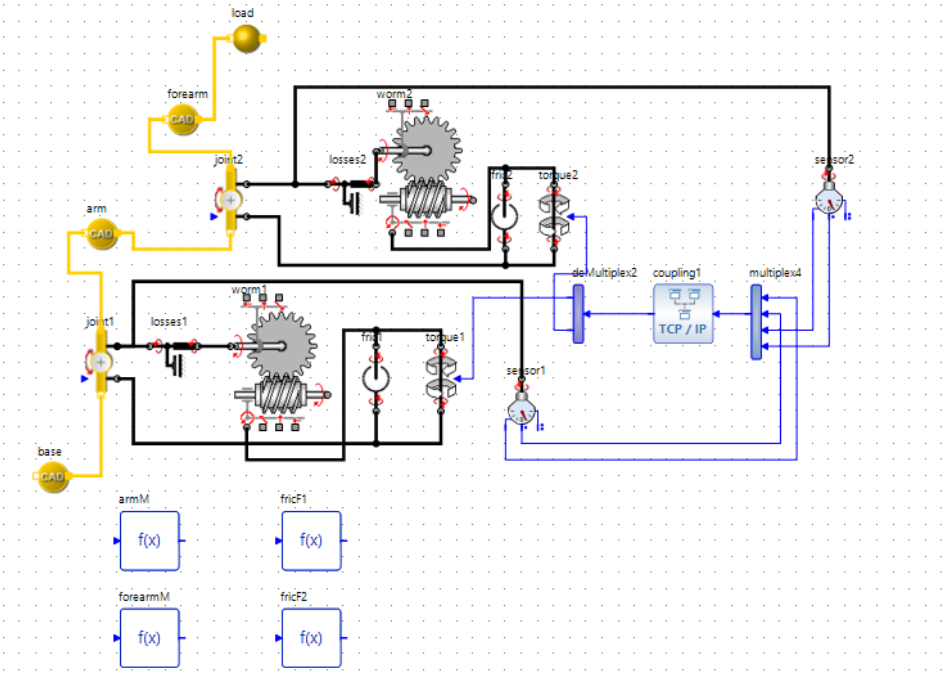

Fig 23. Simulation X System

Using Simulink and Simulation X co-simulation, we can get more realistic mechanical system simulation. Give the angle at joint 1 (theta1) a step of 2 and the angle theta2 at joint 2 a step of 1. Observe the result (figure).

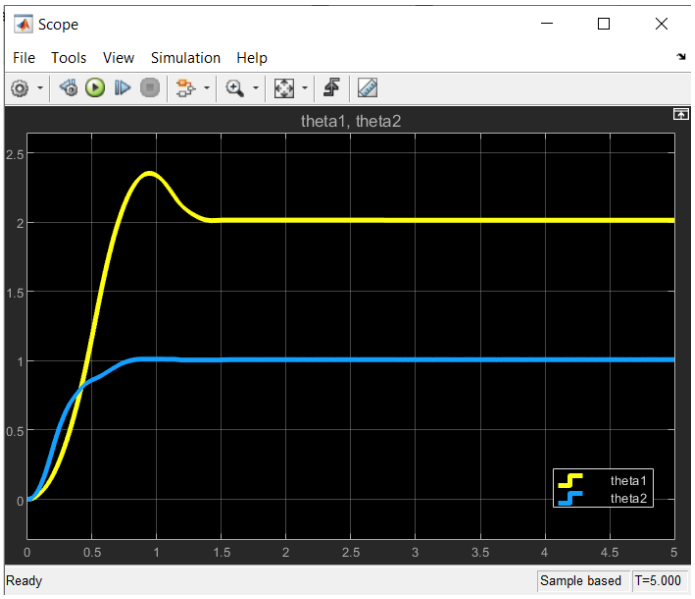

Fig 24. Tuning

The upper arm is heavier than we predicted. We have a large overshoot which forces us to do heuristic tune again. We would like to lower the value of Kp and Ki to reduce the overshoot. After tuning, the new K values for motor 1 is

K = 3, Kp = 40, Ki = 0.8, Kd = 15.

After tuning, the step response is as shown in the figure.

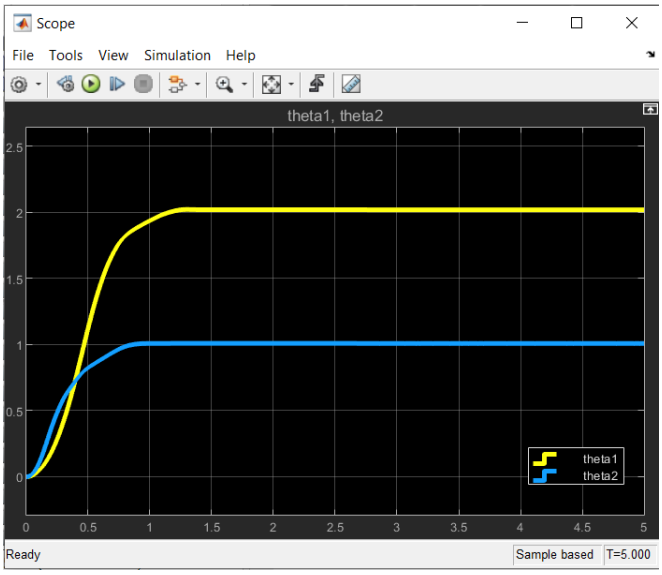

Fig 25. Tuned

The responses settle within 1.5 seconds. We evaluate our RCG and get our final K values.

Joint 1: K = 3, Kp = 40, Ki = 0.8, Kd = 15.

Joint 2: K = 3, Kp = 40 Ki = 1, Kd = 12.

2 Actuator

2.1 RCG

We established RCG for our actuator design based on the technical details of the product (**Yamaha YK800XGP**).

requirement:

- joint 1 must have angular acceleration >=0.25 rad/s^2
- joint 2 must have angular acceleration >=0.8 rad/s^2
- payload must be >=15kg

constraint:

- two motoes cost must not be higher than CA$500.

goal:

- maximize payload
- minimize cost

2.2 Motor

2.2.1 Joint 1

The design aims to pick up a 15 kg load. (two arms combined length is 600mm)

So, using the formula J=m*r^2

Load Inertia is: Jload = 15kg*0.6m^2 = 5.4 Nm

Using Solidworks mass property function we get arm inertia is:

Jarm = 0.42136

So the interia that joint 1 can feel is:

Jall = Jarm + Jload = 5.821 Nm

According to our design goal, we want our robot arm's angular acceleration(a) to be 0.25 rad/s^2.

Using formula: torque = Jall* a = 1.4525 Nm

For joint1, we choose to use a worm gear and the gear ratio is 30.

Torque from motor = torque/30 = 0.0485Nm

In this way, we need a motor which can provide 0.0485Nm torque.

So, we choose the DCX26L EB KL 24V motor for joint 1.

DCX26L EB KL 24V motor can provide us 52.30 mNm torque which is higher than the requirement.

2.2.2 Joint 2

The design aims to pick up a 15 kg load. (fore arm length is 250 mm)

So, using the formula J=m*r^2

Load Inertia is: Jload = 15kg*250mm^2 = 0.9375 Nm

Using SolidWorks mass property function we get arm inertia is:

Jarm = 0.0497

So, the inertia that joint 1 can feel is:

Jall = Jarm + Jload =0.9872 Nm

According to our design goal, we want our robot arm's angular acceleration(a) to be 0.8 rad/s^2.

Using formula: torque = Jall* a = 0.789776 Nm

For joint1, we choose to use a worm gear and the gear ratio is 20.

Torque from motor = torque/20 = 0.039488Nm

In this way, we need a motor which can provide 0.039488Nm torque.

So, we choose the DCX26L EB KL 24V motor for joint 2.

DCX26L EB KL 24V motor can provide us 52.30 mNm torque which is higher than the requirement.

2.3 Gear

2.3.1 Joint 1

The design aims to pick up a 15 kg load.

From the calculation, 1.4525 Nm torque is needed for shoulder joint1. However, the motor we choose (DCX26L EB KL 24V motor) can only provide us with 52.30 mNm torque.

We decided to use a worm gear whose gear ratio is 30.

Worm gears are used to transform the torque along x-axis to z-axis and increase the torque to drive the arms.

2.3.2 Joint 2

The design aims to pick up a 15 kg load. From the calculation, 0.9872 Nm torque is needed for shoulder joint2. However, the motor we choose (DCX26L EB KL 24V motor) can only provide us with 52.30 mNm torque.

We decided to use a worm gear whose gear ratio is 20.

Worm gears are used to transform the torque along x-axis to z-axis and increase the torque to drive the arms.

3 C code

3.1 PID

```
void PID(double speed_target, double current_speed, bool motor_id) {
```

The procedure PID accepts desired speed and actual speed to control.

```
double error = speed_target - current_speed; // error calculation
```

Calculate error, then calculate the PID branches respectively.

```
double  ntegral = +(error + old_error) * deltaT / 2.0; // I
double ed = (error - old_error) / deltaT;              // edot
double x = Kp * error + Ki *  ntegral + Kd * ed;
```

```
double pwn_duty = ((x / max_speed) * 255 > 255) ? 255 : (x / max_speed) * 255; //
calculate duty cycle
double direction = speed_target > current_speed ? 1 : 0;                       //
direction
```

We are using PWN to control the motor speed.

Finally, we instruct the driver circuit to product the PWM.

```
int motor_dir_pin1 = motor_id ? direction_control1_1 : direction_control2_1;   //
set the pins so this function is modularized
int motor_dir_pin2 = motor_id ? direction_control1_2 : direction_control2_2;
int motor_pwm_pin = motor_id ? PWM1 : PWM2;
digitalWrite(motor_dir_pin1, direction); // H bridge need to reverse for other
half bridge
digitalWrite(motor_dir_pin2, !direction);
analogWrite(motor_pwm_pin, pwn_duty);
```

3.2 Homing Logic

First, we need a flag to indicate the homing is done, as homing only needs to be done once at start.

```
static bool homingdone;
```

After start, if our two sensors are not at the same phase (the relative sensor is not home), we need to do homing.

```
while (phase1 <= phase2 * 1.01 && phase1 >= phase2 * 0.99) // allowing 1% error
{
```

Depending on the position, we adjust the arms either forward

```
    while (phase1 < phase2) // if the motor is in the backward position
    {
        forward();
    }
    stopMotor();
    sleep(1000);
```

or backward.

```
    while (phase1 > phase2) // if the motor is in the forward position
    {
        reverse();
    }
    stopMotor();
    sleep(1000);
```

Then we continue the loop.

```
}
homingdone = true; // homing done flag
```

3.3 Sensor Logic

Sensor outputs count instead of actual angle. So, we need to recover the angles.

First, we know from 1.2.4 the least detectable angle is 1/16 degrees.

```
#define RESOLUTION 16
```

The procedure sensor accepts the reading (count) and returns an angle in radians.

```
double sensor(int count) {
```

Next, we read the angle in degrees.

```
    double deg = ((double)count) / RESOLUTION;
```

Finally, we return the radians.

```
    return deg / 180 * 3.14;
}
```

3.4 Inverse Kinematics

The inverse kinematics is implemented and tested in Matlab.

```
function y = fcn(x1,y1)
    arm1 = 0.35; %350mm upper arm length
    arm2 = 0.25; %250mm forearm length
```

With known arm length, implement the inverse kinematics algorithm [3].

```
    b2 = (x1^2 + y1^2 - arm1^2 - arm2^2)/(2*arm1*arm2);
    s2 = sqrt(1-b2^2);
    angle1 = atan2(arm2^2*s2*x1 + (arm1 + arm2*b2)*y1, (arm1 + arm2*b2)*x1 -
arm2*s2*y1);
    angle2 = -acos(b2);
```

Return the two angles.

```
    y = [angle1,angle2];
```

3.5 Direct Kinematics

The direct kinematics is implemented and tested in Matlab.

The direct kinematics recovers the x and y coordinates given two angles. Since we know the direction of the angles, it is easy to know the coordinates.

```
function coord = fcn(angle1,angle2)
    arm1 = 0.35; %350mm upper arm length
    arm2 = 0.25; %250mm forearm length

    x1 = arm1*cos(angle1) + arm2*cos(angle1+angle2);
```

```
y1 = arm1*sin(angle1) + arm2*sin(angle1+angle2);

coord = [x1,y1];
```

3.6 Testing

Next, we test the output of inverse/direct kinematics, the arm goes to (0.6,0), (0,0.5), (-0.5,0.2) and back.

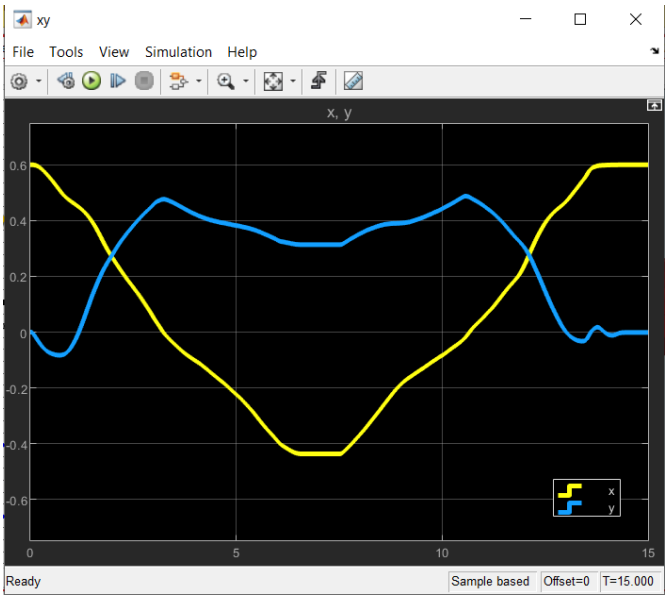

Fig 26. Kinematics

The figure shows the x and y coordinates. We can see that the inverse and direct kinematics successfully control the path.

References

[1] Maxon catalog:
https://canvas.ubc.ca/courses/83907/files/18304886?module_item_id=3937630

[2] E4P Encode Data Sheet: https://people.ece.ubc.ca/leos/pdf/e391/Info/E4P_datasheet.pdf

[3] Inverse Kinematics in 2D: https://www.alanzucconi.com/2018/05/02/ik-2d-2/

Appendix

static friction

```
function y = fcn(w,tau)
    Mg = 97.96;
    mu = 0.002;
    tau0 = Mg * mu;
    stuck = (w == 0) & (abs(tau) < tau0);
    if stuck
        y = tau;
    else
        y = 0;
    end
end
```