

Lab 3. Motor Control

1. Purpose

Lab 3의 목적은 ROBOTIS' Dynamixel을 활용하여 encoder data를 입력받고 feedback position control을 구현하는데 있다. Motor control과 관련된 classic control theory를 C++ program을 통해 구현하고 실험을 통해 확인한다. 다양한 변수들을 변화시키며 확인하여 motor control에 관련된 이론과 실제 실험 결과가 consistent한지 확인한다.

2. Experiment Sequence

본 lab은 7가지 problem으로 구성되어 있다. 전체적인 lab의 내용은 motor control을 위해 Beaglebone과 dynamixel U2D2를 연결하고 motor control을 할 수 있는 C++ 코드와 결과를 확인할 수 있는 Matlab 코드 등을 구현, development environment를 구성한다. 그 다음 velocity measurement를 위해 numerical differential을 구현하고 low pass filter를 통해 noise amplification을 어떻게 조절할 수 있는지 확인하다. 다음으로 Dynamixel SDK를 활용하여 motor를 control할 수 있는 코드를 구현, position, velocity, current mode를 통해 motor를 각각 control 해본다. 위의 과정을 성공적으로 진행한 후 PD controller, PID controller를 step input에 대해 구현하고 P gain, I gain, D gain이 control에 미치는 영향을 분석한다. 마지막으로 PID controller를 sinusoidal input에 대해 test하여 적절한 gain 값을 찾고, chirp signal에 대해 test하여 frequency response에 대해 이해한다. 아래는 구현하게 될 7가지 problem이다.

(1) Problem 3A. Development Environmental Setup

Development environment를 구성, 데이터를 저장하기 위한 file input/output을 설정하고 데이터 해석을 위한 plotting을 실시한다. 이를 통해 motor angle value와 motor velocity value를 얻는다.

(2) Problem 3B. Velocity Measurement (Numerical differentiation with LPF)

Motor의 velocity measurement를 numerical differentiation을 통해 얻고 Low pass filter를 통해 noise amplification을 조절한다.

(3) Problem 3C. Practicing Position, Velocity and Current Control

Dynamixel SDK를 활용하여 motor control을 위한 코드를 완성하고 position, velocity, current mode를 활용하여 motor를 control 해보고 각각의 특징을 확인한다.

(4) Problem 3D. PD Position Controller (reference position: step input)

PD controller를 구현하고 P gain과 D gain 값 변화에 따라 어떤 경향성을 보여주는지 step input에 대해 확인해본다.

(5) Problem 3E. PID Position Controller (reference position: step input)

PID controller를 구현하고 I gain의 의미와 PD controller와 비교해 control에 주는 영향을 step input에 대해 비교해보며 확인한다.

(6) Problem 3F. PID Position Controller (reference position: sinusoidal input)

PID controller를 이용해 sinusoidal input에 대해 test해보고 결과를 확인한다.

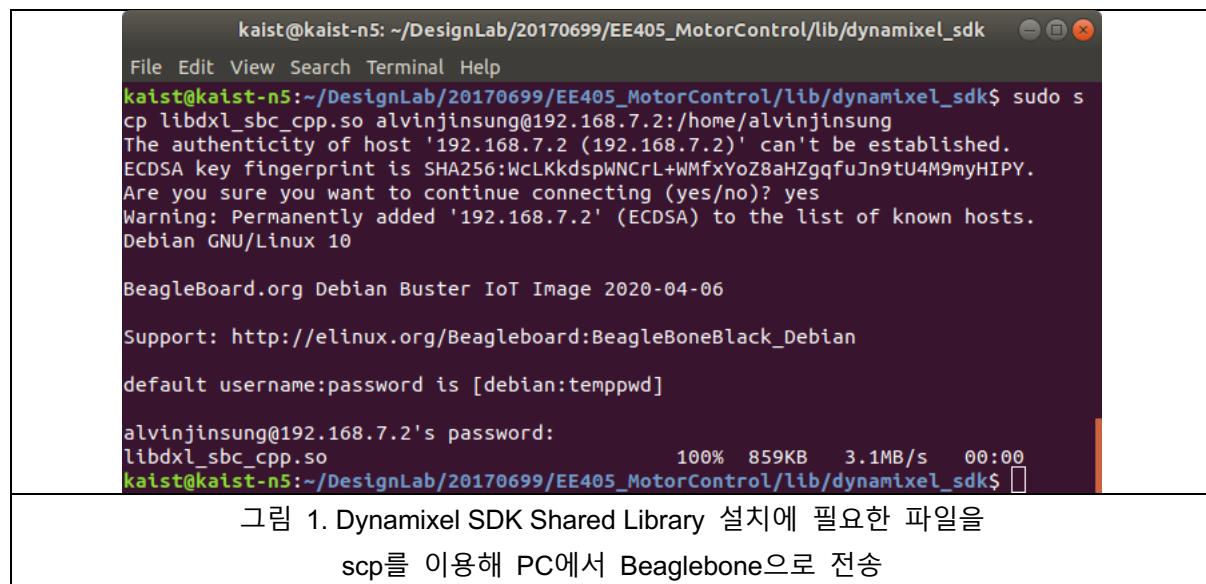
(7) Problem 3G. Understanding Frequency Response (with Chirp Signal)

PID controller를 이용해 chirp signal에 대해 test 해보고 결과를 통해 frequency response를 이해한다.

3. Experiment Results

Problem 3A. Development Environment Setup

가장 먼저 development environment를 구성하였다. PC와 Beaglebone을 키고 연결하였으며 PC와 Beaglebone 사이 NFS를 구성하였다. 그 다음 Beaglebone에 Dynamixel SDK Shared Library를 아래와 같이 설치하였다.



```
kaist@kaist-n5: ~/DesignLab/20170699/EE405_MotorControl/lib/dynamixel_sdk
File Edit View Search Terminal Help
kaist@kaist-n5:~/DesignLab/20170699/EE405_MotorControl/lib/dynamixel_sdk$ sudo s
cp libdxl_sbc_cpp.so alvinjinsung@192.168.7.2:/home/alvinjinsung
The authenticity of host '192.168.7.2 (192.168.7.2)' can't be established.
ECDSA key fingerprint is SHA256:WcLKKdspWNCrL+WMfxYoZ8aHZgqfuJn9tU4M9myHlPY.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.7.2' (ECDSA) to the list of known hosts.
Debian GNU/Linux 10

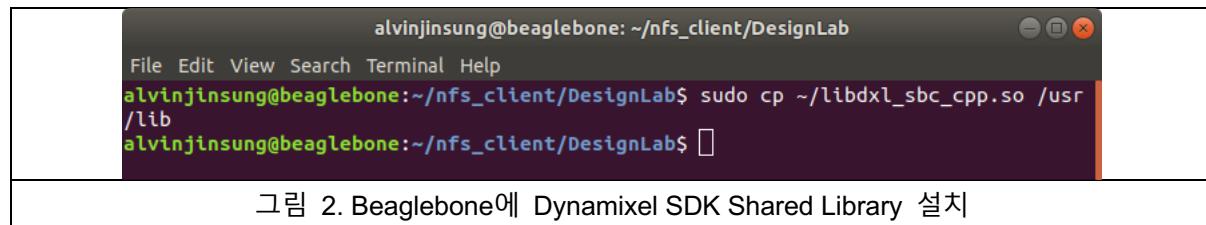
BeagleBoard.org Debian Buster IoT Image 2020-04-06

Support: http://elinux.org/Beagleboard:BeagleBoneBlack_Debian

default username:password is [debian:temppwd]

alvinjinsung@192.168.7.2's password:
libdxl_sbc_cpp.so          100%  859KB   3.1MB/s   00:00
kaist@kaist-n5:~/DesignLab/20170699/EE405_MotorControl/lib/dynamixel_sdk$
```

그림 1. Dynamixel SDK Shared Library 설치에 필요한 파일을
scp를 이용해 PC에서 Beaglebone으로 전송



```
alvinjinsung@beaglebone: ~/nfs_client/DesignLab
File Edit View Search Terminal Help
alvinjinsung@beaglebone:~/nfs_client/DesignLab$ sudo cp ~/libdxl_sbc_cpp.so /usr
/lib
alvinjinsung@beaglebone:~/nfs_client/DesignLab$
```

그림 2. Beaglebone에 Dynamixel SDK Shared Library 설치

다음으로 Beaglebone과 dynamixel U2D2, power supply를 아래와 같이 연결하였다.



그림 3. Beaglebone과 dynamixel U2D2를 연결한 모습

올바른 build를 위해 CMakeLists.txt 파일을 수정하였다. Pre-installed된 package를 find_package()를 통해 CMake가 자동으로 찾을 수 있도록 해주었으며, include_directories()를 통해 CMake가 header 파일을 찾기 위해 살펴보아야 하는 directory를 설정하였다. link_directories()를 통해 CMake가 library file을 찾기 위해 살펴보아야 하는 directory를 설정하였다. add_library()를 통해 static library file을 생성하였다. add_executable()을 통해 executable을 설정하였다. 변형한 CMakeLists.txt파일은 아래와 같다.

```
cmake_minimum_required(VERSION 3.10)
project(EE405_MotorControl LANGUAGES CXX)

add_compile_options(-std=c++17)

### find package: pre-installed library The CMake can automatically find following
installed package.
find_package(Eigen3 REQUIRED)
find_package(Threads REQUIRED)

### include_directories: We can set include directories, The CMake can look for the header
file now in this path.
include_directories(${EIGEN3_INCLUDE_DIRS}
    include
    include/dynamixel_sdk
    src)

### link_directories: We can set link directories, The CMake can look for the library file
now in this path in order to link library and executable file.
link_directories(
    lib
    lib/dynamixel_sdk
)

### add_library: How to make a static library file using the following code.
### If you want to create a shared library, change 'STATIC' to 'SHARED'.
### However, for cross-compiling, it is not recommended to change to 'SHARED'.
add_library(EE405_robot_manipulator STATIC include/Controller/ArticulatedSystem.cpp
    include/FileIO/MatrixFileIO.cpp
    include/Common/LowPassFilter.cpp
    include/Common/NumDiff.cpp
)

### add_executable: Make executable file using following motor_control.cpp code.
add_executable(motor_control src/motor_control.cpp )
```

```
### target_link_libraries: link between exe file and library file.
### EE405_robot_manipulator: The static library which we make. please refer to
add_library().
### dxl_sbc_cpp: The dxl_x64_cpp shared library is from the Dynamixel SDK and enables motor
rotation in the program.
target link libraries(motor_control EE405_robot_manipulator dxl_sbc_cpp )
```

그림 4. CMakeLists.txt 파일

이를 이용해 compiler를 GCC arm-linux-gnueabihf로, compile mode를 Release mode로 설정하여 build하였다. Build directory에서 static library와 executable이 생성된 것을 확인하였고 object file이 생성된 것도 올바른 directory로 이동해 확인할 수 있었다.

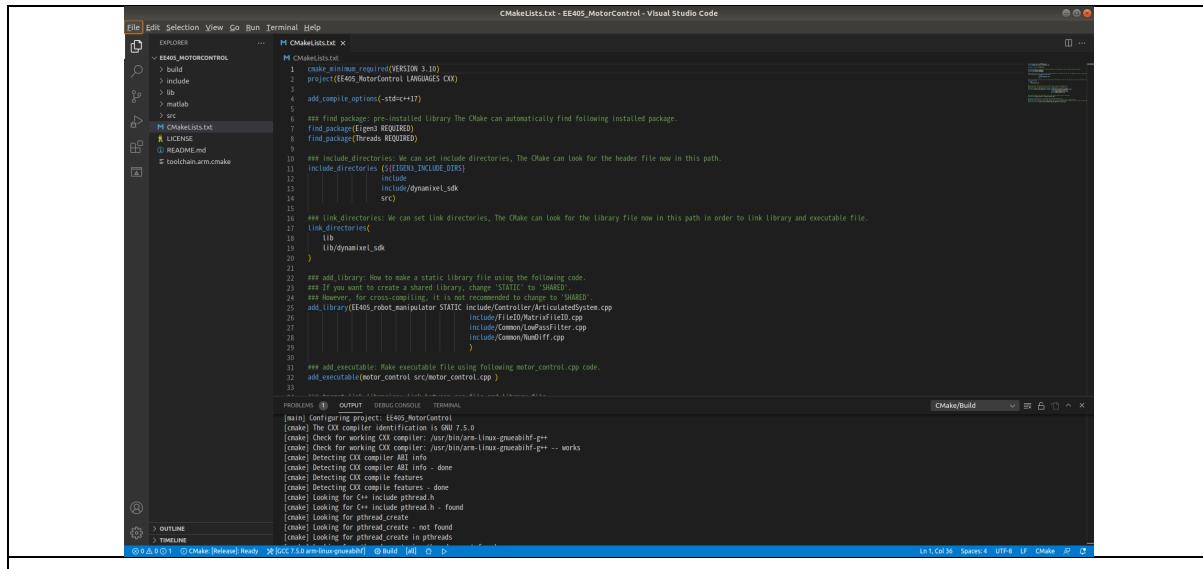


그림 5. Compiler: GCC arm-linux-gnueabihf, Compile mode: Release mode로 build한 모습

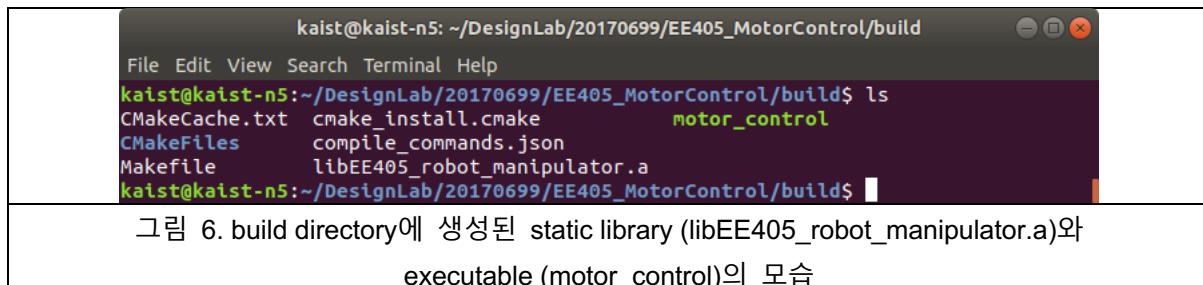


그림 6. build directory에 생성된 static library (libEE405_robot_manipulator.a)와 executable (motor_control)의 모습

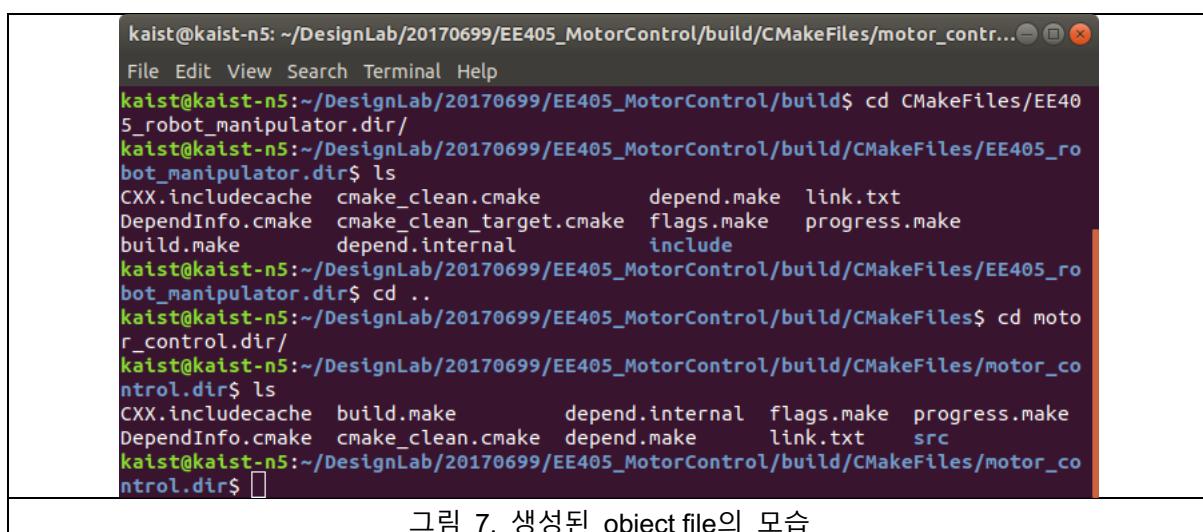
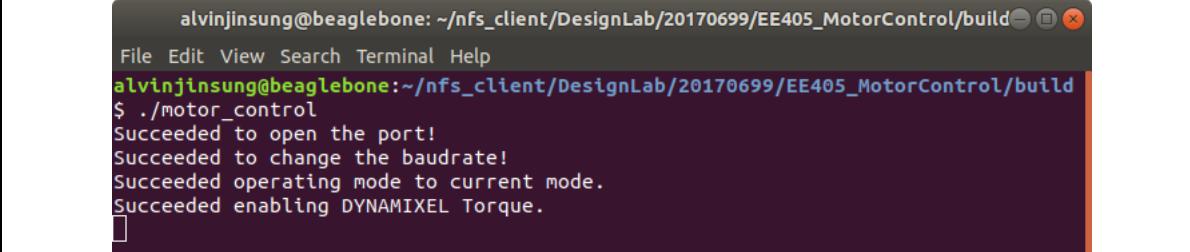


그림 7. 생성된 object file의 모습

Executable이 잘 생성된 것을 확인하였고 이를 실행해 보았다. 실행 후 손으로 motor를 직접 회전시켰으며 **ctrl+c**를 통해 program을 종료하였다.



```
alvinjinsung@beaglebone: ~/nfs_client/DesignLab/20170699/EE405_MotorControl/build
File Edit View Search Terminal Help
alvinjinsung@beaglebone:~/nfs_client/DesignLab/20170699/EE405_MotorControl/build
$ ./motor_control
Succeeded to open the port!
Succeeded to change the baudrate!
Succeeded operating mode to current mode.
Succeeded enabling DYNAMIXEL Torque.
```

그림 8. motor_control executable을 beaglebone에서 실행한 모습



그림 9. 손으로 motor를 회전시키는 모습

프로그램 종료 후 build directory 내부에 recorded_data.csv라는 csv 파일이 생성된 것을 확인할 수 있었다. 이 파일은 실행시간, current position(angle), current velocity, target position(angle), target velocity, target torque 등의 정보를 저장하고 있다.

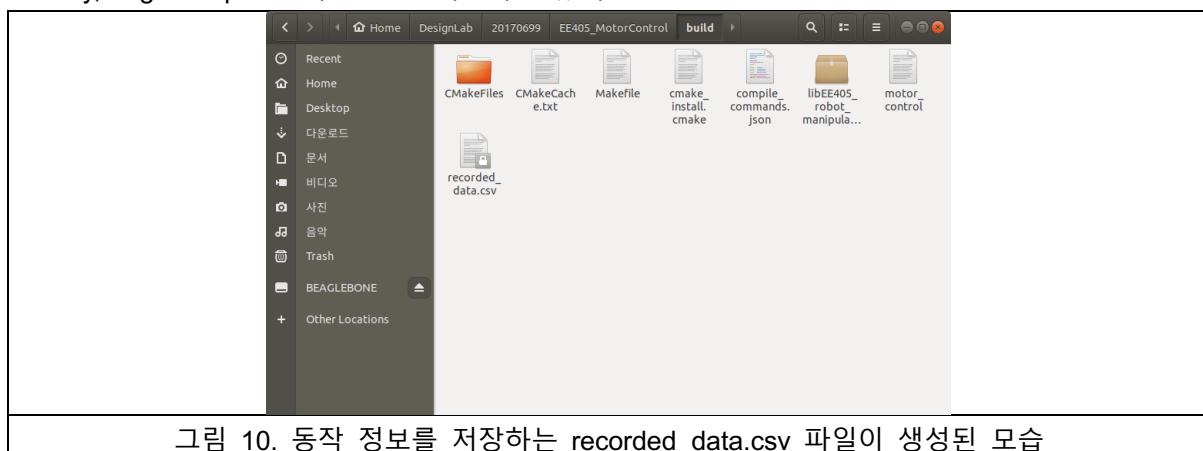


그림 10. 동작 정보를 저장하는 recorded_data.csv 파일이 생성된 모습

이렇게 생성된 정보를 통해 시각적으로 동작 과정을 확인하기 위하여 Matlab 코드를 이용해 csv 파일을 plot하였다. 주어진 plot_data.m 파일을 이용해 동작 과정에 대한 정보들을 그래프로 확인할 수 있었다.

```

kaist@kaist-n5: ~/DesignLab/20170699/EE405_MotorControl/matlab
File Edit View Search Terminal Help
kaist@kaist-n5:~/DesignLab/20170699/EE405_MotorControl$ cd matlab/
kaist@kaist-n5:~/DesignLab/20170699/EE405_MotorControl$ matlab
MATLAB is selecting SOFTWARE OPENGL rendering.
Gtk-Message: 09:57:40.382: Failed to load module "canberra-gtk-module"

```

그림 11. 주어진 Matlab 코드 실행 모습

```

% This script plots the control loop time versus elapsed time.
% It reads data from a CSV file and plots it.
% The plot shows the control loop time in seconds over time.

% Set the path to the recorded data file
addpath('/home/kaist/DesignLab/20170699/EE405_MotorControl/build')
csv_title = 'recorded_data_test'; % Enter the name of the CSV file containing the data you want to check here.

% Load the data from the CSV file
data = load(csv_title + ".csv");

% Extract the elapsed time and control loop time
elapsed_time = data(:,1);
control_loop_time = data(:,2);

% Plot the control loop time versus elapsed time
figure(1)
plot(elapsed_time, control_loop_time, 'k.', 'MarkerSize', 9);
xLabel('Elapsed time [sec]')
yLabel('Control loop time [sec]')
grid minor
set(gca,'FontSize',13)

% Plot the control loop time versus elapsed time
figure(2)
plot(elapsed_time, control_loop_time, 'k.', 'MarkerSize', 9);
xLabel('Elapsed time [sec]')
yLabel('Control loop time [sec]')
grid minor
set(gca,'FontSize',13)

```

그림 12. plot_data.m 파일

Control loop time에 해당한 figure 1 그래프를 살펴보았다.

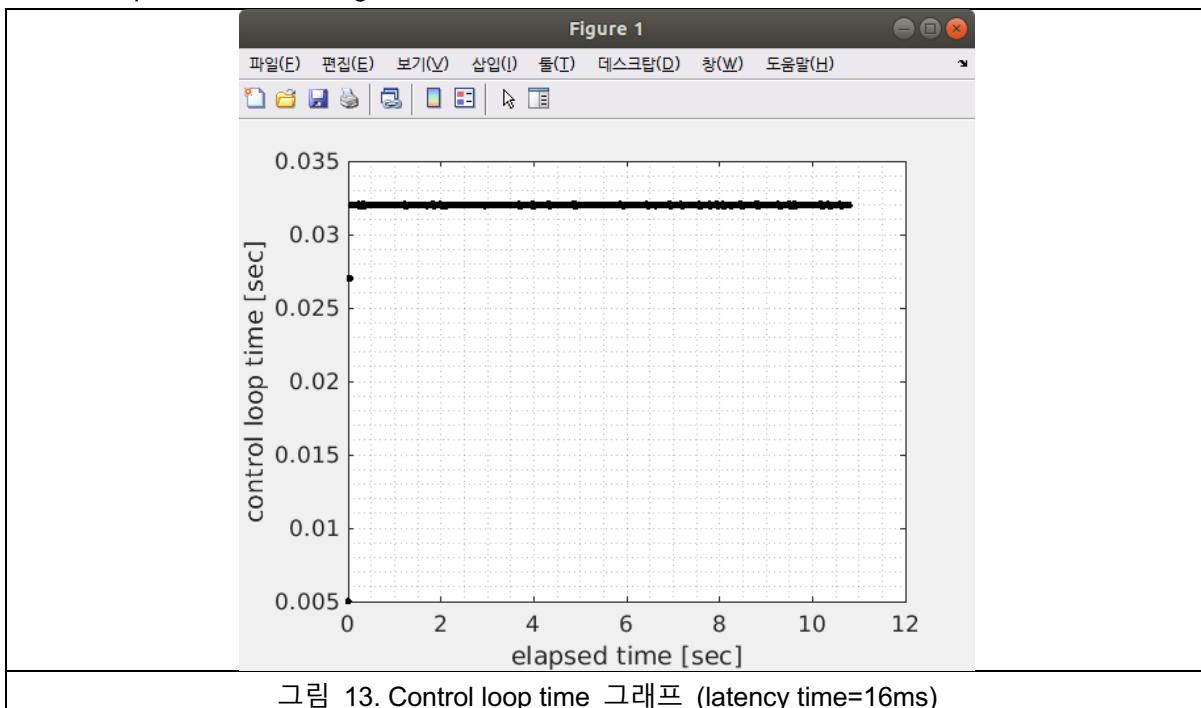


그림 13. Control loop time 그래프 (latency time=16ms)

위의 그래프에서 확인할 수 있듯이, 현재 각 step마다 0.032초가 소요되며 이는 control frequency가 200Hz가 아니라는 뜻이다. 이를 해결하기 위해 아래의 명령어를 입력하여 control frequency가 200Hz를 만족하도록 하였다.

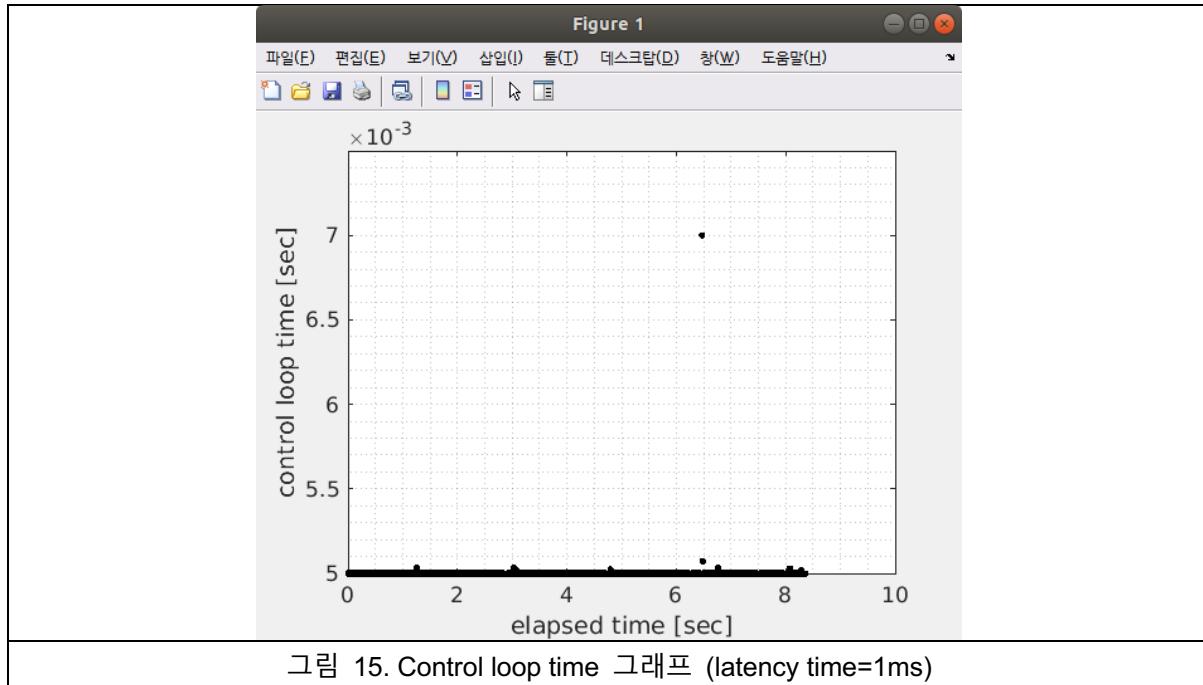
```

alvinjinsung@beaglebone: ~
File Edit View Search Terminal Help
alvinjinsung@beaglebone:~$ cat /sys/bus/usb-serial/devices/ttyUSB0/latency_timer
16
alvinjinsung@beaglebone:~$ echo 1 | sudo tee /sys/bus/usb-serial/devices/ttyUSB0
/latency_timer
[sudo] password for alvinjinsung:
1
alvinjinsung@beaglebone:~$ 

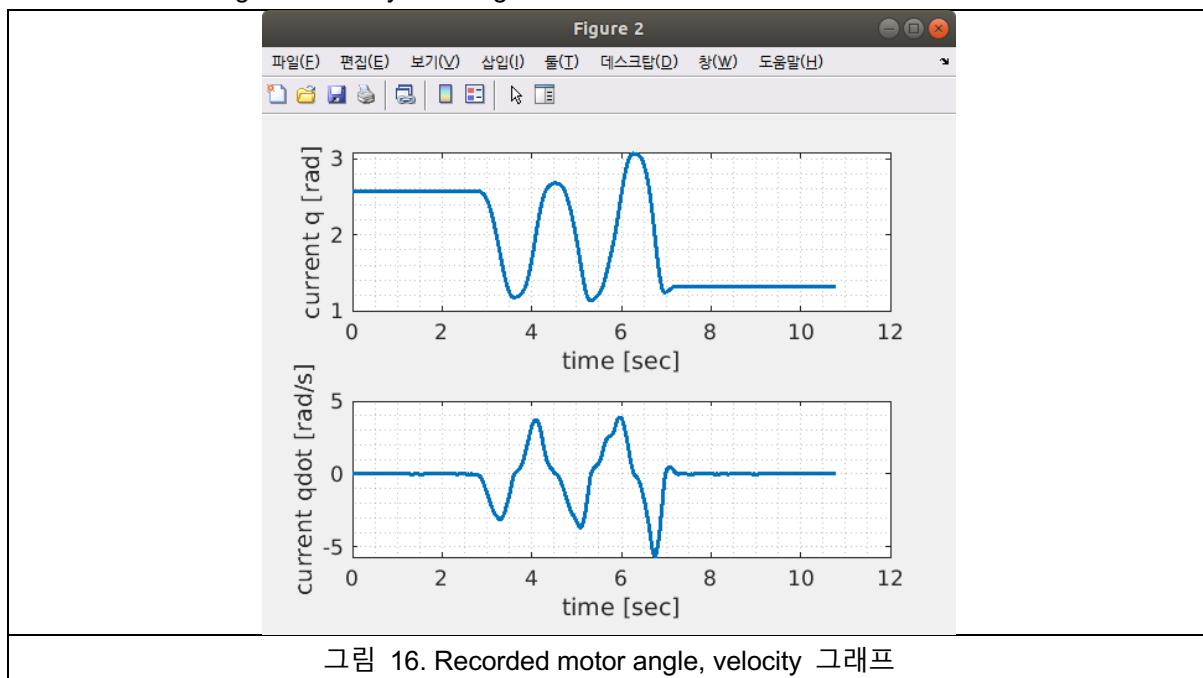
```

그림 14. Latency time을 16ms에서 1ms로 변경한 모습

변경한 이후 다시 실행하여 csv파일을 생성, Matlab을 통해 plot한 결과 아래와 같이 200Hz control frequency를 만족하는 것을 확인할 수 있었다.



Recorded motor angle과 velocity 역시 figure 2 그림에서 확인할 수 있었다.



Problem 3B. Velocity Measurement (Numerical Differentiation with Low pass filter)

Motor의 velocity measurement를 위해 numerical differentiation을 통해 계산하였다. Numerical differentiation을 이용한 velocity 계산은 아래와 같이 할 수 있다.

$$\dot{x} = \frac{d}{dt}x \approx \frac{x[k] - x[k-1]}{\Delta T}$$

이 때, ΔT 가 굉장히 작기 때문에 Quantization error가 더욱 증폭되게 된다. 그렇기 때문에 이러한 quantization error를 smoothing 해주지 않게 되면 실제 값과 매우 큰 차이를 보이게 된다. 따라서 Low pass filter를 적용한다. High-frequency component를 filtering 해주어 quantization error를 smotthing 해주는 효과를 지니게 된다. Numerical differentiation은 NumDiff.cpp에, low pass filter는 LowPassFilter.cpp에 구현하였으며 코드는 아래와 같다.

```
/*
 * @file NumDiff.cpp
 * @author Jiwan Han (jw.han@kaist.ac.kr)
 * @brief
 * @version 0.1
 * @date 2023-03-27
 *
 * @copyright Copyright (c) 2023
 *
 */
#include "NumDiff.hpp"

NumDiff::NumDiff(const Eigen::VectorXd& u, double dt_args)
: dt(dt_args)
{
    InitializeMemberValue(u);
}

void NumDiff::InitializeMemberValue(const Eigen::VectorXd& prev_u_args)
{
    prev_u = prev_u_args;
}

Eigen::VectorXd NumDiff::ComputeNumericalDerivative(const Eigen::VectorXd& u)
{
    Eigen::VectorXd y = (u-prev_u)/dt; /*This answer is incorrect, Remove y=u, Please
implement the proper output formula. */
    prev_u = u;
    return y;
}
```

그림 17. NumDiff.cpp 파일

```
/*
 * @file LowPassFilter.cpp
 * @author Jiwan Han (jw.han@kaist.ac.kr)
 * @brief
 * @version 0.1
 * @date 2023-03-27
 *
 * @copyright Copyright (c) 2023
 *
 */
#include "LowPassFilter.hpp"

LowPassFilter::LowPassFilter(const Eigen::VectorXd& init_matrix, double L_args, double
T_args)
: L(L_args), T(T_args)
{
    InitializeMemberValue(init_matrix, init_matrix, init_matrix);
}

void LowPassFilter::InitializeMemberValue(const Eigen::VectorXd& prev_u_args,
                                         const Eigen::VectorXd& y_args,
                                         const Eigen::VectorXd& prev_y_args )
```

```

{
    prev_u = prev_u_args;
    y = y_args;
    prev_y = prev_y_args;
}

Eigen::VectorXd LowPassFilter::FilterAndGetY(const Eigen::VectorXd& u)
{
    y = ((2-L*T)/(2+L*T)) * prev_y + ((L*T)/(2+L*T)) *(u+prev_u); /*This answer is
incorrect, Remove y=u, Please implement the proper output formula. */
    prev_u = u;
    prev_y = y;
    return y;
}

```

그림 18. LowPassFilter.cpp 파일

이를 build하여 실행하였고, motor를 손으로 돌려보며 test 하였다. 아래는 해당 test 과정에서 numerical differentiation을 이용해 구한 joint angular velocity, low pass filter를 적용한 결과, 그리고 Articulated System Class에서 GetJointVelocity() function을 이용한 결과를 plot한 것이다.

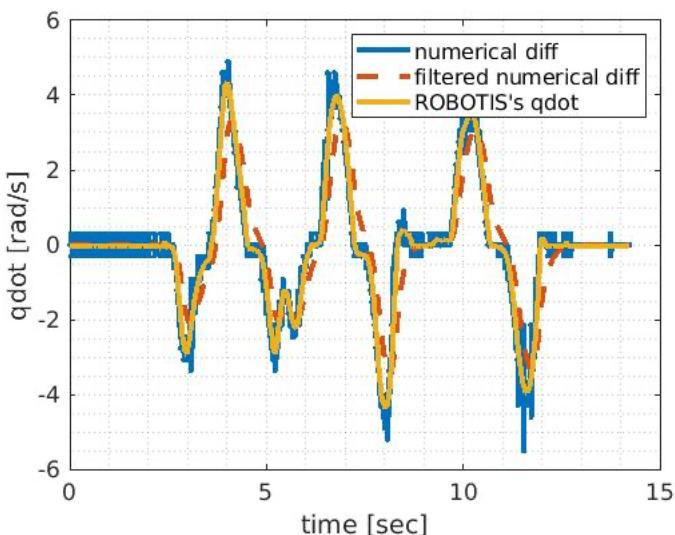
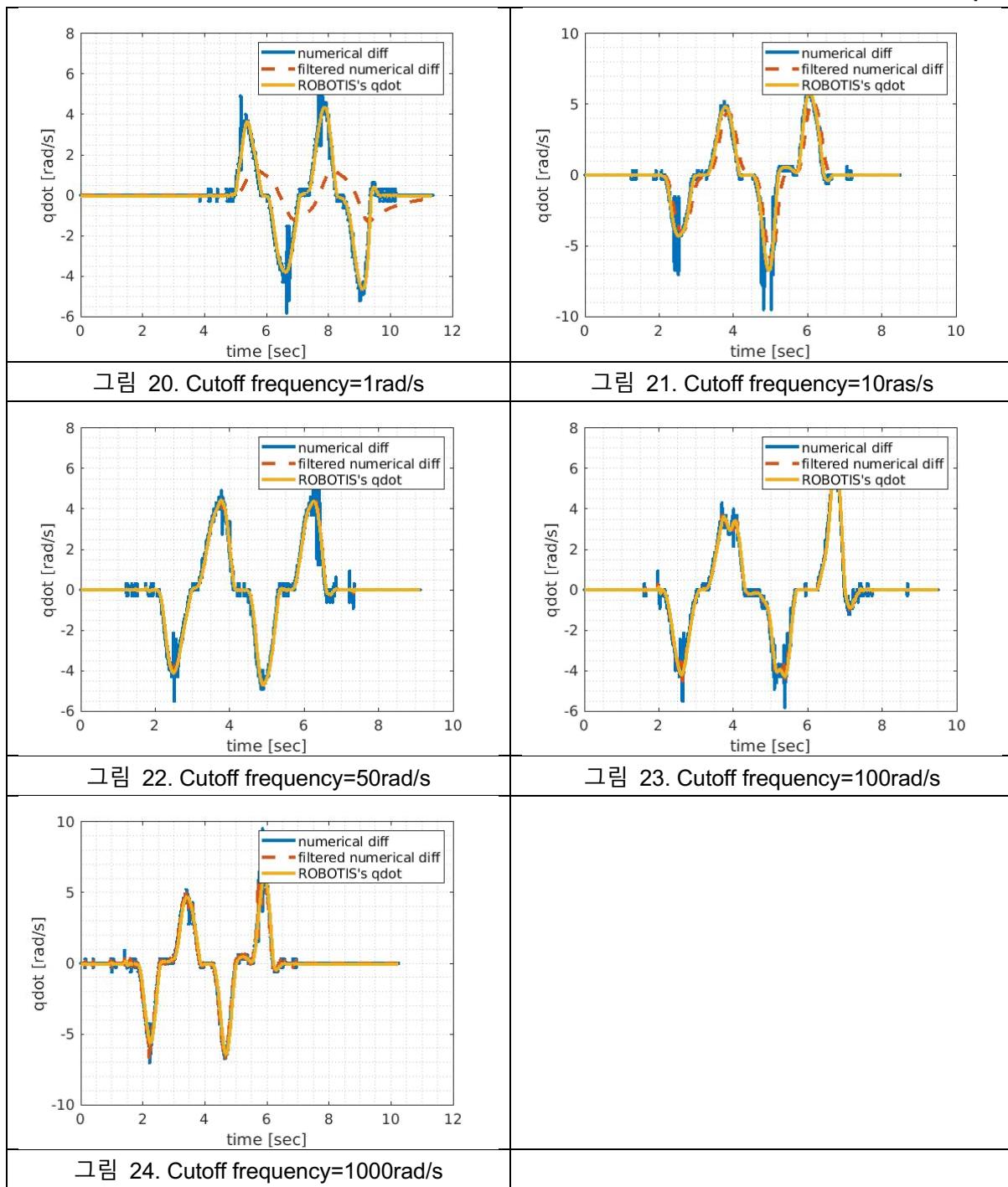


그림 19. Numerical differentiation, low pass filtered numerical differentiation, GetJointVelocity()을 이용해 얻은 joint angular velocity를 plot한 결과

위의 그림과 같이 numerical differentiation을 그대로 적용한 경우에는 quantization error가 증폭되고 있는 모습을 확인할 수 있었으나, low pass filter를 적용할 경우 GetJointVelocity()로 얻은 값과 유사하게 smoothing되는 것을 확인할 수 있었다. Low pass filter의 cutoff frequency는 gain이 -3dB 이 되는 시점의 frequency를 의미하는데 cutoff frequency에 따른 경향성을 확인하기 위해 cutoff frequency를 1rad/s, 10rad/s, 50rad/s, 100rad/s, 1000rad/s로 설정하고 그래프 경향성의 변화를 확인하였다. Cutoff frequency가 낮을 때는 noise가 아닌 부분에 대해서도 filtering이 되어 과도하게 smoothing 되는 것을 볼 수 있었다. Cutoff frequency가 점점 커짐에 따라 filtering되는 부분이 점점 줄어들어 50rad/s 정도에서는 GetJointVelocity()를 통해 얻은 값과 거의 유사하게 얻어지는 것을 볼 수 있었다. 그러나 cutoff frequency가 계속해서 커짐에 따라 noise들도 filtering이 되지 않고 numerical differentiation을 통해 얻은 결과에서 smoothing이 거의 이루어지지 않는 모습을 확인할 수 있었다. 즉, 원래 신호를 보존하면서 noise만을 filtering할 수 있는 적절한 cutoff frequency가 설정되어야 하며 이후 실험에서는 50dB로 설정하고 진행하였다. 아래는 각각의 frequency별로 plotting한 결과이다.



Problem 3C. Practicing Position, Velocity, and Current Control

Motor control을 하는데 target position 값을 받아 조작하는 position mode, target velocity 값을 받아 조작하는 position mode, target torque 값을 받아 조작하는 current mode가 있다. 각각의 mode 특징을 알기 위해 각각에 대해 motor control을 test 해보았다.

- Position control mode

Position mode로 변환하기 위해 motor_control.cpp 파일에서 ArticulatedSystem::Mode::POSITION

으로 설정하였다. 그 다음 targetQ(target position)을 1rad, targetQdot을 0rad/s로 설정하고 articulated_system.SetGoalPosition(target)를 실행하였다. 아래는 그 과정이다.

```
78     ArticulatedSystem articulated_system(dof, ArticulatedSystem::Mode::POSITION, "/dev/ttyUSB0",
79                                     dynamixel_id_set, position_resolution, motor_torque_constants); //1 DOF manipulator (In other words, one motor system)
```

그림 25. Position mode로 변경한 모습

157	////////// Constant target //////////
158	targetQ <= 1; // M_PI;
159	targetQdot <= 0;
160	////////// Constant target END //////////

그림 26. Target position을 1rad, target velocity를 0rad/s로 설정한 모습

167	////////// POSITION CONTROL //////////
168	articulated_system.SetGoalPosition(targetQ);
169	////////// POSITION CONTROL END //////////

그림 27. "targetQ"로 target position을 설정한 모습

이를 build하고 run한 결과 motor가 1rad에 해당하는 위치로 회전하고 멈추었다. 이때 motor를 회전시키려 손으로 힘을 가했을 때, motor가 회전하지 않는 것을 확인할 수 있었다. Goal position에 정확히 도달하였기에 외부 힘으로 인해 이를 변형시키려 하려고 해도 target position을 유지하기 위해 motor에서 조절하기에 회전하지 않았다.

- Velocity control mode

Velocity mode로 변환하기 위해 motor_control.cpp 파일에서 ArticulatedSystem::Mode::VELOCITY로 설정하였다. 그 다음 targetQ(target position)을 1rad, targetQdot을 0rad/s로 설정하고 articulated_system.SetGoalVelocity(targetQ)를 실행하였다. 아래는 그 과정이다.

```
78     ArticulatedSystem articulated_system(dof, ArticulatedSystem::Mode::VELOCITY, "/dev/ttyUSB0",
79                                     dynamixel_id_set, position_resolution, motor_torque_constants); //1 DOF manipulator (In other words, one motor system)
```

그림 28. Velocity mode로 변경한 모습

157	////////// Constant target //////////
158	targetQ <= 1; // M_PI;
159	targetQdot <= 0;
160	////////// Constant target END //////////

그림 29. Target position을 1rad, target velocity를 0rad/s로 설정한 모습

171	////////// VELOCITY CONTROL //////////
172	articulated_system.SetGoalVelocity(targetQdot);
173	////////// VELOCITY CONTROL END //////////

그림 30. "targetQdot"으로 target velocity를 설정한 모습

이를 build하고 run한 결과 motor가 회전하지 않고 velocity를 0rad/s로 유지하였다. 이때 motor를 회전시키려 손으로 힘을 가했을 때, motor가 회전하지 않는 것을 확인할 수 있었다. Motor에서 velocity는 Position을 통해 측정되기에 만약 position이 변화하게 된다면 velocity가 생성되기에 target velocity가 0rad/s인 상황에서 이를 유지하기 위해 외부 힘이 주어져도 회전하지 않았다.

- Current control mode

Current mode로 변환하기 위해 motor_control.cpp 파일에서 ArticulatedSystem::Mode::CURRENT로 설정하였다. targetTorque를 0으로 설정하고 articulated_system.SetGoalTorque(targetTorque)를 실행하였다. 아래는 그 과정이다.

```
78     ArticulatedSystem articulated_system[dof, ArticulatedSystem::Mode::CURRENT,
79             "/dev/ttyUSB0",
80             dynamixel_id_set, position_resolution, motor_torque_constants], //1 DOF manipulator (In other words, one motor system)
```

그림 31. Current mode로 변경한 모습

```
175     //////////////////// CURRENT CONTROL ///////////////////
176     targetTorque.setZero();
177     articulated_system.SetGoalTorque(targetTorque);
178     //////////////////// CURRENT CONTROL END ///////////////////
```

그림 32. Target torque를 0으로 지정하고 “targetTorque”로 target torque를 설정한 모습

Motor에 의해 생성된 torque는 current에 proportional하게 target torque를 지정하면 그에 해당하는 current가 발생되게 된다. 이 경우에는 target torque가 0이기에 build하고 run한 결과 motor가 회전하지 않고 그 위치를 유지하였다. 이때 motor를 회전시키려 손으로 힘을 가했을 때, position mode와 velocity mode의 경우와 달리 어렵지 않게 회전하는 것을 확인하였다. 이는 만약 motor가 회전을 저지하기 위해 torque를 생성한다면 0으로 설정한 target torque에서 벗어나기에 특별한 저지가 없는 것이다. 만약 motor의 power switch를 끄고 외부 힘을 통해 회전시켰을 경우 additional resisting torque가 느껴지며 motor LED가 깜빡이는 것을 확인할 수 있었다. Cogging torque라고도 불리는 이것은 motor의 power가 꺼졌을 때 permanent magnet과 motor winding의 magnetic field 사이 interaction으로 인해 생성된다. 이렇게 조금 더 큰 저항을 가지게 된 상태에서 손으로 돌릴 때 생성되는 additional current에 의해 motor led가 깜빡이게 된다. 아래는 관찰한 motor LED의 모습이다.

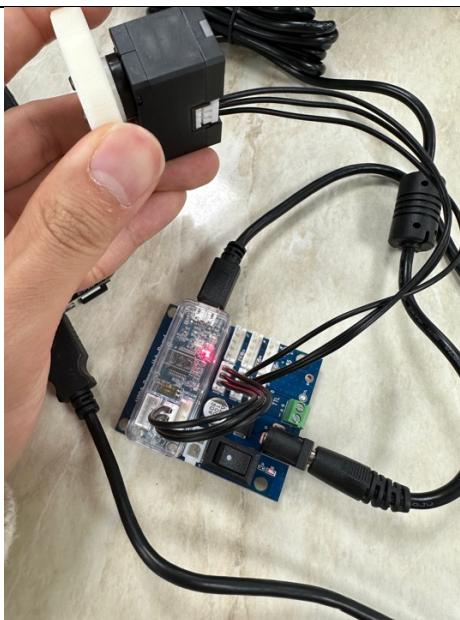


그림 33. Power off 상태에서 motor를 손으로 돌렸을 때 motor LED에 불이 들어오는 모습

세가지 mode를 test 해보았으며 각각의 mode의 특징을 확인하였다. Current mode에서 P controller를 구현해 보았다. Target velocity가 0rad/s로 설정되어 있다.

```
180     ////////////////// Velocity P control USING Current Mode //////////////////
181     for (size_t i = 0; i < dof; i++)
182     {
183         Kp(i, i) = 0.01;
184     }
185     targetTorque = -Kp * (currentQdot - targetQdot) ;
186     articulated_system.SetGoalTorque(targetTorque);
187     ////////////////// Velocity P control USING Current Mode END //////////////////
```

그림 34. Current mode를 이용해 P controller를 구현한 모습

이를 build하여 run 하였을 때, velocity가 target velocity에서 벗어나면 저항이 커지는 것을 확인할 수 있었다. 0.01로 설정되어 있는 P gain 값을 키우게 되면 그 저항이 더욱 커졌으며 저항은 error에 proportional하다. 앞의 Velocity mode에서는 PI controller인데 이와 비교하면 target velocity에서 벗어날 때 P controller의 경우 error가 일정하게 유지된다면 그 저항도 일정하게 유지된다. 그러나 PI controller의 경우 I gain도 존재하여 error가 시간에 따라 축적되어 저항이 축적된 error에 proportional하게 결정되기에 error가 일정하게 유지되더라도 저항은 시간이 흐를수록 더욱 커지게 된다.

Problem 3D. PD Position Controller (reference position: step input)

Current mode에서 PD controller를 구현하였다.

```
78     ArticulatedSystem articulated_system[dof, ArticulatedSystem::Mode::CURRENT, "/dev/ttyUSB0",
79                                     dynamixel_id_set, position_resolution, motor_torque_constants]; //1 DOF manipulator (In other words, one motor system)
  그림 35. Current mode로 변경한 모습
```

```
189     //////////////////// CURRENT CONTROL ///////////////////
190     //////////////////// PID Position controller //////////////////
191     for (size_t i = 0; i < dof; i++)
192     {
193         Kp(i,i) = 0.5;
194         Ki(i,i) = 0.1;
195         Kd(i,i) = 0.05;
196     }
197     Eigen::VectorXd error = currentQ - targetQ;
198     Eigen::VectorXd error_dot = currentQdot - targetQdot;
199     integral_error += error * dt;
200     targetTorque = -Kp*error -Kd*error_dot -Ki*integral_error; /* Implement here. you have to implement the PID Controller */
201     articulated_system.SetGoalTorque(targetTorque);
202     //////////////////// PID Position controller END //////////////////
203     //////////////////// CURRENT CONTROL END //////////////////
```

그림 36. PID controller 구현 코드 (Ki 값을 0으로 하여 PD controller 구현)

Target joint angle을 1rad으로 설정하고 실험을 진행하였다. P gain은 proportional gain을 의미하여 error에 proportional 한 값을 가진다. 이는 스프링으로 이해할 수 있는데 error가 커질수록 그에 비례하여 control system의 response 역시 커지게 된다. D gain은 derivative gain을 의미하며 error의 미분 값, 즉 error 변화 경향성에 proportional 한 값을 가진다. 이는 damper로 이해할 수 있는데, error가 빠르게 변하게 되면 그에 비례하여 control system의 response 역시 커지게 된다. 이를 확인하기 위해 P gain 값과 D gain 값을 바꾸어 보면 그 경향성을 확인해보았다. Disturbance를 위해 Motor를 손으로 지긋이 눌러주었다.



그림 37. Motor를 손으로 지긋이 눌러 disturbance를 준 모습

우선 K_p 값을 0.1부터 0.7까지 0.1씩 증가시켰고, $K_i=0$, $K_d=0.01$ 로 고정하고 그라프를 확인하였다. 결과는 아래와 같다.

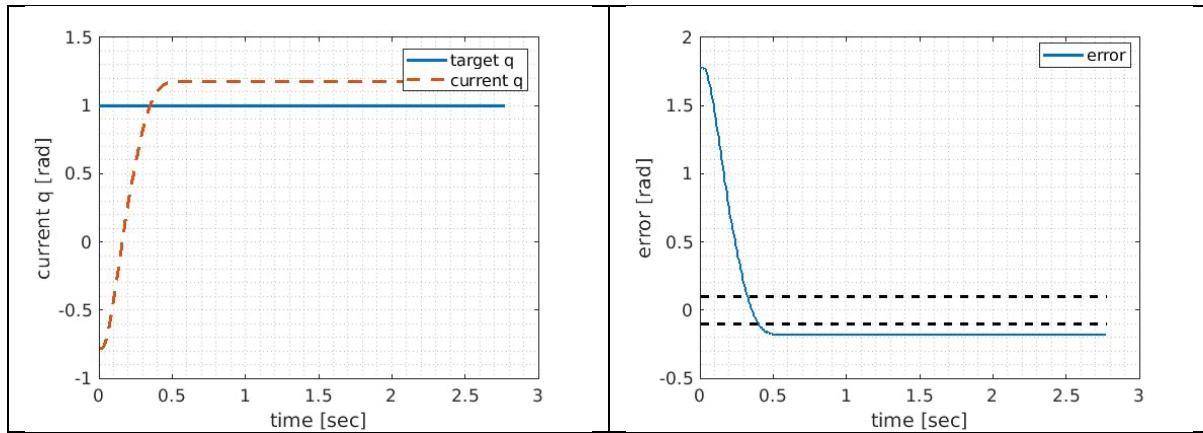


그림 38. $K_p=0.1$ 일 때 current position과 target position, error를 plot한 그래프

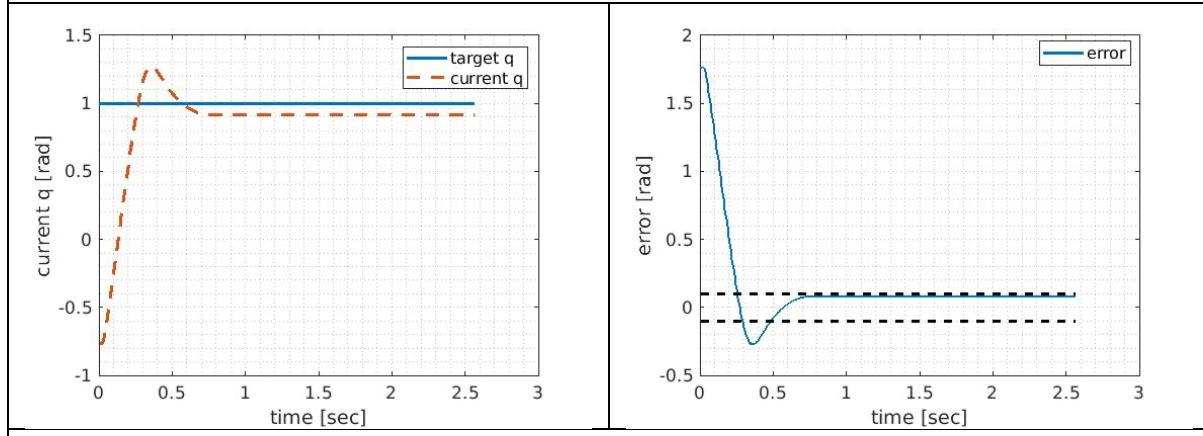


그림 39. $K_p=0.2$ 일 때 current position과 target position, error를 plot한 그래프

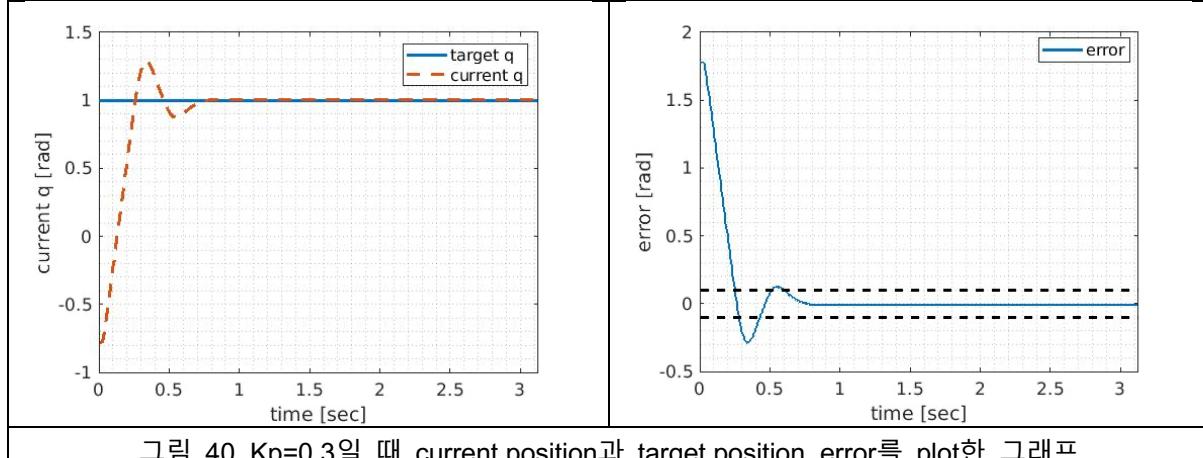
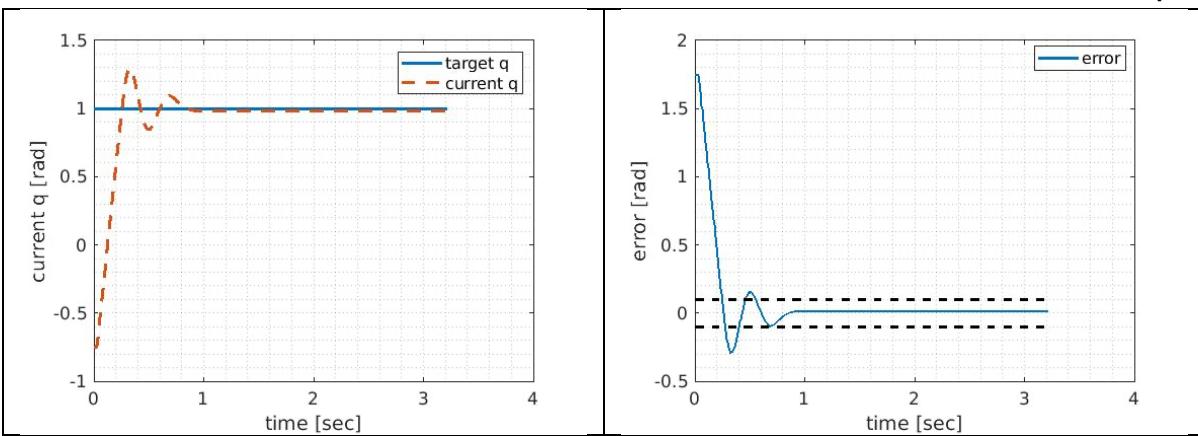
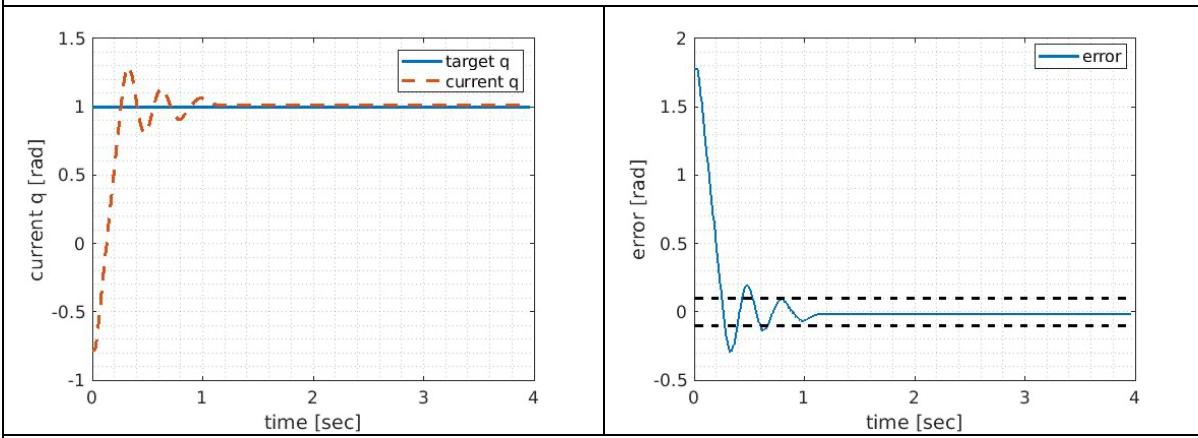
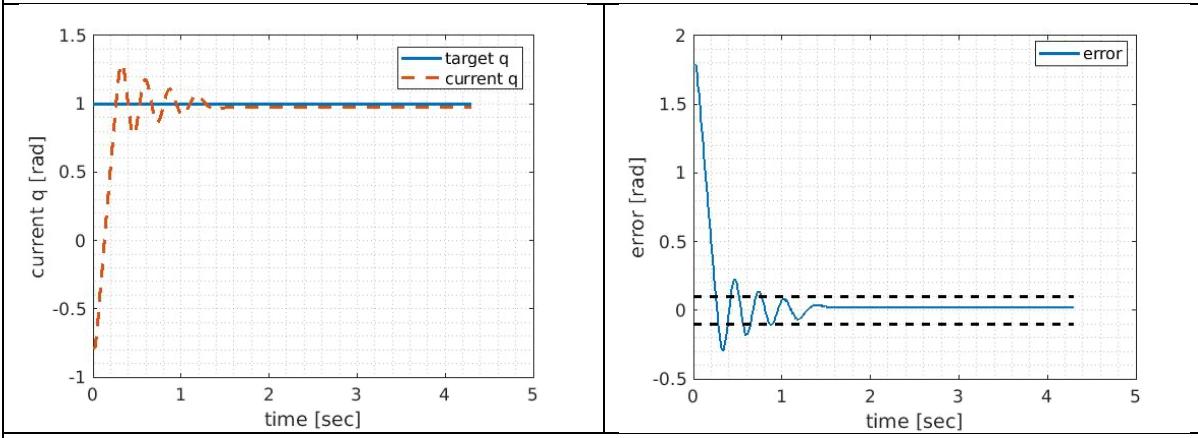


그림 40. $K_p=0.3$ 일 때 current position과 target position, error를 plot한 그래프

그림 41. $K_p=0.4$ 일 때 current position과 target position, error를 plot한 그래프그림 42. $K_p=0.5$ 일 때 current position과 target position, error를 plot한 그래프그림 43. $K_p=0.6$ 일 때 current position과 target position, error를 plot한 그래프

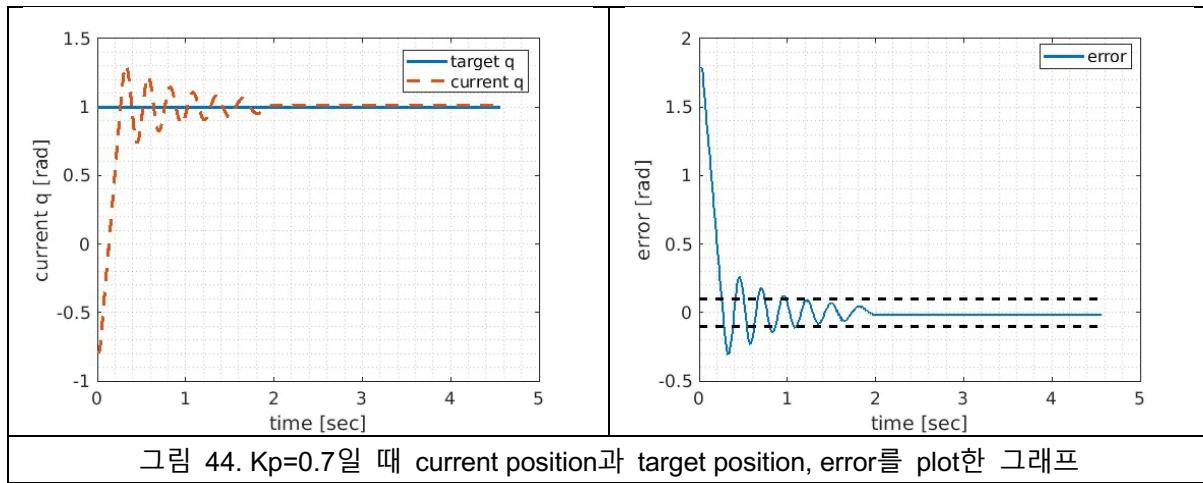


그림 44. $K_p=0.7$ 일 때 current position과 target position, error를 plot한 그래프

위 그래프에서 확인할 수 있듯이, K_p 값, 즉 P gain이 커질수록 error에 더욱 크게 proportional한 크기로 control system이 작동하였다. 탄성계수가 더욱 큰 스프링이 사용되었다 생각하면 이해하기 쉽다. Response frequency도 더욱 커진 것을 확인할 수 있었다. 다음으로 K_d 값을 0.01부터 0.05까지 0.01씩 증가시켰고, $K_p=0.3$, $K_i=0$ 으로 고정하고 그래프를 확인하였다. 결과는 아래와 같다.

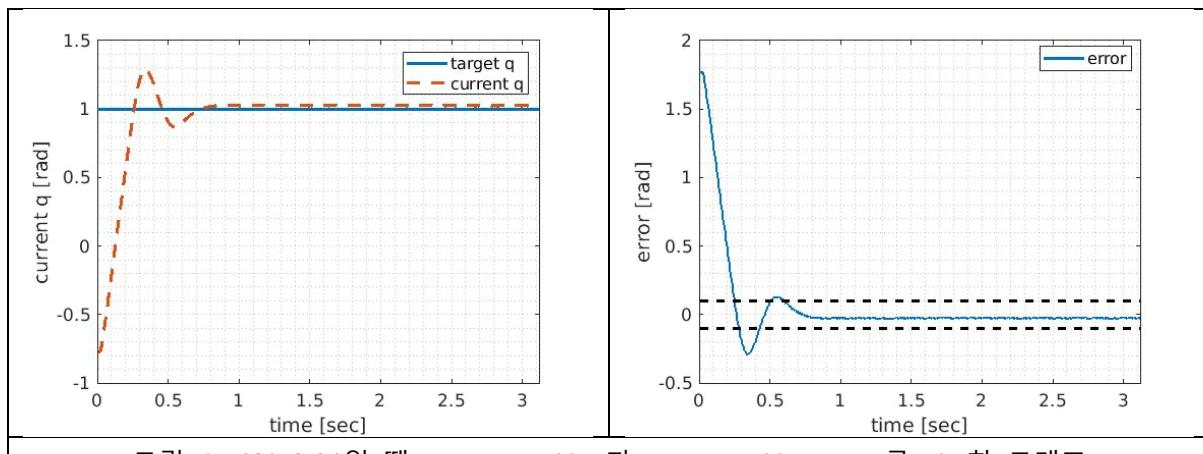


그림 45. $K_d=0.01$ 일 때 current position과 target position, error를 plot한 그래프

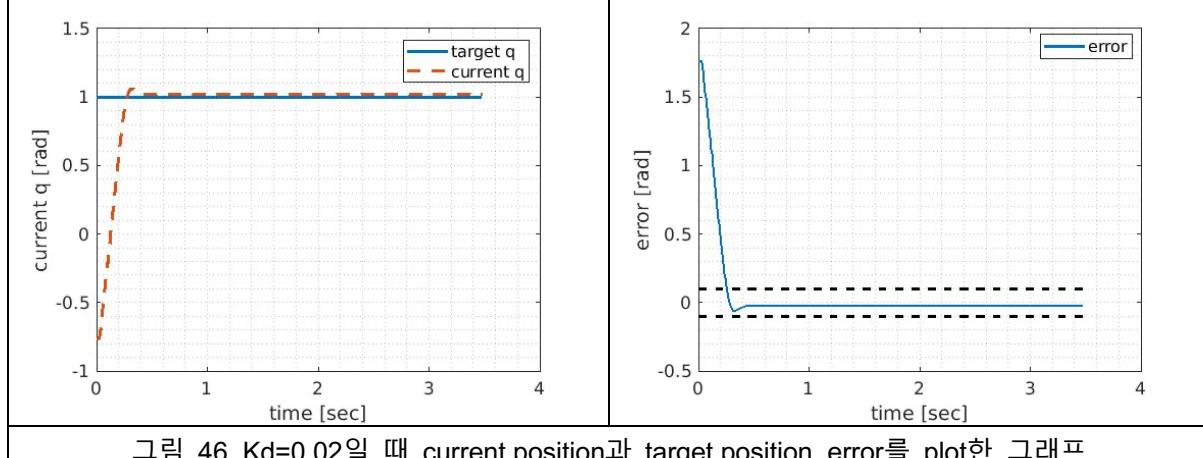
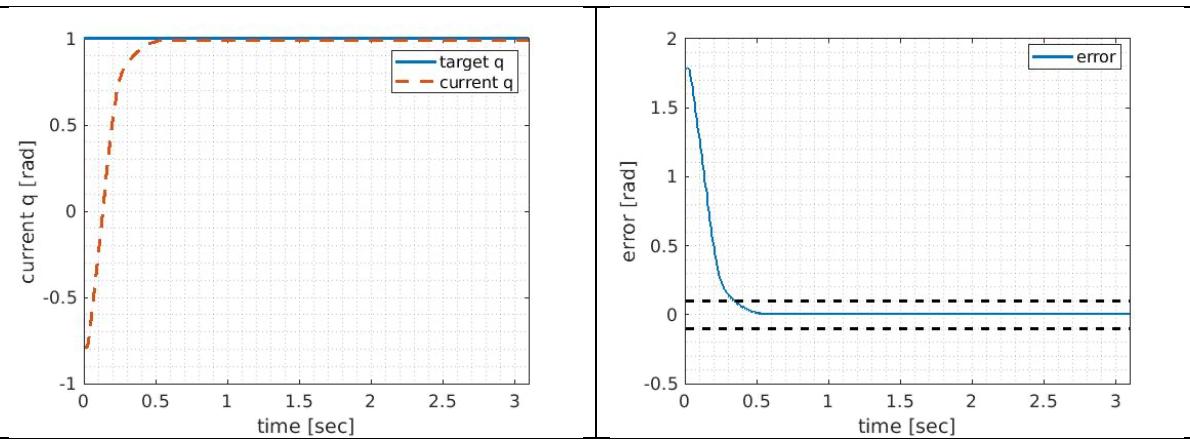
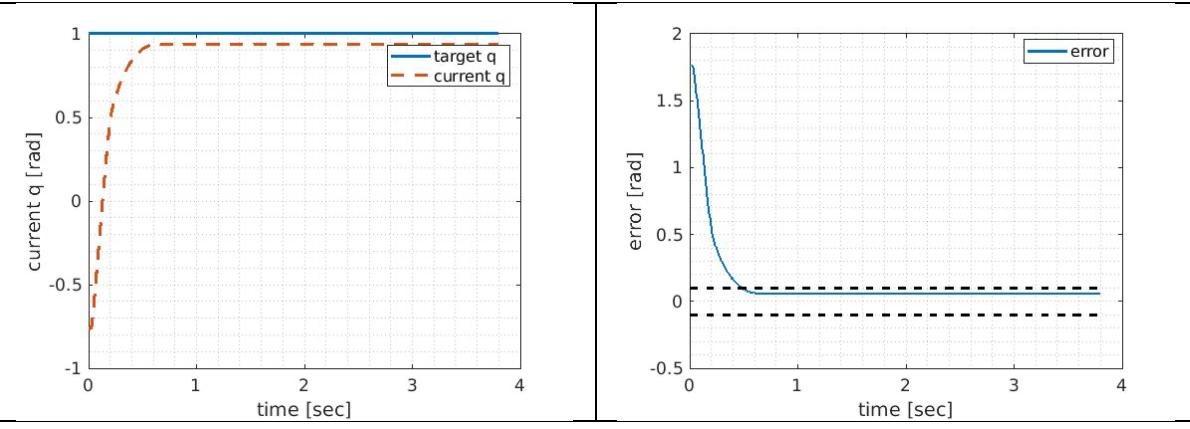
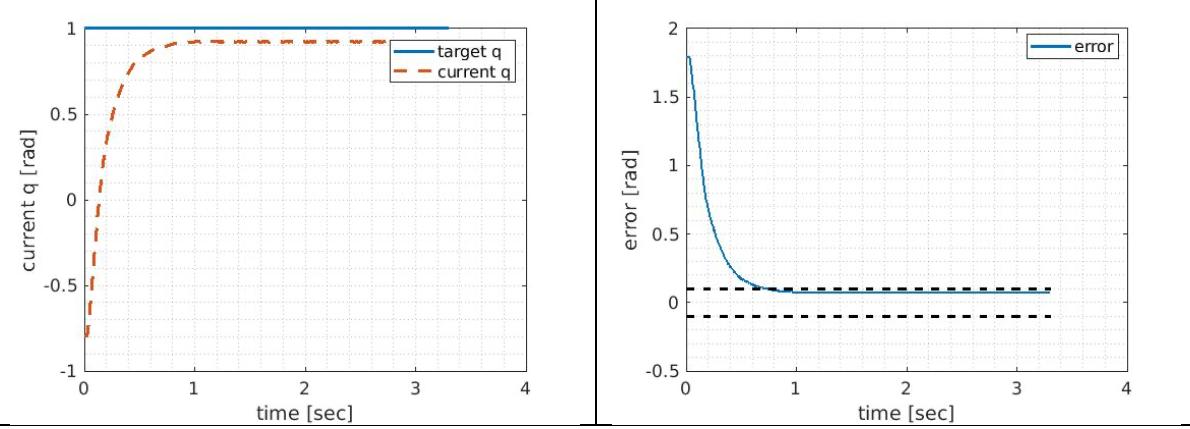


그림 46. $K_d=0.02$ 일 때 current position과 target position, error를 plot한 그래프

그림 47. $K_d=0.03$ 일 때 current position과 target position, error를 plot한 그래프그림 48. $K_d=0.04$ 일 때 current position과 target position, error를 plot한 그래프그림 49. $K_d=0.05$ 일 때 current position과 target position, error를 plot한 그래프

위 그림에서 확인할 수 있듯이, K_d 값, 즉 D gain이 커질수록 error의 미분값에 더욱 크게 proportional한 크기로 control system이 작동하였다. Damping 상수가 큰 damper가 사용되었다 생각하면 이해하기 쉽다. 또한 overshoot를 줄이는 모습을 보여주었으며 D gain을 조절하여 system이 error에 얼마나 빠르게 반응하는지 response time을 조절할 수 있다. D gain이 과도하게 커지게 되면 overdamping이 이루어져 target position에 도달하지 못하는 모습을 볼 수 있다. PD controller의 경우 steady-state error가 여전히 남아있는 것을 확인할 수 있었다. 이는 final value theorem을 통해서도 확인 가능하다. 이를 해결하기 위해 I gain을 추가하였다.

Problem 3E. PID Position Controller (reference position: step input)

Current mode에서 PID controller를 구현하였다. K_i 값을 0.01, 0.05, 0.1을 주어 각각 확인해 보았으며, $K_p=0.3$, $K_d=0.05$ 으로 고정하고 그래프를 확인하였다. 결과는 아래와 같다.

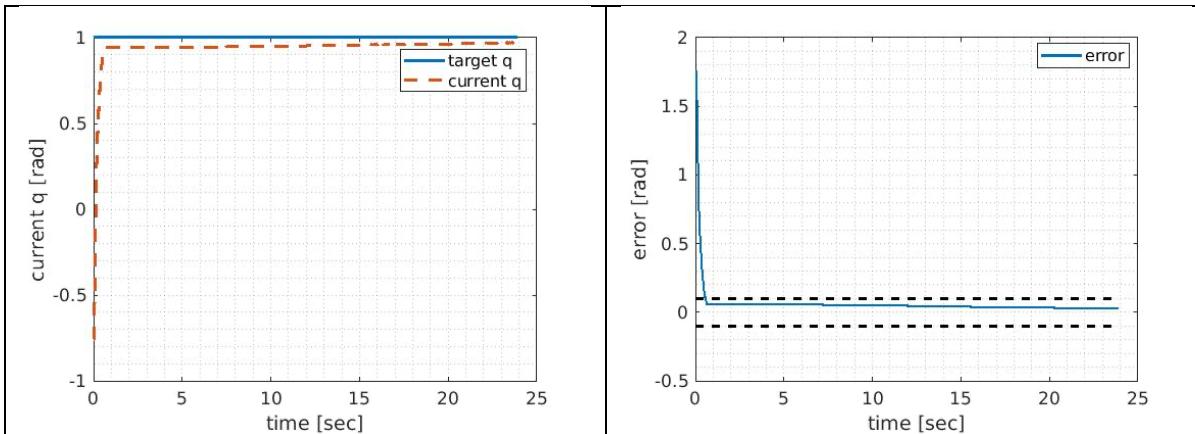


그림 50. $K_i=0.01$ 일 때 current position과 target position, error를 plot한 그래프

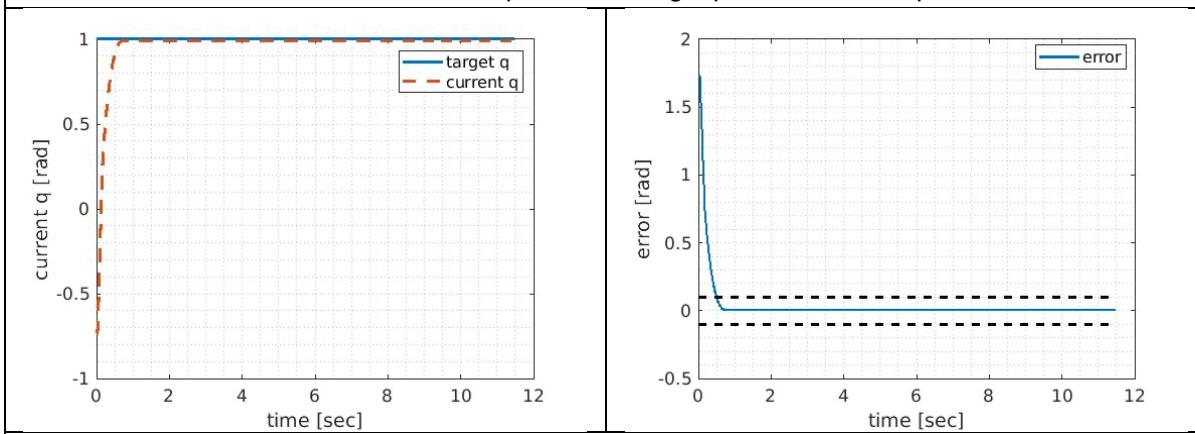


그림 51. $K_i=0.05$ 일 때 current position과 target position, error를 plot한 그래프

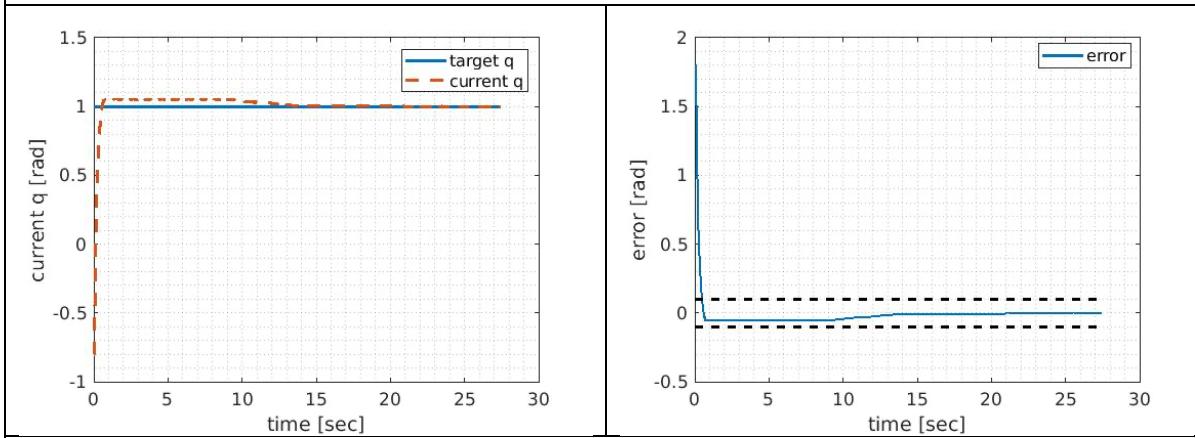


그림 52. $K_i=0.1$ 일 때 current position과 target position, error를 plot한 그래프

위 그래프에서 확인할 수 있듯이, K_i 값, 즉 I gain이 커질수록 error의 적분값에 더욱 크게 proportional한 크기로 control system이 작동하였다. I gain을 사용할 경우 steady-state error가 시간이 흐를수록 0으로 가는 모습을 보여주었으며 이는 I gain 값이 클수록 더욱 빠르게 0으로 도달하는 모습을 보였다. 이는 final value theorem을 통해서도 확인 가능하다. I gain이 너무 클 경우 overshoot 또는 undershoot이 나타날 수 있으며 response time이 P gain이나 D gain에 비해 길다.

Problem 3F. PID Position Controller (reference position: sinusoidal input)

구현한 PID controller를 이용하여 이번에는 Sinusoidal input에 대한 response를 확인하였다. Sine wave의 amplitude를 1rad으로 설정하고 frequency를 0.77Hz로 설정하였다. 해당 input에 대해 PID controller gain을 tuning하여 5초 이후에 error가 0.1 rad과 -0.1 rad 사이에 위치하도록 설정하였다. $K_p=0.5$, $K_i=0.1$, $K_d=0.05$ 일 때 이를 만족하였다. 결과는 아래와 같다.

```

148     double amp = 1;
149     //////////// fixed frequency /////////////
150     double sin_wave_frequency = 2*M_PI*0.77; // [\omega, rad / s]
151     //////////// fixed frequency END ///////////
152
153     //////////// For Chirp Signal ///////////
154     // double sin_wave_frequency = 2*M_PI*0.1*total_elapsed_time_sec; // [rad / s] for chirp signal.
155     //////////// For Chirp Signal ///////////
156
157     //////////// Constant target ///////////
158     // targetQ << 1;// M_PI;
159     // targetQdot << 0;
160     //////////// Constant target END ///////////
161
162     //////////// Sine wave target ///////////
163     targetQ << amp*sin(sin_wave_frequency*total_elapsed_time_sec);
164     |targetQdot << sin_wave_frequency*amp*cos(sin_wave_frequency*total_elapsed_time_sec);

```

그림 53. Target position을 sinusoidal input으로 설정한 모습

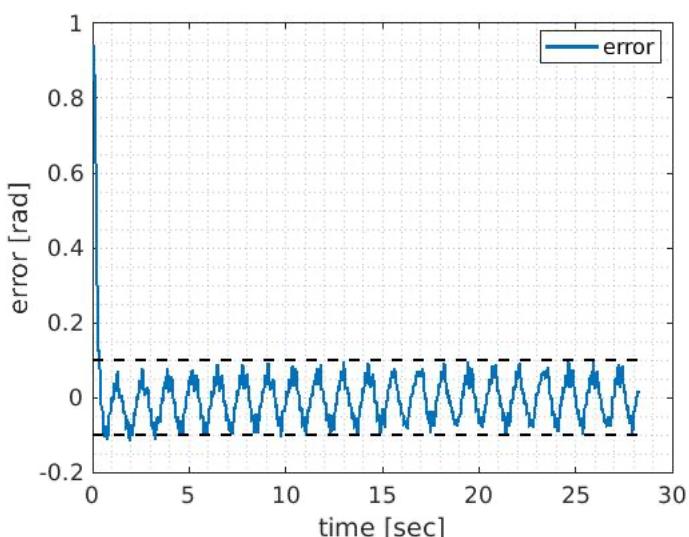


그림 53. Sinusoidal input에 대한 error 결과

Problem 3G. Understanding Frequency Response (with Chirp Signal)

위에서 설정한 gain값으로 이번에는 chirp signal input에 대한 response를 확인해보았다. Chirp signal은 시간에 따라 frequency가 계속해서 증가하는 signal로 frequency 증가에 따른 PID controller의 frequency response를 이해해 볼 수 있다. Chirp signal 구현과 실행 결과 그레프는 아래와 같다.

```

148
149 /////////////// fixed frequency /////////////
150 double sin_wave_frequency = 2*M_PI*0.77; // [\omega, rad / s]
151 ////////////////// fixed frequency END /////////////
152
153 ////////////////// For Chirp Signal /////////////
154 double sin_wave_frequency = 2*M_PI*0.1*total_elapsed_time_sec; // [rad / s] for chirp signal.
155 ////////////////// For Chirp Signal /////////////
156
157 ////////////////// Constant target /////////////
158 // targetQ << 1;// M_PI;
159 // targetQdot << 0;
160 ////////////////// Constant target END /////////////
161
162 ////////////////// Sine wave target /////////////
163 targetQ << amp*sin(sin_wave_frequency*total_elapsed_time_sec);
164 targetQdot << 2 * sin_wave_frequency*amp*cos(sin_wave_frequency*total_elapsed_time_sec);
165 ////////////////// Sine wave target END /////////////

```

그림 54. Target position을 chirp signal로 설정한 모습

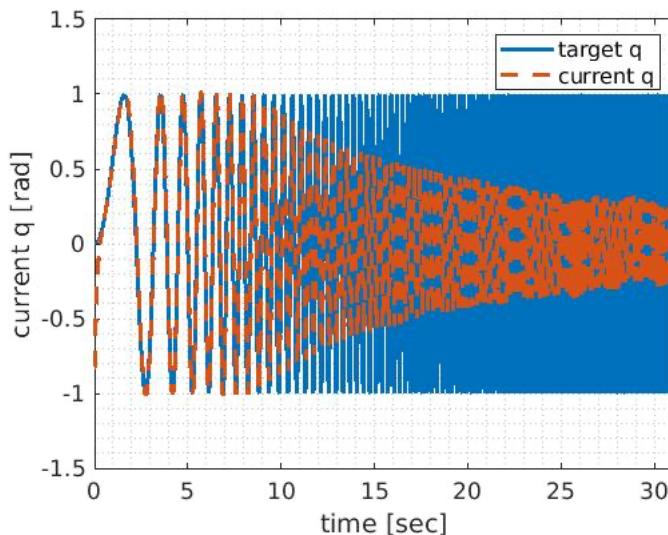


그림 55. Target position과 current position을 plot 한 모습

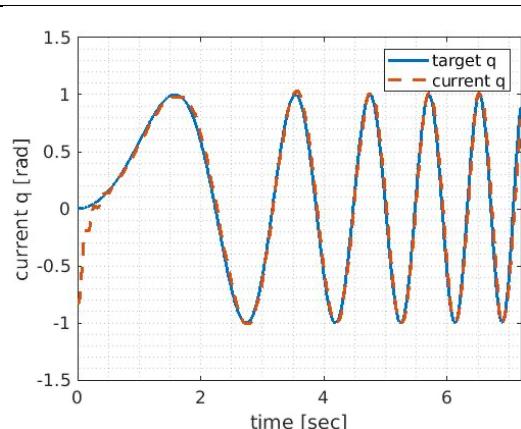


그림 56. Low frequency 구역 모습

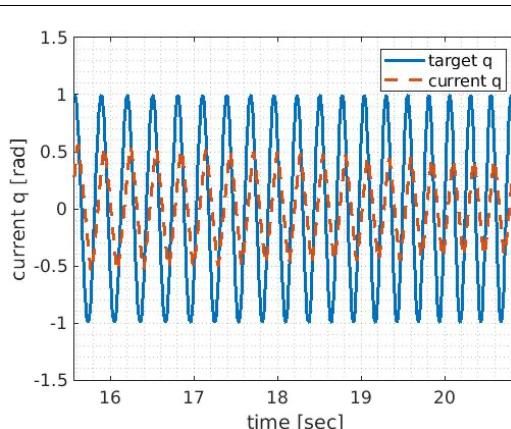


그림 57. High frequency 구역 모습

해당 결과를 확인하면 chirp signal에서 frequency가 낮은 동작 초기 구간에서는 control system이 input에 대해 올바르게 response하여 position을 잘 따라가는 것을 볼 수 있다. 그러나 frequency가 높은 동작 후반 구간에서는 control system이 input을 따라가지 못하고 phase delay가 일어나며 제대로 response하지 못하는 것을 확인할 수 있다. 이를 통해 control system에서 cutoff frequency 보다 frequency가 큰 input signal의 경우 system이 제대로 response를 하지 못하는 것을 보였다.

4. Discussion

(1) Topic: Measurement of motor angular velocity using numerical differentiation with low pass filter

a. Plot the three following motor angular velocity results

- Result of obtaining joint angular velocity through numerical differentiation
- the result of applying the low pass filter
- motor angular velocity obtained by using the GetJointVelocity() function in Articulated System Class.

<그림 19>에서 위에 3가지 그래프를 plot하였다. 이를 통해 numerical differentiation에서 나타나는 quantization error amplification을 확인할 수 있었고 해당 noise를 low pass filter를 통해 효과적으로 제거할 수 있는 것을 확인하였다.

b. Plot the tendency of the results as the cutoff frequency changes. (For example, plot the filtering noise result when f = 1Hz, f = 10 Hz, f = 50 Hz, f = 100 Hz, f = 1000 Hz, etc)

<그림 20>부터 <그림 24>에서 cutoff frequency가 1rad/s, 10rad/s, 50rad/s, 100rad/s, 1000rad/s일 때 motor motion에 대한 angular velocity를 low pass filter를 이용한 numerical differentiation 방법으로 계산하여 plot하였다. 이를 통해 cutoff frequency가 커질수록 smoothing이 줄어들고 높은 frequency의 범위에서도 filtering이 이루어지지 않는 것을 보았다. 따라서 원래 signal은 보존하고 noise만 filtering하기 위해서는 적절한 cutoff frequency 설정이 필요했다.

c. Discuss why we required the low pass filter when we use the numerical differentiation and Discuss the characteristics of the cutoff frequency, including following elements. (Related lab procedure: 26, 27)

- The reason for amplified noise when performing numerical differentiation
- Why we use the low pass filter.
- The meaning of the cutoff frequency
- the results of the plot the tendency of cutoff frequency
- trade-off problem of cutoff frequency

Numerical differentiation은 아래의 식에 의해 계산된다.

$$\dot{x} = \frac{d}{dt}x \approx \frac{x[k] - x[k-1]}{\Delta T}$$

이 때, ΔT 가 굉장히 작기 때문에 Quantization error가 더욱 증폭되게 된다. 이러한 증폭된 error로 인한 noise를 줄이고 high frequency fluctuation을 감소시키기 위해 low pass filter를 사용하게 된다. 여기서 low pass filter의 cutoff frequency는 input signal에 대해 gain이 -3dB이 되는 frequency를 의미하며 이 frequency 이상의 frequency 구역에 대해서 filtering이 이루어진다. 위에서 설명했듯이 cutoff frequency가 커질수록 더 빠른 response가 가능하나 noise 역시 통과하게 되고 그렇다고 너무 작을 경우 input signal이 과도하게 filtering되는 문제가 발생한다. 따라서 적절한 cutoff frequency를 설정하는 것이 매우 중요하다.

(2) Topic: Characteristics of each PID gain of the position controller

a. Discuss the characteristics of P gain and D gain of the position PD controller including the following elements.

- The description about physical meaning of the P and D gain of the position controller from

control theory

- Explanation of the effect of P gain on the steady-state error using the Final Value Theorem
- Explanation of the effect of D gain on the system transient response
- The tendency of actual experiment results when we P gain increase
- the tendency of the experiment results for the system transient response as the D gain increases
- (Optional) Characteristics of each P and D gain which you learned during actual tuning

P gain은 proportional gain으로 error에 proportional하다. D gain은 derivative gain으로 error의 rate of change에 proportional하다. PD controller는 아래와 같이 표현할 수 있다.

$$f = -D\dot{x} - K(x - x_{des})$$

여기서 $-K(x - x_{des})$ 가 P gain을 의미하고 $-D\dot{x}$ 가 D gain을 의미한다. P-control term은 spring으로, D-control term은 damping으로 해석할 수 있다. PD controller를 통해 constant disturbance가 있는 system을 control한다고 가정하면 아래와 같이 표현할 수 있다.

$$\begin{aligned} M\ddot{x} &= f + d \\ f &= -D\dot{x} - K(x - x_{des}) \end{aligned}$$

이를 정리하면,

$$M\ddot{x} + D\dot{x} + K(x - x_{des}) = d$$

여기서 Laplace transform을 사용하면,

$$\begin{aligned} X(s) &= \frac{K}{Ms^2 + Ds + K} X_{des}(s) + \frac{1}{Ms^2 + Ds + K} D(s) \\ X_{des}(s) &= \frac{x_{des}}{s}, D(s) = \frac{d}{s} \end{aligned}$$

따라서 final value theorem에 의해,

$$\lim_{s \rightarrow 0} sX(s) = x_{des} + \frac{1}{K}d$$

여기서 steady-state error가 남아있는 것을 확인할 수 있다. P gain이 커질수록 줄어들기는 하지만 아예 0으로 만들 수는 없다. 이러한 P gain은 커질수록 response time이 빨라지지만 너무 클 경우 overshoot과 oscillation이 생길 수 있다. D gain은 P gain만 있었을 때 없었던 damping 효과와 overshoot 상쇄, response time 조절 등의 효과를 지닌다. 그러나 D gain 역시 너무 커질 경우 overdamping 등이 일어날 수 있다. P gain과 D gain이 늘어날 때 경향성은 <그림 38>부터 <그림 49>에서 확인할 수 있다. 이론적 설명과 실제 실험 결과가 일치하는 것을 확인할 수 있었다.

b. Discuss about the characteristics of i gain of the position PID Controller including

- Explanation of the effect of I gain on the steady-state error using the Final Value Theorem
- Experimental proof that The I gain makes the steady-state error goes to zero

I gain은 integral gain으로 시간에 대한 error integral 값에 proportional하다. 이는 누적된 error를 의미하며 steady state error를 0으로 만들어준다. PID controller를 통해 constant disturbance가 있는 system을 control한다고 가정하면 아래와 같이 표현할 수 있다.

$$\begin{aligned} M\ddot{x} &= f + d \\ f &= -D\dot{x} - K(x - x_{des}) - I \int (x - x_{des}) \end{aligned}$$

이를 정리하면,

$$M\ddot{x} + D\dot{x} + K(x - x_{des}) + I \int (x - x_{des}) = d$$

여기서 Laplace transform을 사용하면,

$$X(s) = \frac{Ks + I}{Ms^3 + Ds^2 + Ks + I} X_{des}(s) + \frac{s}{Ms^3 + Ds^2 + Ks + I} D(s)$$

$$X_{des}(s) = \frac{x_{des}}{s}, D(s) = \frac{d}{s}$$

따라서 final value theorem에 의해,

$$\lim_{s \rightarrow 0} sX(s) = x_{des}$$

여기서 steady-state error가 사라지는 것을 확인할 수 있다. I gain이 커질수록 더욱 빠르게 error가 0에 도달한다. 시간을 흐르면서 I gain으로 인해 steady-state error가 0으로 가는 경향성은 <그림 50>부터 <그림 52>에서 확인할 수 있다.

(3) Topic: Frequency response

- a. Using the meaning of example from the bode plot result on page 11 of the lecture_control_part2.pdf, Discuss the experimental result including how the amplitude and phase delay phenomena change as the frequency increases.

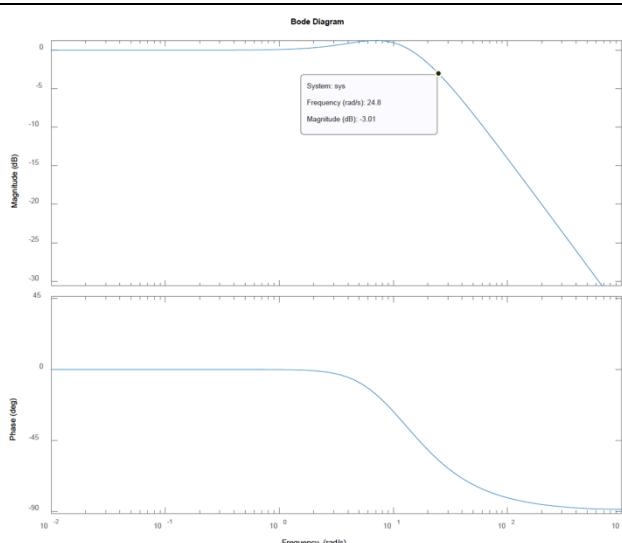


그림 58. Page 11 of lecture_control_part2.pdf

<그림 58>이 의미하는 바는 frequency가 높아질수록 이에 대한 control system의 response는 amplitude의 경우 감소하고 phase의 경우 phase delay가 생긴다는 것이다. 즉, high frequency, cutoff frequency보다 높은 frequency의 input signal에 대해서는 system이 제대로 respond하지 못한다는 것을 의미한다. <그림 55>부터 <그림 57>에서 이를 실험적으로 확인하였으며 low frequency 영역에서는 정상적이던 response가 high frequency 영역에서 amplitude가 계속해서 줄어들고 phase delay가 생기는 등 제대로 된 response가 이루어지지 않는 것을 관측했다. 이를 통해 frequency response를 정확히 이해할 수 있었다.

5. References

- [1] Lab3 Experiment Guide.pdf
- [2] Lab3 Procedure.pdf
- [3] lecture_control_part1.pdf
- [4] lecture_control_part2.pdf
- [5] Eigen Library, https://eigen.tuxfamily.org/index.php?title=Main_Page
- [6] Dynamixel XC330-T288-T <https://emanual.robotis.com/docs/en/dxl/x/xc330-t288/>
- [7] Dynamixel SDK Github <https://github.com/ROBOTIS-GIT/DynamixelSDK>
- [8] The PID Controller & Theory Explained, <https://www.ni.com/ko-kr/shop/labview/pid-theory-explained.html>
- [9] Wikipedia, PID controller, https://en.wikipedia.org/wiki/PID_controller