

Lab 5. Robot Manipulator

1. Purpose

Lab 5의 목적은 joint space와 task space에서 4 DOF(Degree of Freedom)을 가진 robot manipulator를 control하는데 있다. 먼저 direct teaching을 통해 robot이 user-defined waypoints를 찾아가도록 한다. 다음으로는 user의 keyboard input을 이용하여 end-effector의 position을 teleoperation control을 통해 제어한다. 이는 world linear velocity를 Jacobian을 통해 표현하고 inverse kinematics를 통해 얻어낸다. 두 task 모두 target joint angle이 결정되고 velocity mode를 이용하여 joint space P controller가 작동한다.

2. Experiment Sequence

본 lab은 2가지 problem으로 구성되어 있다. 첫번째는 direct teaching으로 manipulator가 세개의 waypoints를 반복적으로 따라가는 것이다. 두번째는 teleoperation으로 end-effector position을 keyboard input에 따라 control하는 것이다. 아래는 구현하게 될 2가지 problem이다.

(1) Problem 5A. Direct Teaching

세개의 waypoint가 지정되어 이를 manipulator가 반복적으로 따라가는 것이 task이다. Waypoint는 joint angle로 define되어 dimension이 DOF와 동일한 4이다. 첫번째 waypoint는 program이 시작할 때 initial joint angle이다. 두번째 세번째 waypoint는 user가 define하게 된다. 본 problem을 해결하기 위해 가장 먼저 user-defined waypoints를 'waypoints.csv' 파일에 저장하는 코드를 구현한다. 그 다음, first-order polynomial을 이용하여 trajectory generation을 수행한다. 각 motor의 joint angle을 계산하고 velocity mode를 이용한 joint space P position controller를 구현한다.

- Development setup
- Save two user-defined waypoints in joint space
- Generate trajectory
- Joint space P position control using velocity mode

(2) Problem 5B.

Keyboard input을 통한 teleoperation을 이용하여 end-effector position을 control하는 task이다. 먼저 task space command를 keyboard input으로 받는다. 그 다음 world linear velocity에 해당하는 Jacobian matrix를 계산한다. 이후 inverse kinematic을 이용하여 desired joint angle을 얻어내고 velocity mode를 사용하여 joint space P controller를 구현한다.

- Development setup
- Get task space command from the keyboard input
- Compute the Jacobian matrix
- Compute the desired joint angles using inverse kinematics
- Joint space P position control using velocity mode

3. Experiment Results

Development setup

가장 먼저 개발 환경을 구축하였다. Development PC에 제공된 skeleton code를 다운받았다. Beaglebone에 Debian Linux가 설치된 SD카드를 넣고 활성화 시켰으며 NFS를 통해 development PC 파일들에 접근하도록 하였다. 이 과정들은 Lab1에서 연습한대로 수행하였다. 그 다음 Dynamixel SDK Shared Library를 Beaglebone에 설치하였는데 이는 Lab3에서 이미 설치하여 제대로 설치되었는지 확인하고 다음 단계로 넘어갔다. 위 설정들을 마친 후 Beaglebone을 dynamixel U2D2에 연결하고 U2D2에 power를 공급하였다. Latency timer를 16ms에서 1ms로 변경하였다.

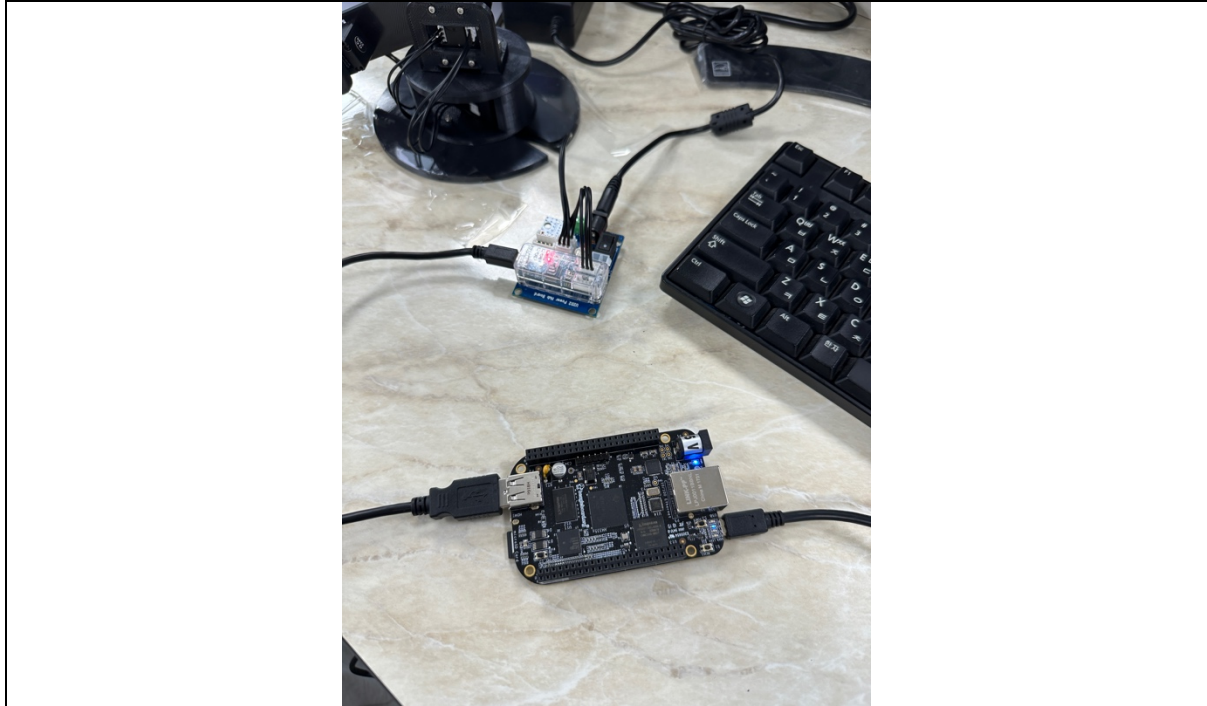


그림 1. Beaglebone을 dynamixel U2D2에 연결한 모습

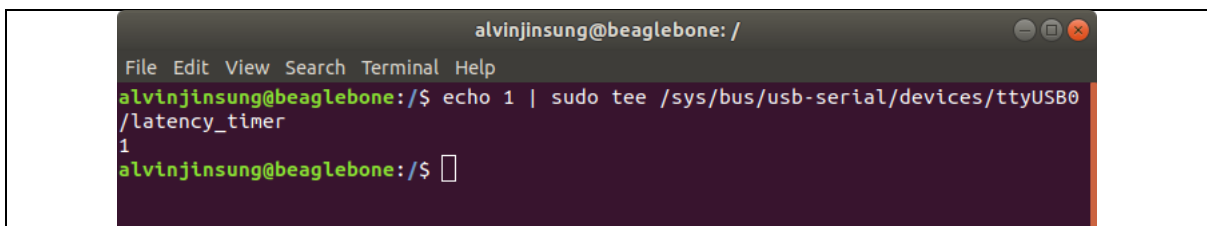


그림 2. Latency timer를 16ms에서 1ms로 변경한 모습

Problem 5A. Direct Teaching

Joint space의 two-user defined waypoints를 저장하기 위해 save_waypoints.cpp 코드를 구현하였다. 우리가 조작할 manipulator의 경우 DOF가 4이기 때문에 코드상에서 int dof 값을 4로 지정하였으며 'ArticulatedSystem.hpp'에 정의되어 있는 GetJointAngle() 함수를 활용하였다. getch() 함수를 활용하여 keyboard input을 입력 받았는데, '1'을 눌렀을 때 첫번째 user-defined waypoint를 저장하도록 GetJointAngle()이 작동하고, manipulator configuration을 변형한 다음 '2'를 눌렀을 때 두

번째 user-defined waypoint를 저장하도록 다시 GetJointAngle()이 작동하였다. 마지막으로 'e'를 눌렀을 때 프로그램이 종료하도록 하였다. 아래는 save_waypoints.cpp 코드이다.

```
/**
 * @file manipulator_control.cpp
 * @author Jiwan Han (jw.han@kaist.ac.kr), Jinyeong Jeong (jinyeong.jeong@kaist.ac.kr)
 * @brief
 * @version 0.1
 * @date 2023-03-27
 *
 * @copyright Copyright (c) 2023
 *
 */

#include <iostream>
#include <fstream>
#include <memory>
#include <string>
#include <unistd.h>
#include <chrono>
#include <Eigen/Dense>
#include <csignal>

#include "Controller/ArticulatedSystem.hpp"
#include "FileIO/MatrixFileIO.hpp"
#include "KeyInput/getche.h"

using namespace Eigen;
using namespace std;

////////// IMPORTANT BELOW COMMAND //////////
// cat /sys/bus/usb-seral/devices/ttyUSB0/latency_timer
// echo 1 | sudo tee /sys/bus/usb-seral/devices/ttyUSB0/latency_timer
// https://emanual.robotis.com/docs/en/dxl/x/xc330-t288/#
////////// IMPORTANT BELOW COMMAND //////////

int main(int argc, char *argv[])
{
    // The manipulator has 4 DOF
    int dof = 4;

    // Dynamixel setup, ID, position resolution, motor torque constant, etc.
    std::vector<uint8_t> dynamixel_id_set;
    std::vector<int32_t> position_resolution;
    std::vector<double> motor_torque_constants;
    for (size_t i = 0; i < dof; i++)
    {
        dynamixel_id_set.emplace_back(i);
        position_resolution.emplace_back(POSITION_RESOLUTION);
        motor_torque_constants.emplace_back(MOTOR_TORQUE_CONSTANT);
    }

    // Initialize the Articulated_system,
    // We need a dof of the system, Operation Mode(Position, Velocity, Current),
    // USB Port information (for serical communication),
    // and dynamixel setup
    ArticulatedSystem articulated_system(dof, ArticulatedSystem::Mode::VELOCITY,
"/dev/ttyUSB0",
dynamixel_id_set,
position_resolution,
motor_torque_constants);

    Eigen::VectorXd waypoint1(dof); waypoint1.setZero(); // the first user-defined waypoint
for direct teaching
    Eigen::VectorXd waypoint2(dof); waypoint2.setZero(); // the second user-defined waypoint
for direct teaching
    char key; // the keyboard input

    std::vector<Eigen::VectorXd> stack_UserDefinedWaypoints; // We will save two user-defined
waypoints in csv files

    articulated_system.DisableTorque();
    while(1){
```


생성된 executable을 실행하여 waypoints를 waypoints.csv파일에 저장하였다. 프로그램을 실행하고 manipulator의 configuration을 변형시킨 다음 '1'을 누르면 첫번째 waypoint가 저장되고 다시 configuration을 변형시킨 다음 '2'를 누르면 두번째 waypoint가 저장된다. 'e'를 눌러 프로그램이 종료되고 waypoints.csv파일이 정상적으로 생성된 것을 확인하였다. 처음에는 생성되지 않아 아래 command를 입력하였다.

```
$ cd ~/EE405_ManipulatorControl
$ sudo chmod 777 build
```

다시 실행하니 정상적으로 작동하는 것을 확인하였다.

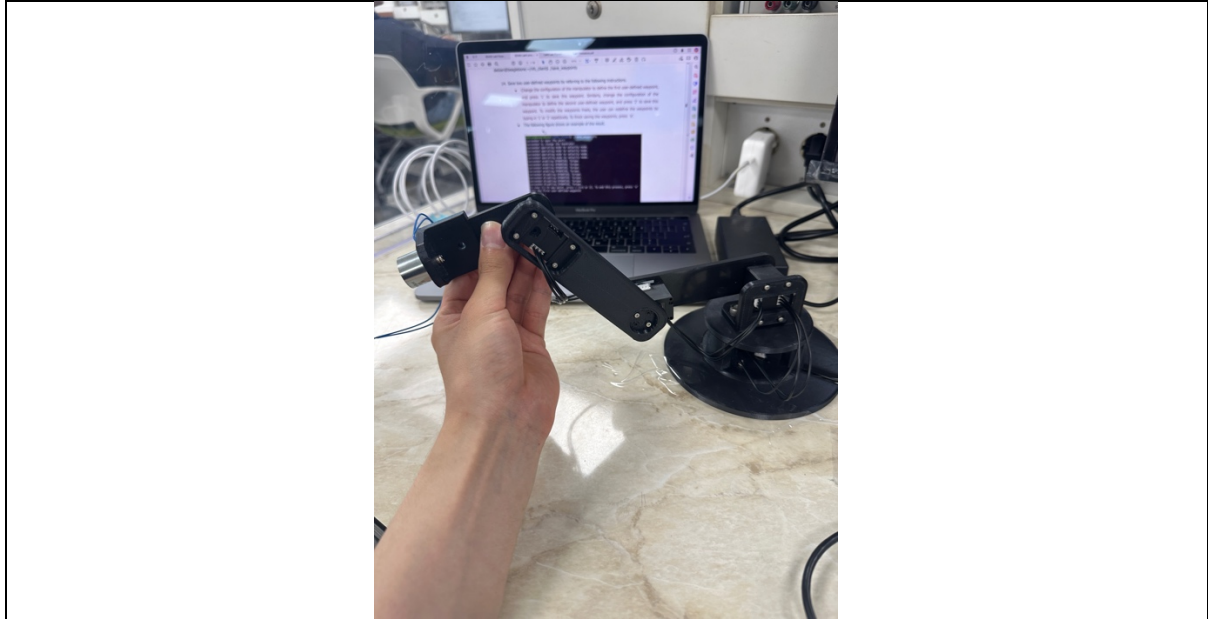


그림 6. 첫번째 waypoint를 지정할 때의 robot configuration 모습



그림 7. 두번째 waypoint를 지정할 때의 robot configuration 모습

```

alvinjinsung@beaglebone: ~/nfs_client/DesignLab/20170699/EE405_ManipulatorControl/build
File Edit View Search Terminal Help
alvinjinsung@beaglebone:~/nfs_client/DesignLab/20170699/EE405_ManipulatorControl/build$ ./save_waypoints
Succeeded to open the port!
Succeeded to change the baudrate!
Succeeded operating mode to velocity mode.
Succeeded operating mode to velocity mode.
Succeeded operating mode to velocity mode.
Succeeded operating mode to velocity mode.
Succeeded enabling DYNAMIXEL Torque.
Succeeded enabling DYNAMIXEL Torque.
Succeeded enabling DYNAMIXEL Torque.
Succeeded enabling DYNAMIXEL Torque.
Succeeded disabling DYNAMIXEL Torque.
Succeeded disabling DYNAMIXEL Torque.
Succeeded disabling DYNAMIXEL Torque.
Succeeded disabling DYNAMIXEL Torque.
To save (i)-th way point, press i (i=1 or 2). To end this process, press 'e'
Save the first user-defined waypoint
waypoint1 :
0.674952
-1.77635
-1.29008
-1.45882
To save (i)-th way point, press i (i=1 or 2). To end this process, press 'e'
Save the second user-defined waypoint
waypoint2 :
2.48351
-0.871301
1.04464
0.989418
To save (i)-th way point, press i (i=1 or 2). To end this process, press 'e'
Succeeded disabling DYNAMIXEL Torque.
Succeeded disabling DYNAMIXEL Torque.
Succeeded disabling DYNAMIXEL Torque.
Succeeded disabling DYNAMIXEL Torque.
alvinjinsung@beaglebone:~/nfs_client/DesignLab/20170699/EE405_ManipulatorControl/build$

```

그림 8. Save_waypoints의 terminal 실행 모습

```

waypoints.csv
1 0.674952, -1.77635, -1.29008, -1.45882
2 2.48351, -0.871301, 1.04464, 0.989418

```

그림 9. Waypoints.csv파일이 정상적으로 생성된 모습

다음 단계로 위에서 저장된 waypoints를 robot이 따라갈 수 있도록 trajectory generation을 수행하였다. manipulator_control_week1.cpp 코드를 통해 수행하며 manipulator의 DOF가 4, control frequency가 100Hz이기에 int dof를 4로, double control_freq를 100으로 지정하였다. DOF가 4이기 때문에 targetQ와 targetQdot의 dimension 역시 4이다. 정상적인 작동을 위해 initialization(current_q) 함수와 TrajectoryGeneration() 함수를 구현해야 한다. initialization(current_q) 함수는 각 row가 joint space에서 direct teaching을 위한 waypoint를 의미하는 stack_waypoints를 define하고 print하는 역할을 한다. 첫번째 waypoint는 manipulator의 initial position이고, 두번째, 세번째 waypoint는 waypoints.csv 파일에 정의된 waypoints이다. stack_waypoints의 dimension은 3 by 4이다. TrajectoryGeneration() 함수는 first-order polynomial을 이용하여 지정된 세개의 waypoints를 반복적으로 따라 이동하는 trajectory를 생성한다. 두 waypoints 사이 node의 개수는 NumNodes로 지정한다. 해당 함수는 각 4개의 motor의 desired joint angle을 반환한다. 위 두 함수는 DirectTeaching.cpp 코드에 구현되어 있으며 코드는 아래와 같다.


```

/**
 * @file DirectTeaching.cpp
 * @author Jinyeong Jeong (jinyeong.jeong@kaist.ac.kr)
 * @brief
 * @version 0.1
 * @date 2023-04-04
 *
 * @copyright Copyright (c) 2023
 *
 */
#include "DirectTeaching.hpp"

DirectTeaching::DirectTeaching(int dof_args)
{
    NumWaypoints=3; // the number of waypoints
    dof=dof_args; // DOF of the manipulator
    stack_waypoints.resize(NumWaypoints, dof); // Each row in the stack_waypoints matrix
    represents a waypoint for direct teaching

    NumNodes=400; // the number of nodes
    NodeIndex=0; // node index. The range of NodeIndex is 0 ~ (NumNodes-1)
    StartPointIndex=0; // Waypoint index of start point in trajectory generation. The range
    of StartPointIndex is 0 ~ (NumWaypoints-1)
    EndPointIndex=1; // Waypoint index of end point in trajectory generation. The range of
    EndPointIndex is 0 ~ (NumWaypoints-1)
}

DirectTeaching::~~DirectTeaching()
{
}

// Define stack_waypoints where each row represents a waypoint for direct teaching in joint
space
// The first waypoint for direct teaching is the initial joint angle when the program runs
void DirectTeaching::initialization(const Eigen::VectorXd &current_q)
{
    // Save two user-defined waypoints to the stack_waypoints using the "waypoints.csv" file
    // Note that the first waypoint for direct teaching is the initial joint angle when the
    program runs
    stack_waypoints <<current_q(0),    current_q(1),    current_q(2),    current_q(3),
    MatrixFileIO::openData("waypoints.csv");

    std::cout << "Each row represents a waypoint for direct teaching in joint space: \n" <<
    stack_waypoints << std::endl;
}

// Generate trajectory between waypoints by using the 1st order polynomial
Eigen::VectorXd DirectTeaching::TrajectoryGeneration()
{
    Eigen::VectorXd targetQ(dof); targetQ.setZero();

    for(int j=0; j<dof; j++){
        targetQ(j)=(stack_waypoints(EndPointIndex, j) - stack_waypoints(StartPointIndex,
j))/NumNodes*NodeIndex + stack_waypoints(StartPointIndex, j);

        /* Implement here. targetQ(j) is the result of the first order polynomial interpolation
        between stack_waypoints(StartPointIndex, j) and stack_waypoints(EndPointIndex, j) */
    }

    NodeIndex++;

    if(NodeIndex>=NumNodes){ // NodeIndex is out of range. The next targetQ reaches the end
    point.

        StartPointIndex++; // Move the start point to the next waypoint
        EndPointIndex++; // Mote the end point to the next waypoint

        NodeIndex = 0; /* Implement here.*/

        if(StartPointIndex>=NumWaypoints){ // StartPointIndex is out of range

```

```

        StartPointIndex = 0; /* Implement here.*/
    }
    if(EndPointIndex>=NumWaypoints){ // EndPointIndex is out of range
        EndPointIndex = 0; /* Implement here.*/
    }
}

return targetQ;
}

```

그림 10. DirectTeaching.cpp 코드

위의 코드에서 point A와 point B 사이 first-order polynomial interpolation의 결과(=C)는 아래와 같이 나타난다.

$$C = \frac{B - A}{NumNodes} * NodeIndex + A$$

(NumNodes: 두 waypoints(A, B) 사이 node 수, NodeIndex: 0 ~ (Numnodes-1))

이를 구현한 부분은 아래와 같다.

```

for(int j=0; j<dof; j++){
    targetQ(j)=(stack_waypoints(EndPointIndex, j) - stack_waypoints(StartPointIndex, j))/NumNodes*NodeIndex + stack_waypoints(StartPointIndex, j);

    /* Implement here. targetQ(j) is the result of the first order polynomial interpolation
    between stack_waypoints(StartPointIndex, j) and stack_waypoints(EndPointIndex, j) */
}

```

그림 11. First-order polynomial interpolation을 구현한 모습

Waypoints를 모두 찍고 다시 처음 waypoints로 돌아와 동작을 계속 반복하도록 하기 위해 index 들을 적당한 condition에서 다시 초기화 시키는 작업을 수행하였다. 해당 과정은 아래 부분에 구현하였다.

```

if(NodeIndex>=NumNodes){ // NodeIndex is out of range. The next targetQ reaches the end point
    StartPointIndex++; // Move the start point to the next waypoint
    EndPointIndex++; // Move the end point to the next waypoint

    NodeIndex=0; /* Implement here.*/

    if(StartPointIndex>=NumWaypoints){ // StartPointIndex is out of range
        StartPointIndex=0; /* Implement here.*/
    }
    if(EndPointIndex>=NumWaypoints){ // EndPointIndex is out of range
        EndPointIndex=0; /* Implement here.*/
    }
}

```

그림 12. Condition에 맞춰 index 값을 초기화하는 모습

DirectTeaching.cpp 코드를 완성하고 해당 코드를 활용하여 manipulator_control_week1.cpp 코드에서 stack_waypoints matrix를 initialization() 함수를 통해 define하고 targetQ를 TrajectoryGeneration() 함수를 통해 지정하였다. 다음으로 velocity mode를 통해 joint space P controller를 구현하였다. Controller는 아래의 식으로 표현될 수 있다.

$$targetQdot = -Kp * (current_q - target_q)$$

anipulator_control_week1.cpp 코드와 controller를 구현한 부분은 아래와 같다.


```

/**
 * @file manipulator_control_week1.cpp
 * @author Jiwan Han (jw.han@kaist.ac.kr), Jinyeong Jeong (jinyeong.jeong@kaist.ac.kr)
 * @brief
 * @version 0.1
 * @date 2023-03-27
 *
 * @copyright Copyright (c) 2023
 *
 */

#include <iostream>
#include <fstream>
#include <memory>
#include <string>
#include <unistd.h>
#include <chrono>
#include <Eigen/Dense>
#include <csignal>

#include "Controller/ArticulatedSystem.hpp"
#include "Common/DirectTeaching.hpp"

using namespace Eigen;
using namespace std;

////////// IMPORTANT BELOW COMMAND //////////
// cat /sys/bus/usb-serial/devices/ttyUSB0/latency_timer
// echo 1 | sudo tee /sys/bus/usb-serial/devices/ttyUSB0/latency_timer
// https://emanual.robotis.com/docs/en/dxl/x/xc330-t288/#
////////// IMPORTANT BELOW COMMAND //////////

bool g_isShutdown = false;

/**
 * @brief Pressing Ctrl + C in a terminal typically sends a SIGINT signal that can cause a
 * running program to terminate.
 * This thread continuously check the signal to stop main function.
 *
 * @param signum signal number: if you push the Ctrl + C on the terminal, It has a positive
 * interger number.
 */
void signalHandler(int signum)
{
    std::cout << "Interrupt signal (" << signum << ") received.\n";
    // Cleanup and close up things here
    // Terminate program
    if(signum > 0)
    {
        g_isShutdown = true;
    }
}

int main(int argc, char *argv[])
{
    signal(SIGINT, signalHandler); // For Checking the Ctrl + C.

    // The manipulator has 4 DOF
    int dof = 4;

    // The control frequency is 100Hz
    double control_freq = 100;
    double dt = 1/control_freq;

    // Dynamixel setup, ID, position resolution, motor torque constant, etc.
    std::vector<uint8_t> dynamixel_id_set;
    std::vector<int32_t> position_resolution;
    std::vector<double> motor_torque_constants;
    for (size_t i = 0; i < dof; i++)
    {
        dynamixel_id_set.emplace_back(i);
        position_resolution.emplace_back(POSITION_RESOLUTION);
        motor_torque_constants.emplace_back(MOTOR_TORQUE_CONSTANT);
    }
}

```

```

// Initialize the Articulated_system,
// We need a dof of the system, Operation Mode(Position, Velocity, Current),
// USB Port information (for serial communication),
// and dynamixel setup
ArticulatedSystem articulated_system(dof, ArticulatedSystem::Mode::VELOCITY,
"/dev/ttyUSB0",
dynamixel_id_set, position_resolution,
motor_torque_constants); //4 DOF manipulator

/**
 * The member function articulated_system.GetJointAngle() returns the joint angles of the
 robot system, which in this context refer to the motor angle.
 */
Eigen::VectorXd currentQ = articulated_system.GetJointAngle();

// Declare targetQ(motor angle), targetQdot(motor velocity)
Eigen::VectorXd targetQ(dof); targetQ.setZero();
Eigen::VectorXd targetQdot(dof); targetQdot.setZero();

/**
 * @brief time instant is initialized.
 * loop_start : represents the moment when the 'while' loop starts.
 * loop_end : represents the moment when the 'while' loop ends.
 * current_time : current time instant.
 * initial_time : The time instant when the program(=main function) starts.
 */
std::chrono::system_clock::time_point loop_start= std::chrono::system_clock::now();
std::chrono::system_clock::time_point loop_end= std::chrono::system_clock::now();
std::chrono::system_clock::time_point current_time= std::chrono::system_clock::now();
std::chrono::system_clock::time_point initial_time= std::chrono::system_clock::now();

// P gain.
Eigen::MatrixXd Kp(dof, dof); Kp.setZero();

// ////////////////////////////////////// Direct teaching mode
////////////////////////////////////
DirectTeaching direct_teaching(dof);
direct_teaching.initialization(currentQ); // initialization for direct teaching
// ////////////////////////////////////// Direct teaching mode END
////////////////////////////////////

// Please refer to the signalHandler function. If g_isShutdown is true, the program will
be shut down.
while(!g_isShutdown){
    current_time= std::chrono::system_clock::now();
    auto loop_elapsed_time_microsec =
std::chrono::duration_cast<std::chrono::microseconds>(current_time - loop_start);
    auto total_elapsed_time_microsec =
std::chrono::duration_cast<std::chrono::microseconds>(current_time - initial_time);

    // For 100 Hz control loop.
    if(loop_elapsed_time_microsec.count()>=dt*1e6)
    {
        double total_elapsed_time_sec =
static_cast<double>(total_elapsed_time_microsec.count())/1e6;
        loop_start = std::chrono::system_clock::now();

        currentQ = articulated_system.GetJointAngle();

        // ////////////////////////////////////// Direct teaching mode
        //////////////////////////////////////
        targetQ = direct_teaching.TrajectoryGeneration(); // get target joint (motor) angle
to use the following P position control
        // ////////////////////////////////////// Direct teaching mode END
        //////////////////////////////////////

        ////////////////////////////////////// P Position control USING Velocity Mode
        //////////////////////////////////////
        for (size_t i = 0; i < dof; i++)
        {
            Kp(i,i) = 4;
        }
        targetQdot = -Kp * (currentQ - targetQ); /* Implement here. You have to implement
joint space P position Controller */
    }
}

```

```

        articulated_system.SetGoalVelocity(targetQdot); // velocity control mode
        ////////////////////////////////////////////////// P Position control USING Velocity Mode END
        //////////////////////////////////////////////////

        loop_end= std::chrono::system_clock::now();
        auto one_step_calculation_time =
std::chrono::duration_cast<std::chrono::microseconds>(loop_end - loop_start);
    }

    return 0;
}

```

그림 13. Manipulator_control_week1.cpp 코드

```

// P Position control USING Velocity Mode
for (size_t i = 0; i < dof; i++)
{
    Kp(i,i) = 4;
}
targetQdot = -Kp * (currentQ - targetQ); /* Implement here. You have to implement joint space P position Controller */
articulated_system.SetGoalVelocity(targetQdot); // velocity control mode
// P Position control USING Velocity Mode END

```

그림 14. Velocity mode controller 구현 모습

CMakeLists.txt 파일을 알맞게 수정하고 ArticulatedSystem.cpp 파일 역시 필요한 부분을 uncomment한 이후 build하였다.

```

### 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_manipulator STATIC include/Controller/ArticulatedSystem.cpp
include/FileIO/MatrixFileIO.cpp
include/Common/DirectTeaching.cpp
# include/Common/Kinematics.cpp
include/KeyInput/getche.cpp
)

### week1 direct teaching

```

그림 15. CMakeLists.txt 파일을 수정한 모습

```

### add_executable: Make executable file using following manipulator_control_week1.cpp code.
add_executable(manipulator_control_week1 src/manipulator_control_week1.cpp )

target_link_libraries(manipulator_control_week1 EE405_manipulator dxl_sbc_cpp)

```

그림 16. CMakeLists.txt 파일을 수정한 모습

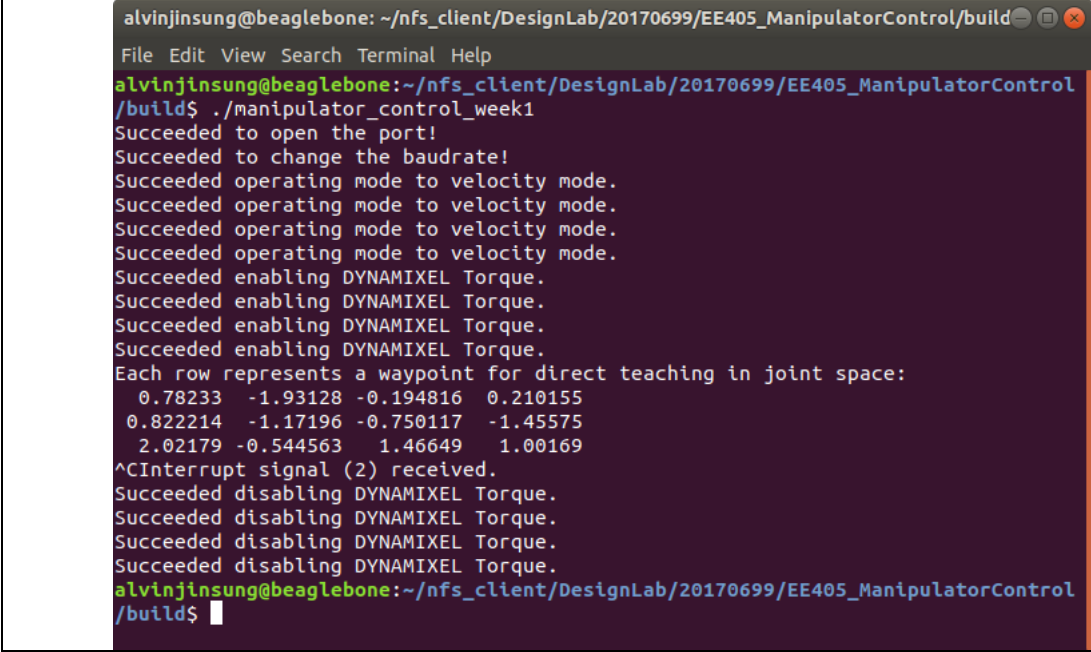
```

// Enable BusWatch dog (Search this very useful function.)
dxl_comm_result = packetHandler->writeByteTxRx(portHandler, dynamixel_id_set.at(i), ADDR_BUS_WATCHDOG, 3, &dxl_error); // 3 = 3 * 20 ms

```

그림 17. ArticulatedSystem.cpp 파일을 수정한 모습

Build된 executable을 실행시켰을 때 우리가 원하는 대로 세개의 waypoints를 순서대로 반복적으로 따라가는 모습을 보여주었다. Initial position을 첫번째 waypoint로, waypoints.csv 파일에 정의된 waypoints를 두번째, 세번째 waypoint로 '첫번째 waypoint' -> '두번째 waypoint' -> '세번째 waypoint' -> '첫번째 waypoint' -> ... 로 반복적으로 실행되었다. 두 waypoints 사이 node 개수가 400으로 지정되어 있고 control frequency가 100Hz이기에 4(=400/100)초가 다음 waypoint로 이동하는데 소요되었다. 아래는 terminal 실행 모습이다.



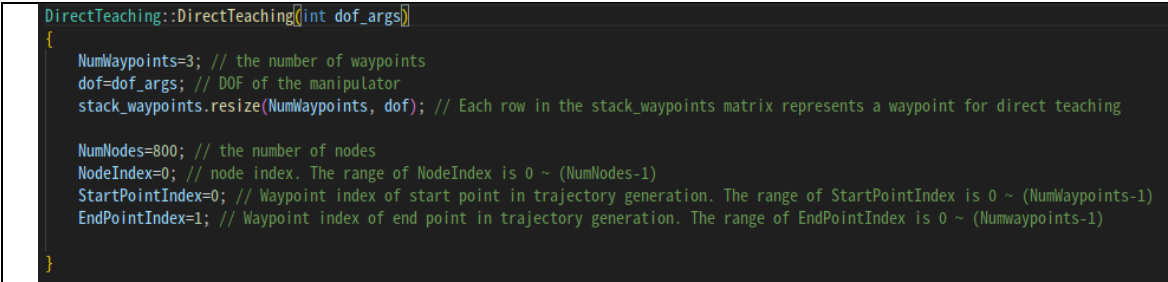
```

alvinjinsung@beaglebone: ~/nfs_client/DesignLab/20170699/EE405_ManipulatorControl/build
File Edit View Search Terminal Help
alvinjinsung@beaglebone:~/nfs_client/DesignLab/20170699/EE405_ManipulatorControl/build$ ./manipulator_control_week1
Succeeded to open the port!
Succeeded to change the baudrate!
Succeeded operating mode to velocity mode.
Succeeded operating mode to velocity mode.
Succeeded operating mode to velocity mode.
Succeeded operating mode to velocity mode.
Succeeded enabling DYNAMIXEL Torque.
Succeeded enabling DYNAMIXEL Torque.
Succeeded enabling DYNAMIXEL Torque.
Succeeded enabling DYNAMIXEL Torque.
Each row represents a waypoint for direct teaching in joint space:
  0.78233  -1.93128  -0.194816  0.210155
  0.822214  -1.17196  -0.750117  -1.45575
  2.02179  -0.544563  1.46649  1.00169
^CInterrupt signal (2) received.
Succeeded disabling DYNAMIXEL Torque.
Succeeded disabling DYNAMIXEL Torque.
Succeeded disabling DYNAMIXEL Torque.
Succeeded disabling DYNAMIXEL Torque.
alvinjinsung@beaglebone:~/nfs_client/DesignLab/20170699/EE405_ManipulatorControl/build$

```

그림 18. Manipulator_control_week1을 실행한 모습

두 waypoints 사이 node 개수를 변경하면 속도가 달라지는 것을 확인할 수 있었다. 800으로 변경하여 실행하였을 때 400일 때보다 훨씬 천천히 이동하는 것을 실제로 볼 수 있었다.



```

DirectTeaching::DirectTeaching(int dof_args)
{
    NumWaypoints=3; // the number of waypoints
    dof=dof_args; // DOF of the manipulator
    stack_waypoints.resize(NumWaypoints, dof); // Each row in the stack_waypoints matrix represents a waypoint for direct teaching

    NumNodes=800; // the number of nodes
    NodeIndex=0; // node index. The range of NodeIndex is 0 ~ (NumNodes-1)
    StartPointIndex=0; // Waypoint index of start point in trajectory generation. The range of StartPointIndex is 0 ~ (NumWaypoints-1)
    EndPointIndex=1; // Waypoint index of end point in trajectory generation. The range of EndPointIndex is 0 ~ (Numwaypoints-1)
}

```

그림 19. NumNodes를 800으로 변경한 모습

Problem 5B. Teleoperation

두번째 task로 keyboard input에 따라 robot을 teleoperate하기 위해서 keyboard input이 들어올 때 해당하는 action을 수행하도록 하는 GetKeyboardInput() function을 작성하였다. Lab에서 사용하는 manipulator의 DOF가 4이고 control frequency가 100Hz, velocity mode를 사용해 joint angle을 control하는 것을 확인하였다. 그 다음 다음 입력들이 keyboard input으로 들어왔을 때 해당하는 action들을 지정하였다.

- r: move 1cm in the world z-axis
- f: move -1cm in the world z-axis
- w: move 1cm in the world y-axis
- s: move -1cm in the world y-axis
- d: move 1cm in the world x-axis
- a: move -1cm in the world x-axis
- i: move to the initial position
- e: end the teleoperation

Teleoperation의 position resolution은 1cm로 설정하였으며 position_resolution variable에서 이를 설정하였다. Eigen::Vector3d variable인 task_command를 정의해 각 keyboard input에 대해 올바른 vector값을 지정해 주었다. 해당 코드(manipulator_control_week2.cpp 중)는 아래와 같다.

```
void GetKeyboardInput()
{
    double position_resolution=0.01; // The position resolution for teleoperation is 1cm
    while(1)
    {
        std::cout << "Press 'r', 'f', 'w', 's', 'd', or 'a' to move the end-effector. Press 'i' to move to the initial position. Press 'e' to end this mode" << std::endl;
        key = getch(); // Get keyboard input from the user
        is_key_updated=true; // When a new keyboard input is received, is_key_updated is set to true
        if(key == 'r') {
            task_command<<0, 0, position_resolution;
        }
        else if(key == 'f') {
            task_command<<0, 0, -position_resolution;
        }
        else if(key == 'w') {
            task_command<<0, position_resolution, 0; /* implement here */
        }
        else if(key == 's') {
            task_command<<0, -position_resolution, 0; /* implement here */
        }
        else if(key == 'd') {
            task_command<<position_resolution, 0, 0; /* implement here */
        }
        else if(key == 'a') {
            task_command<<-position_resolution, 0, 0; /* implement here */
        }
        else if(key == 'e') {
            break;
        }
    }
}
```

그림 20. GetKeyboardInput() function

main function에서 확인해 보면 end effector position을 keyboard input을 받는 와중에 control하기 위해서 thread t1을 생성한 것을 확인할 수 있다.

```
std::thread t1 = std::thread(GetKeyboardInput); // Create thread t1 to get the keyboard input from the user
```

그림 21. thread t1을 생성한 모습

```
t1.join(); // allows the main function to wait until t1 completes its execution
```

그림 22. t1.join()을 통해 t1 execution이 끝날 때까지 main function을 wait하는 모습

다음으로 world linear velocity 계산을 위한 Jacobian matrix를 compute하였다. Kinematics.cpp 파일에서 GetRotationMatrix(r, p, y), GetJacobianMatrix(current_q) function을 구현하였다. GetRotationMatrix() function은 다음의 successive rotation 결과를 return한다.

- 1) Rotation of y (rad) along the body of z-axis
- 2) Rotation of p (rad) along the body of y-axis
- 3) Rotation of r (rad) along the body of x-axis

코드는 아래와 같다.

```

Eigen::Matrix3d Kinematics::GetRotationMatrix(double r, double p, double y)
{
    // Rx : Rotation of r (rad) along the body x-axis
    // Ry : Rotation of p (rad) along the body y-axis
    // Rz : Rotation of y (rad) along the body z-axis
    Eigen::Matrix3d Rx, Ry, Rz;
    Rx.setZero(); Ry.setZero(); Rz.setZero();

    Rx << 1, 0, 0,
        0, cos(r), -sin(r),
        0, sin(r), cos(r);
    Ry << cos(p), 0, sin(p),
        0, 1, 0,
        -sin(p), 0, cos(p);
    Rz << cos(y), -sin(y), 0,
        sin(y), cos(y), 0,
        0, 0, 1;
    return Rz*Ry*Rx;
}

```

그림 23. GetRotationMatrix(r, p, y) function

위 함수를 활용하여 GetJacobianMatrix() function을 구현하였다. 우리가 얻어야 하는 Jacobian Matrix는 DOF가 4이고 x, y, z의 task space로 나타나기에 3 by 4 dimension을 가진다. Manipulator의 configuration과 Jacobian matrix 계산 과정은 아래와 같이 나타난다.

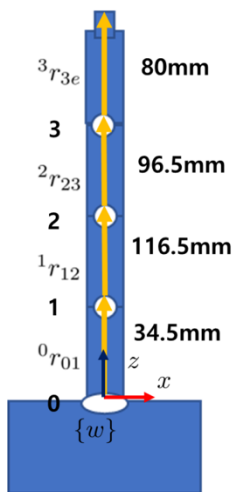


그림 24. Manipulator configuration

$$\begin{aligned}
 {}^w r_{3e} &= {}^w R_3 {}^3 r_{3e} \\
 {}^w r_{2e} &= {}^w r_{3e} + {}^w R_2 {}^2 r_{23} \\
 {}^w r_{1e} &= {}^w r_{2e} + {}^w R_1 {}^1 r_{12} \\
 {}^w r_{0e} &= {}^w r_{1e} + {}^w R_0 {}^0 r_{01} \\
 {}^w V_e &= {}^w \Omega_0 \times {}^w r_{0e} + {}^w \Omega_1 \times {}^w r_{1e} + {}^w \Omega_2 \times {}^w r_{2e} + {}^w \Omega_3 \times {}^w r_{3e} \\
 &= [{}^w P_0 \times {}^w r_{0e} \quad {}^w P_1 \times {}^w r_{1e} \quad {}^w P_2 \times {}^w r_{2e} \quad {}^w P_3 \times {}^w r_{3e}] \begin{bmatrix} \dot{q}_0 \\ \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \end{bmatrix} \\
 &= J(q) \dot{q}
 \end{aligned}$$

그림 25. Jacobian Matrix 계산과정

이를 코드로 구현하면 아래와 같다.

```

Eigen::Vector3d r3e_w= Rw3*r3e_3; /* implement a vector pointing the end-effector frame from the ID 3 motor frame represented in the world frame */
Eigen::Vector3d r2e_w= r3e_w + Rw2*r23_2; /* implement a vector pointing the end-effector frame from the ID 2 motor frame represented in the world frame */
Eigen::Vector3d r1e_w= r2e_w + Rw1*r12_1; /* implement a vector pointing the end-effector frame from the ID 1 motor frame represented in the world frame */
Eigen::Vector3d r0e_w= r1e_w + Rw0*r01_0; /* implement a vector pointing the end-effector frame from the ID 0 motor frame represented in the world frame */

Eigen::MatrixXd JacobianTranslationEE(3, 4); JacobianTranslationEE.setZero();

JacobianTranslationEE << Pw0.cross(r0e_w), Pw1.cross(r1e_w), Pw2.cross(r2e_w), Pw3.cross(r3e_w);/* implement the Jacobian matrix for world linear velocity */

```

그림 26. GetJacobianMatrix(current_q) function

위의 함수들을 모두 합쳐 Kinematics.cpp 코드를 완성하면 아래와 같이 나타난다.

```

/**
 * @file Kinematics.cpp
 * @author Jinyeong Jeong (jinyeong.jeong@kaist.ac.kr)
 * @brief
 * @version 0.1
 * @date 2023-04-04

```



```

*
* @copyright Copyright (c) 2023
*
*/
#include "Kinematics.hpp"

Kinematics::Kinematics()
{
}

Kinematics::~Kinematics()
{
}

// Return the following successive rotation
// rotation of y (rad) along the body z-axis ->
// rotation of p (rad) along the body y-axis ->
// rotation of r (rad) along the body x-axis
Eigen::Matrix3d Kinematics::GetRotationMatrix(double r, double p, double y)
{
    // Rx : Rotation of r (rad) along the body x-axis
    // Ry : Rotation of p (rad) along the body y-axis
    // Rz : Rotation of y (rad) along the body z-axis
    Eigen::Matrix3d Rx, Ry, Rz;
    Rx.setZero(); Ry.setZero(); Rz.setZero();

    Rx << 1, 0, 0,
        0, cos(r), -sin(r),
        0, sin(r), cos(r);
    Ry << cos(p), 0, sin(p),
        0, 1, 0,
        -sin(p), 0, cos(p);
    Rz << cos(y), -sin(y), 0,
        sin(y), cos(y), 0,
        0, 0, 1;
    return Rz*Ry*Rx;
}

Eigen::MatrixXd Kinematics::GetJacobianMatrix(const Eigen::VectorXd &current_q)
{
    // Rotation of the ID 0 motor frame with respect to the world frame
    Eigen::Matrix3d Rw0_;
    Rw0_ << 0, 0, -1,
        -1, 0, 0,
        0, 1, 0;

    // Rotation by the ID 0 motor revolution
    Eigen::Matrix3d R0_0=GetRotationMatrix(0, current_q(0), 0);

    // Rotation of the ID 1 motor frame with respect to the ID 0 motor frame
    Eigen::Matrix3d R01_;
    R01_ << 0, -1, 0,
        0, 0, 1,
        -1, 0, 0;

    // Rotation by the ID 1 motor revolution
    Eigen::Matrix3d R1_1=GetRotationMatrix(0, current_q(1), 0);

    // Rotation of the ID 2 motor frame with respect to the ID 1 motor frame
    Eigen::Matrix3d R12_=GetRotationMatrix(0, 0, M_PI);

    // Rotation by the ID 2 motor revolution
    Eigen::Matrix3d R2_2=GetRotationMatrix(0, current_q(2), 0);

    // Rotation of the ID 3 motor frame with respect to the ID 2 motor frame
    Eigen::Matrix3d R23_=GetRotationMatrix(0, 0, M_PI);

    // Rotation by the ID 3 motor revolution
    Eigen::Matrix3d R3_3=GetRotationMatrix(0, current_q(3), 0);

    // Rotation of the ID 0 motor frame with respect to the world frame
    Eigen::Matrix3d Rw0=Rw0_*R0_0;
    // Rotation of the ID 1 motor frame with respect to the world frame
    Eigen::Matrix3d Rw1=Rw0*R01_*R1_1;
    // Rotation of the ID 2 motor frame with respect to the world frame

```

```

Eigen::Matrix3d Rw2=Rw1*R12_*R2_2;
// Rotation of the ID 3 motor frame with respect to the world frame
Eigen::Matrix3d Rw3=Rw2*R23_*R3_3;

Eigen::Vector3d motor_axis={0, 1, 0}; // This is the motor axis represented in the body
frame (= the motor frame)

Eigen::Vector3d Pw0=Rw0*motor_axis; // The ID 0 motor axis represented in the world frame
Eigen::Vector3d Pw1=Rw1*motor_axis; // The ID 1 motor axis represented in the world frame
Eigen::Vector3d Pw2=Rw2*motor_axis; // The ID 2 motor axis represented in the world frame
Eigen::Vector3d Pw3=Rw3*motor_axis; // The ID 3 motor axis represented in the world frame

Eigen::Vector3d r01_0={0, 0.0345, 0}; // A vector pointing the ID 1 motor frame from the
ID 0 motor frame represented in the ID 0 motor frame
Eigen::Vector3d r12_1={0, 0, 0.1165}; // A vector pointing the ID 2 motor frame from the
ID 1 motor frame represented in the ID 1 motor frame
Eigen::Vector3d r23_2={0, 0, 0.0965}; // A vector pointing the ID 3 motor frame from the
ID 2 motor frame represented in the ID 2 motor frame
Eigen::Vector3d r3e_3={0, 0, 0.08}; // A vector pointing the end-effector frame from the
ID 3 motor frame represented in the ID 3 motor frame

Eigen::Vector3d r3e_w= Rw3*r3e_3; /* implement a vector pointing the end-effector frame
from the ID 3 motor frame represented in the world frame */
Eigen::Vector3d r2e_w= r3e_w + Rw2*r23_2; /* implement a vector pointing the end-effector
frame from the ID 2 motor frame represented in the world frame */
Eigen::Vector3d r1e_w= r2e_w + Rw1*r12_1; /* implement a vector pointing the end-effector
frame from the ID 1 motor frame represented in the world frame */
Eigen::Vector3d r0e_w= r1e_w + Rw0*r01_0; /* implement a vector pointing the end-effector
frame from the ID 0 motor frame represented in the world frame */

Eigen::MatrixXd JacobianTranslationEE(3, 4); JacobianTranslationEE.setZero();

JacobianTranslationEE << Pw0.cross(r0e_w), Pw1.cross(r1e_w), Pw2.cross(r2e_w),
Pw3.cross(r3e_w); /* implement the Jacobian matrix for world linear velocity */

return JacobianTranslationEE;
}

```

그림 27. Kinematics.cpp 코드

이제 위의 Jacobian matrix를 활용하여 inverse kinematics를 통해 원하는 joint angle을 구해낸다. 우리는 Jacobian matrix의 right-inverse를 사용할 것이기에 end-effector의 position이 singularities로부터 멀어야 한다. Singularities로부터 먼 $[0, 0, \pi/2, -\pi/2]$ 를 teleoperation의 initial position으로 설정하였다. 따라서 program이 시작되면 처음 Manipulator의 configuration으로부터 $[0, 0, \pi/2, -\pi/2]$ 로 5초간 움직인 다음 teleoperation이 시작된다. 해당 내용을 구현한 부분은 아래와 같다.

```

if(!is_ready_for_teleoperation){ // Move the end-effector to initial position before starting teleoperation
    targetQ = (initial_position - initialQ)/NodeNum*NodeIndex+initialQ; // trajectory generation using the first-order polynomial
    NodeIndex++;
    if(NodeIndex>NodeNum) {
        is_ready_for_teleoperation=true; // Now, it is ready for teleoperation
        std::cout<<"Start teleoperation!"<<std::endl;
    }
}

```

그림 28. Teleoperation의 initial position($[0, 0, \pi/2, -\pi/2]$)으로 이동

Teleoperation mode가 시작되면 keyboard input에 따라 targetQ가 결정된다. Keyboard input이 'i'일 경우 targetQ는 $[0, 0, \pi/2, -\pi/2]$ 이며 keyboard input이 'r', 'f', 'w', 's', 'd', 'a'일 경우 아래의 식에 의해 targetQ가 계산된다.

$$q_d = q + J^+(q)(x_d - x)$$

$$J^+ = J^T(JJ^T)^{-1}, x_d - x = task_command$$

이를 구현한 코드는 아래와 같다.

```

else if(key=='r' || key=='f' || key=='w' || key=='s' || key=='d' || key=='a') // Move the end-effector based on the task command
{
    Eigen::MatrixXd J=Kinematics::GetJacobianMatrix(currentQ); // J is the Jacobian matrix for linear velocity represented in the world frame
    targetQ= currentQ + (J.transpose()*(J*J.transpose()).inverse()) * task_command; /*implement targetQ using inverse kinematics*/
    prev_targetQ=targetQ; // Update prev_targetQ
}

```

그림 29. Inverse kinematics로 targetQ를 계산한 모습

Keyboard input이 'e'일 경우 control loop가 종료되며 잘못된 keyboard input이 들어오거나 keyboard input이 없을 경우 targetQ를 previous targetQ로 설정한다. 전체적으로 구현한 모습은 아래와 같으며 is_key_updated는 새로운 keyboard input이 receive 되었는지 확인하는 용도로 사용된다.

```

else{
    /////////////////////////////////////////////////// Teleoperation mode Start ////////////////////////////////////////////
    if(is_key_updated){ // Perform teleoperation only when a new keyboard input is received (is_key_updated=true)
        if(key=='i') // Move to the initial position for teleoperation
        {
            targetQ=initial_position;
            prev_targetQ=targetQ; // Update prev_targetQ
        }
        else if(key=='r' || key=='f' || key=='w' || key=='s' || key=='d' || key=='a') // Move the end-effector based on the task command
        {
            Eigen::MatrixXd J=Kinematics::GetJacobianMatrix(currentQ); // J is the Jacobian matrix for linear velocity represented in the world frame
            targetQ= currentQ + (J.transpose()*(J*J.transpose()).inverse()) * task_command; /*implement targetQ using inverse kinematics*/
            prev_targetQ=targetQ; // Update prev_targetQ
        }
        else if(key=='e') // End teleoperation
        {
            break;
        }
        else // When the wrong keyboard input is received, the targetQ is the previous targetQ value
        {
            targetQ=prev_targetQ;
        }

        is_key_updated=false; // Set is_key_updated to false in order to check if the new keyboard input is received
    }
    else{ // Before receiving the new keyboard input, the targetQ is the previous targetQ value
        targetQ=prev_targetQ;
    }
}
///////////////////////////////////////////////// Teleoperation mode END ////////////////////////////////////////////
}

```

그림 30. Keyboard input에 따른 올바른 targetQ를 계산하는 모습

최종적으로 위에서 구현한 부분들을 모두 활용해 velocity mode로 joint space P position controller를 구현하였다. 구현한 코드는 아래와 같다.

```

//////////////////////////////// P Position control USING Velocity Mode ////////////////////////////////////////////
for (size_t i = 0; i < dof; i++)
{
    Kp(i,i) = 4;
}
targetQdot = -Kp*(currentQ - targetQ); /* Implement here. You have to implement joint space P position Controller */
articulated_system.SetGoalVelocity(targetQdot); // velocity control mode
//////////////////////////////// P Position control USING Velocity Mode END ////////////////////////////////////////////
loop_end= std::chrono::system_clock::now();
auto one_step_calculation_time = std::chrono::duration_cast<std::chrono::microseconds>(loop_end - loop_start);

```

그림 31. Velocity mode로 joint space P position controller를 구현한 모습

전체 manipulator_control_week2.cpp 코드는 아래와 같다.

```

/**
 * @file manipulator_control_week12.cpp
 * @author Jiwan Han (jw.han@kaist.ac.kr), Jinyeong Jeong (jinyeong.jeong@kaist.ac.kr)
 * @brief
 * @version 0.1
 * @date 2023-03-27
 *
 * @copyright Copyright (c) 2023
 *
 */

#include <iostream>

```

```

#include <fstream>
#include <memory>
#include <string>
#include <unistd.h>
#include <chrono>
#include <Eigen/Dense>
#include <csignal>
#include <thread>

#include "Controller/ArticulatedSystem.hpp"
#include "KeyInput/getche.h"
#include "Common/Kinematics.hpp"
using namespace Eigen;
using namespace std;

////////// IMPORTANT BELOW COMMAND //////////
// cat /sys/bus/usb-seria/devices/ttyUSB0/latency_timer
// echo 1 | sudo tee /sys/bus/usb-seria/devices/ttyUSB0/latency_timer
// https://emanual.robotis.com/docs/en/dxl/x/x330-t288/#
////////// IMPORTANT BELOW COMMAND //////////

// GetKeyboardInput() function : Compute the task command from the keyboard input for
teleoperation.
// The following keyboard input determines the task command. The position_resolution variable
represents the position resolution for teleoperation
// r : move 1cm in the world z-axis
// f : move -1cm in the world z-axis
// w : move 1cm in the world y-axis
// s : move -1cm in the world y-axis
// d : move 1cm in the world x-axis
// a : move -1cm in the world x-axis
// i : move to the initial position
// e : end the teleoperation
bool is_key_updated = false;
Eigen::Vector3d task_command;
char key='i';
void GetKeyboardInput()
{
    double position_resolution=0.01; // The position resolution for teleoperation is 1cm
    while(1)
    {
        std::cout << "Press 'r', 'f', 'w', 's', 'd', or 'a' to move the end-effector. Press
'i' to move to the initial position. Press 'e' to end this mode" << std::endl;
        key = getch(); // Get keyboard input from the user
        is_key_updated=true; // When a new keyboard input is received, is_key_updated is set to
true
        if(key == 'r') {
            task_command<<0, 0, position_resolution;
        }
        else if(key == 'f') {
            task_command<<0, 0, -position_resolution;
        }
        else if(key == 'w') {
            task_command<<0, position_resolution, 0; /* implement here */
        }
        else if(key == 's') {
            task_command<<0, -position_resolution, 0; /* implement here */
        }
        else if(key == 'd') {
            task_command<<position_resolution, 0, 0; /* implement here */
        }
        else if(key == 'a') {
            task_command<<-position_resolution, 0, 0; /* implement here */
        }
        else if(key == 'e') {
            break;
        }
    }
}
int main(int argc, char *argv[])
{
    std::thread t1 = std::thread(GetKeyboardInput); // Create thread t1 to get the keyboard
input from the user

    // The manipulator has 4 DOF
    int dof = 4;

```

```

// The control frequency is 100Hz
double control_freq = 100;
double dt = 1/control_freq;

// Dynamixel setup, ID, position resolution, motor torque constant, etc.
std::vector<uint8_t> dynamixel_id_set;
std::vector<int32_t> position_resolution;
std::vector<double> motor_torque_constants;
for (size_t i = 0; i < dof; i++)
{
    dynamixel_id_set.emplace_back(i);
    position_resolution.emplace_back(POSITION_RESOLUTION);
    motor_torque_constants.emplace_back(MOTOR_TORQUE_CONSTANT);
}

// Initialize the Articulated_system,
// We need a dof of the system, Operation Mode(Position, Velocity, Current),
// USB Port information (for serical communication),
// and dynamixel setup
ArticulatedSystem articulated_system(dof, ArticulatedSystem::Mode::VELOCITY,
"/dev/ttyUSB0",
dynamixel_id_set, position_resolution,
motor_torque_constants); //4 DOF manipulator

/**
 * The member function articulated_system.GetJointAngle() returns the joint angles of the
 robot system, which in this context refer to the motor angle.
 */
Eigen::VectorXd currentQ = articulated_system.GetJointAngle();
Eigen::VectorXd initialQ = articulated_system.GetJointAngle();

// Declare targetQ(motor angle), targetQdot(motor velocity)
Eigen::VectorXd targetQ(dof); targetQ.setZero();
Eigen::VectorXd targetQdot(dof); targetQdot.setZero();

bool is_ready_for_teleoperation=false;
Eigen::VectorXd prev_targetQ(dof); // Declare prev_targetQ(the previous targetQ value)

// The initial joint position for teleoperation is [0, 0, PI/2, -PI/2]. This configuraiton
is far from singularities.
Eigen::VectorXd initial_position(dof); initial_position << 0, 0, M_PI/2, -M_PI/2;
prev_targetQ = initial_position;

/**
 * @brief time instant is initialized.
 * loop_start : represents the moment when the 'while' loop starts.
 * loop_end : represents the moment when the 'while' loop ends.
 * current_time : current time instant.
 * initial_time : The time instant when the program(=main function) starts.
 */
std::chrono::system_clock::time_point loop_start= std::chrono::system_clock::now();
std::chrono::system_clock::time_point loop_end= std::chrono::system_clock::now();
std::chrono::system_clock::time_point current_time= std::chrono::system_clock::now();
std::chrono::system_clock::time_point initial_time= std::chrono::system_clock::now();

// P gain.
Eigen::MatrixXd Kp(dof, dof); Kp.setZero();

// The following variables are for trajectory generation before starting teleoperation
int NodeIndex=0;
int NodeNum=500;

while(1){
    current_time= std::chrono::system_clock::now();
    auto loop_elapsed_time_microsec =
std::chrono::duration_cast<std::chrono::microseconds>(current_time - loop_start);
    auto total_elapsed_time_microsec =
std::chrono::duration_cast<std::chrono::microseconds>(current_time - initial_time);

    // For 100 Hz control loop.
    if(loop_elapsed_time_microsec.count()>=dt*1e6)
    {
        double total_elapsed_time_sec =
static_cast<double>(total_elapsed_time_microsec.count())/1e6;
        loop_start = std::chrono::system_clock::now();
    }
}

```

```

        currentQ = articulated_system.GetJointAngle();

        if(!is_ready_for_teleoperation){ // Move the end-effector to initial_position
before starting teleoperation
            targetQ = (initial_position - initialQ)/NodeNum*NodeIndex+initialQ; //
trajectory generation using the first-order polynomial
            NodeIndex++;
            if(NodeIndex>NodeNum) {
                is_ready_for_teleoperation=true; // Now, it is ready for teleopeartion
                std::cout<<"Start teleoperation!"<<std::endl;
            }
        }

        else{
            /////////////////////////////////////////////////// Teleoperation mode Start
            ///////////////////////////////////////////////////
            if(is_key_updated){ // Perform teleoperation only when a new keyboard input is
received (is_key_updated=true)
                if(key=='i') // Move to the initial position for teleoperation
                {
                    targetQ=initial_position;
                    prev_targetQ=targetQ; // Update prev_targetQ
                }
                else if(key=='r' || key=='f' || key=='w' || key=='s' || key=='d' || key=='a')
// Move the end-effector based on the task command
                {
                    Eigen::MatrixXd J=Kinematics::GetJacobianMatrix(currentQ); // J is the
Jacobian matrix for linear velocity represented in the world frame
                    targetQ= currentQ + (J.transpose()*(J*J.transpose()).inverse()) *
task_command; /*implement targetQ using inverse kinematics*/
                    prev_targetQ=targetQ; // Update prev_targetQ
                }
                else if(key=='e'){ // End teleoperation
                    break;
                }
                else // When the wrong keyboard input is received, the targetQ is the privious
targetQ value
                {
                    targetQ=prev_targetQ;
                }

                is_key_updated=false; // Set is_key_updated to false in order to check if
the new keyboard input is received
            }
            else{ // Before receiving the new keyboard input, the targetQ is the privious
targetQ value
                targetQ=prev_targetQ;
            }
            /////////////////////////////////////////////////// Teleoperation mode END
            ///////////////////////////////////////////////////

            /////////////////////////////////////////////////// P Position control USING Velocity Mode
            ///////////////////////////////////////////////////
            for (size_t i = 0; i < dof; i++)
            {
                Kp(i,i) = 4;
            }
            targetQdot = -Kp*(currentQ - targetQ); /* Implement here. You have to implement
joint space P position Controller */
            articulated_system.SetGoalVelocity(targetQdot); // velocity control mode
            /////////////////////////////////////////////////// P Position control USING Velocity Mode END
            ///////////////////////////////////////////////////
            loop_end= std::chrono::system_clock::now();
            auto one_step_calculation_time =
std::chrono::duration_cast<std::chrono::microseconds>(loop_end - loop_start);
        }
    }

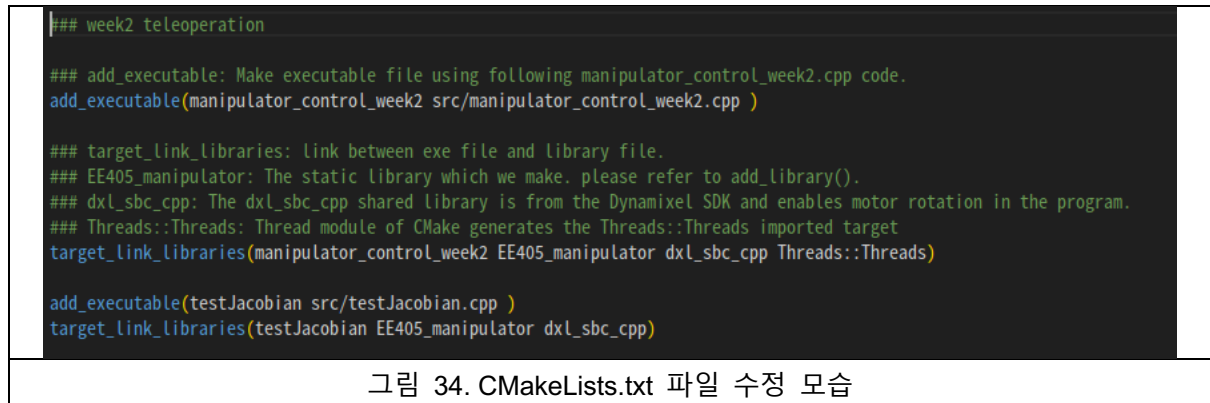
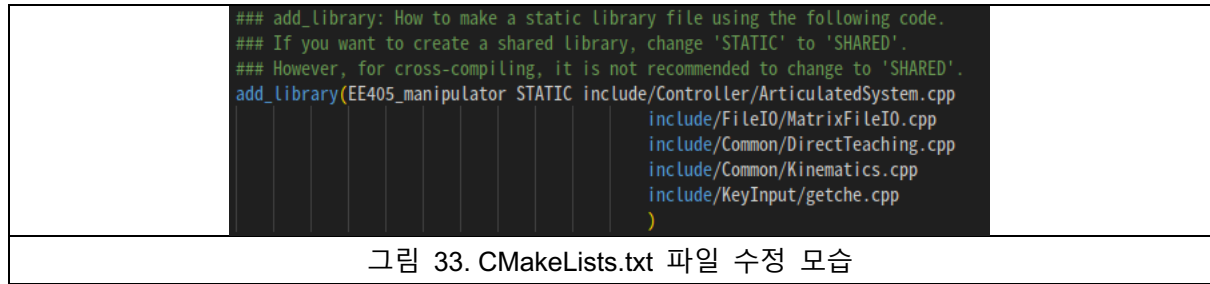
    t1.join(); // allows the main function to wait until t1 completes its execution

    return 0;
}

```

그림 32. manipulator_control_week2.cpp 코드

CMakeLists.txt 파일에서 위의 수정사항들을 반영하여 teleoperation에 필요한 코드들을 build하기 위해 필요한 부분을 적절히 수정하고 성공적으로 build하였다.



Program을 실행해 보기 전에 우리가 계산한 GetJacobianMatrix(current_q)가 올바른 Jacobian matrix를 return하는지 체크하는 과정이 필요하다. Beaglebone에서 “testJacobian”을 실행하여 joint angle을 입력하였다(“0 0 0 0\n”). 그 다음 “getJacobian”을 실행하여 동일한 joint angle 입력 후 같은 Jacobian matrix를 출력하는지 확인하였다. “getJacobian”이 처음에 실행되지 않아 아래의 명령어를 입력하여 다시 실행하였다.

```
$ cd ~/EE405_ManipulatorControl/build
$ sudo chmod 777 getJacobian
```

실행 시 동일한 Jacobian matrix값을 올바르게 출력하는 것을 확인할 수 있었다.



마지막으로 build를 통해 생성된 “manipulator_control_week2” executable을 실행하여 teleoperation이 정상적으로 작동하는지 확인하였다. 프로그램 실행 시 manipulator가 원래의 joint angle에서 $[0, 0, \pi/2, -\pi/2]$ 로 5초간 이동하는 것을 확인하였다.



그림 35. 프로그램 실행 시 $[0, 0, \pi/2, -\pi/2]$ 로 정상적으로 이동한 모습

그 후 아래의 keyboard input이 입력되었을 때 예상했던 움직임대로 manipulator가 정상적으로 움직이는 것을 확인할 수 있었다.

- r: move 1cm in the world z-axis
- f: move -1cm in the world z-axis
- w: move 1cm in the world y-axis
- s: move -1cm in the world y-axis
- d: move 1cm in the world x-axis
- a: move -1cm in the world x-axis
- i: move to the initial position
- e: end the teleoperation

```

alvinjinsung@beaglebone: ~/nfs_client/DesignLab/20170699/EE405_ManipulatorControl/build
File Edit View Search Terminal Help
alvinjinsung@beaglebone:~/nfs_client/DesignLab/20170699/EE405_ManipulatorControl/build$ ./manipulator_control_week2
Succeeded to open the port!
Succeeded to change the baudrate!
Press 'r', 'f', 'w', 's', 'd', or 'a' to move the end-effector. Press 'i' to move to the initial position. Press 'e' to end this mode
Succeeded operating mode to velocity mode.
Succeeded operating mode to velocity mode.
Succeeded operating mode to velocity mode.
Succeeded operating mode to velocity mode.
Succeeded enabling DYNAMIXEL Torque.
Succeeded enabling DYNAMIXEL Torque.
Succeeded enabling DYNAMIXEL Torque.
Succeeded enabling DYNAMIXEL Torque.
Start teleoperation!
Press 'r', 'f', 'w', 's', 'd', or 'a' to move the end-effector. Press 'i' to move to the initial position. Press 'e' to end this mode
Press 'r', 'f', 'w', 's', 'd', or 'a' to move the end-effector. Press 'i' to move to the initial position. Press 'e' to end this mode
Press 'r', 'f', 'w', 's', 'd', or 'a' to move the end-effector. Press 'i' to move to the initial position. Press 'e' to end this mode
Press 'r', 'f', 'w', 's', 'd', or 'a' to move the end-effector. Press 'i' to move to the initial position. Press 'e' to end this mode
Press 'r', 'f', 'w', 's', 'd', or 'a' to move the end-effector. Press 'i' to move to the initial position. Press 'e' to end this mode
Press 'r', 'f', 'w', 's', 'd', or 'a' to move the end-effector. Press 'i' to move to the initial position. Press 'e' to end this mode
Press 'r', 'f', 'w', 's', 'd', or 'a' to move the end-effector. Press 'i' to move to the initial position. Press 'e' to end this mode
Press 'r', 'f', 'w', 's', 'd', or 'a' to move the end-effector. Press 'i' to move to the initial position. Press 'e' to end this mode
Press 'r', 'f', 'w', 's', 'd', or 'a' to move the end-effector. Press 'i' to move to the initial position. Press 'e' to end this mode
Press 'r', 'f', 'w', 's', 'd', or 'a' to move the end-effector. Press 'i' to move to the initial position. Press 'e' to end this mode
Press 'r', 'f', 'w', 's', 'd', or 'a' to move the end-effector. Press 'i' to move to the initial position. Press 'e' to end this mode
Succeeded disabling DYNAMIXEL Torque.
Succeeded disabling DYNAMIXEL Torque.
Succeeded disabling DYNAMIXEL Torque.
Succeeded disabling DYNAMIXEL Torque.
alvinjinsung@beaglebone:~/nfs_client/DesignLab/20170699/EE405_ManipulatorControl/build$

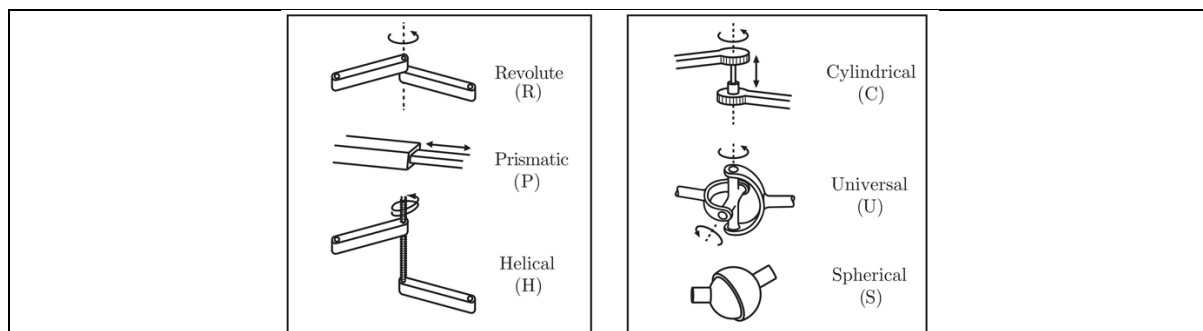
```

그림 36. 프로그램 실행 시 terminal 모습

이렇게 manipulator control을 direct teaching과 keyboard input에 따른 teleoperation 두가지 방법으로 성공적으로 구현하고 실행하였다.

4. Discussion

(1) Compare revolute, prismatic, cylindrical, and spherical joints.



Typical robot joints from the book MODERN ROBOTICS by Kevin M. Lynch and Frank C. Park May 3, 2017

- Revolute Joint

Hinge joint라고도 불리는 revolute joint는 single axis에 대한 rotation motion을 support한다. Joint로 연결된 two bodies의 움직임이 rotation으로 제한되며 DOF는 1이다. Door hinge에 사용되는 joint라 생각하면 되며 robotic arm이나 rotation motion이 필요한 mechanical system에 사용된다.

- Prismatic Joint

Sliding joint라고도 불리는 prismatic joint는 single axis에 대한 linear motion을 support한다. Joint로 연결된 two bodies의 움직임이 fixed axis에 대한 straight line translation으로 제한되며 DOF는 1이다. Telescopic arm이나 linear actuators에 사용된다.

- Cylindrical Joint

Cylindrical joint는 revolute joint와 prismatic joint의 특징을 모두 가지고 있는 joint이다. Joint로 연결된 two bodies의 rotational motion과 linear motion을 모두 support한다. Rotation과 translation 모두 동일한 axis에 대해 이루어지며 DOF는 2이다. Cylindrical joint는 rotation과 translation motion이 모두 필요한 곳에 사용되며, piston-cylinder arrangement나 Mechanisms with rotating shafts 등에 사용된다.

- Spherical Joint

Ball-and-socket joint로도 불리는 spherical joint는 multiple axis에 대한 rotation motion을 support한다. DOF가 3이며 어느 방향이든 움직임이 가능하다. Wide range of motion이 필요한 곳에 사용되며 human body의 hip이나 shoulder joint가 spherical joint에 해당한다. Robot의 end-effector나 robotic hand등에도 활용된다.

(2) In Problem 5A, we utilized polynomial interpolation. Another useful trajectory generation method is the Bezier curve. Explain the advantages and disadvantages of the Bezier curve.

Bezier curve는 computer graphics와 design 분야에서 많이 사용되며 smooth curve와 surface를 represent할 수 있다. Bezier curve의 장단점은 아래와 같다.

Advantages**- simplicity**

: Bezier curve는 이해하고 implement하기 간단하다. Set of control points에 의해 define되며 manipulate하고 adjust하기에 직관적이다.

- Smoothness

: Bezier curve는 smooth한 shape을 만들기에 segment사이 seamless transition을 보장한다.

- Flexibility

: Curve shaping에 있어 flexibility를 준다. Control points의 position과 weight를 조절함으로써 다양한 종류의 curve를 생성할 수 있다.

- Efficiency

: Bezier curve는 compute하고 rendering하는데 computationally efficient하다. Mathematical representation이 간단하기에 computationally lightweight하며 real-time application에 사용하기 적합하다.

Disadvantages

- Lack of Global control

: Bezier curve는 local control curve로 하나의 control point를 modify하는 것이 curve의 일부분만에 영향을 미친다. 이는 전체 curve에 대한 global control이 어려움을 의미한다.

- Limited curve types

: Bezier curve는 smooth, parametric curve를 생성하는데 한정되어 있기에 sharp corner나 complex geometry는 생성이 어렵다.

- Difficulty in curve editing

: Bezier curve는 flexibility를 주지만 세부적인 editing에는 어려움이 있을 수 있다.

(3) Since the DOF of the robot manipulator is four, there is one redundant DOF when controlling the position of the end-effector in Problem 5B. Give an example of the extra task that can be performed by utilizing this redundancy.

Joint space의 DOF가 task space의 dimension보다 크기 때문에 kinematic redundancy가 발생하게 된다. 이는 system에 additional task나 objective을 달성하기 위한 freedom을 줄 수 있다. 이 redundancy를 활용하여 수행할 수 있는 extra task로는 energy consumption을 optimize하거나 Joint torque를 minimize하는 것을 생각할 수 있다. Energy optimization이나 torque minimization에 해당하는 objective function을 추가함으로써 efficient movement를 달성하기 위해 redundant joint를 control할 수 있다. Robot은 joint space 내에서 primary task를 만족하는 다양한 joint configuration을 explore하여 energy consumption이나 torque exertion을 minimize할 수 있다.

(4) Choose your own discussion topic related to Lab 5 and provide an answer to it.

Q. In inverse kinematics, how do you compute \dot{q} when $J(q)$ is not full-rank(i.e. singular) or close to singular?

A. If $J(q)$ is not full rank, \dot{q} will blow up so we add regularization term(Tychonov regularization) and solve the below problem instead.

Remedy

- minimize $\|\dot{q}\|$, while violating $J(q)\dot{q} = \dot{x}$ only minimally
- minimize $\|J(q)\dot{q} - \dot{x}\|^2 + \mu\|\dot{q}\|^2$

By solving this, closed-form solution is given by

$$\dot{q} = (J^T(q)J(q) + \mu I)^{-1}J^T(q)\dot{x}$$

5. References

- [1] EE405 Lab5 Experiment Guide.pdf
- [2] EE405 Lab5 procedure V2.pdf
- [3] lecture_robotics_part2.pdf
- [4] Degrees of Freedom of a Robot,
https://medium.com/@khalil_idrissi/degrees-of-freedom-of-a-robot-c21624060d25
- [5] Bezier curve, https://en.wikipedia.org/wiki/B%C3%A9zier_curve
- [6] Difference between Bezier Curve and B-Spline Curve,
<https://www.tutorialandexample.com/difference-between-bezier-curve-and-b-spline-curve>
- [7] Robots with kinematic redundancy Part 1: Fundamentals, Prof. Alessandro De Luca,
chrome-extension://efaidnbmnnnibpcajpcgiclfefndmkaj/http://www.diag.uniroma1.it/deluca/rob2_en/02_KinematicRedundancy_1.pdf
- [8] Ridge regression, https://en.wikipedia.org/wiki/Ridge_regression