Lab6 System Integration

Objectives

- Learning to integrate all stuff into a system (**SI**)
- Building a **robot manipulation system + teleoperation**
    1. Composed of two computers: PC(user interface), BBB(interacting with env.)
    2. **Vision :** During execution, task scene is obtained by BBB (cam, comm)
    3. **UI, Teleoperation :** The user commands where to pick and place (UDP, CLI interface)
    4. **Teaching, Traj. Generation, Control :** BBB can automatically execute predefined motions (interpolation, UDP, state machine, control)

Problem statement

The content of this lab is to automatically accomplish the predefined sequence of tasks. To do this, use all of what you have learned before (direct teaching, task-space control, teleoperation, etc.) and the concept of finite state machine(FSM).

Problem 6A. Modified direct teaching (week 1)

In lab 5, direct teaching makes the manipulator follow three waypoints repetitively. Instead, in this lab, make the robot follow the predefined two waypoints and maintain its final position.
For this, ‘mode’ is defined. When the mode is 0, a robot follows the two waypoints which are initial joint angles and the first row of user-defined waypoints in ‘waypoints.csv’.
For the mode change, error is defined as ‘currentQ - user_defined_waypoint’. If the error is small, we can consider this task successful, and, in that case, the mode is changed from 0 to 1.
When the mode is set to 1, the waypoints consist of the current position of the robot and the user-defined waypoint on the second line. While implementing this in a similar fashion with lab 5 direct teaching, it is crucial to ensure that the robot **maintains the final waypoint.**

Problem 6B. Teleoperation using UDP communication (week 1)

The second task is teleoperation using UDP communication. In this task, implement the teleoperation by utilizing GPIO control learned in lab2 and task space control learned in lab5 through UDP communication. Specifically, turning on/off the magnet and executing task space control using the keyboard input from the PC shell not BeagleBone shell.

Problem 6C. Finite state machine (week 2)

Utilize the concept of finite state machine to combine (FSM) all the previously defined elements into a single sequence. Implement the FSM defined in ‘Lab procedure.Overview’.

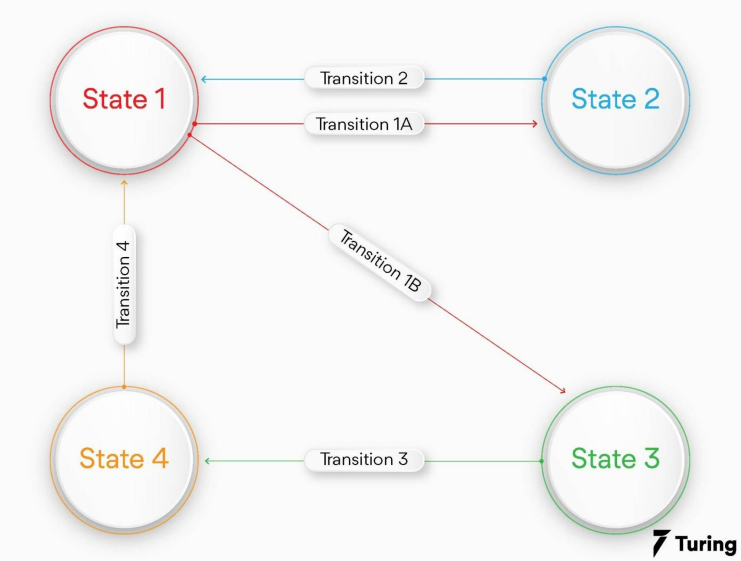
Problem 6D. Choose your own problem (week 3)

Choose your own problem based on the topics of motor control (Lab3), robot manipulator (Lab5) and system integration (Lab6). Note that, your own problem should be one that was not covered in the class and you want to explore more deeply. The report should include the following: **problem definition** (what problem you want to solve), **methodology** (how to solve the problem), and **results**. For the results section, fill it with content that demonstrates how your methodology was effective in solving the problem. TAs does not provide assistance for this.

Backgrounds

1. Finite state machine

A finite-state machine (FSM) is a mathematical model of computation. It is an abstract machine that can be in exactly one of a finite number of ***states*** at any given time. The FSM can change from one state to another in response to some inputs; the change from one state to another is called a ***transition***. An FSM is defined by a list of its states, its initial state, and the inputs that trigger each transition. The diagram of FSM is shown in the following picture.



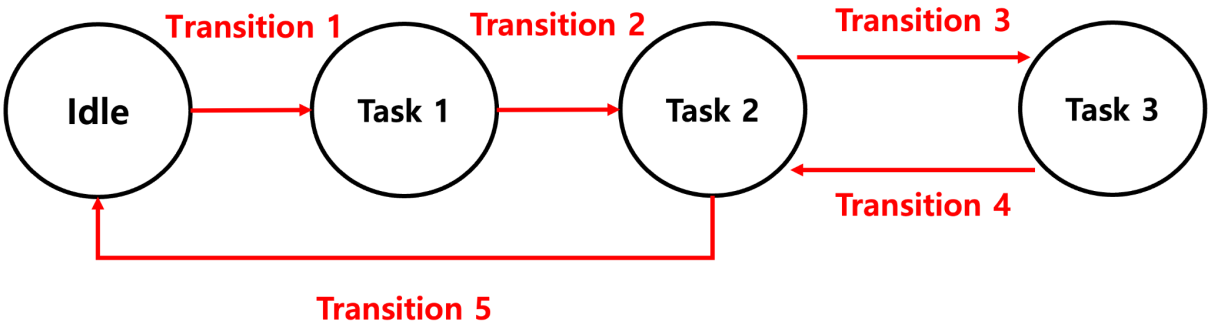
The behavior of state machines can be observed in many devices in modern society that perform a predetermined sequence of actions depending on a sequence of events with which they are presented. Simple examples are: vending machines, which dispense products when the proper combination of coins is deposited; elevators, whose sequence of stops is determined by the floors requested by riders; traffic lights, which change sequence when cars are waiting; combination locks, which require the input of a sequence of numbers in the proper order.

The **advantages of FSM** include the following.

- Finite state machines are flexible
- Easy to move from a significant abstract to a code execution
- Easy determination of reachability of a state

Lab Procedures

I. Overview



The content of this lab is to automatically accomplish the predefined sequence of tasks. So a finite state machine is the proper way to represent a sequence of tasks. Tasks and transition conditions are defined as follows.

States :

- **Idle** : The robot maintains its current position.
- **Task 1** : Using a direct teaching method, move the end-effector of the robot to near the target object.
- **Task 2** : Using the keyboard input from Ubuntu PC, control the end-effector position of the robot and turn on/off the solenoid magnet to pick or place a target object. This process must use visual feedback through a camera.
- **Task 3** : Using a direct teaching method, move the end-effector to the position where the object is to be placed.

Transition :

- **Transition 1** : When the keyboard input is ‘t’.
- **Transition 2** : The error norm between q and last way point is less than a certain threshold.
- **Transition 3** : When the keyboard input is ‘e’. (When object pick is success, the user gives the input ‘e’)
- **Transition 4** : The error norm between q and last way point is less than a certain threshold.
- **Transition 5** : When the keyboard input is ‘x’. (When object place is success, the user gives the input ‘x’)

II. Problems

Problem 6A. Modified direct teaching (week 1)

<Development setup>
The process of connecting the manipulator and the magnet are both well-explained in lab5 and lab2.

<Save way points>
Please refer to the lab5 procedure and save two waypoints to waypoints.csv.

<Generate trajectory>

1. open the CMakeLists.txt in EE405_SystemIntegration directory and uncomment the followings:

```
# ##for Problem1
add_library(EE405_SI_P1 STATIC include/Controller/ArticulatedSystem.cpp
                                include/Common/Kinematics.cpp
                                include/Common/gpio_control.cpp
                                include/FileIO/MatrixFileIO.cpp
                                include/Common/DirectTeaching.cpp
                                )
add_executable(modified_direct_teaching src/modified_direct_teaching.cpp)
target_link_libraries(modified_direct_teaching Threads::Threads EE405_SI_P1 dxl_sbc_cpp)

# ##for Problem2
# add_library(EE405_SI_P2 STATIC include/Controller/ArticulatedSystem.cpp
#                                 include/Common/Kinematics.cpp
#                                 include/Common/gpio_control.cpp
#                                 )

# add_executable(teleoperation_udp src/teleoperation_udp.cpp)
# target_link_libraries(teleoperation_udp Threads::Threads EE405_SI_P2 dxl_sbc_cpp)



# ##for Problem3
# add_library(FSM STATIC include/FiniteStateMachine/fsm.cpp)
# target_link_libraries(FSM EE405_SI_P1)

# add_executable(system_integration src/system_integration.cpp)
# target_link_libraries(system_integration Threads::Threads EE405_SI_P1 FSM dxl_sbc_cpp)

```

2. open the DirectTeaching.cpp file in EE405_SystemIntegration/include/Common/ directory.

3. The input of ‘DirectTeaching::TrajectoryGeneration’ is ‘task_waypoints’. In contrast to before, we will not be using class member variables and instead receive them as function inputs.

   The output of this function should be the trajectory that follows the waypoint which is each row of ‘task_waypoints’. Note that it is crucial to ensure that the robot maintains the final waypoint.

4. open modified_direct_teaching.cpp file in EE405_SystemIntegration/src directory.

5. Check the line 93. Define the ‘task1_waypoints’ and ‘task2_waypoints’ based on the instructions provided in the comments.

6. Check the line 117~140. Write the code, taking into consideration the trajectory to be generated depending on the mode, and the condition of mode change.

7. Build the project using vs code button
    a. Set the compiler as GCC arm-linux-gnueabihf.
    b. Set the compile mode as Release mode. (Do not set Debug mode)

8. Run this code on the BeagleBone.

Problem 6B. Teleoperation using UDP communication (week 1)

In the teleoperation learned in lecture5, keyboard input was given to the BeagleBone terminal. This time, teleoperation must be performed using the keyboard input on the PC local terminal. Keyboard input was transmitted to the BeagleBone through UDP communication.

<Remote Controller PC (talker)>

In the local PC, execute the binary file 'RemoteController_PC'. It is fine to execute what was implemented in lab4 as it is.

<Remote Controller BeagleBone (listener)>

The goal of this section is combining the 'RemoteController_Bone.cpp' implemented in lab4, 'GPIO control' implemented in lab2' and 'manipulator_control_week2'. For this, replace the 'getch' function in ''manipulator_control_week2.cpp'' as a UDP listener.

1. open the CMakeLists.txt in EE405_SystemIntegration directory and uncomment the followings:

```
##for Problem1
add_library(EE405_SI_P1 STATIC include/Controller/ArticulatedSystem.cpp
                               include/Common/Kinematics.cpp
                               include/Common/gpio_control.cpp
                               include/FileIO/MatrixFileIO.cpp
                               include/Common/DirectTeaching.cpp
                               )
add_executable(modified_direct_teaching src/modified_direct_teaching.cpp)
target_link_libraries(modified_direct_teaching Threads::Threads EE405_SI_P1 dxl_sbc_cpp)

##for Problem2
add_library(EE405_SI_P2 STATIC include/Controller/ArticulatedSystem.cpp
                               include/Common/Kinematics.cpp
                               include/Common/gpio_control.cpp
                               )

add_executable(teleoperation_udp src/teleoperation_udp.cpp)
target_link_libraries(teleoperation_udp Threads::Threads EE405_SI_P2 dxl_sbc_cpp)



##for Problem3
# add_library(FSM STATIC include/FiniteStateMachine/fsm.cpp)
# target_link_libraries(FSM EE405_SI_P1)

# add_executable(system_integration src/system_integration.cpp)
# target_link_libraries(system_integration Threads::Threads EE405_SI_P1 FSM dxl_sbc_cpp)

```

2. Open the ‘teleoperation_udp.cpp’ in EE405_SystemIntegration/src directory. Please refer to the comments and complete the GetKeyBoardInput() function in line 58. Most of the code in ‘RemoteController_Bone.cpp’ can be reused.

3. Depending on the keyboard input, corresponding tasks should be executed.
   All the conditions are written in the skeleton code, so you just need to fill in the empty parts.

   3.1. r : move 1cm in the world z-axis
   3.2. f : move -1cm in the world z-axis
   3.3. w: move 1cm in the world y-axis
   3.4. s : move -1cm in the world y-axis
   3.5. d : move 1cm in the world x-axis
   3.6. a : move -1cm in the world x-axis
   3.7. i : move to the initial position
   3.8. e: end the teleoperation
   3.9. q: If the magnet is currently off, turn it on. If it is currently on, turn it off

4. Build the project using vs code button
   4.1. Set the compiler as GCC arm-linux-gnueabihf.
   4.2. Set the compile mode as Release mode. (Do not set Debug mode)

5. Run this code on the BeagleBone.

Problem 6D. Choose your own problem (week 3)

Problem 6C. Finite state machine (week 2)

<Streaming the Webcam>

On the Bone, type below:

$ ./capture -F -c 0 -o | ffmpeg -vcodec mjpeg -i pipe:0 -f mjpeg udp://224.0.0.1:1234

It means, streaming the Webcam screen through udp network with input as capture binary file.

On the PC, type:

$ ffplay -i udp://224.0.0.1:1234

Note that you typed the same network address on the Bone and the PC.

<Save way points>

Please refer to the lab5 procedure and save two waypoints to waypoints.csv. Note that, the first way point is the position where the object is picked up and second one is where the object is placed.

<Remote Controller PC>

In the local PC, execute the binary file 'RemoteController_PC'. It is fine to execute what was implemented in lab4 as it is.

<Finite State Machine>

In c++ code, enum is used to represent the current FSM state for readability of the code.

```
enum FSM_state
{
  FSM_IDLE = 0, // maintain zero position
  FSM_TASK1 = 1, // tracking saved waypoints1
  FSM_TASK2 = 2, // task space control mode
  FSM_TASK3 = 3 // tracking saved waypoints2
};
```

1. open the CMakeLists.txt in EE405_SystemIntegration directory and uncomment the followings:

```
##for Problem1
add_library(EE405_SI_P1 STATIC include/Controller/ArticulatedSystem.cpp
                                include/Common/Kinematics.cpp
                                include/Common/gpio_control.cpp
                                include/FileIO/MatrixFileIO.cpp
                                include/Common/DirectTeaching.cpp
                                )
add_executable(modified_direct_teaching src/modified_direct_teaching.cpp)
target_link_libraries(modified_direct_teaching Threads::Threads EE405_SI_P1 dxl_sbc_cpp)

##for Problem2
add_library(EE405_SI_P2 STATIC include/Controller/ArticulatedSystem.cpp
                                include/Common/Kinematics.cpp
                                include/Common/gpio_control.cpp
                                )

add_executable(teleoperation_udp src/teleoperation_udp.cpp)
target_link_libraries(teleoperation_udp Threads::Threads EE405_SI_P2 dxl_sbc_cpp)



##for Problem3
add_library(FSM STATIC include/FiniteStateMachine/fsm.cpp)
target_link_libraries(FSM EE405_SI_P1)

add_executable(system_integration src/system_integration.cpp)
target_link_libraries(system_integration Threads::Threads EE405_SI_P1 FSM dxl_sbc_cpp)

```

2. Open the system_integration.cpp' in the EE405_SystemIntegration/src directory. All the remaining parts are already filled except for the GetKeyboardInput() function. You can reuse the code implemented in problem 6B.

3. You only need to fill in 'include/FiniteStateMachine/fsm.cpp'. What you need to pay attention to here is the state transition. You need to think carefully about what conditions transition occurs and whether there is an initialization process for the next task.

   What task is performed in a specific state is defined as a function. Output of the functions (task1, task2, task3, idle) is the desired joint position.

   If you reuse the code written in the previous two problems, you can easily fill in the code.

4. Build the project using vs code button
   a. Set the compiler as GCC arm-linux-gnueabihf.
   b. Set the compile mode as Release mode. (Do not set Debug mode)

5. Run this code on the BeagleBone.

6. Show the process of picking up and moving an object while watching the video streaming, as seen in the video (google_drive_link).

Final Report

Each student must write a final report for each experiment. Especially, if the content of the discussion is similar to that of other students, it will result in a deduction or a score of 0. The final report must include the following items:

- Purpose
- Experiment sequence
- Experiment results
- Discussion
- References

The discussion questions for Lab6 experiment are as follows:

1. Are there any examples of finite state machines being used?

2. To define state transitions, it was necessary to determine under what conditions they occur and what initialization process is required for the next task. Discuss what initialization process is needed for each state of FSM defined in I.overview.

3. Define and discuss a new FSM that extends the one defined in the I.overview by adding additional states to perform more diverse tasks.

4. FSM can be used for various robotic applications. Walking hopper can be one of these examples. For the task that the walking hopper walks on, how can states and transitions be defined?
   cf) walking hopper.

5. As mentioned in Problem 6.D, choose your own problem. The report should include problem definition, methodology and result.