ENSC-488
Introduction to Robotics
Spring 2020
Project ROBSIM

In this project, you will simulate a 4-DOF SCARA type (RRPR) manipulator and apply the subject matter of your lectures to the manipulator. SCARA Configuration is shown in Figure 8.5 of your text, the real robot is shown in Figure 1, and a schematic of the real robot is shown in Figure 2 . This simple manipulator is complex enough to demonstrate the main principles of the course while (hopefully!) not bogging you down with too much complexity. Each part of the project builds upon the previous one so that at the end of the project, you will have created an entire library of manipulator software. Two SCARA robots are also there in the lab, and you will get a chance to test some of your functions on the real robot. The project is to be done in groups, with each group consisting of **three** students. Please note that you are free to "adapt" the project, as long as the these adaptations have been discussed with me and gotten approved.

You are also provided with an emulator (it is in fact adapted from the project code of one of the 488 students several years back). The hardware (actual robot) as well as the emulator has the same simple programming interface. The emulator provides a graphic display for your robot, hence, allows you to work on the project without worrying about hardware. The very same code that works for emulator will work on the real robot. You may use hardware/emulator to test your modules before you integrate all modules. A set of header and library files needed for your project will be posted on CANVAS.

1 Component modules

The component modules of ROBSIM are:

1. Basic Matrix Computation Routines:
    - Build the basic routines as described in programming exercises (part 2: 2,3,4) in your text. Please note that the position vector (in your case) will be a 3-tuple (x, y, z) and the rotation matrices will be 3×3, with the additional restriction that all rotations are around Z-axis. You may use public domain source for matrix routines where available. Thoroughly test your routines and make sure that they are bug-free.
2. Forward and Inverse Kinematics:
    - Assign D-H frames to the manipulator. Assign station frame {S} as shown in Figure 1. Assign the origin of Tool frame {T} in the centre of the gripper (note Z_T points downwards). Determine the D-H parameters and the forward kinematic equations. Although numerical values are given to you, it would be better to

derive forward and inverse kinematics with symbolic values and then substitute the numerical values as appropriate parameters in your code.

- Implement the procedures `KIN` and `WHERE` as in programming exercise (part 3: 1,2).
- Implement the procedures `INVKIN` and `SOLVE` as in programming exercise (part 4: 1,2).
- Please note that your routines must respect the joint limits and in case of multiple solutions choose the most appropriate solution (closest to the current configuration). Thoroughly test your routines and make sure that they are bug-free. `You can use the graphic display - using` *DisplayConfiguration()* `command - as a debugging aid!`

3. Trajectory Planner or Collision-free Path Planner:

   - Implement a joint space trajectory planner (similar to Part 7:1,2; but you may choose appropriate type of spline) capable of generating trajectories corresponding to the following command:
     move ARM to {G} via {A}, {B}, and {C} with duration = T seconds; where {A}, {B}, {C} and {G} are TOOL frames specified w.r.t. STATION frame S.
   - You can use either parabolic blends or cubic splines. **You must have velocity continuity** in the trajectory planner (acceleration can be discontinuous; would be nicer if it is continuous but it is not required). As a debugging aid, it will help if you plot the x and y components of the trajectory on an x-y plane with time as an implicit parameter. This "overhead view" will help you visualize the trajectory better since the emulator display does not have the top view of the trajectory.
   - Joint space trajectory planning is the minimum. But if you implement Cartesian space trajectory planning (instead of joint space), that would merit a 5% bonus. Note that it requires jacobian implementation.

Or

4. Implement a path planning module (as and if discussed in class) capable of avoiding collisions of the manipulator with known obstacles in the workspace. For this module, assume that the obstacle cross-section is constant in the $\hat{Z}_0$ direction. For simplicity you may assume that all obstacles have rectangular or circular cross sections. Furthermore, approximate the third link by placing a circle of appropriate radius ${}^2P_{3ORG}$. Essentially, it reduces the manipulator to a 2-link planar robot with a circular tool at the end of second link. The vertices (or radii if you choose to use cylindrical obstacles) of the objects (obstacles) in the environment are to be read from a file. Approximate the output of the collision-avoidance module with straight line segments in joint space. Ideally, this is then fed to the trajectory planner; however, you may directly feed it to

the robot controller with appropriate timing, proportional to the length of the path segment.

5. Dynamic Simulator:

   - Implement a dynamic simulator for the manipulator as in programming exercise (part 6:2). `You should use Maple/Mathematica/Matlab for this, and you must use ‘‘convert’’ function in the applicable software (all of them support it, although the precise name for the function may be different) to directly get c-code for the symbolic expressions, otherwise there is a good chance of introducing bugs if you input the expressions manually in your code`. Again, as in kinematics, it would be better to derive the dynamics equations using symbolic values (at least for mass and length parameters) and then substitute the numerical values. Thoroughly test the simulator by itself since this will act as your simulated robot. One way is to apply a torque to joint 1 only, and see the resulting motion on the emulator.

6. Controller:

   - Implement a trajectory-following control system for the manipulator. As a guideline (you can choose other values), set the servo gains to achieve closed loop stiffness (k_p) of 175.0, 110.0, 40.0 and 20.0 for joints 1 through 4 respectively. Try to achieve approximate critical damping. Put appropriate limits on the maximum torques and forces that joint motors can apply.

7. User interface:

   - The user interface part of your system should have the following capabilities. Please note that this is NOT a project on UI so don’t waste a lot of your time in developing a fancy UI. You can simply read data off from a file. It is the functionality that counts.

     (a) The system should be able to input and output the Tool frame {T} specified in terms of $[x, y, z, \phi]$ with respect to Station frame {S}.

     (b) The user should be able to specify the initial robot configuration (either by specifying the joint angles or by specifying the tool frame) and move the robot to that configuration.

     (c) The system should be able to graphically display the arm motion and workspace obstacles.

     (d) The system should be able to display plots of joint space trajectories and torques. You would use MATLAB/Excel/Gnuplot for this purpose. The joint and torque limits should also be displayed on the same plot so it is visually easy to see if limits are being exceeded.

     (e) The system should be able to detect when joint limits or torque limits are exceeded. The system should inform the user should such an event happen.

Integrate the above modules into a robot programming system. See Figure 3 for a block diagram of your overall system.

2 Project Phases/Demos

The project will be demonstrated in incremental phases.

1. Phase/Demo 1. Implement and demonstrate Forward and inverse kinematics (`KIN/WHERE` and `INVKIN/SOLVE` functions) on both the robot hardware and emulator. User should be able to specify the relevant frames or joint vectors and your system should output the computed frames or joint vectors, as the case is. In case of `WHERE`, the system, given a desired joint vector, should output the corresponding position and orientation of the tool frame with respect to the station frame in terms of a four tuple (x, y, z, ϕ) and also show it graphically. In case of `SOLVE`, the system should, given a desired (x, y, z, ϕ), determine the joint configuration closest to the current joint configuration and move the the robot to that configuration (the tool will be in the desired (x, y, z, ϕ) after the move).

   At the end of this phase, you should be able to demonstrate a pick and place task. General idea is as follows. A simple solid object (a wooden cube, for example) would be the object to be grasped. From the given start gripper frame (say, $\{T_0\}$). You will first move the robot to a given gripper pose, say $\{T_1\}$ that is directly over the object to be grasped. You will then then move the robot down (slowly!) so that the object to be grasped is directly between the gripper jaws (call it $\{T_2\}$), and then close the gripper.

   You will then move back up to $\{T_1\}$, and then go to a location (say $\{T_3\}$) that is directly above where the object is to be placed. It will then move down (again slowly!) to $\{T_4\}$ and open the gripper to release the object from the grasp. Robot will then move back to $\{T_3\}$ and then back to its initial position $\{T_0\}$. **`Demo date: TBA. You will need to sign up for a demo slot on CANVAS.`**

2. Phase/Demo 2. Trajectory/Path Planner: Implement a trajectory/path planner and demonstrate on both, the robot hardware and the emulator.

   For the trajectory planner, user specifies three intermediate Tool frames (w.r.t the station frame {S}), i.e., ${}^{S}_{T_1}T$, ${}^{S}_{T_2}T$, ${}^{S}_{T_3}T$, and a goal frame ${}^{S}_{T_g}T$. Your system should plan and execute a trajectory that starts from rest at the current arm configuration, passes through (or close to) the intermediate frames, and stops at the goal frame.

   You can demonstrate the same pick and place task as in Demo 1, but this time the gripper frames $\{T_i\}$'s will act as via points and the trajectory will be determined by your trajectory planner.

   If you implemented a collision-free path planner with obstacles, it should be demonstrated on both the robot hardware and emulator. User specifies the TOOL frame ${}^{S}_{T}T$

at start and at goal. User also specifies the obstacles in the environment. Your system should plan and execute a collision-free trajectory that starts from rest at the start frame and stops at the goal frame. `Demo: Date TBA. You will need to sign up for a demo slot on CANVAS.`

3. Phase 3a. Dynamic simulator, implemented on emulator only. You need to show the behaviour of your robot dynamic simulator under predefined (e.g. constant) torques/forces.

4. Phase 3b. Fully-functioning ROBSIM system with controller, implemented on emulator only. Besides the above modules/functions, you need to show the behaviour of your controller. When you execute the trajectory, record the error and torque history for each joint for the entire motion; and show the plots (after the motion).

5. Final Project Demo 3: Integrates Phases 1-3. As for trajectory planner, the user specifies the start frame, the goal frame, and three intermediate frames (w.r.t the station frame {S}), ${}^S_T T$. Your system should plan and execute a trajectory that starts from rest at the start frame, passes through (or close to) the intermediate frames, and stops at the goal frame. As the trajectory is executed, record the error and torque history for each joint for the entire motion; and show the plots (after the motion). `Demo: Date TBA. You will need to sign up for a demo slot on CANVAS.` A project report (one per group) about 15 pages of text plus figures and plots) should be submitted at the time of project demonstration. See Section 4 for details.

3 Details

Robot parameters

1. For the project, the manipulator is mounted so that all joint axes are vertical. Assume that $\hat{Z}_0$, $\hat{Z}_1$ and $\hat{Z}_2$ point vertically upwards, and $\hat{Z}_3$ and $\hat{Z}_4$ point vertically downwards.

2. The kinematic parameters are shown in Figure 2. Joint 1 has a joint limit of $\pm 150°$, joint 2 has a joint limit of $\pm 100°$, joint 3 can move between [-200mm, -100mm] (the loewer limit is hardware limit, but the upper limit is artificially imposed so that the gripper does not hit the workbench), and joint 4 has a joint limit of $\pm 160°$. In your simulator, you should assume that the robot has a gripper mounted at the end.

3. The dynamic parameters are as follows. Note that this applies to the simulator only, and not the real robot. m_1, m_2, m_3, m_4 are mass of the four links respectively; $m_1 = 1.7kg$, $m_2 = 1.0kg$, $m_3 = 1.7kg$, and $m_4 = 1.0kg$. For all four links, assume that the mass is all concentrated into a point mass, and the respective locations of centres of mass are shown in Figure 2. Assume some viscous friction at each joint. Of course, gravity acts vertically downward. It would be a a good idea to derive all equations

using a parameter g and make g user defined. This way you could test your robot's behaviour in space (zero gravity)!

Hardware/emulator APIs. As shown in Figure 3, you will be using different functions to "drive" the hardware/emulator in different demos (i.e., (A), (B) and (C)). Do make sure you use the appropriate move/display command.

1. *MoveToConfiguration(conf)* for (A), to move the robot to a given configuration. It will use maximum velocity and acceleration for the motion.This function is for both hardware and emulator. `You must use this function for driving the robot in Demo 1`.

2. *MoveWithConfVelAcc(conf, vel, acc)* for (B), to move the robot such that it moves with desired velocity (vel) and acceleration (acc). This function is for both hardware and emulator. `You must use this function for driving the robot in Demo 2`. Be aware that in current implementation of this function, due to the limitation of hardware programming interface provided by the manufacturer, only velocity (vel) argument is used. Because of this simplification, there may be a slight drift at the end of the trajectory. But in your own controller, you should take all configuration, velocity and acceleration into consideration.

3. *DisplayConfiguration(conf)* for (C), simply display the robot under the given configuration. This function is for emulator only. `You must use this function for driving the robot in Demo 3`.

4. *Grasp(close)*, to close (with close=true) or open (with close=false) the gripper. This function is for both hardware and emulator.

5. Some other functions are provided for general purpose, and they are all for both hardware and emulator.

   - *GetConfiguration(conf)* to retrieve current robot confguation.
   - *StopRobot()* to stop the robot.
   - *ResetRobot()* to reset the robot after calling StopRobot(). This enables the robot to respond to move commands again.

Please note that you must use the command corresponding to the letter at the output of the module as shown in Figure 3 to move the robot - real robot or the virtualrobot (emulator). In above functions, the arguments used for configuration/velocity/acceleration are double arrays of size 4, corresponding to 4 joints. Please read the provided header file *ensc-488.h* for more details. Pay attention to units used for arguments. For revolute joints, they accept *degree* for joint value, *degree/s* for velocity, and $degree/s^2$ for acceleration; for prismatic joints, they accept *mm*, *mm/s* and mm/s^2 respectively.

N.B. Some programming and implementation advice (mainly for those who may not have much programming experience) follows. The project involves non-trivial amount of coding. Therefore, follow structured programming approach. Adopt a modular structure for your code with each functional unit represented by a separate module. Test each module thoroughly before integrating it with the rest of the system. Parameters such as link lengths, joint limits, default torque maximums, etc. should reside in one place and should be easily accessible and modifiable. It may also make sense to use global variables for certain parameters. Believe me, following a disciplined approach to coding will save you considerable headache and many sleepless nights in debugging!

4 Project Report

A project report (one per group) not exceeding 15 pages of text (plots are extra) is due at the time of project demonstration. It should adhere to the following guidelines:

- A succinct description of each module. There is no point repeating material from text. But you must include all final results in each module (for example: frame attachment, D-H parameters, expressions for inverse kinematics, dynamic equations, control laws, and relevant block diagrams with sampling times).
- Include plots of trajectories, velocities, accelerations, torques for a sample trajectory. Make sure the plots are appropriately labeled.
- Any anomalous behaviour that you observed in your simulations/experiments and an explanation for it if you have one.
- The report should clearly state the role of each member of the team and the work carried out by the member (about a paragraph for each member). Also state the methods used by the team for communication between members for various aspects of the project - for example for code development, code sharing, etc. Each member must have a non-trivial contribution code development.

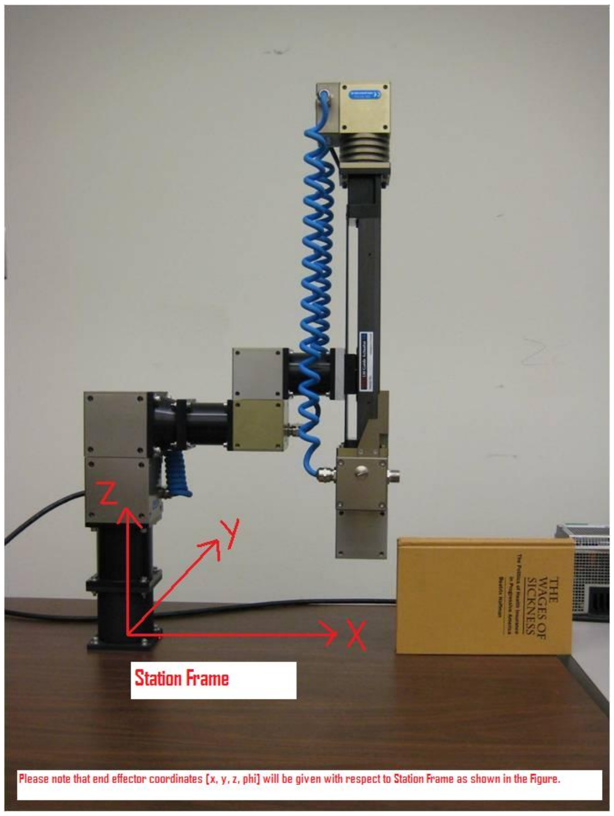
Z
Y
X
Station Frame
Please note that end effector coordinates [x, y, z, phi] will be given with respect to Station Frame as shown in the Figure.

Figure 1: Assigned Station Frame. The robot is shown in joint configuration (0°, 0°, -200mm, 0°).

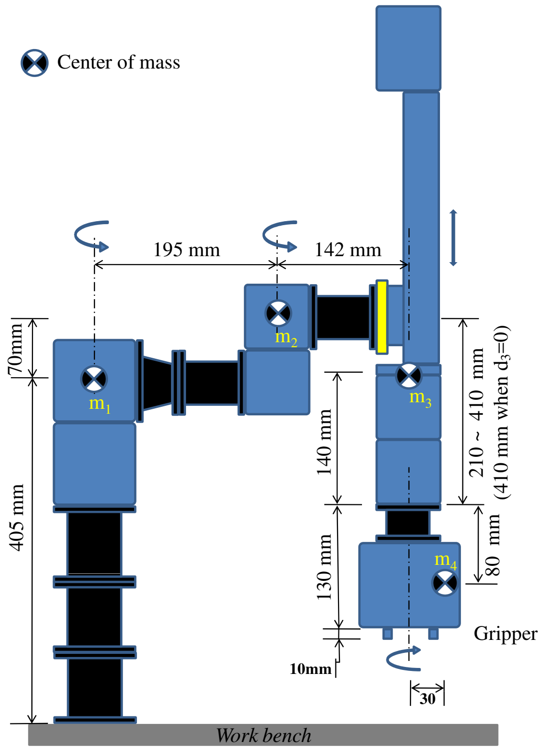

Figure 2: Schematic of SCARA type robot system assembled in 488 Lab.

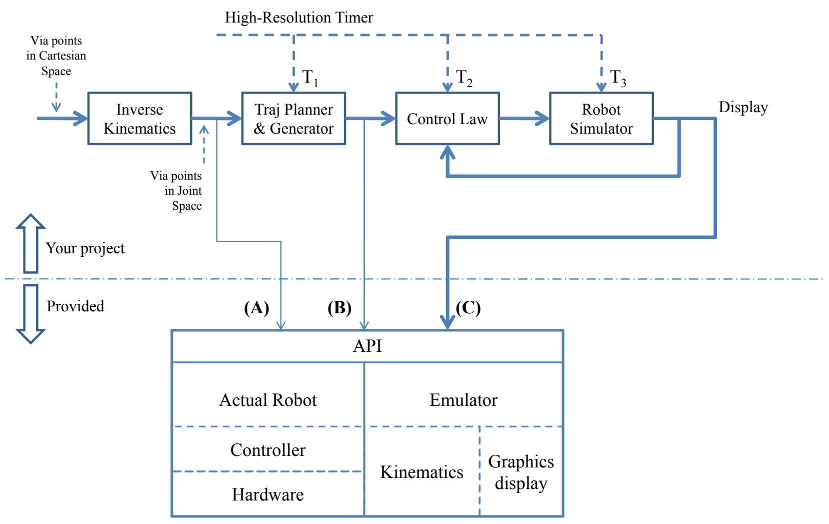

Figure 3: Block diagram of your robot programming system: ROBSIM