QUANSER

INNOVATE. EDUCATE.

STUDENT WORKBOOK

Omni Bundle Robotics Experiment

Developed by:
Jacob Apkarian, Ph.D., Quanser
Paul Karam, B.A.Sc,. Quanser
Derry Crymble, M.A.Sc., Quanser
Amin Abdossalami, M.A.Sc., Quanser
Roopa Samra, M.A.Sc., Quanser

Omni Bundle is powered by:

MATLAB® SIMULINK®

CAPTIVATE. MOTIVATE. GRADUATE.

Quanser Inc.
119 Spy Court
Markham, Ontario
L3R 5H6
Canada
info@quanser.com
Phone: 1-905-940-3575
Fax: 1-905-940-3576

Printed in Markham, Ontario.

For more information on the solutions Quanser Inc. offers, please visit the web site at:
http://www.quanser.com

CONTENTS

1 INTRODUCTION

The purpose of this curriculum is to teach senior undergraduate students various aspects of robotics and haptics. Students will be exposed to concepts such as forward and inverse kinematic modeling of the robot, control design, trajectory planning, and force rendering. The later exercises will take these concepts and apply them to create haptics demonstrations. The robot used in all of the experiments will be the 3D Systems Geomagic Touch (formerly Sensable Technologies PHANTOM® Omni), shown in Figure 1.1, henceforth referred to as the Omni.

Figure 1.1: Omni Bundle

This curriculum is designed to integrate aspects of both robotics and controls to demonstrate how they can be used to create haptics applications.

All of the exercises use Matlab® and QUARC®. Each exercise shows how the QUARC® models should look when the experiment is finished. Students are encouraged to use these figures as a guide.

Topics Covered

- Forward kinematics
- Inverse kinematics
- Position control
- Trajectory generation
- Jacobian and force rendering
- Haptic force generation

1.1 Omni

1.1.1 Joints

The Omni is a robot with six revolute joints. However, only three of the joints are actuated. The three actuated joints are J_1, J_2 and J_3 shown in Figure 1.2. The non-actuated joints are the three wrist joints, J_4, J_5 and J_6, shown in Figure 1.3. The end-effector, sometimes referred to as the tool position, is the tip of the stylus shown in Figure 1.3.

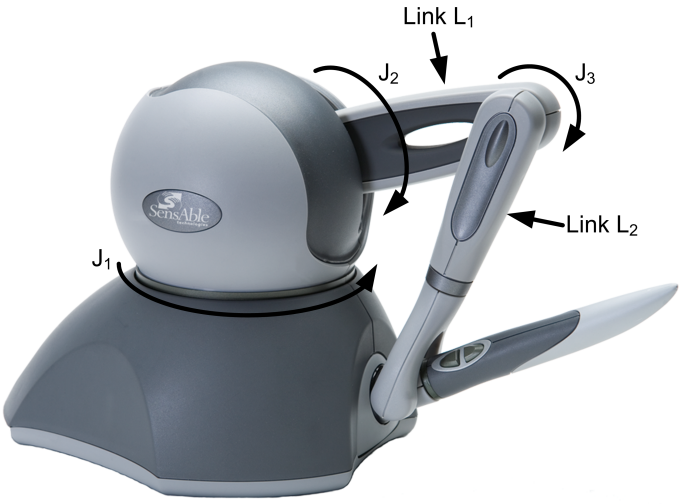


Figure 1.2: The three actuated joints on the Omni[1]

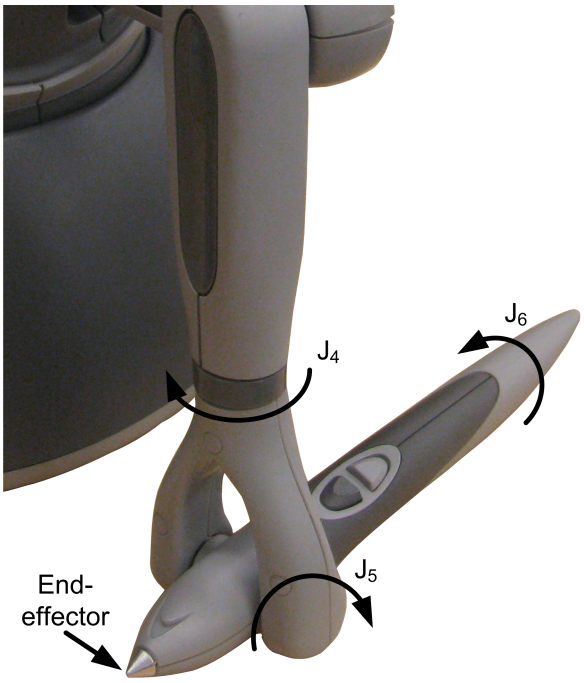


Figure 1.3: Non-actuated joints and the end-effector[1]

[1] Sensable Phantom Omni shown, but joint configuration is consistent with the Geomagic Touch.

1.1.2 Custom Configuration

In many exercises the stylus of the Omni is fixed to Link L_2, as shown in Figure 1.4. In this position joints J_4, J_5 and J_6 remain fixed. This is to enable students to perform kinematic analysis assuming only the first three joints exist.

Figure 1.4: Custom configuration of the stylus

1.1.3 Physical Properties

The link lengths of the Omni are provided in Table 1.1.

Link	Length (m)
L_1	0.132
L_2	0.132

Table 1.1: Physical Parameters

1.1.4 Positions

Various exercises throughout the curriculum require the Omni to be positioned at predetermined points on the base-board. These positions are shown in Figure 1.5.

Figure 1.5: Omni baseboard positions

2 PREFACE

The laboratory sections in this manual is generally organized into two sections.

Pre-Laboratory Assignment section is not meant to be a comprehensive list of questions to examine understanding of the entire background material. Rather, it provides targeted questions or programming assignments for preliminary calculations that need to be done prior to the lab experiments. All or some of the questions in the Pre-Lab section can be assigned to the students as homework. One possibility is to assign them as a homework one week prior to the actual lab session and ask the students to bring their assignment to the lab session.

In-Laboratory Experiment section provides step-by-step instructions to conduct the lab experiments and to record the collected data.

3 FORWARD KINEMATICS

The objective of this laboratory is to develop a forward kinematics analysis of the Omni. The purpose of this analysis is to determine the position and orientation of the end-effector given the joint positions. At the end of this laboratory, you will have created a QUARC® model that will read the joint positions in real time and output the position and orientation of the end-effector.

Topics Covered

- Performing a forward kinematic analysis using the standard Denavit-Hartenberg (D-H) representation.
- Creating a model using QUARC and Simulink blocks to read the joint positions of the Omni.
- Creating a MATLAB function block to implement forward kinematic analysis into the model.
- Determining the reachable space of the device in task space and joint space.

3.1 Pre-Laboratory Assignments

3.1.1 Standard D-H Parameters

Question 3.1
Use the coordinate frames given in Figure 3.1 to derive the standard D-H parameters. For an overview of the standard D-H parameters, refer to Section A. The global frame is frame 0 and the end frame is frame 3.

Note:

- The position shown in Figure 3.1 is not achievable by the robot. In this position, all joint angles are zero. To help visualize the orientation, the position of each joint shown in Figure 3.2 is given in Table 3.1. Notice that in Figure 3.2 the workspace has been removed as the position shown is not achievable otherwise.
- Frame 3 is not the end-effector frame. Later on, you will be provided the position and orientation of the end-effector frame with respect to frame 3.

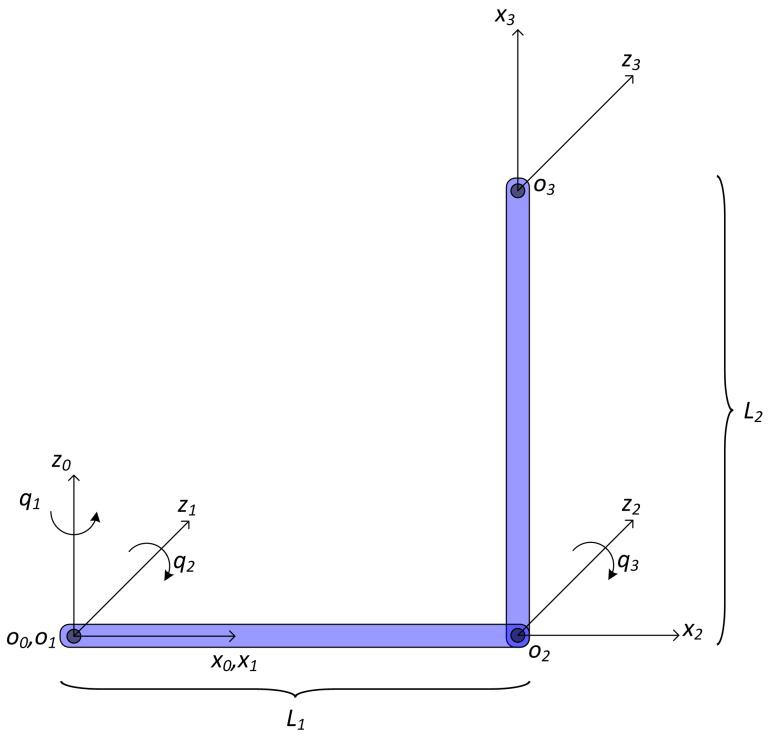


Figure 3.1: Coordinate frames to get the D-H parameters

Figure 3.2: Specific joint orientations

Joint	Position (rad)
q_1	0
q_2	0
q_3	π

Table 3.1: Figure 3.2 joint positions

θ (rad)	d (m)	a (m)	α (rad)

Table 3.2: D-H parameters

3.1.2 MATLAB Functions

1. Write a MATLAB function called **DH.m** as follows:
   - **Inputs** (see Section A for a description of the following):
     - *theta*: Angle measured in radians
     - *d*: Distance measured in meters
     - *a*: Length measured in meters
     - *alpha*: Twist measured in radians
   - **Outputs**

- *A*: The homogeneous matrix

- This function should accept a vector containing the above four inputs and output the homogeneous matrix *A* based on those four quantities.

Question 3.2
Show how this homogeneous matrix is calculated from the parameters θ, *d*, *a* and α.

2. Write a MATLAB function called **ForwardKinematics.m** as follows:

   - **Inputs**
     - *q*: A 1×3 vector that accepts the joint positions of J_1, J_2, and J_3 as q_1, q_2, and q_3 in radians
     - *tool_offset*: A 4×4 matrix that accepts the transformation matrix of the end-effector. The transformation matrix specifies the rotation and translation of the end-effector with respect to frame 3
   - **Outputs**
     - *pos*: A 1×3 vector that outputs the position of the end-effector (x, y and z coordinates) in the global frame in meters
     - *rot*: A 3×3 rotation matrix of the end-effector with respect to the global coordinate frame

   (a) Call **DH.m** three times with the parameters from Question 3.1 to get the homogeneous matrices, A_1, A_2 and A_3. Use L_1 and L_2 as given in Table 1.1.

   (b) Use the homogeneous matrices to find the transformation

$$T_0^3 = A_1 \times A_2 \times A_3$$

   (c) Use the transformation matrix, T_0^3, to transform the tool offset matrix (given with respect to frame 3 coordinates) to global coordinates. Use the following tool offset matrix.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

   (d) From this transformed matrix, extract the rotation matrix and the translation of the end-effector with respect to the global frame. These will be your outputs *rot* and *pos* respectively.

Question 3.3
Call this function with the values of q given in Table 3.7.

Joint Position (rad)	Position (m)	Rotation matrix
$q_1 = 0.8$ $q_2 = -0.3$ $q_3 = 3.3$		$\begin{bmatrix} & \end{bmatrix}$
$q_1 = -0.92$ $q_2 = -0.25$ $q_3 = 3.14$		$\begin{bmatrix} & \end{bmatrix}$
$q_1 = -0.65$ $q_2 = -1.00$ $q_3 = 3.78$		$\begin{bmatrix} & \end{bmatrix}$
$q_1 = 0.57$ $q_2 = -1.7$ $q_3 = 2.8$		$\begin{bmatrix} & \end{bmatrix}$

Table 3.3: Forward Kinematics

3.1.3 Laboratory Files to Submit

The following files should be submitted for evaluation:

- DH.m
- ForwardKinematics.m

3.2 In-Laboratory Experiment

In this laboratory, you will implement your **DH.m** and the **ForwardKinematics.m** files in QUARC to implement a model that reads the joint angles from the Omni and gives the position and rotation of the end-effector with respect to the global frame. You will be given a QUARC block that continuously reads the encoder values from the Omni and gives the joint angles in radians.

Note: Before beginning, ensure that the stylus of the Omni is securely fastened to link 2, as shown in Figure 1.4.

3.2.1 Reading Joint Angles

To begin the forward kinematics analysis, you first need to read the position of each of the first three joints of the Omni. These are joints q_1, q_2 and q_3.

1. In MATLAB, open the Forward Kinematics Simulink model:
   ..\Student Lab Files\02 Forward Kinematics **Forward_Kinematics.mdl**
2. The Forward Kinematics model is preconfigured to read the Omni joint angles, as shown in Figure 3.3

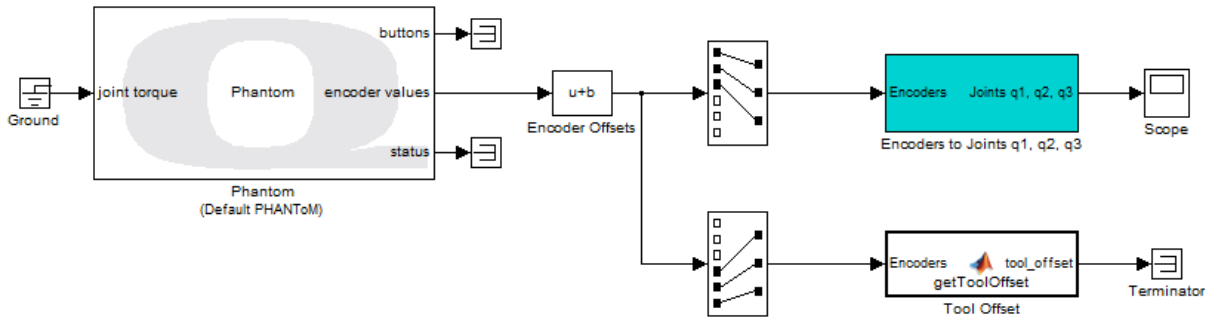


Figure 3.3: Forward Kinematics model

3. Build the simulation by going to the **QUARC** menu, and clicking on **Build**. When the MATLAB Command Window shows that the model has been downloaded to the target, run the simulation by opening the **QUARC** menu and clicking on **Start**.
4. Open the **Scope**. Move around the end-effector of the Omni. The value of joint angles (in radians) will be shown on the scope.

   Question 3.4
   Try to move one joint at a time in the positive direction as defined by the D-H frames earlier. Does the corresponding signal on the scope also move in the positive direction? Record the relative direction (+/-) of the joints compared to the direction defined in the D-H frames.

Joint	Direction
q_1	
q_2	
q_3	

Table 3.4: Joint Directions

5. Insert a **Gain** block between the **Encoders to Joints q1, q2, q3** block and the **Scope**. Enter the gains you found in Question 3.4 as a 1×3 vector. The joint angle calculation should look as shown in Figure 3.4. Restart your model. The joints should now move according to positive orientation defined by D-H frames.

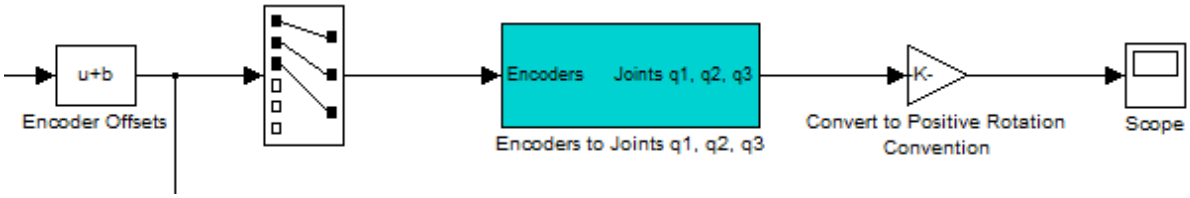


Figure 3.4: Orient joint angles

Question 3.5
Place the end-effector on position 2 of the baseboard. According to the D-H parameters defined earlier, the angles of the end-effector at this point should be given by the values in Table 3.5. Determine the biases (if any) that should be added to each joint in order to get these values.

Joint	Angles (rad)
q_1	0
q_2	-0.3
q_3	3.4

Table 3.5: Position 2 joint angles

Joint	Bias (rad)
q_1	
q_2	
q_3	

Table 3.6: Joint biases

6. Insert a **Bias** block between the **Gain** block and the **Scope**. Enter the biases you found in Question 3.5 as a 1×3 vector. Your angle calculation should look as shown in Figure 3.5. Restart you model.

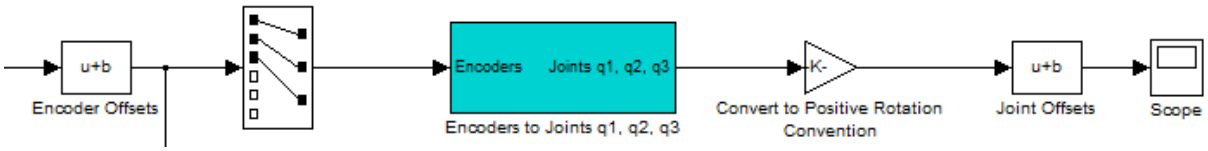


Figure 3.5: Joint orientations

7. Place the end-effector on position 2 of the baseboard. The positions read from the scope for each angle should match the ones shown in Table 3.5. If your positions do not match, either your gain or offset blocks are incorrect or your Omni is not calibrated.

3.2.2 Implementing Forward Kinematics

1. Create a **MATLAB Function** block to implement **DH.m** and **ForwardKinematics.m**:

   (a) In your Simulink model, insert a **MATLAB Function** block.

(b) Copy and paste the code from your **ForwardKinematics.m** file into this block.

(c) Directly underneath, copy and paste the code from your **DH.m** file.

2. Attach a display to all of the outputs of this block.
3. Attach input *q* to the output of the bias block. Attach a display to the input *q*.
4. Attach the **Tool Offset** block to the *tool_offset* input. Your complete model should resemble Figure 3.6.

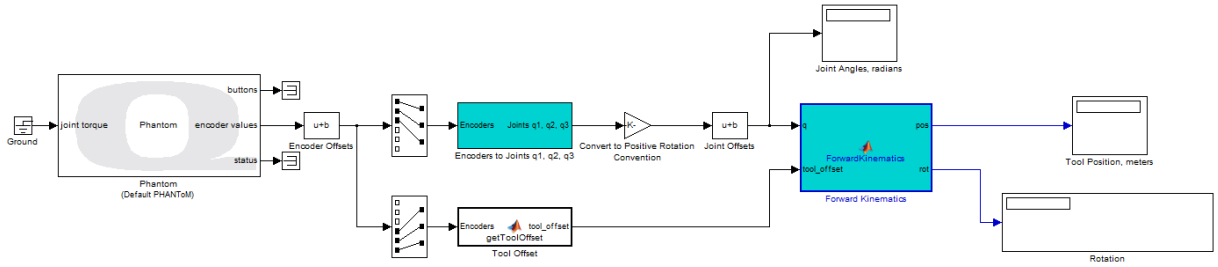


Figure 3.6: Forward kinematics model

5. Build and run your model. As you move the end-effector of the Omni, the displays will show the values of the corresponding signals.

Question 3.6
Move the end-effector to positions 2, 5, 6, and 7 on the board. Record the Cartesian positions and rotations of the end-effector.

Joint Position (rad)	**Position (m)**	**Rotation matrix**
Position 2		[ ]
Position 5		[ ]
Position 6		[ ]
Position 7		[ ]

Table 3.7: Forward Kinematics

3.2.3 Determining the Reachable Space of the Omni

In many applications, it is important to know the reachable space of the robot to ensure that the robot is not assigned any tasks that are outside of its reach. In this section, you will experimentally determine the workspace of the robot in both task space and joint space.

1. Run **Forward_Kinematics.mdl**
2. Position the end-effector at its maximum and minimum x-coordinates in the global frame. Observe the position given by the output signal, *pos*. This gives the reachable space of the robot in the x-direction.

   Question 3.7
   Record the maximum and minimum x-axis positions. Repeat the process to determine the reachable space in the y and z directions.

Joint	Minimum (m)	Maximum (m)
x		
y		
z		

Table 3.8: Reachability in task space

3. Now determine the reachable space in joint space. Move each joint to its maximum and minimum positions and read the signal values from the input signal *q*.

   Question 3.8
   Record the values for the three joints. Note that for joint 3, the reachable space is dependent on the position of joint 2. This is due to the physical limitations provided by the form factor of the device. For joint 3, record the minimum and maximum positions when q_2 is approximately -1.78 rad.

Joint	Minimum (rad)	Maximum (rad)
q_1		
q_2		
q_3 ($q_2 \approx$ -1.82 rad)		

Table 3.9: Reachability in joint space

3.2.4 Laboratory Files to Submit

The following files should be submitted for evaluation:

- ForwardKinematics.mdl

4 INVERSE KINEMATICS

The objective of this laboratory is to develop an inverse kinematics analysis of the Omni. The purpose of the analysis is to determine the position of each joint given the position and orientation of the end-effector. In general, unlike the forward kinematics problem, the inverse kinematics problem may not always have a solution and when a solution exists it may not be unique.

Topics Covered

- Developing the inverse kinematics equations using a geometric approach
- Providing conditions for the equations such that in the reachable space of the Omni, a unique solution is always achieved.

4.1 Pre-Laboratory Assignments

4.1.1 Inverse Kinematics Equations

1. You will use Figure 4.1 to geometrically derive the inverse kinematics equations. Notice that this figure does not show the end-effector. In other words, the inverse kinematics analysis will be performed by assuming that you have the position of frame 3 in global coordinates and based on it you want to find the position of each joint. Frame 3 is indicated by point F_3 on the diagram.
   Note: In Figure 4.1, triangle **-zdk** is in the vertical (x-z) plane, and triangle **YXd** is in the horizontal (x-y) plane.

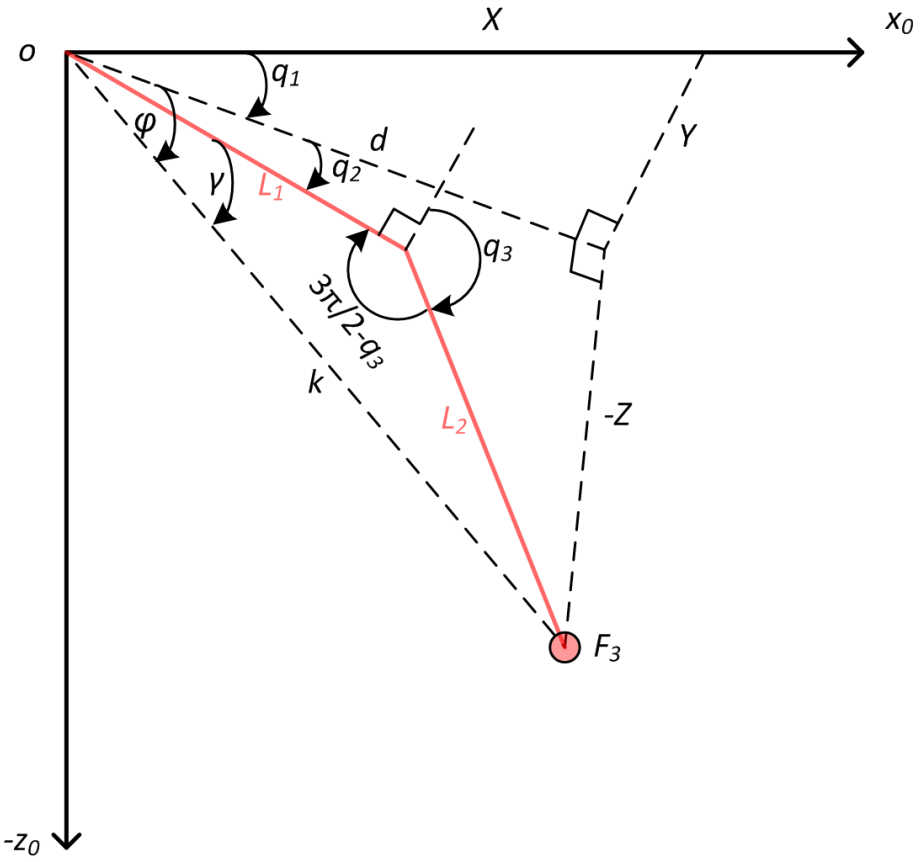


Figure 4.1: Geometry to derive inverse kinematics

Question 4.1
Using geometry, derive equations for the joint angles q_1, q_2 and q_3 in terms of the Cartesian position (x,y,z) of frame 3 with respect to the frame 0. Note that frame 3 is not the end-effector frame. Refer to Section 3.1, to see the location of frame 3.
Use the following tips:

(a) Ensure that the equation for each joint will give a unique solution that lies in the reachable joint space of the device. Use the reachable joint space of the robot you determined in Section 3.2.3. If necessary, use "if conditions" to limit the solutions in the reachable joint space.
(b) Use the reachable joint space of each link to determine the quadrant(s) that each joint operates in. Ensure that your inverse kinematic equations provide solutions in the respective quadrant(s).
(c) Use the Sine Law and the Cosine Law to find q_2 and q_3. q_2 is positive (counter-clockwise) when z $\leq$ 0 and negative otherwise. Use this to determine the sign of q_2.
(d) Due to physical limitations of the device, q_3 is always positive.

2. Note that in general, we are not given the position of frame 3. Rather we are given the position of the end-effector (in global coordinates). However, if we know the position of the end-effector with respect to frame 3, the position of frame 3 (in global coordinates) is trivial to obtain.

   Question 4.2
   Given the *tool_offset* matrix, write the steps needed to get the position of the frame 3 with respect to the global frame. Remember that the end-effector frame is always aligned with frame 3. Also note that the tool offset matrix is given with respect to frame 3, not the global frame.

4.1.2 MATLAB Function Block

1. Open a new Simulink model. Save this model as: **PreLab_Inverse_Kinematics.mdl**.
2. Create a new MATLAB function block called **Inverse Kinematics** as follows:
   - **Inputs**
     - *pos*: A 1×3 vector for the position of the end-effector with respect to the global frame in meters
     - *rot*: A 3×3 matrix for the rotation of the end-effector with respect to the global frame
     - *tool_offset*: A 4×4 matrix for the position and rotation of the end-effector with respect to frame 3.
   - **Outputs**
     - q_1: The position of joint 1 in radians.
     - q_2: The position of joint 2 in radians.
     - q_3: The position of joint 3 in radians.
   - Use L_1 and L_2 given in Table 1.1.
   - Implement your process from Section 4.1.1 to get the position of each joint.
3. Your block should resemble Figure 4.2.

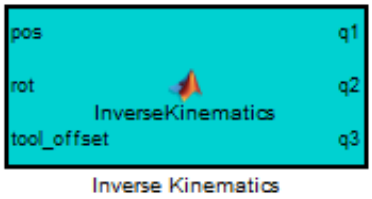


Figure 4.2: Inverse kinematics block

4. Copy and paste the forward kinematics model from Section 3.2.2.
5. Insert a **Constant** block and attach it to the input *q* of the forward kinematics block as shown in Figure 4.3.
6. Attach your inverse kinematics block to the output of the forward kinematics block. Use an identity matrix for *tool_offset*.
7. Attach displays as shown in Figure 4.3. The **Mux** indicated in Figure 4.3 is available in the **Simulink - Signal Routing** library. It converts its inputs into a vector. Double click on the **Mux** to change the number of inputs from two to three.

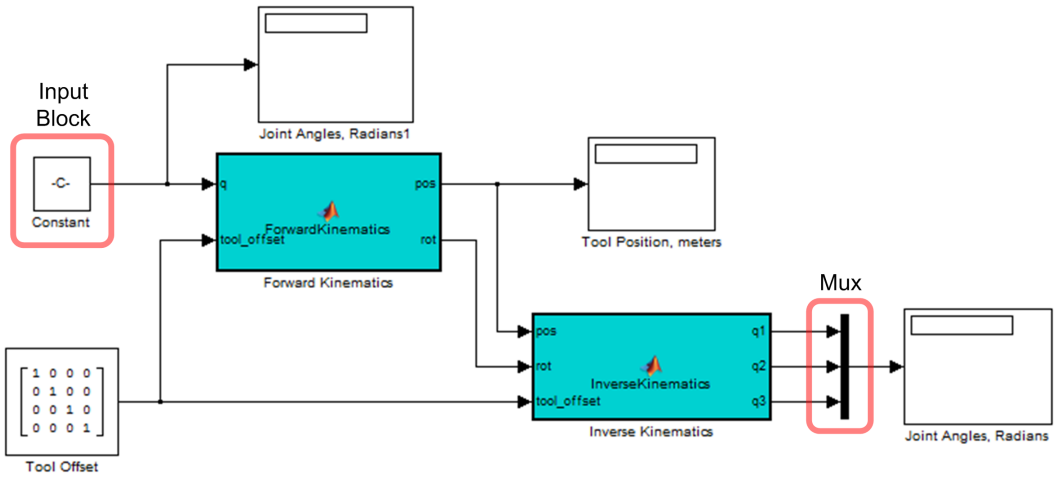


Figure 4.3: Inverse kinematics model

8. As an input to the forward kinematics block, input the joint angles as shown in Table 4.1 (one row at a time). Enter these values as a 1×3 vector in the **Input** block indicated in Figure 4.3.

9. Run your model. Check that the joint angles output from the inverse kinematics block match the input to the forward kinematics block. In other words, both displays should output the same values.

Position	θ_1 (rad)	θ_2 (rad)	θ_3 (rad)
Position 1	0.8	-0.3	3.3
Position 2	-0.92	-0.25	3.14
Position 3	-0.65	-1.00	3.78
Position 4	0.57	-1.7	2.8

Table 4.1: Inverse kinematics inputs

4.1.3 Laboratory Files to Submit

The following files should be submitted for evaluation:

- PreLab_Inverse_Kinematics.mdl

4.2 In-Laboratory Experiment

4.2.1 Implementing Inverse Kinematics

1. Copy and paste **Forward_Kinematics.mdl** from Section 3.2.2 into a new model called **Inverse_Kinematics.mdl**. Attach your inverse kinematics MATLAB Function block from Section 4.1.2 to the forward kinematics block. Your model should resemble Figure 4.4.

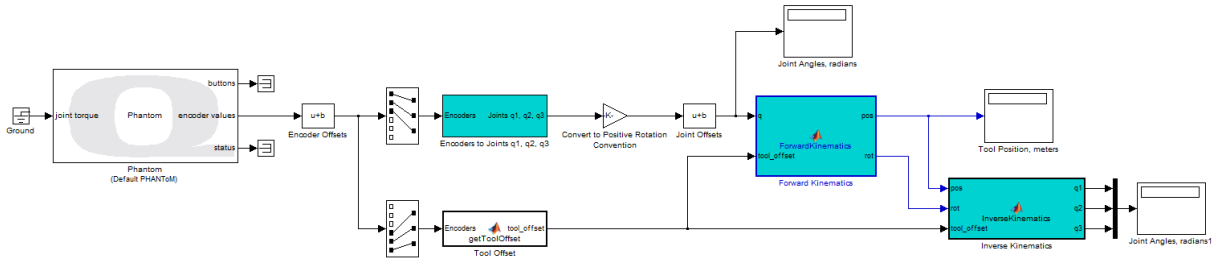


Figure 4.4: Inverse kinematics and forward kinematics

2. Check that the **Simulation mode** is set to *External*.
3. Build and run the model.

Question 4.3
Move the robot to positions 1, 3, 7, and 9 on the board and measure the joint angles and actual positions.

Position	Joint Angles (rad)	End-effector Position (m)
Position 2		
Position 5		
Position 6		
Position 7		

Table 4.2: Inverse Kinematics Results

5 POSITION CONTROL

The objective of this laboratory is to design and study the performance of position controllers for each joint. Proportional-Derivative controllers, or PD controllers, are commonly used for set-point tracking. The proportional and the derivative gains can be tuned appropriately to meet design specifications. You will design PD controllers for each joint to meet specified design specifications. You will also explore the benefits of adding an integrator into the controller to improve tracking. This type of control design is referred to as PID control.

Topics Covered

- Designing PD controllers for given specifications for each joint.
- Learning the effects of changing each gain on the overall performance.
- Learning the effects of changing apparent inertia on the performance of a controller.
- Determining the maximum velocity of each joint at given configurations.
- Studying performance using a PID controller.

5.1 Pre-Laboratory Assignments

5.1.1 PD Control: Transfer Function

In this section, you will design a PD controller for each joint such that the response will meet some design specifications. The closed-loop block diagram applicable to each joint is shown in Figure 5.1. K_p and K_d are the proportional and derivative gains respectively. *J* and *B* are the joint inertia and friction respectively. θ_d is the desired position of the joint and θ_m is the actual position. This is an approximate model of the joints where many physical effects are ignored to achieve a more simplified model. It should be noted that this particular implementation of PD control applies the derivative gain to the feedback position, as opposed to the conventional approach where the derivative gain is applied to the position error. This approach is often referred to as PV control, or set point gain, and is used in this case to improve performance and better approximate a second-order system.

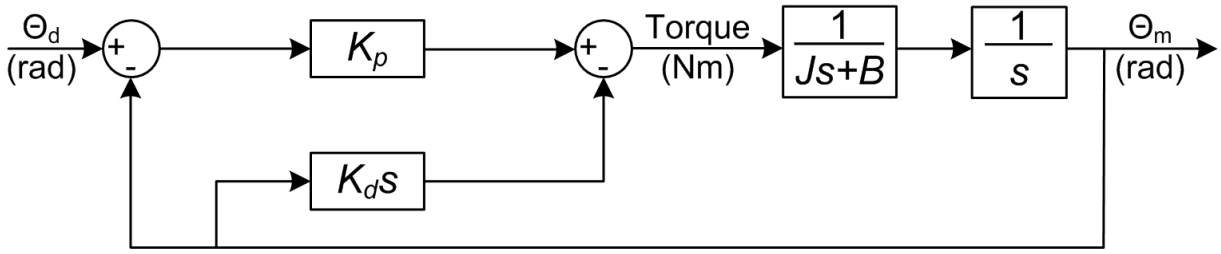


Figure 5.1: Closed-loop PD control block diagram

Question 5.1
Derive the closed-loop transfer function, $\frac{\theta_m}{\theta_d}$, for the block diagram in Figure 5.1.

Question 5.2
The model of a second order transfer function is given by:

$$\frac{\theta_m}{\theta_d} = \frac{\omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2} \tag{5.1}$$

Use this model and your solution to Question 5.1 above to derive equations for the proportional and derivative gains, K_p and K_d.

5.1.2 Effect of PD Gains

Question 5.3
A PD controller can be designed to achieve various specifications depending on the values of the gains K_p and K_d. Qualitatively explain the effects of increasing K_p and K_d on the response of the system.

Question 5.4
Changing K_p and K_d directly affects the bandwidth and the damping ratio of the system. Qualitatively explain, how changing K_p and K_d affects ω_n and ζ.

5.1.3 Design for Specifications

For a second order system given by Equation 5.1, the percentage overshoot formula is given by:

$$PO = 100e^{-\frac{\pi\zeta}{\sqrt{1-\zeta^2}}}$$

The peak time, also known as the time to first peak t_p, is given by:

$$t_p = \frac{\pi}{\omega_n\sqrt{1-\zeta^2}}$$

The desired step input θ_d for joint 1 is given by:

$$\theta_d = \frac{\pi}{18} \text{ rad} \tag{5.2}$$

The design specifications require that for a step input given by Equation 5.2, the following PO and t_p specs are met:

$$PO \leq 25\% \tag{5.3}$$

$$t_p \leq 0.14 \text{ s} \tag{5.4}$$

The joint inertia, J and friction B for each joint are shown in Table 5.1. Note that these joint parameters are position dependent, meaning they are only valid at the given positions of the other two joints. For the purpose of this analysis, these parameters are at the specified positions.

Joint Position	J (Kg m^2)	B (Nm s/rad)
Joint 1 (q_2 = $-\pi/4$ rad, q_3 = π rad	0.0031	0.0089
Joint 2 (q_2 = 0 rad, q_3 = π rad	0.0022	0.0170
Joint 3 (q_2 = 0 rad, q_3 = $-\pi/4$ rad	0.0009	0.0058

Table 5.1: Joint parameters

Question 5.5
Determine the parameters K_p and K_d for a Joint 1 controller so that the specifications given in Equation 5.3 and Equation 5.4 are achieved.

Changing the positions of the links will change the inertia and friction as well. Consequently, for given specs on overshoot and peak time, the response will be different. A demonstration of this will be included in the main laboratory.

1. In Simulink, create a model that is equivalent to the diagram shown in Figure 5.1 for joint 1. The parameters J and B for joint 1 are given in Table 5.1. Provide a step input with a *Final value* of **pi()/18** and *Step time*, *Initial value*, and *Sample time* of **0**.
2. Save your model as **PreLab_Position_Control.mdl**. Your model should resemble Figure 5.2.

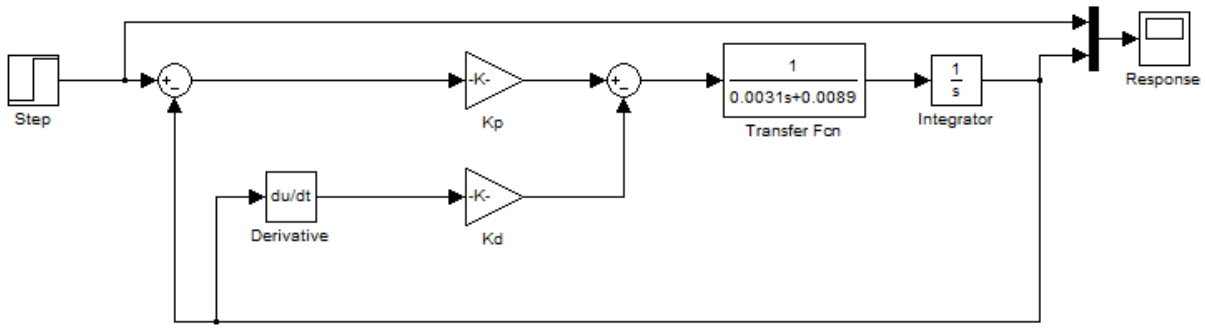


Figure 5.2: PD control model

3. Open the **Response** scope. In the scope window, click on *Parameters* icon on the toolbar. The scope parameters window will open. Go to the *History* tab and check mark *Save data to workspace*. Set the *Variable name* to **step_data** and the *Format* to **Array** as shown in Figure 5.3. This will store the data shown in the scope in an array called **scope_data** on the MATLAB Workspace.

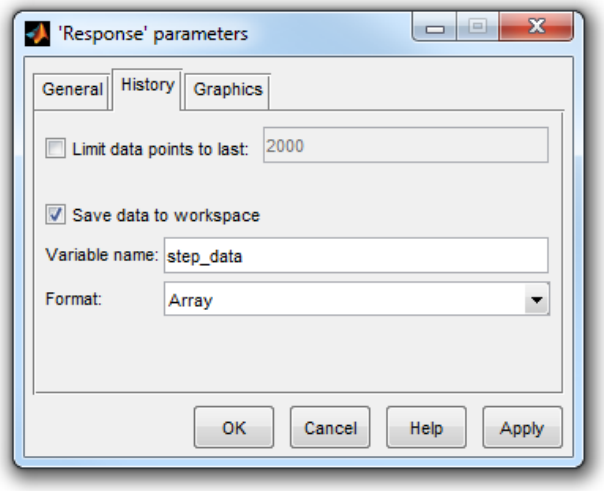


Figure 5.3: Scope parameters window

4. Fill in the PD gains you derived in Question 5.5.

Question 5.6
Run your simulation for 1 second and record the response. You can set the simulation time by changing the *Simulation stop time* to 1 in the Simulink toolbar. You can run this simulation in *Normal* mode.

5. You should notice a variable called *step_data* in the workspace. The first column of this array is the time of the simulation in seconds, the second column shows the reference signal (π/18 rad in this case) and the third column shows the actual response of the system.

Question 5.7
Determine the overshoot and peak time of the actual response. To get these values, you can use the following command in MATLAB. This command will give the properties of the step response such as rise time, overshoot etc.

```
Stepinfo(step_data(:,3), step_data(:,1), pi/18)
```

Question 5.8
Calculate the percentage error in the actual results versus the given specifications.

5.1.4 Laboratory Files to Submit

The following files should be submitted for evaluation:

- PreLab_Position_Control.mdl

5.2 In-Laboratory Experiment

In this laboratory, you will be given a PID controller for the Omni with adjustable gains. Your task will be to study the performance of the robot using different gains.

1. Open **Position_Control.mdl** and notice the **PID Control** block shown in Figure 5.4. The following is a description of its inputs and outputs:

   - **Inputs**
     - *q_cmd*: Accepts a 1×3 vector reference signal for each joint in radians.
     - *Kp*: A 1×3 vector containing the proportional gains for each joint in Nm/rad. This input is modified using slider gains to ensure that the values remain within a safe range. **Saturation** blocks within the **PID Control** block will also prevent unsafe values for the proportional gains.
     - *Kd*: A 1×3 vector containing the derivative gains for each joint in Nms/rad. Similar safety features to those specified for the proportional gain are also implemented to prevent unsafe derivative gains.
     - *Ki*: A 1×3 vector containing the integral gains for each joint in Nm/s. Similar safety features to those specified for the proportional gain are also implemented to prevent unsafe integral gains.
   - **Outputs**
     - *Kp_out*: A 1×3 vector that outputs the actual proportional gains being used for each joint in Nm/rad. This output equals the input *Kp* if the upper and lower thresholds are not exceeded. Otherwise, the output is saturated to either the upper or the lower threshold value.
     - *Kd_out*: A 1×3 vector that outputs the actual derivative gains being used for each joint in Nms/rad. This output equals the input *Kd* if the upper and lower thresholds are not exceeded. Otherwise, the output is saturated to either the upper or the lower threshold value.
     - *Ki_out*: A 1×3 vector that outputs the actual integral gains being used for each joint in Nm/s. This output equals the input *Ki* if the upper and lower thresholds are not exceeded. Otherwise, the output is saturated to either the upper or the lower threshold value.
     - *q_start*: A 1×3 vector that outputs the initial starting position of each joint in radians.
     - *q*: A 1×3 vector that outputs the actual position of each joint in radians.

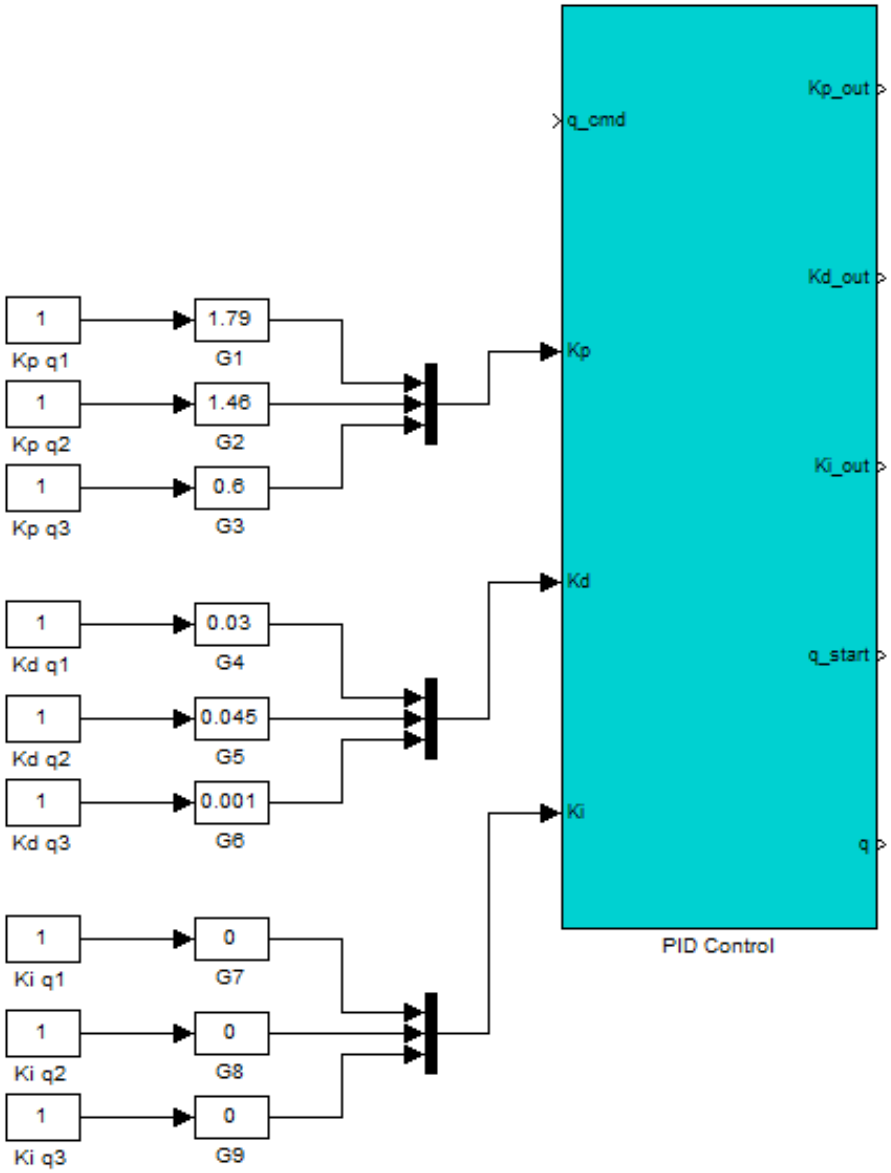


Figure 5.4: PID controller

5.2.1 Joint 1 Control and Effects of K_p

1. Save your model as **Position_Control_Joint1.mdl**.
2. Create a square wave input for joint 1 of magnitude pi/18 about 0. Use a **Signal Generator** block with a *Wave form* of type **Square**, *Amplitude* of **pi/18**, *Frequency* of **0.5**, and *Units* set to **Hertz**.
3. Apply a reference signal of **-pi/4** to joint 2, and a reference of **pi** to joint 3.

   Note: Do not apply a step size larger than pi/18. Larger step inputs could damage the device
4. Create the model as shown in Figure 5.5. The **Goto** and **From** blocks are used to connect signals and keep the model clutter-free. In this case, the signal from the **Signal Generator** block is connected to one of the inputs of the **Mux** at the output, *q*. Corresponding **Goto** and **From** blocks must have the same name.

   The **Demux** block shown in Figure 5.5 is used to split a signal into its components. In this case, we want to plot the actual position of joint 1 and the commanded position for joint 1 on the same plot.

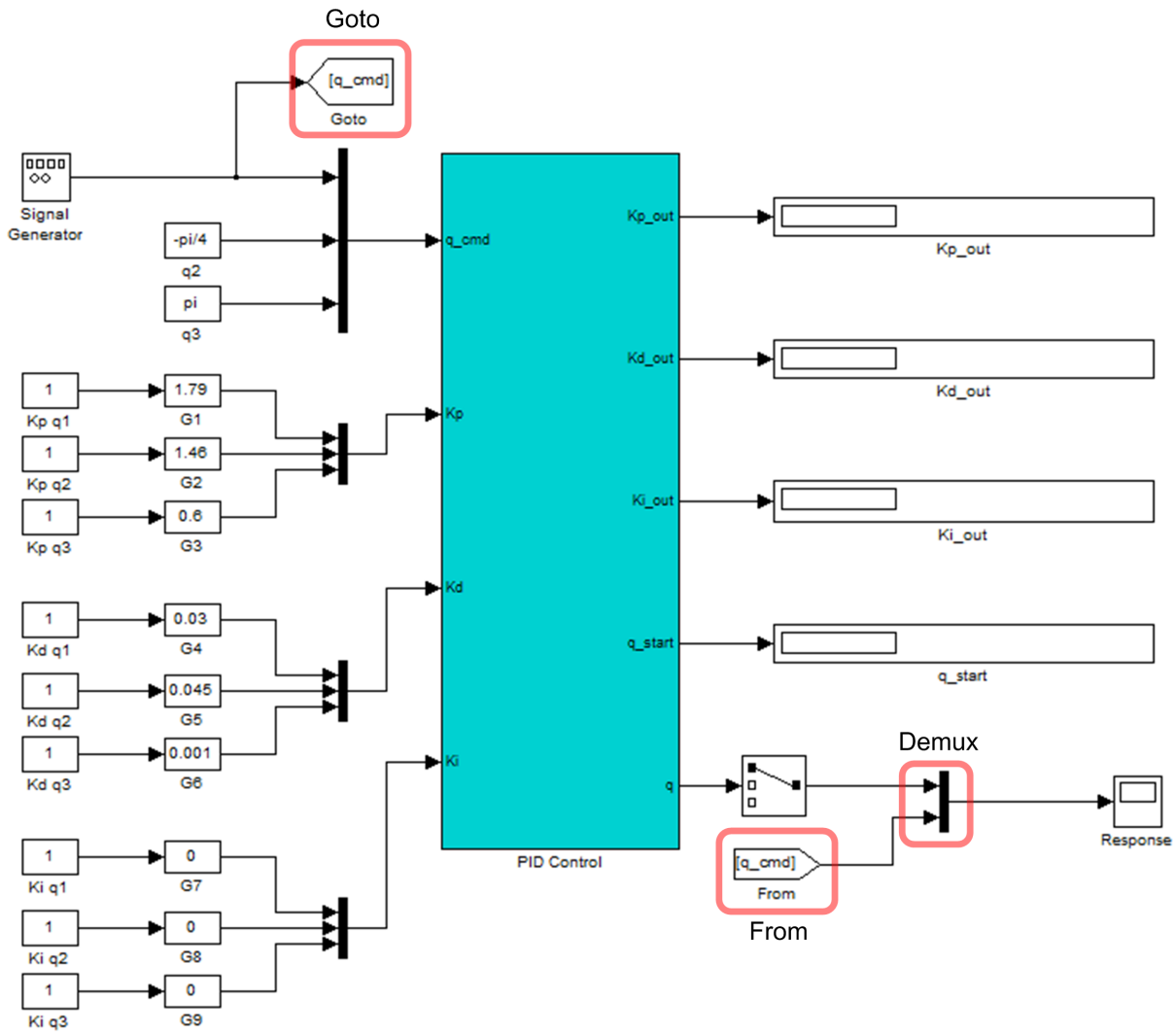


Figure 5.5: Joint 1 position control model

5. Keep the default value of the gains K_p, K_d and K_i (blocks **G1**, **G4** and **G7**) at 1.79 Nm/rad, 0.03 Nms/rad and 0 Nm/s respectively.
6. Before running this model, place the end-effector of the Omni on position 2 of the baseboard.
7. Set the **Simulation mode** to *External* in the Simulink toolbar.

   Question 5.9
   Build and run your model for 5 seconds and record your results.

8. Decrease the proportional gain by changing the value of the block **G1** to 0.8 Nm/rad.
9. Place the end-effector of the Omni on position 2 of the baseboard.

   Question 5.10
   Re-run your model and record the step response.

   Question 5.11
   Explain how decreasing K_p affected the response.

5.2.2 Joint 1 Velocity

In this section, you will determine the maximum velocity that joint 1 can travel with given a step size of π/18 rad and K_p, K_d and K_i equal to 1.79 Nm/rad, 0.03 Nms/rad and 0 Nm/s respectively.

1. Set the value of block **G1** back to 1.79 Nm/rad.
2. From **QUARC Targets** | **Continuous** library, drag and drop the **Second-Order Low-Pass Filter** block into the model. This block differentiates a signal while filtering out the noise due to differentiation. The block is shown in Figure 5.6. Set the cut-off frequency parameter to **200** Hz and the Damping ratio to **1**. The input *x* is the signal to be differentiated. In this case it is the actual position of joint 1 taken from the output *q* of the **PID Control** block. The output *yd* will be the differentiated signal. In this case, it would be the velocity of joint 1.

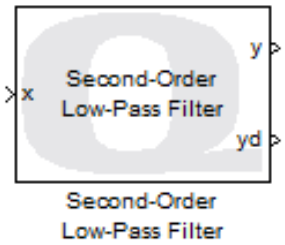


Figure 5.6: Second-order low-pass filter

3. Your model should resemble Figure 5.7

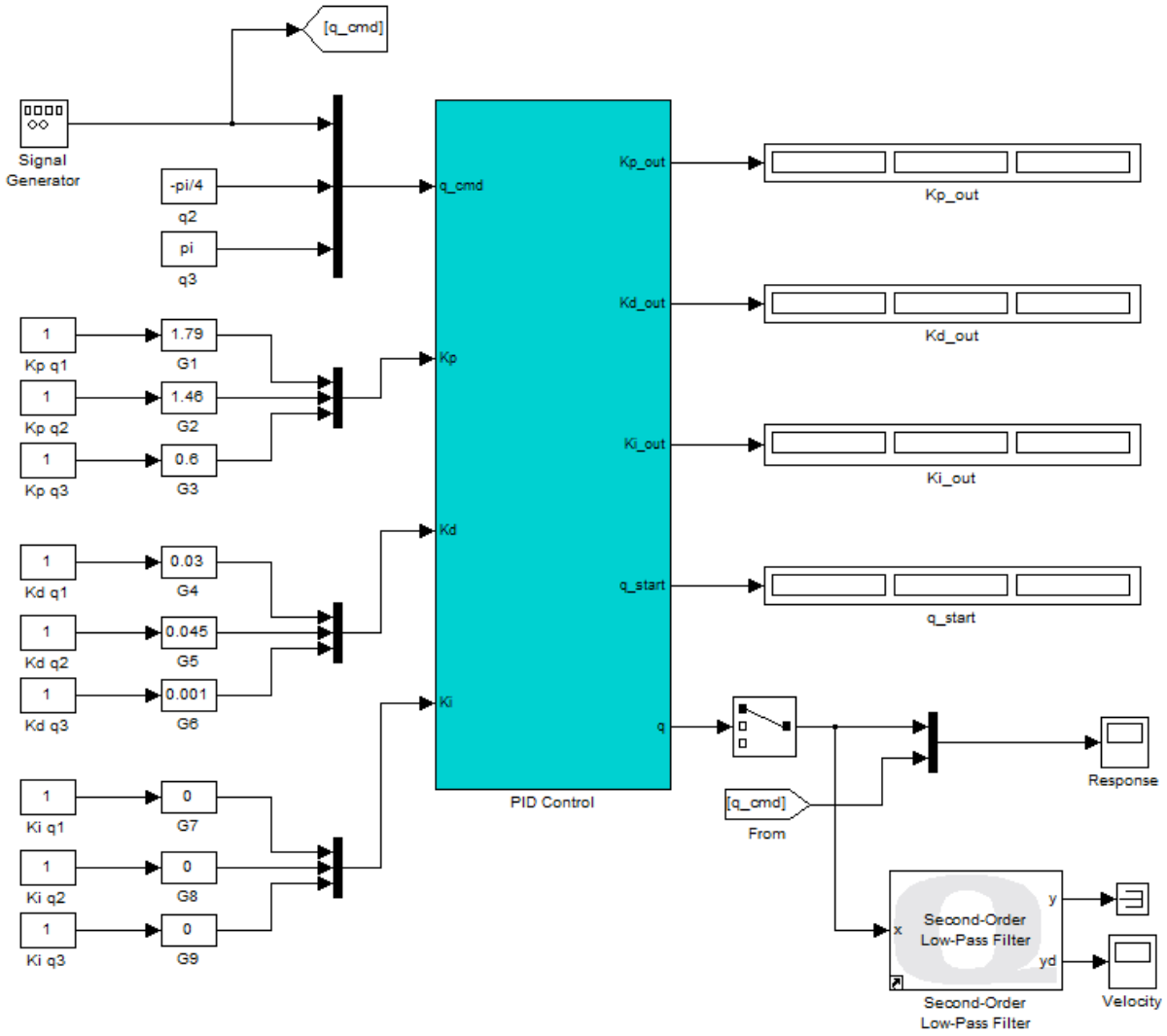


Figure 5.7: Joint 1 velocity measurement model

4. Place the end-effector of the Omni on position 2 of the baseboard.

Question 5.12
Build and run your model and record the velocity, *yd*.

Question 5.13
From your plot, record the magnitude of maximum velocity of joint 1 in both positive and negative directions.

5.2.3 Joint 3 Control and Effects of K_i

1. Start a new Simulink Model and insert the **PID Control** block from **Position_Control.mdl**. Save this model as **Position_Control_Joint3.mdl**.

2. Create a square wave input for joint 3 of magnitude pi/18 about pi. Use a **Signal Generator** block with a *Wave form* of type **Square**, *Amplitude* of **pi/18**, *Frequency* of **0.5**, and *Units* set to **Hertz**.

3. Apply a reference signal of **-pi/4** to joint 2, and a reference of **0** to joint 1.
4. Use a **Bias** block with a bias of **pi** rad to shift the square wave up by π rad. Your model should resemble Figure 5.8.

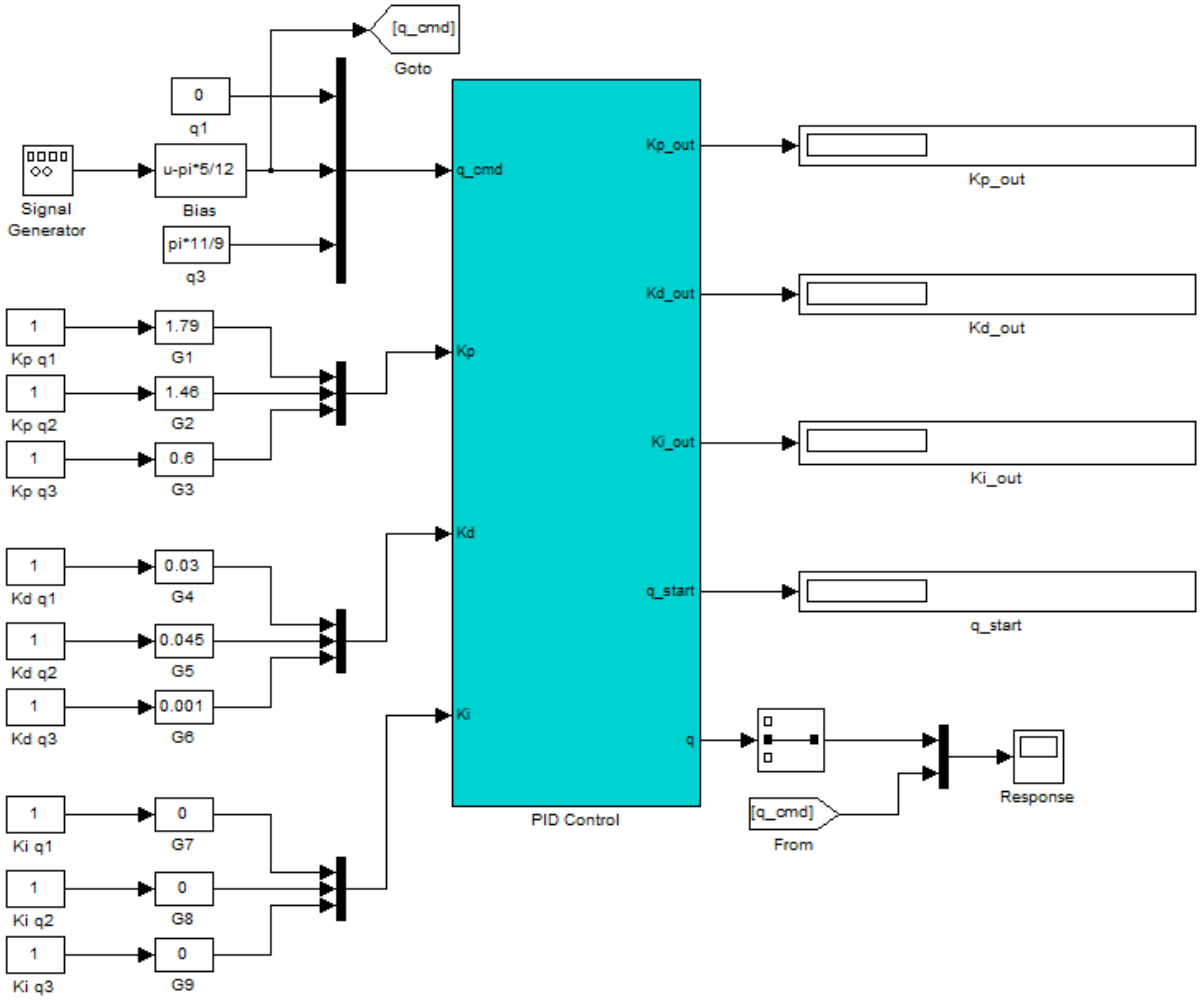


Figure 5.8: Joint 3 position controller

5. Keep the default value of the gains K_p, K_d and K_i (blocks **G3**, **G6** and **G9**) at 0.6 Nm/rad, 0.001 Nms/rad and 0 Nm/s respectively.
6. Place the end-effector of the Omni on position 2 of the baseboard.
7. Set the **Simulation mode** to *External* in the Simulink toolbar.

   Question 5.14
   Build and run your model for 5 seconds and record your results.

8. Increase the derivative gain by changing the value of the block **G6** to 0.01 Nms/rad.
9. Place the end-effector of the Omni on position 2 of the baseboard.

   Question 5.15
   Re-run your model and record the step response.

Question 5.16
Explain how increasing K_d affected the response.

5.2.4 Joint 3 PID Control

Notice the large steady state error in the results in Section 5.2.3. In this section, you will learn how a PID controller can be used to decrease the steady state error in the response.

1. Open the **Signal Generator** block and change the *Amplitude* to **-pi/18** rad and the *Frequency* to **0.1** Hertz. This will allow us to see the step response behavior.
2. Open the slider gain block, **G9**. This gain is the integral gain corresponding to joint 3.
3. Change the value of this gain from **0** Nm/s to **0.5** Nm/s.
4. Place the end-effector of the Omni on position 2 of the baseboard.

   Question 5.17
   Run your model and record your results. You should notice a significant decrease in the steady state error.

5. The time that it takes for the error to remain within 2% of its final value is called the settling time.

   Question 5.18
   Slowly change the gain in the **G9** block to bring the settling time to approximately 1s. Record your result and provide the value of the gain used.

5.2.5 Joint 3 Velocity

In this section you will determine the maximum velocity that joint 3 can travel with given a step size of π/18 rad and K_p and K_d equal to 0.6 Nm/rad and 0.001 Nms/rad respectively. The value of K_i should be set 0 Nm/s.

1. Set the value of block **G6** back to **0.001** Nm/rad. Set the *Amplitude* and *Frequency* of the **Signal Generator** block back to **pi/18** rad and **0.5** Hertz.
2. Modify the model to measure the velocity of joint 3 as outlined in Section 5.2.2. Your model should resemble Figure 5.9.

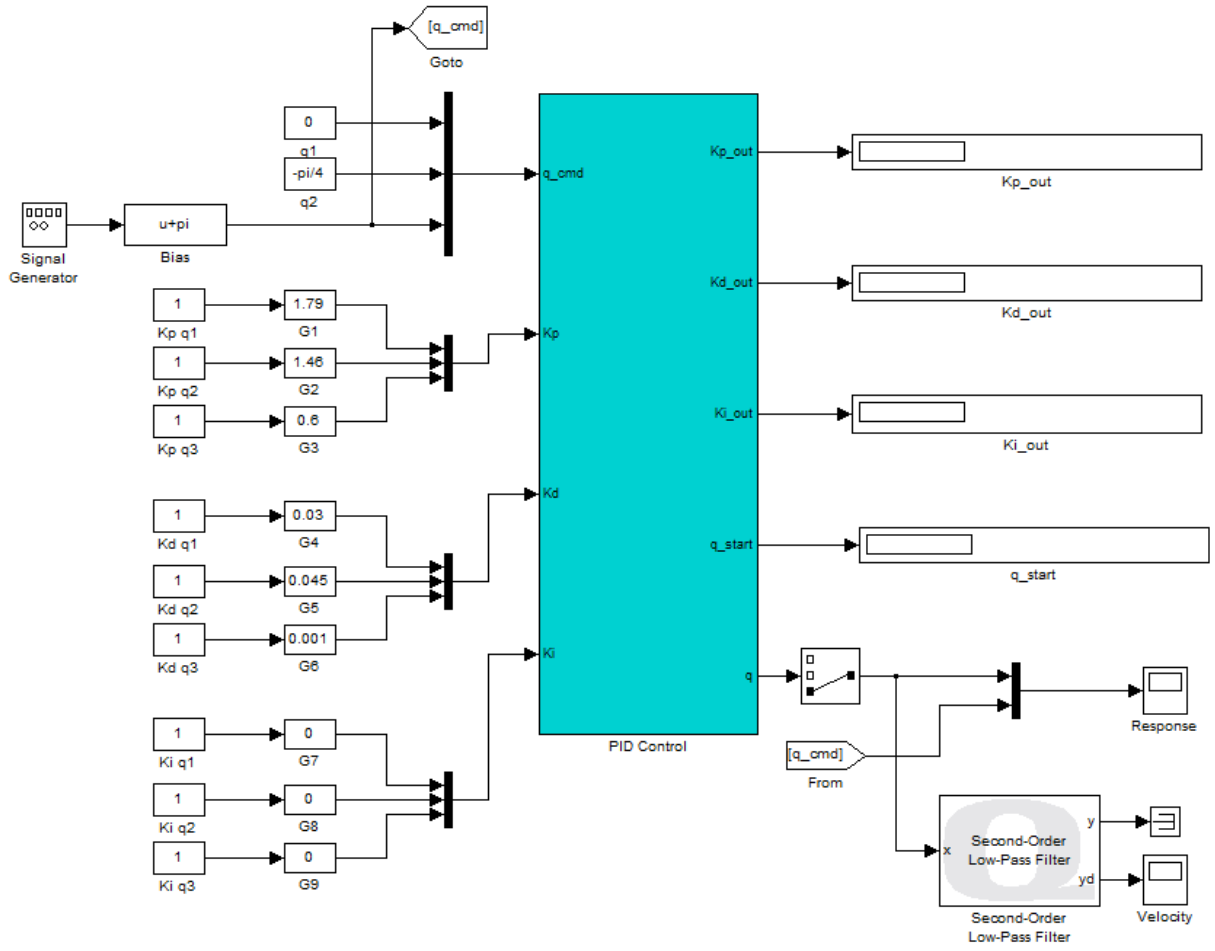


Figure 5.9: Joint 3 velocity measurement model

3. Place the end-effector of the Omni on position 2 of the baseboard.

Question 5.19
Build and run your model and record the velocity, *yd*.

Question 5.20
From your plot, record the magnitude of maximum velocity of joint 3 in both positive and negative directions.

5.2.6 Joint 2 Control and Effects of Gravitational Loading

1. Start a new Simulink Model and insert the **PID Control** block from **Position_Control.mdl**. Save this model as **Position_Control_Joint2.mdl**.
2. Create a square wave input for joint 2 of magnitude **pi/18** about **5*pi/12**.
3. Apply a reference signal of **0** to joint 1, and a reference of **11*pi/9** to joint 3.
4. Use a **Bias** block with a bias of **-5*pi/12** rad to shift the square wave down. Your model should resemble Figure 5.10.

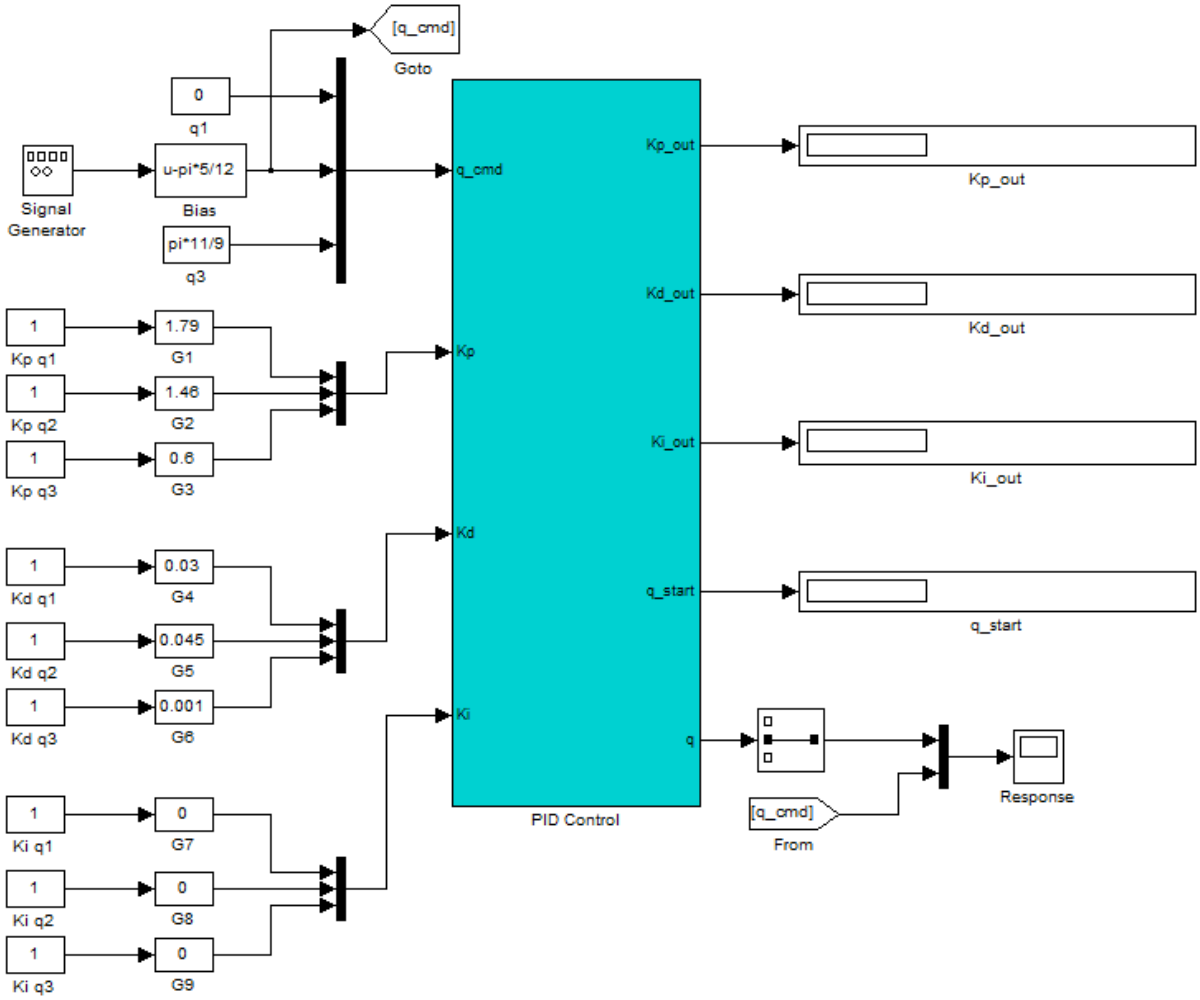


Figure 5.10: Joint 2 position controller

5. Keep the default value of the gains K_p, K_d and K_i (blocks **G2**, **G5** and **G8**) at 1.46 Nm/rad, 0.045 Nms/rad and 0 Nm/s respectively.
6. Place the end-effector of the Omni on position 2 of the baseboard.
7. Set the **Simulation mode** to *External* in the Simulink toolbar.

   Question 5.21
   Build and run your model for 5 seconds and record your results.

   Question 5.22
   Is there any difference between the steady-state error on the positive and negative sides. Explain why or why not?

8. Next change the position of joints 2 and 3 to change the gravitational loading. In the **Bias** block, change the value to **-pi/4** rad. Set the reference signal for joint 3 to **3*pi/4** rad.

   Question 5.23
   Does this configuration increase or decrease the gravitational loading on joint 2?

9. Place the end-effector of the Omni on position 2 of the baseboard.

Question 5.24
Run the model and record your results.

Question 5.25
Explain how changing the gravitational loading changed the response.

5.2.7 Joint 2 PID Control

Notice the large steady state error in the results in Section 5.2.6. In this section, you will create a PID controller to decrease the steady state error in the response.

1. Open the **Signal Generator** block and change the *Amplitude* to **-pi/18** rad and the *Frequency* to **0.1** Hertz.
2. Open the slider gain block, **G8**. This gain is the integral gain corresponding to joint 2.
3. Change the value of this gain from **0** Nm/s to **0.5** Nm/s.
4. Place the end-effector of the Omni on position 2 of the baseboard.

Question 5.26
Run your model and record your results. You should notice a significant decrease in the steady state error.

Question 5.27
Slowly change the gain in the **G8** block to bring the settling time to approximately 2 s. Record your result and provide the value of the gain used.

5.2.8 Joint 2 Velocity

In this section you will determine the maximum velocity that joint 2 can travel with given a step size of π/18 rad and K_p and K_d equal to 1.46 Nm/rad and 0.045 Nms/rad respectively. The value of K_i should be set 0 Nm/s.

1. Set the *Amplitude* and *Frequency* of the **Signal Generator** block back to **pi/18** rad and **0.5** Hertz.
2. Modify the model to measure the velocity of joint 2 as outlined in Section 5.2.2. Your model should resemble Figure 5.11.

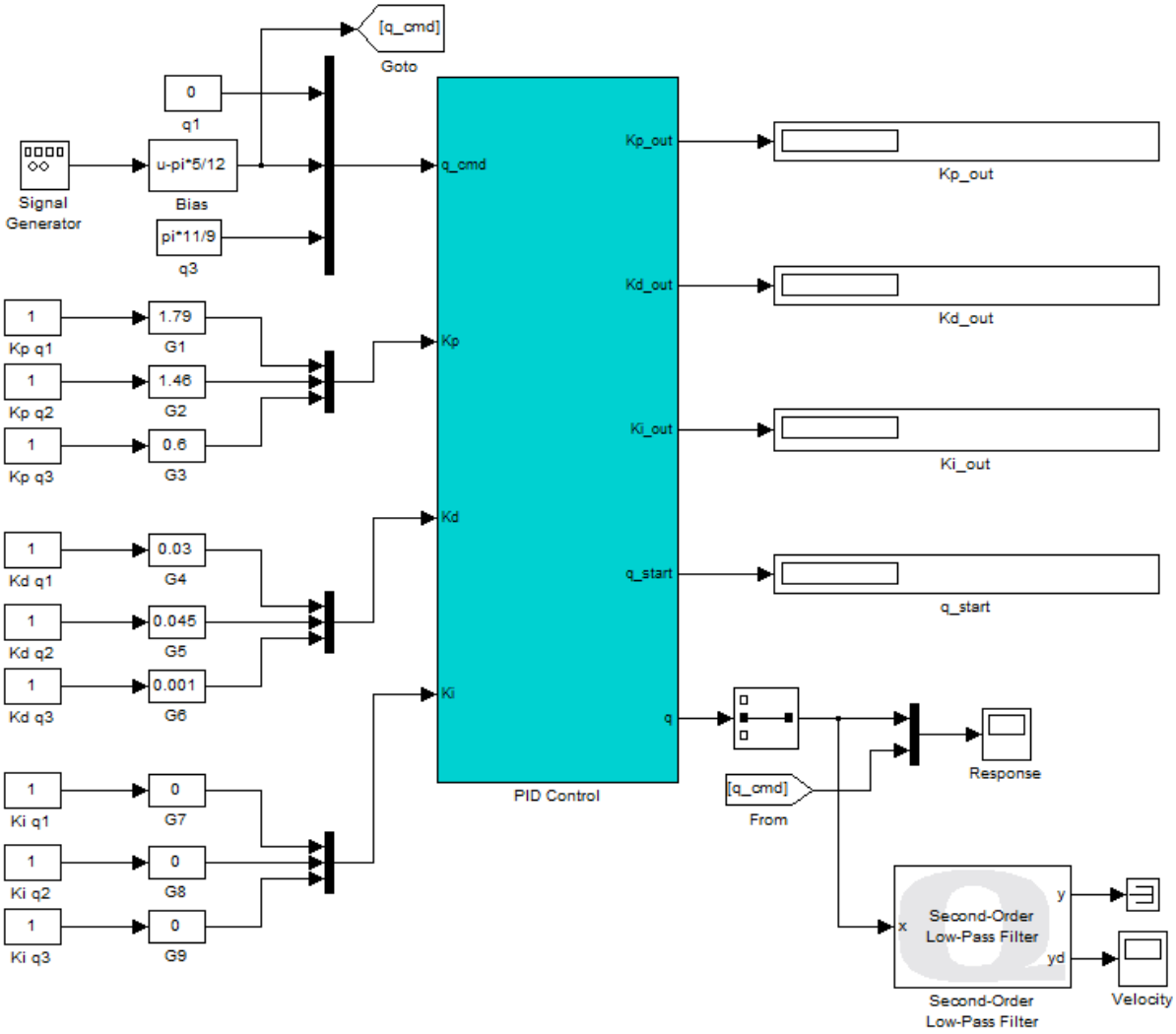


Figure 5.11: Joint 2 velocity measurement model

3. Place the end-effector of the Omni on position 2 of the baseboard.

Question 5.28
Build and run your model and record the velocity, *yd*.

Question 5.29
From your plot, record the magnitude of maximum velocity of joint 2 in both positive and negative directions.

5.2.9 Laboratory Files to Submit

The following files should be submitted for evaluation:

- Position_Control_Joint1.mdl
- Position_Control_Joint2.mdl
- Position_Control_Joint3.mdl

6 TEACH PENDANT IN JOINT SPACE

The objective of this laboratory is to design a teach pendant using the Omni. In a teach pendant experiment, a set of discrete points are taught to the robot. During playback motion, the robot traverses each of the taught points. Programming the robot to do this is a three step process. It involves creating a routine to teach the points to the robot, then creating a desired path between those points that the robot should follow and then finally creating a routine to control the robot along that path.

Depending on the application of the robot, the trajectory between any two points could take any shape. Workspace or task requirements might require the robot to move in straight lines or on a curved path. These trajectories could be planned in either task space or joint space. In task space, it is the end-effector that moves in a specified trajectory. In joint space, you create a trajectory for the joints instead.

Applications for this approach to trajectory planning include pick and place tasks or following a welding contour.

Topics Covered

- Using an existing Simulink model to teach points to the robot.
- Creating linear trajectories between each point in joint space.
- Using an existing Simulink model to control each joint of the robot along the linear path.

6.1 Pre-Laboratory Assignments

In this section, you will write MATLAB functions to create linear trajectories between a set of points at a specified slope. The trajectories will be created in joint space.

1. Write a MATLAB function called **LinearTrajectory.m** as follows:

   - **Inputs**
     - *q_current*: A 1×3 vector that accepts the joint positions at the starting point, in radians
     - *q_next*: A 1×3 vector that accepts the position of each joint at the next point, in radians
     - *step_size*: A 1×3 vector that stores the size of each step to go from *q_current* to *q_next*
   - **Outputs**
     - *trajectory*: An n×3 matrix that outputs the trajectory, in radians, between *q_current* and *q_next* where *n* is the number of points traversed

   (a) This function creates a linear trajectory between two points (*q_current* and *q_next*) according to the step size.

   (b) The trajectory ends only when *q_next* is reached. If *q_next* cannot be reached due to the step size, then for the last step in the trajectory, ignore the step size and append the trajectory with *q_next*.

   (c) For example, if *q_current* = [1 2 3], *q_next* = [3.6 3.8 5.9], and *step_size* = [0.5 0.3 0.5] then the following trajectory is created:

$$\begin{bmatrix} 1.5 & 2.3 & 3.5 \\ 2 & 2.6 & 4 \\ 2.5 & 2.9 & 4.5 \\ 3 & 3.2 & 5 \\ 3.5 & 3.5 & 5.5 \\ 3.6 & 3.8 & 5.9 \end{bmatrix}$$

   Notice that between the last two points, the step size is no longer [0.5 0.3 0.5]. If this step size was used, point *q_next* would have been overshot.

2. Write a MATLAB script called **MakeLinearTrajectory_JointSpace.m** as follows:

   (a) We know from forward kinematics that for every combination of joint angles (q_1, q_2 and q_3), there exists a solution that gives the position of the end-effector in 3D space. To make this function, assume that you have an $m \times 3$ workspace variable that stores the position of each joint for m different points in space. This variable will be called *q*.

   (b) Create a 1×3 vector, *q_dot* in rad/s, to store the speed of each joint.

   (c) Read the *q* matrix one row at a time and determine *q_current* and *q_next*.

   (d) All joints must stop and start at the same time. To do this, you have to determine the joint that would take the longest time to travel. This time is dependent on the distance the joint has to travel and the speed of that joint.

   Question 6.1
   Assume that the joints [q_1, q_2, q_3] have to travel from their initial position of *q_current* = [θ_1 θ_2 θ_3], to *q_next* = [α_1 α_2 α_3]. If the joints travel at their maximum velocity, devise an algorithm to determine the minimum time taken by each joint given that the maximum speed for each joint is given by *q_dot* = [$\dot{q}_1$, $\dot{q}_2$, $\dot{q}_3$].

   Question 6.2
   Using the time taken by each joint, find the maximum possible time to complete the trajectory (i.e. the time taken by the slowest joint). Use it to devise an algorithm to determine the modified speed (*q_dot_modified*) for all joints such that all joints finish their trajectory at the same time.

(e) Create a 1×3 vector, *step_size*, to store the size of step to be taken by each joint. Use *step_size* = 0.001**q_dot_modified*. 0.001 is chosen because it will also be the sampling time of the Simulink models.

(f) Call **LinearTrajectory.m** with *q_current*, *q_next* and *step_size*.

(g) Repeat this process until trajectories between all points have been created, including the last point (point *m*) and the first point (point 1). Store the completed trajectory in *traj_q*.

(h) Finally, concatenate the first row of the *q* matrix with *traj_q* to complete the trajectory.

(i) For example, if *q_current* = [1 2 3], *q_next* = [3.6 3.8 5.9], and *step_size* = [0.5 0.3 0.5] then *traj_q* is:

$$\begin{bmatrix} 1 & 2 & 3 \\ 1.5 & 2.3 & 3.5 \\ 2 & 2.6 & 4 \\ 2.5 & 2.9 & 4.5 \\ 3 & 3.2 & 5 \\ 3.5 & 3.5 & 5.5 \\ 3.6 & 3.8 & 5.9 \\ 3.1 & 3.5 & 5.4 \\ 2.6 & 3.2 & 4.9 \\ 2.1 & 2.9 & 4.4 \\ 1.6 & 2.6 & 3.9 \\ 1.1 & 2.3 & 3.4 \\ 1 & 2 & 3 \end{bmatrix}$$

3. Recall that in Section 5, you determined the maximum speed for each joint for a step input of π/18 rad. In this laboratory, we will create trajectories at approximately 25% of the maximum speed of each joint. This proportion of the maximum speed is chosen to prevent wear and tear on the motors. This speed is:

$$q_{dot} = [1.20.91.7] \tag{6.1}$$

Question 6.3
Test your function with the following *q* matrix, and maximum speed given in Equation 6.1. Plot each column of *traj_q*.

6.1.1 Laboratory Files to Submit

The following files should be submitted for evaluation:

- LinearTrajectory.m
- MakeLinearTrajectory_JointSpace.m

6.2 In-Laboratory Experiment

In this laboratory, you will be provided with two Simulink models: **Teach_Points.mdl** and **Teach_Pendant_Joint_Space.mdl**. **Teach_Points.mdl** will be used to teach different points to the Omni. Then you will use the functions created in the pre-laboratory to make linear trajectories between those points. These trajectories will be used by the second Simulink model to control the robot along that path.

6.2.1 Teaching Points

1. Open **Teach_Points.mdl**, shown in Figure 6.1.

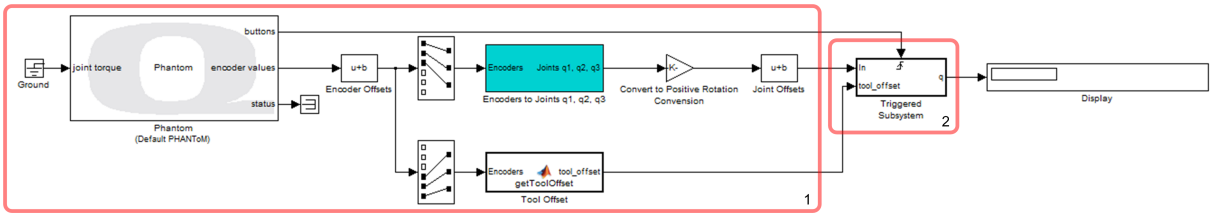


Figure 6.1: Teach pendant model

(a) Section 1: Notice the blocks under section 1 are the same as those you created in Section 3.2.1. These blocks convert the encoder values for each joint into position in radians.

(b) Section 2: The blocks under section 2 record the position of each joint whenever a button is pressed.

The model under the triggered subsystem, shown in Figure 6.2, executes at the rising edge of an event. In this case the event is a button press. When a button on the Omni is pressed, this subsystem executes. It records the position of each joint (in radians) in a workspace variable called *q*. Then it performs forward kinematics analysis to find the position of the end-effector (in meters) and records this information in a workspace variable called *pos*. Every time the button is pressed, a new row is created in each of these variables to store the information. For instance if the button is pressed 4 times, matrices *q* and *pos* on the workspace will be both 4×3. A flowchart of this process if shown in Figure 6.3.

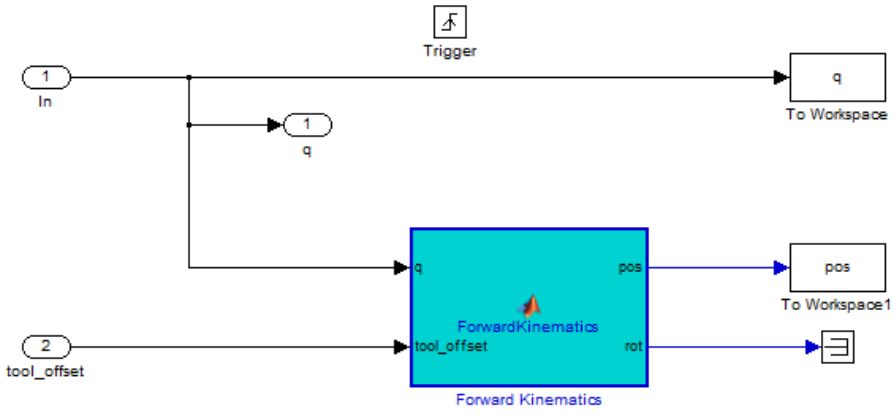


Figure 6.2: Triggered subsystem

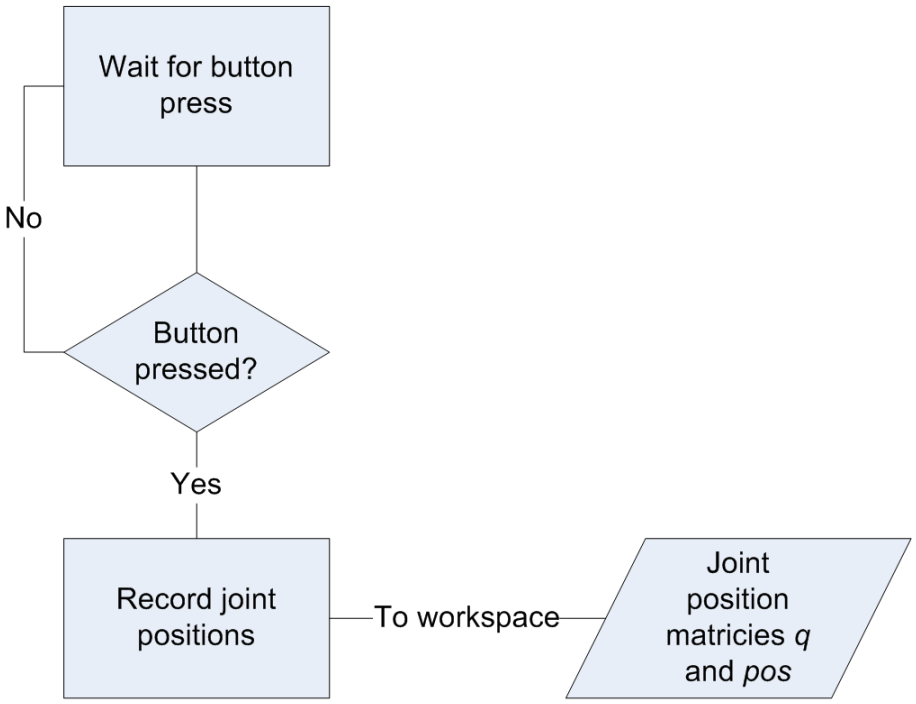


Figure 6.3: **Teach_Points.mdl** flowchart

2. Check to ensure that the **Simulation mode** is set to *External*.
3. Build the model and run it.
4. While the model is still running, move the end-effector approximately 2 to 3 cm above position 4 on the baseboard and press a button once.
5. Repeat step 4 for positions 8 and 3 on the baseboard. Each time, be sure to hold the end-effector approximately 2 to 3 cm above the baseboard.
6. Stop the model. Now you should notice variables *q* and *pos* on the MATLAB workspace. Both of these should be 3×3 matrices.

   Question 6.4
   Record the position values from the workspace.

6.2.2 Creating Trajectories

1. Open **MakeLinearTrajectories_JointSpace.m** from the pre-laboratory and run the file.

   Question 6.5
   Plot each column of *traj_q*.

6.2.3 Controlling the Robot

1. Open **Teach_Pendant_Joint_Space.mdl**, shown in Figure 6.4.

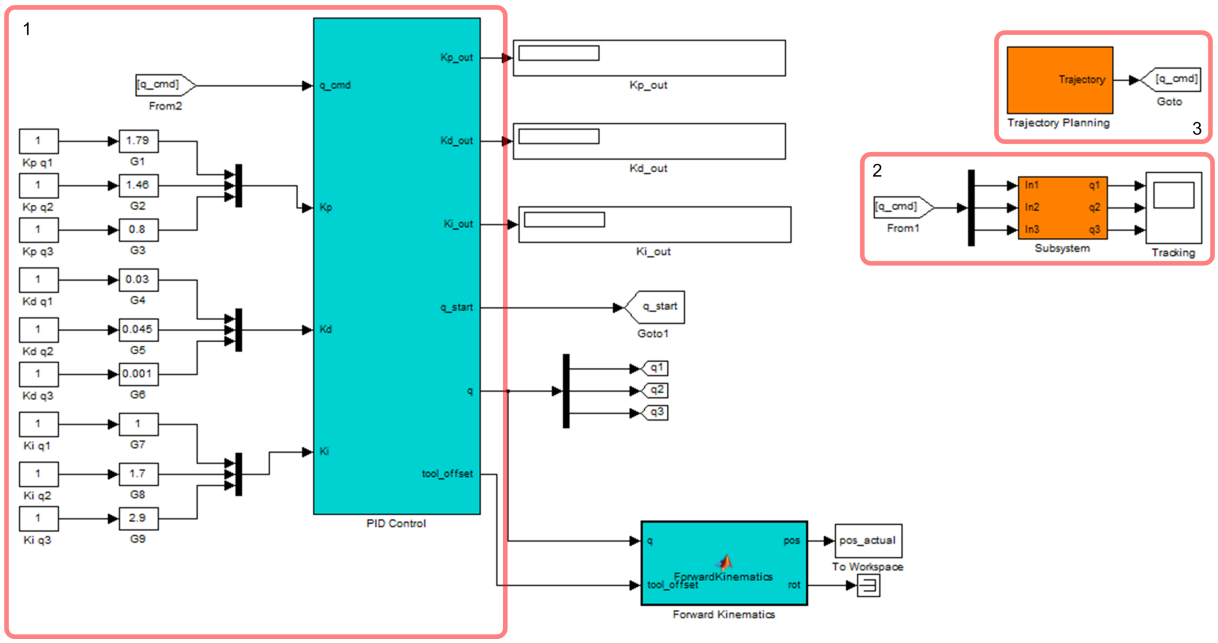


Figure 6.4: Joint space trajectory follower model

(a) Section 1: The model under section 1 is the PID joint controller you used in Section 5.2. Its purpose is to control each joint of the robot according to the trajectory you created.

(b) Section 2: The model under this section reads the trajectory from the workspace variable *traj_q* row by row and sends it to the controller.

(c) Section 3: The model here is just outputting the actual positions of each joint and the desired positions. These are plotted together on the scope called **Tracking**.

2. Before running this model, place the end-effector of the Omni on position 4 of the baseboard.

3. Ensure that the **Simulation mode** is set to *External*. Build and run this model for 10 seconds. Observe how the end-effector traverses through each of the taught points.

Question 6.6
Record the **Tracking** scope which displays the commanded and actual position of each joint.

4. This simulation also output a variable called *pos_actual* onto the MATLAB workspace. This columns 1, 2 and 3 of this matrix contain the path that the end-effector followed in x, y and z coordinates (in meters).

Question 6.7
Use the plot command in MATLAB to plot the top view of the actual position of the end-effector. Exclude the beginning and ending 10% of the data. Also plot the actual points that were taught. The commands are as follows:

```
hold on
plot(pos_actual(1000:9000,2), pos_actual(1000:9000,1))
plot(pos(1,2),pos(1,1),'or','Linewidth', 2)
plot(pos(2,2),pos(2,1),'or','Linewidth', 2)
plot(pos(3,2),pos(3,1),'or','Linewidth', 2)
hold off
```

Question 6.8
Does the end-effector move from point to point in one straight line? Explain

6.2.4 Laboratory Files to Submit

The following files should be submitted for evaluation:

- LinearTrajectory.m
- MakeLinearTrajectory_JointSpace.m

7 TEACH PENDANT IN TASK SPACE

In Section 6, you learned how to teach the Omni different points in space and then created linear trajectories between those points. These trajectories were made in joint space. That is, the motion was applied to the joints. However, in many applications it may be necessary that we design for the path of the end-effector and not the joints. Applications of this design are more intuitive. If the robot is required to move along a welding contour, then it may be more intuitive to provide a path for the end-effector and not the joints. In this laboratory, you will create linear trajectories for the motion of the end-effector. As a result, you will observe that the end-effector moves in a linear motion for one point to the next. This is also referred to as working in the task space.

Topics Covered

- Use an existing Simulink model to teach points to the robot.
- Create linear trajectories between each point in task space.
- Perform inverse kinematics analysis to translate the trajectories into joint space.
- Use an existing Simulink model to control each joint of the robot along the translated path.

7.1 Pre-Laboratory Assignments

In this section, you will write MATLAB functions to create linear trajectories between a set of points at a specified slope. The trajectories will be created in task space.

1. This laboratory will use **LinearTrajectory.m** from Section 6.1
2. Make the following changes to **MakeLinearTrajectory_JointSpace.m** file from Section 6.1 (save the modified file as **MakeLinearTrajectory_TaskSpace.m**):

   (a) Instead of variables *q*, *q_current*, *q_next*, *q_dot* and *traj_q*, use variables *pos*, *pos_current*, *pos_next*, *pos_dot* and *traj_pos* respectively.

   (b) Recall that in Section 6.1, the speed we used for each joint was given by Equation 6.1. When the end-effector moves in the x-y plane, the most work is done by joint 1 and joint 3. Joint 2 does not change significantly. The maximum speed of the end-effector can be calculated using the following:

   ```
   Vmax = r*q_dot
   ```

   where r is the radius from a joint to the end-effector.

   To move the end-effector in the x-direction, joint 3 is used the most. Joint 3 is a distance of 0.132 m (length of link 2) from the end-effector. Joint 1 is used the most to move in the y-direction. Assuming that the angle between links 1 and 2 is $\pi/2$ rad while the end-effector moves in the x-y plane, the distance between joint 1 and the end-effector can be approximated by 0.132 m (length of link 1). Joint 2 is used to move in the z-direction. Once again assuming orthogonal links, the radius from joint 2 to the end-effector can be approximated by 0.132 m (length of link 1).

   Therefore, *pos_dot* is given by:

   ```
   pos_dot = 0.132*[1.7 1.2 0.9]
   pos_dot = [0.2 0.15 0.12]
   ```

3. Test your function with the *pos_dot* specified above, and the following *pos* matrix:

$$\begin{bmatrix} 0.04 & 0.007 & -.0015 \\ 0.011 & 0.048 & 0 \\ 0.06 & 0.05 & -0.07 \end{bmatrix}$$

Question 7.1

In a 3D plot, plot the 3 columns of *traj_pos*. Use the following `plot3` command in MATLAB.

```
plot3(traj_pos(:,1),traj_pos(:,2),traj_pos(:,3))
view(-38,10)
```

You should see a triangle traced out between each of the three taught points. The `view` command above orients the figure so that easy to see. The rotation about the negative y-axis is -38 in degrees, and the elevation is 10 in degrees.

7.1.1 Laboratory Files to Submit

The following files should be submitted for evaluation:

- LinearTrajectory.m
- MakeLinearTrajectory_TaskSpace.m

7.2 In-Laboratory Experiment

In this laboratory, you will be provided with two Simulink models: **Teach_Points.mdl** and **Teach_Pendant_Task_Space.mdl**. **Teach_Points.mdl** is the same model from Section 6.2. It will be used to teach different points to the Omni. Then you will use the functions created in the pre-laboratory to make linear trajectories between those points in task space. These trajectories will be used by the second Simulink model to control the robot along that path.

7.2.1 Teaching Points

1. Open **Teach_Points.mdl**
2. Check to ensure that the **Simulation mode** is set to *External*.
3. Build the model and run it.
4. While the model is still running, move the end-effector approximately 2 to 3 cm above position 4 on the baseboard and press a button once.
5. Repeat step 4 for positions 8 and 3 on the baseboard. Each time, be sure to hold the end-effector approximately 2 to 3 cm above the baseboard.
6. Stop the model. Now you should notice variables *q* and *pos* on the MATLAB workspace. Both of these should be 3×3 matrices.

   Question 7.2
   Record the position values from the workspace.

7.2.2 Creating Trajectories

1. Open **MakeLinearTrajectories_TaskSpace.m** from the pre-laboratory and run the file.

   Question 7.3
   Create a 3D plot of the resultant trajectory.

7.2.3 Controlling the Robot

1. Open **Teach_Pendant_Task_Space.mdl**, shown in Figure 7.1.

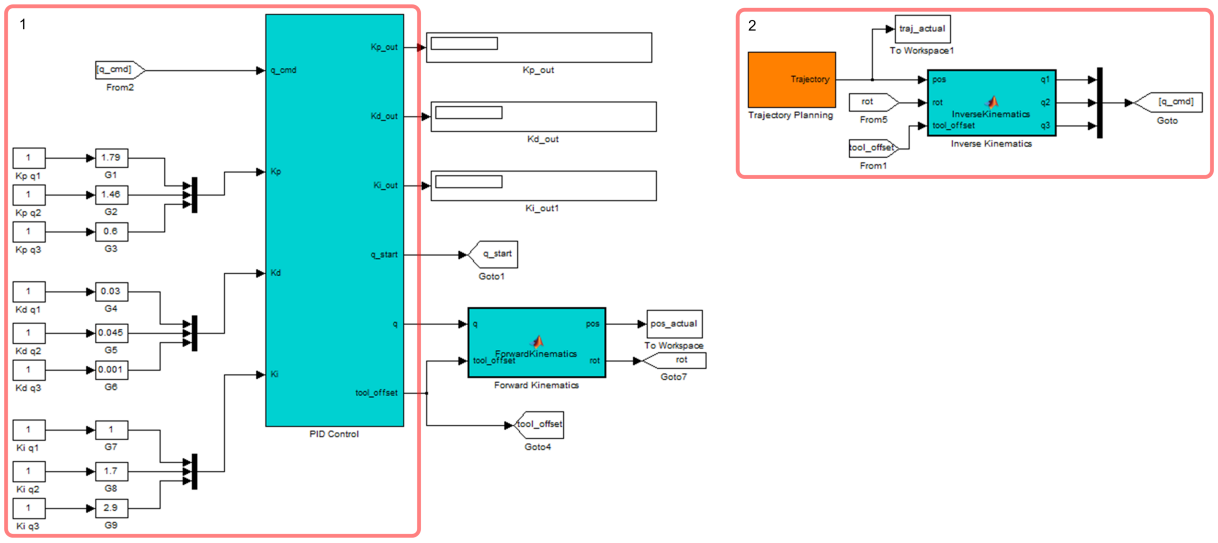


Figure 7.1: Task space trajectory follower model

(a) Section 1: The model under section 1 is the PID joint controller you used in Section 5.2. Its purpose is to control each joint of the robot according to the trajectory you created.

(b) Section 2: The model under this section reads the trajectory from the workspace variable *traj_pos* row by row and performs an inverse kinematics analysis on it to convert it to joint positions. These joint positions are controlled by the PID controller.

2. Before running this model, place the end-effector of the Omni on position 4 of the baseboard.

3. Ensure that the **Simulation mode** is set to *External*. Build and run this model for 10 seconds. Observe how the end-effector traverses through each of the taught points. This simulation will output a variable *pos_actual* onto the MATLAB workspace. Columns 1, 2 and 3 of this matrix contain the actual position of the end-effector in x, y and z coordinates respectively.

Question 7.4
Use the plot command in MATLAB to plot the top view of the actual position of the end-effector as described in Question 6.7. Exclude the beginning and ending 10% of the data. Also plot the actual points that were taught.

Question 7.5
Compare this plot with the one you obtained in Question 6.7.

7.2.4 Laboratory Files to Submit

The following files should be submitted for evaluation:

- LinearTrajectory.m
- MakeLinearTrajectory_TaskSpace.m

8 JACOBIAN

The objective of this laboratory is to derive the jacobian and use it to find the linear velocity and the angular velocity of the end-effector. Given the joint velocities, one can use the jacobian to compute the linear and angular velocities of the end-effector. The relationship between joint velocities and end-effector velocities is given by:

$$\begin{bmatrix} v \\ \omega \end{bmatrix} = [J]\dot{q} \tag{8.1}$$

v is the linear velocity vector, ω is the angular velocity vector, J is the jacobian and $\dot{q}$ is a vector of joint velocities.

Topics Covered

- Derive the jacobian
- Use the relationship shown in Equation 8.1 to find linear and angular velocities of the end-effector.

8.1 Pre-Lab Assignments

8.1.1 Derivation of the Jacobian

The jacobian can be easily derived using the forward kinematics analysis you performed in Section 3.1. For an *n*-link manipulator, the jacobian is given by:

$$J = [J_1 J_2 ... J_n]$$

where the ith column corresponds to the i[th] joint. For a revolute joint, the i[th] column is given by:

$$J_i = \begin{bmatrix} z_{i-1} \times (o_n - o_{i-1}) \\ z_{i-1} \end{bmatrix} \tag{8.2}$$

where z_i is the unit vector corresponding to the ith frame and on is the vector from origin of frame o_0 to origin of frame o_n (o_n is the end-effector frame). Similarly o_{i-1} is the vector from o_0 to o_{i-1}. All of these vectors must be expressed in global coordinates.

Note: The vectors z_{i-1} and o_{i-1} are dependent on the position of the end-effector. Thus, the jacobian is also position dependent.

1. Open your Simulink model **Forward_Kinematics.mdl**. Delete all of the displays and disconnect the **Bias** block from the **MATLAB Function** block. Your model should look as shown in Figure 8.1. Save your model as **PreLab_Jacobian.mdl**.

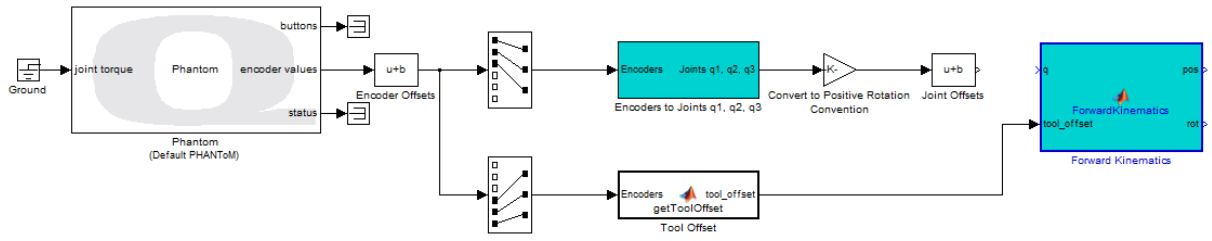


Figure 8.1: Modified forward kinematics model

Question 8.1
From the transformation matrix T_0^i, show how you can find the orientation of axis z_0, z_1 and z_2.

Answer 8.1
The orientation of each of the z_i axes is given by the third column of the T_0^i transformation matrix.

□□□

Question 8.2
From the transformation matrix T_0^i, show how you can find the position of o_i.

Answer 8.2
The position of o_i is given by the fourth column of the T_0^i transformation matrix.

□□□

2. Modify the **MATLAB Function** block to calculate the jacobian as follows:

(a) Although the Omni has two obvious links, for the purposes of calculating this Jacobian, you have to assume that it has a third link between joint 1 and joint 2. This link has zero length. Thus the jacobian will be a 6×3 matrix. Using Question 8.1 above, modify the MATLAB Function block to find the orientation of axis z_0, z_1 and z_2.

(b) Using Question 8.2 above, modify the MATLAB Function block to find the position of o_0, o_1 and o_2.

(c) The position of o_n is given by the output *pos* that you have already calculated. Now you have all the information to calculate the jacobian.

(d) Apply Equation 8.2 to calculate the jacobian. Make this an output of the MATLAB Function block. Your block should look as shown in Figure 8.2.

Figure 8.2: Forward kinematics block with jacobian

(e) Attach a display to the output *jac*.

Question 8.3

Input the vectors shown in Table 8.1 to the input *q* one at a time and record the resultant jacobian matrix. Your model should look as shown in Figure 8.3.

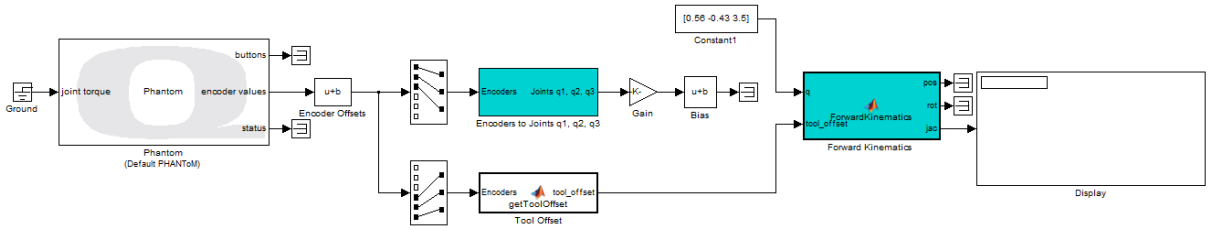


Figure 8.3: Modified model to calculate jacobian.

q (rad)	Jacobian
[-1 -1.3 3.4]	
[0.56 -0.43 3.5]	

Table 8.1: Sample joint positions to calculate the jacobian

8.1.2 Laboratory Files to Submit

The following files should be submitted for evaluation:

- PreLab_Jacobian.mdl

8.2 In-Laboratory Experiment

8.2.1 Linear and Angular Velocity of the End-effector

1. Open **PreLab_Jacobian.mdl** you created in the pre-laboratory. Resave this model as **Jacobian.mdl**.
2. Delete the display on the output *jac* and reconnect the **Bias** block to the input *q* of the **Forward kinematics** block. Your model should resemble Figure 8.4.

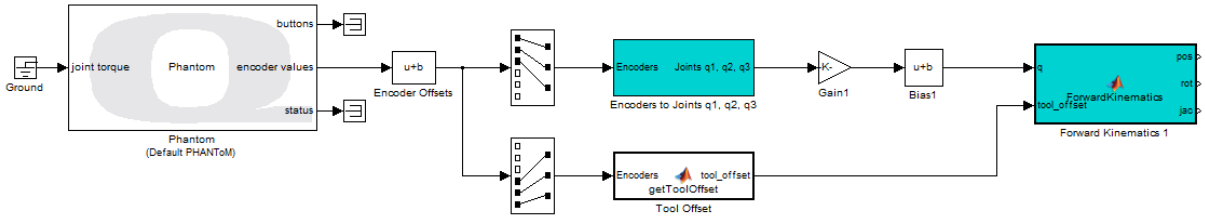


Figure 8.4: Model to calculate the jacobian.

3. In this laboratory, you will be adding several additional blocks to this model. To keep the model simple, it is better to take the existing model and make it into a subsystem before adding further blocks.
   (a) Select all of the blocks in the model.
   (b) While all the blocks are selected, right click on one of the blocks and click Create Subsystem. This command will combine all the blocks into a subsystem and your model will look as shown in Figure 8.5.

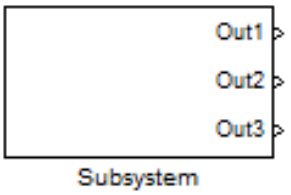


Figure 8.5: Subsystem

   (c) Double click on this subsystem to view the model underneath. Notice the output ports **Out1**, **Out2** and **Out3** attached to the outputs *pos*, *rot*, and *jac* respectively. Rename these output ports to match their respective outputs.
   (d) Attach another output port to the input *q* of the forward kinematics block and name this port *q*.
   (e) Click on the Go to parent system button on the Simulink toolbar to return to the root model. The subsystem should now resemble Figure 8.6.

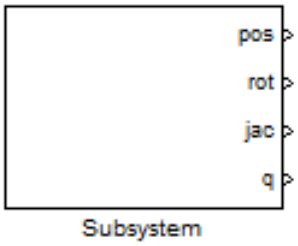


Figure 8.6: Modified subsystem

   (f) Rename the subsystem to Omni

4. From the **QUARC Targets** | **Continuous** library, drag and drop the **Second-Order Low-Pass Filter** block into the model. We will use this block to differentiate the signals. Set the *Cut-off frequency* to **200** Hz, and the *Damping ratio* equal to **1**.

5. According to Equation 8.1, we need to differentiate *q* to get $\dot{q}$. Multiply $\dot{q}$ with *J* to get the linear and angular velocity vectors.

6. Plot the output of this product on two different scopes. Your model should resemble Figure 8.7. To create three inputs to the scope, open the scope, click on *Parameters* and change the *Number of axes* from 1 to 3.

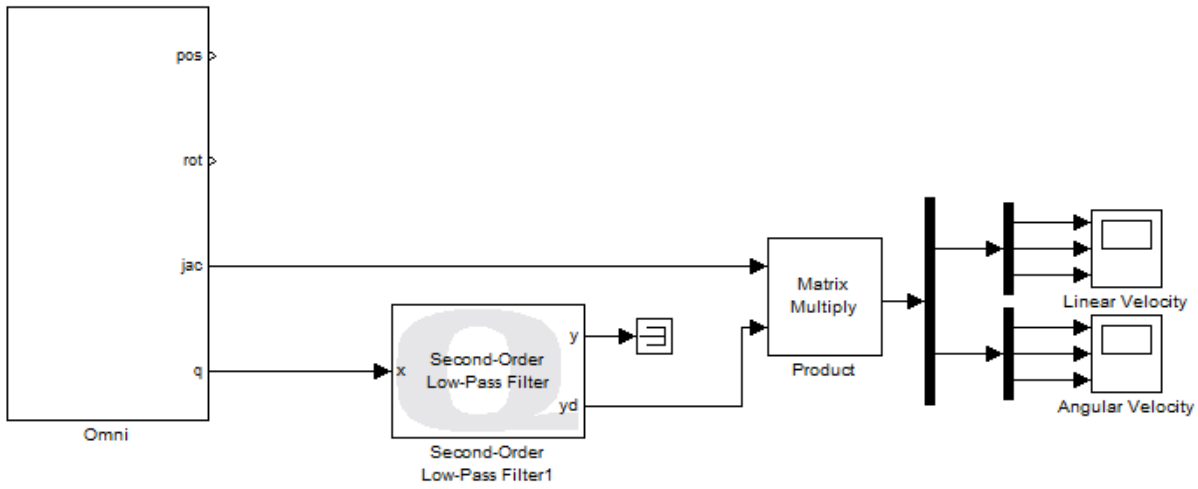


Figure 8.7: Model to get end-effector velocities

7. Set the **Simulation mode** to *External* in the Simulink toolbar.

8. Build and run your model. Move the end-effector around and observe the linear velocities on the scope. The output of the product block is ordered as follows:

$$\begin{bmatrix} v_x \\ v_y \\ v_z \\ \omega_x \\ \omega_y \\ \omega_z \end{bmatrix}$$

9. We can verify the linear velocities by differentiating the output *pos* of the Omni block. Given that this output is the position of the end-effector, its derivative will give us the linear velocity of the end-effector. Modify your model to resemble Figure 8.8.

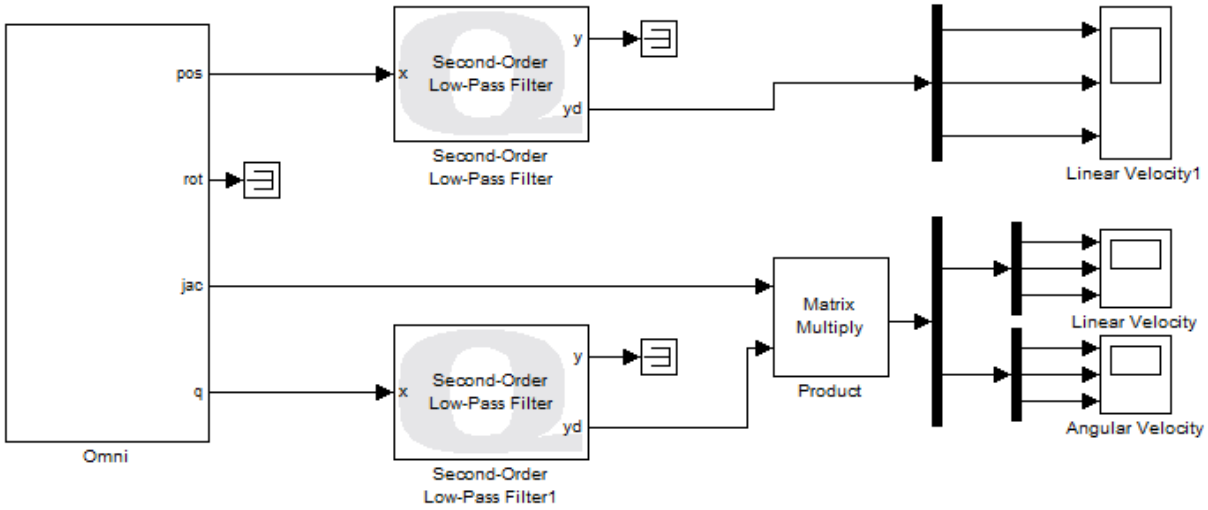


Figure 8.8: Verification of linear velocities

10. Build and run the model.

Question 8.4

Move the end-effector around and record the results from both linear velocity scopes. The velocities on the scopes should match. Record the angular velocity response as well.

Answer 8.3

The measured linear velocities, and the linear velocities calculated using the Jacobian are shown in Figure 8.9 and Figure 8.10. The angular velocities are shown in Figure 8.11.

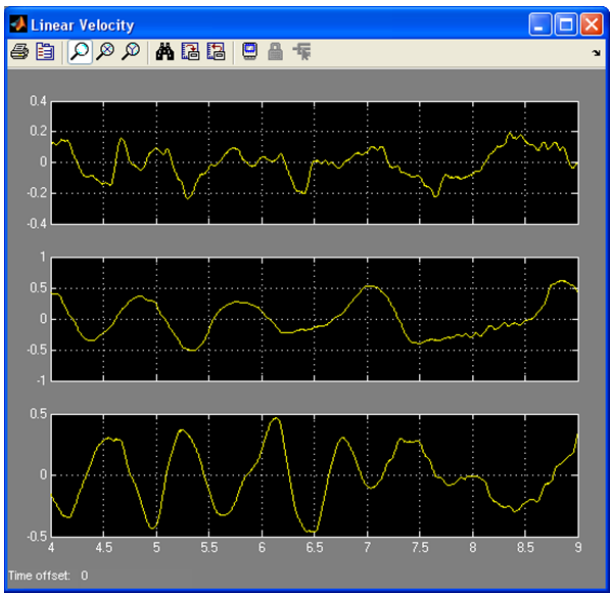


Figure 8.9: Linear velocity result scope 1

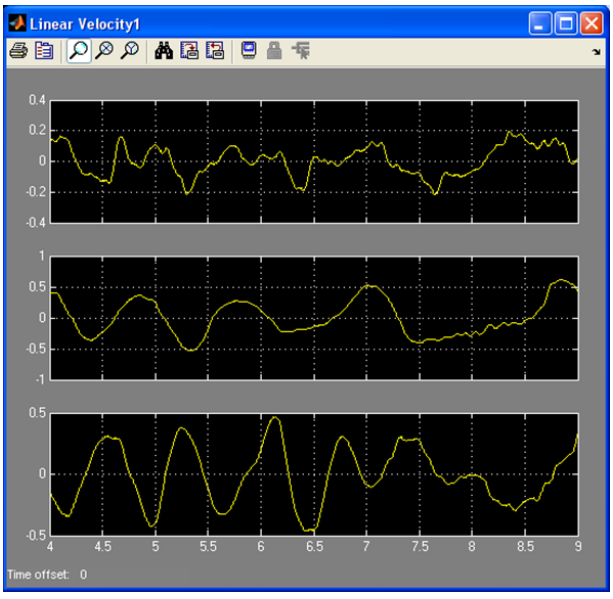


Figure 8.10: Linear velocity result scope 2

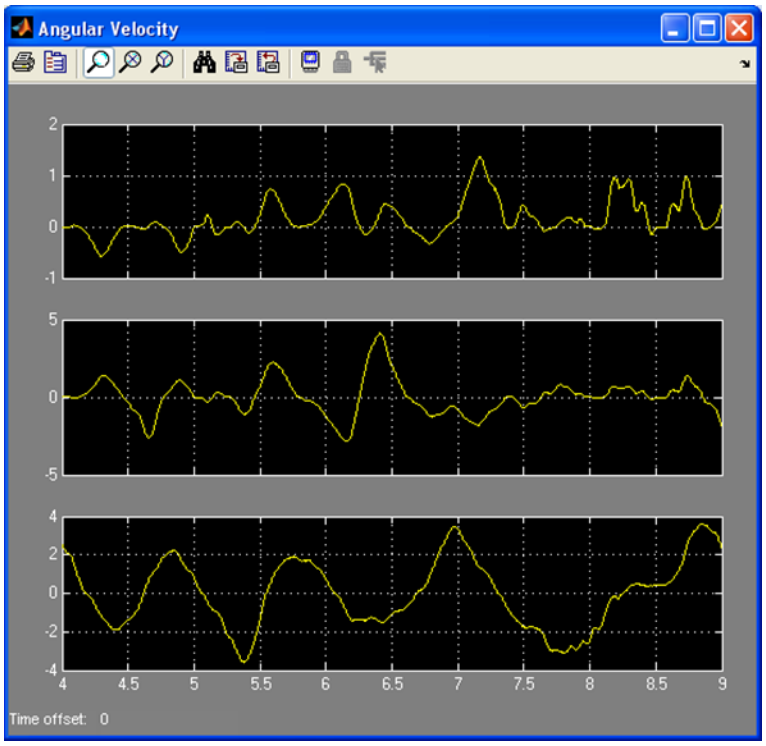


Figure 8.11: Angular velocity result

□□□

8.2.2 Laboratory Files to Submit

The following files should be submitted for evaluation:

- Jacobian.mdl

9 FORCE RENDERING

The objective of this laboratory is to use the jacobian from Section 8 to translate forces in the task space to torques in joint space. This relationship is given by:

$$\tau = J^T F \tag{9.1}$$

τ is the torque vector, J is the jacobian, F is the vector of forces in the task space.

Topics Covered

- Use the jacobian to translate task space forces into joint torques.
- Be able to feel the desired force output at the end-effector of the Omni.

9.1 In-Laboratory Experiment

You will modify the model from Section 8 to calculate joint torques based on the force to be felt at the end-effector.

9.1.1 Applying Forces in Task Space

1. Open **Jacobian.mdl** from Section 8.2.1.
2. Copy only the **Omni** block into a new model and save this model as **Force_Rendering.mdl**.
3. Double click on the Omni block to look underneath.
4. Use the **Math Function** block in Simulink to transpose the jacobian.
5. According to Equation 9.1, J^T is a 3×6 matrix and F must be an array of six elements. However the task space forces are only 3-dimensional. Create an input port for task space force (call it *Task Force*) and use a **Mux** to concatenate it with a zero array of three elements representing dummy task space torques. Now the force can be multiplied with the jacobian transpose to get torques. Your model should resemble Figure 9.1.

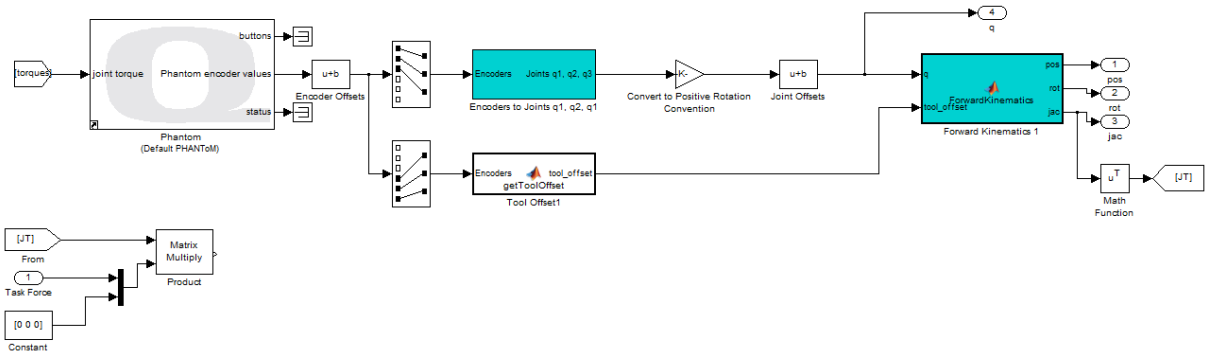


Figure 9.1: Force rendering model

6. Recall that in Section 3.2.1 we attached a gain block to the output of the **Encoders to Joints q1, q2, q3** block to orient the direction of rotation of each joint according to our D-H convention. Now we have to do the same thing to convert it back to the convention used by the Omni block. Create a gain block with the gains [-1;-1;-1] and attach it to the output of the product. Now you have the joint torques in the correct orientation.
7. Attach a unit delay to prevent an algebraic loop.
8. Connect the joint torque port on the **Phantom** block to the output of the **Unit Delay** block. Your model should resemble Figure 9.2.

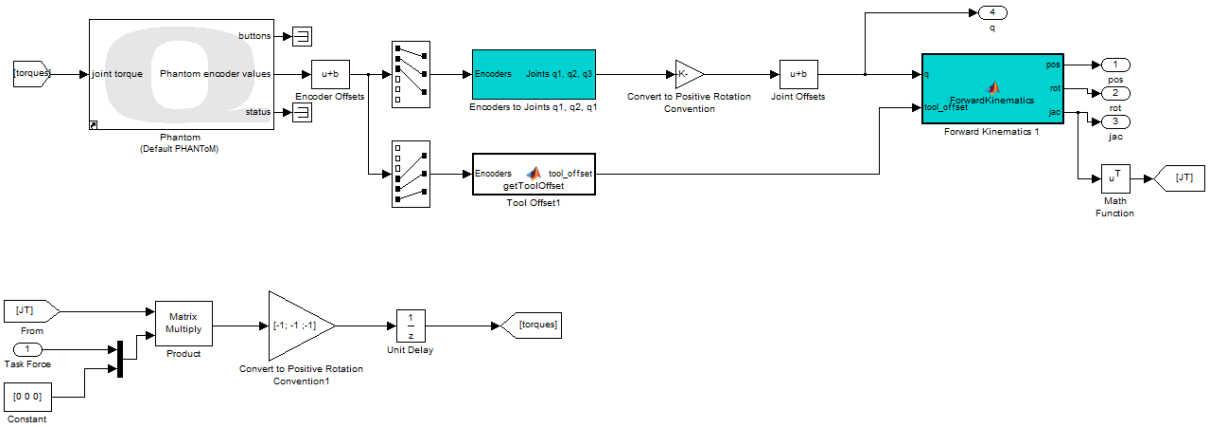


Figure 9.2: Complete force rendering model

9. Go to the parent model. Notice the input node *Task Force* you created earlier. Attach a **Constant** input to this node. Enter in the vector [1 0 0]. Your model should resemble Figure 9.3.

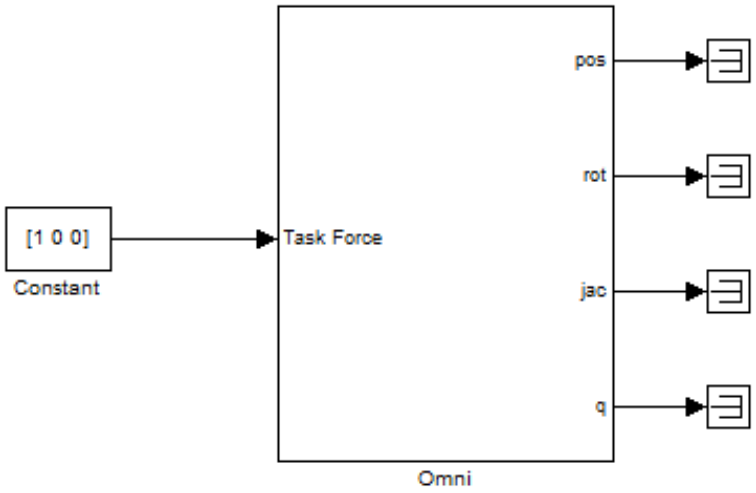


Figure 9.3: Omni model with force input

10. Set the **Simulation mode** to *External* in the Simulink toolbar.

    Note: Before running your model, hold the Omni end-effector in the middle of the workspace. Do not let go of the end-effector while the model is running.

11. Run your model. You should feel the end-effector applying a force equal to 1 N against your hand in the x-direction.

12. Stop the model.

13. Change the value of the **Constant** block to [0 1 0] N. This will apply force of 1 N in the y-direction. Make sure to hold on to the end-effector before running the model.

14. Repeat to apply force of 1 N in the z-direction.

15. Increase the force in any direction to 2 N. You should now feel the Omni applying a larger force in the corresponding direction.

    Note: To prevent damage to the device, do not apply forces larger than 3 N in any direction.

9.1.2 Laboratory Files to Submit

The following files should be submitted for evaluation:

- Force_Rendering.mdl

10 HAPTIC GRAVITY WELL

The objective of this laboratory is to design a haptic gravity well. A gravity well is a single point in 3D space. It is used to attract the haptic device towards the point. Generally the gravity well has a radius of influence in which force feedback is applied to the device to pull it towards the point. Outside this radius, there is no force applied and the device moves freely. This is a common way to find or snap on to points in 3D space. Inside the radius of influence, the force that is applied to pull the device is given by $F = kx - bv$, where k is the stiffness of the spring used to pull the device, x is the vector from the device to the point, b is the damping coefficient, and v is the velocity. In this way, when the device is outside the radius of influence, the force is zero and as soon as it enters the radius of influence, it is pulled towards the point by the force given above.

This laboratory is two-fold: you will first design a virtual environment and then interact with it using your haptic device. You will use the QUANSER Visualization Initialize block to design the graphics and the appropriate Simulink model of your haptic device.

Topics Covered

- Creating a virtual environment for the gravity well.
- Creating a Simulink model of the gravity well.

10.1 Pre-Laboratory Assignments

10.1.1 The Virtual Environment

The virtual environment for this laboratory consists of two objects: The gravity well itself and the avatar. An avatar is a virtual object that tracks the position input of the user; that is, it follows the end-effector position/orientation of the haptic device. Here you will learn to use the QUARC Visualization Initialize block to create virtual environments.

10.1.1.1 The QUARC Visualization

The QUARC Visualization Initialize block is used to create 3D graphical environments. To get started, QUARC includes a five part tutorial on *Creating Your First Visualization* to demonstrate some of the features and basic utilization of the blocks. For the haptic gravity well, we will use existing images and primitives to build the virtual environment as follows:

1. In the **Simulink Library Browser**, go to **QUARC Targets | User Interface | Visualization** and drag and drop the **Visualization Initialize** block into a new model. Save this model as **PreLab_Haptic_Gravity_Well.mdl**.
2. Double-click on this block to open the **Parameters** window as shown in Figure 10.1. See below for a description of each of the labeled sections.

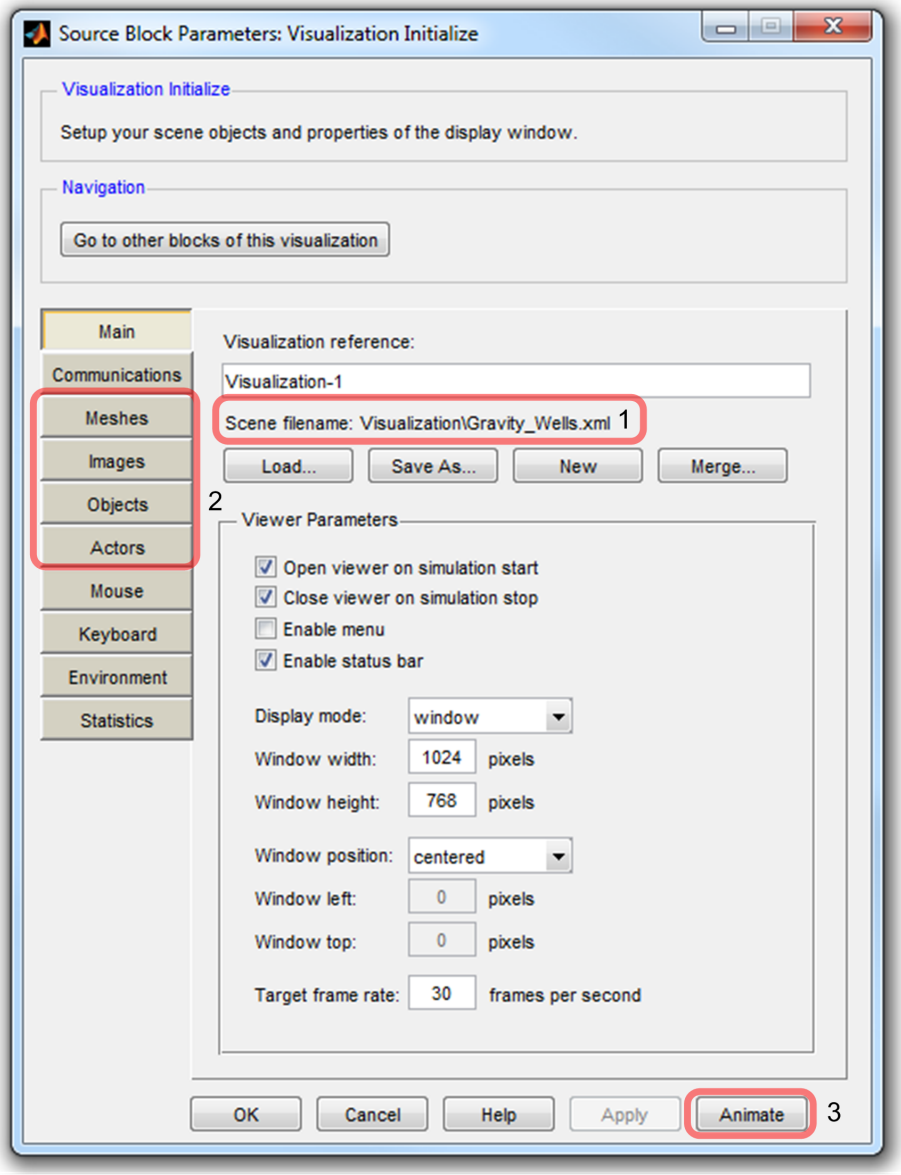


Figure 10.1: Parameters window of Visualization Initialize block

(a) Section 1: Browse to a scene file that defines the graphical world.

(b) Section 2: Add Meshes, Images, Objects, and Actors to the visual environment.

(c) Section 3: Animate the environment to preview the graphical objects.

10.1.1.2 The Avatar

First we will create a spherical avatar. The purpose of this avatar will be to follow the position of the end-effector of the Omni on the screen.

1. In the *Main* parameters window, click *Load* and select the scene file **Visualization | Gravity_Wells.xml**. Click *Open*.
2. Click on the *Meshes* button to open the *Mesh* parameter window shown in Figure 10.2.

3. Click on the *Add* button, and navigate to **Visualization | Meshes | sphere.x3d**. The sphere primitive is a three dimensional model of a sphere created in Solidworks and rendered in 3ds Max. Leave the *Resource identifier* field black and click *Add*.

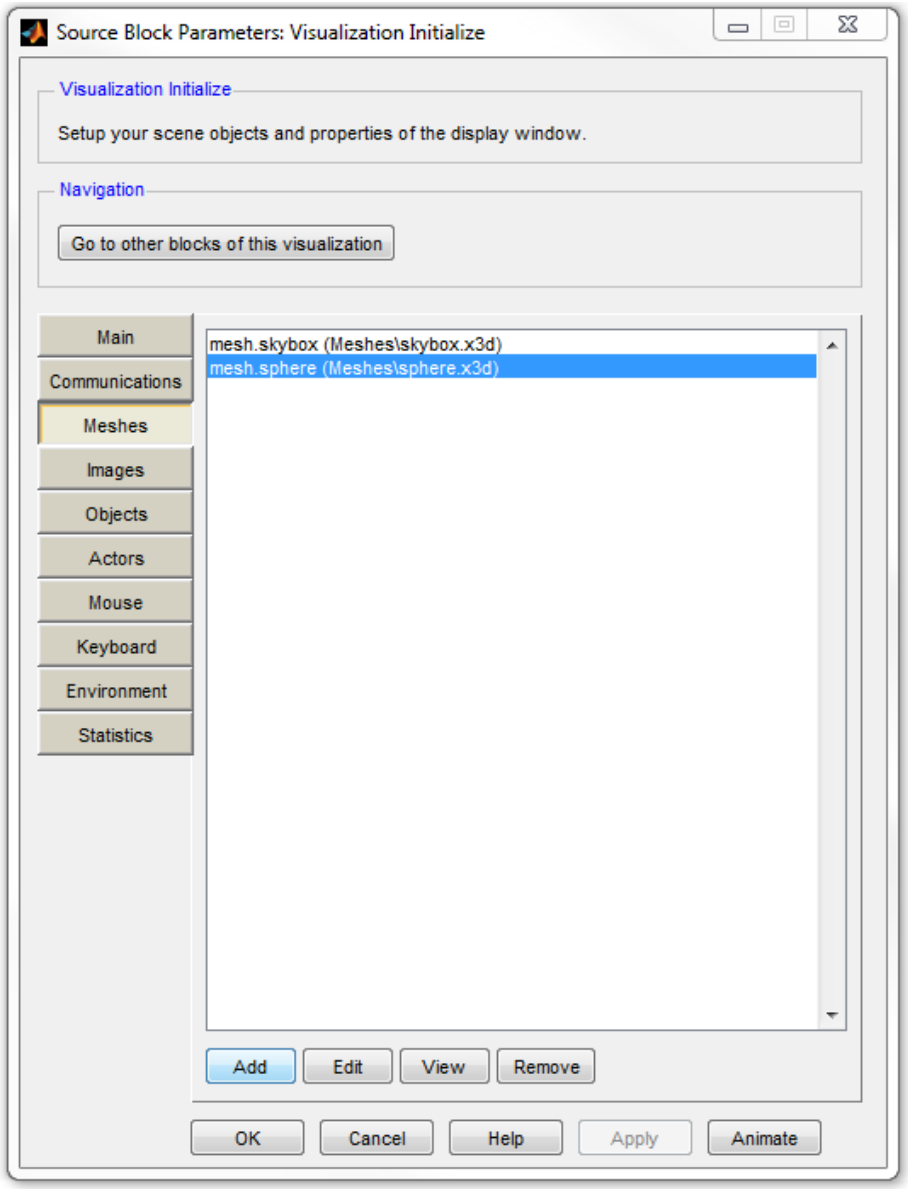


Figure 10.2: Mesh list in Visualization Parameters

4. Next we will add images to add texture to the mesh primitives.
5. Click on the *Images* button and click on the *Add* button.
6. Add the image **Visualization | Images | yellow.png**. This will be the texture of the avatar sphere.
7. Next add the texture for the gravity well located at **Visualization | Images | moon_surface.png**.
8. Navigate to the *Objects* parameter window. Objects bring together textures and meshes to create complete 3D visual elements.
9. Click Add, and select the Mesh Resource mesh.sphere and texture image.yellow. Specify the resource identifier object.avatar, and click Add.
10. Repeat Step 9 to create and a *moon_surface* textured sphere called **object.well**.

10.1.1.3 Actors

The last step to creating a visual environment is to create actors using the objects we added in Section 10.1.1.2. The actors attribute 3D properties to the objects to specify their location and behaviour in space.

1. Navigate to the *Actors* parameter window. Notice that the location of the camera, the background "skybox", and light source are already defined.
2. We will begin by creating the avatar actor. Click on *Add*, and select the object resource **object.avatar**.
3. Leave the parent actor as **None**, and the position and orientation as **[0 0 0]**.
4. The locations and sizes of actors in the visual environment should be scaled by $\times 10$ to make changes in their location more obvious. Specify the scale property of the avatar as **[0.2 0.2 0.2]** to create a sphere **0.02** m in diameter.
5. Make sure to set the *Render priority* to **1** so that the avatar is always visible when inside a gravity well. Click *OK*.
6. Repeat Steps 2-5 to create the gravity well actor located at **[-1.5 0 0]**, with a scale of **[0.6 0.6 0.6]**, and render priority of **0**.
7. Click *Apply* to complete the scene.
8. Click on the *Animate* button to preview the environment. The scene should resemble Figure 10.3. Click *OK* to close the *Visualization Initialize properties* window.

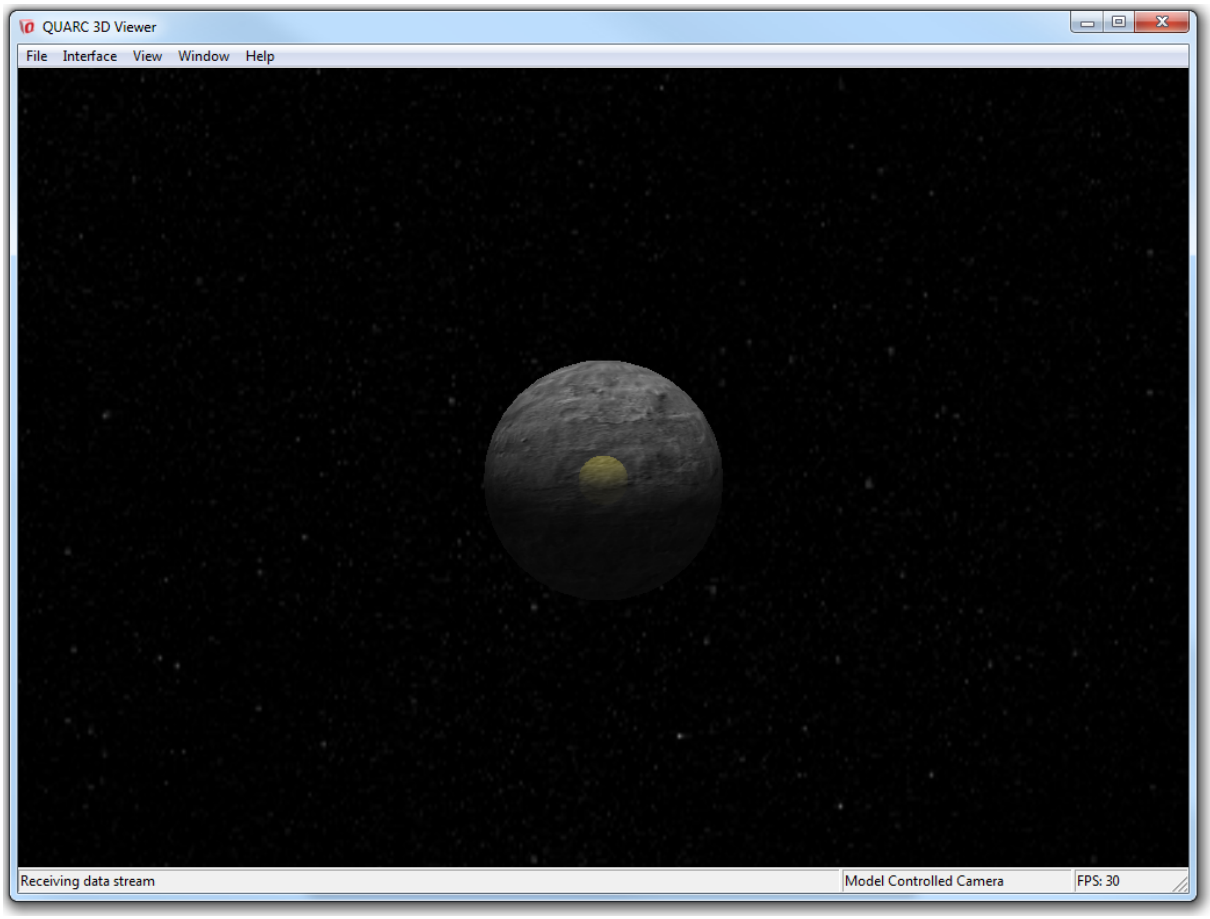


Figure 10.3: Virtual world animation

9. Add a **Visualization Set Variables** block to the model from **QUARC Targets | User Interface | Visualization** in the Simulink Library Browser. This block will allow you to modify the parameters of the actors in the scene.
10. Open the *Visualization Set Variables properties* window, and click on the  button.
11. Navigate to the **actor.well** variable, select the *Position* property and click on the  button to add it to the list of selected variables.
12. Repeat Setp 11 to add the position variable for the avatar. Click *Apply* and then *OK*.

10.1.2 Laboratory Files to Submit

The following files should be submitted for evaluation:

- GravityWell.xml
- PreLab_Haptic_Gravity_Well.mdl

10.2 In-Laboratory Experiment

In this laboratory, you will interface the Omni with the virtual world you created in the pre-laboratory.

10.2.1 Interfacing with the QUARC Visualization

1. Open **Haptic_Gravity_Well.mdl**. This model contains an **OMNI** block that is similar to the block you built in Section 9.1, but models the Omni using all 6 joints.
2. Ensure that the stylus is detached from link 2 by removing the stylus lock. In this laboratory, it will be easier to handle the end-effector in this configuration.
3. From your **PreLab_Haptic_Gravity_Well.mdl**, drag and drop the **Visualization Initialize** and **Visualization Set Variables** blocks into this new model. Your model should look resemble Figure 10.4.

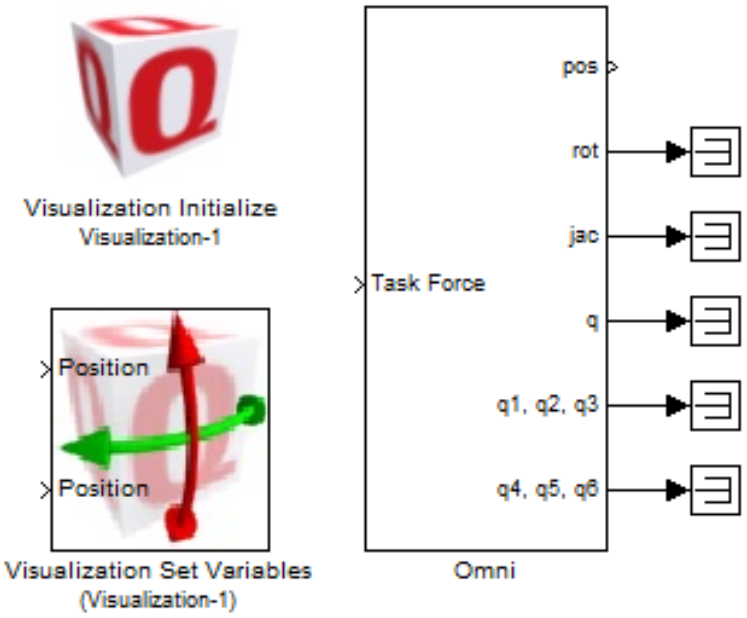


Figure 10.4: Full Omni block including joints 4, 5 and 6

4. We want the position of the end-effector to be mirrored by the position of the avatar on the screen. The position of the end-effector is given by the output *pos* of the OMNI block, and the position of the avatar can be specified using the **Visualization Set Variables** block. However, we cannot connect them directly because the coordinate frame used by the Omni is not the same as the coordinate frame used by the QUARC Visualization.
5. Recall from Section 3.1.1, the global coordinate frame defined by our D-H parameters is given by Figure 10.5. The coordinate frame used by the QUARC Visualization is shown in Figure 10.6.

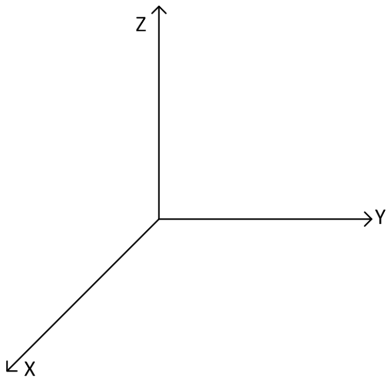


Figure 10.5: Omni coordinate frame

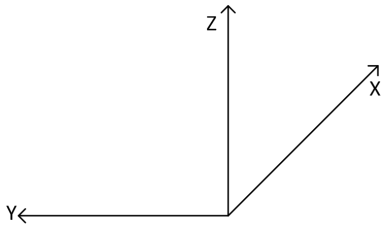


Figure 10.6: Visualization coordinate frame

6. Make a subsystem block that converts the position output of the Omni in the coordinate frame shown in Figure 10.5, to the position of the avatar in the coordinate frame shown in Figure 10.6. To do this, simply add a gain block to reverse the appropriate axes and scale the output to the visualization by a factor of $\times 10$. Call this block **Coordinate Transform**. Your model should resemble Figure 10.7.

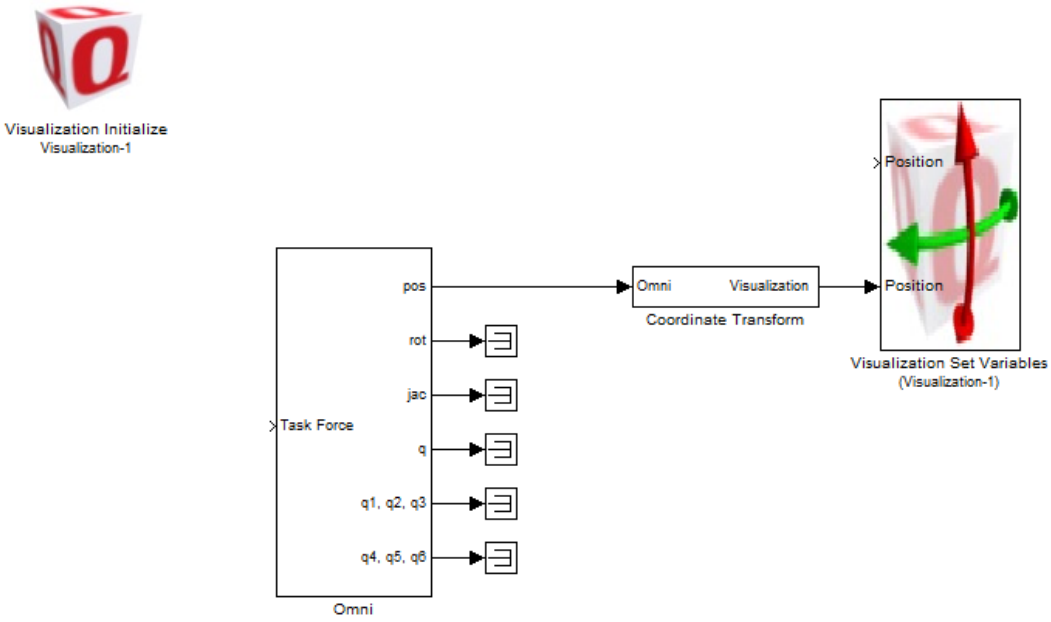


Figure 10.7: Avatar coordinate transformation

7. Recall that in Section 3.2.3, you determined the workspace of the Omni. It is important to ensure that we place the gravity well within a reachable range of the end-effector in all directions. Remember that the radius of the gravity well is 0.03 m. So if we place the gravity well at [0.15 0 0], then it will span the x direction from 0.12 m to 0.18 m and in the y and z directions from -0.03 m to 0.03 m. These are well within the reachable space of the device.

8. Set the position input of the **Visualization Set Variable** block for the wells avatar to **[0.15 0 0]**. Once again note that this is the position given in the coordinate frame of the Omni not the **QUARC Visualization** block. Use the **Coordinate Transform** block to transform this input to the Visualization coordinates before using them.

9. Now the right hand side of the model is complete. Set the *Task Force* input to **[0 0 0]**. Your model should now resemble Figure 10.8.

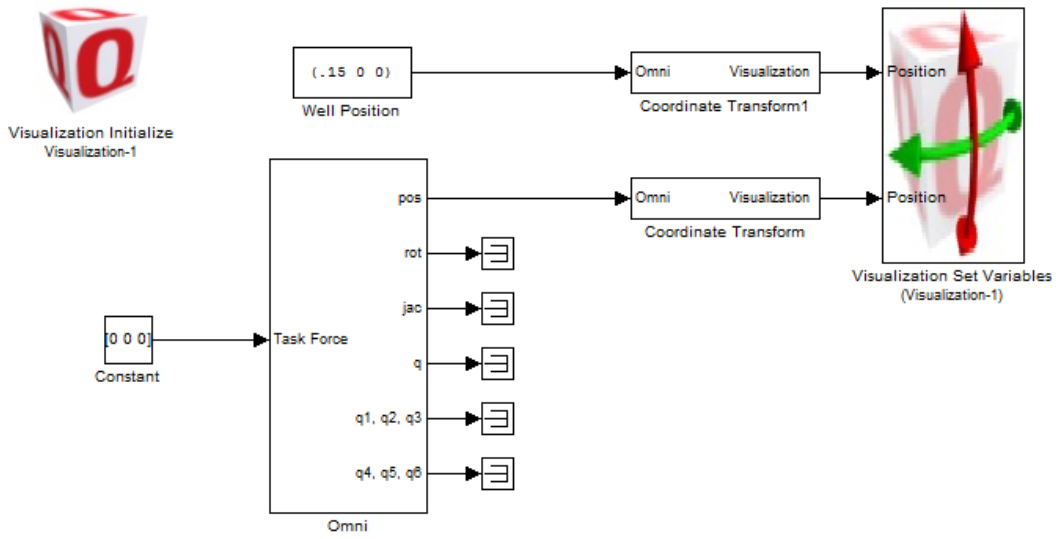


Figure 10.8: Visualization variables configured

10. Set the **Simulation mode** to *External*.
11. Build and run this model. Move the end-effector around. You should observe the avatar following the position of the end-effector.

10.2.2 Modeling Forces

The gravity well is modeled as a piecewise continuous function. If the position of the avatar is within the radius of influence, the avatar is attracted towards the point with a spring-damper force. Otherwise, the force is set to zero.

Recall that we set the diameter of the gravity well to 0.06 m in the visualization. Therefore, the radius of influence will be 0.03 m. As soon as the avatar comes into contact with the gravity well, it should be pulled in towards the center.

1. The gravity well force can be modeled as follows:
   (a) Let $E = p_{well} - p_{avatar}$ where E is the Error, p_{well} is the well position, and p_{avatar} is the avatar position.
   (b) If the magnitude of the error, E, is less than the radius of influence set:

$$F = kE - bv,$$

   otherwise set $F = 0$ where v is the velocity, k is the spring stiffness, b is the damping, and F is the force. The velocity, v, is calculated by differentiating the error. Remember to use a **Second-Order Low-Pass Filter** block to differentiate the signal.
   (c) Recall that we placed the well at [0.15 0 0] m. These are the coordinates for the well position, p_{well}. Avatar position, p_{avatar} is the position of the end-effector taken from the output pos of the **OMNI** block.
   (d) Use k = 100 N/m and b = 0.01 N s/m
2. Create a Simulink subsystem to implement Steps 1a through 1d above. This model should have the following inputs and outputs:
   - **Inputs**
     - *well pos*: Position of the well measured in meters
     - *avatar pos*: Position of the avatar measured in meters
   - **Outputs**

- *force*: Force of the well measured in Newtons

3. Attach a scope to the output force. Your model should resemble Figure 10.9.
   Note: Do not connect the gravity Well model to the OMNI block yet.

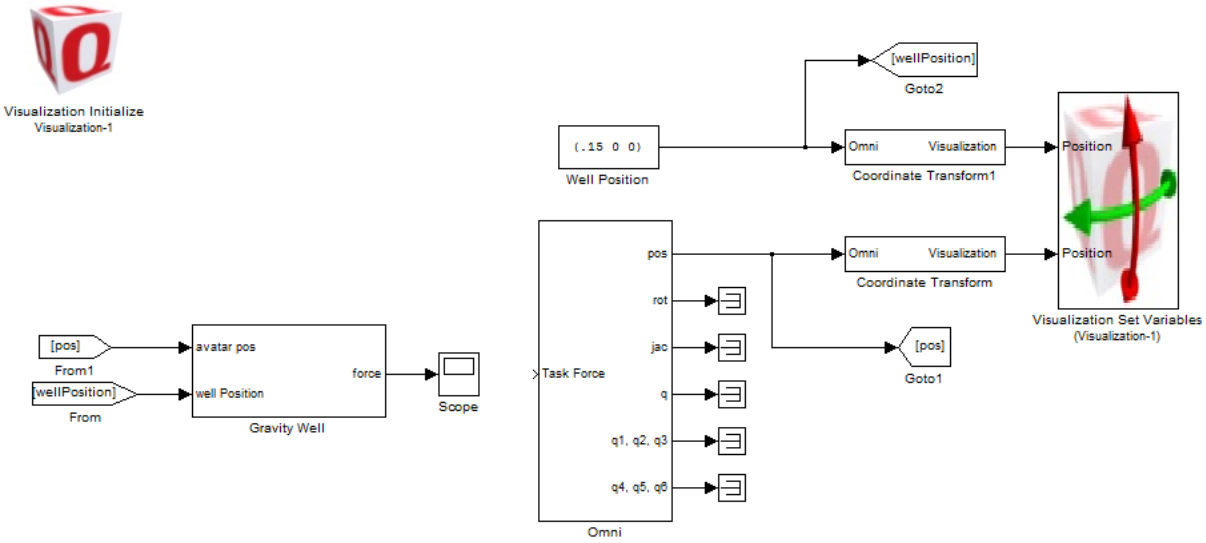


Figure 10.9: Gravity well model

4. Set the **Simulation mode** to *External* in the Simulink toolbar.
5. Build and run the model.
6. Open the scope and check if the forces make sense. Outside the gravity well, the forces should be zero. Inside, they should be in the direction towards the center of the gravity well. The magnitude of the forces in each of x, y and z directions should not exceed 3 N.
7. Once you have verified that the model is behaving correctly. Connect the output *force* to the *Task Force* input of the **OMNI** block.
8. Build and run the model. Now you should be able to feel the forces applied by the gravity well.

   Question 10.1
   You may notice that for the model to recognize that the avatar is touching the gravity well, it looks for the intersection of the center of the avatar and the outline of the gravity well. This is why the avatar enters the gravity well half way before you begin to feel the attractive force. How would you change the model in order to pull the avatar in towards the center as soon as the outline of the avatar touches the gravity well?

   Answer 10.1
   If the magnitude of the Error is less than the radius of influence combined with the radius of the avatar, set $F = k\,E - bv$, otherwise set F = 0.

   □□□

9. Implement the updated error calculation.

10.2.3 Laboratory Files to Submit

The following files should be submitted for evaluation:

- GravityWell.xml
- Haptic_Gravity_Well.mdl

10.3 Bonus

To complete the Bonus section, you will be provided with **Teach_Points_Full.mdl**. This model is similar to **Teach-Points.mdl** used previously, except it outputs the position of the end-effector based on all six joints and not just the first three. Since the stylus is no longer fastened to link 2, the tool position is no longer constant. *Tool_offset* varies as a function of joints 4 and 5. This model performs a full forward kinematic analysis that includes all 6 joints.

The bonus task is to use this file to click points in space where the gravity wells should be created. You are given a MATLAB script to automatically generate the virtual environment for you based on the number of gravity wells you create. Your challenge is to create a **MATLAB Function** block to model the gravity wells.

1. Open **Teach_Points_Full.mdl**.
2. Build and run the model. While the model is running, click at least 3 points in space to record their positions. Stop the model once you are finished.
3. Open **MakeGraphics.m**. This file looks at how many points you clicked and modifies the **Gravity_Wells_Bonus.xml** file to add the additional gravity wells to the scene.
4. Run the script.
5. Create a MATLAB Function block to model the forces from all the gravity wells. The function should be designed as follows:
   - **Inputs**
     - *wellPos*: An $n \times 3$ vector that accepts the position of each well in meters. n is the number of wells created by **Teach_Points.mdl**.
     - *avatarPos*: Position of the avatar measured in meters
     - *vel*: Velocity of the end-effector (for use calculating $F = kx + bv$) measured in m/s.
   - **Outputs**
     - *force*: Force of the gravity wells measured in Newtons
   - Write the function to handle unknown number of gravity wells. You can do this by checking how many gravity wells there are and then calculating the forces from all of those gravity wells. At the end, you can simply add all of those forces to obtain the net force on the avatar.

     The radius of each gravity well will be 0.03 m.

     Use k = 100 N/m and b = 0.01 Ns/m
6. Add a **Visualization Initialize** block and **Visualization Set Variable** block and configure the visualization to use the **Gravity_Wells_Bonus.xml** scene file. Your model should resemble Figure 10.10.

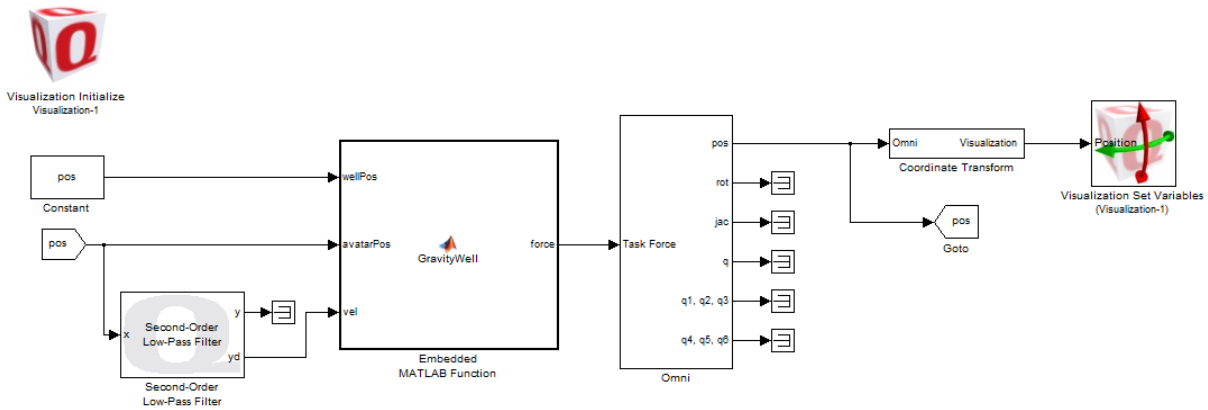


Figure 10.10: Multiple gravity well model

7. Save your model as **Bonus_Haptic_Gravity_Well.mdl**.

10.3.1 Laboratory Files to Submit

The following files should be submitted for evaluation:

- Bonus_Haptic_Gravity_Well.mdl

11 HAPTIC WALL

The objective of this laboratory is to create virtual wall contacts. When the avatar comes in contact with the wall, repulsive forces are applied to push the avatar away from the wall. The force applied by the wall is a spring-damper force given by $F = kx - bv$, where k is the stiffness of the spring used to push the avatar, x is the vector from the avatar to the wall, b is the damping coefficient and v is the avatar velocity. In this way, when the avatar is outside the wall, the force is zero and as soon as it begins to penetrate the wall, it is pushed outwards by the force given above.

Topics Covered

- Creating a Simulink model to implement a virtual wall parallel to the x-y plane.

11.1 In-Laboratory Experiment

11.1.1 Modeling Forces

Recall the global coordinate frame of the Omni, shown in Figure 11.1.

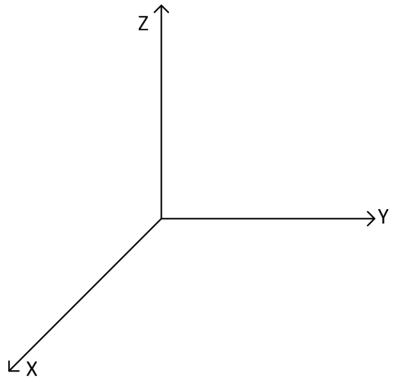


Figure 11.1: Omni coordinate frame

To model a wall parallel to the x-y plane, we require only the value of the z-coordinate of the wall. Figure 11.2 shows the cross-section of the wall from the front.

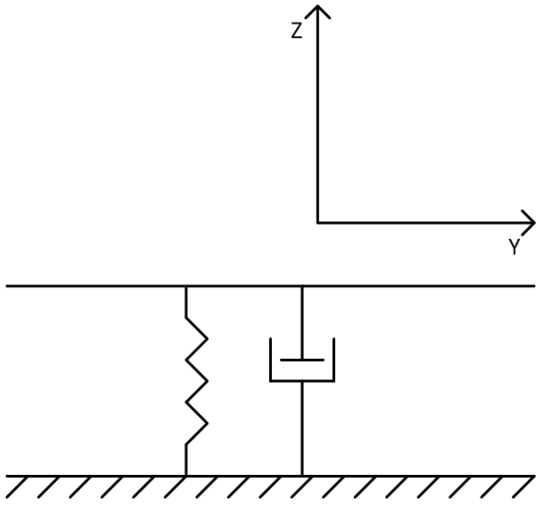


Figure 11.2: Cross-section of the wall

1. The wall force can be modeled as follows:

   (a) $E = z_{wall} - z_{end}$, where E is the error, z_{wall} is the z position of the wall, and z_{end} is the z position of end-effector.

   (b) If the error is greater than zero, set:

   $$F = kE - bv,$$

   otherwise set F = 0, where v is the velocity, k is the spring stiffness, b is the damping, and F is the force. The velocity, v, is calculated by differentiating the error. Remember to use a **Second-Order Low-Pass Filter** block to differentiate the signal.

   (c) Locate the wall at z_{wall} = -0.05 m.

   (d) Use k = 300 N/m and b = 0.01 Ns/m

2. Open **Haptic_Wall.mdl** and note the full kinematic block **OMNI**. This model contains an **OMNI** block that is similar to the block you built in Section 9.1.

3. Create a Simulink subsystem to implement Steps 1a through 1d above. This model should have the following inputs and outputs:

   - **Inputs**
     - *pos*: Position of the end-effector in meters
     - *wall pos*: Position of the wall measured in meters
   - **Outputs**
     - *force*: Force of the wall measured in Newtons

   This model is shown in Figure 11.3.

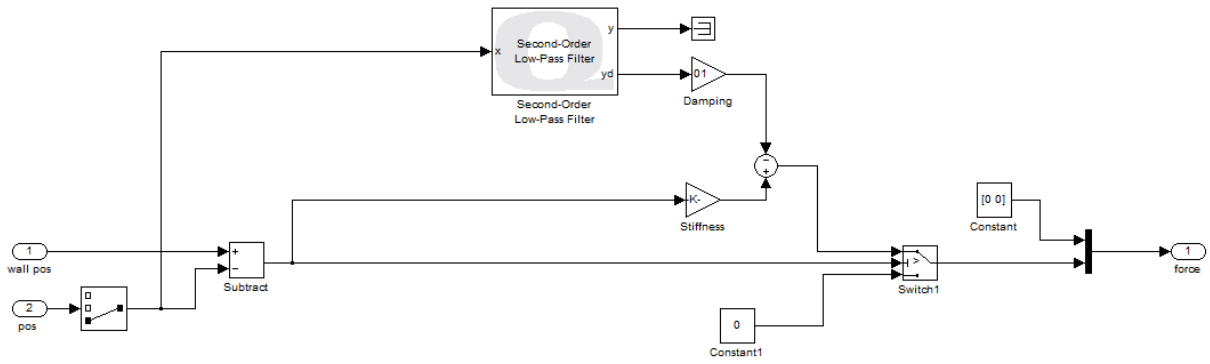


Figure 11.3: Wall force model subsystem

4. Attach a scope to the output force. Your model should resemble Figure 11.4.
   Note: Do not connect the haptic wall model to the OMNI block yet.

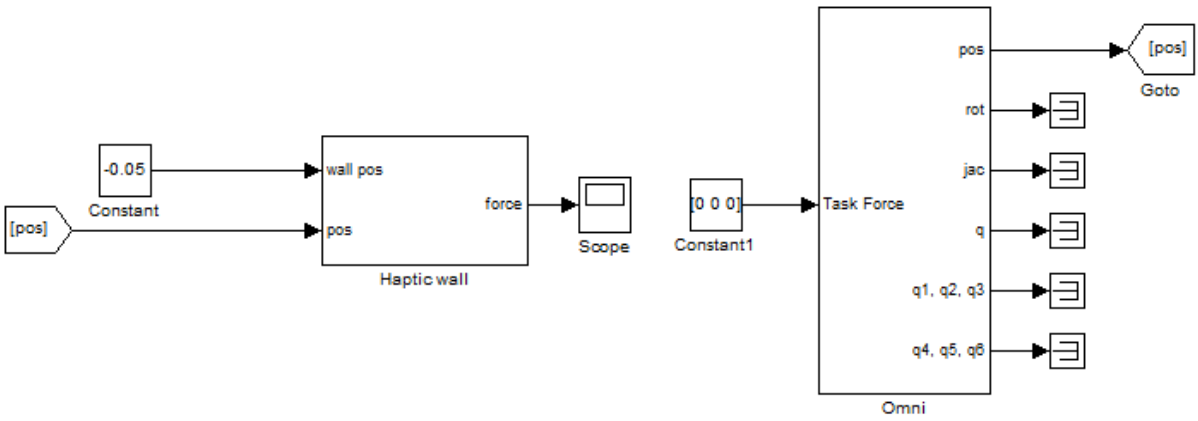


Figure 11.4: Haptic wall model

5. Set the **Simulation mode** to *External* in the Simulink toolbar.

6. Build and run the model.

7. Move the end-effector of the Omni up above z_{end} = 0.05 m to exit the wall and bring it down below z_{end} = -0.05 m to penetrate the wall. Open the scope and check if the forces make sense. The x and y forces should always be zero. Away from the wall, the z force should be zero. As you begin to penetrate the wall, the force should be applied in the positive z direction. Even at maximum penetration, the magnitude of this force should not exceed 24 N.

8. Once you have verified that the model is behaving correctly. Connect the output *force* to the *Task Force* input of the **OMNI** block.

9. Build the model.
   Note: Before running your model, hold the end-effector of the Omni in a position above z_{end} = -0.05 m. This is needed to ensure that when the model starts, there is no initial penetration of the wall.

10. Run the model. Now you should be able to feel the forces applied by the wall.

11.1.2 Laboratory Files to Submit

The following files should be submitted for evaluation:

- Haptic_Wall.mdl

11.2 Bonus

The bonus challenge is to implement a haptic puncture through wall. Push against the wall until a threshold force is reached, at which point the wall breaks and the avatar is allowed to pass freely to the other side of the wall. Once the avatar is on the other side of the wall, the wall restores itself to resist the avatar going back through and the same scenario repeats.

Implement the wall contact force as a spring only. No damper is required.

You may implement the code to calculate the force in a **MATLAB Function** block.
Start with the model **Bonus_Haptic_Pucture_Through_Wall.mdl**.

11.2.1 Requirements

- The wall has to be implemented in the x-y plane of the Omni coordinate frame.
- The wall should be located at z_{wall} = 0 m and it must have thickness = 0.02 m.
- When the avatar is on either side of the wall, the force should be zero
- When the avatar is inside the wall or touching the wall, the force should be calculated using a spring model: $F = kx$. Where k is the spring stiffness and x is the vector from the avatar to the wall.
- The force should always be a repelling force. So if the avatar approaches the wall from the top direction, the force should be in the positive z-direction and if the avatar approaches the wall from the bottom direction, the force should be in the negative z-direction.
- The wall should break to let the avatar pass through to the other side when the threshold force is reached. Once the avatar is on the other side, the wall should become solid again.
- If the avatar pushes against the wall but retreats before the threshold force is reached, the wall should remain solid and the avatar should remain on the same side of the wall it was before. If it loses contact with the wall, the force should become zero.
- Use k = 100 N/m, with a threshold force of 3 N

11.2.2 Hints

- Create a variable called *touching* that keeps track of weather the avatar is touching the wall or not. *touching* = 1 if the wall is being touched, *touching* = 0, if the avatar is not in contact with the wall.

  Think of the wall as having elastic properties. It may be possible for the avatar to stretch the wall such that its position is on the other side of the wall without the wall breaking. For instance, if the avatar approaches the wall from the top and begins to push against it, the threshold force will not be reached until the avatar is 0.01 m below the wall (the thickness of the wall is 0.02 m. therefore $F = 100(0.02 + 0.01)$ = 3 N (threshold force)
- Calculate the force based on whether or not the avatar is touching the wall, which side of the wall it is on, and weather the wall is broken or not.
- Create a variable called *dir* that keeps track of which side of the wall the avatar is on. *dir* = 1 if the avatar is on the top side, *dir* = -1 if avatar is on the bottom. Keep in mind that the side that the avatar is on changes only when the wall is broken.
- Create a variable called *broken* that keeps track of weather the wall is currently broken or not. *broken* = 1 when the wall is broken, and 0 otherwise. By default, *broken* should equal 0. *broken* should have value 1 only during the length of time in which the threshold force is reached *and* the avatar comes out on the other side of the wall. In other words, if the threshold value is reached, *broken* should equal 1 and it should remain 1 until the position of the avatar is strictly on one side of the wall. If the wall breaks and the avatar sits inside the broken wall, the status of the wall should remain broken.

- If you need to know the value of a signal in the previous sample time, use the **Unit Delay** blocks as shown in Figure 11.5. Inside this block, set the *Initial condition* to be the value(s) that you want the signal to have initially.

11.2.3 Model

Your model may resemble Figure 11.5, but keep in mind this model only shows one solution to implement this behavior. There may be other solutions.

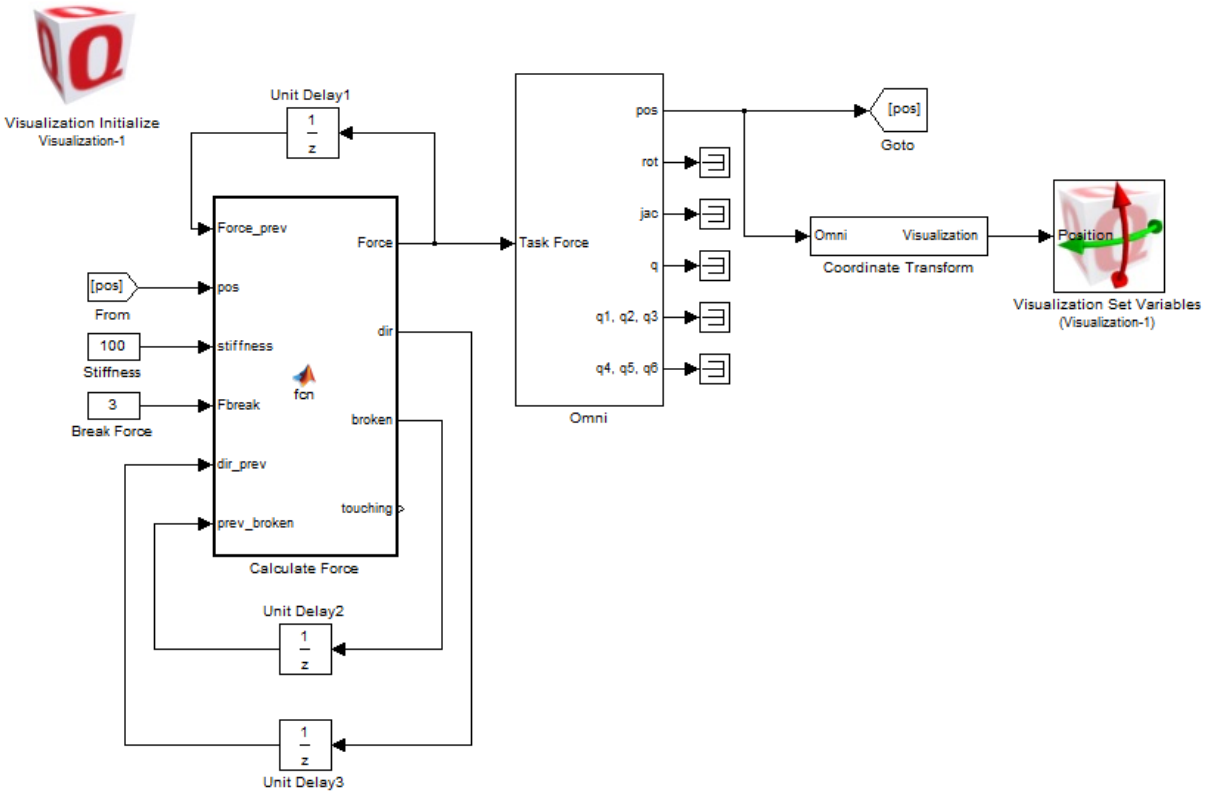


Figure 11.5: Example model to implement haptic wall puncture

11.2.4 Laboratory Files to Submit

The following files should be submitted for evaluation:

- Bonus_Haptic_Puncture_Through_Wall.mdl

12 HAPTIC PONG

The objective of this laboratory is to create a haptic game of Pong. You will be given a partial model of the game that implements the game in 1D. Your task will be to modify the model to make it a 2D game. Pong consists of 3 objects: a ball, a paddle, and 3 walls. The position of the paddle is controlled by the position of the end-effector of the Omni. The purpose of the game is to hit the ball to bounce it off of the walls. The game is over if the ball falls below the paddle.

Topics Covered

- Modifying a 1D game of Pong to a 2D game.
- Learning how to model dynamic objects.
- Using the spring-damper model from earlier exercises to apply forces to the ball and the paddle.

12.1 Pre-Laboratory Assignments

12.1.1 Studying the Model

In this section, you will be given an overview of the existing model of the game. The game is implemented in 1D, meaning the ball can travel in the vertical direction only. Contact forces from the top wall and the paddle are used to model the physical interaction between the paddle and the ball.

1. Open **Pong1D.mdl**. Four major parts of **Pong1D.mdl** are highlighted in Figure 12.1, and outlined below:

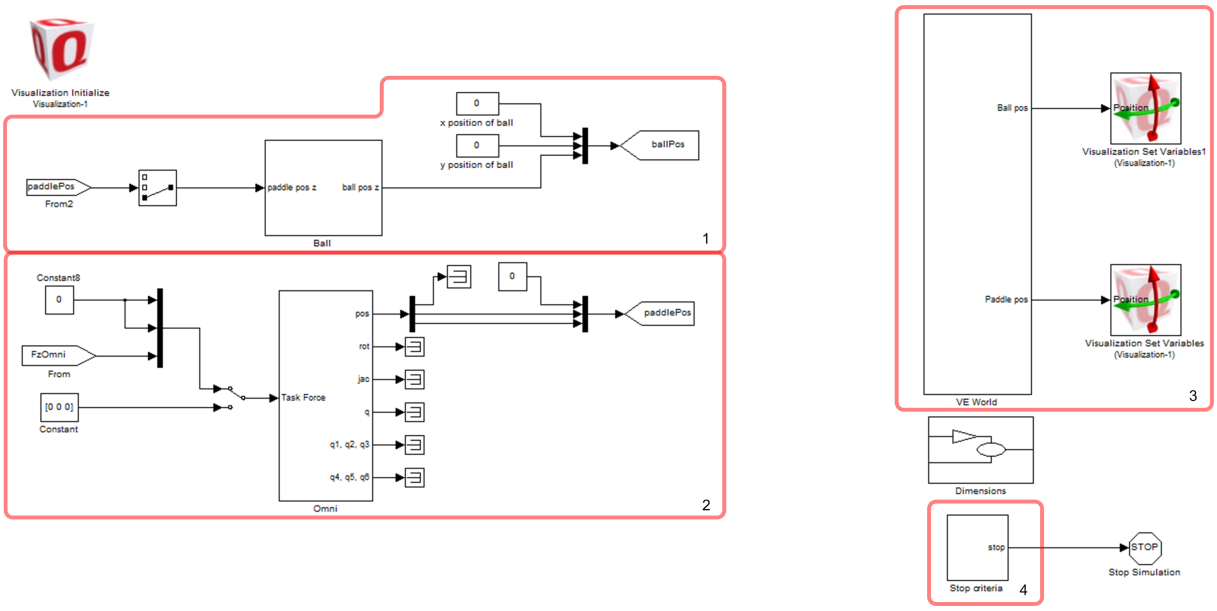


Figure 12.1: 1D Pong model

(a) Section 1: Section 1 models the ball. It looks at the position of the ball relative to the top wall and the paddle. If the ball is in contact with either of these objects, the blocks under this section calculate the force on the ball and the paddle. A dynamic model of the ball is setup inside the **Ball** subsystem.

(b) Section 2: Section 2 contains the **OMNI** block to apply forces and read position of the end-effector.

(c) Section 3: The blocks under section 3 model the virtual environment. They define the parameters of the simulation, and update the position of the paddle and ball in the virtual world. The model under the VE World subsystem ensures that visually, the ball does not appear to penetrate the walls even though this is the actual case. Depending on the stiffness of the walls, the ball does penetrate into the walls, however the position of the ball is saturated to stay within the walls and above the paddle whenever the ball hits these objects.

(d) Section 4: The blocks under section 4 ensure that the simulation stops when the ball falls 0.03 m below the paddle.

2. Double click on the block labeled **Ball** to open the subsystem. The model is shown in Figure 12.2

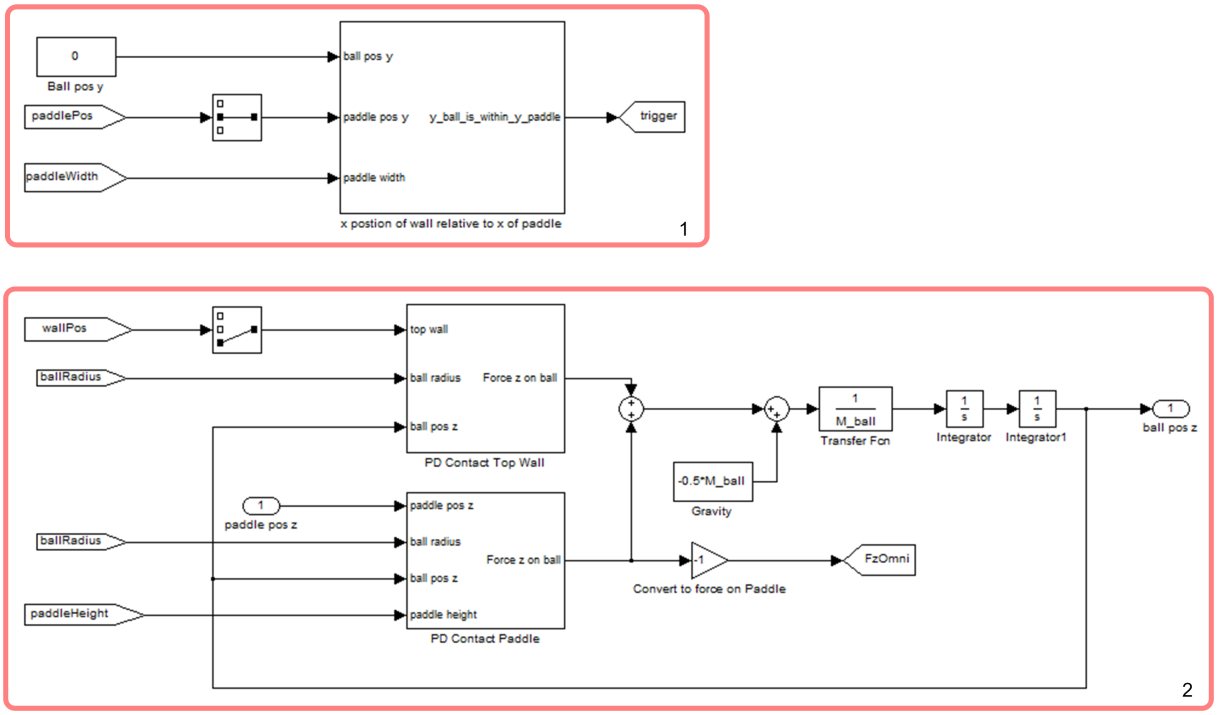
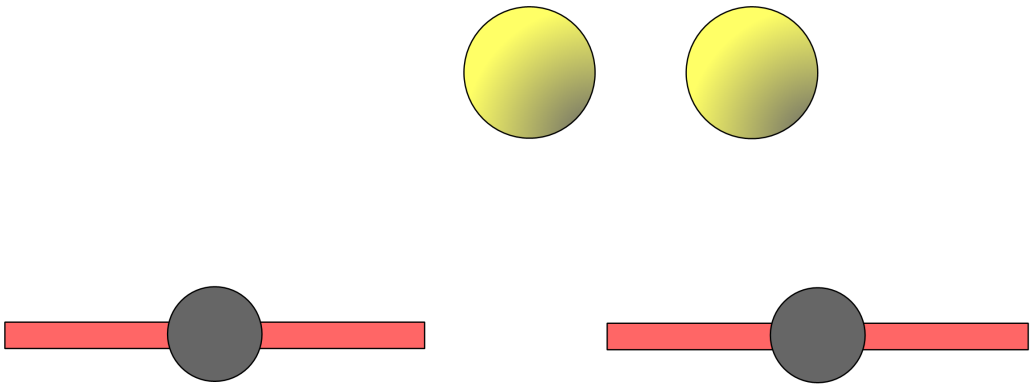


Figure 12.2: Ball model

(a) Section 1: The blocks under section 1 look at the horizontal position of the ball with respect to the horizontal position of the paddle. If the ball is within the paddle's reach, the output *trigger* is 1 else it is 0. This output is used later to determine if the ball is touching the paddle. The output *trigger* states are shown in Figure 12.3.



(a) *trigger* = 0 (b) *trigger* = 1

Figure 12.3: *trigger* states depending on the horizontal position of the paddle

(b) Section 2: The blocks under section 2 model the dynamics of the ball in the vertical direction only. The dynamic equation of the ball is given by:

$$F_z = m\ddot{z} \tag{12.1}$$

Where F_z is the sum of all forces acting on the ball in the vertical direction, m is the mass of the ball and $\ddot{z}$ is the acceleration of the ball in the vertical direction.

F_z consists of the following forces:

$$F_z = F_{z_{paddle}} - F_{z_{topwall}} - mg \tag{12.2}$$

$F_{z_{paddle}}$ is the force applied on the ball when it hits the paddle. This contact force is calculated using a spring-damper model inside the **PD Contact Paddle** block. The negative of this force is the force applied on the paddle. Hence the output of this block is negated and the output *FzOmni* is applied to the Omni.

$F_{z_{topwall}}$ is the force applied on the ball when it hits the top wall. This contact force is calculated using a spring-damper model inside the **PD Contact Top Wall** block.

The position of the ball in the vertical direction is calculated by integrating Equation 12.1 twice:

$$z = \iint \frac{F_z}{m} \tag{12.3}$$

3. Double click on **PD Contact Top Wall** subsystem to look underneath it. The model is shown in Figure 12.4

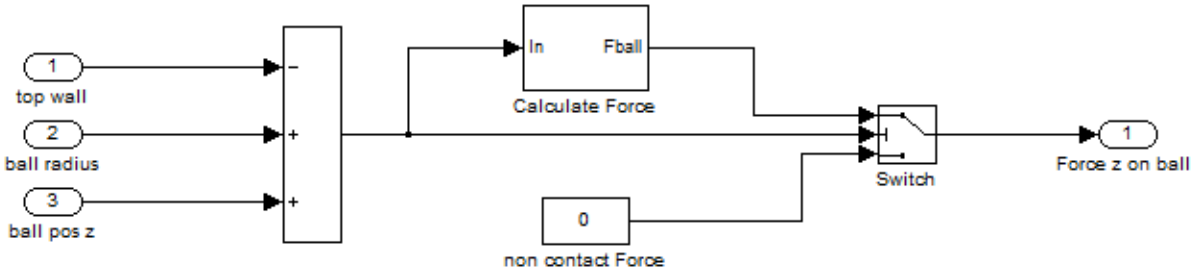


Figure 12.4: **PD Contact Top Wall** model

This model determines if there is a contact between the ball and top wall. The contact is determined by looking at the vertical position of the ball only. If the vertical position of the ball is greater than or equal to the vertical position of the top wall, there is contact. Once contact has been established, the force is calculated using a spring-damper model otherwise the force is set to zero.

4. Double click on **PD Contact Paddle** subsystem to look underneath it. The model is shown in Figure 12.5

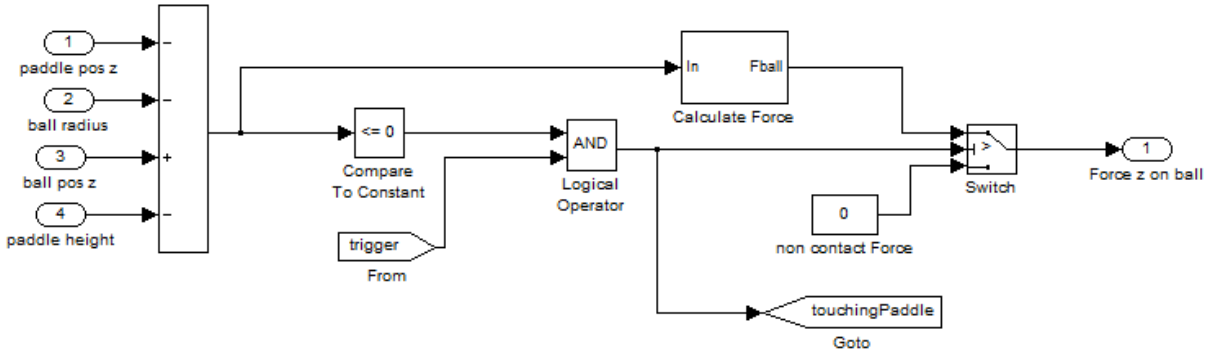


Figure 12.5: **PD Contact Paddle** model

The logic to determine if there is a contact between the paddle and the ball is more complex than the top wall model because the width of the paddle does not span from the left wall to the right wall. The horizontal and the vertical position of the ball relative to the paddle must be determined. The relative horizontal position can be found using the output *trigger* discussed earlier. If *trigger* is 1 and the vertical position of the ball is less than or equal to the vertical position of the paddle, there is contact. The output *touchingPaddle* is 1 when the ball is in contact with the paddle and 0 otherwise. Once contact has been established, the force is calculated using a spring-damper model, otherwise the force is set to zero.

12.2 In-Laboratory Experiment

12.2.1 1D Model

1. Open **Pong1D.mdl**. This is a model of Pong in one dimension.
2. Before you can run this model, you have to run a MATLAB script to setup some of the variables used by the game. From the directory where the laboratory files are saved, open **SetupPong.m**. In this script you will notice that the mass of the ball is set to 0.1 Kg and the stiffness and damping of the walls is set to 500 N/m and 0.5 Ns/m respectively.
3. Run the setup script to load the variables onto the workspace.
4. Notice that there is a **Manual Switch** in front of the **Omni** block which feeds into the *Task Force* input. Check that the *Switch* is in the downward position to feed a constant 0 N force to the Omni.
5. Ensure that the **Simulation mode** is set to *External*.
6. Build **Pong1D.mdl**. Before running this model, hold the end-effector of the Omni in your hand.
7. Run the model and switch to the 3D Viewer.
8. Stop the model and observe the behaviour of the paddle and ball.
9. Enable the haptics by switching the **Manual Switch** to route the signal from the force calculation blocks.
10. Re-run the model. There should be able to feel a force downwards every time the ball hits the paddle. You may also notice that it is a lot easier to keep the ball on the paddle with a minimal number of bounces while the haptics are turned on. This is because humans are better able to control most systems using both sight (visual feedback) and feel (proprioceptive feedback) rather than visual feedback alone.
11. Turn off haptics and try to hold the ball on the paddle with a minimum number of bounces using visual feedback alone. Achieving the same damping should be a lot more difficult.
12. Play with the game to get comfortable using the paddle.

12.2.2 Extending to 2D Model

In this section you will extend the model so that the game is implemented in 2D. In this configuration you should be able to hit the ball left/right to bounce it off of the side walls as well. Three additional contact models will have to be created: contact with left wall, contact with right wall and contact with the paddle to generate forces in the horizontal direction.

1. Open and save **Pong1D.mdl** as **Pong2D.mdl**.
2. First, you need to setup a dynamic model of the ball in the horizontal direction.

Question 12.1
Identify the dynamic equation of the ball in the horizontal direction. Your solution should include equations resembling Equation 12.1, Equation 12.2, and Equation 12.3.

Answer 12.1

$$F_y = m\ddot{y}$$

$$F_y = F_{y_{paddle}} + F_{y_{leftwall}} + F_{y_{rightwall}}$$

$$y = \iint \frac{F_y}{m}$$

□□□

3. Create a model to implement the dynamic system. Create this model in the subsystem shown in Figure 12.2.

4. Study the **PD Contact Top Wall** model. Use this model as a reference to create a subsystem block called **PD Contact Right Wall**. These two blocks should be very similar to each other, the only difference being that one is implemented in the vertical direction and the other in the horizontal.

   The position of the right wall is given by the first element of the signal *wallPos*.

   For stiffness, use the variable *stiffness* and for damping, use *damping*. These variables have already been setup in the **SetupPong.m** script.

5. Similarly, create a block to implement contact with the left wall. Call this block **PD Contact Left Wall**.

   The position of the left wall is given by the negative of the first element of the signal *wallPos*.

   For stiffness and damping, use the same *stiffness* and *damping* variables stated previously.

6. Create a susbsystem called **D Contact Paddle**. The lateral contact with the paddle requires a relative change in the position of the paddle with respect to the ball. This contact will be established using a damper model only. The logic for this contact model should be implemented as follows:

```
If touchingPaddle > 0
    F = lateral_damping * v
else
    F = 0
end
```

   where *lateral_damping* is the damping coefficient and *v* is the velocity. The velocity is calculated by differentiating the signal *paddle pos y - ball pos y*. See Step 4 in Section 12.1.1 for a more detailed outline of the signal *touchingPaddle*.

   Modify **SetupPong.m** to include the variable *lateral_damping* with a magnitude 2 Ns/m.

   The negative of the lateral contact force is the force on the paddle. Negate this force and use a **Goto** block called *FyOmni* to feed this force to the Omni.

7. Sum the lateral and vertical forces, and feed this force to the dynamic model.

8. Change the input *ball pos y* of the **y position of wall relative to y of paddle** block to the output of your dynamic model. Your model should resemble Figure 12.6.

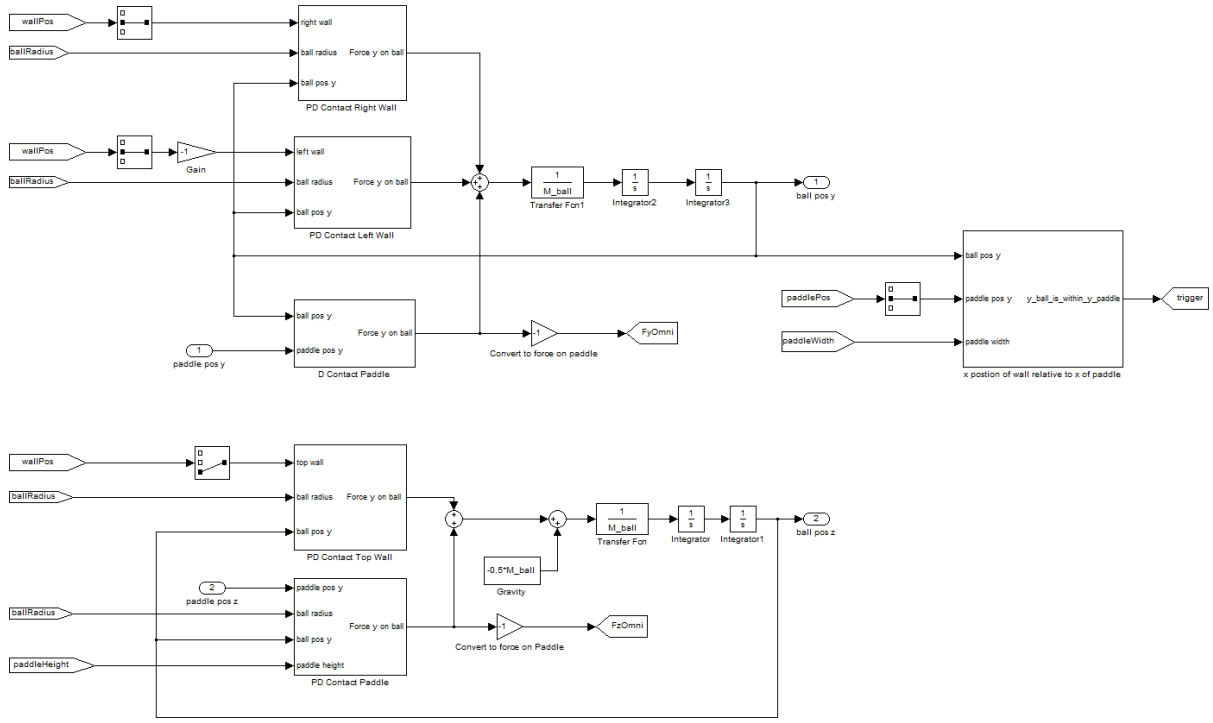


Figure 12.6: 2D Pong ball model

9. The parent model should now resemble Figure 12.7.

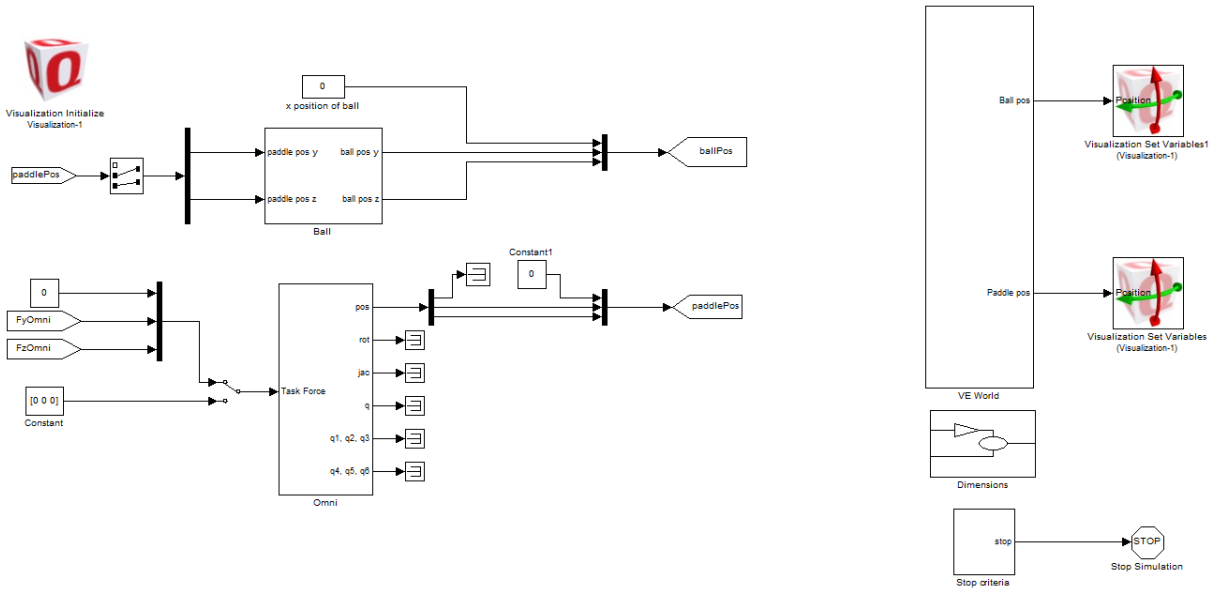


Figure 12.7: Complete 2D Pong model

12.2.3 Testing the Model

1. Make sure that the switch in front of the **Omni** block is routed to the constant **[0 0 0]**. It is important to run the model without haptics first in order to ensure that the model is working properly. In case of an error, sending large forces to the Omni could damage the device.

2. Build and run the model. You should now be able to hit the ball sideways to bounce it off of the side walls. Play

with the game to make sure that the ball behaves as expected.

3. Once you are satisfied with the behavior of the ball, turn on the haptics. Now you should be able to feel the force of the ball in both horizontal and vertical directions as it strikes the paddle.

12.2.4 Laboratory Files to Submit

The following files should be submitted for evaluation:

- Pong2D.mdl

APPENDIX A

DENAVIT-HARTENBERG CONVENTION

A.1 Introduction

Although it is possible to carry out forward kinematics analysis analytically from geometry, this approach becomes tedious for multi-link robots with many joints. It is useful to use a standard technique to derive the forward kinematics of a robot. The Denavit-Hartenberg convention (D-H convention) is the technique used in this curriculum to perform the forward kinematics analysis of the Omni.

A.2 Labeling

1. For a serial robot with n joints, label all joints from joint 1 to joint n. Label joint i with variable q_i.
2. Label all links from link 0 to link n. Link 0 is the link between the base of the robot and joint 1. In some cases, if joint 1 is located at the base, then link 0 is taken to have length zero. If two joints are located on top of each other, then the link between those joints is also taken to have length 0.
3. Attach a coordinate frame (frame 0 to frame n) rigidly to the end of each link. Due to some links that have 0 lengths, the origin of some frames may coincide. In this way, when joint i moves, the coordinate frame i also moves in the same way. Frame 0 is the base frame, attach it to the base. This will be taken as the global frame since it does not move with any joint.

A.2.1 How to Orient the Frames?

1. Draw the robot such that all joint variables are zero.
2. Label all joints and links.
3. Label all joint axis. Joint i has joint axis z_i. The direction of z_i axis is the direction of positive motion of joint i.
4. Label the base origin o_0 anywhere on the z_0 axis.
5. Choose x_0 and y_0 conveniently to complete the base frame according to the right-hand-rule.
6. Next setup the rest of the frames as follows:

   Find the line perpendicular to both z_{i-1} and z_i. Label this line axis x_i. Label o_i at the intersection of x_i and z_i. Choose y_i to form a right-hand frame.

   Special Cases:

   (a) *If z_{i-1} and z_i are parallel*: Choose o_i to be located at joint i+1. Choose x_i to be the line that passes through o_i and that is normal from z_{i-1} to z_i. Choose y_i to form a right-hand frame.

   (b) *If z_{i-1} and z_i intersect*: Choose x_i to be perpendicular to both z_{i-1} and z_i. Place o_i at the intersection of z_{i-1} and z_i. Choose y_i to form a right-hand frame.
7. Setup the end-effector frame, o_n ($x_n\, y_n\, z_n$). Make z_n in the same direction as z_{n-1}. Make o_n the center of the end-effector or anywhere on the end-effector as desired. Set x_n and y_n to form a right hand frame.

A.2.2 Terminology

Term	Variable	Description
Homogeneous matrix	A_i	Transforms a point in frame i coordinates to frame $i-1$ coordinates. Each homogeneous matrix has the form: $\begin{bmatrix} R^i_{i-1} & d^i_{i-1} \\ 0 & 1 \end{bmatrix}$ where R^i_{i-1} is the rotation matrix from frame i to frame $i-1$ and d^i_{i-1} is the distance from frame i to frame $i-1$.
Transformation matrix	T^j_i	Transforms a point in frame j coordinates to frame i coordinates. Frame j and i are not consecutive. A transformation matrix can be obtained from a set of homogeneous matrices: $T^j_i = A_{i+1}\,A_{i+2}...A_{j-1}\,A_j$ (A.1) Each transformation matrix has the form: $\begin{bmatrix} R^j_i & d^j_i \\ 0 & 1 \end{bmatrix}$ where R^j_i is the rotation matrix from frame j to frame i and d^j_i is the distance from frame j to frame i.

Table A.1: Terminology

A.3 D-H Convention

The D-H convention provides a systematic way of arriving at the homogeneous matrix, A_i. Once the frames have been setup, the parameters shown in will help derive the D-H parameters.

Parameter	Description
θ_i	*Angle*: The angle between x_{i-1} and x_i, measured about z_{i-1}.
d_i	*Offset*: The distance from o_{i-1} to the intersection of x_i and z_{i-1}, measured along z_{i-1}.
a_i	*Length*: The distance from the intersection of x_i and z_{i-1} to o_i, measured along x_i.
α_i	*Twist*: The angle between z_{i-1} and z_i, measured about x_i.

Table A.2: D-H Parameters

From these parameters, the homogeneous matrix can be calculated as follows:

$$A_i = Rot_{z,\theta_i}\,Trans_{z,d_i}\,Trans_{x,a_i}\,Rot_{x,\alpha_i}$$

$$A_i = \begin{bmatrix} cos(\theta_i) & -sin(\theta_i) & 0 & 0 \\ sin(\theta_i) & cos(\theta_i) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & a_i \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(\alpha_i) & -sin(\alpha_i) & 0 \\ 0 & sin(\alpha_i) & cos(\alpha_i) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{(A.2)}$$

Once all the A_i matrices are known, Equation A.1 can be used to derive the transformation matrix between any two frames.

A.4 Example: A planar Elbow

The planar elbow shown in Figure A.1 is a two link manipulator with two revolute joints. Steps 1 to 7 of Section A.2.1 have already been performed.

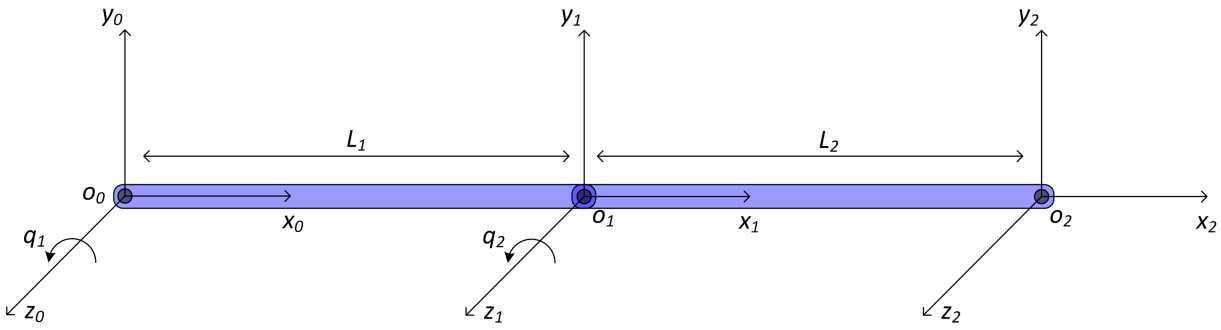


Figure A.1: Two-link planar elbow

The D-H parameters of this manipulator, derived using the definitions in Table A.2 are shown in Table A.3.

Link	$\boldsymbol{\theta}_i$	d_i	a_i	$\boldsymbol{\alpha}_i$
1	q_1	0	L_1	0
2	q_2	0	L_2	0

Table A.3: D-H Parameters for planar elbow

Substituting these values into Equation A.2 will result in two homogeneous matrices A_1 and A_2. Then T_0^2 can be calculated by using:

$$T_0^2 = A_1\, A_2$$

where T_0^2 is the transformation matrix needed to transform any point in frame 2 coordinates to the global frame (frame 0). Therefore, T_0^2 will have the form:

$$T_i^j = \begin{bmatrix} R_0^2 & d_0^2 \\ 0 & 1 \end{bmatrix}$$

Assume a point $p_2 = \begin{bmatrix} p_{x_2} \\ p_{y_2} \\ p_{z_2} \end{bmatrix}$ is given in frame 2 coordinates. To transform this point from frame 2 to the frame 0 coordinates:

$$p_0 = R_0^2 p_2 + d_0^2$$

Expand your research with open architecture robotics platforms

ROBOTICS

Open Architecture Robots

KUKA Robot

KUKA C2sr Controller

Robot and controller sold separately through KUKA.

QUARC Control Design Software

Pre-Configured PC

VP-6 Denso Robot

Denso Controller

QUARC Control Design Software

Pre-Configured PC

HAPTICS

6 DOF Telepresence System

5 DOF Haptic Wand

HD² High Definition Haptic Device

More than 20 years of experience in the field of mechatronics and controls enables Quanser to design and implement innovative platforms for research in Robotics and Haptics. Quanser technology allows advanced research and teaching in robotics by delivering an open architecture platform based on popular industrial and commercial robots. These open architecture systems can be customized for research in such areas as robot-assisted surgery, force feedback teleoperation, space and undersea expeditions and human rehabilitation systems.

To discuss your robotics research needs, please email info@quanser.com

QUANSER

INNOVATE. EDUCATE.

INFO@QUANSER.COM +1-905-940-3575 QUANSER.COM

Solutions for teaching and research. Made in Canada.