Lab 07
Mysha, Shaaf Faroque

7.1. GRIPPER 60

Task 7.1 Jaw Position (10 points)

Develop a MATLAB function `function success = positionJaw(position)` that accepts a distance argument representing the linear position of a jaw with respect to the center and moves the jaws to this distance. For this task, you'll have to

$$x = l_1 \cos(\theta_1) + l_2 \cos(\theta_1 + \theta_2)$$

$$y = l_1 \sin(\theta_1) + l_2 \sin(\theta_1 + \theta_2)$$

$y = 0$, find θ_1 and θ_2 in terms of x.

Total length from motor to jaw:

$$r = \sqrt{x^2 + y^2}$$

$$r = x \quad \because y = 0.$$

$$\cos(\theta_2) = \frac{x^2 - l_1^2 - l_2^2}{2 l_1 l_2} \Rightarrow \theta_2 = \cos^{-1}\left(\frac{x^2 - l_1^2 - l_2^2}{2 l_1 l_2}\right)$$

for θ_1:

$$\alpha = \cos^{-1}\left(\frac{l_1^2 + x^2 - l_2^2}{2 l_1 x}\right)$$

$$\theta_1 = \pi - \alpha$$

```
% Gripper Control Functions


% Main function to initiate the gripper movement

function status = grip_object()

   % Define a multiplier and step size for the desired position

   step_multiplier = 96;        % Multiplier for angle steps

   angle_step = 0.29;           % Step size in mm (or some unit depending on calibration)
```

```
    % Compute target jaw position
    targetPos = step_multiplier * angle_step;


    % Call function to set the jaw to the computed position
    status = setposition_jaw(targetPos);
end
```

```
% Function to set the jaw to a specific position in mm
function success = setposition_jaw(jawPos)
    % Link lengths for the gripper mechanism (in mm)
    link1 = 8.68;
    link2 = 25.91;
```

```
    % Define valid motion range for the jaw (fully closed to fully
open)
    jaw_min = 10;                          % Minimum physical limit
    jaw_max = link1 + link2;               % Maximum reachable position
```

```
    % Input validation: Check if desired position is within physical
limits
    if jawPos < jaw_min || jawPos > jaw_max
        fprintf('Position %.2f mm is outside valid range [%.2f mm,
%.2f mm]\n', jawPos, jaw_min, jaw_max);
        success = false;
        return;
    end
```

```
    % Error function based on forward kinematics of 2-link system
    % This implicitly solves for gripper joint angle such that the
total
    % horizontal extension matches the desired jawPos
    error_fn = @(theta) ...
```

```matlab
        link1 * cos(theta) + link2 * cos(asin(-link1 * sin(theta) / link2)) - jawPos;

    % Initial guess for the solver (assumed close to zero for symmetry)
    initial_guess = 0;

    % Use fsolve to find the angle that satisfies the geometric constraint
    gripper_angle = fsolve(error_fn, initial_guess, optimoptions('fsolve', 'Display', 'off'));

    % Define mechanical angle limits for the gripper servo (in radians)
    angle_limits = deg2rad([-150, 150]);

    % Check if the computed angle is within mechanical constraints
    if gripper_angle < angle_limits(1) || gripper_angle > angle_limits(2)
        fprintf('[Limit Error] Computed angle %.2f rad exceeds range [-150°, 150°].\n', gripper_angle);
        success = false;
        return;
    end

    % Attempt to command the servo motor using Arbotix library
    try
        % Create Arbotix controller object (adjust COM port as needed)
        gripper = Arbotix('port', 'COM8', 'nservos', 5);

        % Send position command to servo #5 with moderate speed
        gripper.setpos(5, gripper_angle, 60);
```

```
        % Display success message
        fprintf('-> Jaw set to %.2f mm (angle = %.2f rad)\n', jawPos, 
gripper_angle);
        success = true;
```

```
    catch
        % In case of communication or execution failure
        fprintf('[Error] Unable to move gripper to specified 
position.\n');
        success = false;
    end
end
```

The length of the cube side was 2.8 mm.
And the resulting angle jawangle was found to be around 1.7 rads.

Task 7.2 Gripping (20 points)

Develop a MATLAB function gripObject that assumes that the jaws are at the position where they are tangent to the cube on each side and sets the motor position to a value that ensures a successful grip. Gradually increase the goal angle of the servomotor from its current position in increments of 0.29° to experimentally determine the $\Delta\vartheta$ required to ensure a successful grip between wood and plastic.

```
function status = grip_object()
    % Define a multiplier and step size for the desired position
    step_multiplier = 96;        % Multiplier for angle steps
    angle_step = 0.29;           % Step size in mm (or some unit 
depending on calibration)

    % Compute target jaw position
    targetPos = step_multiplier * angle_step;

    % Call function to set the jaw to the computed position
```

```
    status = setposition_jaw(targetPos);
end
```

At around 1.25 rad, the grip was enough to pick the object.

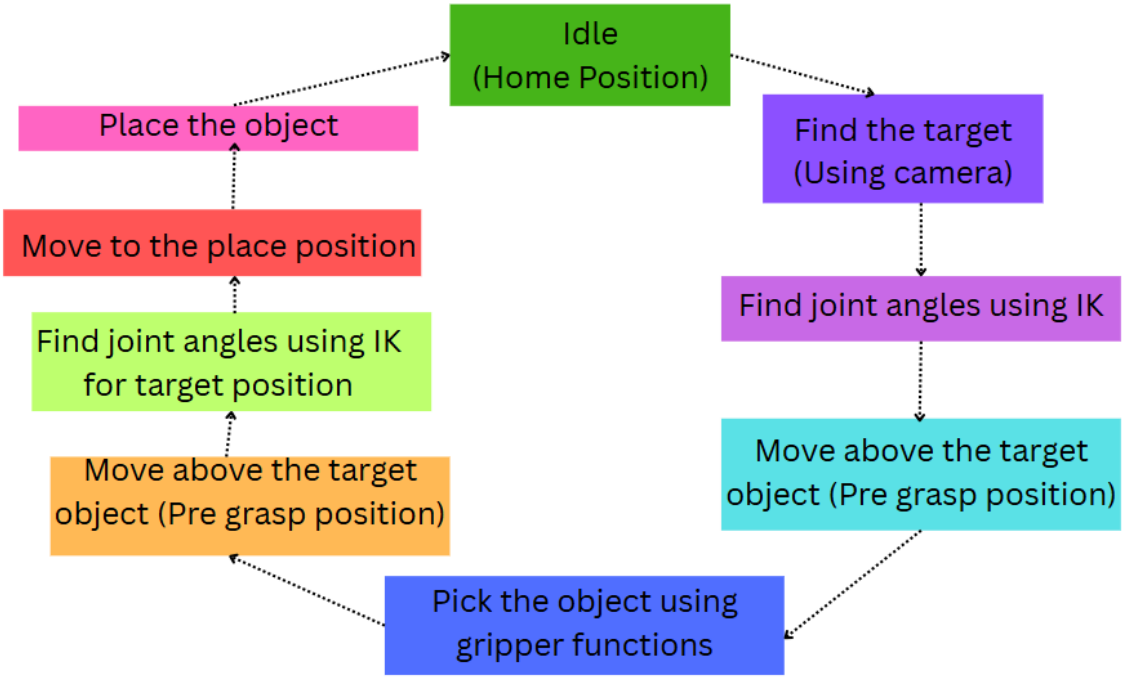
Task 7.3 **FSM (30 points)**

Draw a state-transition diagram, on paper, of an FSM corresponding to the following scenario:

- System is in idle state till it receives pick location, (x_1, y_1, z_1, ϕ_1) and place location, (x_2, y_2, z_2, ϕ_2).
- Geometry of the object to be picked and placed, including its orientation, is known before hand.
- The locations can be assumed to lie in the interior of the manipulator's workspace, and the object is in an orientation so that it can be picked.
- System should verify the final placement location, before determining that the task has concluded.
- Smooth motion and accurate placement[a] is desirable.

You can draw inspiration from this MATLAB example for a pick and place workflow.

[a] You'll have to plan your gripper picking and releasing strategy, considering the accuracy of your system, determined in earlier labs.



Task 7.4

Implement the system described by the previous FSM in MATLAB for Phantom X Pincher and the cube object. Your submission should have a single main robot loop, leveraging Stateflow. Provide documentation of your implementation, properly commented code for all functions, a video of your best execution, and identify and comment on points of improvement. Following are resources available for this task:

- A MATLAB example for a pick and place workflow in simulation using Stateflow
- Tutorial on getting started with Stateflow and its different features
- Self-paced online course on Stateflow (1.5 Hours)

```
% Main Script: Initializes the robot arm, processes depth image, and
runs pick-and-place demo

clc; clear all; close all;

% Initialize Arbotix with COM4 port and 5 servos
arb = Arbotix('port', 'COM4', 'nservos', 5);

% Move servo 4 to a specific angle
arb.setpos(4, pi/1.5);

% Capture and process depth image (visual + depth info)
[img, ig] = depth_example();

% Start the autonomous pick and place sequence
armDemo(arb);


%% Main Routine: Pick-and-place finite state machine
function armDemo(arb)
```

```
placed = 1;
STATE = 1;
global focalLength principalPoint

% Camera calibration parameters
focalLength = [1.4053e+03, 1.4053e+03];
principalPoint = [951.5109, 526.6981];

% State machine loop
while STATE ~= 8
    switch STATE
        case 1
            homePosition(arb); % Move arm to home
            STATE = STATE + 1;
            pause(4);

        case 2
            [targets, howMany] = findTargets(); % Locate targets
            STATE = STATE + 1;

        case 3
            if placed <= howMany
                prev = goToTarget(targets(placed), arb); % Move to
target
                STATE = STATE + 1;
                pause(5);
            end

        case 4
            pickTarget(targets(placed), arb, prev); % Pick up
object
```

```
            STATE = STATE + 1;
            pause(4);

        case 5
            goToPlace(arb); % Move to drop location
            STATE = STATE + 1;
            pause(6);

        case 6
            place(arb); % Place the object
            placed = placed + 1;
            STATE = STATE + 1;
            pause(3);

        case 7
            if placed > howMany
                STATE = 8;
                homePosition(arb); % Done. Go home.
            else
                homePosition(arb);
                STATE = 3; % Repeat for next object
                pause(3);
            end

        otherwise
            STATE = 8; % Exit condition
    end
  end
end
```

```matlab
%% Move robot arm to predefined home position
function homePosition(arb)
   arb.setpos([pi/6, pi/3, pi/12, pi/2, 0], [100 100 100 100 100]);
end


%% Image processing: Detect targets (colored objects)
function [targets, howMany] = findTargets()
   global img ig BW

   % Capture depth image
   [img, ig] = depth_example();

   % Mask out undesired image areas (frame boundaries)
   img(:, 1:80, :) = 0;
   img(:, 500:640, :) = 0;
   img(425:480, :, :) = 0;
   img(1:65, :, :) = 0;

   figure; imshow(img);

   % Create color masks
   [blueMask, ~] = createBlueMask(img);
   [yellowMask, ~] = createYellowMask(img);
   [redMask, ~] = createRedMask(img);
   [greenMask, ~] = createGreenMask(img);

   % Combine all masks and detect edges
   BW = blueMask | redMask | yellowMask | greenMask;
   BW = edge(BW, 'Canny');
```

```
    figure; imshow(BW);

    % Extract bounding boxes of detected regions
    targets = regionprops(BW, "BoundingBox");
    howMany = length(targets);
end


%% Move arm to above the target and return joint configuration
function prev = goToTarget(target, arb)
    global BW ig principalPoint pose_obj

    % Create mask of the selected target
    cube_mask = false(size(BW));
    cube_mask(round(target.BoundingBox(2)):round(target.BoundingBox(2)
+ target.BoundingBox(4)), ...
              round(target.BoundingBox(1)):round(target.BoundingBox(1)
+ target.BoundingBox(3))) = true;

    % Find centroid of the object
    cubes = cube_mask & BW;
    stats = regionprops(cubes, 'Centroid');
    object_center = stats.Centroid;
    row = round(object_center(2));
    col = round(object_center(1));

    % Get depth value and convert to 3D camera coordinates
    depth = ig(row, col);
    [x,y,z] = pixelToCameraCoords(col, row, depth, principalPoint(1),
principalPoint(2));
```

```
    % Adjust coordinates
    % x = -x;
    % z = 68 - z;
    % Passing manual pick position to verify
    pose_obj = [-13, -13, 4];

    % Go slightly above object
    p_dest = pose_obj + [0 0 5];
    currentPos = arb.getpos();
    jointAngles = findOptimalSolution(p_dest, currentPos(1:4));
    jointAngles = jointAngles - [pi/9.5 0 0 0]; % Tuning offset
    prev = jointAngles;

    % Move to target
    arb.setpos([jointAngles 0], [60 60 60 60 60]);
end


%% Pick up the target using gripper
function pickTarget(target, arb, prev)
    global pose_obj

    % Go slightly below object to grab
    p_dest = pose_obj - [0 0 4];
    currentPos = arb.getpos();
    jointAngles = findOptimalSolution(p_dest, currentPos(1:4));
    jointAngles = jointAngles - [pi/9.5 0 0 -0.3];

    % Move down and grab object
```

```
    arb.setpos([jointAngles 0], [60 60 60 60 60]);
    pause(3);
    temp = arb.getpos();
    arb.setpos([temp(1:4), 1.1], [60 60 60 60 60]); % Close gripper
    pause(2);
    arb.setpos([prev 1.1], [60 60 60 60 60]); % Lift object
end



%% Move arm to drop location
function goToPlace(arb)
    arb.setpos(1, pi/2); % Turn base
end



%% Drop the object at predefined location
function place(arb)
    currentPos = arb.getpos();
    [x, y, z, ~] = pincherFK(currentPos);
    pose_obj = [x, y, z];
    p_dest = pose_obj - [0 0 3];

    jointAngles = findOptimalSolution(p_dest, currentPos(1:4));
    jointAngles = jointAngles - [pi/9 0 0 -0.3];

    % Lower and open gripper
    arb.setpos([jointAngles currentPos(5)], [60 60 60 60 60]);
    pause(3);
    temp = arb.getpos();
    arb.setpos([temp(1:4), 0], [60 60 60 60 60]); % Open gripper
```

```
    pause(3);
    arb.setpos([currentPos(1:4) 0], [60 60 60 60 60]); % Return back
end
```

```
%% Convert pixel to camera coordinates using depth
function [X_cm, Y_cm, Z_cm] = pixelToCameraCoords(u, v, Z_m, cx, cy)
    u0 = cx * (640/1920);
    v0 = cy * (480/1080);

    % Use intrinsic calibration to compute 3D coords
    X = ((u - u0) * Z_m) / 465.6;
    Y = ((v - v0) * Z_m) / 471.03;

    X_cm = X * 100;
    Y_cm = Y * 100;
    Z_cm = Z_m * 100;
end
```

```
%% --- Kinematics Helper Functions ---

% Kinematics  Helper Functions
function vs = skew(v)
    vs = [0 -v(3) v(2);
          v(3) 0 -v(1);
         -v(2) v(1) 0];
end
```

```
function T = exp(S, theta)
```

```
    w_skew = skew(S(1:3)/norm(S(1:3)));

    exp_w = eye(3) + w_skew*sin(theta) + w_skew^2*(1-cos(theta));

    G = eye(3)*theta +
(1-cos(theta))*w_skew+(theta-sin(theta))*w_skew^2;

    T = [exp_w, G*S(4:6);

         zeros(1,3), 1];

end
```

```
function [S1, S2, S3, S4, Tsb] = getTsb()

    l_1=10;

    l_2=4.5;

    l_3=10.7;

    l_4=10.5;

    w1 = [0;0;1];

    q1 = [0;0;l_1];

    S1 = [w1;cross(-w1,q1)];


    w2 = [1;0;0];

    q2 = [0;0;l_1+l_2];

    S2 = [w2;cross(-w2,q2)];


    w3 = [1;0;0];

    q3 = [0;0;l_1+l_2+l_3];

    S3 = [w3;cross(-w3,q3)];


    w4 = [1;0;0];

    q4 = [0;0;l_1+l_2+l_3+l_4];

    S4 = [w4;cross(-w4,q4)];
```

```
    syms theta_1 theta_2 theta_3 theta_4
```

```
    Tsb = 
exp(S1,theta_1)*exp(S2,theta_2)*exp(S3,theta_3)*exp(S4,theta_4);
end
```

```
function zeroConfig= getZeroConf()
    l_1=10;
    l_2=4.5;
    l_3=10.7;
    l_4=10.5;
    l_5=9.5;
    zeroConfig = [1 0 0 0;
         0 1 0 0;
         0 0 1 l_1 + l_2 + l_3 + l_4 + l_5;
         0 0 0 1];
end
function [x,y,z,R] = pincherFK(jointAngles)
    M = getZeroConf();
    syms theta_1 theta_2 theta_3 theta_4
    [~, ~, ~, ~, Tsb] = getTsb();
    T = double(subs(Tsb,[theta_1 theta_2 theta_3 
theta_4],jointAngles(1:4))*M);
    x = T(1,4);
    y = T(2,4);
    z = T(3,4);
    R = T(1:3, 1:3);
end
```

```
function angles = findJointAngles(x,y,z,phi)
    angles = zeros(4,4);
    l_1=10;
```

```
   l_2=4.5;

   l_3=10.7;

   l_4=10.5;

   l_5=9.5;


   theta1_1 = atan2(y,x);

   theta1_2 = theta1_1 + pi;


   r = sqrt(x^2 + y^2);

   s = z - (l_1+l_2);


   r_ = r - l_5*cos(phi);

   s_ = s - l_5*sin(phi);


   D = ((r_*r_) + (s_*s_) - l_3^2 - l_4^2)/(2*l_3*l_4);


   num = sqrt(1-D*D);


   theta3_1 = atan2(num,D);

   theta3_2 = atan2(-num,D);


   theta2_1 = atan2(s_,r_) - atan2(l_4*sin(theta3_1), l_3 +
l_4*cos(theta3_1));

   theta2_2 = atan2(s_,r_) - atan2(l_4*sin(theta3_2), l_3 +
l_4*cos(theta3_2));


   theta4_1 = (phi - theta2_1 - theta3_1);

   theta4_2 = (phi - theta2_2 - theta3_2);


   angles(1,:) = [theta1_1+pi/2 -theta2_1+pi/2  -theta3_1 -theta4_1];

   angles(2,:) = [theta1_1+pi/2 -theta2_2+pi/2  -theta3_2 -theta4_2];
```

```
   angles(3,:) = [theta1_2+pi/2 -(-theta2_1+pi)+pi/2 -theta3_1
-theta4_1];
   angles(4,:) = [theta1_2+pi/2 -(-theta2_2+pi)+pi/2 -theta3_2
-theta4_2];

end


function solution = findOptimalSolution(desiredPos, currentPos)
    x = desiredPos(1);
    y = desiredPos(2);
    z = desiredPos(3);
    phi = -pi/2;

   solutions = findJointAngles(x, y, z, phi);

   B = cellfun(@checkJointLimits, num2cell(solutions, 2),
'UniformOutput', false);
   B = cell2mat(B);  % B is now n_solutions × 4 logical matrix

   % Identify solutions where ALL joints are within limits
   valid_solutions_mask = all(B, 2);  % True only if all joints in a
row are true

   % Keep only valid solutions (original joint angles)
   valid_solutions = solutions(valid_solutions_mask, :);

   % If no valid solutions, return empty or handle error
   if isempty(valid_solutions)
       solution = [];  % Or throw an error/warning as needed
       return;
   end
```

```
    % Absolute errors for all joints of all valid solutions
    delta = abs(valid_solutions - currentPos(1, 1:4));

    s = sum(delta, 2);

    % Find the solution with the minimum total error
    [~, idx] = min(s);
    solution = valid_solutions(idx, :);  % Return the actual joint
angles
end
```

```
function isValid = checkJointLimits(jointAngles)
    % joint angle limits in radians (-150 to 150 degrees)
    thetaLimits = [-150*pi/180, 150*pi/180]; % Convert degrees to
radians

    % Check if all joint angles are within limits
    isValid = all(jointAngles >= thetaLimits(1) & jointAngles <=
thetaLimits(2));
end
```

State Number	What It Does	Details
STATE = 1	**Go to Home Position**	Calls `homePosition(arb)` to reset the robot to a known posture.
STATE = 2	**Find Targets (Objects)**	Uses image processing (`findTargets()`) to detect colored blobs (red, green, blue, yellow) and saves their bounding boxes.
STATE = 3	**Move Above the Target Object**	Moves the robot to a position **above** the current object.

`STATE = 4`	**Pick Up the Object**	Moves slightly down and uses the gripper to pick the object.
`STATE = 5`	**Move to the Drop Zone**	Rotates or repositions the arm toward a predefined "place" location.
`STATE = 6`	**Place the Object**	Lowers the arm and opens the gripper to drop the object.
`STATE = 7`	**Decide Whether to Repeat or End**	If more objects are left, go back to `STATE = 3`. Otherwise, move to home and end.
`STATE = 8`	**End of Process**	FSM exits the loop; pick-and-place is done.

Initialize the robot (`Arbotix` with 5 servos).

Move the arm to a neutral "home" position.

Use a depth camera to locate colored objects.

For each object:

- Move the arm above it
- Pick it up
- Move to a place location
- Drop it

Repeat for all detected objects

A brief video of the pick and place:
https://www.canva.com/design/DAGm4dR0OnQ/QqBiL06DHYHFOy_d2q8Mog/edit?utm_content=DAGm4dR0OnQ&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton

Improvements can be made in the perception logic and some offsets needs to be looked at for the x,y,z coordinates.

.