MAE C163B / C263B – Dynamics of Robotic System
Project No. 3 – Jacobian Derivation

Professor: Jacob Rosen
Teaching Assistant: Xiaochen Li

David Blake Dee
Student ID: 106045624

Part 1: Jacobian Derivation

The Jacobian is a 6x6 matrix of partial differentials that govern two important relationships in robotics: joint velocity to end effector motion (translational and rotational), and joint torques to applied loads. There are two ways to derive the Jacobian. The first method involves iterative propagation techniques, and the second method involves explicit differentiation. This section dives into deriving the Jacobian using each technique for a SCARA Mitsubishi Arm.

1.1 Velocity Propagation

Velocity propagation is a method of deriving the Jacobian through its relationship between the motion of the end effector and the angular velocities of each of the independent joints. This is expressed in the follow equation:

$$v = J(\Theta)\,\dot{\Theta}$$

$$v = \begin{bmatrix} v_x & v_y & v_z & \omega_x & \omega_y & \omega_z \end{bmatrix}^T$$

$$\dot{\Theta} = \begin{bmatrix} \dot{\theta}_1 & \dot{\theta}_2 & \dot{\theta}_3 & \dot{d}_4 \end{bmatrix}^T$$

In order to establish this relationship between all the joint velocities and the end effector, the relationships between each adjacent link needed to be solved for, and then iteratively chained till the final relationship was found. This was done with the following equation relating the effect of angle velocity on an adjacent link's translational and rotational motion:

$$^{i+1}\omega_{i+1} = {}^{i+1}_{i}R\,{}^{i}\omega_i + \begin{bmatrix} 0 \\ 0 \\ \dot{\theta}_{i+1} \end{bmatrix}$$

$$^{i+1}v_{i+1} = {}^{i+1}_{i}R\,({}^{i}\omega_i \times {}^{i}P_{i+1} + {}^{i}v_i) + \begin{bmatrix} 0 \\ 0 \\ \dot{d}_{i+1} \end{bmatrix}$$

As seen above, the inputs for these iterative calculations required information about relating the geometry of the two joints (and ultimately all the joints to each other). In order to achieve this, a parametric DH table describing the robot was created. This was done by symbolically defining the DH parameters within each link, and then using the "SerialLink" function to chain them.

```
SCARA:: 4 axis, RRRP, modDH, slowRNE, Symbolic
+---+-----------+-----------+-----------+-----------+-----------+
| j |     theta |         d |         a |     alpha |    offset |
+---+-----------+-----------+-----------+-----------+-----------+
|  1|         q1|         D1|          0|          0|          0|
|  2|         q2|          0|         L1|          0|          0|
|  3|         q3|          0|         L2|          0|          0|
|  4|          0|         q4|          0|         pi|          0|
+---+-----------+-----------+-----------+-----------+-----------+
tool:    t = (0, 0, -38), RPY/xyz = (0, 0, 0) deg
```

Next, forward kinematics was run on the table with the relevant matrices extracted:

```
% running forward kinematics
T01 = SCARA.A(1,th);
T12 = SCARA.A(2,th);
T23 = SCARA.A(3,th);
T34 = SCARA.A(4,th);
T4ee = SCARA.tool;
T0ee = T01*T12*T23*T34*T4ee;

% extracting rotation and translation matrices from transformation matrices
[R01,P1] = tr2rt(T01);
[R12,P2] = tr2rt(T12);
[R23,P3] = tr2rt(T23);
[R34,P4] = tr2rt(T34);
[R4ee,Pee] = tr2rt(T4ee);
```

This information was then plugged into the equations above, iteratively solved for each link, and then combined into the Jacobian matrix. This matrix was, however, in reference to the end effector, and so the Jacobian Rotational matrix describing the relationship between the base and end effector frames was multiplied with in order to put the Jacobian in reference to the base:

```
%calulating angular velocities
w1 = R01.'*w0 + [0;0;dt1];  % first revolute
w2 = R12.'*w1+[0;0;dt2]; % second revolute
w3 = R23.'*w2+[0;0;dt3]; % third revolute
w4 = R34.'*w3+[0;0;0];  % prismatic revolute
wee = R4ee.'*w4+[0;0;0]; %End Effector

%calculating linear velocites
v1 = collect(R01.'*(cross(w0,P1)+v0),[dt1,dt2,dt3,dt4])
v2 = collect(R12.'*(cross(w1,P2)+v1),[dt1,dt2,dt3,dt4])
v3 = collect(R23.'*(cross(w2,P3)+v2),[dt1,dt2,dt3,dt4])
v4 = collect(R34.'*(cross(w3,P4)+v3) +[0;0;dt4],[dt1,dt2,dt3,dt4])
vee = collect(R4ee.'*(cross(w4,Pee)+v4),[dt1,dt2,dt3,dt4])
```

```
%% Combining to derive Jacobian
% extracting coefficients from equations
[J_vee] = equationsToMatrix([vee],[dt1,dt2,dt3,dt4])
[J_wee] = equationsToMatrix([wee],[dt1,dt2,dt3,dt4])

% stacking linear and angular contributions (in frame 4)
J4 = simplify([J_vee;J_wee])
% using Jacobian Rotational Matrix to shift frame to base frame
R0ee = t2r(T0ee)
R4eeJ = [R0ee.',zeros(3);zeros(3),R0ee.']
J0 = simplify(R4eeJ*J4)
```

```
J0 =

[- L2*sin(t1 + t2) - L1*sin(t1), -L2*sin(t1 + t2), 0,  0]
[  L2*cos(t1 + t2) + L1*cos(t1),  L2*cos(t1 + t2), 0,  0]
[                             0,                0, 0, -1]
[                             0,                0, 0,  0]
[                             0,                0, 0,  0]
[                             1,                1, 1,  0]
```

1.2 Force Propagation

The second method used to derive the Jacobian was force propagation, which involved back propagating forces from the end effector to the first link. The equivalent equation follows:

$$\tau = J^T f$$

$${}^i f_i = {}^i_{i+1}R^{i+1} f_{i+1}$$

$${}^i n_i = {}^i_{i+1}R^{i+1} n_{i+1} + {}^i P_{i+1} \times {}^i f_i$$

Assuming the applied loads are as follows:

$$f_{ee} = \begin{bmatrix} f_x & f_y & f_z \end{bmatrix}^T$$

$$n_{ee} = \begin{bmatrix} n_x & n_y & n_z \end{bmatrix}^T$$

The following forces and moments were back propagated:

```
% Force Propagation
syms fx fy fz nx ny nz
fee = [fx; fy; fz];
nee = [nx; ny; nz];
% See discussion notes on slide 26!  % torque propagation
f4 = simplify([R4ee.'*fee]);         n4 = simplify([R4ee.'*nee + cross(Pee,f4)]);
f3 = simplify([R34.'*f4]);           n3 = simplify([R34.'*n4 + cross(P4,f3)]);
f2 = simplify([R23.'*f3]);           n2 = simplify([R23.'*n3 + cross(P3,f2)]);
f1 = simplify([R12.'*f2]);           n1 = simplify([R12.'*n2 + cross(P2,f1)]);
```

However, the only relevant forces and moments to us are the ones that are acting against our revolute joints, which means only those that act in the respective z axis of each frame. As such, we need to multiply our results to only evaluate those contributions:

$$\tau_i = {}^i n_i^T \hat{z}_i = \begin{bmatrix} {}^i n_{ix} & {}^i n_{iy} & {}^i n_{iz} \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

$$f_i = {}^i f_i^T \hat{z}_i = \begin{bmatrix} {}^i f_{ix} & {}^i f_{iy} & {}^i f_{iz} \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

```
% For revolute joints we only want the torques on z components
tq1 = collect([n1.']*[0;0;1]);
tq2 = collect([n2.']*[0;0;1]);
tq3 = collect([n3.']*[0;0;1]);
% For prsimatic joints, we only want forces on z directions as well
tq4 = collect([f4.']*[0;0;1]);
```

Finally, we connected the solved taus together and extracted the coefficients, solving for our transposed Jacobian matrix as described in the original equation. To transform this matrix to the 0-reference frame as a regular Jacobian, we first need to take the transpose, and then multiply the transpose by the Jacobian Rotational matrix we solved for in Part 1.1:

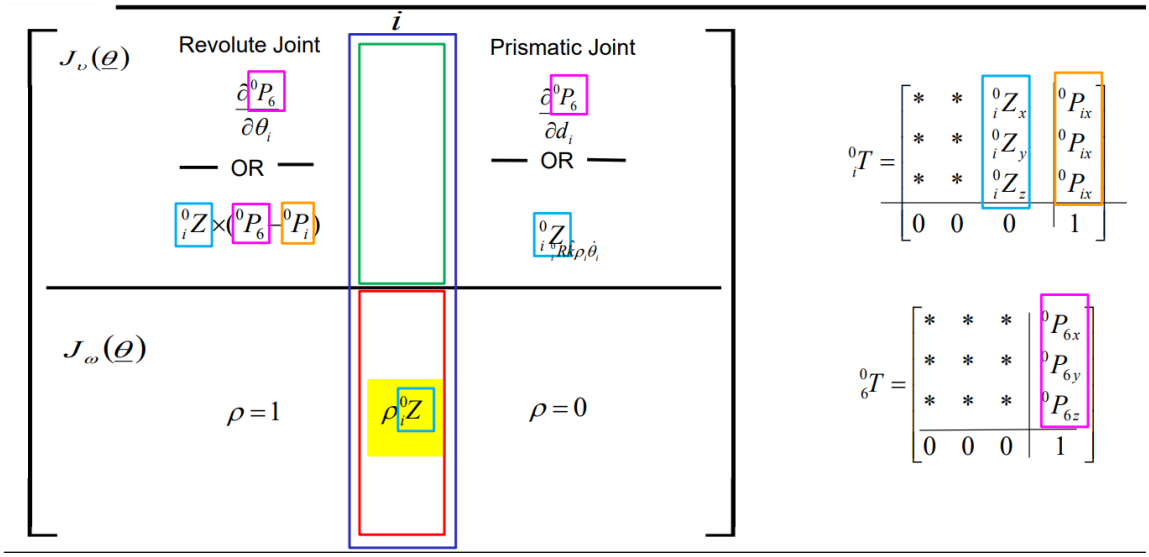
```
%% Form Jacobian
J4_FP_T = equationsToMatrix([tq1;tq2;tq3;tq4],[fx fy fz nx ny nz]); % Fill this part
J4_FP = [J4_FP_T.']; % Fill this part
%% Transform to Base Frame
% Now lets transform to base frame
J0_FP = simplify(R4eeJ * J4_FP);
```

Ultimately, we received the same Jacobian we solved for in Part 1.1, verifying our results:

```
[- L2*sin(t1 + t2) - L1*sin(t1), -L2*sin(t1 + t2), 0,  0]
[  L2*cos(t1 + t2) + L1*cos(t1),  L2*cos(t1 + t2), 0,  0]
[                             0,                0, 0, -1]
[                             0,                0, 0,  0]
[                             0,                0, 0,  0]
[                             1,                1, 1,  0]
```

Part 1.3 Explicit Differentiation

The third way to derive the Jacobian involves explicitly relating the links all in one matrix according to the following rules:



The first step of the method involved first collecting all the colored parts above. The blue, purple, and orange on the right-hand side were collected by solving for the transformation matrices and then extracting the information:

```
%% Direct Differentiation
% for explicit differentiation
T02 = simplify(T01*T12);
T03 = simplify(T02*T23);
T04 = simplify(T03*T34);

% Extracting rotation and position info
[R01,P01] = tr2rt(T01);
[R02,P02] = tr2rt(T02);
[R03,P03] = tr2rt(T03);
[R04,P04] = tr2rt(T04);
[R0ee,P0ee] = tr2rt(T0ee);
```

```
% Extracting Zs from rotation info
z01 = R01(:,3)
z02 = R02(:,3)
z03 = R03(:,3)
z04 = R04(:,3)
z0ee = R0ee(:,3)
```

Using these values, the upper half of the Jacobian containing the linear contributions were solved for every link:

```
% Linear Jacobian stuff
v1_dd = cross(z01,P0ee-P01)
v2_dd = cross(z02,P0ee-P02)
v3_dd = cross(z03,P0ee-P03)
v4_dd = z04
vee_dd = cross(z0ee,P0ee-P0ee)
```

Next, the lower half of the Jacobian containing the angular contributions were solved for by applying the rules stated in the diagram above:

```
% Angular Jacobian stuff
w1_dd = z01
w2_dd =z02
w3_dd = z03
w4_dd = [0;0;0]
wee_dd = [0;0;0]
```

Finally, the two halves were squished together, forming the answer Jacobian matrix:

```
% Squishing together
J0_vdd = [v1_dd v2_dd v3_dd v4_dd]
J0_wdd = [w1_dd w2_dd w3_dd w4_dd]
J0_dd = simplify([J0_vdd;J0_wdd])
```

```
J0_dd =

[- L2*sin(t1 + t2) - L1*sin(t1), -L2*sin(t1 + t2), 0,  0]
[  L2*cos(t1 + t2) + L1*cos(t1),  L2*cos(t1 + t2), 0,  0]
[                             0,                0, 0, -1]
[                             0,                0, 0,  0]
[                             0,                0, 0,  0]
[                             1,                1, 1,  0]
```

Part 2: Singularities

2.1 Singularity Configuration

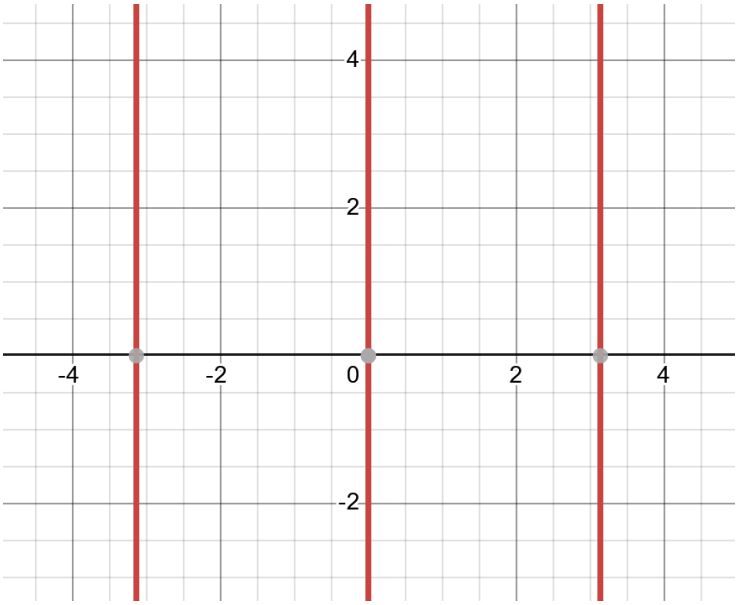
Configurations of joint angles are defined to reach singularity when the determinant of the Jacobian equals zero. This makes intuitive sense; if the Jacobian is the map from joint angles to task space, then when it maps to a lower dimension a rank is lost and a singularity is reached. In order to solve for this, we first solved for the determinant of the Jacobian. Since determinants can only be determined for square matrices, we cut out the two rows of zeroes corresponding to the roll and pitch orientations (which we cannot control with our SCARA arm and are therefore irrelevant) to create a 4x4 matrix and solved for the determinant:

```
%% Singularity

% Jacobian Final
JF = [J0(1,:);J0(2,:);J0(3,:);J0(6,:)] %taking out rows of all zeros
JF_det = det(JF)
```

The following determinant was solved for:

```
JF_det =

L1*L2*sin(t1 + t2)*cos(t1) - L1*L2*cos(t1 + t2)*sin(t1)
```

Since the angle of joint 1 – t1 – is freely revolving around the origin, it doesn't affect the singularity discussion. By setting the equation of the determinant above equal to zero and graphing in Desmos (x-axis is t2, y-axis is determinant), we get the following:



As seen above, the roots of the equation (where the determinant is zero) lie on k(pi), where k is an integer. As such, the singularity lies where t2 is pi, when the arm is folded on itself, and 0, when the arm is fully outstretched.

2.2 Analytical Implications

The first implication of reaching a singularity configuration is that the SCARA loses a degree of freedom. This can be observed by setting t2 to zero in the Jacobian, which reduces the matrix to the following:

$$ {}^{0}J_4 = \begin{bmatrix} -L2\sin(\theta_1) - L1\sin(\theta_1) & -L2\sin(\theta_1) & 0 & 0 \\ L2\cos(\theta_1) + L1\cos(\theta_1) & L2\cos(\theta_1) & 0 & 0 \\ 0 & 0 & 0 & -1 \\ 1 & 1 & 1 & 0 \end{bmatrix} $$

Solving for Vx and Vy based on the Jacobian relationship:

$$ v_x = [-L2\sin(\theta_1) - L1\sin(\theta_1)]\dot{\theta}_1 + [-L2\sin(\theta_1)]\,\dot{\theta}_2 $$

$$ v_y = [L2\cos(\theta_1) + L1\cos(\theta_1)]\dot{\theta}_1 + [L2\cos(\theta_1)]\,\dot{\theta}_2 $$

$$\frac{v_x}{-L2\sin(\theta_1)} = \left[1 + \frac{L1}{L2}\right]\dot{\theta}_1 + \dot{\theta}_2$$

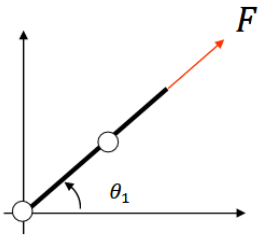
$$\frac{v_y}{L2\cos(\theta_1)} = \left[1 + \frac{L1}{L2}\right]\dot{\theta}_1 + \dot{\theta}_2$$

$$\frac{v_y}{L2\cos(\theta_1)} = \frac{v_x}{-L2\sin(\theta_1)}$$

As seen, Vy and Vx cannot move independently of each other, and the SCARA robot can only move in the radial direction effectively losing a degree of freedom.

The second implication is that the SCARA can lock up – that is, get stuck in a configuration in which its joints cannot actuate a torque to affect movement in response to a force:

$$\theta_1 = \theta; \quad \theta_2 = 0$$



$$^0F = \begin{bmatrix} Fc_1 \\ Fs_1 \end{bmatrix}$$

In this scenario:

$$\underline{^0\tau} = {}^0J(\underline{\theta})^T\,\underline{^0F} = \begin{bmatrix} -L_1s_1 - L_2s_{12} & L_1c_1 + L_2c_{12} \\ -L_2s_{12} & L_2c_{12} \end{bmatrix}$$

$$\underline{^0\tau} = {}^0J(\underline{\theta})^T\,\underline{^0F} = \begin{bmatrix} -L_1s_1 - L_2s_{12} & L_1c_1 + L_2c_{12} \\ -L_2s_{12} & L_2c_{12} \end{bmatrix}\begin{bmatrix} Fc_1 \\ Fs_1 \end{bmatrix} =$$

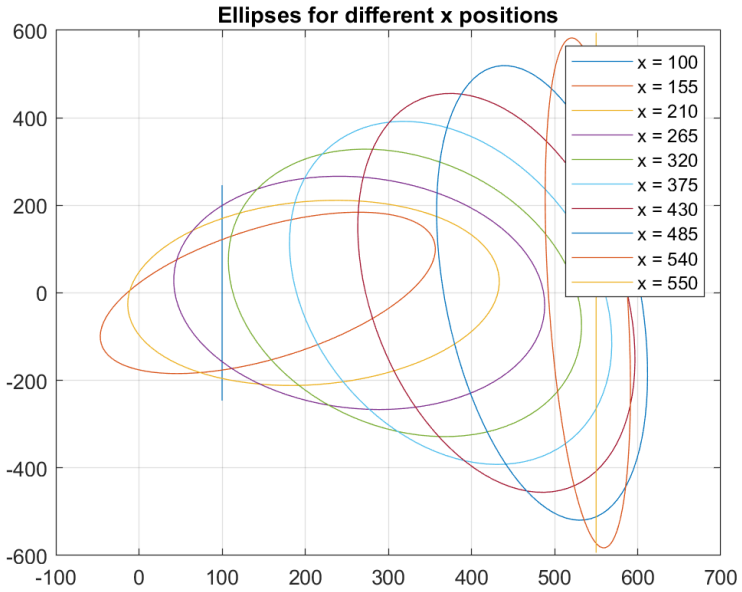
$$\begin{bmatrix} -Fs_1c_1(L_1 + L_2) + Fs_1c_1(L_1 + L_2) \\ -Fs_1c_1(L_2) + Fs_1c_1(L_2) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Here, the torque is zero in response to a finite force, locking the SCARA up.

Part 3: Jacobian Ellipsoids

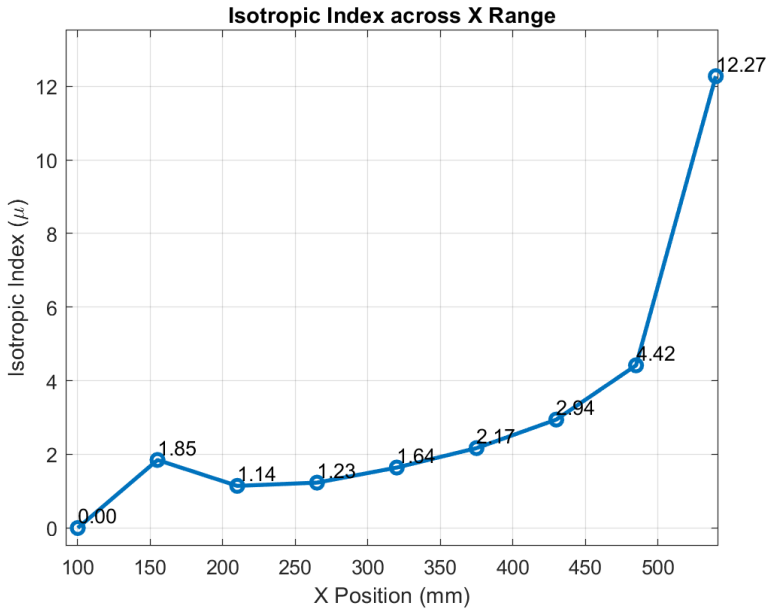
Manipulability ellipsoids were created for a simplified, 2R model of the SCARA robot above with $L1 = 325\ mm$ and $L2 = 225\ mm$.

To create the ellipsoid graph for a 2R robotic manipulator, range of x-coordinates was defined, keeping the y-coordinate fixed. Because the lower bound for the workspace is 100mm, the x range was started there instead of at 0mm. For each x-coordinate, the robot's joint angles was calculated using inverse kinematics. Subsequently, we computed the Jacobian matrix at these angles and derived its eigenvalues and eigenvectors. These were used to calculate the isotropic index and plot ellipses representing the manipulability at each position.



X_Value	Eigenvalue1	Eigenvalue2
100	-7.1746e-43	60625
155	16910	57740
210	41011	53714
265	48016	72834
320	41415	1.1161e+05
375	33513	1.5774e+05
430	24350	2.1118e+05
485	13931	2.7192e+05
540	2257.2	3.3997e+05

The isotropic indices were also plotted against the x-coordinates to visualize how manipulability varies across the robot's workspace.



As we can see, the lowest isotropic index (closest to sphere) was at the 210mm x range, right around $\theta_1 = 0.76$ and $\theta_2 = -2.45$. This is the optimal configuration of the robot arm and should be centered around when designing the task.

Methods:

```
%% Ellipsoid
% Define link lengths and x range
L1 = 325; L2 = 225;
x_range = 100:55:550; % Adjusted to include the endpoint if needed

% Initialize storage for table and isotropic indices
x_values = x_range; % Directly using x_range for table
eigenvalue1 = zeros(length(x_range), 1);
eigenvalue2 = zeros(length(x_range), 1);
isotropic_indices = zeros(1, length(x_range));

% Define symbolic variables for differentiation
syms t1 t2 real;

% Equations for 2R model
xeq = L1*cos(t1) + L2*cos(t1 + t2);
yeq = L1*sin(t1) + L2*sin(t1 + t2);

% Jacobian for 2R Model
J2R = [diff(xeq,t1) diff(xeq,t2); diff(yeq,t1) diff(yeq,t2)];
```

```
% Loop over x positions
figure; % For ellipse plots
hold on;
for idx = 1:length(x_range)
    i = x_values(idx);
    y = 0; % Set y-coordinate to 0
    [t1_curr, t2_curr] = IK2R(i, 0, L1, L2); % Implement IK2R
    J2R_curr = double(subs(J2R, [t1, t2], [t1_curr, t2_curr]));
    [f, g] = eig(J2R_curr*J2R_curr.');
    evals = diag(g);
    eigenvalue1(idx) = evals(1);
    eigenvalue2(idx) = evals(2);
    isotropic_indices(idx) = sqrt(max(evals)/min(evals));
    [X_c, Y_c] = ellipse(f, g, i, y); % Implement ellipse
    plot(X_c, Y_c, 'DisplayName', ['x = ', num2str(i)]);
end
title("Ellipses for different x positions");
legend show;
grid on;


% Plotting isotropic indices
figure; % New figure for isotropic index plot
plot(x_values, isotropic_indices, '-o', 'LineWidth', 2, 'MarkerFaceColor', 'r');
xlabel('X Position (mm)');
ylabel('Isotropic Index (\mu)');
title('Isotropic Index across X Range');
grid on;

% Create and display the table
EigenvalueTable = table(x_values', eigenvalue1, eigenvalue2, ...
                        'VariableNames', {'X_Value', 'Eigenvalue1', 'Eigenvalue2'});
disp(EigenvalueTable);
```

```
%% Ellipsoid functions
% Function to compute the ellipse coordinates
function [X, Y] = ellipse(f, g, x, y) % f: eigenvectors, g: eigenvalues
    if sqrt(g(1,1)) > sqrt(g(2,2))
        a = sqrt(g(1,1)); % Major axis length
        b = sqrt(g(2,2)); % Minor axis length
        angle = vpa(atan2(f(2,1), f(1,1))); % Orientation angle
    else
        a = sqrt(g(2,2)); % Major axis length
        b = sqrt(g(1,1)); % Minor axis length
        angle = vpa(atan2(f(2,2), f(1,2))); % Orientation angle
    end

    % Rotation matrix for the ellipse
    rot_mat = [cos(angle), -sin(angle); sin(angle), cos(angle)];

    % Generate ellipse points
    t = linspace(0, 2*pi, 100);
    X_ut = a * cos(t);
    Y_ut = b * sin(t);

    X = []; Y = [];
    % Rotate and translate ellipse points
    for i = 1:length(t)
       coord_new = rot_mat * [X_ut(i); Y_ut(i)];
       X = [X, coord_new(1) + x];
       Y = [Y, coord_new(2) + y];
    end
end

% Function to solve inverse kinematics for 2R manipulator

function [t1, t2] = IK2R(x, y, L1, L2)
    % Compute the distance to the target point
    r = sqrt(x^2 + y^2);

    % Check if the target point is reachable
    if r > L1 + L2
        error('Target point is outside the reachable workspace.');
    end

    % Law of cosines to find t2
    cos_t2 = (x^2 + y^2 - L1^2 - L2^2) / (2 * L1 * L2);
    t2 = atan2(sqrt(1 - cos_t2^2), cos_t2);  % Elbow down solution

    % Find t1
    k1 = L1 + L2 * cos(t2);
    k2 = L2 * sin(t2);
    t1 = atan2(y, x) - atan2(k2, k1);

end
```