**Robot of choice:**

[UFACTORY 850 Robotic Arm (6 DoF)](https://www.robotshop.com/products/ufactory-850-robotic-arm-6-dof?srsltid=AfmBOorizQZb91fL49T7mWHgCyuXeY46ynQjKfjf5H8r7qRLgQC09v1h82E)

**Joint Range/Limits:**

|  |  |
| --- | --- |
| Joint | Range |
| J1 |  |
| J2 |  |
| J3 |  |
| J4 |  |
| J5 |  |
| J6 |  |

**D-H Table:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Kinematics |  | d(mm) | (deg) | a(mm) |
| Joint1 | **0** | **364** | **90** | **0** |
| Joint2 | **90** | **0** | **180** | **390** |
| Joint3 | **90** | **0** | **-90** | **150** |
| Joint4 | **0** | **426** | **-90** | **0** |
| Joint5 | **0** | **0** | **90** | **0** |
| Joint6 | **0** | **90** | **0** | **0** |

**Equations:**

Sphere equation:

Normal to sphere:

Distance into sphere:

Parameterized circle:

%% --- Robot Model Implementation and FK Verfication ---

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

disp('Robot Model Implementation and FK Verfication...');

% constant for unit conversion

deg = pi/180;

% define links using DH params for UFACTORY 850

L(1) = Link([0, 0.364, 0, 90\*deg], 'offset', 0, 'R');

L(2) = Link([0, 0, 0.390, 0], 'offset', 90\*deg, 'R');

L(3) = Link([0, 0, 0.150, -90\*deg], 'offset', 90\*deg, 'R');

L(4) = Link([0, 0.426, 0, 90\*deg], 'offset', 0, 'R');

L(5) = Link([0, 0, 0, -90\*deg], 'offset', 0, 'R');

L(6) = Link([0, 0.090, 0, 0], 'offset', 0, 'R');

% apply joint limits

L(1).qlim = [-360\*deg, 360\*deg];

L(2).qlim = [-132\*deg, 132\*deg];

L(3).qlim = [-242\*deg, 3.5\*deg];

L(4).qlim = [-360\*deg, 360\*deg];

L(5).qlim = [-124\*deg, 124\*deg];

L(6).qlim = [-360\*deg, 360\*deg];

% create SerialLink object

robot = SerialLink(L, 'name', 'UFACTORY 850');

% verify FK

q\_test = zeros(1,6);

T\_fk\_object = robot.fkine(q\_test);

T\_fk\_matrix = T\_fk\_object.T;

disp('Forward Kinematics Verification at q = [0, 0, 0, 0, 0, 0]:');

disp('End-Effector Pose (T\_fk):');

disp(T\_fk\_matrix);

position\_xyz = T\_fk\_matrix(1:3, 4)';

fprintf('End-Effector Position (x,y,z) in m: [%.4f, %.4f, %.4f]\n', position\_xyz);

%% --- Trajectory Planning ---

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

disp('Trajectory Planning...');

%sphere params in meters

sphere\_center = [-0.6; -0.4; 0.8];

sphere\_radius = 0.15;

%circle params

circle\_normal = [1; 1; 0.45];

circle\_normal = circle\_normal / norm(circle\_normal);

circle\_radius = 0.04;

plane\_offset = sqrt(sphere\_radius^2 - circle\_radius^2);%sphere\_radius \* 0.5;

%calculations

circle\_center = sphere\_center + plane\_offset \* circle\_normal;

%make 2 orthogonal basis vectors (u & v) for circle plane

if abs(circle\_normal(1)) < 0.9

temp = [1; 0; 0];

else

temp = [0; 1; 0];

end

u = cross(circle\_normal, temp);

u = u / norm(u);

v = cross(circle\_normal, u);

num\_points = 100;

time\_parameterization = linspace(0, 2\*pi, num\_points);

% trajectory generation

positions = zeros(3, num\_points);

orientations = zeros(3, 3, num\_points);

for i = 1:num\_points

%calculate position on circle

positions(:, i) = circle\_center + circle\_radius \* cos(time\_parameterization(i)) \* u + circle\_radius \* sin(time\_parameterization(i)) \* v;

%calculate normal vector

normal\_vector = positions(:,i ) - sphere\_center;

normal\_vector = normal\_vector / norm(normal\_vector);

%z-axis pointing to sphere center

z\_axis = -normal\_vector;

%x-axis perpendicular to normal and circle normal

x\_axis = cross(circle\_normal, z\_axis);

if norm(x\_axis) < 1e-6

x\_axis = cross([1; 0; 0], z\_axis);

if norm(x\_axis) < 1e-6

x\_axis = cross([0; 1; 0], z\_axis);

end

end

x\_axis = x\_axis / norm(x\_axis);

%y-axis complete right handed coordinate system

y\_axis = cross(z\_axis, x\_axis);

%store orientation matrix

orientations(:, :, i) = [x\_axis, y\_axis, z\_axis];

end

fprintf('======================================\n');

fprintf('Sphere Center: [%.3f, %.3f, %.3f] m\n', sphere\_center);

fprintf('Sphere Radius: %.3f m\n', sphere\_radius);

fprintf('Circle Center: [%.3f, %.3f, %.3f] m\n', circle\_center);

fprintf('Circle Radius: %.3f m\n', circle\_radius);

fprintf('Circle Normal: [%.3f, %.3f, %.3f]\n', circle\_normal);

fprintf('Num Trajectory Points: %d\n', num\_points);

fprintf('======================================\n\n');

%% --- Inverse Kinematics ---

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

disp('Inverse Kinematics...');

%preallocate joint angles arr

q\_trajectory = zeros(num\_points, 6);

%inital guess for IK

q0 = [0, -45\*deg, -45\*deg, 0, -45\*deg, 0];

%solve IK for each trajectory pt

for i = 1:num\_points

%create homogeneous transformation matrix

T\_desired = [orientations(:,:,i), positions(:,i); 0 0 0 1];

%solve IK using ikine w/ mask

q\_sol = robot.ikine(T\_desired, q0, 'mask', [1 1 1 1 1 1]);

%check if solution is within joint lims

if isempty(q\_sol) || any(isnan(q\_sol))

fprintf('IK failed at point %d. using previous solution.', i);

if i > 1

q\_sol = q\_trajectory(i-1, :);

else

q\_sol = q0;

end

end

%store solution

q\_trajectory(i,:) = q\_sol;

%use curr solution as init guess for next iteration

q0 = q\_sol;

end

disp('\nInverse Kinematics Complete');

%verify joint lims

fprintf('======================================\n');

disp('Joint limit verification:');

for j = 1:6

q\_min = min(q\_trajectory(:, j));

q\_max = max(q\_trajectory(:, j));

limit\_min = L(j).qlim(1);

limit\_max = L(j).qlim(2);

fprintf('Joint %d: [%.2f, %.2f] deg | Limits: [%.2f, %.2f] deg\n', j, q\_min/deg, q\_max/deg, limit\_min/deg, limit\_max/deg);

if q\_min < limit\_min || q\_max > limit\_max

fprintf('Joint %d exceeds limits', j);

end

end

fprintf('======================================\n\n');

%% --- Animation ---

%%%%%%%%%%%%%%%%%%%%

disp('Animation...');

figure('Name', 'Robot Drawing Circle on Sphere', 'NumberTitle', 'off');

hold on;

grid on;

axis equal;

xlabel('X');

ylabel('Y');

zlabel('Z');

title('UFACTORY xARM 850 - Circular Path on Sphere Surface');

%set axis limits for better visualization

axis([sphere\_center(1)-0.6, sphere\_center(1)+0.6, sphere\_center(2)-0.6, sphere\_center(2)+0.6, 0, sphere\_center(3)+0.6]);

%draw sphere

[x\_sphere, y\_sphere, z\_sphere] = sphere(30);

surf(sphere\_center(1) + sphere\_radius \* x\_sphere, sphere\_center(2) + sphere\_radius \* y\_sphere, sphere\_center(3) + sphere\_radius \* z\_sphere, 'FaceAlpha', 0.3, 'EdgeAlpha', 0.1, 'FaceColor', 'red');

%draw desired circle path

plot3(positions(1,:), positions(2,:), positions(3,:), 'r--', 'LineWidth', 2, 'DisplayName', 'Desired Path');

%actual path

trace\_line = plot3(nan, nan, nan, 'b-', 'LineWidth', 2, 'DisplayName', 'Actual Path');

legend('Location', 'northeast');

%animation loop

for i = 1:num\_points

%plot robot at current config

robot.plot(q\_trajectory(i,:), 'workspace', [sphere\_center(1)-1, sphere\_center(1)+1, sphere\_center(2)-1, sphere\_center(2)+1, 0, sphere\_center(3)+1], 'trail', 'b-', 'nobase');

%update trace line

set(trace\_line, 'XData', positions(1, 1:i), 'YData', positions(2,1:i), 'ZData', positions(3,1:i));

pause(0.05);

end

fprintf('======================================\n');

disp('Animation Complete');

fprintf('======================================\n\n');

%% --- Verification ---

%%%%%%%%%%%%%%%%%%%%%%%

disp('Verification...');

%calculate actual end effector pos

actual\_positions = zeros(3, num\_points);

for i = 1:num\_points

T\_actual = robot.fkine(q\_trajectory(i, :));

actual\_positions(:,i) = T\_actual.t;

end

%calculate pos errors

position\_errors = vecnorm(positions - actual\_positions);

mean\_error = mean(position\_errors);

max\_error = max(position\_errors);

fprintf('======================================\n');

disp('Verification Results:');

fprintf('Mean Position Error: %.4f mm\n', mean\_error \* 1000);

fprintf('Max Position Error: %.4f mm\n', max\_error \* 1000);

fprintf('======================================\n\n');

**Workload:**

Bengaly:

* Robot choice
* Equations: 85%
* Programming: 15%

Vladyslava:

* Equations: 15%
* Programming: 85%
* Animation
* TImeline