ESE 5582 Homework 1 / Data Driven Control /

Seif Elkhashab / 02-12-2024 /

Diffrential Equations Using Data Methods

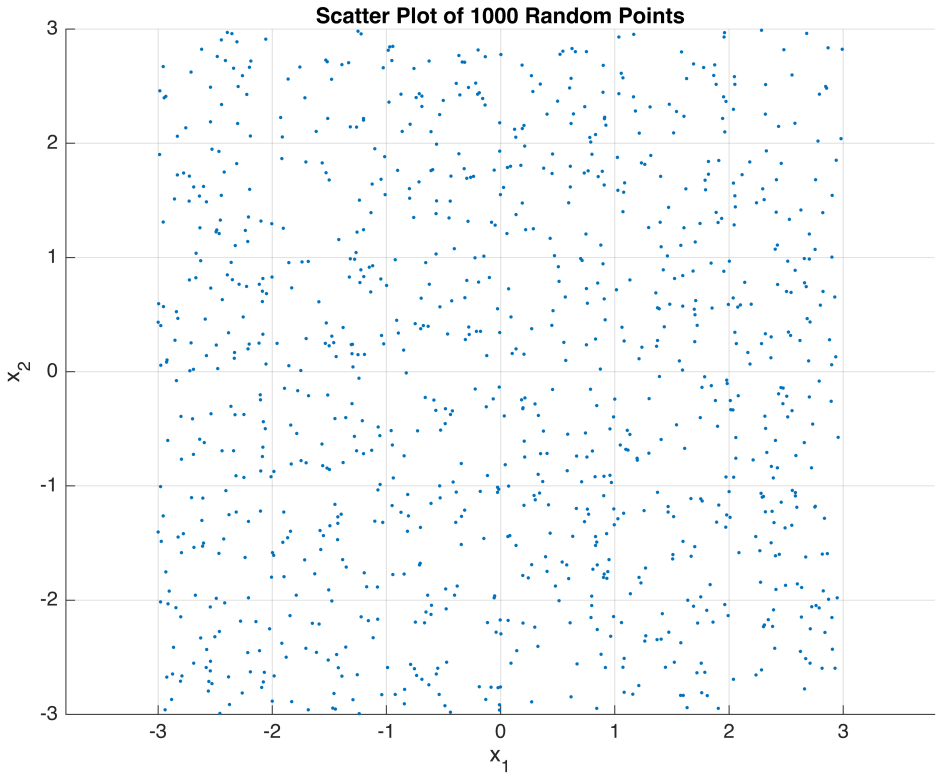
1. Consider the classical Van der Pol system

$$\dot{x}_1 = x_2,$$

$$\dot{x}_2 = (1 - x_1^2)x_2 - x_1.$$

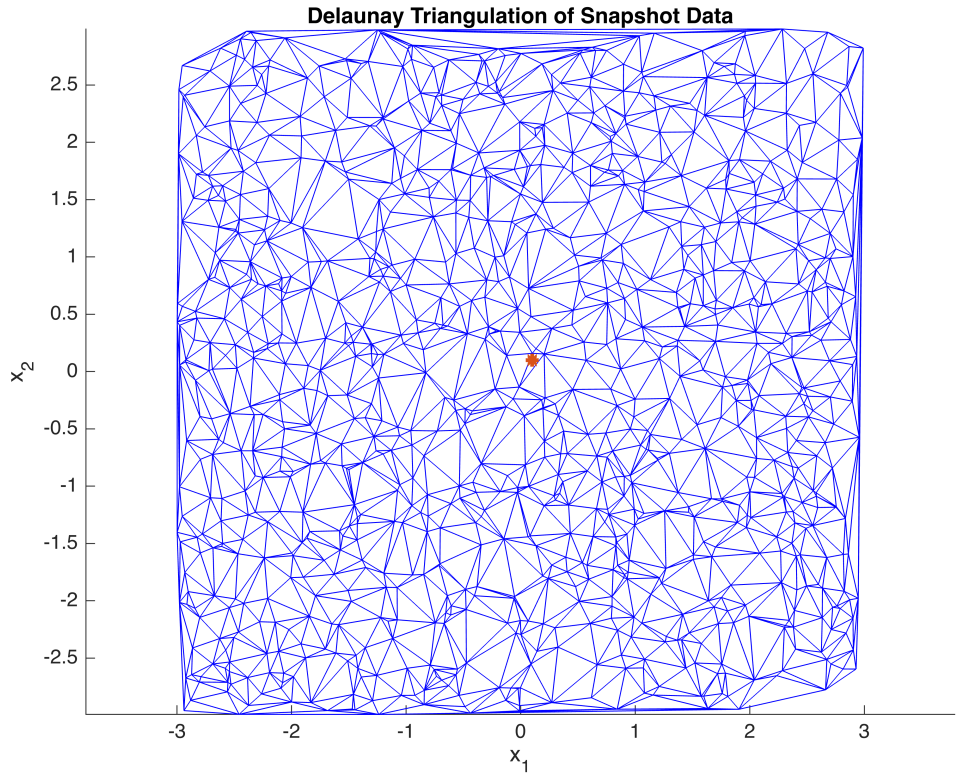
(a) Let T = 0.1 and generate snapshot data on the domain [-3; 3] X [-3; 3].

```
% Time Step
T = 0.1;

% Number of points
N = 1000;

% Generate random points in the range [0, 1] and scale them to [-3, 3]
x1 = 6 * rand(N, 1) - 3;
x2 = 6 * rand(N, 1) - 3;

% Plot as a scatter plot
figure;
scatter(x1, x2,'.');
title('Scatter Plot of 1000 Random Points');
xlabel('x_1');
ylabel('x_2');
xlim([-3, 3]);
ylim([-3, 3]);
axis equal;
grid on;
```



(b) Familiarize yourself with the MATLAB function delaunay and use it to generate a Delaunay triangulation of the snapshot data.

```
% Combine x1 and x2 into a single matrix where each row is a point
points = [x1, x2];

% Generating the Delaunay triangulation
tri = delaunay(points(:,1), points(:,2));

% Choosing a random point
Point_rand = [0.1,0.1];

% Plotting the triangulation
figure;
hold on;
triplot(tri, points(:,1), points(:,2));
plot(Point_rand(1),Point_rand(2),'*', 'MarkerFaceColor', 'red',
'LineWidth',5);
title('Delaunay Triangulation of Snapshot Data');
xlabel('x_1');
ylabel('x_2');
axis equal;
```

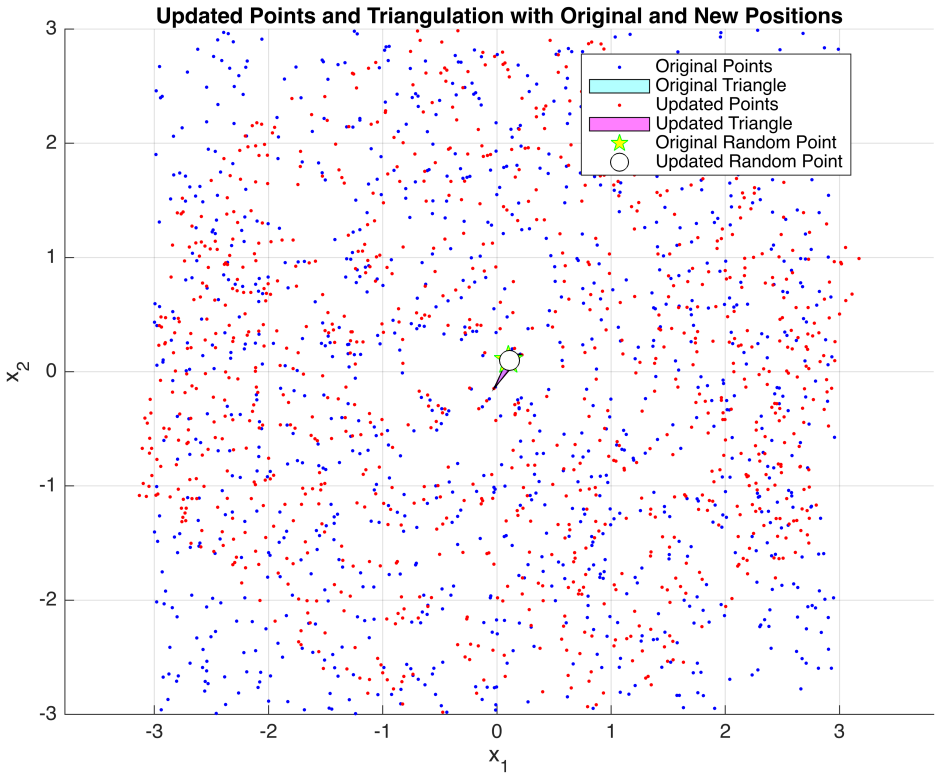


(c) Use the interpolation idea introduced in class to obtain a one-step prediction of any point $x \in [-3,3] \times [-3,3]$.

```
% Update point positions using ODE
newPoints = zeros(size(points));
for i = 1:size(points, 1)
    [t, y] = ode45(@vdp1, [0, T], points(i, :));
    newPoints(i, :) = y(end, :);
end

% Find the triangle containing the random point in the original
triangulation
TRI = triangulation(tri, points);
ti = pointLocation(TRI, Point_rand);

% Calculate transformation matrices for each triangle
transformations = zeros(size(tri, 1), 6); % Initialize array for
transformation matrices
for i = 1:size(tri, 1)
    origTriPoints = points(tri(i, :), :);
    newTriPoints = newPoints(tri(i, :), :);

    % Construct matrices for original and updated triangle points
```

```
    A = [origTriPoints, ones(3, 1)];
    B = [newTriPoints, ones(3, 1)];

    % Calculate transformation matrix using inverse
    X = B' / A';

    % Calculate the transformation matrix and reshape it into a 1x6 row
    transformations(i, :) = reshape(X(1:2, :)', 1, []);

end

% Update the position of the random point using the transformation of its
containing triangle
if ~isempty(ti)
    transformation = transformations(ti, :);
    Point_rand_new = [Point_rand, 1] * [transformation(1:3);
transformation(4:6); 0 0 1]';
    Point_rand_new = Point_rand_new(1:2);
else
    Point_rand_new = Point_rand;
end

figure;
hold on;

% Plot original points
scatter(points(:,1), points(:,2), '.', 'MarkerEdgeColor', 'blue',
'DisplayName', 'Original Points');

% Highlight the original triangle containing the random point
if ~isempty(ti)
    origTriPoints = points(tri(ti, :), :);
    patch('Faces', [1 2 3], 'Vertices', origTriPoints, 'FaceColor', 'cyan',
'FaceAlpha', 0.3, 'EdgeColor', 'black', 'DisplayName', 'Original Triangle');
end

% Plot updated points
scatter(newPoints(:,1), newPoints(:,2), '.', 'MarkerEdgeColor', 'red',
'DisplayName', 'Updated Points');

% Highlight the updated triangle containing the new position of the random
point
newTRI = triangulation(delaunay(newPoints(:,1), newPoints(:,2)), newPoints);
newTi = pointLocation(newTRI, Point_rand_new);
if ~isempty(newTi)
    newTriPoints = newTRI.Points(newTRI.ConnectivityList(newTi, :), :);
    patch('Faces', [1 2 3], 'Vertices', newTriPoints, 'FaceColor',
'magenta', 'FaceAlpha', 0.5, 'EdgeColor', 'black', 'DisplayName', 'Updated
Triangle');
end
```

```
% Plot the original random point with increased size and distinct color
plot(Point_rand(1), Point_rand(2), 'p', 'MarkerSize', 15,
'MarkerEdgeColor', 'green', 'MarkerFaceColor', 'yellow', 'DisplayName',
'Original Random Point');

% Plot the new position of the random point
plot(Point_rand_new(1), Point_rand_new(2), 'o', 'MarkerSize', 10,
'MarkerEdgeColor', 'black', 'MarkerFaceColor', 'white', 'DisplayName',
'Updated Random Point');

% Finalize the plot
xlim([-3, 3]);
ylim([-3, 3]);
title('Updated Points and Triangulation with Original and New Positions');
xlabel('x_1');
ylabel('x_2');
axis equal;
grid on;
legend('Location', 'best');
hold off;
```

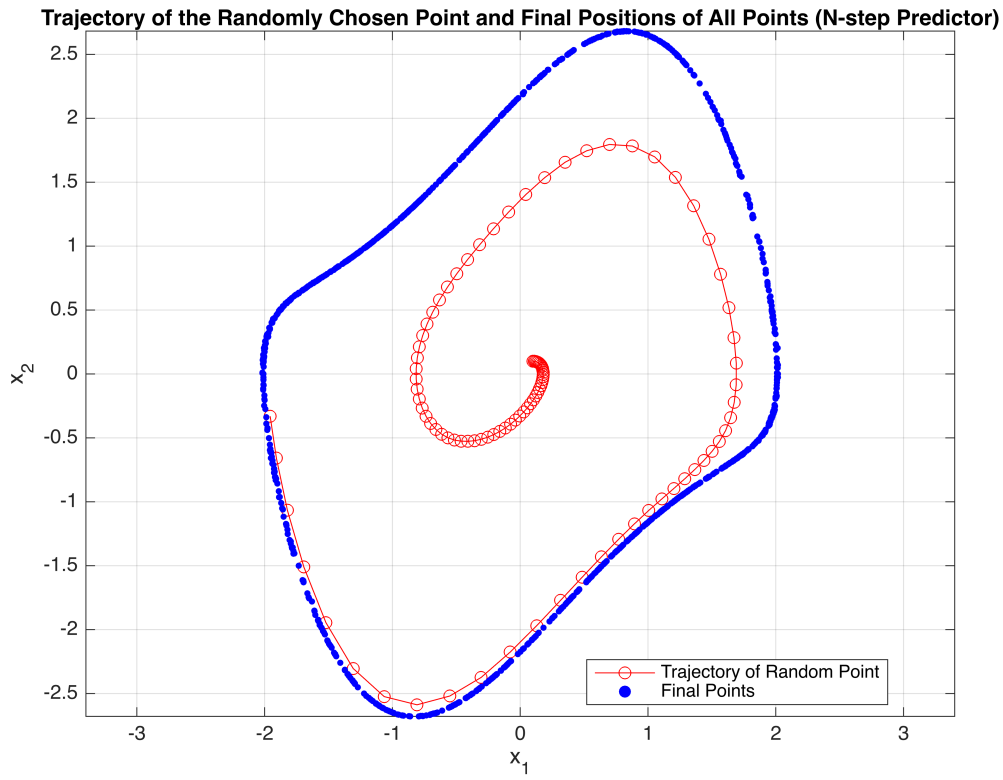


(d) Generalize your result from the previous task to obtain an N-step predictor.

```
% Define the number of prediction steps
N_steps = 100;
```

```
% Initialize an array to store the trajectory of the randomly chosen point
Point_rand_trajectory = zeros(N_steps+1, 2);
Point_rand_trajectory(1, :) = Point_rand;

% Maintain the index of the randomly chosen point
randomPointIndex = find(ismember(points, Point_rand, 'rows'));

% Loop over N steps for prediction
for step = 1:N_steps
    % Update all points using ode45
    for i = 1:size(points, 1)
        if i ~= randomPointIndex  % Exclude the randomly chosen point by
index
            [~, y] = ode45(@vdp1, [0, T], points(i, :));
            points(i, :) = y(end, :);
        end
    end

    % Update the Delaunay triangulation for the current point configuration
    tri = delaunay(points(:,1), points(:,2));
    TRI = triangulation(tri, points);

    % Find the triangle that contains the Point_rand using the updated
triangulation
    ti = pointLocation(TRI, Point_rand);

    if ~isempty(ti)  % Check if the point is inside any triangle
        transformation = transformations(ti, :);

        % Apply the transformation to Point_rand
        Point_rand_new = [Point_rand, 1] * [transformation(1:3);
transformation(4:6); 0 0 1]';
        Point_rand = Point_rand_new(1:2); % Update Point_rand
    end

    % Store the updated position of Point_rand
    Point_rand_trajectory(step+1, :) = Point_rand;
end

% Update point positions using ODE for final positions after N_steps
finalPoints = zeros(size(points)); % Initialize array for final positions
for i = 1:size(points, 1)
    [~, y] = ode45(@vdp1, [0, T * N_steps], points(i, :));
    finalPoints(i, :) = y(end, :);
end

% Visualization of the trajectory of the randomly chosen point and final
positions of all points
figure;
```

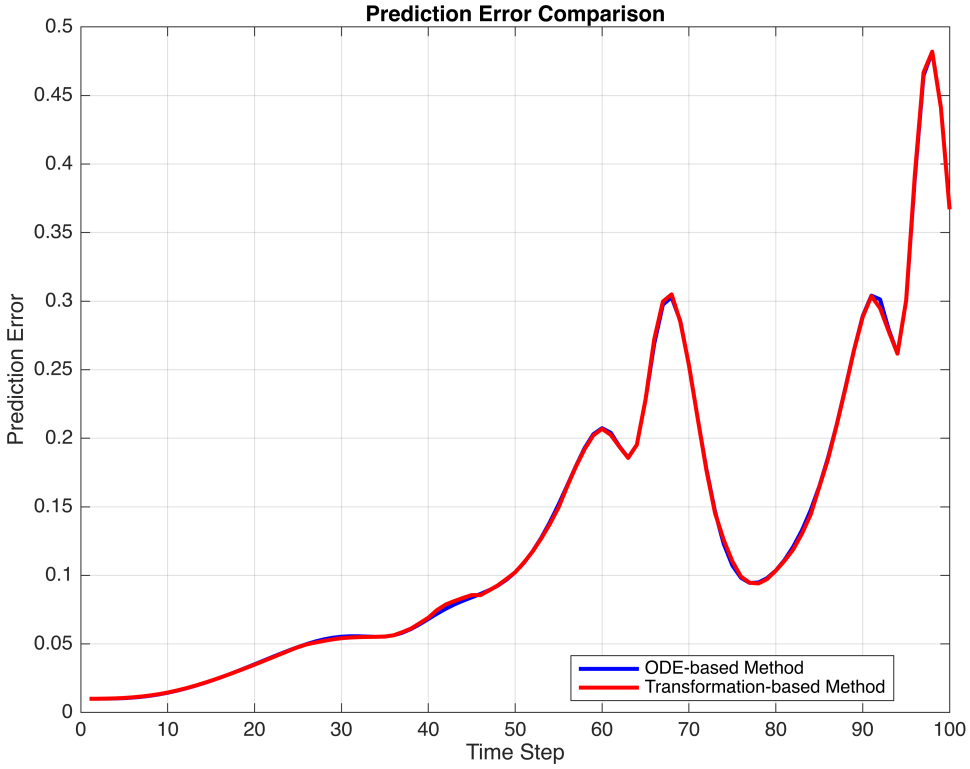
```
plot(Point_rand_trajectory(:,1), Point_rand_trajectory(:,2), 'r-o');
hold on;
scatter(finalPoints(:,1), finalPoints(:,2), 10, 'filled', 'b');
title('Trajectory of the Randomly Chosen Point and Final Positions of All
Points (N-step Predictor)');
xlabel('x_1');
ylabel('x_2');
legend('Trajectory of Random Point', 'Final Points', 'Location', 'Best');
axis equal;
grid on;
hold off;
```



(e) Compare the prediction accuracy with the results obtained using ode45.

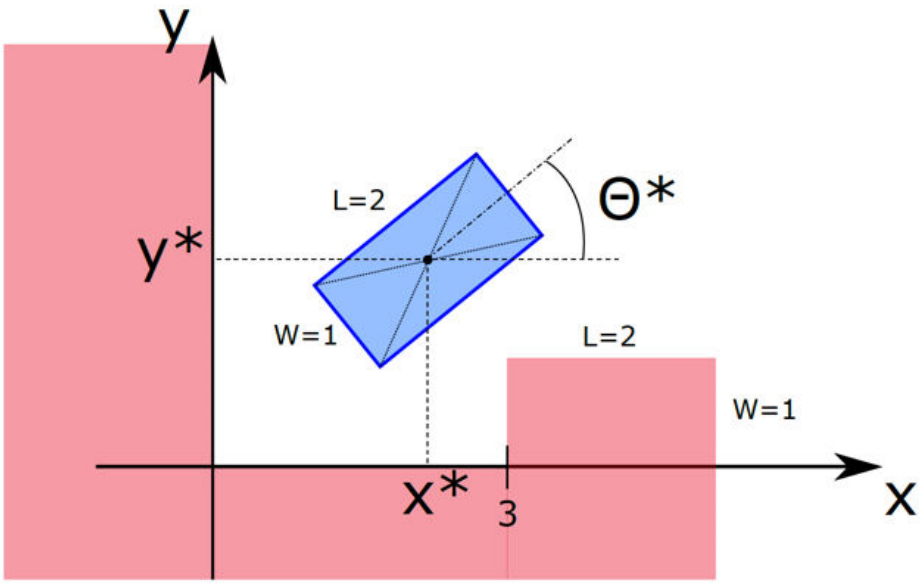
```
% Define the time step and number of steps
T = 0.1;
N_steps = 100;

% Initialize arrays to store the errors for each method
error_ode = zeros(N_steps, 1);
error_transformation = zeros(N_steps, 1);

% Initialize the random point
Point_rand = [0.1, 0.1];

% Maintain the index of the randomly chosen point
```

```
randomPointIndex = find(ismember(points, Point_rand, 'rows'));

% Loop over N steps for prediction
for step = 1:N_steps
    % Update all points using ODE-based method
    [t, y] = ode45(@vdp1, [0, T], Point_rand);
    Point_ode = y(end, :);

    % Update all points using the transformation-based method
    triangleIndex = pointLocation(TRI, Point_rand);
    if ~isempty(triangleIndex)
        transformation = [transformations(triangleIndex,1:3);
transformations(triangleIndex,4:6)];
        Point_transformation = transformation * [Point_rand'; 1];
        Point_transformation = Point_transformation(1:2)';
    else
        Point_transformation = Point_rand;
    end

    % Compute the error for each method
    error_ode(step) = norm(Point_ode - Point_rand);
    error_transformation(step) = norm(Point_transformation - Point_rand);


    Point_rand = Point_ode;
end

% Plot the errors over time
figure;
plot(1:N_steps, error_ode, 'b', 'LineWidth', 2);
hold on;
plot(1:N_steps, error_transformation, 'r', 'LineWidth', 2);
title('Prediction Error Comparison');
xlabel('Time Step');
ylabel('Prediction Error');
legend('ODE-based Method', 'Transformation-based Method', 'Location',
'Best');
grid on;
hold off;
```



There appears to be very low error between the two methods.

2. Consider the following description in the task space:

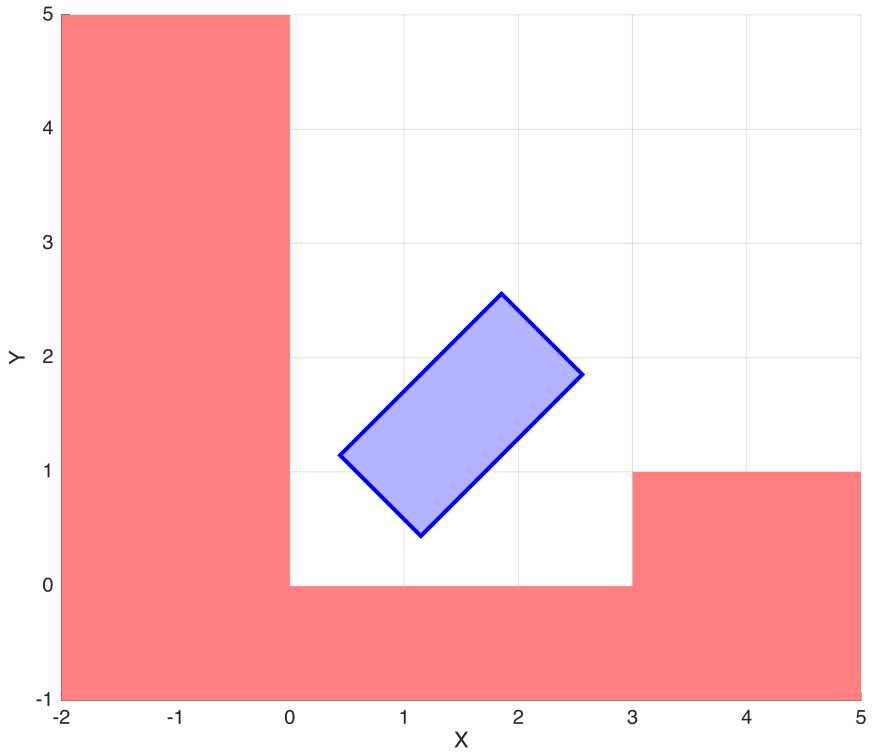


Devise a computational approach to mark all points (x^*, y^*, θ^*) (in the con guration space/state space) that result in a collision of the blue car with the red obstacles (in the

task space). Include your code and an illustration of the point-based description of the constraint in the three-dimensional (x, y, θ)-space.

```
% Ininitial position of the car
x = 1.5;
y = 1.5;
theta = pi / 4;

Wx = linspace(-1,5,101);
Wy = linspace(-1,5,101);
Wtheta = linspace(0,2*pi,100);

% Obstacles (for plotting)
corners_ob_1 = [3  1;
                3 -1;
                5 -1;
                5  1];
corners_ob_2 = [0 5;
               -2 5;
               -2 0;
                0 0];
corners_ob_3 = [3  0;
                3 -1;
               -2 -1;
               -2  0];

states = [x y theta];

% Rectangle parameters
length = 1;
width = 2;
center = [states(1), states(2)];  % Center point (x, y)

% Calculate the corners of the rectangle before rotation
halfWidth = width / 2;
halfLength = length / 2;
corners = [center(1) - halfWidth, center(2) - halfLength;  % Lower left
           center(1) + halfWidth, center(2) - halfLength;  % Lower right
           center(1) + halfWidth, center(2) + halfLength;  % Upper right
           center(1) - halfWidth, center(2) + halfLength]; % Upper left

% Rotation matrix
R = [cos(theta) -sin(theta); sin(theta) cos(theta)];

% Rotate each corner around the center
rotatedCorners = zeros(size(corners));
```

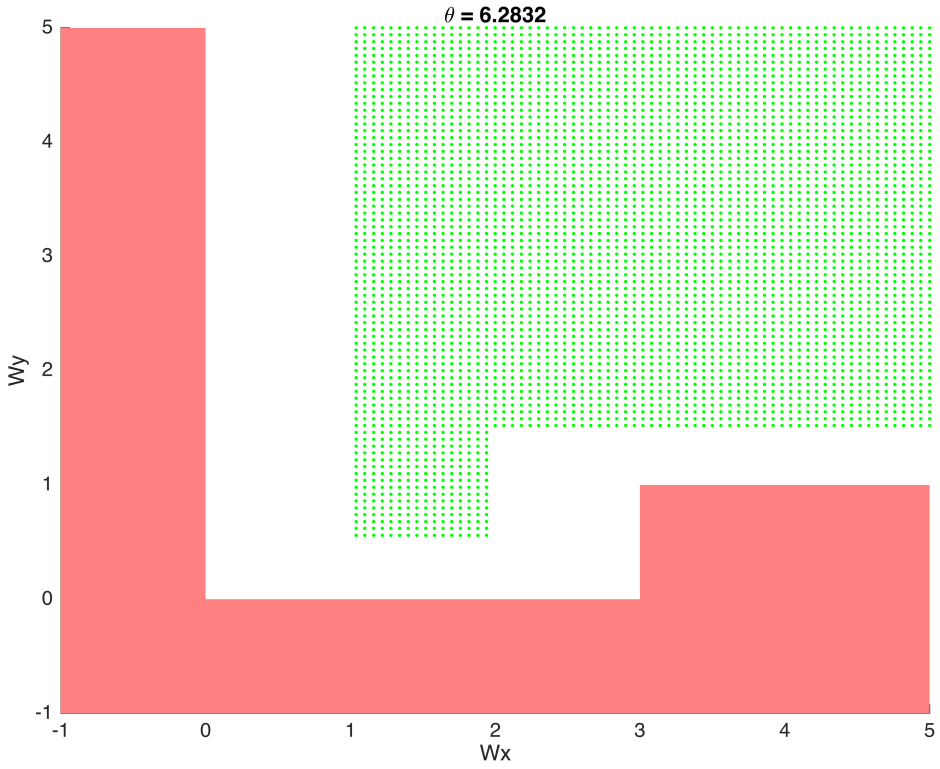
```
for i = 1:size(corners, 1)
    cornerShifted = corners(i, :) - center;
    rotatedCornerShifted = R * cornerShifted';
    rotatedCorners(i, :) = rotatedCornerShifted' + center;
end


% Plot

lightRed = [1, 0.5, 0.5];
lightBlue = [0.7, 0.7, 1];
figure()
hold on;
patch('XData', corners_ob_1(:,1), 'YData', corners_ob_1(:,2), 'FaceColor',
lightRed, 'EdgeColor', 'none');
patch('XData', corners_ob_2(:,1), 'YData', corners_ob_2(:,2), 'FaceColor',
lightRed, 'EdgeColor', 'none');
patch('XData', corners_ob_3(:,1), 'YData', corners_ob_3(:,2), 'FaceColor',
lightRed, 'EdgeColor', 'none');


patch('XData', rotatedCorners(:,1), 'YData', rotatedCorners(:,2),
'FaceColor', lightBlue, 'EdgeColor', 'b', 'LineWidth', 2);
axis equal;
grid on;
xlabel('X'), xlim([-2 5]);
ylabel('Y'), ylim([-1 5]);
```




```
% Initialize the 3D matrix for intersection data
intersectionMatrix = zeros(size(Wx,2), size(Wy,2), size(Wtheta,2));

% Loop over each position and orientation
for h = 1:size(Wx,2)
    x = Wx(h);
    for i = 1:size(Wy,2)
        y = Wy(i);
        for j = 1:size(Wtheta,2)
            theta = Wtheta(j);
            % Recalculate the center based on current x and y
            center = [x, y];

            % Recalculate corners based on current center
            corners = [center(1) - halfWidth, center(2) - halfLength;  %
Lower left
                       center(1) + halfWidth, center(2) - halfLength;  %
Lower right
                       center(1) + halfWidth, center(2) + halfLength;  %
Upper right
                       center(1) - halfWidth, center(2) + halfLength]; %
Upper left


            % Rotation matrix
            R = [cos(theta) -sin(theta); sin(theta) cos(theta)];
```

```
            % Rotate each corner around the current center for current theta
            rotatedCorners = zeros(size(corners));
            for k = 1:size(corners, 1)
                cornerShifted = corners(k, :) - center;
                rotatedCornerShifted = R * cornerShifted';
                rotatedCorners(k, :) = rotatedCornerShifted' + center;
            end
            intersection = 0;

            if checkIntersection(rotatedCorners, corners_ob_1) == 1 || ...
               checkIntersection(rotatedCorners, corners_ob_2) == 1 || ...
               checkIntersection(rotatedCorners, corners_ob_3) == 1
                intersection = 1;
            end

            % Store the intersection information
            intersectionMatrix(h, i, j) = intersection;
        end
    end
end

% Number of frames for the animation
nFrames = size(Wtheta,2);

figure;
hold on;
xlim([-1, 5]);
ylim([-1, 5]);
xlabel('Wx');
ylabel('Wy');
title('Intersection Animation');

% Plot the obstacles once
patch('XData', corners_ob_1(:,1), 'YData', corners_ob_1(:,2), 'FaceColor',
lightRed, 'EdgeColor', 'none');
patch('XData', corners_ob_2(:,1), 'YData', corners_ob_2(:,2), 'FaceColor',
lightRed, 'EdgeColor', 'none');
patch('XData', corners_ob_3(:,1), 'YData', corners_ob_3(:,2), 'FaceColor',
lightRed, 'EdgeColor', 'none');

% Create an empty scatter plot
hScatter = scatter([], [], 'g.');

% Create an animation
for frame = 1:size(Wtheta,2)
    % Find the indices where there's an intersection for the current theta
    [intersectX, intersectY] = find(intersectionMatrix(:,:,frame) == 0);

    % Convert the indices to actual Wx and Wy values
```

```
    plotX = Wx(intersectX);
    plotY = Wy(intersectY);

    % Update the scatter plot data
    set(hScatter, 'XData', plotX, 'YData', plotY);

    % Update the title to indicate the current theta
    title(['\theta = ', num2str(Wtheta(frame))]);

    % Use drawnow to update the figure
    drawnow;

    % Pause to control the speed of the animation
    pause(0.01);
end
hold off;
```




```
function intersection = checkIntersection(rectCorners, obstacleCorners)
```

Function definition is misplaced or improperly nested.

```
    % Append the first corner at the end to simplify edge computation
    rectCorners = [rectCorners; rectCorners(1,:)];
    obstacleCorners = [obstacleCorners; obstacleCorners(1,:)];

    % Check all edges of both polygons
```

```
    for poly = 1:2
        if poly == 1
            corners = rectCorners;
        else
            corners = obstacleCorners;
        end

        for i = 1:size(corners,1)-1
            % Compute the edge and its normal (perpendicular)
            edge = corners(i+1,:) - corners(i,:);
            normal = [-edge(2), edge(1)];

            % Project both shapes onto the normal
            [minA, maxA] = projectShape(rectCorners, normal);
            [minB, maxB] = projectShape(obstacleCorners, normal);

            % Check for overlap
            if maxA < minB || maxB < minA
                % No overlap, so no intersection
                intersection = 0;
                return;
            end
        end
    end

    % Overlap on all axes, polygons intersect
    intersection = 1;
end

function [minProj, maxProj] = projectShape(corners, axis)
    % Initialize min and max projections to infinity
    minProj = inf;
    maxProj = -inf;

    % Normalize the axis (just in case)
    axis = axis / norm(axis);

    % Project each corner onto the axis and update min/max projections
    for i = 1:size(corners,1)
        projection = dot(corners(i,:), axis);
        minProj = min(minProj, projection);
        maxProj = max(maxProj, projection);
    end
end

function dxdt = vanDerPolSystem(t, x)
    dxdt = [x(2); (1 - x(1)^2) * x(2) - x(1)];
end
```