ME 5250 - Tutorial 2

In this tutorial, you will learn how to use the MATLAB Robotics Systems Toolbox to solve the inverse kinematics (IK) of robotic arms, i.e., determining the joint configurations required to achieve a desired end-effector pose. There are two primary methods for solving the IK: analytical (closed form) and numerical solutions, both of which are implemented in MATLAB.

Analytical Inverse Kinematics Solver

The analytical inverse kinematics solver computes all possible joint configurations that satisfy the desired end-effector pose using closed-form solutions. It is typically used for robots with specific structures (e.g., six degrees of freedom manipulators with a spherical wrist). One of the standard methods for deriving these solutions is the Pieper's method (e.g., see Chapter 4.6 in the textbook by J. Craig), which is also the backbone of the "analyticalInverseKinematics" function in the MATLAB Robotics Systems Toolbox. To obtain the analytical inverse kinematics solution, we first define the analyticalInverseKinematics object for our robot.

```
% Load the robot model
robot = loadrobot('abbIrb120','DataFormat','row');

% Create the analytical IK solver
aik = analyticalInverseKinematics(robot);
```

Kinematic Group Type

The Kinematic group type is a property of the analyticalInverseKinematics object (e.g., aik in the example above) that classifies the structure of the kinematic group, represented as a character vector. Each character in the vector specifies the type of joint, starting from the base to the end effector:

- **P**: Prismatic joint.
- **R**: Revolute joint that is not part of a spherical wrist.
- **S**: Revolute joint forming part of a spherical wrist.

Note: A spherical wrist comprises three consecutive revolute joints (S) with orthogonal axes.

```
% Display details for the available kinematic groups
groupType = aik.KinematicGroupType;
disp(groupType);
```

The analytical IK solver works only if the kinematic group has the pattern 'XXXSSS', where:

- **XXX** can be:
  - 'RRR': Three revolute joints that are not part of a wrist.
  - 'SSS': Three revolute joints that form a wrist.

If the kinematic group contains a prismatic joint (P), it is **invalid** for the analytical IK solver in the MATLAB Robotics Systems Toolbox.

Verifying Robot Compatibility for Analytical IK

To check if a robot's kinematic group is suitable for analytical IK, use the **IsValidGroupForIK** property, which indicates whether the kinematic group has the required pattern ('XXXSSS').

```
% Check if the selected kinematic group is valid for AIK
isValid = aik.IsValidGroupForIK;
disp(isValid); % Returns true if compatible
```

Generating an IK Function

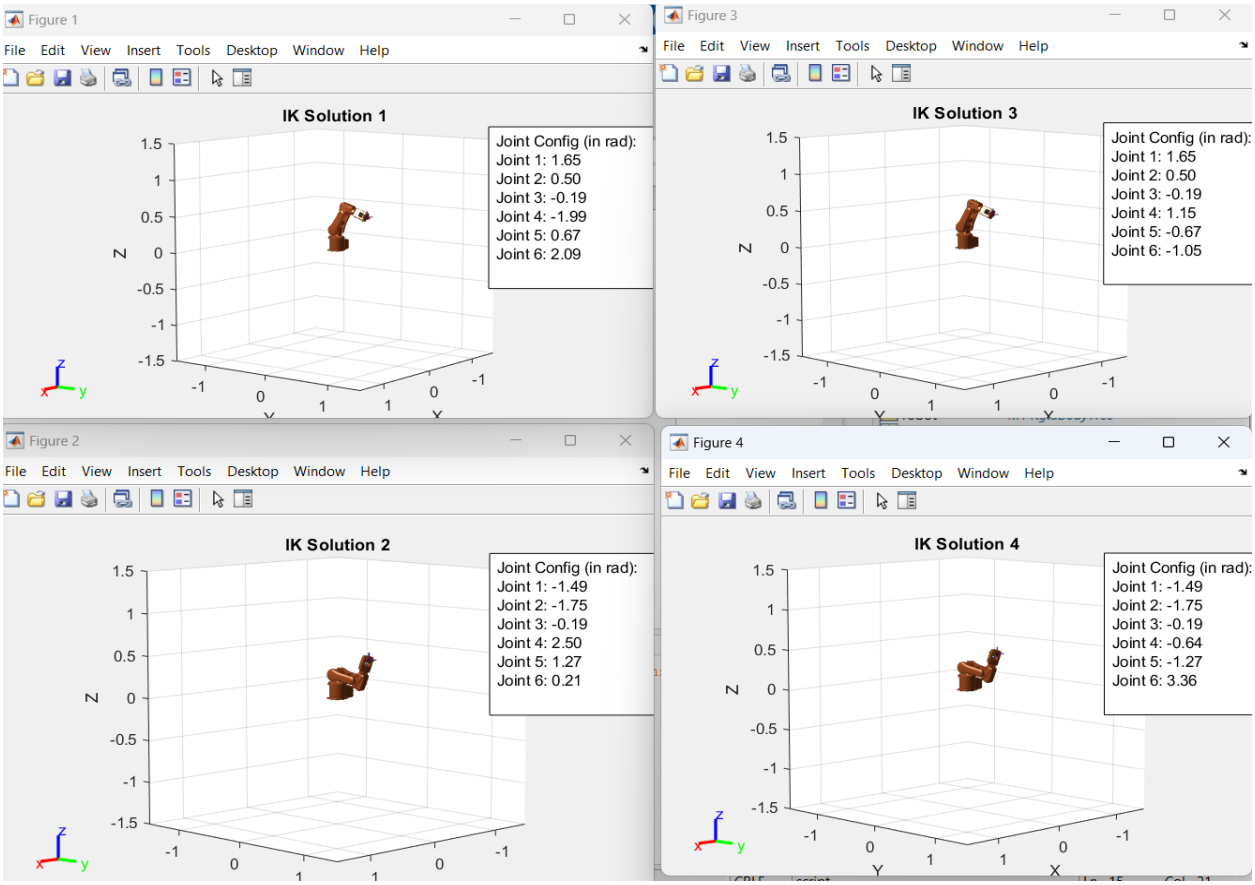
For a robot with a valid kinematic group, the closed form solution to the IK problem can be obtained by using the function called "`generateIKFunction`". For example, the following command generates an .m file named "robot_IKfunction" in your working folder, which can be used to find closed form solution for any given pose of the end effector frame for the robot whose inverse kinematics object is "aik".

```
generateIKFunction(aik, robot_IKfunction)
```

Example 1: We will obtain the closed form solution for one of the built-in robots in the MATLAB Robotics Systems Toolbox: AbbIrb120, which is a 6-DOF robot with a spherical wrist.

```
% Load the robot model
robot = loadrobot('abbIrb120', 'DataFormat', 'row');

% Create Analytical Inverse Kinematics Solver for robot
aik = analyticalInverseKinematics(robot);

% Display the kinematic group used by the AIK solver
groupType = aik.KinematicGroupType;
disp(groupType);

% Generate the IK function for the
generateIKFunction(aik, 'abbIK');
```

```
% Define the desired end-effector position
eePosition = [0, 0.5, 0.5];
theta = pi/3; % 45 degrees in radians
rotationZ = [cos(theta), -sin(theta), 0;
             sin(theta), cos(theta), 0;
             0,          0,          1];
% Combine rotation and translation into a transformation matrix
eePose = [rotationZ, eePosition'; 0 0 0 1];

% Solve inverse kinematics using the generated IK function
ikConfig = abbIK(eePose);  % This uses the generated function

% Plot each IK solution in separate figures
numSolutions = size(ikConfig, 1);
for i = 1:numSolutions
    figure; % Create a new figure for each solution
    show(robot, ikConfig(i,:));
    title(['IK Solution ', num2str(i)]);
     % Add annotation with configuration values in the top-right corner
    configText = sprintf('Joint Config (in rad):\n');
    for j = 1:length(ikConfig(i,:))
        configText = sprintf('%sJoint %d: %.2f\n', configText, j, ikConfig(i,j));
    end
    annotation('textbox', [0.75, 0.8, 0.2, 0.1], 'String', configText, ...
               'FitBoxToText', 'on', 'BackgroundColor', 'white', ...
               'EdgeColor', 'black', 'FontSize', 10);
end
```



Numerical Inverse Kinematics Solver

The numerical inverse kinematics (IK) method in MATLAB is an iterative approach used to solve inverse kinematics problems for robotic systems. This method works by iteratively minimizing the error between the current and desired end-effector poses. Starting with an initial guess, this iterative process continues until the error is minimized within a specified tolerance or the maximum number of iterations is reached. If successful, the solver returns the joint variables that achieve the desired pose; otherwise, it provides information about the failure. This method can be applied to any serial manipulator. Unlike the analytical approach that gives all possible solutions, numerical methods generate one possible solution that is usually the "closest" to the initial guess. The MATLAB Robotics Systems Toolbox has a function named "inverseKinematics", which can be used to obtain numerical solutions through two other algorithms: Broyden–Fletcher–Goldfarb–Shanno (BFGS) Gradient Projection (default selection) and the Levenberg-Marquardt (LM) algorithms.

BFGS Gradient Projection Algorithm

This is a quasi-Newton method that approximates second-derivative information using gradients from previous iterations. It is effective when configurations are near joint limits or when the initial guess is far from the solution.

Parameters: These are some of the important parameters of this algorithm.

- MaxIterations, MaxTime: Limits on iterations and runtime.
- EnforceJointLimits and AllowRandomRestarts: Ensures joint limits are respected and restarts if needed.

Levenberg-Marquardt (LM) Algorithm

This algorithm combines gradient descent and Gauss-Newton methods for faster convergence when the initial guess is close to the desired solution. It is particularly useful for solving IK along a trajectory, where the solution for one pose can be used as the initial guess for the next.

Parameters: The LM algorithm includes additional parameters beyond those used in the BFGS Gradient Projection method

- ErrorChangeTolerance: Limits changes in error per iteration.
- DampingBias and UseErrorDamping: Controls damping for stability.

For more information regarding these algorithms please refer to the link here inverse-kinematics-algorithms .

To obtain the numerical inverse kinematics solution, we first define the inverseKinematics object for our robot. When creating an inverse kinematics solver object, you can specify the algorithm using the 'SolverAlgorithm' parameter:

```
% Load the robot model
robot = loadrobot('abbIrb120', 'DataFormat', 'row');
ik = inverseKinematics('RigidBodyTree', robot,'SolverAlgorithm','LevenbergMarquardt');
```

You can also adjust other solver parameters to fine-tune the algorithm's behavior:

```
ik = inverseKinematics('RigidBodyTree', robot);
ik.SolverAlgorithm = 'LevenbergMarquardt';
ik.MaxIterations = 2000;
ik.SolutionTolerance = 1e-5;
```

To find a joint configuration that achieves a specified end-effector pose, we use the “ik” function

```
[configSol,solInfo] = ik(endeffector,pose,weights,initialguess)
```

Input parameters

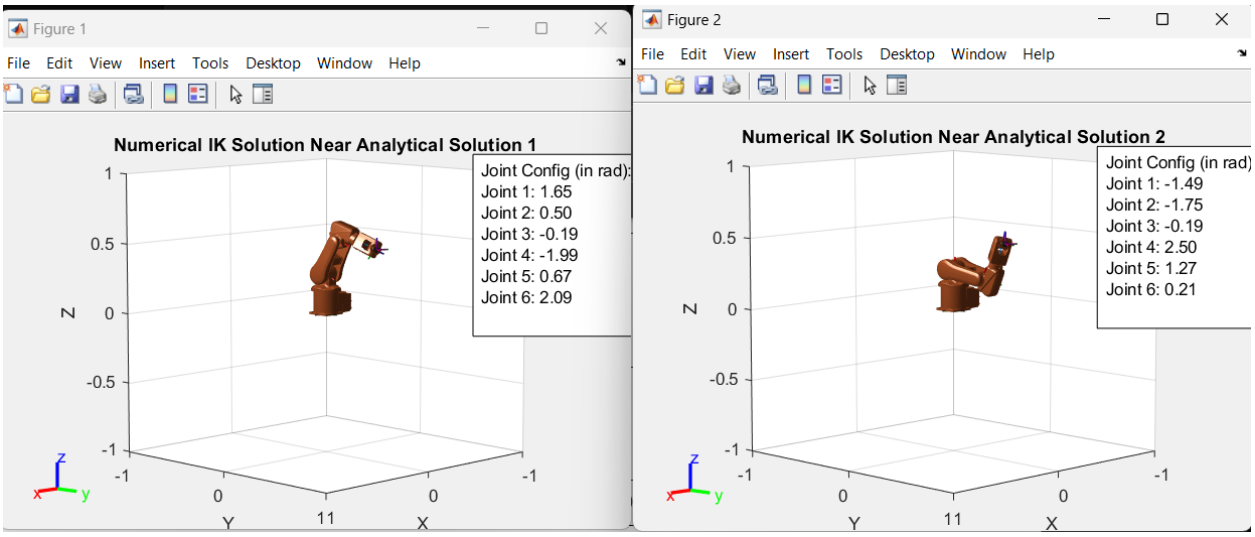
- End-effector name, specified as a character vector. The end effector must be a body on the rigidBodyTree object specified in the inverseKinematics System object.
- End-effector pose, specified as a 4-by-4 homogeneous transform. This transform defines the desired position and orientation of the rigid body specified in the endeffector property.
- Weight for pose tolerances, specified as a six-element vector. The first three elements correspond to the weights on the error in orientation for the desired pose. The last three elements correspond to the weights on the error in xyz position for the desired pose.
- Initial guess of robot configuration, specified as a structure array or vector. Use this initial guess to help guide the solver to a desired robot configuration.

Output

- configSol contains the joint angles that achieve the desired pose.
- solInfo provides details about the solver's performance.

Example 2: Here, we solve the inverse kinematics for the same robot in Example 1 using the numerical inverse kinematics solver by providing initial guesses, which are closer to different analytical solutions from Example 1.

```
% Load the robot model
robot = loadrobot('abbIrb120', 'DataFormat', 'row');

% Create the numerical IK solver using any of the two algorithms, If you don’t
specify by default it uses BFGS algorithm.
ik = inverseKinematics('RigidBodyTree',
robot,'SolverAlgorithm','LevenbergMarquardt');

% Set weights for the IK solver (position and orientation)
weights = [1, 1, 1, 1, 1, 1];

% Define the desired end-effector position and orientation
eePosition = [0, 0.5, 0.5];
theta = pi/3; % 60 degrees in radians
rotationZ = [cos(theta), -sin(theta), 0;
             sin(theta), cos(theta), 0;
             0,          0,          1];

% Combine rotation and translation into a transformation matrix
eePose = trvec2tform(eePosition) * rotm2tform(rotationZ);

% Defining initial guess near Analytical IK solutions from your previous results
initial_guess_near_analyticalSolutions = [
    1.2, 0.36, -0.1, -1.74, 0 1.8;
    -1, -1, -0.1, 2.1, 1.28, 0
];
% Solve inverse kinematics using initial guess near Analytical IK solutions from
your previous results
for i = 1:size(initial_guess_near_analyticalSolutions, 1)
    initialGuess = initial_guess_near_analyticalSolutions(i, :);

    % Solve IK
    [configSol, solutionInfo] = ik('tool0', eePose, weights, initialGuess);

    % Check if the solution is valid
    if solutionInfo.ExitFlag > 0
        % Plot the solution in a new figure
        figure;
        show(robot, configSol);
        title(['Numerical IK Solution Near Analytical Solution ', num2str(i)]);
        % Add annotation with configuration values in the top-right corner
        configText = sprintf('Joint Config (in rad):\n');
        for j = 1:length(configSol)
            configText = sprintf('%sJoint %d: %.2f\n', configText, j,
configSol(j));
        end

        annotation('textbox', [0.75, 0.8, 0.2, 0.1], 'String', configText, ...
                   'FitBoxToText', 'on', 'BackgroundColor', 'white', ...
                   'EdgeColor', 'black', 'FontSize', 10);
    else
        disp(['No valid solution found for initial configuration ', num2str(i)]);
    end
end
```



In the above example we used different initial guesses, and we ended up getting different solutions (solutions 1 and 2 in Example 1) for the same end effector pose.