RBE 3001 – Unified Robotics III

Prof. Agheli

Lab 4: Velocity Kinematics

Introduction

In Lab 1 we communicated with the robot by sending commands to the joints through a MATLAB code. In Lab 2, we used forward kinematics (FK) to calculate the robot's position in the task space given the joint variables, where the FK method in Robot class takes joint angles as an input; implemented real-time stick model visualization in MATLAB; and had the arm move to vertices of a triangle. In Lab 3, we implemented inverse kinematics and performed trajectory generation techniques in joint space and in task space to generate a smooth trajectory from one vertex to another. In Lab 4, we will implement velocity kinematics (a.k.a. differential kinematics). Velocity kinematics involves calculating the Jacobian matrix of the robot arm and using it to calculate task-space velocities for the robot, identify its singular configurations, and implement a numerical algorithm to solve the inverse kinematics problem in an iterative manner. We will then compare trajectory generation using this approach to that of polynomial trajectories.

Objectives

Upon completion of this lab, you will be able to:

1. Formulate the Jacobian matrix and calculate the forward velocity kinematics of the RBE 3001 robot arm.
2. Create MATLAB scripts and functions to implement your velocity kinematics calculations while interacting with the robot.
3. Implement a numerical approach to solving the inverse kinematics problem.
4. Perform trajectory following using speed & direction commands.
5. Visualize and characterize motion trajectories.

LAB 4: VELOCITY KINEMATICS

LAB PROCEDURES

<u>The lab is a team assignment</u>. ALL TEAM MEMBERS ARE EXPECTED TO ACTIVELY PARTICIPATE IN ALL ASPECTS TO RECEIVE CREDIT. Make sure that you keep a record of all your work, demonstrate all steps and actively get involved during the lab. Work together as a team in identifying and documenting the kinematics of the robot. Follow good team programming and work practices. Ensure all team members actively program and make commits even if collaborating at the same time.

THERE IS NO INDIVIDUAL PRELAB FOR LAB 4.

<u>Lab Procedure</u>

1. **Calculate the forward velocity kinematics of the 3-DOF robot arm.**

   Formulate the 6 by 3 manipulator Jacobian ***J(q)***. This Jacobian maps a 3 by 1 vector of the instantaneous joint space velocities to a 6 by 1 vector of instantaneous end-effector translational and angular velocities at the configuration ***q***, that is $\dot{p} = J(q)\dot{q}$. For the upper half of the Jacobian, use the partial derivative based approach introduced in class (first method). Be clear to document all steps and clearly identify the individual columns of the Jacobian and how they are calculated.

   In your lab report, include the derivation of your solution. You should include figure(s) showing any intermediate variables that you use. The naming convention must match that used in your code.

2. **Implement the forward velocity kinematics in MATLAB.**

   In this part of the lab you will implement the previously calculated forward velocity kinematics in MATLAB. As such you will write a new method in your Robot class:

   - `jacob3001()`
     - This method takes your configuration ***q*** (i.e. all of the <u>current</u> joint angles at the time the function is run) and returns the corresponding numeric 6 by 3 Jacobian matrix.
     - Using the MATLAB symbolic toolbox for your real-time Jacobian implementation will likely make your code run extremely slow! <u>Derive closed-form Jacobian and then implement.</u> You may (but are not required to) use the symbolic toolbox to generate the solution, but then take the output from that and hardcode it into the method.

   In your report be sure to show and clearly describe the calculations and describe the workflow. Explain your implementation and choices you made.

3. Validate your Jacobian calculation

Validate your Jacobian implementation by invoking the '`jacob3001(q)`' method. Pass it a vector of joint variables that would take the manipulator to an overhead configuration i.e. the arm fully extended along the $\boldsymbol{z_0}$ axis – this is a singular configuration. Calculate the determinant of J_p (i.e. only the first three rows of J corresponding to position), and display it on screen. If your Jacobian calculation is correct, the first column of the Jacobian should contain all 0 (or very small numbers close to 0) and its determinant should be 0 (or a very small number close to 0) at this configuration. In your report show these results and describe what you see. Why is the first column close to zero – what does that mean? Why is the determinant zero?

Repeat this for one other singular configuration (such as the arm fully outstretched with the last two links aligned). In your report show these results, describe why you see and what it means physically.

Hint: this procedure is not a comprehensive validation of your Jacobian, but it verifies a necessary condition for your Jacobian to be valid, therefore it can help you detect mistakes in your calculations before you proceed with the rest of this lab.

Sign-off #1: Demonstrate that you can calculate the Jacobian with test cases given by the SAs.

4. Calculate the forward velocity kinematics in MATLAB

Write a method in the robot class that calculates the forward velocity kinematics (i.e., write a method that solves $\dot{p} = J(q)\dot{q}$) that can be used in real-time while your robot is running.

- `fdk3001()`
    - This method takes your configuration $\boldsymbol{q}$ (i.e. all of the current joint angles at the time the function is run) and the vector of instantaneous joint velocities $\dot{q}$ as inputs.
    - It should return the 6x1 vector including the task-space linear velocities $\dot{p}$ and angular velocities $\dot{\omega}$.
    - Use q as input to the '`jacob3001()`' method that you developed in Part 2 to calculate the Jacobian matrix for the current configuration, $J(q)$ from within this method.

Hint: Make sure you are not reading data from the robot too quickly, or your velocities may be zero.

5. Live plot of the task-space velocity vector

Select three arbitrary triangle vertices in the task space that require motion of all motors between each motion from one vertex to the next (it is recommended to use the same set of points from Lab 3). Generate a quintic trajectory along each direction in task space between each pair of vertices (1->2, 2->3, 3->1). In this case, impose the zero velocity and acceleration constraints at the start and the end of the trajectory. You may essentially just reuse Part7 of Lab 3 for this.

Have the robot execute the trajectory and plot the stick model of the arm in real-time as the robot moves along this trajectory drawing out a triangle. Continuously read joint angles and joint velocities as it moves and use these data to continuously calculate the resulting end effector velocity vector $\dot{p}$ for each time step where data is received using the '`fdk3001()`' method.

In the stick model, display the velocity vector using an arrow having its base coincident with the position of the end effector, and pointing towards the velocity direction with a length proportional to the speed in that direction. For this part, show linear velocity only, that is, the upper half of $\dot{p}$.

Hint: for this task, you will find using the 'quiver3' command in MATLAB helpful in plotting velocity vectors.

Sign-off #2: Demonstrate real-time plotting of your stick model with the velocity vector as the arm traverses the triangular trajectory.

For the report, be sure to describe the process and what you see. Include appropriate figures of the arm moving (at least 1 approximately in the middle of each segment of motion).
Create a Figure with 3 subplots showing:

- Make a subplot with 3 lines that show the linear velocities along the x ,y, and z directions vs time throughout the complete trajectory, being sure to have unique lines and a legend.
- Make a subplot with 3 lines that show the angular velocities about the x ,y, and z directions vs time throughout the complete trajectory, being sure to have unique lines and a legend.
- Make a subplot with magnitude (i.e. scalar speed) of the linear velocity vs time throughout the complete trajectory.

6. Discover and avoid singularities

In this part, you implement an emergency stop that prevents the robot from reaching a singular configuration. Write a program that deliberately attempts to send the robot into a kinematic singularity while it continuously calculates/monitors the determinant of the Jacobian as the configuration changes. You should use the '`jacob3001()`' developed in Part 2 to calculate Jacobian and its determinant in real-time. Implement a safety check that looks how close you are to a singularity and stops sending position commands if the robot gets too close (i.e. the determinant of J_p becomes too close to zero). Recall that J_p is the first three rows of J, so you will be taking the determinant of the 3x3 top half of the Jacobian. Display an error message on the live plot and stop the motion when this occurs.

Sign-off #3: Demonstrate real-time calculation of the Jacobian and monitoring of the determinant as the robot moves through the triangle used before. Show the emergency stop functionality described above as the stick model approaches a singularity.

For the report, describe the workflow and your implementation. Why is this important and what can it prevent from happening?
Create a Figure with 2 subplots showing:

- Make a 3D plot that traces the motion of the robot's tip (you do not need to show the whole arm, but you may) as it moves from near the center of the workspace to the selected singular configuration.

- Make a 2D plot of the determinant of J_p vs time for the same motion.

Extra Credit #1: Velocity-based motion planning in task space

Implement essentially the same task performed in Part 6 of Lab 3, but instead of creating a polynomial trajectory, we will command the robot to move in the direction of the target point as a set speed.

Write a program that takes in the same triangle vertices used in Lab 3 and attempts to follow the same general motion. For each motion path (i.e. 1->2, 2->3, and 3->1) you should set the appropriate vertex to be the target position in the task space and move at constant speed towards that target. Then, in a continuous loop running as fast as you reasonably can:

- Determine a unit vector pointing from the current measured robot tip point (will need to apply FK to the measured joints) to the current target.
- Create a desired 3x1 task space instantaneous velocity vector by multiplying an arbitrary speed (you pick a reasonable one) by the unit vector you calculated
- Use the inverse of the top 3x3 portion of the Jacobian for the current configuration to calculate the inverse velocity kinematics (i.e. the desired instantaneous joint velocities).
- Command the robot to move at this desired velocity. *NOTE: You cannot control the velocity of the robot directly, only the position. However, if you know how fast your loop runs, you know how much to move the end effector each iteration to achieve this speed..*

Sign-off EC1 (+5.00): Demonstrate inverse velocity kinematics based control of the manipulator along this triangle path.

For the report (+5.00): again create a figure with the same 3 subplots from Lab 3 showing the task space trajectory and include in your report:

- Make a subplot with 3 lines that shows the x, y and z positions (in mm) vs time (in seconds), being sure to have unique lines and a legend.

- Make a subplot with 3 lines that shows the x, y and z velocities (in mm/s) vs time (in seconds).

- Make a subplot with 3 lines that shows the x, y, and z accelerations (in mm/s^2) vs time (in seconds).

Compare the results of this approach to that in Lab 3.

Extra Credit #2: Numerical Inverse Kinematics

Did you know that in addition to the Geometric and Algebraic IK approaches, there is a third method of calculating Inverse Kinematics using the velocity kinematics? If you are interested in learning it, do this extra credit to see how it works!

In this part, you will implement a solution to the inverse kinematics problem based on velocity kinematics. This allows an iterative approach to a numeric solution without explicitly needing to determine a closed-form inverse kinematics solution.

Write a MATLAB method '`ik_3001_numerical()`' in robot class that implements the numerical inverse kinematics algorithm. That is, you should determine where the robot is (can be an arbitrary start point), determine the desired target (that is the input to the IK), and gradually move the stick model of the arm (it can be a virtual representation for this portion, the real robot will not move) in the direction pointing from the current position of the end-effector to the final position of the end-effector.

This approach is very similar to what you implemented in EC1. However, in EC1, the robot moved each time step. You will use a similar loop in this section, but you will not be moving the real robot. Instead, each iteration of the loop, you will update a variable keeping track of what the new joint angles would be after applying the jacobian over a small timestep.

Define the magnitude of the end-effector velocity for each motion step and use the inverse of Jacobian to convert the end-effector velocity into joint velocities (like in EC1). Define a time interval for arm motion in each iteration so that the increment in joint angles for each iteration can be calculated by multiplying joint velocities with the defined time interval. Repeat this operation until the end effector position (calculated by passing your joint angle variable into fk3001()) is within some tolerance of the desired target. Once the robot has reached the target point, read out joint angles – these correspond to the IK result without ever explicitly determining a closed-form solution for the IK.

To validate your algorithm, do the following – note that the following steps are done in simulation, and there is no need to use the physical robot arm.

a) Plot a stick model of the arm at an arbitrary configuration q. Impose $y = 0$, so that the manipulator lies entirely on the x-z plane. Set the point of view of your plot so that you look at the manipulator "from the side."
b) Use the MATLAB function 'ginput' to sample an arbitrary point in the x-z plane of the robot. Make sure the point you select is within the arm's workspace. Pass the point to the algorithm and iterate until the algorithm converges (the difference between end effector position and target position is within the range of acceptance). Update the plot at each iteration to show the robot converging to the target position.

Sign-off EC2 (+5.00): Demonstrate iterative inverse kinematics with an arbitrary target end-effector position.

For the report (+5.00): validate your algorithm with at least 3 different target points and include it in your report. Include plots of the arm converging to the target position of each of the 3 targets *(Hint: you can show the stick model with "hold on" so you can see the sequence of motions to target).* Plug the joint values at the end of execution of your algorithm into your forward kinematics method. Compare the results of forward kinematics and designed target positions. Also, compare the results to your closed form IK solution from Lab 3. Be sure to include data in your lab report, along with some comments about the performance of the iterative IK algorithm (how much does the ending position of numerical IK differ from the designed position).

Extra Credit #3

Power up the robot arm and use the iterative IK algorithm which has been validated in simulation to make the arm move to a predefined target position. Configure the robot at an arbitrary pose in x-z plane, imposing $y = 0$. Display the real-time stick model of the arm. Use the 'ginput' function to select a new point in the x-z plane, then use the iterative inverse kinematics algorithm to solve for the new joint variables and send those variables as position commands to the robot. *Hint: you don't have to send new joint angles at each iteration to the arm, wait for the algorithm to converge and then send the joint angles in the last iteration, i.e. the robot only moves one time.*

Watch the robot reach for the new configuration and simultaneously update the real-time stick model. Now pick a new point in the x-z plane. Repeat the process for at least 3 different points.

Sign-off EC3 (+10.00): Demonstrate the iterative IK solver working and the real robot moving to the determined targets.

Extra Credit #4 (If not done already – you can not receive credit twice)

Create a live 3D plot of the arm using CAD from the real robot as described in the previous labs.

Sign-off EC4 (+10.00): Demonstrate the live animated 3D CAD model of the arm moving alongside the real robot to an SA/TA and also include screenshots (and maybe video link) in the report.

Notes:

- Be sure to understand and clearly document the CODE you write. Ensure all of the code is well documented. Tag a specific commit in your repository with your final version of the code.
- Clearly describe all the steps you took, what decisions you made and why, and your overall system architecture. Be sure to describe your high-level communication. Make figures, flowcharts, etc.
- Describe all results obtained and show the plots acquired, especially as where specifically noted in the assignment. Feel free to further elaborate or add additional plots and results if it helps with your explanation.
- In the report, clearly show the forward kinematics of your robot arm. Include a clear sketch showing your joint parameters, frames, dimensions, etc. You should define a frame on each link, and solve

for all intermediate transformations (as homogeneous transformation matrices), and then multiply together to get the forward kinematics as a homogeneous transformation matrix.

Please note, completing the task in the assignment sheet and putting in a corresponding plot does NOT satisfy the requirement. You must describe what is going on, why it's going on, and show an understanding and appreciation for what you see. The labs are NOT just making the robot move, understanding, clearly presenting, and thoroughly describing what happens is at least as important.

LAB 4: VELOCITY KINEMATICS

LAB 4 SIGNOFF TABLE (50 POINTS TOTAL, + UP TO 30 EXTRA CREDIT)

TASK	***DETAILS***	***INSTRUCTION STEP***	***POINTS***
Validate Jacobian Calculation	Show that your function calculates the numeric Jacobian for the given configurations requested.	3	15
Real-time velocity vector plotting	Show the live stick model as the robot follows the trajectory with velocity vector overlay.	5	20
Singularity detection	Monitor the determinant of the Jacobian in real-time and alert when close to singularity.	6	15
Inverse velocity kinematics control	Show the motion profile along the triangle using velocity based control	EC1	+5
Iterative IK solver	Demonstrate iterative solver for the inverse kinematics using inverse velocity kinematics.	EC2	+5
Demonstrate IK solver on robot	Solve IK using the iterative solver for arbitrary targets and move to them.	EC3	+10
Animated 3D CAD Model	Create a live 3D model of the arm based on CAD data	EC4	+10 (if not previously awarded)

Lab 4: Velocity Kinematics

Team Number:

Lab Report and GitHub

Grading Rubric: Comments Score

(50 points) Total Points from Sign-Offs		**/50**
(30 points) Submitted lab report		
Introduction: Effectively presents a brief description of the main objectives of the lab.		**/3**
Methodology: Gives enough details to allow for replication of procedure. Put lab procedure into own words.		**/2**
Results: Presents visuals clearly and accurately, presents findings clearly and with brief and sufficient support of data collected.		**/10**
Discussion: Comprehensive explanation of all results collected. All results should be explained in detail describing the cause and the result of the data collected**.** All explanations should provide sufficient and logical explanation for the statement, addresses other issues pertinent to lab.		**/12**
Conclusion: Concise explanation of methods explored and learned during the lab		**/3**
(20 points) GitHub hosted code		
Readability: Each line of the code should be commented with a sane comment or fully self-documenting. Use reasonable variable names and provide documentation for your functions.		**/4**
Formatting: Code should be properly indented (i.e. it should *not* be all in a single column). Code should obey a self-consistent naming convention.		**/4**
Issues: Create Issues to manage work. Lab procedures should be broken into Github Issues and delegated by the team to the team during lab sessions. Each commit should be associated with an issue, issues should be assigned, and tasks shared to complete labs as a team. Also, you must make use of branches using gitflow		**/4**
Commits: GitHub repositories should show consistent, frequent commits demonstrating code efforts for a given lab. Each commit must have: Short description of what changed, two new lines, then a sentence describing why it was changed and the Issue ID		**/4**
Code Reuse: Code in future labs should make use of code created in previous labs. Reuse should not take the form of copy/paste to redefine functions. Utilize object-oriented coding.		**/4**
(+10 points) Include EC1 and EC2 in the report	**Team Score: Maximum score: 100 points**	