MEAM 5200 Lab 2 Report

John D'Ambrosio, Cedric Hollande

March 24, 2024

1 Methods

I employed a simple strategy to effectively calculate the velocity kinematics, using the findings from the forward kinematics:

1. I created a helper function `get_axis_of_rotation` in my `FK` class, which created the subsequent homogeneous transformation matrices, T_n^0, for each of the 7 joints, and extracted the axis of rotation of each joint, z_n^0, at each step.

2. Then, I extracted the joint position variables, which were multiplied by their given offsets, p_n^0, for each subsequent transformation by passing the `jointPosition` matrix from the `FK.forward` function. I also extracted the end-effector position, p_e^0.

3. Next, I calculated the linear component of the Jacobian, J_v, by taking the cross product of z_n^0 and $p_e^0 - p_n^0$ for each of the n joints, and adding that to the $J_{1:3,n_{th}}$ column of the Jacobian matrix.

4. For the angular component, J_w, I simply set $J_{3:6,n_{\text{th}}}$ to z_n^0 for each of the 7 joints, indicating J is a matrix in $\mathbb{R}^{6\times 7}$.

5. I could then use my Jacobian to easily calculate the forward velocity kinematics by doing $\xi = J(q)\dot{q}$ and returning the ξ vector.

6. For my inverse velocity kinematics, where $J \in \mathbb{R}^{6\times 7}$, I have to use the pseudoinverse, $\dot{q} = J^{\dagger}\xi$. Thus, I used least-squares, through numpy's `np.linalg.lstsq`, to find an approximate best solution that

$$\begin{aligned} &\min_{\dot{q}} \|\dot{q}\|^2 \\ &\text{s.t.} \quad J(q)\dot{q} = \xi \end{aligned}$$

1.1 Code Explanation

We used John D'Ambrosio's code for testing. The code base consists of 5 scripts that work in tandem with each other, with the `calculateFK.py` and `calcJacobian.py` being the two most foundational scripts. The `calculateFK.py` script uses the DH parameters found in Lab 1 to compute the homogeneous transformation matrices and joint positions in its `forward` function and returns the axis of rotation for each joint in the `get_axis_of_rotation` function, as mentioned in the previous section. These calculations are then directly used in the `calcJacobian.py` script, which calculates the Jacobian using the methods described in the previous section. The Jacobian script is then included in the `FK_velocity.py` to perform the forward velocity kinematics. The Jacobian script is also included in the `IK_velocity.py` script to calculate the inverse velocity kinematics using the least-squares method as described earlier. The `calcAngleDiff.py` script contains a helper function that computes the axis of rotation from the current end-effector orientation to the desired end-effector orientation, which is done by calculating the skew-symmetric part of the relative rotation and extracting the omegas. This function is directly used in the `follow.py` script. In the `follow.py` script, I modified the angular velocity variables used for tracking and then created equations to model an ellipse trajectory, projected in the x-y plane of the base frame, and a line trajectory along the z-axis of the base frame.

2 Evaluation

2.1 Testing Process and Confidence in Solutions

Our evaluation commenced with simulated tests designed to match expected outcomes formulated from manual computations. We scrutinized these against terminal outputs, confirming consistency. Thereafter, tools `visualize.py` and `follow.py` allowed for further assessment of the code's alignment with theoretical models and visualization standards.

2.2 Zero Configuration Test

The robotic arm was initialized in a zero configuration, to start off with easier calculations and more intuitive results for `FK_velocity`'s response to isolated joint actuations.

```
meam520-vm:~/meam520_ws/src/meam520_labs/lib$ python FK_velocity.py

dq=0 for all joints
[0. 0. 0. 0. 0. 0.]

dq=pi/2 only for joint:  1
[0.         0.13823008 0.         0.         0.         1.57079633]

dq=pi/2 only for joint:  2
[ 0.7696902   0.         -0.13823008  0.          1.57079633  0.        ]

dq=pi/2 only for joint:  3
[0.         0.13823008 0.         0.         0.         1.57079633]

dq=pi/2 only for joint:  4
[-0.27331856  0.          0.00863938  0.         -1.57079633  0.        ]

dq=pi/2 only for joint:  5
[0.         0.13823008 0.         0.         0.         1.57079633]

dq=pi/2 only for joint:  6
[ 0.32986723  0.          0.13823008  0.         -1.57079633  0.        ]

dq=pi/2 only for joint:  7
[ 0.          0.          0.          0.         -0.         -1.57079633]
```

Figure 1: *Terminal outputs displaying the end-effector velocity vector when individual joints are actuated with velocity* $\frac{\pi}{2}$ *in the zero configuration.*

Expectation and Results

- Theoretically, in a zero configuration, actuating a single joint should influence the end-effector velocity vector in a predictable pattern, aligned with the joint's type (rotational or translational) and axis.
- Initial static tests yielded a zero vector output for the end-effector velocity with all joint velocities at rest, affirming the static equilibrium hypothesis.

- Sequential actuations, with individual joints set to a velocity of $\frac{\pi}{2}$ radians per second, produced distinct non-zero end-effector velocities. These velocities conformed to the expected outcomes calculated by hand, thus reinforcing the accuracy of the `FK_velocity` function.

2.3 Geometric Intuition and Singularities

Using geometric intuition, there are few ways singularities can occur in the 7-DOF Panda arm. Firstly, a wrist Singularity can occurs when the axes of last three joints of the robot align. In this configuration, the robot loses one degree of freedom, meaning it can no longer move the end-effector in all directions, which is a common singularity for robotic arms with a spherical wrist. Another singularity can occur when the arm is stretched out straight (either fully extended or fully retracted), the shoulder joint's movement becomes less effective in moving the end-effector in some directions. This is similar to what humans experience when trying to lift something with an outstretched arm. Additionally, an elbow singularity can occur when the elbow is fully extended or retracted, limiting the arm's ability to move the end-effector inwards or outwards. A final singularity can occur when the first wrist joint directly aligns with the base joints of the Panda arm, which we actually observed in our ellipse testing. Mathematically speaking, these configurations are reflected in our calculations. When a manipulator arm is at a singularity, where the end-effector's degrees of freedom are compromised, we see that the Jacobian looses rank.

2.4 End Effector Trajectory Tracking

The end effector's precision in trajectory tracking was thoroughly assessed, with particular attention given to its ability to follow basic geometrical paths, such as `follow.py line`.

However, the incorporation of least-squares optimization introduces an inherent complexity to the IK solutions. In scenarios lacking a direct solution or presenting infinite solutions, the optimization seeks to approximate a solution that best minimizes the l^2 norm of the error for the joint velocities. Although this method finds the best possible approximation, it is not without its drawbacks. Small errors inherent to any optimization-based solution can accumulate over time, resulting in a divergence from the intended trajectory. Particularly, when running the `follow.py` script, these errors can compound, especially when running the ellipse and figure-eight trajectories, leading to observable deviations that grow with each iteration of the control loop. This effect becomes more pronounced when controlling orientation, where small errors in angular velocity can significantly impact the end-effector's orientation over time.

If we were to also control orientation, we would have create a secondary task vector, using some mathematical formula, and then project that onto the null-space of the Jacobian, so that the secondary task does not influence our end-effector position, which is the desired goals. Thanks to the nature of optimization problems, we can easily add that secondary task to our main optimization task,

so that in future work, we can ensure that we meet other tasks, like orienatation control.

2.5 Conclusion

Our extensive suite of tests, both in simulated environments and using visualization tools, verified the robustness of the theoretical model. The `FK_velocity` function consistently delivered precise predictions of the end-effector velocities. The testing also revealed that while the system's trajectory tracking capabilities are underpinned by least-squares optimization strategies, the potential for error accumulation poses a challenge that must be addressed to maintain accuracy over time. Future enhancements will focus on optimizing our IK algorithms by adding secondary tasks to our objective function to reduce the long-term drift associated with error propagation in trajectory tracking.

3 Analysis

In the context of our observations and data, our forward and inverse velocity kinematics performed very well and yielded strong results. The data and observations reveal coherent performance that aligns with the expected characteristics of the Panda's kinematic design

3.1 Forward Velocity Kinematics

The forward velocity kinematics of the Panda arm exhibited highly predictable and accurate behavior, particularly in simulation. The direct mapping from joint velocities to end-effector velocities yielded consistent and repeatable outcomes. Movements that required linear paths or were along the principal axes of the robot's workspace were executed with remarkable precision, capitalizing on the manipulator's strength in straightforward task execution.

3.2 Inverse Velocity Kinematics

Conversely, the inverse velocity kinematics—responsible for calculating the necessary joint velocities to achieve a desired end-effector velocity—introduced complexities. While the arm adeptly handled typical trajectories, slight deviations became apparent in more intricate paths, due to the accumulation of least-squares error, especially when simultaneous position and orientation precision was required. These deviations, although minor, could be attributed to the inherent challenges of solving the inverse problem where infinite or no direct solutions exist.

3.3 Movement Strengths and Weaknesses

Experimentally, the Panda arm excelled in simple movements, such as following the line trajectory. For more complex and nonlinear trajectories, the arm initially tracked the trajectories well, but as more least-squares error accumulated, the tracking accuracy decreased each iteration, until it eventually reached a singularity or near self-collision. We had to stop the robot

3.4 Velocity Control Versus Position Control

The choice between velocity control and position control largely depends on the task's nature and the desired outcome. Velocity control offers a smoother operation for tasks that require continuous motion, in applications such as painting or welding, where the end-effector must move at a constant rate. On the other hand, position control is favorable in applications requiring high precision at specific points, such as assembly or part placement, where the exact end-effector location is critical.

3.5 Conclusion of Analysis

Overall, the results from our kinematic analysis are logical and demonstrate the Panda arm's adeptness in a variety of tasks, validating the kinematic models used. The arm's strength lies in executing fluid, natural motions, where velocity control enables graceful movement. Its shortcomings become noticeable in highly constrained spaces or tasks that demand extreme precision, where position control might offer better results. The lab's findings provide valuable insights into the arm's operational envelope, informing better decision-making for task planning and control strategy selection.