CALIFORNIA POLYTECHNIC STATE UNIVERSITY
CENG - DEPARTMENT OF ELECTRICAL ENGINEERING

EE 471- Vision Based Robotic manipulation

Lab #5 Velocity Kinematics of Robot Manipulators

Presented to:
Siavash Farzan

Presented By:

Rodrigo Menchaca
Giovanni Dal Lago

OCT 28, 2025

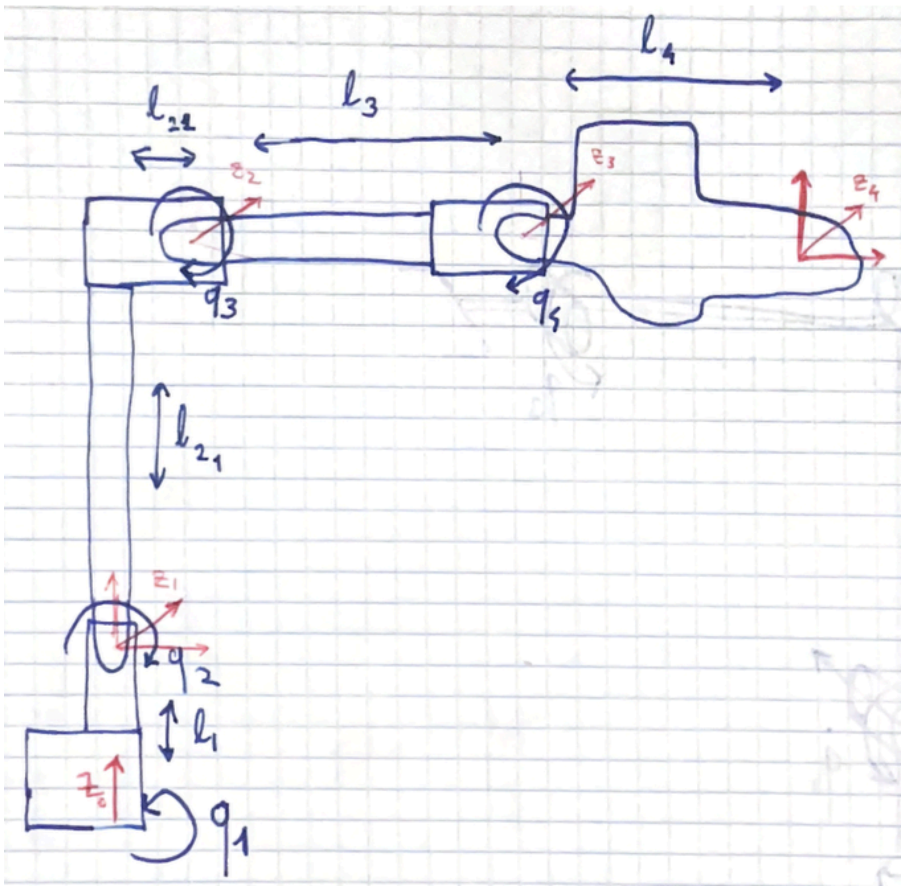
INTRODUCTION

This lab investigates velocity kinematics for the OpenManipulator-X and applies it to real-time, task-space motion control. We implement the 6×4 geometric Jacobian J(q) to map joint rates to end-effector spatial velocity p˙=[v⊤, ω⊤]⊤, validate forward velocity kinematics p˙=J(q)q˙, and use inverse velocity kinematics via the Jacobian pseudoinverse q˙=J†(q)p˙des to track a piecewise-linear path at constant Cartesian speed. The target motion is a triangular trajectory defined by three waypoints plus a return, executed with velocity commands under safety limits on task-space and joint speeds. Data are collected for joint states, end-effector pose, and velocities for quantitative evaluation.

We compare velocity-based control with the polynomial, position-based trajectories from the previous lab in terms of path straightness, smoothness of joint profiles, tracking accuracy, and real-time adaptability. Analysis includes 3D path visualization, joint velocity histories, pose and velocity time series, and waypoint convergence metrics. The results highlight trade-offs between preplanned trajectories and reactive velocity control on this platform, considering hardware noise, loop timing, and singularity sensitivity.

Part 1 : Manipulator Jacobian Derivation for OpenManipulator-X

1) ***kinematic stetch***



2) Obtaining z0i and o0i:
For each joint i, the transformation matrix 0Ti describes the orientation and position of frame i relative to the base frame. The unit vector z0i corresponds to the joint axis direction, which is given by the third column of the rotation submatrix R0i (the first three elements of the third column of 0Ti). The position vector o0i represents the origin of frame i relative to the base, which is given by the first three elements of the fourth column of 0Ti. Thus,

z0i = R0i[:,3],   o0i = 0Ti[0:3,3]

3) Expressions for the Jacobian Columns

For each revolute joint i = 1…4:

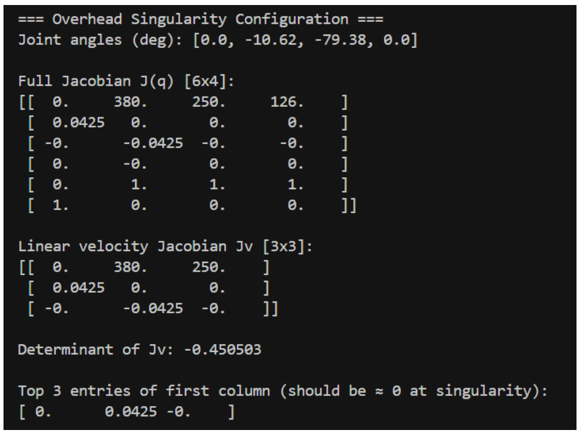
$$J_i = \begin{bmatrix} z_{i-1}^0 \times (o_4^0 - o_{i-1}^0) \\ z_{i-1}^0 \end{bmatrix}$$

Then stack them:

$$J(q) = \begin{bmatrix} z_0^0 \times (o_4^0 - o_0^0) & z_1^0 \times (o_4^0 - o_1^0) & z_2^0 \times (o_4^0 - o_2^0) & z_3^0 \times (o_4^0 - o_3^0) \\ z_0^0 & z_1^0 & z_2^0 & z_3^0 \end{bmatrix}$$

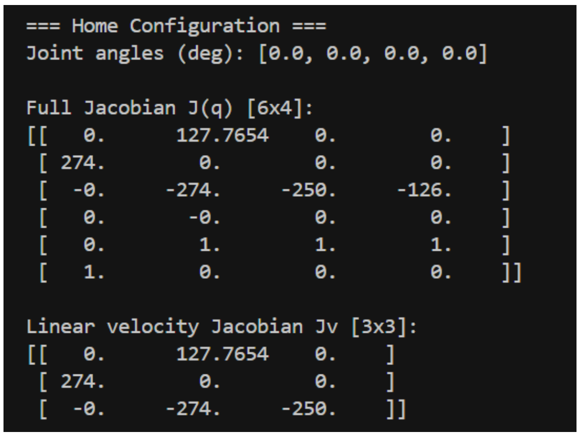
Part 2: Implement the Manipulator Jacobian Computation in Python

Once we derived the Jacobian for the Open-ManipulatorX robot arm. We implemented these same derivations in python for faster calculation. We created the function get_jacobian() which receives the joint configuration vector that represents the 4 joint angles in **degrees** [q1,q2,q3,q4]. It then calculates the Manipulator Jacobian matrix **J(q).** In order to implement this function we first create our four homogenous transform matrices by implementing a quick for loop. After the homogenous transforms are built we quickly created an empty 6x6 matrix to use as our Jacobian further along the code implementation. We then proceed with extracting the end effector origin O4, Joint origins(O1,O2,O3,O4) and the Joint Z_axes(revolute). After all of these parameters are extracted from our input, we begin with building up our Jacobian matrix J by performing the necessary matrix multiplication and then stacking the arrays to format our Jacobian into the correct form. During our Prelab we performed a sanity test to verify correct implementation of our Jacobian calculation. Figures 2.a and 2.b shows our results from the Prelab which align with the expected values given to us, this further verifies our correct implementation of the get_jacobian() function.



```
=== Overhead Singularity Configuration ===
Joint angles (deg): [0.0, -10.62, -79.38, 0.0]

Full Jacobian J(q) [6x4]:
[[  0.      380.      250.      126.    ]
 [  0.0425   0.        0.        0.    ]
 [ -0.      -0.0425  -0.       -0.    ]
 [  0.      -0.        0.        0.    ]
 [  0.       1.        1.        1.    ]
 [  1.       0.        0.        0.    ]]

Linear velocity Jacobian Jv [3x3]:
[[  0.      380.      250.    ]
 [  0.0425   0.        0.    ]
 [ -0.      -0.0425  -0.    ]]

Determinant of Jv: -0.450503

Top 3 entries of first column (should be ≈ 0 at singularity):
[ 0.      0.0425 -0.    ]
```

Figure 2.a: Jacobian Overhead Configuration



```
=== Home Configuration ===
Joint angles (deg): [0.0, 0.0, 0.0, 0.0]

Full Jacobian J(q) [6x4]:
[[   0.      127.7654     0.        0.    ]
 [ 274.        0.         0.        0.    ]
 [  -0.     -274.      -250.     -126.    ]
 [   0.       -0.         0.        0.    ]
 [   0.        1.         1.        1.    ]
 [   1.        0.         0.        0.    ]]

Linear velocity Jacobian Jv [3x3]:
[[   0.      127.7654     0.    ]
 [ 274.        0.         0.    ]
 [  -0.     -274.      -250.    ]]
```

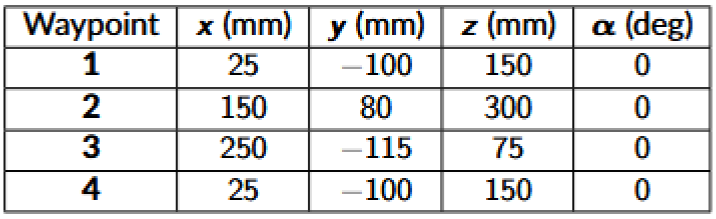
Figure 2.b: Jacobian Home configuration

Part 3. Implement Forward Velocity Kinematics in Python.

After our Jacobian implementation was built and verified of correct functionality. We began building the Forward Velocity Kinematics method in python. Named get_fwd_vel_kin(), this function receives two inputs: our joint angle vector(degrees) and our joint velocity vector(rad/s). it then performs our velocity kinematics method to output a 6x1 velocity vector $[v^{\top}; !^{\top}]^{\top}$. In order to implement Velocity vector **pdot,** we first called our get_jacobian() function to obtainin our **J(q),** we then perform matrix multiplication **pdot = J(q)*qdot** to output our velocity vector **Pdot.** Since we some times might need to use our joint velocities in radians, we decided to implement our degrees to radians part outside of the get_fwd_vel_kin() in order to be able to use our desired unit, depending on the situation.

Part 4. Velocity-based motion planning in task space.

After our Velocity and Jacobian functions were implemented and tested for full functionality. We began implementing these methods to develop Velocity-based motion planning for our Open-ManipulatorX arm. We created the same trajectory we have been using for past labs shown in Figure 3.



Waypoint	x (mm)	y (mm)	z (mm)	α (deg)
1	25	−100	150	0
2	150	80	300	0
3	250	−115	75	0
4	25	−100	150	0

Figure 3: Task-space waypoints for velocity based trajectory tracking

In order to develop our algorithm we develop our control loop by getting the current joint angles using our developed functions to then calculate the current End-Effector pose with the use of our get_ee_pos(). Once we have our current pose for the End-Effector, we would use those values to calculate our direction vector U=(Ptarget-Pcurrent)/||(Ptarget-Pcurrent)||. Once the direction vector is calculated we can use this vector to generate our desired velocity needed in order to achieve our desired position, **Vdes = Vdes*U.** After all of this was calculated we computed the required joint velocities by using the Jacobian pseudo-inverse. After we have the joint velocities, we convert them from rad/s to deg/s and then use the robot.write_velocities() function to start our trajectory. As we are moving through our trajectory, we continuously recorded different values in order to develop the required plots. When implementing our control loop, we achieved a frequency of 20Hz when we had a commanded velocity of 50mm/s. In order to ensure we didn't damage the robot by excessive speeds we limited the robot velocity to 45deg/sec.

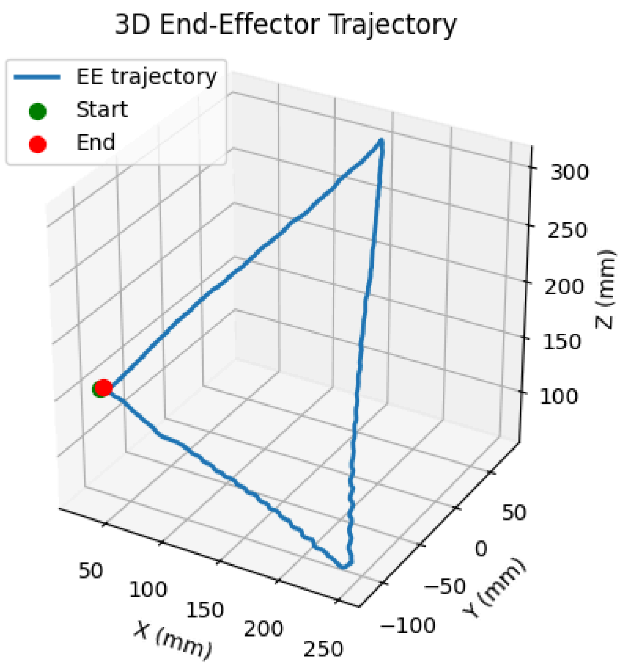


Figure 4: 3D trajectory plot

Figure 4 shows the 3D trajectory of the OpenManipulator-X end-effector as it follows the defined triangular path in task space. The blue curve represents the actual motion of the end-effector, while the green and red markers indicate the start and end positions respectively. The trajectory demonstrates consistent tracking of the commanded waypoints, with smooth transitions along each edge and minor deviations near the corners due to abrupt direction changes. Overall, the motion closely matches the desired task-space path, confirming proper operation of the implemented velocity-based control algorithm. In comparison with the triangular path during lab 4, this trajectory path was somewhat wavy along the corners of the triangle, this is mainly because of the fact that we are controlling the robot by velocity and not position trajectory planning. Since we had to slow down the robot as it was approaching the change in trajectory, this cause that wavy trajcectory unlike our previously generated trajectory during lab 4.

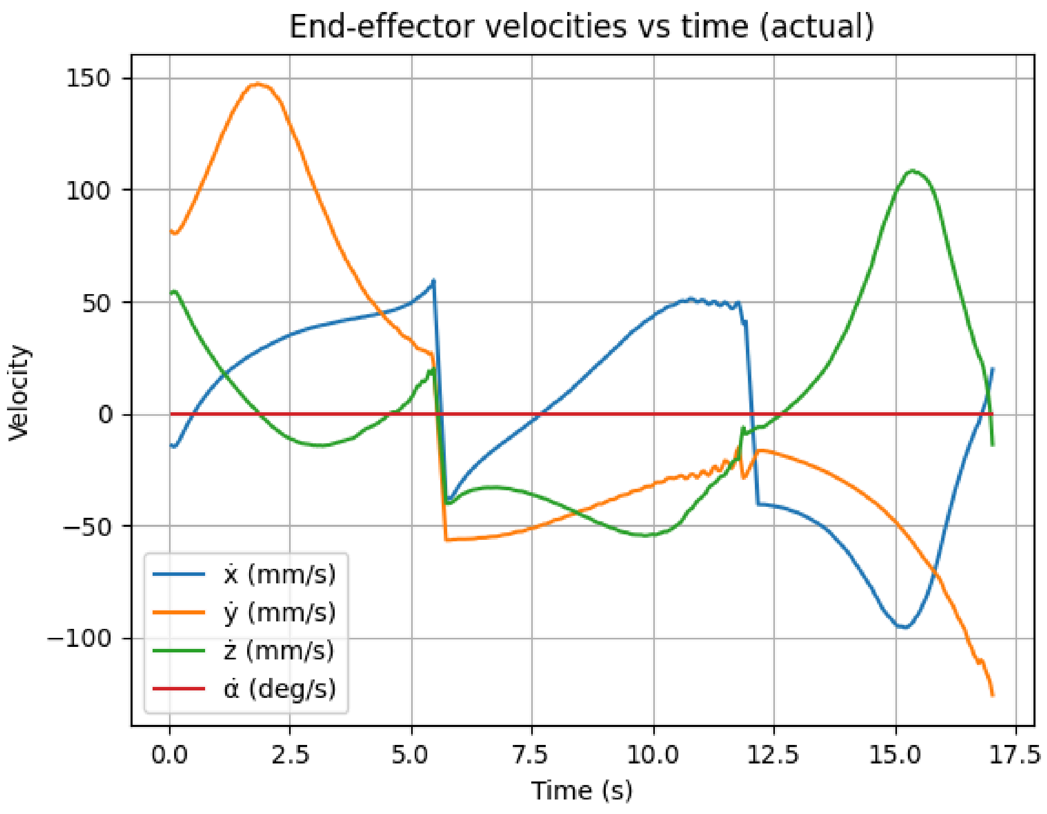


Figure 5: End effector Joint velocities

Figure 5 presents the measured end-effector velocity components over time during the velocity-based trajectory execution. The plot shows the ẋ, ẏ, and ż Cartesian velocity components along with the angular velocity α̇. Each segment corresponds to motion between waypoints, where the direction and magnitude of the velocity change according to the commanded path. The data demonstrates that the end-effector maintained an approximately constant magnitude of around 50 mm/s, with distinct transitions between trajectory edges. Small oscillations and discontinuities observed near the waypoint transitions are attributed to abrupt changes in motion direction and limited control loop resolution. Overall, the results confirm successful real-time execution of the commanded Cartesian velocities within the expected limits.

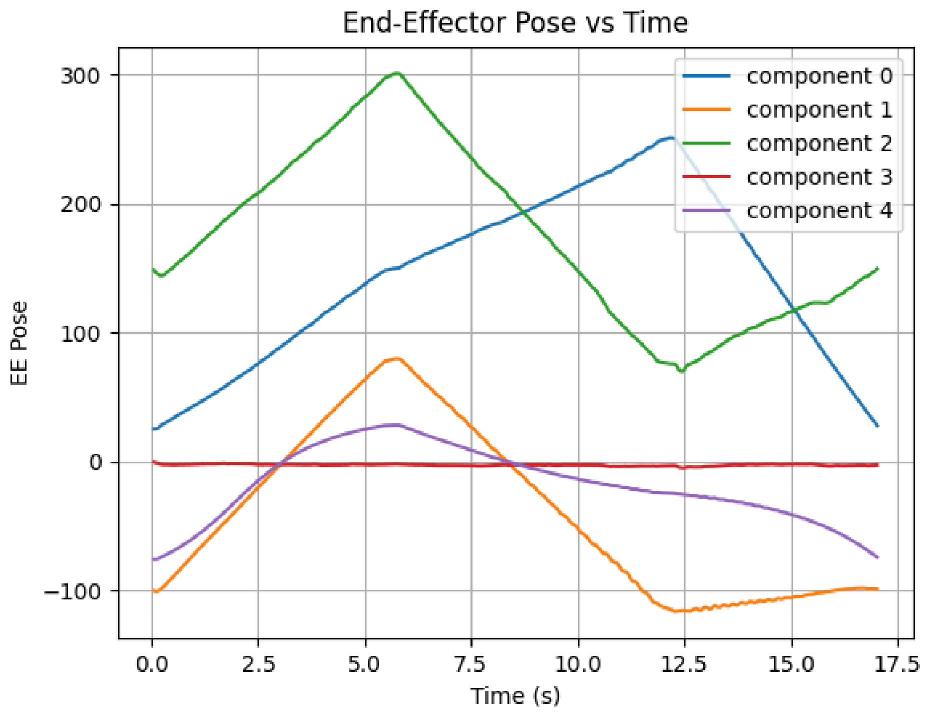


Figure 6: End Effector Pose vs Time

Figure 6 illustrates the end-effector pose components over time as the manipulator followed the commanded trajectory. Each line represents one Cartesian coordinate of the end-effector position throughout the motion sequence. The data shows distinct linear segments corresponding to each path leg, consistent with the triangular task-space trajectory. Minor nonlinearities and small deviations appear near the transition points, primarily due to the controller's finite update rate and accumulated joint-level rounding errors. Overall, the end-effector pose evolution confirms that the velocity-based control achieved stable and continuous motion between waypoints with minimal overshoot.

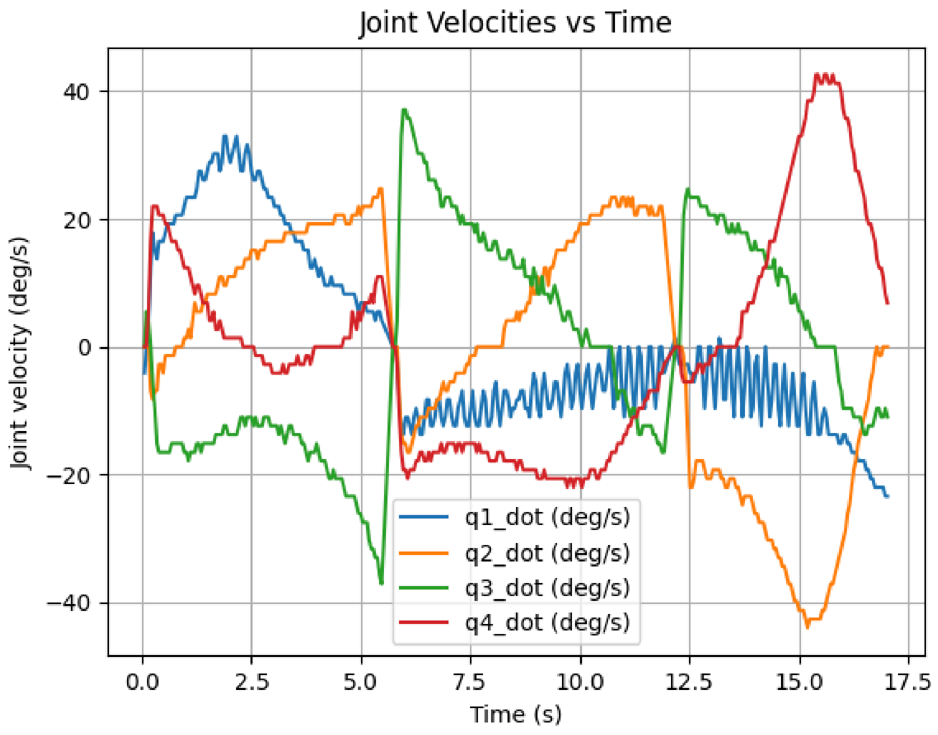


Figure 7: EE Joint Velocities vs time

Figure 7 displays the joint velocity profiles ($\dot{q}_1$–$\dot{q}_4$) throughout the experiment. The plotted data captures the continuous adjustment of joint speeds required to maintain constant Cartesian velocity in task space. Each joint exhibits smooth variations in speed, with transient peaks during transitions between trajectory segments. The observed high-frequency fluctuations correspond to feedback latency and quantization effects in motor position sensing. Despite this, the overall velocity behavior remains stable and within the safety limits, demonstrating effective coordination among all joints to achieve the desired end-effector motion. Although the joint velocities follow, a somewhat straight path, they do experience a lot of noise, this is mostly because of the servo motors capacity to keep a desired speed.

Comparison between lab 4 and Lab 5:

Path Quality:
The velocity-based control produced nearly straight task-space paths, as shown by the 3D trajectory in Figure 4. Minor deviations were visible near waypoint corners due to abrupt velocity direction changes. In comparison, the polynomial trajectory from Lab 4 generated smoother, pre-planned curves that adhered more precisely to the defined path, but lacked flexibility to adjust in real time.

Smoothness:
Polynomial trajectories yield inherently smoother joint motions since joint velocities are derived from continuous polynomial functions with guaranteed continuity up to acceleration. In contrast, the velocity-based controller updates velocities discretely each control cycle, leading to small

discontinuities when the desired direction changes. Despite this, the observed motion remained smooth enough for real-time execution without visible jittering.

Ease of Implementation:
Velocity control is simpler to implement conceptually, requiring only Jacobian computation and real-time feedback. Polynomial trajectory planning requires offline computation of boundary conditions and trajectory coefficients, which adds complexity. However, once implemented, polynomial methods provide predictable motion with minimal tuning effort.

Real-Time Adaptability:
Velocity control offers a clear advantage in adaptability. Since the end-effector velocity command is computed online, the manipulator can instantly respond to updated targets or obstacle avoidance routines. Polynomial trajectories, being pre-computed, cannot adjust mid-execution without full re-planning.

Tracking Accuracy:
Lab 4's polynomial trajectories provided slightly better waypoint accuracy because each position target was predefined and followed exactly through interpolation. Velocity control achieved comparable results but showed small steady-state errors near waypoints due to finite loop timing and noise.

CONCLUSION:

This lab successfully demonstrated the implementation and application of velocity kinematics on the OpenManipulator-X. By constructing the geometric Jacobian and using its pseudoinverse, we achieved real-time control of the end-effector to follow a triangular trajectory at a constant Cartesian velocity. The results verified the accuracy and responsiveness of the velocity-based approach, highlighting its adaptability compared to preplanned trajectories. Although minor noise and instability were observed, the experiment illustrated the effectiveness of Jacobian-based control for practical robotic motion. This lab deepened our understanding of differential kinematics and its significance in real-time robotic control systems.