Visual Servoing

Matthew Ybarra
EECS
College of Engineering
Berkeley, USA
mjybarra03@berkeley.edu

Tenzin Norphel
EECS
College of Engineering
Berkeley, USA
tnorphel@berkeley.edu

***Abstract*—In our project, we implemented controller "Jointspace Velocity Control" where we tune Proportional as "k_p" and Derivative as "k_d" to reduce jittering for sawyer arm. Moreover, design is to make sure the sawyer arm to reach to desired trajectory, and keep track of error in between those paths. In order to perform accurate trajectory, we have to consider delay and stability as the sawyer arm moves to its desired trajectory. Along with delay and stability, we need to account for error that comes from subtracting the current position from the desired position. As we get to the correct trajectory, we will plot the graph and see how it behaves, comparing "Actual" and "Desired" onto graph.**

I. Introduction

THIS PROJECT is based on how the interaction between controller as an input that feed into computer and see the end results of the output, which is based on end-effector of robotic arm. [1].

II. Methods

A. Jointspace velocity controller

In this controller, a PD controller is implemented using the error

$$e(t) = \theta_d(t) - \theta(t),$$

where we need to account for the current joint position $\theta(t)$ and the desired joint position $\theta_d(t)$. Therefore, $\theta_d(t)$ it is a position vector, $\frac{d\theta_d(t)}{dt}$ the velocity vector, and $\frac{d^2\theta_d(t)}{dt^2}$ the acceleration vector.

Therefore, joint-space velocity is easier to implement than torque because velocity does not require full dynamic implementation, but instead, tuning k_d and k_p only.

Here, we only care about the error because joint-space velocity is associated with position after taking the first derivative. So, we want our system to have high stability and accuracy in order to reach the desired trajectory. By adjusting k_d and k_p, we can reduce the error, maintain oscillations, and ensure that our trajectory stays within equilibrium.

Normally, the Integral would be incorporated making it a PID controller. Due to it being difficult to calculate the integral, we instead approximate it using an exponential moving average with an integral term k_w.

B. Jointspace torque controller

In this controller, we implement the same PD (proportional-derivative) controller. We use the Lagrangian equation to derive equations of motion and determine joint torques to reach the desired joint motions.

C. Workspace velocity controller

Here, unlike jointspace velocity and torque, where we compute θ to find the joint angles, instead we use the end effector (the end or tip of the arm) to calculate the joint angles using the Jacobian matrix.

D. Equations

JOINTSPACE VELOCITY CONTROLLER

Since the control law has the form:

$$\text{(Torque) } \tau = -k_v\dot{e} - k_p e$$

and the error e is defined as:

e = x - x_d, where x = current joint-space trajectory, x_d = desired joint-space trajectory

Therefore, in order to achieve joint-space velocity control, we take the derivative of position x and x_d to obtain velocity:

$$\frac{de}{dt} = \frac{dx}{dt} - \frac{dx_d}{dt}$$

We also incorporate the approximated integral component as:

$$\text{integral error} = k_w(\text{previous integral error}) + \text{current error}$$

Thus, we can rewrite the equation to obtain the joint velocity command:

$$\boxed{u = k_p e - k_d\dot{e} + \text{integral error}}$$

JOINTSPACE TORQUE CONTROLLER

The equation of motion is given by:

$$\tau_d(t) = M(q_d)\ddot{q}_d + c(q_d, \dot{q}_d)\dot{q}_d + g(q_d)$$

where:

- $M(q_d)$ is the mass inertia matrix
- $c(q_d, \dot{q}_d)$ represents the Coriolis matrix
- $g(q_d)$ is the gravitational force term

When dealing with torque control, we rewrite the equation as:

$$M(\theta)\ddot{\theta} + C(\theta,\dot{\theta})\dot{\theta} + N(\theta,\dot{\theta}) = \tau \quad (1)$$

where,

- $M(\theta)$ is the mass inertia matrix
- $C(\theta,\dot{\theta})$ is the Coriolis matrix
- $N(\theta,\dot{\theta})$ represents gravity
- τ is the torque

Final Control Law

The control force is given by:

$$\tau = \tilde{M}(\theta)(\ddot{\theta}_d - k_v\dot{e} - k_p e) + C(\theta,\dot{\theta})\dot{\theta} + N(\theta,\dot{\theta}) \quad (2)$$

where:

k_v and k_p are control gains used for stability. $\dot{e}$ and e represent error terms.

This expression is derived from workspace velocity control.

$$\tau = -k_v\dot{e} - k_p e, \quad (1)$$

$$e = \theta - \theta_d, \quad (2)$$

where, e is error

$$\frac{de}{dt} = \frac{dx}{dt} - \frac{dx_d}{dt}, \quad (3)$$

$$K_p(x - x_d) - k_d(\dot{x} - \dot{x}_d) = 0, \quad (4)$$

$$\tau_d(t) = M(q_d)\ddot{q}_d + C(q_d,\dot{q}_d)\dot{q}_d + g(q_d) \quad (5)$$

where:

- $M(q_d)$ is the mass-inertia matrix,
- $C(q_d,\dot{q}_d)$ represents the Coriolis matrix,
- $g(q_d)$ accounts for gravitational forces,
- τ_d is the control torque.

$$\boxed{\tau = \tilde{M}(\theta)(\ddot{\theta}_d - k_v\dot{e} - k_p e) + C(\theta,\dot{\theta})\dot{\theta} + N(\theta,\dot{\theta})} \quad (6)$$

WORKSPACE VELOCITY CONTROLLER

Here, we can reapply the same error definition from jointspace but now we incorporate jacobian.

Here, we can use the same error definition as:

$$e = \theta - \theta_d$$

The dynamics of the manipulator in joint-space has the form:

$$M(\theta)\ddot{\theta} + C(\theta,\dot{\theta})\dot{\theta} + N(\theta,\dot{\theta}) = \tau$$

from the joint-space torque control expression

The only change with this expression is to incorporate the Jacobian as:

$$\dot{x} = J(\theta)\dot{\theta}, \quad \text{where } J(\theta) = \frac{\partial f}{\partial \theta}$$

$$\dot{\theta} = J^{-1}\dot{x}$$

Writing the Dynamics in Terms of the Jacobian

$$\widetilde{M} = J^{-T}MJ^{-1}$$

$$\widetilde{C} = J^{-T}\left(CJ^{-1} + M\frac{d}{dt}(J^{-1})\right)$$

$$\widetilde{N} = J^{-T}N$$

$$F = J^{-T}\tau$$

Final Dynamics Equation

$$\boxed{\widetilde{M}(\dot{\theta})\ddot{x} + \widetilde{C}(\theta,\dot{\theta})\dot{x} + \widetilde{N}(\theta,\dot{\theta}) = F}$$

E. Tuning Procedure

For our tuning procedure, we initially set small values, such as $k_p = 0.05$ for all proportional gains. We also tested with $k_v = 0.01$, and additionally introduced k_w setting it to $k_w = 0.9$.

After testing for a while, we realized that increasing k_p helps improve the movement of the Sawyer robot, allowing it to move more smoothly. On the other hand, k_v needs to be kept as small as possible to ensure that the Sawyer does not get stuck in one position.

For the **Joint-Space Velocity Controller**, the gains are:

$$K_p = \begin{bmatrix}0.4 & 2 & 1.7 & 1.5 & 2 & 2 & 3\end{bmatrix}$$
$$K_v = \begin{bmatrix}2 & 1 & 2 & 0.5 & 0.8 & 0.8 & 0.8\end{bmatrix}$$
$$K_w = \begin{bmatrix}2 & 1 & 2 & 0.5 & 0.8 & 0.8 & 0.8\end{bmatrix}$$

III. Experimental Results

As we chose joint-space velocity control for the controller, our primary design is to ensure that the Sawyer arm reaches the desired trajectory and keeps track of the error along the path. To achieve an accurate trajectory, we tuned K_p and K_v to be as small as possible in order to avoid delays and prevent the system from struggling to reach its destination.

In terms of experiments, we first ensured that all three trajectories ran with our controller of choice, the joint space controller. This was to ensure that our math was correct and that our controller code was error-free. While no specific experiments were run for this, we heuristically changed values to ensure that all three trajectories ran all the way through.

For the follow_ar_tag function, we took inspiration from the execute_path function as well as methods provided within the paths.py file. Specifically, we used get_ik and trajectory_point to help us calculate desired velocities and positions for our controller.

For our experiments, we decided to tune values within follow_ar_tag and test the sawyer with both the controller active and with only feedforward. Specifically, our implementation of follow_ar_tag keeps track of the distance between the end effector and the ar tag, divides this path into smaller points in space, and travels to those points in space. It also uses these points to calculate the velocity. The values we tuned

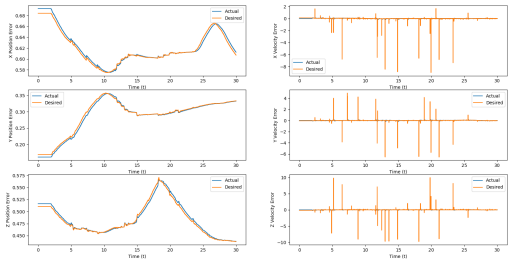

Fig. 1. Open Loop Controller, Delta-Step-Rate = 0.1, Loop-Rate = 300 ms

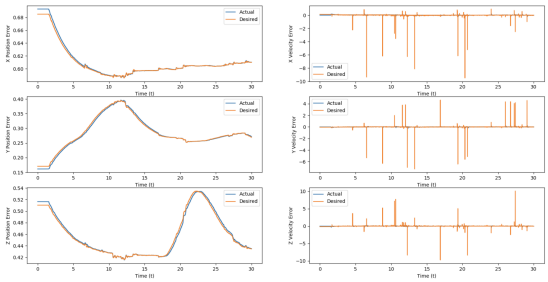

Fig. 2. Jointspace Controller, Delta-Step-Rate = 0.1, Loop-Rate = 300 ms

and experimented on were the amount of subdivisions (Delta-Step-Rate) we decided to do as well as varying the rate the function would loop through (loop-rate), as this determined how fast the velocity would be.

IV. Discussion

We decided to test the following value pairs for the delta step rate and the loop rate respectively: 0.1 and 300 ms, 0.2 and 250 ms, 0.3 and 200 ms. The logic for these values comes from how our code was implemented. The delta step rate determines how many points to follow from the end effector to the ar tag so a value of 0.1 would split the path in 10 points. The smaller the number the more points the end effector would have to travel. For the loop rate, we use it to calculate the velocity and a higher rate directly correlates with a higher velocity, and a lower rate with a lower velocity. Thus, we decided to have delta step rate and loop rate pairs that would correlate with each other; more points and slower velocity to have the end effector travel slower and fewer points and higher velocity to have the end effector travel faster.

Looking at Figure 1 and Figure 2, we compared delta step rate and loop rate 0.1 and 300 ms respectively, and with the controller on and off. Figure 2 shows the position error being much smoother and having an easier time being brought down to zero. You can also see in the velocity error spikes. These represent times when the end effector would slightly jolt which can be seen in the video linked within this paper. When compared to figure 1, the controller was able to keep the error smoother and experience less jolting. This specific input resulted in the end effector moving the slowest which also helped contributed to overall smoothing.

Looking at Figure 3 and Figure 4 we compared delta step rate and loop rate 0.2 and 250 ms respectively, and with the controller on and off. You can see the open loop controller had a harder time bringing the error down compared to the jointspace controller. The jointspace controller also had less jolts when compared to the open loop controller.

Looking at Figure 5 and Figure 6 we compared delta step rate and loop rate 0.3 and 200 ms respectively, and with the controller on and off. When compared to other test rates, these rates were the least smooth with having many more jolts as well as having a harder time bringing the error down. The open loop had more jolts than the jointspace controller as seen in the velocity error graphs. However, both position error graphs for the jointspace and open loop controller had, heuristically, the same result.

Out of the 3 configurations and 2 controllers, we believed that the Jointspace controller with delta step rate and loop rate values of 0.2 and 250 ms respectively behaved the best; as seen in figure 4. This configuration had values in between the others so we believed that it acted as a sweet spot between not being too slow and too fast.

V. Difficulties

Our controller and visual serving were far from perfect. As can be seen in the attached video link as well as the various graphs, the jointspace controller, while better than open loop, has much more room for improvement. In the future, we would like to test different k_w, k_p, and k_v values to tune the controller better. We would also like to try experimenting and implementing the other controllers that we did not implement and testing them against our current jointspace controller. Along with this, our code could also be improved to help mitigate the stuttering and jolting the sawyer experienced. In the future, we would like to continue testing different rates as well as experimenting with different approaches calculating desired velocities and positions. We believe that the way we calculated these values contributed to the jolting seen in the sawyer and trying different methods to calculate these values such as taking inspiration from the interpolate function found in controllers.py.

VI. Applications

In general, visual servoing can be used in the medical field, especially for different surgical methods, and maintaining accuracy when treating injuries and stitching wounds. All of these applications require precision and efficiency. [2] In a medical procedure, surgery requires an incision that allows the minirobot to explore inside and reach the desired trajectory (i.e., pathogen).

During surgical operations, robotic arms are used to perform incisions and then perform laparoscopy, surgery inside the gastrointestinal tract. Even surgical operations like thoracoscopy for removing lung tissues for patients who have

breathing difficulty or people who constantly smoke. All of these operation require computer vision as well as visual servoing because it is essential to send minitubes through small apertures to track down the root of the disease. [3]

Even in remote areas, visual servoing serves as a key role in aiding farmers, for irrigation, different agricultural methods, for collecting fruits with the help of multifingered robot hands. [3]

VII. Conclusion

As we obtained all the graphs for Delta step rate for both the Open Loop Controller and Jointspace Controller, we noticed that both the actual and desired trajectories lie on top of each other. In other words, the interaction between the actual and desired graph works well in the jointspace velocity controller. After tuning k_p and k_v values to the desired value, the error between the current and desired position reduced to a significant amount. The same goes for velocity after taking the time derivative of position. Therefore, implementing a jointspace velocity controller is more time efficient in terms of calculating joint angles.

Acknowledgment

The authors would like to express their sincere gratitude to EE106B Distinguished Professor S. Shankar Sastry for his guidance and support throughout the project. Special thanks to our lab TAs, K. El-Refai and Yarden Goraly, for their invaluable assistance in the development and conceptual understanding of key controller components, including PID control, joint-space velocity control, joint-space torque control, and workspace velocity control.

Additionally, we extend our appreciation to Jaeyun Stella for her support in logistics and policy-related aspects of the project.

Graphs

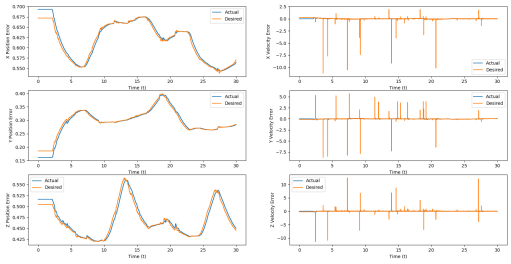

Fig. 3. Open Loop Controller, Delta-Step-Rate = 0.2, Loop-Rate = 250 ms

References

[1] R. M. Murray, Z. Li, and S. S. Sastry, *A Mathematical Introduction to Robotic Manipulation*, Boca Raton, FL, USA: CRC Press, 1994, p. 83.

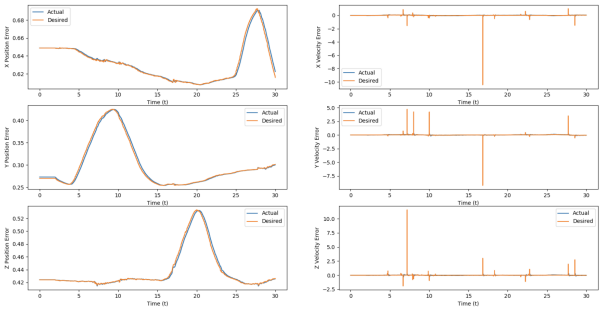

Fig. 4. Jointspace Controller, Delta-Step-Rate = 0.2, Loop-Rate = 250 ms

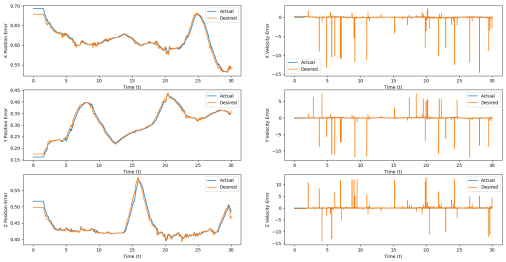

Fig. 5. Open Loop Controller, Delta-Step-Rate = 0.3, Loop-Rate = 200 ms

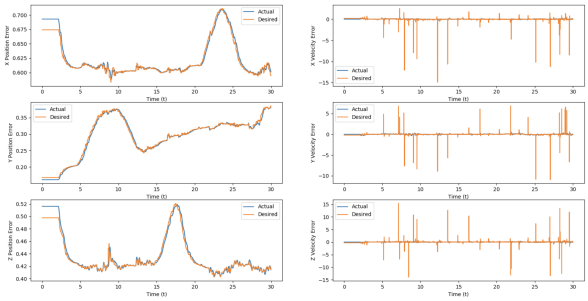

Fig. 6. Jointspace Controller, Delta-Step-Rate = 0.3, Loop-Rate = 200 ms

[2] R. M. Murray, Z. Li, and S. S. Sastry, *A Mathematical Introduction to Robotic Manipulation*, Boca Raton, FL, USA: CRC Press, 1994, p. 398.

[3] R. M. Murray, Z. Li, and S. S. Sastry, *A Mathematical Introduction to Robotic Manipulation*, Boca Raton, FL, USA: CRC Press, 1994, p. 398.

Appendix

- GitHub Repository: tnorphel
- Visual Servoing Implementation

AUTHORS

Tenzin Norphel is currently pursuing EECS (Electrical Engineering and Computer Science) at University of California, Berkeley

Matthew Ybarra is currently pursuing EECS (Electrical Engineering and Computer Science) at University of

California, Berkeley