An Analytical Comparison of Trajectory Generation Methods

Charlie Mawn, Luca Odio, Alex Qazilbash

https://github.com/c-mawn/final-funrobo

Our final project details an analytical comparison of several different trajectory generation methods. We implemented five different methods in Python, and, using the provided visualizer tool, we examined the differences between each method.
We have also written a primer on time-optimal time scaling, a method of optimizing robot trajectory to maximize efficiency, completing the trajectory in as little time as possible.

Polynomial Trajectory Generation

Polynomial generation uses an nth order polynomial to generate a smooth trajectory between two points. Polynomials are continuous and differentiable, so position-dependent properties like velocity and acceleration may also be continuous and differentiable depending on the order of the original polynomial.

The trajectory is constructed as a position function, like the following:

$$q(t) = a_0 + a_1 t + a_2 t^2 + \ldots + a_n t^n$$

We can compute the velocity and acceleration and other position-dependent derivatives in relation to the position function

$$v(t) = \frac{d}{dt} q(t) = a_1 + 2a_2 t + 3a_3 t^2 + \ldots + (n) a_n t^{n-1}$$

$$a(t) = \frac{d^2}{dt^2} q(t) = 2a_2 + 6a_3 t + 12a_4 t^2 + \ldots + (n)(n-1) a_n t^{n-2}$$

$$\ldots$$

$$f(t) = \frac{d^k}{dt^k} q(t) = k!\, a_k + (k+1)!\, a_{k+1} t + \frac{(k+2)!}{2!} a_{k+2} t^2 + \ldots + \frac{n!}{(n-k)!} a_n t^{n-k}$$

for $k = \frac{n-1}{2}$. Since the final and initial conditions are known for the position, position-dependent derivatives, and time, it is possible to find coefficients a_0- a_n. Take that,

$$x = \left[a_0,\ a_1,\ \ldots,\ a_n\right]$$

and

$$b = \left[q_{0,i},\ v_0,\ a_0,\ \ldots, \frac{d^k}{dt^k}q(t)_0,\ \ q_{f,i},\ v_f,\ a_f,\ \ldots, \frac{d^k}{dt^k}q(t)_f\right]$$

where $i \in \{x,\ y,\ z\}$ and indicates the axis of reference. Then, construct matrix A such that,

$$Ax = b$$

where the elements of A are the coefficients of each function on x, also written here as,

$$\begin{aligned} A = [\ & 1,\ t_0,\ {t_0}^2,\ \ldots,\ {t_0}^n, \\ & 0,\ 1,\ 2t_0,\ 3\,{t_0}^2,\ \ldots,\ (n)\,{t_0}^{n-1}, \\ & 0,\ 0,\ 2,\ 6t_0,\ 12{t_0}^2,\ \ldots,\ (n)(n-1){t_0}^{n-2}, \\ & 0_1,\ 0_2,\ \ldots,\ 0_k,\ k!,\ (k+1)!\,t_0,\ \frac{(k+2)!}{2!}{t_0}^2,\ \ldots,\ \frac{n!}{(n-k)!}{t_0}^{n-k} \\ & 1,\ t_f,\ {t_f}^2,\ \ldots,\ {t_f}^n, \\ & 0,\ 1,\ 2t_f,\ 3\,{t_f}^2,\ \ldots,\ (n)\,{t_f}^{n-1}, \\ & 0,\ 0,\ 2,\ 6t_f,\ 12{t_f}^2,\ \ldots,\ (n)(n-1){t_f}^{n-2} \\ & 0_1,\ 0_2,\ \ldots,\ 0_k,\ k!,\ (k+1)!\,t_f,\ \frac{(k+2)!}{2!}{t_f}^2,\ \ldots,\ \frac{n!}{(n-k)!}{t_f}^{n-k}\ ] \end{aligned}$$

Then, solve

$$x = A\backslash b$$

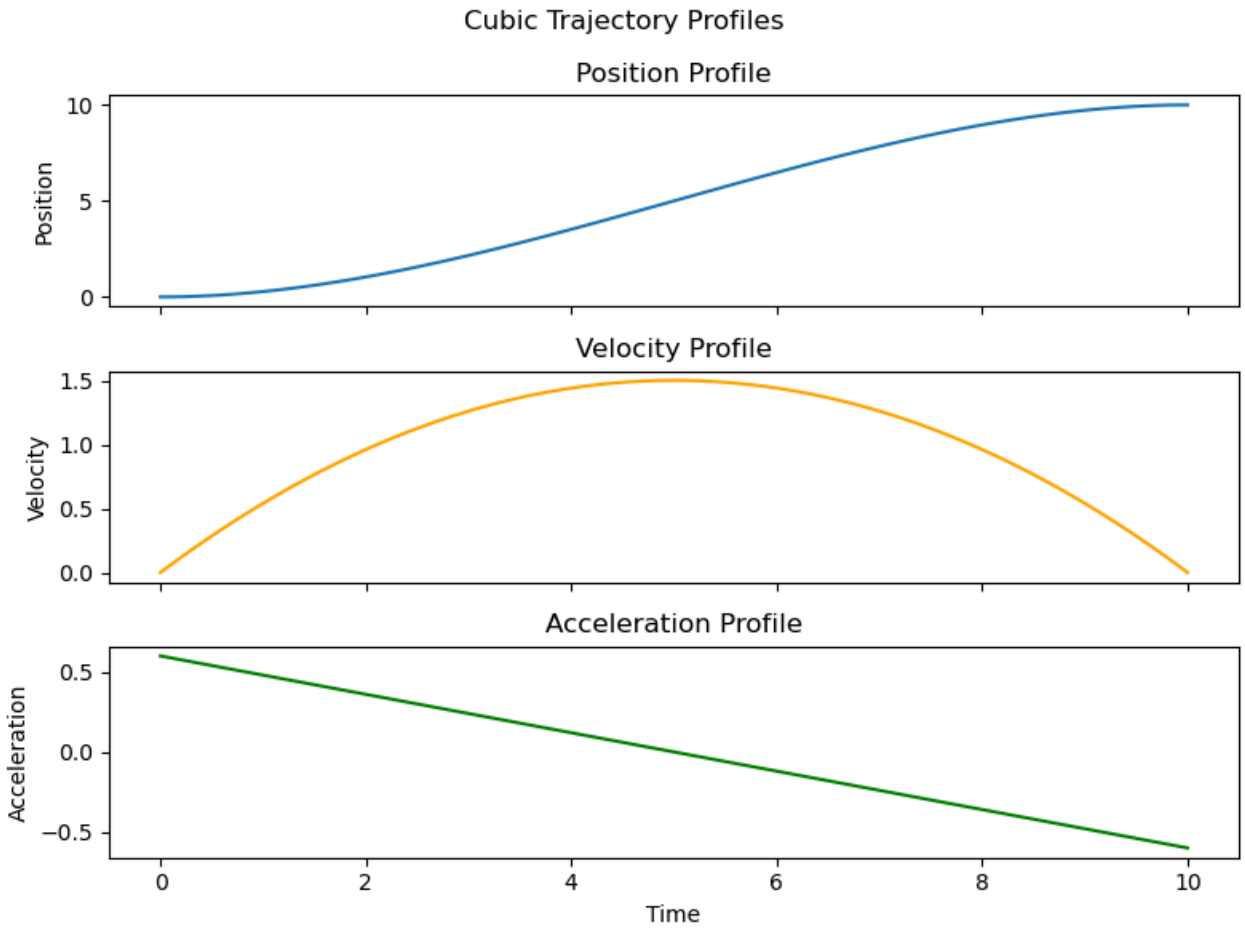
and the path completely defined. The path is parameterized over time into a trajectory according to how many time steps are appropriate for the situation.

Smooth trajectories and dependent properties are important because piecewise and discontinuous trajectories can result in greater wear on the physical system.

We implemented three different polynomial trajectory generation methods: cubic, quintic and septic.

Cubic Polynomial

Our first method uses cubic polynomials to generate a path between points. The position of the end effector follows a third order polynomial function. Thus, trajectories generated via this approach are continuous up to velocity. Note that the acceleration profile is linear, so it does not always begin and end at zero.

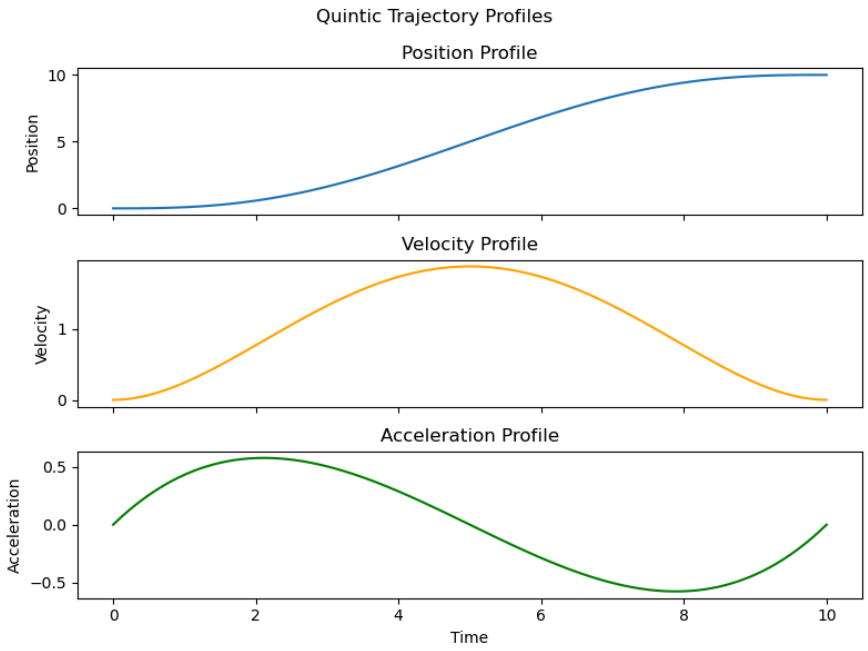


This Video shows our implementation of the cubic polynomial trajectory generation method in the visualizer tool.

Quintic Polynomial

The quintic method uses fifth order polynomials to generate the path between points. As a polynomial method it follows the same theory as the cubic approach. However, the higher order allows for increased differentiability, so this approach generates trajectories which are continuous up to acceleration.

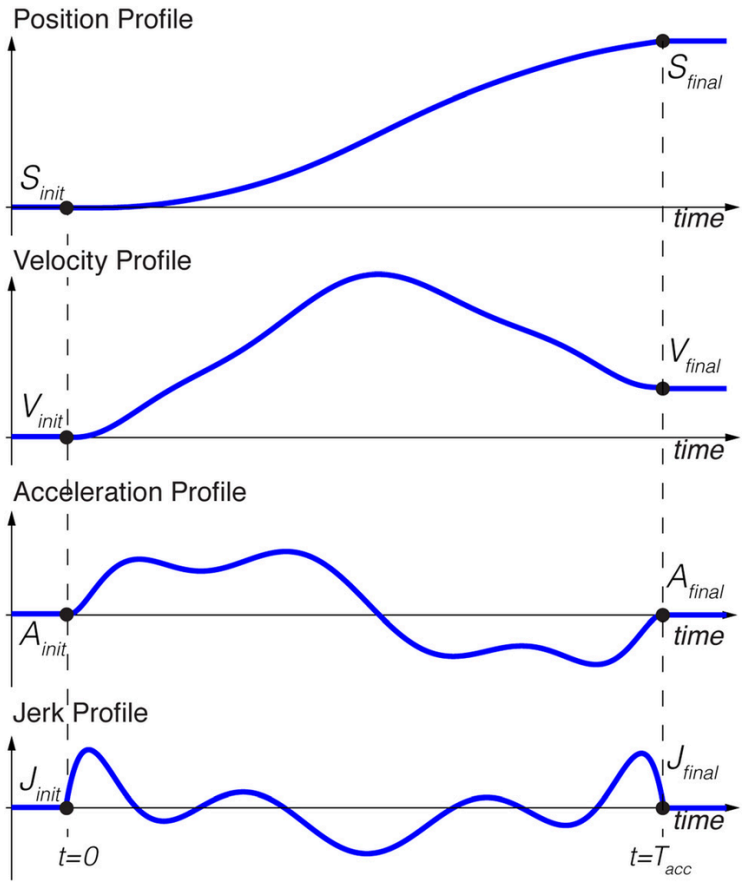
For quintic trajectories, the acceleration profile is cubic. Thus, the trajectory can be designed to begin and end with zero acceleration.



This Video shows our implementation of the quintic polynomial trajectory generation method in the visualizer tool.

Septic Polynomial

In following with the theme, the septic method generates a smooth path between two points using a seventh order polynomial. As before, the higher degree of polynomial allows for increased differentiability. In this case the jerk profile is quadratic. Controlling jerk is important for smooth motion in a robot, and the quadratic jerk profile of this method means that the trajectory can be designed such that the jerk begins and ends at zero.



This Video shows our implementation of the septic polynomial trajectory generation method in the visualizer tool.

Trapezoidal

The trapezoidal trajectory generation method is very useful because it takes into account system limits. Depending on the context, trapezoidal trajectories can be optimized for parameters like speed, energy consumption, and wear on motors.

Typically, the motion is broken up into three distinct phases: acceleration, constant velocity, and deceleration.

The peak velocity is defined as follows

$$\frac{q_{f,i}-q_0}{t_f} < V \leq \frac{2(q_{f,i}-q_0)}{t_f}$$

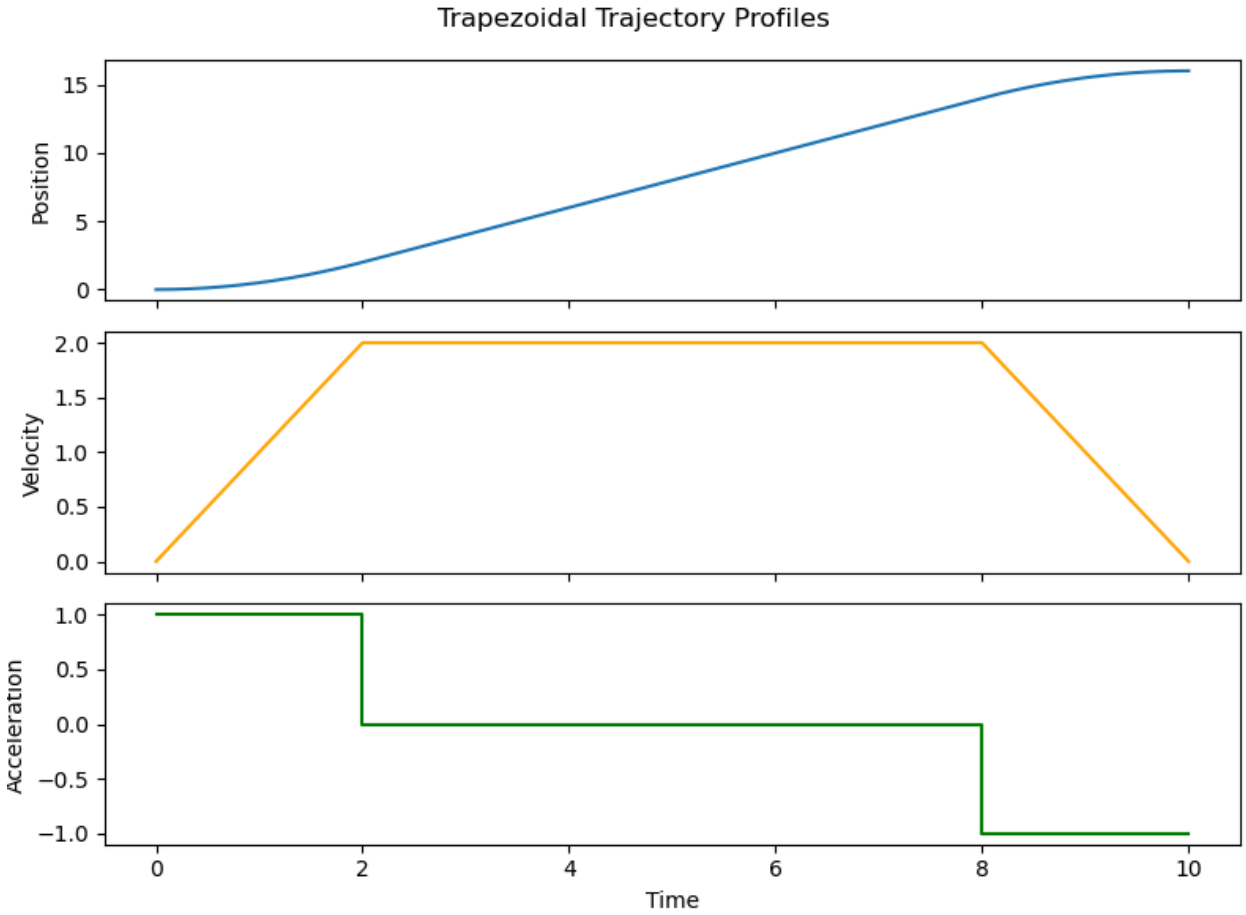
where $i \in \{x,\ y,\ z\}$ and indicates the index (axis) for which the equation is being evaluated. The variables q_0 and q_f represent the initial and final positions of the end effector. Similarly, t_f is the final timestep for the trajectory. Using the velocity V, it is possible to calculate the blend time,

$$t_b = \frac{(q_0 - q_f + V t_f)}{V}$$

and the peak acceleration,

$$\alpha = \frac{V}{t_b}$$

The blend time is how long the acceleration and deceleration phases last. Some phases may be skipped or modified if the time or distance doesn't allow for a full acceleration and deceleration period. Concatenating these phases results in the trapezoidal trajectory profile below.



For this project, we only implemented point-to-point trapezoidal trajectory generation so the initial and final velocities were always zero.

Notably, the velocity and acceleration profiles are discontinuous. Such discontinuity can result in jerky motion, which, in a physical robot, results in extra wear on robotic components. This can be mitigated by integrating parabolic blends, although our implementation does not include this feature.

This Video shows our implementation of the trapezoidal trajectory generation method in the visualizer tool.

Natural Spline

The natural spline (in this scenario the Cubic Bézier spline) is used as an interpolation method where the robot trajectory must pass through multiple points, instead of just a start and end point. This is important for trajectory generation because it is very common for a robot arm to traverse between multiple waypoints, instead of just two.

To accomplish this goal of passing through points, we need to create a path that has a few key restrictions:

- The end effector pose must pass through each of the waypoints
- The position of the end effector must be continuous across the entire path
- The velocity of the end effector must be continuous across the entire path
- The acceleration of the end effector must be continuous across the entire path

The Cubic Bézier spline is perfect for this task because it has C2 continuity (continuous through acceleration). Read this for more information about cubic splines.

For our implementation, we used the *`scipy.interpolate.CubicSpline`* module to perform the calculations in python, where we pass in the time vector, and each of the waypoints. Read the `scipy.interpolate.CubicSpline` documentation here.

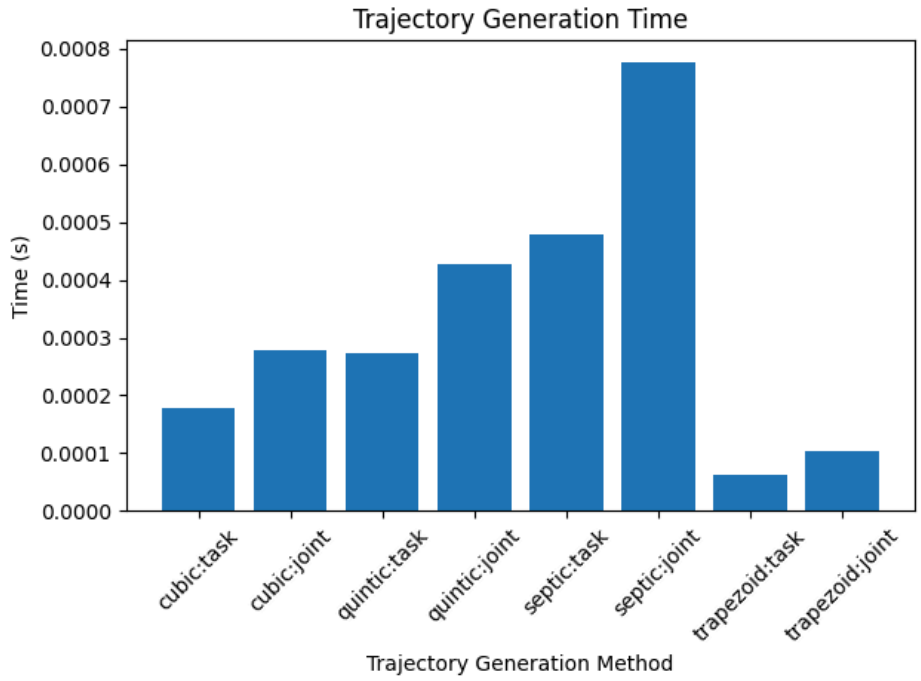
This Video shows our implementation of the spline trajectory generation in the visualizer tool.

Analysis

Our primary goal of the project is to analyze the key distinctions between the implemented trajectory generation methods.

The most important metric we analyzed was computational efficiency. This is an important metric because real time trajectory generation is super important in robotic systems, and minimizing the time it takes to calculate those trajectories can make the whole system run faster.

The graph below shows the average time it took to compute the trajectory for each of the different generation methods (cubic, quintic, septic, trapezoidal) in task space and joint space.



As we can see from the data, the trapezoidal trajectory generation methods were computed extremely fast, likely due to the fact that it does not need to do any matrix solves to compute the trajectory. However, the trapezoidal method is not continuous in acceleration, thus it is not as robust in robotic arms that are constantly accelerating and decelerating quickly.

For the other three methods, higher order polynomials result in longer computational times, seemingly with an exponential relationship. So, the higher the order of the polynomial, the system will have a smoother movement, but the computational efficiency is much lower.

Spline trajectory generations are not listed in this graph because they are used for inherently different tasks. Splines are best suited for multiple waypoints, where these methods are best for point-to-point trajectories.

So, looking at computational efficiency, trapezoidal trajectory generation is clearly the fastest to be computed.

Outside of just computational efficiency, some trajectory generation methods are better than others for specific tasks for some examples:

- Splines are best for multi-point trajectories
- Quintic is best for a compromise between smooth motion and computational efficiency

- Septic is best for very smooth motion, without any computational efficiency constraint

Time-Optimal Time Scaling

Time-optimal time scaling is a process which optimizes the robot trajectory to minimize the time needed to complete the path. See our primer (attached in our project files) for more information.

Long Link:
https://docs.google.com/document/d/12L1_ClMs2Dt-MmfMWh8l-gK26NiLVoOqhfvHx13QrLM/edit?usp=sharing

Kinematics Derivation

The trajectory generation methods used in this project rely on the derivations and code for DH tables, jacobian matrices, and inverse kinematics, as implemented in the previous mini-projects. The following is a documentation of the theory used to find these derivations.

Analytical Inverse Kinematics Derivation

5-DOF Robot Diagram:

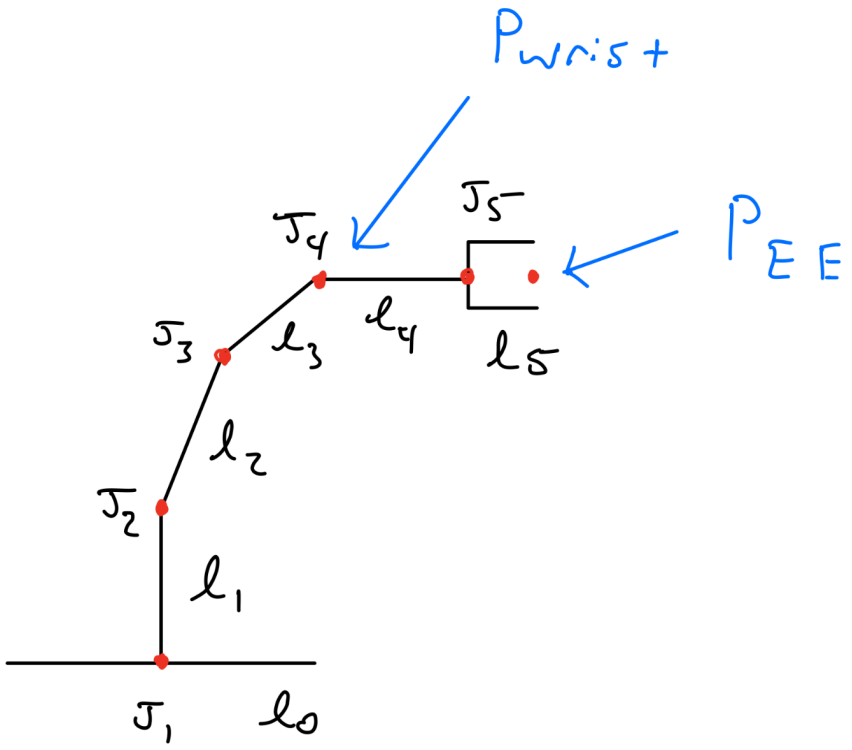


Fig. 1
A diagram of the 5-DOF robot with joints and links labeled

This diagram outlines the links and joints of the arm as well as indicating the position of the end effector and the wrist for kinematic decoupling. Kinematic decoupling is the process of splitting up the inverse kinematic derivation into two different problems which makes solving each problem simpler. In this case we are splitting the arm at the wrist because joint 5 only affects the orientation of the end effector, not the position.

The general process for solving this inverse kinematics problem is to:

1. Solve for $\Theta_1,\Theta_2,\Theta_3$ geometrically
2. Compose a rotation matrix ($^0R_3$) from the base frame to the wrist
3. Solve for $^3R_5$ using the $^0R_3$ matrix
4. Solve for Θ_4,Θ_5 using $^3R_5$

1. Solving for Θ_1, Θ_2, Θ_3 Geometrically

Using kinematic decoupling, finding $\Theta_1,\Theta_2,\Theta_3$ becomes much easier. To do this, we can look at the section of the arm from the base to the wrist from two different perspectives. The first one is viewing the arm from the top, and the second is viewing the arm from a frame that remains parallel to the links as Θ_1 changes.

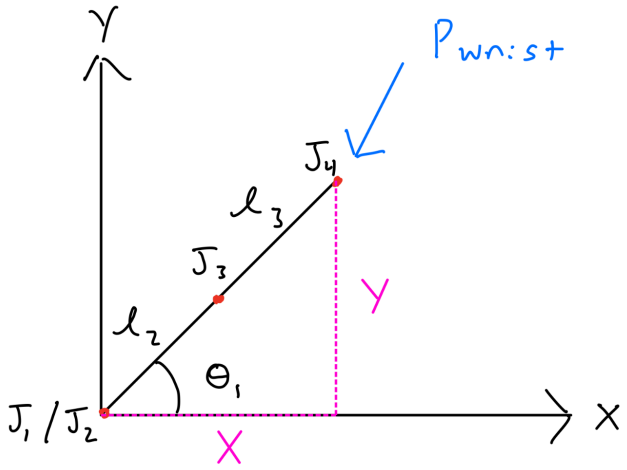


Fig. 2

The section of the arm from base to wrist viewed from the top down, making solving for Θ_1 much simpler.

Looking at this diagram we can see that the only thing influencing the angle of Θ_1 is the position of the wrist. Because this is an inverse kinematics problem, we had the position of the end effector and not the position of the wrist. To find that we used the following equation.

$$P_{wrist} = P_{EE} - (\ell_4 + \ell_5)\ {}^0R_5\ Z_1$$

In essence, this equation uses the position of the end effector and subtracts the last two links (which are always co-linear) which are rotated using the Euler rotations (transformed into a rotation matrix).

With the position of the wrist, we easily found Θ_1 using the equation:

$$\Theta_1 = Tan^{-1}(y/x)$$

However, the rotation of the arm can also be flipped by 180 degrees and have the joints bend the opposite direction in order to reach the same end effector position. This means that there is a second solution for Θ_1 which is calculated as follows, and shown in the diagram below:

$$\Theta_1 = Tan^{-1}(y/x) +/- \pi$$

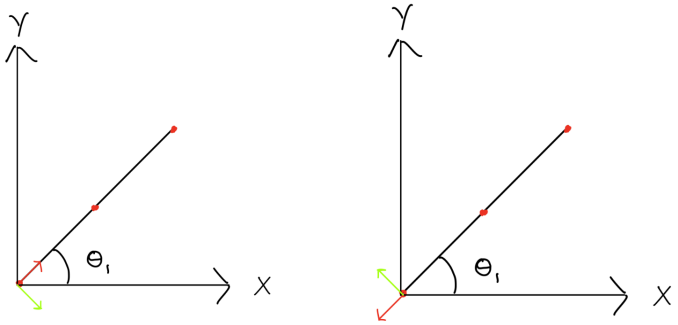


Fig. 3
Although the wrist still reaches the same position, Θ_1 is pointing in the opposite direction as indicated by the frame

With Θ_1 calculated, both Θ_2 and Θ_3 can be calculated like a 2-DOF arm using a plane that is parallel to the arm as Θ_1 changes.

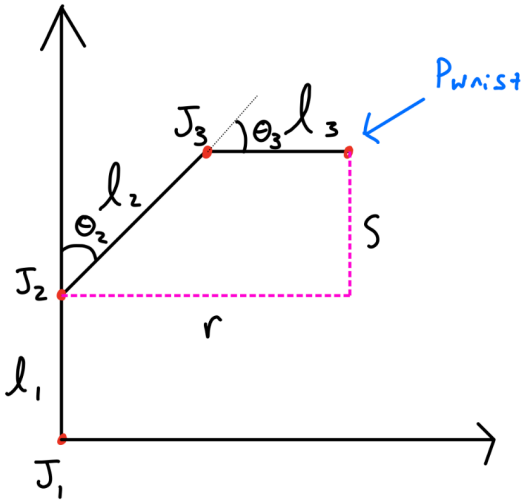


Fig. 4
The section of the arm from base to wrist viewed from a frame that stays parallel to the links of the arm as Θ_1 changes, essentially making this a 2-DOF arm.

Because the plane is rotating parallel to the arm, it no longer operates in the x-y plane. However, we were able to redefine it by finding the radius (r) of the arm, and the height of the second and third link (s) by subtracting link 1 from the z-height of the wrist. Both of these define this offset plane. However, since r is defined by a square root, there must be a positive and negative solution. This is because for different solutions of Θ_1 the arm may need to extend in a positive or negative solution. This is shown in both the equations and the diagrams below.

$$r = +/- \sqrt{(P_{Wrist\,X})^2 + (P_{Wrist\,Y})^2}$$

$$s = P_{Wrist\,Z} - l_1$$

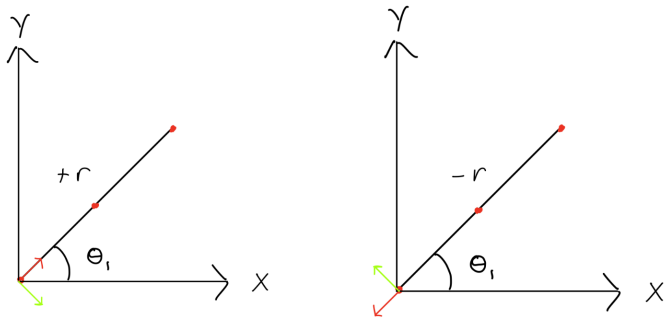


Fig. 5

Depending on which way Θ_1 may be rotated and the target wrist position, r may need to be either positive or negative, changing the solution of Θ_2

With the alternative plane defined we can then solve for Θ_2 and Θ_3 using the geometric relationships shown in the diagram below. However, because of there are two different configurations for the arm ("elbow up" and "elbow down") there are also two solutions for Θ_3.

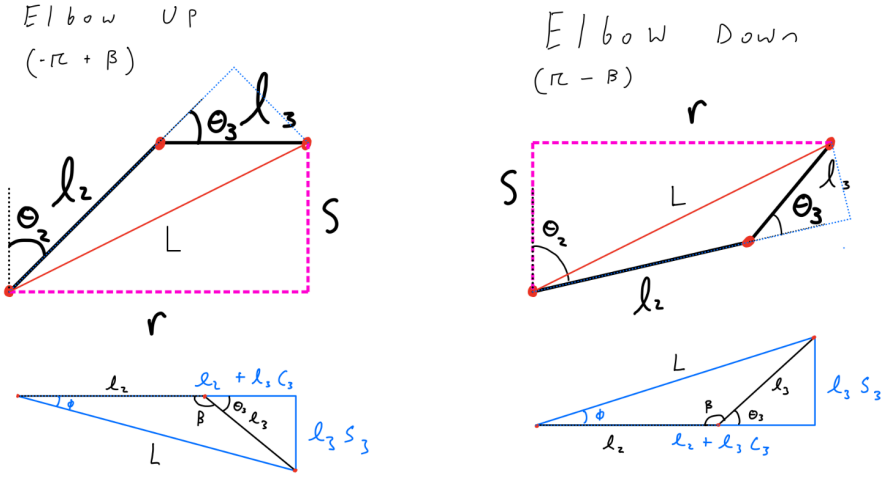


Fig. 6

There are two different configurations for Joint 2 and Joint 3 (elbow up/elbow down) resulting in two different solutions for Θ_3

Both Θ_3 and Θ_2 can be calculated via the equations below. Θ_2 is a function of r, so depending on which solution of r is used, a different solution is calculated.

$$\beta = cos^{-1}\left(\frac{l_2^2 + l_3^2 - L^2}{2l_2l_3}\right)$$

$$\Theta_3 = +/- (\pi - \beta)$$

$$\phi = tan^{-1}\left(\frac{l_3S_3}{l_2 + l_3C_3}\right)$$

$$\Theta_2 = tan^{-1}\left(\frac{r}{s}\right) - \phi$$

2. Calculating ${}^0R_3$ Using Θ_1, Θ_2, Θ_3

Using our DH table from the previous mini project, and plugging these theta values into our derived rotation matrices (shown below), we were able to compose a rotation matrix from the base frame to the wrist frame.

Joint	θ	d	a	α
1	θ_1	l_1	0	$\pi/2$
1.5	$\pi/2$	0	0	0
2	θ_2	0	l_2	π
3	θ_3	0	l_3	π
4	θ_4	0	l_4	0
4.5	$-\pi/2$	0	0	$-\pi/2$
5	θ_5	l_5	0	0

Fig. 7
DH Table for the 5-DOF Arm from MP1

$${}^0R_1 = \begin{bmatrix} C\theta_1 & 0 & S\theta_1 \\ S\theta_1 & 0 & -C\theta_1 \\ 0 & 1 & 0 \end{bmatrix}$$

$${}^1R_{1.5} = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$${}^{1.5}R_2 = \begin{bmatrix} C\theta_2 & S\theta_2 & 0 \\ S\theta 2 & -C\theta_2 & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

$$^{2}R_{3} = \begin{bmatrix} C\theta_3 & -S\theta_3 & 0 \\ S\theta3 & C\theta_3 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$^{0}R_{3} = {}^{0}R_{1} * {}^{1}R_{1.5} * {}^{1.5}R_{2} * {}^{2}R_{3}$$

3. Solve for $^{3}R_{5}$ Using $^{0}R_{3}$

Since we have the rotation matrix from the base frame to the end effector (using Euler angles) and the rotation matrix from the base frame to the wrist frame, it becomes trivial to find the rotation matrix from the wrist frame to the end effector frame.

$$^{3}R_{5} = {}^{0}R_{5}^{T} * {}^{0}R_{3}$$

$$^{3}R_{5} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}$$

4. Solve for Θ_4,Θ_5 using $^{3}R_{5}$

We now have $^{3}R_{5}$ calculated numerically, and if we can also calculate it symbolically, we can solve for individual terms within the symbolic matrix. Using the DH table above and the rotation matrices we derived from the first mini project, we were able to symbolically multiply them together to get a symbolic $^{3}R_{5}$.

$$^{3}R_{4} = \begin{bmatrix} C\theta_4 & -S\theta_4 & 0 \\ S\theta_4 & C\theta_4 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$^{4}R_{4.5} = \begin{bmatrix} 0 & 0 & 1 \\ -1 & 0 & 0 \\ 0 & -1 & 0 \end{bmatrix}$$

$$^{4.5}R_{5} = \begin{bmatrix} C\theta_5 & -S\theta_5 & 0 \\ S\theta_5 & C\theta_5 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$ {}^{3}R_{5} = {}^{3}R_{4} * {}^{4}R_{4.5} * {}^{4.5}R_{5} $$

$$ {}^{3}R_{5} = \begin{bmatrix} C\theta_5 S\theta_4 & -S\theta_4 S\theta_5 & C\theta_4 \\ -C\theta_4 C\theta_5 & C\theta_4 S\theta_5 & S\theta_4 \\ -S\theta_5 & -C\theta_5 & 0 \end{bmatrix} $$

With this we can see that certain terms only contain a Θ_4 or Θ_5 term, meaning that we can quite easily solve for these values using our numerical ${}^{3}R_{5}$.

$$ \Theta_4 = tan^{-1}(\frac{r_{23}}{r_{13}}) $$

$$ \Theta_5 = tan^{-1}(\frac{-r_{31}}{-r_{32}}) $$

5. Verify Possible Solutions

Now with all 5 theta values we can actually set the position of our 5-DOF arm. However, since we had multiple solutions for Θ_1, Θ_3, and r, we actually have 8 different potential solutions. The different combinations are shown below, although the different solutions for r are represented as a difference in sign for Θ_2 since that is the only joint angle directly affected by r.

Θ_1	Θ_2	Θ_3	Θ_4	Θ_5
+0	+r	+(π - β)	+0	+0
+0	+r	-(π - β)	+0	+0
+0	-r	+(π - β)	+0	+0
+0	-r	-(π - β)	+0	+0
+π	+r	+(π - β)	+0	+0
+π	+r	-(π - β)	+0	+0
+π	-r	+(π - β)	+0	+0
+π	-r	-(π - β)	+0	+0

Fig. 8

The table of all possible solutions for the 5-DOF arm. While there are 8 possible solutions, not all of them are always valid, and in fact we were not able to find more than 4 valid solutions at any one time

However, not all 8 solutions are valid. They may either exceed joint limits or may not have the correct end effector positions and orientation. In order to determine which solutions are valid for any given target end effector position and orientation, the calculated joint angle must be checked against the joint limits for each joint, and the end effector position generated by the joint values must be checked against the target.

Jacobian and Inverse Jacobian Derivation

To construct our Jacobian, we first created our DH table, using the same process from Mini-Project 1.

Joint	θ	d	a	α
1	θ_1	l_1	0	$\pi/2$
1.5	$\pi/2$	0	0	0
2	θ_2	0	l_2	π
3	θ_3	0	l_3	π
4	θ_4	0	l_4	0
4.5	$-\pi/2$	0	0	$-\pi/2$
5	θ_5	l_5	0	0

Fig. 9
The DH Table for the 5-DOF arm from MP1

Then using the following formula, we calculated the homogeneous transformation matrices between each frame.

$$H_i = \begin{bmatrix} C\theta_i & -S\theta_i C\alpha_i & S\theta_i S\alpha_i & a_i C\theta_i \\ S\theta_i & C\theta_i C\alpha_i & -C\theta_i S\alpha_i & a_i S\theta_i \\ 0 & S\alpha_i & C\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The following are each of our transformation matrices.

$${}^{0}H_{1} = \begin{bmatrix} \cos(\theta_1) & 0 & \sin(\theta_1) & 0 \\ \sin(\theta_1) & 0 & -\cos(\theta_1) & 0 \\ 0 & 1 & 0 & l_1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$
{}^{1}H_{1.5} = \begin{bmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

$$
{}^{1.5}H_{2} = \begin{bmatrix} \cos(\theta_2) & \sin(\theta_2) & 0 & l_2 \cos(\theta_2) \\ \sin(\theta_2) & -\cos(\theta_2) & 0 & l_2 \sin(\theta_2) \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

$$
{}^{2}H_{3} = \begin{bmatrix} \cos(\theta_3) & \sin(\theta_3) & 0 & l_3 \cos(\theta_3) \\ \sin(\theta_3) & -\cos(\theta_3) & 0 & l_3 \sin(\theta_3) \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

$$
{}^{3}H_{4} = \begin{bmatrix} \cos(\theta_4) & -\sin(\theta_4) & 0 & l_4 \cos(\theta_4) \\ \sin(\theta_4) & \cos(\theta_4) & 0 & l_4 \sin(\theta_4) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

$$
{}^{4}H_{4.5} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

$$
{}^{4.5}H_{5} = \begin{bmatrix} \cos(\theta_5) & -\sin(\theta_5) & 0 & 0 \\ \sin(\theta_5) & \cos(\theta_5) & 0 & 0 \\ 0 & 0 & 1 & l_5 \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

We then calculated the total transformation matrix by multiplying the matrices together as follows:

$$^{0}H_{5} = {}^{0}H_{1} * {}^{1}H_{1.5} * {}^{1.5}H_{2} * {}^{2}H_{3} * {}^{3}H_{4} * {}^{4}H_{4.5} * {}^{4.5}H_{5}$$

(In Python implementation, we use the given helper function *dh_to_matrix* to convert each row of our DH table to the given H matrices.)

This homogenous matrix allows us to find the end effector position given the joint angles of the robot, a necessary step of the numerical solution for inverse kinematics.

We then need to find the Jacobian matrix of the robot. To do this, we want to determine the effect of each joint on the end effector position, so we took the translational components of the total homogeneous transformation matrix as a vector $\begin{bmatrix} x \\ y \\ z \end{bmatrix}$ then took the partial derivative of each component with respect to each angle. We then created our Jacobian as follows:

$$J_v = \begin{bmatrix} dx/d\theta_1 & dx/d\theta_2 & dx/d\theta_3 & dx/d\theta_4 & dx/d\theta_5 \\ dy/d\theta_1 & dy/d\theta_2 & dy/d\theta_3 & dy/d\theta_4 & dy/d\theta_5 \\ dz/d\theta_1 & dz/d\theta_2 & dz/d\theta_3 & dz/d\theta_4 & dz/d\theta_1 \end{bmatrix}$$

Since the matrix is non-square, we cannot take the actual inverse of this matrix, so we must instead take the pseudoinverse. We did this using the following formula, which includes a small offset to avoid singularities.

$$\mathrm{J}_v^+ = (J_v^T J_v + \lambda \mathrm{I})^{-1} J_v^T$$

Numerical Inverse Kinematics Derivation

The method for finding the numerical solution to the inverse kinematic problem is to use an iterative solver. In each iteration, it calculates an end effector pose (the Newton-Raphson method doesn't solve for orientation) and checks if the error is within a specified tolerance. If it is, a solution has been reached, but if it has not, the solver adjusts the joint angle values in the direction of the correct end effector position and iterates again. To begin, the solver needs an initial guess (Θ_i) and target position (x_d) with which it calculates the error.

$$e = x_d - \theta_i$$

If the error is less than the tolerance, it will then calculate a new guess using the inverse jacobian derived above.

$$\theta_{i+1} = \theta_i + J^{\dagger} e$$

This calculation results in joint angles that can be used to recalculate the end effector position, allowing the solver to continue iterating using this same process. However, we also limit the total number of iterations so that if the position is unreachable, the solver will stop, instead of iterating indefinitely.

Conclusion

Overall, we are very happy with the outcome of the project. We believe that we successfully analyzed some key distinctions between different trajectory generation methods, and when and why we would use them. We faced a lot of challenges on the way, especially with integrating our trajectories with the existing visualizer tool.

If we had more time, we would definitely dive deeper into more specialized trajectory generation methods like sequential quadratic programming or RRT (rapidly exploring random trees). We would also like to have analyzed deeper into the ‘smoothness’ of motion for each of the methods, but we found this difficult to do quantitatively, and were reliant on qualitative examining.