Performance Comparison of Different Sampling, Smoothing and Path Planning Algorithms for a 2 Link 4 DOF Robotic Manipulator

CS 5335: Robotics Science and Systems, Professor Robert Platt

Dev Vaibhav
Dept. Electrical and Computer Eng.
Northeastern University
vaibhav.d@northeastern.edu

Siddharth Maheshwari
Dept. Electrical and Computer Eng.
Northeastern University
maheshwari.si@northeastern.edu

I. Abstract

This paper compares the performance of different sampling algorithms with various path planning algorithms and smoothing methods for a two-link 4-DOF robotic manipulator in a 3D environment. The goal of the study is to identify the most effective combination of algorithms for optimizing motion planning. The sampling algorithms used were uniform random sampling, Gaussian sampling, and Bridge sampling. The path planning algorithms evaluated were Dijkstra, RRT, PRM, A*, and RRT*, while the path smoothing algorithms included Polynomial Interpolation, Bézier Curve, Cubic Splines, B-Spline, and Piecewise Cubic Hermite Interpolating Polynomial (PCHIP) interpolations. The results show the takeaway in different scenarios (static or dynamic environment). The findings of this research could be useful in improving the efficiency and accuracy of robotic motion planning in real-world applications.

II. SIMULATOR:

We have used the two-link 4-DOF robotic arm provided in HW3 as the base and developed our code by making modifications to it in MATLAB. Past experience with the software and environment helped us a lot in overcoming the initial library setup/ and speeding up our learning curve. We focused directly on the sampling and smoothing algorithms [1] which we plan to implement and compare.

The robot arm is constrained by the joint limits as given below

$$q_{min} = \begin{bmatrix} -\pi/2 & -\pi & 0 & -\pi \end{bmatrix}$$

$$q_{max} = \begin{bmatrix} \pi/2 & 0 & 0 & 0 \end{bmatrix}$$

It can be noticed that the third joint (q_3) remains fixed as both its minimum and maximum value are equal. Because of this simplification, we were able to view the 4D configuration space as 3D. Visualization in figure 2 helped a lot in getting a better understanding of sampling-based motion planning algorithms.

We also changed the locations of the obstacles from the original code. Fig 1. is a visualization of the robot setup along with three spherical obstacles.

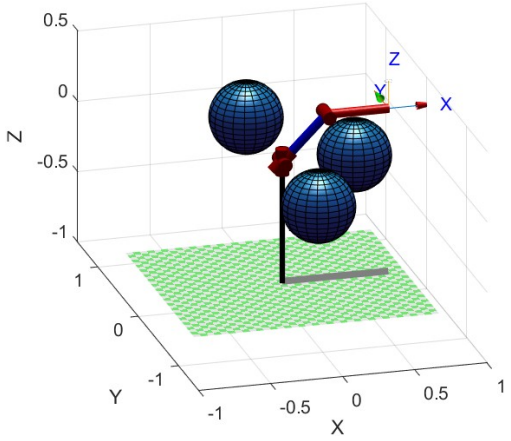

Fig. 1: 2 link 4-DOF robot arm with spherical obstacles

III. Sampling Algorithms:

Robotic systems need motion planning to navigate complex and dynamic environments. The selection of a sampling algorithm is a crucial step in motion planning since it has a significant impact on the effectiveness and precision of the process. There isn't a single sampling technique that performs well in every situation because

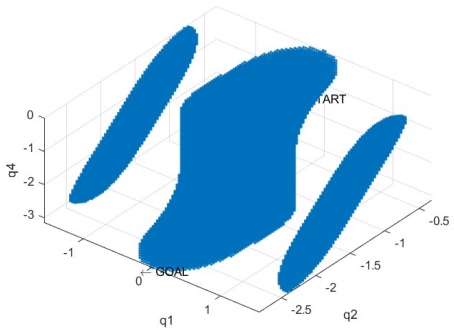

Fig. 2: Configuration Space

each one has advantages and disadvantages of its own. To choose the best strategy for a given task, it is critical to assess and compare various sampling algorithms.

A. Uniform Sampling

Uniform sampling is a method of generating random samples from a given distribution such that each sample has an equal probability of being selected. In other words, it involves selecting a random value from a range of possible values, with each value having an equal chance of being selected.

For example, if we want to generate a random number between 1 and 10 with uniform sampling, we would assign an equal probability to each number between 1 and 10. This means that each number has a 1/10 or 0.1 probability of being selected.

Uniform sampling can be useful in a variety of applications, such as simulation, optimization, and statistical analysis. It can help to generate random data that follows a particular distribution, which can be used to test hypotheses, estimate parameters, and make predictions. It can also be used to generate random inputs for simulations or optimization algorithms, which can help to explore a range of possible outcomes and identify optimal solutions.

Overall, uniform sampling is a simple and widely used method for generating random samples, and it forms the basis for many more advanced techniques in statistics and machine learning.

MATLAB's function `rand` is used to generate random samples between a range. Since the generated samples need to be within the joint limits, method provided in [2] is used.

B. Gaussian Sampling

Uniform sampling is not a good way to find paths through narrow passageways. To overcome this limitation, Gaussian Sampling is used.

Gaussian sampling, also known as normal distribution sampling, is a process of generating random numbers from a Gaussian distribution. The Gaussian distribution, also known as the normal distribution, is a probability distribution that is widely used in statistics to describe the variation of a random variable.

A Gaussian distribution is defined by the equation 1 and looks like Fig. 3

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} \tag{1}$$

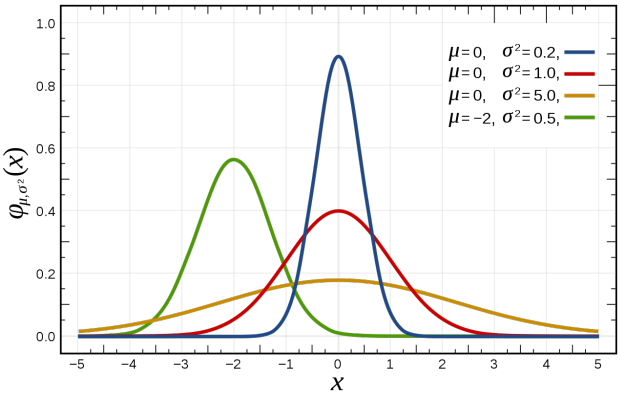

Fig. 3: Normal Distribution curve

Steps to generate a Gaussian sample:

1) Sample points uniformly at random (as before)
2) For each sampled point, sample a second point from a Gaussian distribution centered at the first sampled point with some standard deviation
3) Discard both samples if both samples are either free or in collision
4) Keep the free sample if the two samples are not both free or both in a collision (that is, keep the sample if the free/collision status of the second sample is different from the first)

The process is explained mathematically in Fig 4

Gaussian: $q_1 \sim \mathcal{U}$; $q_2 \sim \mathcal{N}(q_1, \sigma)$; if $q_1 \in Q_{\text{free}}$ and $q_2 \notin Q_{\text{free}}$, add q_1 (or vice versa).

Fig. 4: Gaussian Sampling summarized mathematically

A standard deviation ($\sigma = 0.75 * (q_{max} - q_{min})$) is chosen for this implementation.

MATLAB's function `normrnd` is used to generate the gaussian samples.

C. Bridge Sampling

Bridge sampling works by introducing a new "bridge" distribution that connects the two probability distributions of interest. Here, the bridge distribution is a combination of the two distributions: uniform and normal.

Steps to generate a Bridge sample:

1) Sample a point uniformly at random (as before)

2) For each sampled point, sample a second point from a Gaussian distribution centered at the first sampled point with some standard deviation
3) Define a third sample which is the average of these two samples
4) Keep the third sample if the two samples are not both free or both in a collision and the third sample is collision-free

Fig 5 describes these steps mathematically.

Bridge: $q_1 \sim \mathcal{U}$; $q_2 \sim \mathcal{N}(q_1, \sigma)$; $q_3 = (q_1 + q_2)/2$; if $q_1, q_2 \notin Q_{\text{free}}$ and $q_3 \in Q_{\text{free}}$, add q_3.

Fig. 5: Bridge Sampling summarized mathematically

It is observed that upon increasing the standard deviation, sampling becomes faster but it still takes a lot of time to generate a bridge sampling. More discussion about this is in the Results section. So, we don't think it can be applied to real-time systems unless we already have a pool of these samples computed offline.

IV. Path Planning Algorithms:

Path planning is an essential task in robotics that involves finding an optimal path for a robot to navigate from its starting point to a desired goal location. In complex environments, finding an optimal path can be challenging, and researchers have proposed various path-planning algorithms to address this issue.

Each path-planning algorithm has its strengths and weakness. Some algorithms are designed to be more efficient, while others prioritize finding the shortest path or avoiding obstacles. Evaluating and comparing different path-planning algorithms can help identify the most effective approach for a particular task and environment.

Our research involves implementing and comparing various path planning algorithms to identify which approach works best for our two-link 4 DOF robotic manipulator in a 3D environment. By examining the strengths and weaknesses of different algorithms, we hope to provide insights that will contribute to the development of more effective robotic systems.

A. Dijkstra's algorithm

Dijkstra's algorithm is a popular algorithm used in computer science to find the shortest path between two nodes in a weighted graph. It is named after Dutch computer scientist Edsger Dijkstra and is often used in routing and as a subroutine in other graph algorithms.

The algorithm works by maintaining a priority queue of vertices to be explored. Initially, only the starting vertex is in the queue, with a distance of zero. The algorithm then repeatedly extracts the vertex with the smallest distance from the priority queue and relaxes all of its neighboring vertices, updating their distances if a shorter path is found. The algorithm terminates when the destination vertex is extracted from the queue.

The algorithm maintains two data structures: a set of visited vertices and a set of tentative distances. The visited set contains all vertices that have been fully explored, while the tentative distance set contains the shortest distance to each vertex found so far. Fig. 6 shows the Dijkstra's algorithm pseudo code.

```
samples_new = samples
adjacency_new = adjacency
Add q_start and q_goal to samples_new
For i = size(samples_new,1)-1:size(samples_new,1)
    For j = 1:size(samples_new,1)
        If j != i
            Calculate the distance between sample i and sample j and store it in dist(j,1)
            Store the index of sample j in dist(j,2)
        End If
    End For
    Remove the zero entry for distance when i = j
    Sort dist by distance values in ascending order
    For j = 1:length(sorted_dist)
        If the edge between the jth neighbor and the current sample i is collision free
            Add the distance value as the weight for the edge between i and j in adjacency_new
        Else
            Set the weight for the edge between i and j to zero in adjacency_new
        End If
    End For
End For
Set the weight for the diagonal entries in adjacency_new to zero
Compute the shortest path from the second last sample to the last sample using Dijkstra's algorithm
and store the distances and the indexes of the samples in path_indexes
If the number of elements in path_indexes is greater than 1
    Set path_found to 1
    Initialize an empty array called path
    For i = 1:length(path_indexes)
        Add the coordinates of the sample with index path_indexes(i) to the ith row of path
    End For
End If
```

Fig. 6: DIJKSTRA pseudo code

B. Probabilistic RoadMap (PRM):

The probabilistic roadmap planner is a motion planning algorithm in robotics, which solves the problem of determining a path between a starting configuration of the robot and a goal configuration while avoiding collisions. An example of a probabilistic road map algorithm explores feasible paths around a number of polygonal obstacles. In our case, we have three spherical obstacles.

The basic idea behind PRM is to take random samples from the configuration space of the robot, test them for whether they are in the free space, and use a local planner to attempt to connect these configurations to other nearby configurations. The starting and goal configurations are added in, and a graph search algorithm is applied to the resulting graph to determine a path between the starting and goal configurations. Fig 7 is the pseudo-code which we have used for the implementation.

Input: number n of samples, number k number of nearest neighbors
Output: PRM $G = (V, E)$

initialize $V = \emptyset$, $E = \emptyset$
while $|V| < n$ **do** *// find n collision free points q_i*
  $q \leftarrow$ random sample from Q
  if $q \in Q_{\text{free}}$ **then** $V \leftarrow V \cup \{q\}$
end while
for all $q \in V$ **do** *// check if near points can be connected*
  $N_q \leftarrow k$ nearest neighbors of q in V
  for all $q' \in N_q$ **do**
    if path$(q, q') \in Q_{\text{free}}$ **then** $E \leftarrow E \cup \{(q, q')\}$
  end for
end for

Fig. 7: PRM pseudo code

There are however, some problems associated with PRM which are solved by RRT:

- two steps: graph construction, then graph search
- hard to apply to problems where edges are directed, i.e. kinodynamic problems

C. Rapidly Exploring Random Tree (RRT)

It's a sampling-based approach that explores the configuration space of a robot to find a collision-free path from a starting configuration to a goal configuration. The basic idea of the RRT algorithm is to grow a tree of connected configurations, where each configuration represents a valid state of the robot. The tree is grown randomly, with new configurations being added to the tree in a way that balances the exploration of unexplored areas with progress towards the goal configuration (biasing).

Once the RRT algorithm has constructed the tree, it searches for a path from the start configuration to the goal configuration. This path is found by traversing the tree from the goal configuration back to the start configuration. The algorithm has been shown to be effective in many different robotic applications, such as motion planning for autonomous vehicles, manipulator planning for industrial robots, and path planning for aerial vehicles.

RRT solves the problems faced in PRM by:

- creating a tree instead of a graph. So, no graph search needed!
- making a tree rooted at start or goal. So, edges can be directed.

Fig 8 shows the implemented pseudo code

Input: q_{start}, q_{goal}, number n of nodes, stepsize α, β
Output: tree $T = (V, E)$

initialize $V = \{q_{\text{start}}\}$, $E = \emptyset$
for $i = 0 : n$ **do**
  if rand$(0, 1) < \beta$ **then** $q_{\text{target}} \leftarrow q_{\text{goal}}$
  else $q_{\text{target}} \leftarrow$ random sample from Q
  $q_{\text{near}} \leftarrow$ nearest neighbor of q_{target} in V
  $q_{\text{new}} \leftarrow q_{\text{near}} + \frac{\alpha}{|q_{\text{target}} - q_{\text{near}}|}(q_{\text{target}} - q_{\text{near}})$
  if $q_{\text{new}} \in Q_{\text{free}}$ **then** $V \leftarrow V \cup \{q_{\text{new}}\}$, $E \leftarrow E \cup \{(q_{\text{near}}, q_{\text{new}})\}$
end for

Fig. 8: RRT pseudo code

Chosen step size ($\alpha = 0.2$), Goal biasing parameter ($\beta = 0.1$), and threshold distance (0.3) within which if q_{new} comes closer to q_{goal} and edge between them is collision-free, goal node and an edge between q_{new} and q_{goal} is added to the tree.

D. Rapidly-exploring Random Tree Star (RRT*):

RRT* (Rapidly-exploring Random Tree Star) is an extension of the Rapidly-exploring Random Tree (RRT). It aims to efficiently explore the state space of a robotic system by constructing a tree-like structure. It works by iteratively adding new nodes to the tree, with the goal of eventually finding a path from the initial state to the goal state. The algorithm uses a heuristic to bias the tree expansion towards the goal region, making it more likely to find a solution quickly.

The "Star" in RRT* refers to the way the algorithm optimizes the path found by the tree. RRT* uses a cost function to determine the quality of the path, and it iteratively rewires the tree to find a lower-cost path. This approach ensures that the algorithm finds an optimal solution while exploring the state space efficiently.

It has been used in various applications, including robotic manipulation, autonomous navigation, and UAV flight planning. It is known for its scalability and robustness in high-dimensional state spaces.

Initially, we tried to find the radius using the formula in the pseudo-code but it did not work out as expected. So, we hard-coded the search radius to `0.5` after observing some iterations and distance between q_{new} and other nodes.

Fig 9 shows the implemented pseudo code. A little tweak similar to RRT is implemented here to bias the tree growth toward the goal. Fig 10 shows the nodes in the tree generated by RRT*.

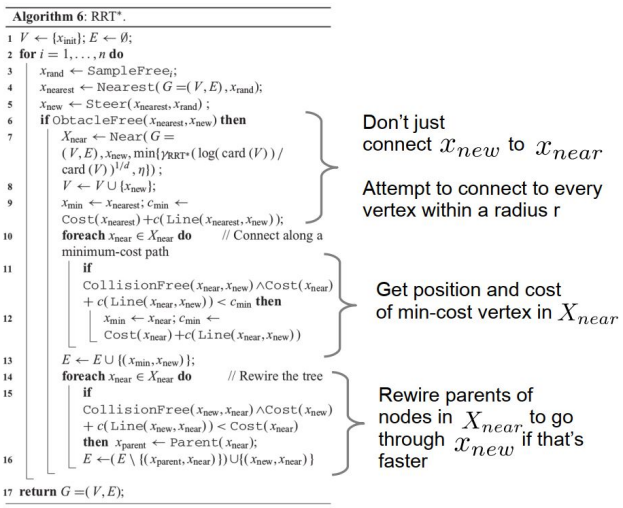

Fig. 9: RRT* pseudo code

E. A-star (A*)

In our research, we choose to implement the A* algorithm due to its popularity and effectiveness in finding optimal paths in complex environments. A* uses heuristics to estimate the distance to the goal from each potential path and selects the path with the lowest estimated cost, making it an efficient and widely used approach for path planning.

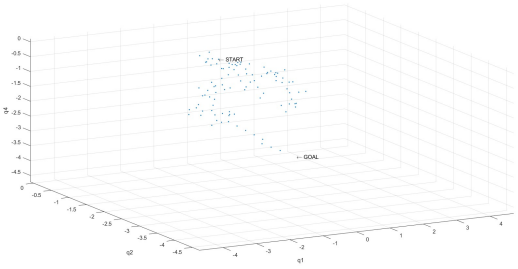

Fig. 10: RRT* tree growth (only nodes shown)

The A* algorithm also allows for tuning of the heuristic function to optimize for different criteria, such as minimizing travel time or avoiding obstacles. Furthermore, its optimality and completeness make it a reliable choice for robotic motion planning.

Therefore, we selected the A* algorithm as one of the path planning algorithms to evaluate its effectiveness for our two-link 4 DOF robotic manipulator in a 3D environment. By comparing A* with other path-planning algorithms, we aim to provide insights into its performance and limitations, which may contribute to the development of more effective path-planning strategies for robotics.

Fig. 11 shows the implemented pseudo-code.

```
function [path, path_found] = A_star(adj_matrix,samples, start_idx, goal_idx, heuristic_func)
    Initialize empty open_list and closed_list
    Initialize g_scores and f_scores arrays with inf for all nodes
    Set g_score of start_idx to 0 and f_score of start_idx to heuristic_func(start_idx, goal_idx)
    Set parent_nodes of start_idx to start_idx

    Add start_idx to the open_list

    while open_list is not empty:
        Get the node with the lowest f_score from the open_list (current_node)

        If current_node is goal_idx:
            Reconstruct the path from start_idx to goal_idx using the parent_nodes array
            Set path_found to true
            Return the path and path_found

        Remove current_node from the open_list and add it to the closed_list

        For each neighbor of current_node:
            If neighbor is in the closed_list, skip to the next neighbor

            Calculate the tentative_g_score as the g_score of current_node plus the distance between current_node and neighbor

            If neighbor is not in the open_list, add it to the open_list and calculate its h_score using the heuristic_func

            If tentative_g_score is greater than or equal to the g_score of neighbor, skip to the next neighbor

            Update the g_score, f_score, and parent_node of neighbor with the tentative values

        End For
    End while

    If the goal is not found, return an empty path and path_found is false
End function
```

Fig. 11: A* pseudo code

V. PATH SMOOTHING ALGORITHMS:

After the motion planning algorithms generate a path, we shorten it using the technique in HW3 which results in a lot fewer waypoints. This is then fed to the interpolation algorithms. We used the path generated by RRT* with uniform sampling because this had most zig-zags and was fast to compute.

A. Linear Interpolation:

Linear interpolation is a method of estimating a value between two known values on a straight line. In other words, it is a way to find an intermediate value between two points on a line. This technique is commonly used in various fields such as engineering, physics, and computer graphics.

The process of linear interpolation involves finding the equation of the line that passes through two given points, and then using this equation to estimate the value of an unknown point between the two known points. The formula for linear interpolation is given by equation 2:

$$y = y_1 + \frac{(y_2 - y_1)}{(x_2 - x_1)} * (x - x_1) \tag{2}$$

where:
y is the estimated value
y_1 and y_2 are the y-coordinates of the two known points
x_1 and x_2 are the x-coordinates of the two known points
x is the x-coordinate of the unknown point

To use linear interpolation, you need to know the coordinates of two points on a line and the value of one of the coordinates at an intermediate point. For example, if you know the temperature at two different times and want to estimate the temperature at a specific time between those two times, you can use linear interpolation.

B. Polynomial Interpolation:

Polynomial interpolation is a commonly used technique for path smoothing. We used this technique to smoothen the path of a robot navigating through a complex environment. For each joint of the robot, we fit a polynomial curve (in a least-squares sense) to the original path using the `polyfit` and `polyval` functions in MATLAB. The resulting smoothed path closely follows the original path and is stored in a matrix. This technique allows for a smoother and more continuous path that reduces the risk of collisions with obstacles and improves overall performance.

We have used (n-1) degree polynomial, where n is the total number of waypoints.

In our code, we first define the time parameter t and the interpolation time parameter ti. We then use a loop to perform polynomial interpolation separately for each joint of the robot.

C. Bézier Curve:

These parametric curves make use of 'control points' to define the shape. At their core, they make use of Bernstein polynomial functions. Given a set of $n + 1$ control points P_0, P_1, ..., P_n, the corresponding Bézier curve (or Bernstein-Bézier curve) is given by equation 3

$$C(t) = \sum_{i=0}^{n} P_i B_{i,n}(t) \tag{3}$$

where, $B_{i,n}(t)$ is a Bernstein polynomial and $t \in [0, 1]$. A Bernstein polynomial of degree n is defined by 4

$$B_{i,n}(t) = \binom{n}{i} t^i (1-t)^{n-i} \tag{4}$$

where,

$$\binom{n}{i} = \frac{n!}{i!(n-i)!} \tag{5}$$

The algorithm's ability to generate smooth and continuous paths with minimal changes in direction or speed makes it an ideal choice for creating efficient and safe navigation strategies for robots. The process involves defining the number of control points, computing the Bezier coefficients, defining spacing between control points, and generating equally spaced points to compute joint angles using the Bezier curve formula. By applying this formula to each joint separately, a smooth path is generated, which is then stored in a matrix.

D. Cubic Splines:

To interpolate data points and produce a continuous, smooth curve, mathematicians use spline interpolation. In interpolating problems, spline interpolation is often preferred to polynomial interpolation because it yields similar results, even when using low degree polynomials, while avoiding Runge's phenomenon for higher degrees. The speciality about cubic splines is that it uses a third order polynomial and when the data points are connected using this curve, it produces a continuous and differentiable curve. Since the resulting curve exhibits C^2 continuity, its first and second derivatives are also continuous. This characteristic guarantees the curve's smoothness and natural-looking shape, which makes it a prime option for path smoothing.

We first generate a path with the help of a path planning algorithm. It is then shortened and interpolated using spline interpolation to produce a continuous, smooth curve. The spline interpolation is carried out in MATLAB using the `interp1` function. The initial path and the appropriate interpolation interval are specified. Robot motion planning is then done using the resulting smoothed path.

By utilizing spline interpolation for path smoothing, we may reduce the effect of data noise and provide our robotic systems with reliable and precise motion planning.

E. B-Spline or Basis Spline :

In order to interpolate a series of waypoints, the B-spline path smoothing technique builds a control polygon based on those waypoints. It is a generalization of the Bézier curve. Interior points are calculated using the control polygon, and a polynomial is built using MATLAB's `bsplinepolytraj` function based on these points. The polynomial is assessed at a set of evenly spaced locations between 0 and 1 to get the smooth path.

The control polygon generation algorithm begins by dividing the original path into four sets of waypoints. These waypoints are used to create a control polygon, which is a polygonal approximation of the original path. If the original path has only one segment, the algorithm returns it as the smoothed path. Otherwise, the algorithm creates matrices to calculate the interior points of the control polygon. It also adjusts the endpoints of the control polygon to ensure a smooth transition between segments. The resulting smoothed path approximates the original path while minimizing abrupt changes in direction.

F. Piecewise Cubic Hermite Interpolating Polynomial (PCHIP):

Another interpolation method is `pchip` which stands for Piecewise Cubic Hermite Interpolating Polynomial.

To implement this method in MATLAB, we first create a time vector t that spans the length of the original path. We then define a new time vector ti with smaller intervals, which determines the resolution of the interpolation. For example, we use 0.01 as the interval size, but this value can be adjusted to get smoother or coarser paths.

Finally, we use the `interp1` function to compute the smoothed path by interpolating the original path at the new time vector ti. The `'pchip'` option specifies the type of interpolation to use. The resulting smoothed path will have the same number of dimensions as the original path and will be defined at the new time vector ti.

VI. RESULTS :

A. Path Planning Algorithm Results and Analysis :

The sampling algorithm used for path planning can significantly impact the computational time required. Our experiments showed that uniform sampling with Dijkstra or A* provides the fastest results for online computation, while bridge or Gaussian sampling can be used for offline calculations. It is worth noting that bridge sampling was very slow, taking almost 16.5 minutes to find 100 samples.While, uniform sampling is the fastest, taking only 22.73 seconds to find 100 samples. On the other hand, Gaussian/Normal Sampling takes 65.00 seconds to perform the same task, with the runtime decreasing as standard deviation is increased. Overall, the choice of sampling algorithm will depend on the specific requirements of the task at hand.

Regarding the planning algorithm, we found that Dijkstra is the fastest option and provides the shortest path. A* is even faster, but its path length is slightly longer than that of Dijkstra. On the other hand, RRT provides a fast path planning solution but has a longer path length

than Dijkstra and A*. Therefore, the choice of a suitable planning algorithm should depend on the specific application requirements, such as the desired speed and path length. Therefore, our study provides a comprehensive analysis of various sampling and planning algorithms for path planning in a static environment. Our findings can guide the selection of appropriate algorithms for different robotic applications, considering factors such as the environment, computation time, and accuracy.

	Uniform Sampling		Gaussian/ Normal Sampling		Bridge Sampling	
Path Planning Algorithm	Runtime	Path Length	Runtime	Path Length	Runtime	Path Length
PRM	3.546567	3	9.434046	5	10.26381	3
RRT	1.015241	[22,28]	7.841516	[23,29]	CANCELLED	
Dijkstra	0.003367	3	0.00317	3	0.004106	3
RRT*	7.192849	[22,28]	26.55914	[21,25]	CANCELLED	
A*	0.002449	7	0.00256	8	0.002878	6

Fig. 12: Motion Planning Algorithm results comparison

Regarding Table 12, we captured data for 10 iterations for each combination: 5*3*10 = 150 iterations total. For Runtime, mean result is reported.

[22,28] mean that the path length was within that particular range for the 10 iterations.

Here in Table 12, **CANCELLED** denotes that we canceled those experiments as it took a lot of time to find samples (waited easily > 5 mins)

Key Takeaways:

- Use Bridge/ Gaussian Sampling for offline calculation while Uniform for online.
- Dijkstra is fast and provides shortest path.
- A* is fastest but path length is a bit more.
- RRT is fast but path length is more.

B. Path Smoothing / Interpolation Algorithms Results and Analysis:

For path smoothing all the figures from Fig 13 to Fig 17 shows the path in reduced dimension configuration space for the same RRT* path and shortened path but different smoothing algorithms applied on them. Here is how various figures show different algorithms:

-Fig 13 shows Polynomial Interpolation for (n-1) degrees of polynomial where n is the waypoints in the shortened path (4 here).

-Fig 14 shows the Bézier Curve for the given shortened RRT* path.

-Fig 15 shows the Cubic Spline on the shortened RRT* path.

-Fig 16 shows the Basis Spline on the shortened RRT* path.

-Fig 17 shows the Piecewise Cubic Hermite Interpolating Polynomial (PCHIP) on the shortened RRT* path.

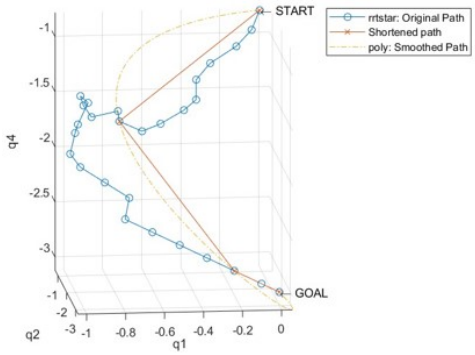

Fig. 13: Polynomal Interpolation

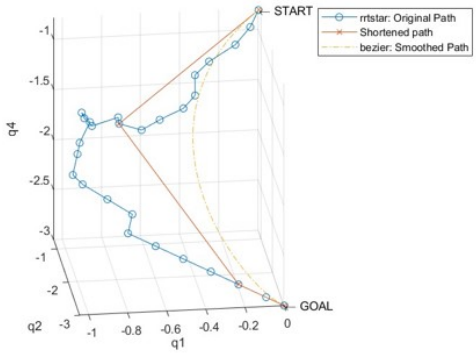

Fig. 14: Bezier Curve Interpolation

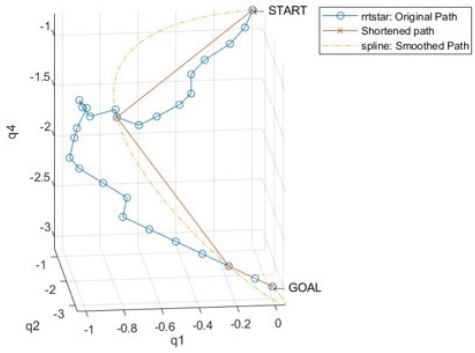

Fig. 15: Cubic Spline Interpolation

To compare the performance of different interpolation algorithms shown in Table 18, we used the RRT* path generated from uniform sampling, which had the most zig-zags. The original path length from RRT* with uniform sampling was 28. To make the path smoother, we used a path shortening algorithm which reduced the path length to 4. This shortened path was then fed to the interpolation algorithms for further smoothing and to produce a more natural-looking and continuous path.

Key Takeaways:

- Linear Interpolation is the most efficient algorithm

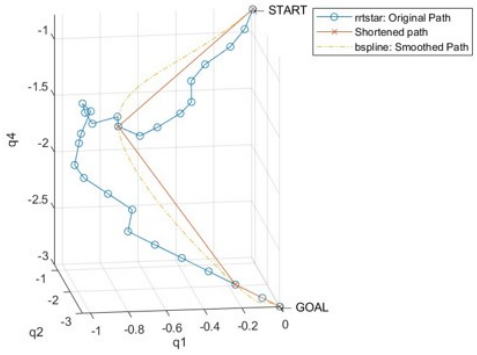

Fig. 16: B-Spline Interpolation

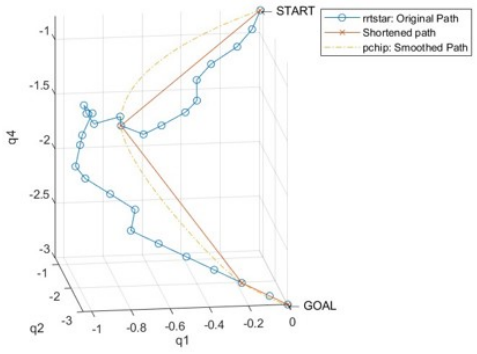

Fig. 17: PCHIP Interpolation

Path Smoothing/ Interpolation Algorithm	Run Time	% Points in collision
Linear Interpolation	0.000178	0
Polynomial Interpolation	0.007155	0
Spline	0.028115	0
Bezier Curve	0.009217	45.54
B-Spline	0.138516	0
Piecewise Cubic Hermite Interpolating Polynomial (PCHIP)	0.008928	0

Fig. 18: Path Smoothing/ Interpolation Algorithm results comparison

for path smoothing in terms of runtime.

- Bezier Curves tend to have around 45% collision with obstacles after path smoothing.
- Apart from Linear Interpolation all other algorithms, it is recommended to check for obstacles in the new path as it may have intersecting points with obstacles.

Looking at the different interpolation methods we can say the following::

- **Linear Interpolation:** Requires at least 2 points.
- **Polynomial Interpolation:** Higher degree polynomial required if the waypoints increase and are in a very zig-zag fashion which is computationally more expensive.
- **Spline:** Requires at least 4 points. Requires more memory and computation time than 'pchip' but the path is C^2 continuous.
- **Bezier Curve:** The curve passes through a convex hull of control points. Hence, does not pass through the waypoints.
- **B-Spline:** As seen from the Fig 16, closer to the shortened path but more computationally expensive.
- **Piecewise Cubic Hermite Interpolating Polynomial (PCHIP):** Requires at least 4 points. Requires more memory and computation time than 'linear'.

VII. Challenges

During the implementation of the different algorithms, we faced various challenges that needed to be addressed. One such challenge was that the RRT* tree was expanding away from the goal. After careful analysis, it was discovered that one of the function's arguments was being overwritten inside the function, resulting in the RRT* tree's expansion away from the goal. This challenge was overcome by fixing the duplicate variable name issue, which helped to ensure the correct execution of the RRT* algorithm.

Another challenge that we faced during the implementation was finding appropriate functions for different types of interpolation. We found that some interpolation methods were not readily available in the standard libraries, which required us to create our own interpolation functions. We tackled this challenge by researching different types of interpolation methods and going through various tutorials. Eventually, we were able to find suitable functions or create our own functions for all the required interpolation methods. Overall, these challenges provided valuable learning experiences that helped us improve our programming skills and problem-solving abilities.

VIII. Future Improvement Areas

- **Trying all combinations which are left**: One potential future improvement for our path planning algorithm is to try all combinations of sampling, interpolation, and smoothing methods that we did not test in our experiments. This could potentially lead to discovering a more optimal combination that performs better than the ones which we already tested. However, this approach could be computationally expensive and time-consuming, especially if the number of combinations is large.
- **Using much more sophisticated obstacles**: Another future improvement for our path planning algorithm is to use much more sophisticated obstacles, such as ones that move over time or have

complex shapes. This could make the path planning problem more challenging and realistic. However, this would require more advanced sensors and perception algorithms to detect and track these obstacles, as well as more complex and computationally expensive path-planning algorithms to navigate around them.

IX. Conclusions

In conclusion, we have successfully implemented all the planned path planning and smoothing algorithms. For online computation in static environments, we recommend using uniform sampling with Dijkstra or A*, while for dynamic environments, uniform sampling with RRT or RRT* is preferred due to their efficiency. When it comes to path smoothing and interpolation, we have found that Spline is an excellent option, as it produces a C^2 continuous curve, resulting in a smooth and natural-looking path. However, it requires at least four points to function properly. Overall, our implementation of these algorithms has proven to be effective and efficient for path planning and smoothing in various scenarios.

References

[1] Ravankar A, Ravankar AA, Kobayashi Y, Hoshino Y, Peng CC. Path Smoothing Techniques in Robot Navigation: State-of-the-Art, Current and Future Challenges. Sensors (Basel). 2018 Sep 19;18(9):3170. doi: 10.3390/s18093170. PMID: 30235894; PMCID: PMC6165411.

[2] https://www.mathworks.com/help/matlab/ref/rand.html#buiavoq-9

[3] https://en.wikipedia.org/wiki/Normal_distribution#/media/File:Normal_Distribution_PDF.svg