Two Dimensional Reach-Avoid-Stay Path Planning using RRT in Python

Sanghoon Choi and Paulina Bakos Lang

***Abstract*— This paper explores a modified implementation of the Rapidly exploring Random Tree (RRT) algorithm to solve the Reach-Avoid-Stay problem for a drone operating in a two-dimensional grid environment. The goal is to dynamically generate a feasible path from a given start point to a target zone while avoiding obstacle zones, under velocity and acceleration constraints. Many path finding algorithms were explored, but RRT was particularly good for this problem as it is scalable to maze configurations and has a comparatively low computational cost. Additional key optimizations were introduced: (1) a goal bias constant to guide the exploration process towards the target zone, and (2) a proportional gain factor KP helping to adjust control inputs based on the distance to the sampled node. The proposed approach significantly improved path efficiency and reduced the computational time.**

I. Introduction

Path and motion planning are prevalent topics nowadays. They aim to direct a system from an initial state to a target state while avoiding any unsafe zones. From navigating a warehouse to self-driving cars, these tasks have many applications where avoiding unsafe zones can be critical. Each of these tasks can be divided into three parts: Reach, Avoid, and Stay.

Reach and Avoid

The objective of the "Reach" component is to guide the system from a given initial state to a target. There exists an abundance of algorithms (called path planning algorithms) for this purpose. Many of them are extensively described in [1] and [2], and nicely summarized in [3].

The primary goal of the "Avoid" component is to ensure that the system remains clear of any obstacles or unsafe zones that could be obstructing the trajectory to the target. There are many ways to avoid obstacles, well described in [4] and [5], and they usually go hand in hand with the "Reach" component. Obstacle avoidance is especially important in dynamic spaces (instead of static ones) where obstacles are not known beforehand or their locations can update through time.

Reach-Avoid algorithms can be divided into different categories based on their approach style. Notably, these are Graph Search, Heuristic Search, Local-Avoidance Based, AI, Sampling Based, and Planner-Based. Further down are a few algorithms worth mentioning.

Also note that these algorithms are suited for different types of systems. We can divide systems as follows [6]:

Discrete/Continuous. Discrete systems evolve according to a discrete time step and are common in signal processing. Continuous systems evolve smoothly with time such as drones or robots displacing smoothly.

Linear/Nonlinear. Linear systems are simple systems that follow a linear model such as a simple mass and spring relation. Nonlinear systems are more complex and better reflect real-life interactions.

Time-Variant/Time-Invariant. For time-variant systems, parameters change over time such as a robot whose motion degrades with time. On the other hand, for time-invariant systems, parameters stay constant. Here are path planning algorithms worth mentioning:

A. A Search Algorithm*

A* is a heuristic map-based path planning algorithm. That is, it maps out the space with nodes and equips each node with a weight associated to its minimal distance to the target. It is a complete algorithm which means that it finds most (or all) of the possible paths. In this way it has an advantage over say, RRT, since it will typically find shorter and more optimal paths. However, the practical performance of the A* search heavily depends on its heuristic function. In the worst case, the number of nodes that is expanded is exponential in the depth of the solution. This also assumes that the goal state exists, but if not, the algorithm does not terminate [7]. The challenge for A* comes in the form of developing/selecting a heuristic function that is tailored for the given problem. Beyond this, it necessitates greater storage space to map out the full domain. Due to time complexity concerns and data storage limitations, A* is better suited for smaller environments [8]. Some applications include robot navigation, route planning and scheduling, game AI (for non-player character pathfinding), and even finds a purpose in optimizing AI decision-making [9].

B. Rapidly Exploring Random Tree

Rapidly exploring Random Tree (RRT) proposed by LaValle [10] is a sampling-based path planning algorithm. It randomly generates a search tree from the initial state to the final state and can be optimized further once the target is reached. When optimization is not a priority, it can be stopped after the first path is completed and hence saves on time and computational cost. It also performs well in complex environments, such as multi-dimensional ones. On the other hand, due to RRT's behavior of sampling nodes and immediately linking the RRT tree's closest node to it, the path quality suffers from a lack of smoothness. Applications of RRT are numerous, including 3D navigation, real-time robot path planning in unknown environments [11], and robotic manipulators (ex. industrial robotic arms).

C. Probabilistic Roadmap

Probabilistic Roadmap Method (PRM) is another sampling-based path planning algorithm. PRM operates differently from RRT by constructing a roadmap during an offline learning phase [12]. This roadmap consists of randomly sampled nodes in free space, which are connected by feasible local paths (which take obstacles into account). Once constructed, this roadmap can be queried rapidly for path planning between any start and target location. One of its strengths includes path planning in a large environment. Another strong advantage is that it produces a smoother path than RRT. However, due to the extra learning phase, PRM can take more time to compute. PRM is particularly useful in static, large-scale environments where path planning can be decoupled from real-time constraints. Applications include autonomous vehicles, underwater navigation, and manipulation tasks where the environment is well-known in advance.

D. Artificial Potential Field Method

Artificial Potential Field Method (APF) is a Local-Avoidance-Based algorithm that works on the idea of vector fields. The target has an attractive vector field that pulls the robot towards itself and obstacles have repelling fields that push the robot away. This continuous control approach enables real-time computation, and allows smooth flow to the target, like a particle in the water. Its main advantages lie in a low run time and cost. Despite this, APF has a significant drawback - the local minima problem, where the robot may get trapped in a region with no clear gradient leading to the goal. Researchers have proposed various modifications, such as introducing random perturbations, or intelligent techniques such as the "follow wall" strategy [13]. In terms of obstacle avoidance, it performs especially well when obstacles are known beforehand and their locations are known for certain. On the other hand, another problem with this algorithm is that the potential field could make it impossible for the robot to pass between two close obstacles where otherwise it would be able to. Many applications are water-related, such as underwater vehicles and autonomous sailboats [14].

E. Vector Field Histogram

Vector Field Histogram [15] is a real-time sensor-dependent local obstacle avoidance method. It creates a two-dimensional map that is continuously updated with estimated obstacle densities. The way to proceed is decided based on a polar histogram associated with the robot's position. This method performs well in cluttered areas and does not face the same issue as the APF method above.

F. FGM Method

Follow the gap method (FGM) [16] is a method heavily based on geometry, that calculates what the authors call a "gap array" around the robot and chooses the trajectory that passes through the largest gap. Its geometric approach helps generate safer trajectories and avoids the local minima problem - a pitfall as outlined in the APF method.

Conclusion for Reach and Avoid

This section on Reach and Avoid to arrive at the target zone presents an array of path finding methods, each with unique strengths and trade-offs. A* is optimal in smaller or well-defined spaces, whereas sampling-based approaches like RRT and PRM offer scalability for high-dimensional and larger environments at the cost of path smoothness or optimality. APF stands out in real-time performance, but has potential issues with the local minima problem.

Stay

After reaching the target zone, the drone must remain within a safe and stable region. This "Stay" phase ensures zero velocity convergence and maintains the drone within a defined boundary inside the target zone. Stability in this phase is crucial, particularly in applications where holding position is as important as reaching the target.

A. Station-Keeping

In this control mode, a drone (or other remotely operated vehicles) maintains a fixed position autonomously, counter-acting external disturbances such as wind or currents. This is particularly useful in scenarios where the operator must focus on tasks like inspections or data collection without manually adjusting the drone's position. For underwater applications, station-keeping can be achieved using a Doppler Velocity Logger (DVL) [17]. By emitting and analyzing acoustic signals, the DVL estimates velocity relative to the seafloor, allowing precise position control. Aerial drones, on the other hand, may use GPS, optical flow sensors, or inertial measurement units (IMUs) for similar station-keeping control.

B. Lyapunov Methods

These are typically scalar functions that measure how "far" the system is from a stable equilibrium (in the situations we are interested in, this would be the target zone). The Lyapunov method allows a design of a control input u that makes the velocity of the drone decrease over time. One such study [18] applied this method to a drone model to stabilize rotational motion along the x, y, and z axes. By designing controllers based on the Lyapunov stability theory, the system ensured stable hovering while preventing violations of position and angle constraints. The UAV state converged asymptotically to the desired trajectory, demonstrating how Lyapunov-based control can be used for precise station-keeping and maintaining a hover position.

II. Problem Formulation

The objective of this project is to find a controllable path on the given operational domain from a random initial position x_0 to the target zone. The system is a discrete-time controlled linear system defined as:

$$x_{k+1} = Ax_k + Bu_k, \quad k \in \mathbb{Z}_+$$

where

$$A = \begin{bmatrix} 1 & \tau & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & \tau \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad B = \begin{bmatrix} 0.5\tau^2 & 0 \\ \tau & 0 \\ 0 & 0.5\tau^2 \\ 0 & \tau \end{bmatrix}$$

and τ is the sampling time. Here, $(x_k)_1$ and $(x_k)_3$ represent the x (horizontal) and y (vertical) positions, while $(x_k)_2$ and $(x_k)_4$ represent horizontal and vertical velocities. $(u_k)_1$ and $(u_k)_2$ denote the input forces in the x and y directions, respectively.

The operational domain is as follows:

$$X = \{x \in \mathbb{R}^4 | x_1, x_3 \in [0, 5], x_2, x_4 \in [-1, 1]\}$$

The unsafe region X_u is defined as $X_u = \cup_{k=1}^{4} X_u^k$, where:

$$X = \{x \in \mathbb{R}^4 | [1, -1, 1, -1]^\top \le x \le [2, 1, 2, 1]^\top\},$$
$$X = \{x \in \mathbb{R}^4 | [1, -1, 3, -1]^\top \le x \le [2, 1, 5, 1]^\top\},$$
$$X = \{x \in \mathbb{R}^4 | [3, -1, 1, -1]^\top \le x \le [4, 1, 2, 1]^\top\},$$
$$X = \{x \in \mathbb{R}^4 | [3, -1, 2.5, -1]^\top \le x \le [4, 1, 3.5, 1]^\top\}$$

The target set X_t is defined as:

$$X = \{x \in \mathbb{R}^4 | [4, -0.1, 4, -0.1]^\top \le x \le [5, 0.1, 5, 0.1]^\top\}$$

So the problem can be reformulated as: Given an arbitrary initial value x_0 in $X \backslash (X_t \cup X_u)$, find a control signal $\{u_k\}_{k \in \mathbb{Z}}$ in Z with a resulting trajectory $\{x_k\}_{k \in \mathbb{Z}}$ in Z.

With additional conditions:

- $u_k \in [-1, 1] \times [-1, 1]$ for all $k \in \mathbb{Z}_+$ (control constraint)
- $\exists N \in \mathbb{N}$ such that $x_k \in X \backslash X_u$ for all $k \in [0; N-1]$, $x_n \in X_t$ (reach-avoid), and
- $x_k \in X_t$, for all $k < N$ (stay)

III. Proposed Method

Among the various methods discussed, sampling-based approaches like RRT stand out for our Reach and Avoid task. It is well-suited for this problem thanks to its scalability across various maze configurations and sizes. Unlike grid-based methods that require exhaustive searches, RRT efficiently explores the space and finds feasible paths with relatively low computational cost. Additionally, RRT can be easily adapted to a given control system, allowing easy integration with the drone's dynamic constraints. Also, optimization is not a requirement, so the computational cost is minimized by choosing the first completed path.

In regards to the "Stay" part of this control problem, proportional damping was used. When a branch of the RRT tree is in the target zone, the control input u is modified such that it opposes the velocity of the drone, forcing its velocity to reduce to zero. As outlined in the definition of the target set X_t, upon reaching a velocity of $< \pm 0.1$, the drone is considered to be in a stable state. The governing equation for the "Stay" control input is as follows:

$$u = -(x_{velocity} - \text{sign}(x_{velocity}) \times (x_{velocity} \times 0.1))$$

For a positive x_{vel} case, we can prove this by rewriting:

$$u = -(x_{velocity} \times (1 - 0.1)) = -0.9 \times x_{velocity}$$

Since our system dynamics are:

$$x_{k+1} = Ax_k + Bu_k$$

and B applies u to velocity components, we get the following update rule for velocity:

$$v_{k+1} = v_k + B(-0.9v_k)$$

Approximating $B \approx 1$ for simplicity:

$$v_{k+1} = (1 - 0.9)v_k = 0.1v_k$$

Iterating this update rule:

$$v_{k+1} = 0.1v_k$$
$$v_{k+2} = 0.1v_{k+1} = 0.1^2 v_k$$
$$v_{k+n} = 0.1^n v_k$$

Since 0.1^n decreases exponentially as $n \to \infty$, we have:

$$\lim_{n \to \infty} v_{k+n} = 0$$

This proof holds for a negative x_{vel} as well.

■

To solve the Reach-Avoid-Stay problem, Python was chosen as the language of choice. Python provides a sizable mathematical library with NumPy, and a robust plotting library with Matplotlib. To begin, the program requires input parameters to build the environment as outlined earlier:

1) System dynamics matrices A and B
2) Start state x_0
3) Target zone boundaries
4) Unsafe zone boundaries
5) RRT parameters: max iterations and step size

Once the program is complete, Matplotlib will generate a grid with the path from x_0 to the target zone. If no path was found, the program will terminate once the max iterations has been reached, and a grid with RRT branches that do not reach the target zone will be generated.

The program's main loop (Algorithm 1) runs the helper functions in-order: sampling a new node, checking its validity, and adding it to the tree. This process repeats until the target zone is found, or until the max iteration count is reached.

Algorithm 1 RRT Algorithm

```
1:  Set $T = x_0$
2:  for _ in $max\_iterations$ do
3:      Sample a $x_{new}$ in grid
4:      $x_{nearest}$ = node in $T$ closest to $x_{new}$
5:
6:      Steer towards $x_{rand}$:
7:          $u = clip(K_p \times \frac{x_{rand} - x_{nearest}}{sampling\_time}, -1, 1)$
8:          $x_{new} = Ax + Bu$
9:
10:     if $x_{nearest}$ is in target_zone then
11:             $u = -(x_{vel} - sign(x_{vel}) \times (x_{vel} \times 0.1))$
12:     end if
13:     if velocity $x_{new} > 1$ then
14:         Re-calculate $u$ based on velocity of $x_{new}$
15:         *see Algorithm 2
16:     end if
17:     $x_{new} = Ax + Bu$
18:
19:     if $x_{new}$ not in an unsafe_zone then
20:             Add $x_{new}$ to $T$
21:     end if
22:
23:     if $x_{new}$ in target_zone then
24:             Set $x_{goal}$ to $x_{new}$
25:         Break from loop
26:     end if
27: end for
```

In Algorithm 1, shorthand x_{vel} is used, which denotes $x_{velocity}$. Algorithm 2 outlines the method for which the control vector u is modified if the resultant x_{new} has an x or y velocity > 1.

Algorithm 2 Modify u based on velocity of x_{new}

```
if velocity of $x_{new} > 1$ then
    Calculate excess velocity past $\pm 1$
    $u[\_] = -excess$
end if
```

From the system matrix A and input matrix B, the isolated velocity updates are:

$$x_{new} = x_{nearest} + \tau u$$

Writing these in terms of velocity, we have:

$$x_{new\_x\_vel} = x_{nearest\_x\_vel} + \tau u_x$$
$$x_{new\_y\_vel} = x_{nearest\_y\_vel} + \tau u_y$$

Every node in the tree has velocity ≤ 1 since they were created by this update rule, and the initial node's velocity is 0.

Also, since $u = -(x_{new\_vel} - 1) \leq 1$ and $\tau u \in [0, 1]$,

$$\begin{aligned} x_{new\_vel} &= x_{nearest\_vel} + \tau(-(x_{new\_vel} - 1)) \\ &\leq 1 - \tau u \\ x_{new\_vel} &\leq 1 \end{aligned}$$

■

In reality, $\tau u \in [0, 1]$ is often within $[0, 0.1]$, as $u = clip(K_p \times \frac{x_{rand} - x_{nearest}}{sampling\_time}, -1, 1)$ rarely produces x_{vel} that is significantly above 1. This is preferable as it keeps the velocity closer to 1 for faster movement.

Algorithm 3 Extract Path and Control Vectors

```
Initialize: Empty lists $path$ and $control\_inputs$
if $goal\_node$ exists then
    $node \leftarrow goal\_node$

    while $node$ is not null do
        Append $node$ to $path$
        if $node$ has control input $u$ then
            Append $u$ to $control\_inputs$
        end if
        $node \leftarrow node.parent$
    end while

    Reverse $path$ and $control\_inputs$
end if
```

The source code for this project is available at this GitHub repository in the RRT Python folder.

IV. Results

Was the Goal Achieved?

A modified implementation of RRT was successfully implemented to address the Reach-Avoid-Stay problem in a two-dimensional grid maze. The system was able to generate a collision-free path from the starting point to the target zone while respecting the control constraints imposed by the drone's dynamics.

The first iteration of the RRT planned path was inefficient, and produced mostly dead-end results. The paths that it explored were generally in the top-left and bottom-right corners, which was due to the bottom left obstacle zone "pushing" the drone from venturing into the center of the grid. Through experimenting, a few key optimizations were put into place for not only better performance, but more reliability and consistency when it comes to reaching the target zone.

Algorithm Efficiency and Parameter Tuning

A. Random Sampling Bias Constant

By incorporating a $goal\_bias$ into the random sampling process, the algorithm could be guided towards the target zone. This modification improved the convergence speed by reducing the number of iterations required to find a feasible path. However, further testing revealed that an excessively high $goal\_bias$ limits exploration of the overall environment, potentially resulting in missed alternative routes, and in the worst case scenario, getting stuck. See Algorithm 4 for implementation.

Algorithm 4 Smart Sampling Nodes

```
Initialize $path$ and $control\_inputs$ lists

Set $goal\_bias$ = 0.2
Generate $float\_rand \in [0, 1]$

**if** $float\_rand < goal\_bias$ **then**
    Sample $node\_new$ from $target\_zone$
**else**
    Sample from anywhere in the grid
**end if**
```

B. Proportional Factor K_p

The application of a proportional control factor allowed the system to adjust the control input u: it scales the difference between the random sample (x_{rand}) and the nearest node ($x_{nearest}$). The difference represents how far the new sample node is from the nearest node. The higher the difference, the stronger the control input u is applied to move in that direction. Note that u is still capped between -1 and 1 as per the project guidelines. Since K_p increases the control input based on the distance from the target, the system can move more aggressively towards the sampled points. This can significantly decrease the run time of the RRT algorithm. However, likewise with the $goal\_bias$ constant mentioned previously, a K_p value that is too large can lead to some adverse effects:

1) The control input may always be maxed out ($u_{1,2} = \pm 1$), resulting in motion that is too aggressive.
2) Overshooting and thus its corrections may make the path longer, and create unnecessary movements and oscillations. This also makes the algorithm struggle with maze configurations that have tight turns.

RRT Simulation Outputs

Using Algorithms 1–4 and additional helper functions, we simulated the Reach-Avoid-Stay problem. The following figures are the resulting plots illustrating the planned paths. The red and blue boxes represent unsafe zones and the target zone, respectively. The dotted lines represent the branches of the RRT, where the blue one is the found path from starting state to target zone.

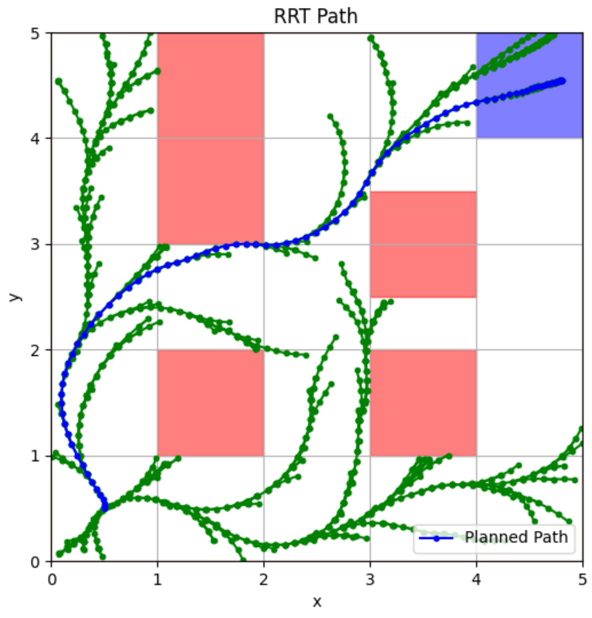

Fig. 1. Simulation with starting location at bottom left, and target zone at top right.

Comparison with Alternative Approaches

A. Manual Control Strategy

An initial implementation was to manually set control inputs based on the drone's position and velocity to steer towards the target zone. While similar to the APF method, this approach was a naive and impractical approach for motion planning. Since it was attempted in a "manual way" (with no proven algorithms), too many edge cases existed for the control input: avoiding obstacles/walls, adjusting speed, and reaching the target. When avoiding obstacles, small changes in inputs lead to large, unintended accelerations, making the drone difficult to control. The lack of global planning also meant that the system did not plan ahead, leading to situations that have resulted in dead ends.

B. MATLAB's plannerRRT

MATLAB's plannerRRT's stateSpaceSE2 object was another option that was explored, which is designed for two-dimensional pathfinding. Although promising, this method was ultimately unsuitable for our application because it did not support the inclusion of a custom control vector u and included additional state variables, such as orientation, that were not relevant to our problem. This necessitated the development of a custom RRT implementation tailored to our specific requirements.

V. Conclusion

The modified RRT algorithm not only generated paths that avoided unsafe zones but also proved generalizable across varying configurations of starting points, obstacle arrangements, and target zones. By integrating parameter

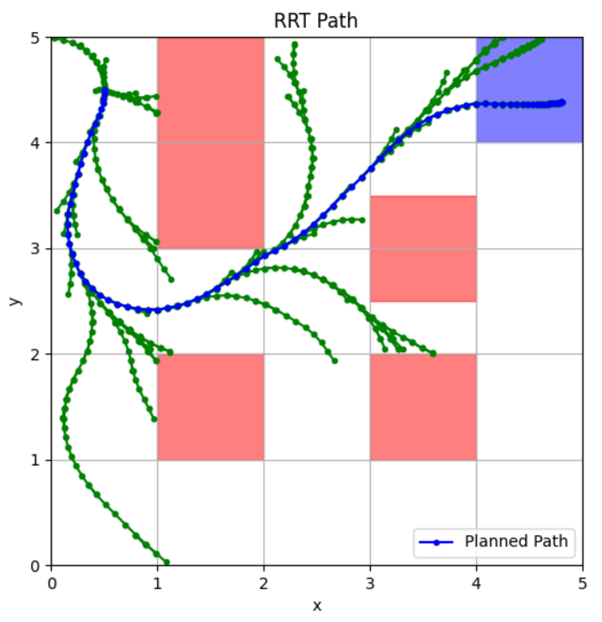

Fig. 2. Other starting states can be configured as well. Shown here is a top left starting location. The target zone can also be configured to a different location.

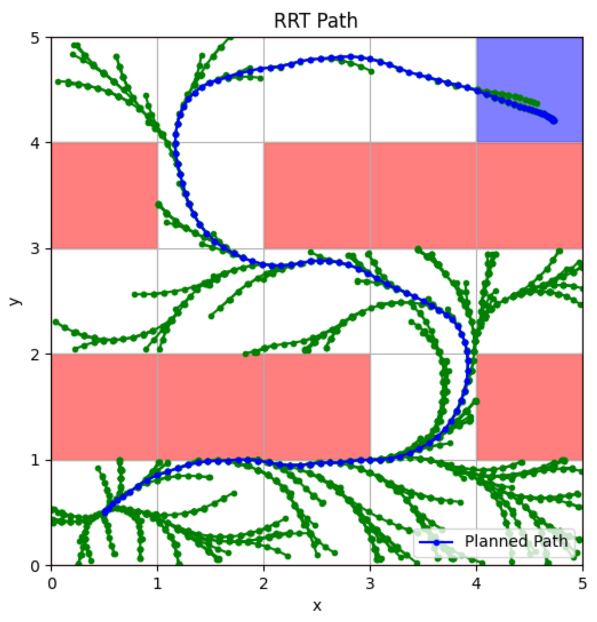

Fig. 3. The algorithm is also generalizable to different map configurations.

optimizations - specifically in the random sampling bias and proportional control factor - we achieved a significant reduction in computational time, while enhancing the reliability and consistency of reaching the target zone.

In summary, the results demonstrate that the modified RRT approach, with carefully tuned parameters, can effectively solve the Reach-Avoid-Stay problem in a two-dimensional grid environment. Future work may explore further optimization of control inputs, and integration of more advanced Stay methods such as Lyapunov stability.

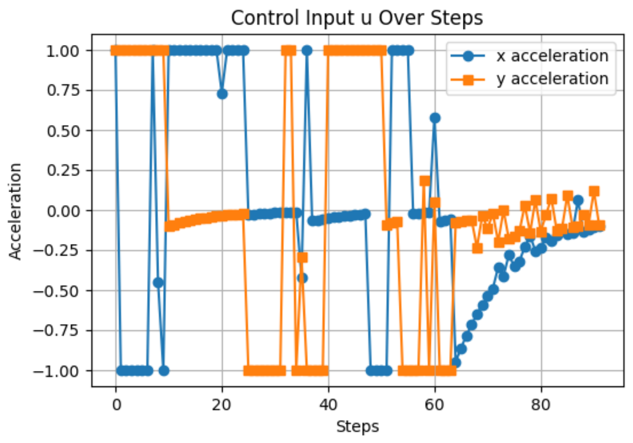

Fig. 4. The resultant $\boldsymbol{x}$ and $\boldsymbol{y}$ accelerations (control input $\boldsymbol{u}$) from $\boldsymbol{x_0}$ to the target zone of Fig.1.

References

[1] A. Gasparetto, P. Booscariol, A. Lanzutti, and R. Renato, "Path Planning and Trajectory Planning Algorithms: A General Overview," *Mechanisms and Machine Science*., vol. 29, pp. 3-27, 2015. DOI: 10.1007./978 − 3 − 319 − 14705 − 5_1.

[2] H. Qin, S. Shao, T. Wang, X. Yu, Y. Jiang, and Z. Cao, "Review of Autonomous Path Planning Algorithms for Mobile Robots," *Dones*., vol. 7, no. 3, pp. 211, 2023. DOI: 10.3390. *XXX* .7030211.

[3] Z. Tang and H. Ma, "An Overview of Path Planning Algorithms," *IOP Science*., vol. 804, 2021. DOI: 10.1088/1755-1315/804/2/022024

[4] V. Kunchev, L. Ivancevic, and A. Finn, " Path Planning and Obstacle Avoidance for Autonomous Mobile Robots: A Review," *Knowledge-Based Intelligent Information and Engineering Systems*., vol. 4252, 2006. DOI: 10.1007/11893004

[5] A. N. A. Rafai, N. Adzhar, N. I. Jaini, "A Review on Path Planning and Obstacle Avoidance Algorithms for Autonomous Mobile Robots," *Journal of Robotics*., no. 1, 2022. DOI: 10.1155/2538220

[6] R. Baraniuk, "2.1: System classifications and properties," Engineering LibreTexts, https://eng.libretexts.org/Bookshelves/Electrical_Engineering/Signal_Processing_and_Modeling/Signals_and_Systems_(Baraniuk_et_al.)/02

[7] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 2nd ed. pp. 97-104. Upper Saddle River: Prentice-Hall, 2010.

[8] B. Fu et al., "An improved A* algorithm for the industrial robot path planning with high success rate and short length," *Robotics and Autonomous Systems*., vol. 106, pp. 26–37, Aug. 2018. DOI: 10.1016/j.robot.2018.04.007

[9] A. Ravikiran, "A* search algorithm explained: Applications & Uses," Simplilearn.com, https://www.simplilearn.com/tutorials/artificial-intelligence-tutorial/a-star-algorithm (accessed Apr. 1, 2025).

[10] J. J. Kuffner and S. M. LaValle, "RRT-connect: An efficient approach to single-query path planning," *IEEE International Conference on Robotics and Automation*., vol.2, pp. 995-1001, 2000. DOI: 10.1109/ROBOT.2000.844730.

[11] Y. Tian et al., "Application of RRT-based local path planning algorithm in unknown environment," *International Symposium on Computational Intelligence in Robotics and Automation*., pp. 456–460, Jun. 2007. DOI: 10.1109/cira.2007.382896

[12] J. D. Marble and K. E. Bekris, "Asymptotically near-optimal

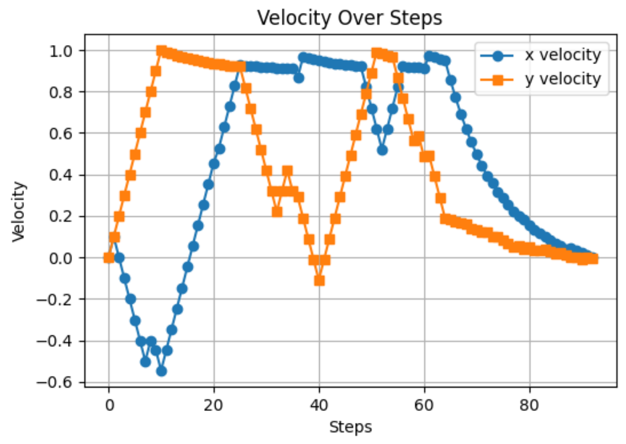

Fig. 5. The resultant x and y velocities from x_0 to the target zone of Fig.1.

planning with Probabilistic Roadmap Spanners," *IEEE Transactions on Robotics*., vol. 29, no. 2, pp. 432–444, Apr. 2013. DOI:10.1109/tro.2012.2234312

[13] Md. A. Rahman and Md. A. Azad, "To escape local minimum problem for multi-agent path planning using improved artificial potential field-based regression search method," *Proceedings of the 2017 International Conference on Information Technology*., pp. 371–376, Dec. 2017. DOI:10.1145/3176653.3176735

[14] I. Iswanto, A. Ma'arif, O. Wahyunggoro, and A. Imam, "Artificial potential field algorithm implementation for quadrotor path planning," *International Journal of Advanced Computer Science and Applications*., vol. 10, no. 8, 2019. DOI: 10.14569/ijacsa.2019.0100876

[15] J. Borenstein and Y. Koren, "The vector field histogram-fast obstacle avoidance for mobile robots," *IEEE Transactions on Robotics and Automation*., vol. 7, no. 3, pp. 278-288, June 1991. DOI: 10.1109./70.88137.

[16] V. Sezer and M. Gokasan, "A novel obstacle avoidance algorithm: 'Follow the gap method,'" *Robotics and Autonomous Systems*., vol. 60, no. 9, pp. 1123–1134, Sep. 2012. DOI:10.1016/j.robot.2012.05.021

[17] A. Viggen, "How a DVL simplifies ROV navigation and maneuvering during underwater inspections," Blueye, https://www.blueyerobotics.com/blog/how-a-dvl-simplifies-rov-navigation-and-maneuvering (accessed Apr. 1, 2025).

[18] A. Khadhraoui, M. Saad, "Barrier Lyapunov Function Based Trajectory Tracking Controller of a Quadrotor UAV," *2023 Canadian Conference on Electrical and Computer Engineering, Electronics and Energy*., vol. 8, p. 100617, Sept. 2023. DOI: 10.1109/CCECE58730.2023.10289068