Learning to Accurately Throw Paper Planes

Marcus Kornmann[1], **Qimeng He**[1], **Alap Kshirsagar**[1], **Kai Ploeger**[1], **Jan Peters**[1,2,3]

[1]TU Darmstadt [2]DFKI [3]Hessian.AI
`alap@robot-learning.de`

Abstract: The task of accurately throwing a paper plane poses significant challenges in the realm of dynamic robotic manipulation, demanding adaptability to unpredictable aerodynamic properties of paper planes. This paper presents a novel approach to accurate paper plane throwing using reinforcement learning and trajectory encoding. We introduce a method that combines a Variational Autoencoder (VAE) for encoding paper plane trajectories with a Soft Actor-Critic (SAC) algorithm to learn optimal throwing strategies. Our approach dynamically adapts to the unique aerodynamic properties of randomly generated paper plane designs. Our preliminary experiments demonstrate that incorporating information from previous throws improves performance, particularly when generalizing to unseen plane designs. By addressing the complexities of this task, our work has the potential to advance the learning of dynamic robotic manipulations.

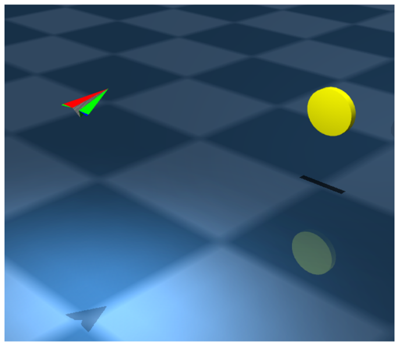
Keywords: Reinforcement Learning, Dynamic Manipulation, Paper Planes

1 INTRODUCTION

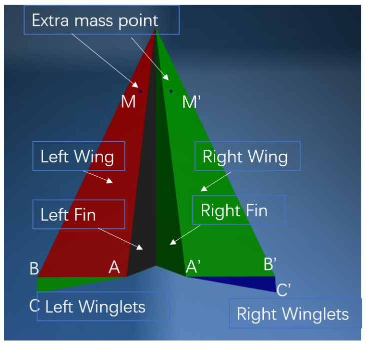
Robotic manipulation tasks frequently involve complex dynamics, accurately throwing a paper plane at a target as a representative example. Despite its apparent simplicity, this task requires precise control over the initial launch conditions and adaptation to the plane's unpredictable aerodynamic properties. It highlights broader challenges in dynamic object manipulation, where real-time adaptability is essential. Addressing this problem can enhance robotic dexterity across a range of applications. While recent work by Tanaka et al. [1] demonstrated the robotic folding of paper planes, the precise throwing of paper planes remains an open challenge.

In this work, we focus on robot learning to accurately throw paper planes at a given target, aiming to develop techniques that apply to a wide range of dynamic manipulation tasks. We propose an episodic reinforcement learning (RL) framework to optimize initial velocity and orientation for successful throws, using MuJoCo for simulation (see Fig. 1). The robot sets the plane's initial velocity and orientation, determining its trajectory. We randomize target positions and plane properties for robustness and improved sim-to-real transfer. We use a Variational Autoencoder (VAE) to generate latent embeddings of past plane trajectories, capturing flight behavior patterns without explicit aerodynamic modeling. We use the Soft Actor-Critic (SAC) Algorithm to optimize the initial conditions for launching a paper plane based on the target location and the latent trajectory embeddings of previous throws. By learning from past throws, our agent can handle various planes without needing plane-specific models and can adapt to unseen planes based on prior throws.

Reinforcement learning has been successfully applied to various object-throwing tasks, such as a dart-throwing robot [2], a ball-throwing robot [3], and TossingBot [4], showcasing its ability to enhance accuracy through continuous interaction with the environment. However, these works differ from ours in key aspects. For the dart-throwing robot, aerodynamic effects are minimal, as darts have stable flight dynamics. TossingBot, on the other hand, emphasizes joint learning of grasping and throwing arbitrary objects but does not account for the complex aerodynamic properties that significantly influence lightweight objects like paper planes. In contrast, our work focuses on optimizing the precise launch conditions for paper planes, where aerodynamics and object-specific characteristics play a critical role in achieving accurate throws.



(a)



(b)

Figure 1: Illustrations of the task and the paper plane model in MuJoCo. (a) The robot's task is to throw a paper plane as accurately as possible onto the target yellow disc. (b) Our initial paper plane model is based on a standard symmetric design. For domain randomization, we adjust vertex coordinates to alter wing and winglet length and angles asymmetrically, and reposition the extra mass point to create asymmetric inertia.

2 METHODS

Our approach to training a reinforcement learning agent for accurate paper plane throwing is shown in Figure 2. We use domain randomization to generate diverse plane models, capturing the real-world variability in plane design. We employ a Variational Autoencoder (VAE) to encode trajectory information, allowing the agent to learn about the plane's unique dynamics from past throws. We frame the paper plane throwing problem as a Markov Decision Process (MDP) and implement the Soft Actor-Critic (SAC) algorithm to train the agent.

2.1 Domain Randomization and Handling Plane Behaviors

We begin with a symmetrical base model and apply domain randomization to simulate the natural asymmetries found in real paper planes. By altering vertex positions (see Figure 1b), we introduce asymmetries in wing and winglet angles, as well as variations in mass distribution, influencing the plane's flight path. In MuJoCo, we model the environment with air density and viscosity parameters to closely mimic realistic conditions. Due to these randomized plane characteristics, each plane displays unique aerodynamic behaviors. Instead of focusing on physical characteristics, the agent learns from the behavior captured in trajectory data using a VAE. The VAE compresses this data into a latent embedding, encapsulating the plane's flight dynamics without needing explicit physical attributes, making our method applicable to real-world scenarios where precise measurements are often unavailable. The architecture of the VAE is described in Appendix A.1 and the effects of modifying domain parameters on the flight trajectory are described in Appendix A.3.

2.2 Reinforcement Learning Framework and Training Scenarios

Our environment is formulated as an MDP, where states include spatial information and latent representations of past throws, actions define initial velocity and orientation, and rewards are based on the minimal distance to the target (see Appendix A.2 for more details). We use the SAC algorithm for continuous action learning. Training occurs in two scenarios: (1) a one-step setting where each throw is treated as an independent episode, and previous trajectories are stored in a "plane store" for reference, and (2) a multistep setting where episodes consist of multiple throws, progressively accumulating trajectory data to refine the agent's strategy. In the multistep scenario, we employ a discount factor to prioritize later throws, leveraging the agent's increasing familiarity with the

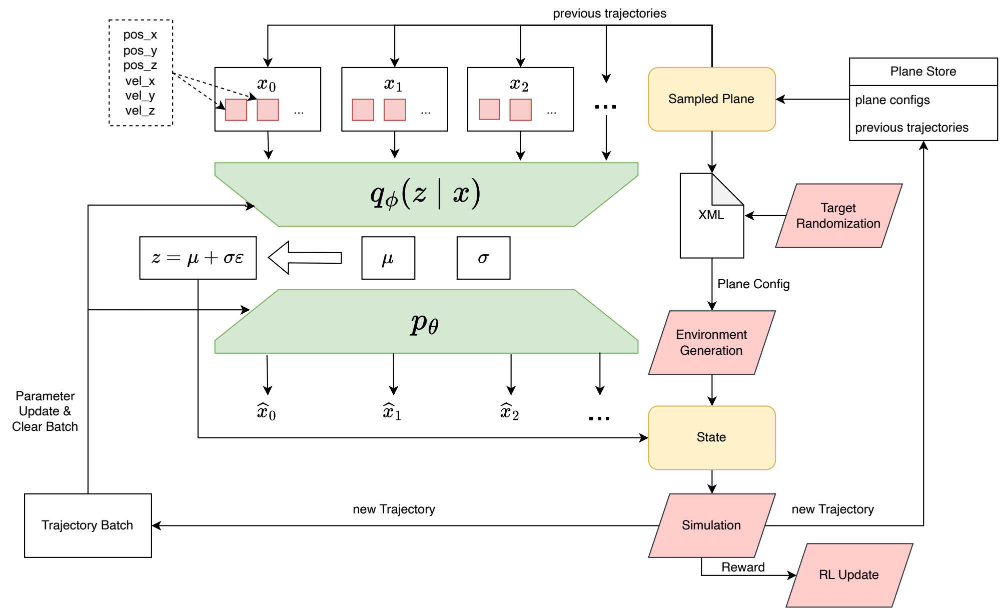


Figure 2: Schematic representation of the paper plane throwing optimization approach. The system comprises a plane store containing diverse plane configurations and their previous throw trajectories, an environment generator, a Variational Autoencoder (VAE) for trajectory encoding, and a Reinforcement Learning (RL) agent.

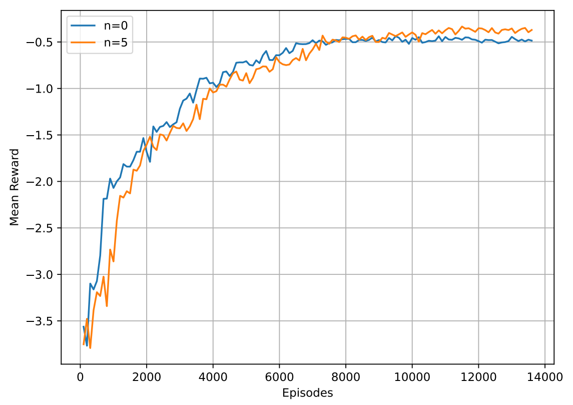
plane's behavior. The VAE is initially pre-trained on collected trajectories and updated intermittently, supporting SAC in real-time trajectory encoding and enabling the agent to generalize across diverse throwing tasks.

3 EXPERIMENTS

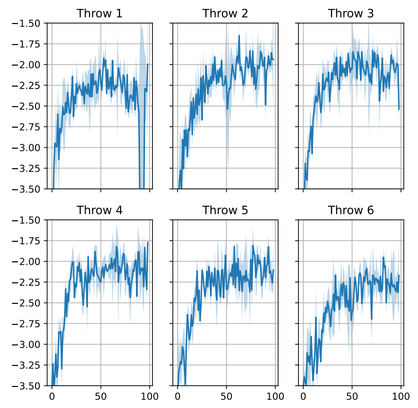
As described in the Methods section, we implement two distinct training scenarios: a one-step setting with a plane store and a multistep setting with episode-based trajectory accumulation. Here, we detail the specific parameters and implementation choices for each scenario.

For the one-step setting, we use a plane store containing 10 different plane designs. We conduct a comparative analysis of two models: one where the agent has no knowledge about prior throws and another that incorporates the trajectories of up to five previous throws. As illustrated in Figure 3a, the model focusing solely on environmental information initially demonstrates superior performance. However, as the simulation progresses, the model that utilizes individual plane behavior data achieves better performance. The benefits of incorporating information from previous throws become even more evident when evaluating the models' performance on a set of 20 previously unseen paper planes. In these novel scenarios, the model that utilizes prior throw information consistently outperforms the model that relies exclusively on environmental configuration.

For the multistep setting we also use a set of 10 different plane designs. Each episode consists of $n + 1$ throws, where $n = 5$. We train the agent for 15000 episodes. The buffer of previous trajectories starts empty at the beginning of each episode and is filled progressively. Figure 3b illustrates the evolution of agent performance across multiple throws within a single episode, providing insights into how the accumulation of trajectory information impacts the agent's ability to optimize its throwing strategy. The performance of the first throw, which occurs without any prior trajectory information, initially shows an upward trend. However, we observe a subsequent decline in performance as training progresses. In contrast to the first throw, the second throw, which incorporates



(a)



(b)

Figure 3: (a) Mean reward progression over training episodes, comparing agents with and without knowledge of past trajectories. (b) Rewards evolution across multiple throws in a single episode, with subsequent throws incorporating prior throw data.

information from the trajectory of the first throw, demonstrates consistent improvement throughout the training process. Interestingly, the performance of throws that utilize information from multiple previous trajectories (throws 3-6) exhibits oscillatory behavior. Moreover, we observe that as the number of previous trajectories used increases, there is a trend toward decreased performance.

4 Discussion and Future Work

Our study reveals distinct performance patterns in a reinforcement learning agent tasked with optimizing paper plane throws across two training scenarios, highlighting the significance of historical trajectory information. Initially, the agent performs better without utilizing past throw trajectories due to a consistent throwing strategy, but after around 8000 iterations, the model that incorporates historical data begins to excel by leveraging the unique flight characteristics of each paper plane for more precise throws. Although the second scenario shows an initial performance improvement followed by a decline—suggesting potential overfitting—the agent benefits from limited historical data, particularly in the second throw. Qualitative observations indicate a strategic shift from high velocity in the first throw to lower speeds in subsequent attempts, hinting at a reliance on past trajectories.

For future work, we aim to investigate why the first throw enhances the second throw's performance while additional trajectory information may be less beneficial. Conducting ablation studies to isolate the impacts of various trajectory components will be crucial, as will increasing the complexity of the paper plane model to improve adaptability. Furthermore, we plan to incorporate a robotic model for executing paper plane throws, beginning with simulations to assess the agent's strategies in controlled environments before transitioning to real-world applications. This approach will test the robustness and adaptability of our methods in practical scenarios, bridging the gap between simulation and real-world robotics and dynamic object manipulation tasks. Overall, our research demonstrates the effectiveness of combining reinforcement learning with trajectory encoding to enhance paper plane throwing strategies, offering valuable insights for the learning of dynamic robotic manipulations.

Acknowledgments

This research was supported by Research Clusters “The Adaptive Mind”, “Third Wave of AI”, and LOEWE research priority program “WhiteBox” under grant LOEWE/2/13/519/03/06.001(0010)/77, funded by the Excellence Program of the Hessian Ministry of Higher Education, Science, Research and the Arts. The authors acknowledge the grant “Einrichtung eines Labors des Deutschen Forschungszentrum für Künstliche Intelligenz (DFKI) an der Technischen Universität Darmstadt” of the Hessisches Ministerium für Wissenschaft und Kunst.

References

[1] R. Liu, J. Liang, S. Sudhakar, H. Ha, C. Chi, S. Song, and C. Vondrick. Paperbot: Learning to design real-world tools using paper. *arXiv preprint arXiv:2403.09566*, 2024.

[2] C. Obayashi, T. Tamei, and T. Shibata. Assist-as-needed robotic trainer based on reinforcement learning and its application to dart-throwing. *Neural Networks*, 53:52–60, 2014.

[3] Y.-G. Kang and C.-S. Lee. Deep reinforcement learning of ball throwing robot’s policy prediction. *The Journal of Korea Robotics Society*, 15(4):398–403, 2020.

[4] A. Zeng, S. Song, J. Lee, A. Rodriguez, and T. Funkhouser. Tossingbot: Learning to throw arbitrary objects with residual physics. *IEEE Transactions on Robotics*, 36(4):1307–1319, 2020. doi:10.1109/TRO.2020.2988642.

[5] A. Vaswani. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017.

[6] P. K. Diederik. Adam: A method for stochastic optimization. *(No Title)*, 2014.

A Appendix

A.1 Architecture of the Variational Autoencover (VAE)

The architecture of our VAE is shown in Figure 4.

- Encoder: The encoder uses sinusoidal positional encoding [5], followed by a linear layer and a ReLU activation function. This is followed by multi-head attention [5], another linear layer with ReLU, and finally two separate linear layers that output the mean ($\boldsymbol{\mu}$) and logarithm of the variance ($\log \boldsymbol{\sigma}^2$) of the latent distribution.
- Decoder: The decoder consists of a linear layer, sinusoidal positional encoding, multi-head attention, a LSTM layer, and a final linear layer. The input to the decoder is the latent representation $\boldsymbol{z} = \boldsymbol{\mu} + \boldsymbol{\sigma}^2 \cdot \epsilon$ with $\epsilon \sim \mathcal{N}(0, \boldsymbol{I})$.

We choose this architecture for our VAE model because it effectively addresses the challenges posed by variable-length trajectory data. The combination of Attention and LSTM enables the model to handle sequences of different lengths, capture long-range dependencies, and encode contextual information. The multi-head attention mechanism in the encoder allows the model to focus on relevant parts of the input trajectory, generating a rich and informative latent representation. The LSTM layer in the decoder helps maintain the temporal coherence of the generated trajectories by capturing and utilizing the relevant information from the entire sequence. Additionally, the sinusoidal positional encoding in both the encoder and decoder preserves the temporal structure and ordering of the points in the trajectory. We train the VAE with the following loss function:

$$\mathcal{L}_{\text{VAE}} = \mathbb{E}q_\phi(z|x)[\log p_\theta(x|z)] - D_{\text{KL}}(q_\phi(z|x)||p(z))$$

where $q_\phi(z|x)$ is the encoder and $p_\theta(x|z)$ is the decoder.

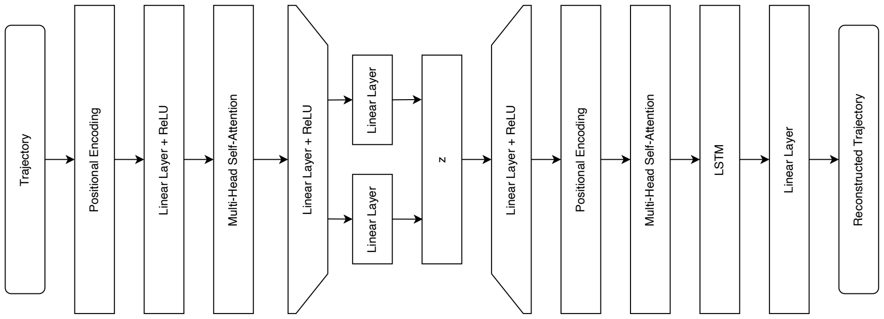


Figure 4: Architecture of the Variational Autoencoder (VAE) used for learning latent representations of paper plane trajectories. The encoder (left) processes variable-length input sequences using sinusoidal positional encoding, multi-head attention, and fully connected layers to generate the latent distribution $\boldsymbol{z}$. The decoder (right) takes a sample $\boldsymbol{z}$ from the latent distribution and reconstructs the input sequence using positional encoding, multi-head attention, an LSTM layer, and a final linear layer. The VAE architecture enables the model to handle trajectories of different lengths and capture the essential aerodynamic behavior of the paper planes in the latent space.

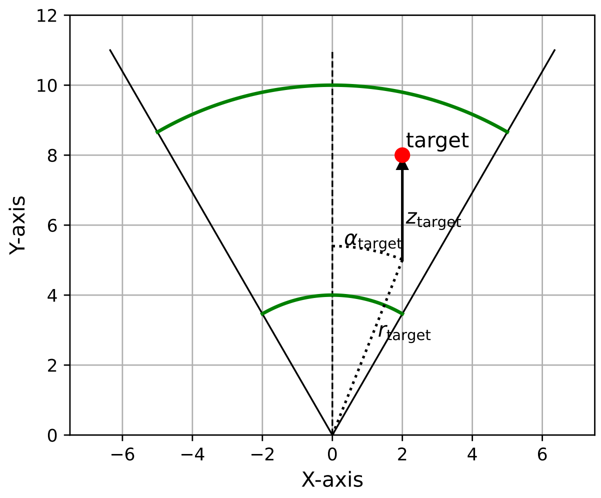


Figure 5: Illustration of the constraints and parameters defining target positions. The target always lies within the area bounded by the angle range (depicted by black diagonal lines) and the radius limits (shown as green circular arcs). Each target point is uniquely characterized by three parameters: its angle from the y-axis (α), its radial distance on the x-y plane, and its height (z) above this plane.

A.2 Enviroment Details

Our environment is modeled as a MDP $= (\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{T}, \gamma)$. Every state $s \in \mathcal{S}$ is represented as a tuple $(z_{\text{initial}}, z_{\text{target}}, d, \mathbf{v}, \mathbf{z}_1, \ldots, \mathbf{z}_n)$. The first components are a representation of the current setup, while the later components $(\mathbf{z}_1, \ldots, \mathbf{z}_n)$ encode the trajectories of previous throws. z_{initial} and z_{target} are the z-coordinates of the initial and target position, d is the Euclidean distance between the plane and the target, and $\mathbf{v}$ is the normalized difference between the plane's initial position and the target's position. Specifically, $\mathbf{v}$ is calculated by taking the elementwise difference between the target position and the initial position and normalizing this difference vector so that its components sum to 1. This state representation was chosen to ensure spatial invariance, allowing the model to generalize across different throwing scenarios regardless of absolute position in 3D space. $\mathbf{z}_1, \dots \mathbf{z}_n$ are the latent representations of the n previous throws with the same plane. If fewer than n throws have been recorded, the missing values are filled with zeros. An action $a \in \mathcal{A}$ is a 3-tuple, setting the initial orientation around the x- and z-axis and the initial speed in meters per second along this orientation. The reward $r \sim R(\cdot|s, a)$ is the negative minimal distance of the plane to the target during the whole trajectory.

The position of the target is sampled from a variable circle around the origin defined by a 3-tuple $(r_{\text{target}}, \alpha_{\text{target}}, z_{\text{target}})$ where $r_{\text{target}} \sim \mathcal{U}(4, 10)$ [meters] is the radius of this circle, $\alpha_{\text{target}} \sim \mathcal{U}(-30, 30)$ [degrees] is the angle to the target relative to a reference orientation and $z_{\text{target}} \sim \mathcal{U}(0.5, 2.2)$ [meters] is the z-coordinate of the target. The interaction between these parameters can be seen in figure 5.

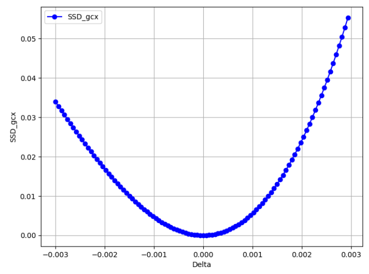
A.3 Effect of modifying domain parameters on the flight trajectory

We select the center of gravity, geometry of wings, geometry of winglets as domain parameters. As shown in Figure 1b, we modified them by adjusting the position of some vertexes (B,B',C,C',M,M'). We use Sum of Squared Difference (SSD) between current trajectory (x_i, y_i, z_i) and trajectory of standard or base model $(x_{base}, y_{base}, z_{base})$ to evaluate the effect of different parameters:

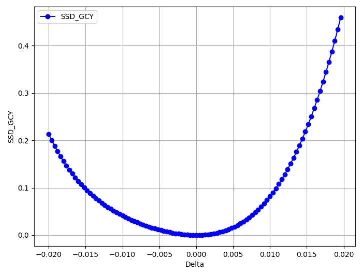
$$SSD = \sum_{i=1}^{n}((x_i - x_{base})^2 + (y_i - y_{base})^2 + (z_i - z_{base})^2)$$

During the simulation it is assumed that the wind velocity is 0 and initial conditions for every throw, such as the velocity, position and orientation, is consistant. In each experiment the parameter of interest is divided into 100 levels so we throw the paper plane 100 times to generate corresponding position data points and then calculate SSD values. The result is shown in Figure 6. We can find that modifying the position of center of gravity and geometry of wings affect the trajectory slightly. there are unusual data points on the plot of modifying x-coordinate of C and C' but overall the value of SSD is small. A significant effect on the flight trajectory is observed when modifying the z-coordinates of vertices C and C' of the winglets.

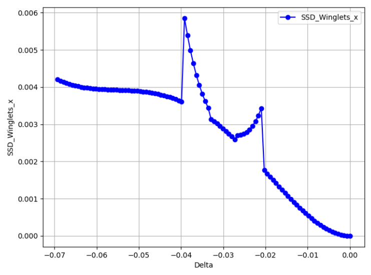
It is observed from Figure 6 that modifying the z-coordinates of vertices C and C' of the winglets affects the trajectory significantly. By focusing on the winglet, we can more effectively manage the complexity of our experimental design while at the same time significantly influencing a plane's aerodynamic properties. The parameter selection process involves training our model on a diverse set of 130 randomly generated plane designs, with each design's agent undergoing 5000 iterations of training. This approach helps us identify a range of plane designs for which our algorithm can effectively learn to hit the target, while still allowing us to test our model's ability to generalize across different plane configurations within this established range. Next, we use planes generated from the identified parameter range to collect a set of trajectories, which are then used to pre-train our VAE. We set the dimension for the hidden linear layers to 16 and the dimension of the latent space to 8. As discussed in the methods chapter, each point in a trajectory consists of a 10-dimensional vector, so our input dimension is 10. We train the VAE for 15 epochs using the Adam [6] optimizer with a learning rate of 10^{-3}. The pre-trained VAE serves as the initialization for all subsequent experiments. Finally, we train the SAC model, varying the number n of previous throws considered. The SAC policy is initialized randomly, while the VAE is initialized using the pre-trained VAE. Both models are trained simultaneously, as described in the methods section.



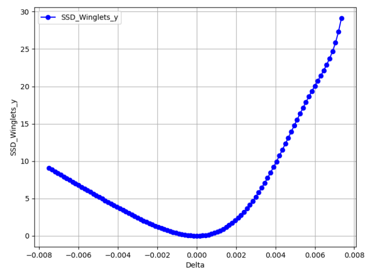
(a)



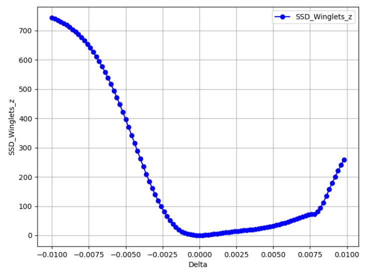
(b)



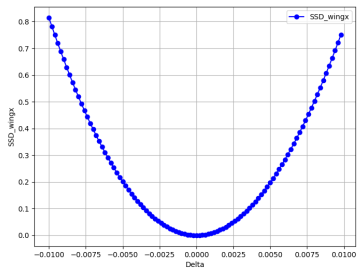
(c)



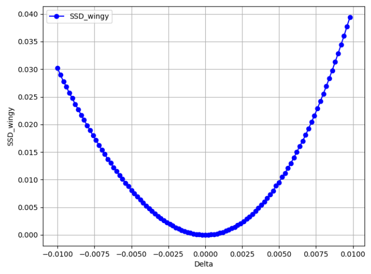
(d)



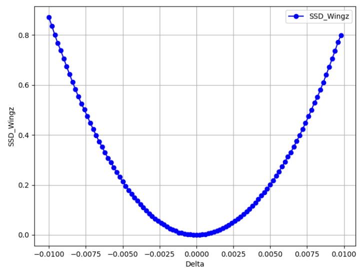
(e)



(f)



(g)



(h)

Figure 6: Effect of different parameters on the SSD of various trajectories realtive to the standard paper plane model trajectory. (a) and (b) shows the effect of modifying the center of gravity by modifying x- and y- coordinates of M and M', z-coordinate is linear constrained by x- and y-coordinate. (c)-(e) shows the effect of modifying geometry of winglets by modifying the x-, y- and z-coordinates of C and C'. (f)-(h) shows the effect of modifying geometry of wings by modifying position of B and B' .