Akademia Górniczo-Hutnicza

w Krakowie

Faculty: WIMiR	**Academic Year:** 2025/2026	**Year of study:** 5th	**Field of science:** Mechatronic Engineering

Student(s) name:	Luca Eugenio Belaunzarán Consejo
Course:	Individual Research Project Related to Mechatronic Design
Project name:	**Learning inverse lift mappings with REINFORCE**
Project Topic:	Reinforcement Learning
Supervisor:	dr hab. inż. Wojciech Lisowski
Date: 16/02/2026	**Grade:**

Contents

1 Introduction

The problem of inverse aerodynamic analysis is as old as the aerospace field itself. It involves finding the necessary conditions to procure a specified aerodynamic behaviour as opposed to the usual approach of predicting the behaviour based on set conditions. The allure of these types of analyses is clear for problems related to design and optimization, and indeed these are the main focuses found in literature. In said cases, inverse aerodynamic analysis would, in theory, present a direct solution from desired performance metrics in contrast to a cumbersome iterative process. However, inverse aerodynamics can also be used to greatly simplify active control problems, particularly when dealing with morphic aerodynamic structures such as morphic wings. It is in the context of this largely unexplored niche that this project was chosen and undertaken.

Despite the undeniable potential of inverse aerodynamic analysis in solving a wide range of problems, its current use is in the industry and academia is still small in scope and shallow in depth. This is due in no small part to the highly complex nature of the phenomena involved in aerodynamic, aeroelastic and aeroservoelastic analysis. However, it must also be said that approaching these types of problems from a control perspective is fundamentally different than from an optimization perspective. Particularly, the number of possible degrees of freedom involved in optimization problems is largely unbounded which creates a stark contrast to control problems where the number of degrees of freedom is set directly by the system's characteristics. For example, a morphic wing with varying geometric torsion along its span only has as many degrees of freedom as it has actuators.

A reduced degrees of freedom set, combined with straight-forward performance metrics invites questions about the feasibility of Machine Learning (ML) approaches in mapping between a morphic aerodynamic structure and its corresponding aerodynamic response as the structure's configuration changes state. Such an approach would bypass a classical control's typical requirements for a differentiable model or an analytical inverse of the system. Instead, the system is treated like a black box and an agent is trained to produce desired aerodynamic behaviour from learned experience. The main trade-off with ML models in this case is the uncertainty inherent to any experience-based solution. Particularly, the data fed into the system has to be carefully curated to account for the entire operational range which the agent is expected to encounter. Even then, the behaviour of the agent will always remain stochastic and should be treated as such. A possible solution to introduce determinism is the distillation of a fuzzy-logic based control approach from a trained agent's policy. This goes far beyond the scope of this research project.

Amongst the different paradigms in ML, Reinforcement Learning (RL) stands out as an ideal choice for a first approximation to an experience-based inverse aerodynamic mapping agent. This is due to RL's eyes-shut approach regarding environmental phenomena and the difficulty to implement supervised learning models with the unlabeled and continuous data that characterizes this application. Consequently, REINFORCE(Williams 1992), arguably the most basic RL algorithm was chosen for this project. The agent was implemented within a simulation framework developed alongside it this semester. A physics-in-the-loop configuration was constructed using Phillips' solution to Prandtl's LLT (Phillips 2004) to compute the spanwise lift distribution generated by a fixed wing with a particular geometric torsion and under set flight conditions. Training was carried out in a time-independent fashion, presenting a new randomized target distribution in each step.

The remainder of this report is structured as follows. Section 2 presents the necessary theoretical background. Section 3 goes into the modeling of the problem within a ML context. Section 4 discusses the methodology used to propose, train and test a RL model for this application. Testing and results are presented in section 5. Lastly, sections 6 and 7 respectively contain discussion and conclusions on this project.

2 Background

This section presents the theoretical framework supporting the use of REINFORCE to approximate inverse aerodynamic mappings. Basic terminology related to RL is put forth, followed by the derivation of REINFORCE as a policy optimization method. Later on, requirements to apply REINFORCE in a continuous action space are laid out. Finally, reward normalization is justified using the EGLP lemma. This entire section and the project itself assumes that the nature of the phenomena involved is stochastic.

The derivations presented in this section follow the layout found in Achiam (2018) which was the main pedagogical source for this work.

2.1 Environment, Agent, Policy and Reward

The field of RL deals primordially with agents and environments. Put simply, an environment is the surroundings that encapsulate an agent while an agent is an entity contained inside of the environment which can exert actions that perturb the environment, although the environment's state might change on its own as well. An agent is aware of the state s_t of the environment through an observation o_t which contains a selection of data that describes the current environmental state in a way which is meaningful to the agent. The agent has a policy π which determines what action a_t to take based on the observation received. When training, policies are stochastic and actions are sampled from a given distribution. In contrast, when testing, actions are taking deterministically. This section deals mainly with policy optimization and thus it is understood that π_θ represents a probability distribution. An agent's policy is described by a set of parameters denoted as θ. An agent also deduces from the environment a reward r_t which evaluates the quality of the current state. The cumulative reward R is called the return and the agent's purpose is to maximise it. Therefore, the agent must take the correct actions such that the return is maximised. In RL, an agent's policy is iteratively adjusted through experimentation in order to increase the return obtained. This process is called policy optimization. For practical reasons a state will be used to refer to an observation for the remainder of this report.

2.2 Policy Optimization

Let τ be a trajectory. That is, a chronological sequence of tuples containing a state perceived by an agent and the corresponding action which was taken:

$$\tau = \{(s_0, a_0), (s_1, a_1), (s_2, a_2), ...\}$$

Where s_0 is the environment's initial state. In a stochastic system, s_0 is considered to be randomly sampled from distribution $p_0(\cdot)$. The subsequent state transitions are also modeled stochastically and depend on the previous state as well as the previous action:

$$s_{t+1} \sim P(\cdot|s_t, a_t)$$

It is important to note that each action depends on the agent's policy. Thus, the probability of any specific trajectory τ with T steps given an agent with policy π_θ is:

$$P(\tau|\pi_\theta) = p_0(s_0) \prod_{t=0}^{T} P(s_{t+1}|s_t, a_t)\pi_\theta(a_t|s_t) \tag{1}$$

It should be noted that $\pi_\theta(a_t|s_t)$ represents the probability that π_θ will select action a_t from state s_t. Using equation 1, an expression for the expected cumulative reward of a given policy $J(\pi_\theta)$ can be obtained by taking the product of the probability of each trajectory with its respective cumulative reward $R(\tau)$ and integrating it across every possible trajectory.

$$J(\pi_\theta) = \int_\tau P(\tau|\pi_\theta)R(\tau)d\tau = \underset{\tau\sim\pi}{E}[R(\tau)] \tag{2}$$

From this equation, it can be said that the end goal of RL is finding the optimal policy π^* that maximises $J(\pi_\theta)$. In order to do so, the gradient of $J(\pi_\theta)$ with respect to θ is taken. This gradient will represents the direction of maximum rate of change of $J(\pi_\theta)$ in the parameter space. Subsequently, parameters can be updated in the direction computed:

$$\nabla_\theta J(\pi_\theta) = \nabla_\theta \int_\tau P(\tau|\pi_\theta)R(\tau)d\tau = \int_\tau \nabla_\theta P(\tau|\pi_\theta)R(\tau)d\tau$$

Log derivative properties can be used:

$$\frac{d}{dx}\log(f(x)) = \frac{f'(x)}{f(x)} \rightarrow f'(x) = f(x)\frac{d}{dx}\log(f(x))$$

$$\nabla_\theta P(\tau|\pi_\theta) = P(\tau|\pi_\theta)\nabla_\theta \log(P(\tau|\pi_\theta))$$

Going back to equation 1, the log probability of a trajectory is taken, transforming the product into a sum:

$$\log\left(P(\tau|\pi_\theta)\right) = \log\left(p_0(s_0)\right) + \sum_{t=0}^{T}\left(\log\left(P(s_{t+1}|s_t,a_t)\right) + \log\left(\pi_\theta(a_t|s_t)\right)\right)$$

When taking the gradient of such a log probability with respect to θ, terms which do not depend directly on θ can be dropped:

$$\nabla_\theta \log\left(P(\tau|\pi_\theta)\right) = \cancel{\nabla_\theta \log\left(p_0(s_0)\right)} + \sum_{t=0}^{T}\Big(\cancel{\nabla_\theta \log\left(P(s_{t+1}|s_t,a_t)\right)} + \nabla_\theta \log\left(\pi_\theta(a_t|s_t)\right)\Big)$$

$$\nabla_\theta \log\left(P(\tau|\pi_\theta)\right) = \sum_{t=0}^{T}\nabla_\theta \log\left(\pi_\theta(a_t|s_t)\right)$$

Thus:

$$\nabla_\theta J(\pi_\theta) = \int_\tau \nabla_\theta P(\tau|\pi_\theta)R(\tau)d\tau$$

$$\nabla_\theta J(\pi_\theta) = \int_\tau P(\tau|\pi_\theta)\nabla_\theta \log\left(P(\tau|\pi_\theta)\right)R(\tau)d\tau$$

In the previous equation, an integral across all trajectories of an arbitrary term multiplied by the probability of each trajectory can be otherwise expressed as an expected value:

$$\nabla_\theta J(\pi_\theta) = \underset{\tau\sim\pi}{E}\left[\nabla_\theta \log\left(P(\tau|\pi_\theta)\right)R(\tau)\right]$$

$$\nabla_\theta J(\pi_\theta) = \underset{\tau\sim\pi}{E}\left[\sum_{t=0}^{T}\nabla_\theta \log\left(\pi_\theta(a_t|s_t)\right)R(\tau)\right] \tag{3}$$

From statistics, any expected value from a probability distribution can be adequately approximated given a sample size large enough. It follows that the Equation 3 can be numerically approximated:

$$\hat{g} = \frac{1}{N}\sum_{\tau\in N}\sum_{t=0}^{T}\nabla_\theta \log\left(\pi_\theta(a_t|s_t)\right)R(\tau) \tag{4}$$

Where N is the number of trajectories used to compute the gradient. Equation 4 was first derived by Williams (1992). Importantly, $\hat{g}$ represents the simplest policy gradient in RL and is at the core of REINFORCE.

2.3 Continuous action spaces

As discussed in the previous subsections, during training π_θ represents a θ dependent distribution from which an agent may sample an action. Furthermore, approximating $\hat{g}$ numerically using equation 4 requires log probabilities of such a distribution to be taken. When the agent is a neural network (which is often the case, including in this work) and the action space (the set of all possible actions that an agent can choose from) is discrete, then turning the network's output into a categorical distribution is trivial. However, when the action space is continuous, a different way to introduce stochasticity is required. If the space is unidimensional (one actuator), then a normal distribution can be introduced at the output of the network with the mean located at the output of the previous layer. When the action space is multidimensional (multiple actuators), a multivariate gaussian distribution is used instead.

In both unidimensional and multidimensional cases, the distribution variance is added to the agent's parameters in the form of the standard deviation σ for normal distributions or the covariance matrix Σ for the multivariate gaussian case. In order to avoid invalid negative values, these parameters are often defined as $\log(\sigma)$. The covariance matrix is a square matrix that represents the covariance present when sampling each pair of elements in the action vector. For practical reasons, it is convenient to restrict Σ to a diagonal matrix in order to represent it as a vector which implies that the variance present in each dimension of the action space is completely independent. This does not necessarily imply that each of the action elements sampled is completely independent as the agent's policy can still learn correlations amongst them. This will translate to each action element being sampled from a distribution with a dependent mean and an independent variance. For a k-dimensional action space, the log probability of a diagonal gaussian distribution described by mean vector μ_θ and variance vector σ_θ is derived from the gaussian univariate (normal) Probability Density Function (PDF) as follows:

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

Thus, for an agent with an unidimensional action space:

$$\pi_\theta(a|s_t) = \frac{1}{\sqrt{2\pi\sigma_\theta^2}} e^{-\frac{(a-\mu_\theta)^2}{2\sigma_\theta^2}}$$

Due to the absence of variance correlation in a k-dimensional diagonal gaussian distribution, the probability of an action vector in a k-dimensional action space sampled from such a distribution is simply the product of the probability of each of its elements:

$$\pi_\theta(a|s_t) = \prod_{i=1}^{k} \frac{1}{\sqrt{2\pi\sigma_{\theta i}^2}} e^{-\frac{(a_i-\mu_{\theta i})^2}{2\sigma_{\theta i}^2}}$$

Equation 4 requires computing the log probability of this distribution which turns the product into a sum of logs:

$$\log\left(\pi_\theta(a|s_t)\right) = \sum_{i=1}^{k} \log\left(\frac{1}{\sqrt{2\pi\sigma_{\theta i}^2}} e^{-\frac{(a_i-\mu_{\theta i})^2}{2\sigma_{\theta i}^2}}\right)$$

$$\log\left(\pi_\theta(a|s_t)\right) = \sum_{i=1}^{k} \left(\log\left(\frac{1}{\sqrt{2\pi\sigma_{\theta i}^2}}\right) + \log\left(e^{-\frac{(a_i-\mu_{\theta i})^2}{2\sigma_{\theta i}^2}}\right)\right)$$

$$\log\left(\pi_\theta(a|s_t)\right) = \sum_{i=1}^{k} \left(-\frac{1}{2}\log\left(2\pi\sigma_{\theta i}^2\right) - \frac{(a_i-\mu_{\theta i})^2}{2\sigma_{\theta i}^2}\right)$$

$$\log\left(\pi_\theta(a|s_t)\right) = -\frac{1}{2}\sum_{i=1}^{k} \left(\log(2\pi) + \log\left(\sigma_{\theta i}^2\right) + \frac{(a_i-\mu_{\theta i})^2}{\sigma_{\theta i}^2}\right)$$

$$\log\left(\pi_\theta(a|s_t)\right) = -\frac{1}{2}\left(k\log(2\pi) + \sum_{i=1}^{k} \left(\log\left(\sigma_{\theta i}^2\right) + \frac{(a_i-\mu_{\theta_i})^2}{\sigma_{\theta i}^2}\right)\right) \tag{5}$$

2.4 Reward normalization

Every probability distribution is normalized such that:

$$\int_x P(x) = 1$$

The gradient of a probability can be taken with the help of the same log derivative properties used in the derivation of equation 3:

$$\nabla \int_x P(x) = \nabla 1 = 0$$

$$0 = \int_x \nabla P(x)$$

$$0 = \int_x P(x) \nabla \log\left(P(x)\right)$$

$$\underset{x\sim P}{E}\left[\nabla \log\left(P(x)\right)\right] = 0 \tag{6}$$

Equation 6 is known as the Expected Gradient Log Probability Lemma (EGLP Lemma). From this lemma, it follows that for any function $b(s)$ which only depends on state:

$$\underset{a_t \sim \pi_\theta}{E}\left[\nabla_\theta \log\left(\pi_\theta(a_t|s_t)\right) b(s_t)\right] = 0$$

This means that it is possible to add or subtract any number of such functions in equation 3 without any consequence (Sutton et al. 1999). Of particular interest is subtracting a baseline $b(\tau)$ from the return of a given trajectory:

$$R'(\tau) = R(\tau) - b(\tau)$$

$$\nabla_\theta J(\pi_\theta) = \underset{\tau\sim\pi}{E}\left[\sum_{t=0}^{T} \nabla_\theta \log\left(\pi_\theta(a_t|s_t)\right)(R'(\tau))\right]$$

A baseline can be used to ensure that the agent perceives equal amounts of positive and negative reinforcement which can help in training in cases where reward values are extreme.

3 Problem Formulation

In order to train and test an agent to perform inverse aerodynamic mappings, a rectangular wing with dynamically changing geometric torsion is proposed. In order to describe an instantaneous geometric torsion state (from hereon referred to as wing state), a vector of bounded angles of attack is used (figure 1). Torsion is specified at discrete points corresponding to actuator positions and then interpolated across the rest of the span. The lift distribution generated by the wing state and set flow parameters is approximated using Phillips' solution to Prandtl's Lifting Line Theory (LLT). Relevant wing and flow parameters are specified in table 1. Not specified in table 1 are the fluid's density and viscosity, as well as atmospheric pressure. These values are computed automatically during the simulation based on the specified flight altitude using the International Standard Atmosphere model(International Organization for Standardization 1975).

Parameter		**Value**
Wingspan	b	6 $[m]$
Airfoil	-	NACA2412
Chord	c	1 $[m]$
Flight Altitude	h	1000 $[m]$
Fluid Velocity	V_∞	$166\hat{\imath}$ $[ms^{-1}]$
Actuator Range	Ω	$\pm5°$
Actuator Centre	ω_0	$0°$
Actuators	N	18
Domain Size	M	202
Baseline variance	σ_b	2

Table 1: Simulation settings

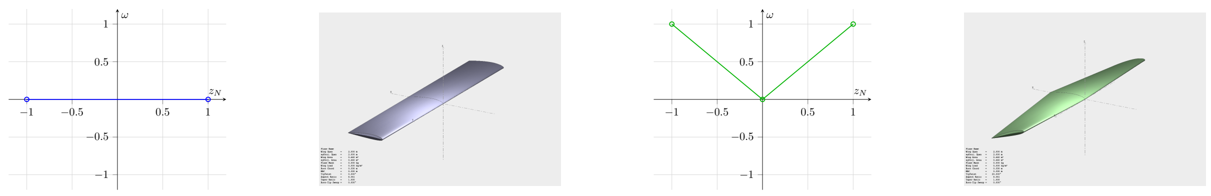


Figure 1: Vectors describing geometric wing torsion

3.1 MDP Formulation

The problem is approached as a Markov Decision Process (MDP) where the next environmental state does not depend on environmental history. An MDP is characterized by a state space an action space, state transitions and a reward function. All of the aforementioned characteristics found in the MDP implemented for this project are presented in table 2.

Element	**Implementation**	**Details**
State s	Target lift distribution	Specified across M points
Action a	Actuator position	N values bounded by $\omega_0 \pm \Omega$
Result s'	Simulated according to a	Specified across M points
Reward r	Negated integral of error	$r = -\int_M \lvert s_t - s'_t \rvert$
Transitions	Unaffected by a	No temporal dependency, each timestep represents an independent trial
Policy π_θ	Diagonal Gaussian	N-dimensional distribution. μ from Neural Network (NN) output, log of σ stored as parameter: $\sigma = e^{\phi}$ where ϕ is the learnable log sigma parameter vector.

Table 2: MDP parameters

3.2 Contextual Bandit Simplification

As mentioned in the previous subsection, the state transitions were deliberately made to be independent of any actions taken by the agent during training. This was done to avoid involving actuator dynamics which not only decreases computational load but also gives the agent more immediate feedback. Actuator dynamics are bypassed completely in the simulation, and the environment immediately evaluates the agent's action, reducing the problem to a contextual bandit. Each step is an isolated trial where the agent receives a state and takes an action with no sequential dependency. This aligns more with the scope of the project in creating an inverse mapping solution and not necessarily an active control. Additionally, this setup removes the need for reward discount factors or reward-to-go schemes. After training, actuator dynamics were added to the simulation for testing, although this was done purely for visualisation purposes.

4 Methodology

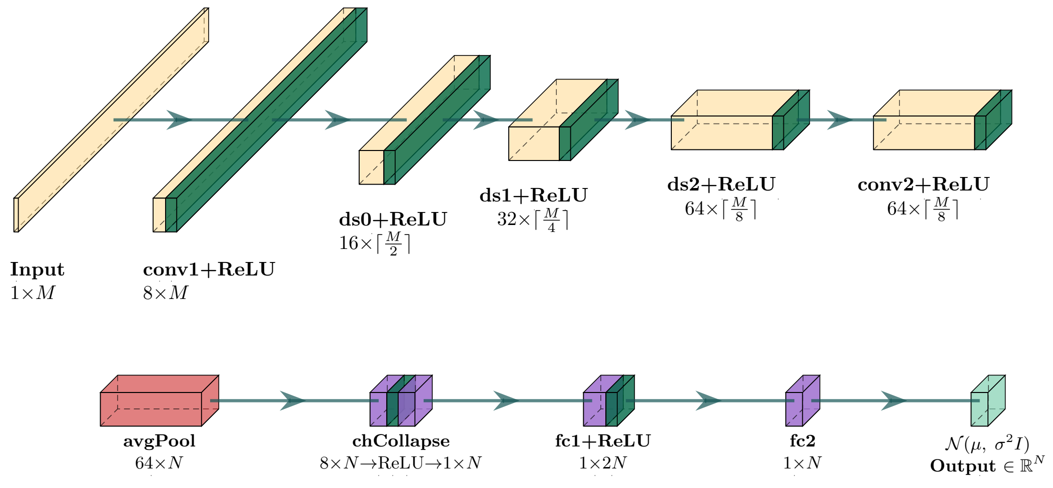
This section describes the methodology followed to train an agent to perform inverse lift mapping. First, an overview of the agent implemented, a Convolutional Neural Network (CNN) and its architecture is presented, followed by a description of the training procedure and the integration of the agent within the simulation environment. The hyperparameters related to the learning process are discussed in the final part of this section.

4.1 Network Architecture

A complete diagram of the network's architecture considering $N = 18$ and $M = 202$ can be found in figure 2. The model's input is a unidimensional tensor with M elements, each representing the local target lift at their corresponding stations across the span. In order to best preserve the spatial structure of the data during convolutional downsampling, 1D layers are used exclusively in the network. The broad idea is to use convolutional layers to extract local features across the target span and map them to each actuator. Fully connected layers can then be used to model the effect of each actuator on the whole span, a key phenomenon in finite wing aerodynamics. The model's first layer, **conv1**, is convolutional with unitary stride, kernel size of 5 elements and 2 elements of padding to avoid downsampling. This initial layer feeds 8 channels of data directly into a ReLU activation layer which in turn is followed by the downsampling layer sequence. The downsampling layer sequence is made up of convolutional layers **ds0**, **ds1**, **ds2**, etc. Each layer maintains the 5-element and 2-element kernel and padding sizes. However, a 2-element stride is used in combination with a doubling in channels to effectively halve the elements in the spanwise direction without information loss. The number of downsampling layers in this sequence is determined by the following expression:

$$n = \lfloor \log_2(M/N) \rfloor$$

This ensures that the size of the final spanwise direction after downsampling is still larger than the network's output size (N). This prevents the upcoming pooling stage, **avgPool** from having to upsample. The downsampling layers feed into a convolutional layer **conv2** with kernel and padding sizes of 5 and 2 respectively. This layer does not result in any change in the tensor shape and is followed by the aforementioned pooling layer **avgPool** which performs Adaptive Average Pooling to essentially



Input
$1\times M$
conv1+ReLU
$8\times M$
ds0+ReLU
$16\times\lceil\frac{M}{2}\rceil$
ds1+ReLU
$32\times\lceil\frac{M}{4}\rceil$
ds2+ReLU
$64\times\lceil\frac{M}{8}\rceil$
conv2+ReLU
$64\times\lceil\frac{M}{8}\rceil$
avgPool
$64\times N$
chCollapse
$8\times N\rightarrow$ReLU$\rightarrow 1\times N$
fc1+ReLU
$1\times 2N$
fc2
$1\times N$
$\mathcal{N}(\mu,\,\sigma^2 I)$
Output $\in\mathbb{R}^N$

Figure 2: NN architecture

interpolate its input into a N sized tensor. After the pooling step, the data tensor is flattened by the two-staged **chCollapse** sequence comprised by two fully connected layers with a ReLU activation layer in between them. The channel reduction is carried out with a rounded square ratio, meaning that the depth d in the input tensor to **chCollapse** will first be flattened to $\sqrt{d}$ and then subsequently to 1 after the intermediate activation stage. The flattening sequence feeds into the last layers of the network, a pair of fully connected layers **fc1** and **fc2** with another intermediate ReLU activation layer. The first layer in this pair duplicates the size of the data tensor while the second one halves it to arrive to the final output shape of $1 \times N$. Actions are sampled using a diagonal gaussian distribution with the mean at the network's output and the diagonal variance vector defined by e^{ϕ} where $\phi \in \theta$. In order to bound the sampled action to the valid action state, the following transformation is applied:

$$a_t = \frac{\Omega z_t}{1 + |z_t|} + \omega_0$$

Where z_t is the action vector sampled from the gaussian distribution. it should be noted that this solution was chosen over a more typical transformation such as $\tanh(x)$ due to the more even function density within the action range.

4.2 Training and Data Loading

As mentioned in section 3, each simulated timestep during agent training is an isolated trial where a new target lift distribution is provided to the agent. Each of these distributions is generated randomly in the simulation's random optimizer module. To ensure a physically valid and attainable target, a preliminary simulation is run to compute the maximum and minimum lifts possible in the system (which will correspond to every actuator at the maximum or minimum position respectively). These lift curves are stored as the upper and lower distribution masks. The random optimizer will generate a random number of points from the range $\left[\frac{N}{2}, 2N\right]$ and place them randomly along the span, ensuring that there is one point at either end. A random value $\in [-1, 1]$ is selected for both endpoints. The remaining points will be filled from the outside in; the linearly interpolated point between the two closest already set bounding points is used as the mean for a squashed normal distribution from which a value is sampled. The standard deviation used to sample from a distribution is σ_b weighed by $\frac{d_{\min}}{b}$ where $d_{\min}$ is the spanwise distance to the closest bounding point. Distribution squashing is used to bound the sampled values to the same $[-1, 1]$ range. This approach ensures a continuous distribution. The upper and lower lift masks provided to the random optimizer serve a vector of valid lift ranges $[l_l, l_u]$ and the random curves generated can be mapped to the valid range by interpolating the corresponding range at each of its points. The result is a randomly generated, physically possible target lift curve whose difficulty can be adjusted by changing σ_b. The same upper and lower lift masks are used to calculate the total possible lift distance across the span which is then used to scale every target distribution received by the agent. This allows the agent to work with a more comfortable range of $[-1, 1]$ local lift values rather than numbers in the order of $[kN]$. The same scaling is performed on s'_t for consistency.

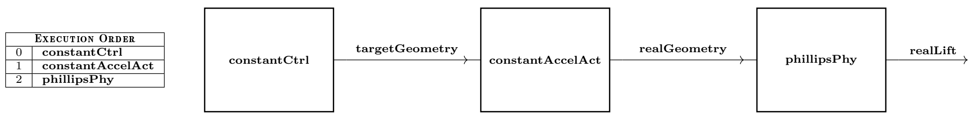


Figure 3: Simulation setup for preliminary run

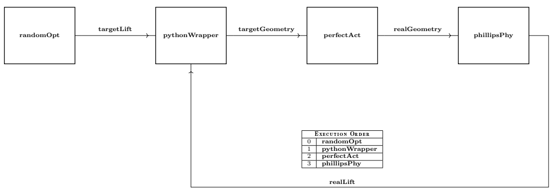


Figure 4: Simulation setup for training run

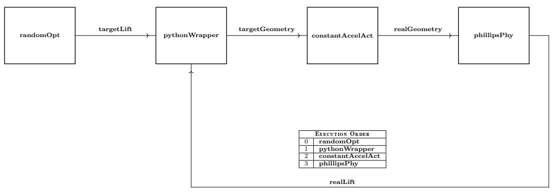


Figure 5: Simulation setup for testing run

The agent's reward is estimated by taking the absolute value of the difference between s'_t and s_t and integrating it over the span using a trapezoidal approximation. Training is done in batches with a gradient approximation and a parameter update after each batch. For each batch, reward is normalized subtracting the mean of all the rewards obtained in the batch and dividing it by its standard deviation. This allows the agent to receive both reinforcement and diminishment in sensible quantities when computing the gradient. It should also be mentioned that due to the simulation framework's signal-based nature, each module is stepped in a set sequence, and thus the agent will not have access to s'_t until the next timestep. To solve this problem, the agent first computes and stores the reward corresponding to the previous step before producing the current step's action. A simple implementation of this solution introduces a vector misalignment issue between the state, action and reward vectors for the current batch. To avoid this problem, a timestep is dropped.

4.3 Simulation

The simulation was configured and run using the UASISI simulation framework developed alongside this project. This is a signal-based framework written in C++ which is optimized for problems involving wings. Specifically, it provides classes that facilitate representing different data types distributed over a wing and storing its corresponding coordinates. UASISI is so far a collection of libraries which means that every simulation has to be compiled before execution. However, UASISI is also quite modular which allows for a wide range of modules to be implemented. Particularly, the **pythonWrapper** module allows the interpretation of python code and its interfacing with the rest of the system using pybind11's embedding capabilities (Jakob, Rhinelander, and Moldovan 2017). Unlike the rest of the modules and the framework, including the main simulation `.cpp` files, changes in python scripts used by **pythonWrapper** do not require compilation to take effect. In this case, the agent was implemented in a python script using torch's pytorch library(Paszke et al. 2019). A diagram of the modules used in each simulation, as well as the stepping order is provided in figures 3, 4 and 5. Additionally, UASISI's source code and the configuration used to run the experiments in this report is available on github.[1]

4.4 Hyperparameters

The hyperparameters used during training and testing for this project, as well as their corresponding macros used in code are laid out in table 3. It is worth clearing up that the parameter `BATCHESPERTARGET` sets the frequency in which a new target is generated by the optimizer. If set to 0, a new target will be provided every step. During experimentation, this resulted in poor performance and low convergence and thus, it was set to change with every new batch. However, the agent still sees each step as a fresh trial. It is also noteworthy that the batch size for testing is not specified. This is because testing was performed with a simulator module that simulated actuator dynamics. Such a module was also used during the preliminary simulation run and the maximum actuator settling time found in the system was recorded. Its equivalent in steps was used as the testing batch size. With the settings utilized, $t_s = 1.762$ $[s]$ which is the duration in simulation time of each testing batch.

[1]`https://github.com/Bel16ario/uasisi`

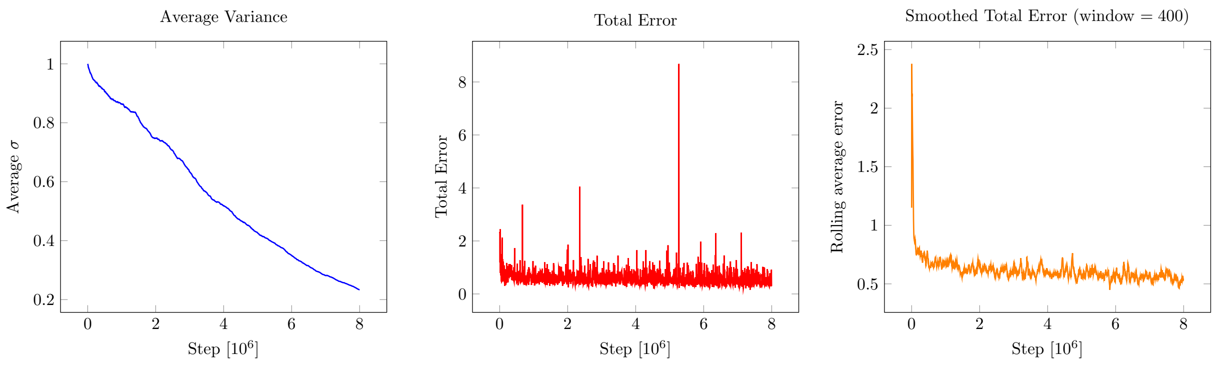


Figure 6: Agent training metrics

HYPERPARAMETER	MACRO/SYMBOL	VALUE
Batch size (training)	`UPDR`	400
Batches trained	`NUPDSTRAIN`	20,000
Batches tested	`NUPDSTEST`	30
Batches per target	`BATCHESPERTARGET`	1
Timestep size	`STPSZ`	1 [ms]
Learning rate	lr	0.001
Initial log sigma	ϕ	0
Optimizer	-	Adam (Kingma and Ba 2014)

Table 3: Hyperparameters

5 Experiments

This section presents the results obtained during training and testing of the agent. Training convergence is evaluated through the evolution of the variance vector $\sigma = e^{\phi}$ in the model as well as the value of the total error computed in the reward function. In contrast, testing performance is evaluated visually, identifying strengths and pitfalls of the trained policy. A study in the the impact of the number of batches per target on the training convergence of the agent is also put forth.

5.1 Training convergence

Figure 6 presents the results from training the agent for 20,000 parameter updates totaling 8 million simulation steps. The average σ value shows how certain the agent is in choosing a particular action. In this case, we see a steady decay in the average σ value from an initial value of 1 to just over 0.2 at the end of training, indicating that the agent is steadily becoming more confident its action selection and less interested in further exploration. The total error signal is quite noisy due to the sheer volume of training data produced. Since each parameter update is effectuated using one batch of data corresponding to a single target distribution, it follows that a particularly difficult target will cause the error to unavoidably spike. Training data was logged every 100 steps and sampled every 50 to create the plots. Downsampling the data even further by taking a rolling average with a 400 element window, a clear pattern is visible. The error exhibited by the agent is slowly decreasing after a steep drop during the first tens of thousands of steps. All of these plots seem to indicate that the agent is gradually converging towards an optimal policy. However, it must also be mentioned that there are diminishing returns on further training and policy optimization is quite slow.

5.2 Batches per target

The effects of the number of batches per target on training convergence were studied by recording shorter training runs with different `BATCHESPERTARGET` settings. Specifically, 100, 10 and 1 batches per target were trialed as well as a new target every step. The results for the batch-per-target case are omitted since the data is available in figure 6 with a much longer horizon. Figures 7 and 8 show the agent's training metrics for each of the remaining cases. It is evident that a higher amount of batches per target results

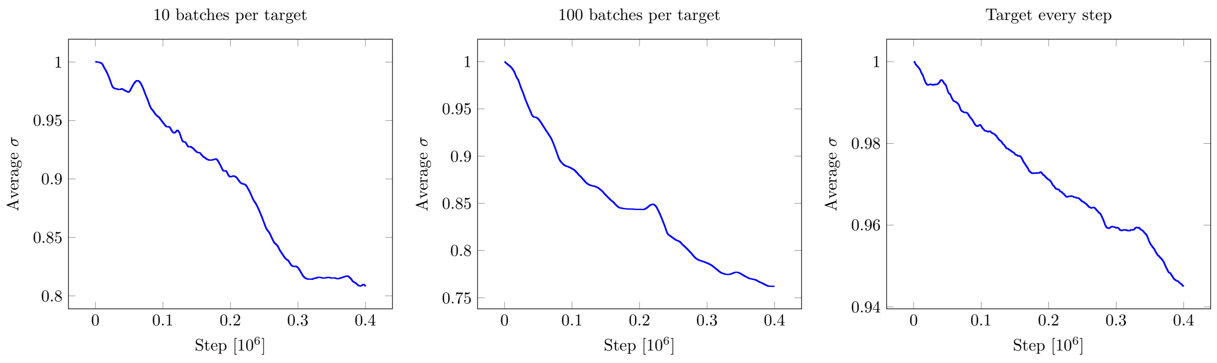


Figure 7: Ablation impact on variance evolution

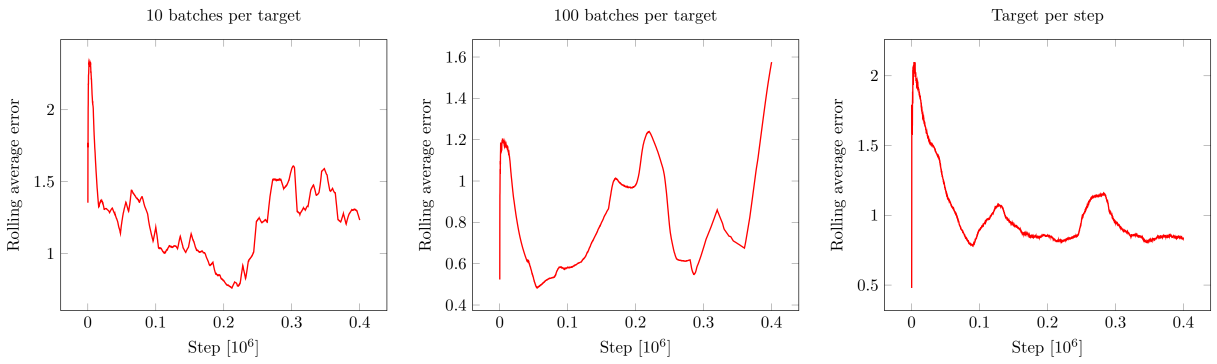


Figure 8: Ablation impact on error evolution (window = 400)

in a faster variance decay as the agent masters the given distribution and becomes very confident. In the relatively short training period of 1000 parameter updates, the agent which receives a new target every step does not manage to reduce its variance past a sigma value of 0.94. In contrast, a batch per target reaches about 0.91, 10 batches per target gets down to approximately 0.82 and 100 batches per target manages to drop to 0.76. Looking at the smoothed error evolution, the opposite pattern emerges and fewer batches per target seem to produce less noisy error. This can be explained by agents overfitting to a particular target and being unprepared to process unknown targets. A batch per target seems to be the best compromise and it was thus selected for the project.

5.3 Testing

A test simulation run was carried out with the trained model and introducing actuator dynamics. The model was placed in evaluation mode and the output diagonal gaussian layer was disabled to ensure determinism. The agent was put through 30 trials and enough time was given to fully move every actuator before a new target was introduced. Some selected data frames from this testing run are presented in figure 9. These frames correspond to the last step in their respective target. The corresponding torsion vectors that generate these distributions are shown in figure 10. Visually, it is apparent that the agent manages to track the target lift distribution and reduce the error signal. Also of interest is the zig-zag learned by the agent that show up in all of the data frames. This is quite unconventional and usually undesired in the aerospace field but with no smoothness penalty, the agent has no incentive to generate smooth geometry.

6 Discussion

The results shown in the previous section show evidence of the feasibility of RL approaches to inverse aerodynamic mapping. A basic policy optimization method such as REINFORCE was succesful in approximating the geometry needed to produce arbitrary lift distributions. However, there are significant limitations to these findings as well as a lack of defined metrics to properly quantify performance. Some

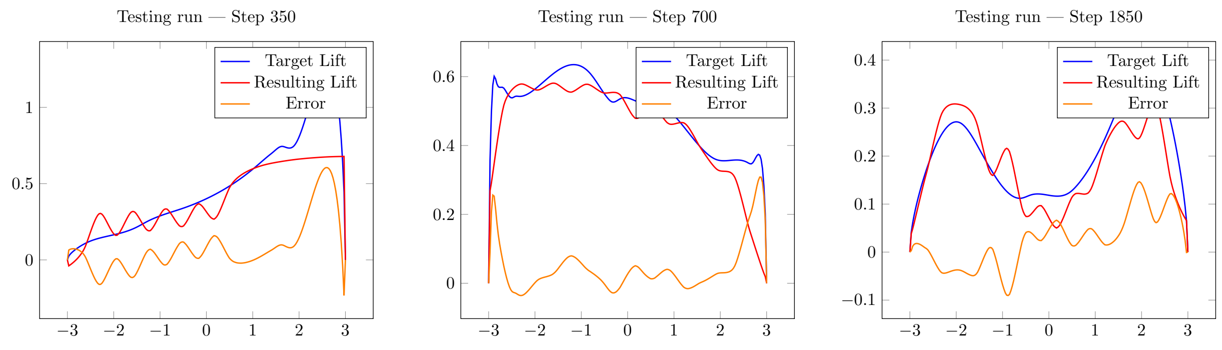


Figure 9: Target and real lift distributions along with their error signal.

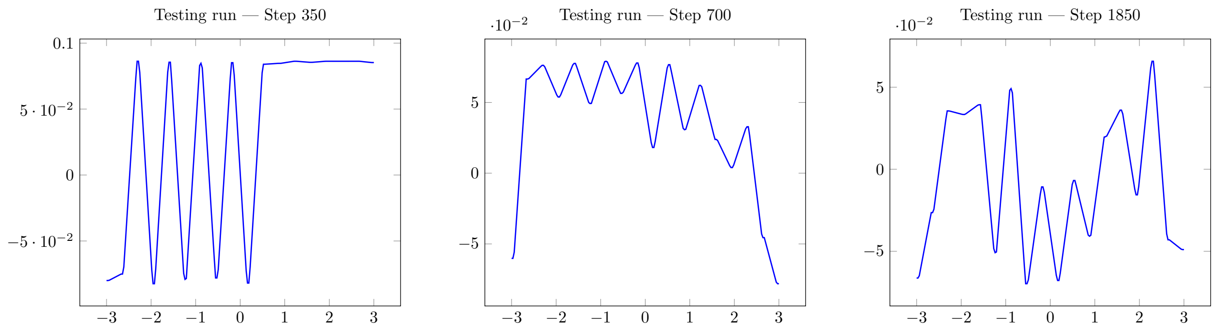


Figure 10: Actions exerted by the agent. Y-axis is in radians

of these limitations are discussed in this section, followed by possible improvements to agent performance and a discussion on the scope of these findings.

6.1 Limitations

There are numerous limitations to the agent trained during this project relating not only to the aerodynamic aspect but also some ML considerations. Firstly, when it comes to aerodynamic limitations, it must be clearly stated that the physical simulation used to train and test the agent depends on a plethora of assumptions. Chiefly amongst them is steady flow conditions with a constant Reynolds' number and no spanwise velocity component. Furthermore, aeroelastic effects caused by the exerted lift are also ignored and even the dynamic actuator simulation is highly idealised which prevents any oscillations from taking place. In this regard, these assumptions allowed the agent to observe very little of the environment. In practice, these assumptions are illogical because the agent's own action space perturbs them significantly.

From the ML side, this project used a fairly basic implementation of REINFORCE adapted to work with this particular problem. It must be said, however, that standard REINFORCE is quite rudimentary when compared to more recent RL methods and thus can be very limited. This is apparent in figure 6 when looking at the the greatly diminishing returns during training. REINFORCE's limitations are only compounded by the natural difficulty in mapping an intricate target distribution with a limited quantity of actuators . Aditionally, it is also evident that the reward function needs to be reformulated to consider not only the total error present in a state-action pair. Bonuses could be implemented to entice the agent into finding solutions which improve the quality of a lift distribution, like slope matching, and not only its total error. This could prevent the agent from relying on zig-zag geometry to track a target distribution.

6.2 Further work

The biggest contribution possible to improving the applicability of these findings is the proposal of standardized performance metrics which would allow us to quantify an agent's performance across a varying range of flight conditions. It could also be used to determine how well an agent trained using an aerodynamic model such as Phillips' solution to Prandtl's LLT performs using more comprehensive fluid simulation methods, particularly computational fluid dynamics (CFD). Having some of the aforementioned metrics is essential in order to implement changes in the agent that seek to maximise performance. Some clear candidates for agent improvements include the aggregation of flight condition data to the observations received by the agent. The reward function could also be reworked to better track distributions as discussed in the previous subsection. An actor-critic architecture (Mnih et al. 2016) could be implemented where a secondary deterministic network (the critic), learns to predict expected reward from a given state which is then used as a baseline during reward normalization, reducing gradient variance. Learning rate scheduling and forced σ periodic resets could be implemented to attempt to reduce training stagnation and induce further parameter exploration respectively.

Adding temporal-based state observations to the actor's input is another clear avenue of further questioning. However, it must be said that this would turn the problem from an inverse mapping agent into an active controller which would require exhaustive aeroelastic and aeroservoelastic physical modelling in order to become applicable to real-world use cases. A more pragmatic line of research would be to train an identical agent in terms of its NN architecture but using real-world data in a Hardware-in-the-loop setup. Valuable information could then be extracted from analyzing the similarity in the final parameters found in each agent. Another possible extension to this research could be implementing an non-diagonal gaussian distribution at the agent's output during training. The resulting trained covariance matrix could provide a valuable insight into the agent's internal influence mapping. Such a matrix could be directly comparable to the influence matrix computed in the aerodynamic simulation module. A fuzzy-logic mapping approach distilled from the trained covariance matrix could also be benchmarked against the trained agent.

7 Conclusion

During the development of this project, an RL agent using REINFORCE was trained to perform inverse lift mappings on a twisting morphic wing with 18 actuators. Through the training, the agent became capable of tracking target lift distributions purely from learned experience and without any analytical approach to the problem. The agent treated each step like an independent trial and attempted to recreate the received target using its available 18 actuators placed along the span. Reinforcement was provided to the agent based on the total integrated error signal resulting from the target and resulting lift distributions. The model's confidence was tracked through the σ values controlling the output's diagonal gaussian distribution and was found to steadily rise with any training performed. During the training horizon used in this research, this evolving σ vector never stabilized which suggest that further training could yield additional improvement.

The decision to turn this problem into a contextual bandit proved effective, not only in greatly reducing the training time required but also in giving the agent enough information for successful mapping. These mapping capabilities obtained from a contextual bandit translated well when tested in a dynamic actuator simulation. An ablation study on the number of batches per different target during training was performed, where a batch per target was found to be optimal for reducing and stabilizing the mean error resulting from an agent's action. When testing the trained model, this setting also proved to more effectively track lift distributions inside a dynamic simulation. However, there was no metric used to formally quantify this and is merely a visual observation. Another important visual observation is the persistently present patterns created by the agent in its action space resulting from an insufficient reward formulation that only considers total error produced.

The results obtained during this study leave a lot of room for further enquiry. Most importantly, standardized performance metrics are crucial to evaluate and compare the effectiveness of this and other future approaches in tackling the same problem. However, the potential for RL approaches in inverse aerodynamic mapping is undeniable. A multitude of possible improvements to the approach taken in this study are readily available and easily implementable. Chiefly amongst them is an overhaul of the reward function and the implementation of an actor-critic architecture.

References

Achiam, Joshua (2018). *Spinning Up in Deep Reinforcement Learning*. `https://spinningup.openai.com`. Accessed: 2026-02-10.

Belaunzarán Consejo, Luca Eugenio (2025). *UASISI: Simulation Framework for Morphic Wing Control.* `https://github.com/Bel16ario/uasisi`.

International Organization for Standardization (1975). *ISO 2533:1975 — Standard Atmosphere.* URL: `https://www.iso.org/standard/7472.html`.

Jakob, Wenzel, Jason Rhinelander, and Dean Moldovan (2017). *pybind11 – Seamless operability between C++11 and Python.* `https://github.com/pybind/pybind11`.

Kingma, Diederik P. and Jimmy Ba (Dec. 22, 2014). "Adam: A method for stochastic optimization". In: *UvA-DARE (University of Amsterdam).* DOI: `10.48550/arxiv.1412.6980`. URL: `https://dare.uva.nl/personal/pure/en/publications/adam-a-method-for-stochastic-optimization(a20791d3-1aff-464a-8544-268383c33a75).html`.

Mnih, Volodymyr et al. (Feb. 4, 2016). "Asynchronous methods for deep reinforcement learning". In: *arXiv (Cornell University).* DOI: `10.48550/arxiv.1602.01783`. URL: `http://arxiv.org/abs/1602.01783`.

Paszke, Adam et al. (Dec. 3, 2019). "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *arXiv (Cornell University)* 32, pp. 8026–8037. DOI: `10.48550/arxiv.1912.01703`. URL: `http://arxiv.org/abs/1912.01703`.

Phillips, W. F. (Jan. 1, 2004). "Lifting-Line analysis for twisted wings and Washout-Optimized wings". In: *Journal of Aircraft* 41.1, pp. 128–136. DOI: `10.2514/1.262`. URL: `https://doi.org/10.2514/1.262`.

Sutton, Richard S. et al. (1999). "Policy gradient methods for reinforcement learning with function approximation". In: *Proceedings of the 13th International Conference on Neural Information Processing Systems.* NIPS'99. Denver, CO: MIT Press, pp. 1057–1063.

Williams, Ronald J. (May 1, 1992). "Simple statistical gradient-following algorithms for connectionist reinforcement learning". In: *Machine Learning* 8.3-4, pp. 229–256. DOI: `10.1007/bf00992696`. URL: `https://doi.org/10.1007/bf00992696`.