Learning to Fly – Building an Autopilot System based on Neural Networks and Reinforcement Learning

Dissertation for the degree of MSc. in Computer Science at FernUniversität Hagen

Felix Eckstein*

June 2020

Supervisor: Prof. Dr. Wolfram Schiffmann
Lehrgebiet Rechnerarchitektur
Fachbereich Informatik

* *Student of Computer Science at FernUniversität Hagen, Matr.-#: 8161569, eckstein@embedded-engineering.de*

Abstract

This work contributes to the final goal of building an autopilot system based on artificial neural networks. Firstly, an overview is given on the state of the art of reinforcement learning in continuous spaces and the deep deterministic policy gradient (DDPG) algorithm utilized in this work. This is followed by reasoning about the application of reinforcement learning techniques on aircraft control and the formulation of continuous control tasks as Markov decision processes respectively Markov games. Based on this theory, a flexible software framework for experimentation is implemented (`Markov-Pilot`, `https://github.com/opt12/Markov-Pilot`) that supports the definition of multiple tasks in a simulated aircraft environment with multiple reinforcement learning agents. Eventually, experiments were conducted using this software to determine a suitable reward structure for the flight control task definition. Several 3-axes flight controllers were trained using different algorithmic settings. The results are compared with conventional PID control, which was outperformed by one of the trained controllers. A results summary and an outlook on future research desiderata conclude this work.

CONTENTS

1 INTRODUCTION

In commercial airliners, autopilot systems are indispensable to reduce the crew workload. The usage of automated flight control is dictated by the commercial interest to enable only two pilots to safely operate a flight for hours while maintaining safety and comfort for hundreds of passengers. But even in private aircraft, which are oftentimes used for the pure joy of flying with no commercial needs, pilot assistant systems become more and more important. Such systems come either in the form of flight directors guiding the pilot's actions, or in the form of autopilot systems directly acting on the aircraft controls.

Directly intervening autopilot systems become especially important in emergency situations, where the stress level of the (maybe inexperienced) pilot considerably increases and an autopilot system can help to safely bring down passengers and equipment. This thesis belongs to the extended research area on the subject of *Emergency Lading Assistant Systems* that is performed at FernUniversität Hagen, Chair of Computer Architecture[1].

Autopilot systems are well established and work reliably in modern aircraft. Most of these systems are basically built upon proportional-integral-derivative (PID) controllers, that have a sound theoretical basis in systems theory and are well understood – in theory as well as in practice. PID control is for sure the workhorse of industrial control and is usually a good choice in situations where the disturbance on the controlled value stays in a limited range. Arguments in favor of PID control are its simplicity and the minimum amount of data needed.

In a former investigation (cf. [Eckstein, 2018]) the author developed a system leveraging conventional PID controllers to track precalculated trajectories to a safe landing field. It became obvious, that the control is very sensitive to painstakingly tuning the PID controller parameters. This tuning process is done mostly manually and is tedious and error prone. The simplicity of the PID control inevitably leads to some limitations:

- The behavior of the controller has to be tuned regarding the rise time, the admissible overshoot, the settling time and the residual steady-state error. As the PID's tuning parameters have an influence on all of the four behavior objectives, the resulting controller is always a compromise. And this compromise is not easy to figure out.
- PID control provides only linear and symmetric control. This is not a problem as long as the first order Taylor series expansion around the systems set-point is sufficiently accurate. However, in case of greater deviations from the working point, non-linearities or asymmetries in the controlled system, PID controllers yield non-satisfactory results.

1 https://www.fernuni-hagen.de/rechnerarchitektur/en/fas-ela.shtml

- Moreover, PID control is quite susceptible to systems with very distinct time-constants. In the case that the controlled system shows fast dynamic behavior overlaid by slow dynamics, PID control tends to oscillate and to get unstable or to have inacceptable slow settling times.
- The advantage of a PID controller to only need a single deviation measurement as input is also a major disadvantage: It just cannot include more observations into its control actuation even if a lot more helpful information was available.

Of course, there are way more classic control algorithms which are far more advanced than simple PID control. There is a whole plethora of advanced adaptive and model predictive control algorithms and optimal control theory is an entire field of research on its own. But in recent years, this toolkit was amended by a novel tool which is commonly called *Artificial Intelligence* (AI).

Besides the fact, that the ideas behind AI, namely *Artificial Neural Networks* (ANN), are not really that new, but are around for decades, nowadays the hard- and software is available to really leverage this technology and apply it to more advanced problems. The reason, that AI is such a trending buzzword today is not because it was freshly invented, but because it is possible to compute even non-trivial models on comparably common hardware by using more or less user-friendly standard software stacks.

In the current work –instead of conventional PID controllers– artificial neural networks shall be used to control the aircraft. The system shall learn and train its behavior by *reinforcement learning* (RL) techniques in a simulated environment. This work contributes to the final goal of building an autopilot system based on artificial neural networks trained by *machine learning* (ML), namely reinforcement learning techniques.

A learning environment shall be set up, and procedures shall be developed to automatically train machine learning agents. Once set up, this environment shall be used to train a 3-axes flight controller which can eventually be compared to conventional proportional–integral–derivative (PID) based control algorithms.

Of course, ANNs and AI-technology were not only chosen because of its hipness and as artificial intelligence and machine learning are ubiquitous buzzwords, but because ANNs have some specific advantages from flight control might benefit:

- ANNs can cope with a multitude of input parameters. So, a lot more measurements can be fed into the controller to be incorporated into the determination of actions.
- ANNs have non-linear activation functions. This non-linearity makes them well suited to control non-linear or asymmetric processes. The non-linear behavior of ANNs can have a positive influence on the effectiveness of the system control, especially regarding settling time and overshoot. Such non-linearities are also very well suited to handle different time constants which regularly lead to difficulties with PID control.

- To a certain degree, ANNs generalize their trained behavior to previously unseen situations. Such a generalization can, in the best case, adapt to a changed environment. Be it changed aircraft parameters due to damage, adverse weather or even due to icing.
- The most interesting promise of machine learning and ANNs is their self-learning capability. In contrast to the process of manually tuning parameters and models like in classic control, the ANN can be trained in an automated fashion and learn the desired behavior on itself. This training can be performed safely in a simulated environment and even the data collected later on in the field can be used to further refine the ANN based controller.

To train ANN based flight control, the technology of reinforcement learning shall be used. In reinforcement learning, an agent interacts with its (simulated) environment by issuing actions (control commands to the aircraft) to maximize a certain reward function. This reward function is a scalar value that is permanently calculated and judges the quality of the agent's actions. In the case of the controller to be developed, this judgment shall reward the quality of set-point tracking and shall penalize twitchy flight maneuvers.

The calculation of the trajectories to be followed and the resulting set-points is out of scope for this work.

In a naïve (unsupervised) reinforcement learning setup, the agent has no prior knowledge of the environment and the shape of the reward. It just tries out different actions and experiences the resulting consequences with respect to the reward. In the course of this exploration, the agent learns which actions lead to a high reward and which actions shall better be avoided. By exploiting this learned knowledge, the actions get better over time and the agent eventually builds up an internal model of its interactions with the environment. When confronted with different environmental conditions during the learning phase, such a model oftentimes generalizes quite good to unknown situations which were not explicitly presented during the learning phase.

This investigation shall mainly answer two questions:

- How can the problem of aircraft flight-control be translated into a reinforcement learning problem that can benefit form machine learning techniques in automated training?
- Is there a point in using reinforcement learning algorithms to implement low-level flight control? Is it possible to entirely train a 3-axes flight controller without manual parameter tuning but yielding a competitive performance?

The outline of this work is as follows: Before diving into reinforcement learning, the methodology based on a design based research approach is described briefly. The first technical section gives an overview over the state of the art of reinforcement learning in continuous spaces and presents the deep deterministic policy gradient (DDPG)

algorithm used in this work. This is followed by reasoning about the application of reinforcement learning techniques on aircraft control. The subsequent section describes how the, so far implicitly formulated, tentative design is implemented into the `Markov-Pilot` software framework for experimentation. The experiments conducted with this software are described in the experiments section, before a results summary and an outlook on future research desiderata concludes this work.

2 DESIGN METHODOLOGY

Before reviewing the state of the art of RL in continuous state and action spaces, the chosen approach to tackle the problem of applying RL techniques to low-level aircraft control shall be described from a methodology point of view.

The work conducted follows a *Design Science Research* approach with the two main outcomes after [Vaishnavi, Kuechler, and Petter, 2004/19]:

1. the creation of new knowledge through design of novel or innovative artifacts (things or processes) and
2. the analysis of the artifact's use and/or performance with reflection and abstraction.

The individual process steps, from the first idea of applying reinforcement learning to aircraft control to the outcome of this work wrapped up in the conclusion at the end, can be mapped almost directly to the process steps of the *Design Science Research Process Model* shown in figure 2.1.

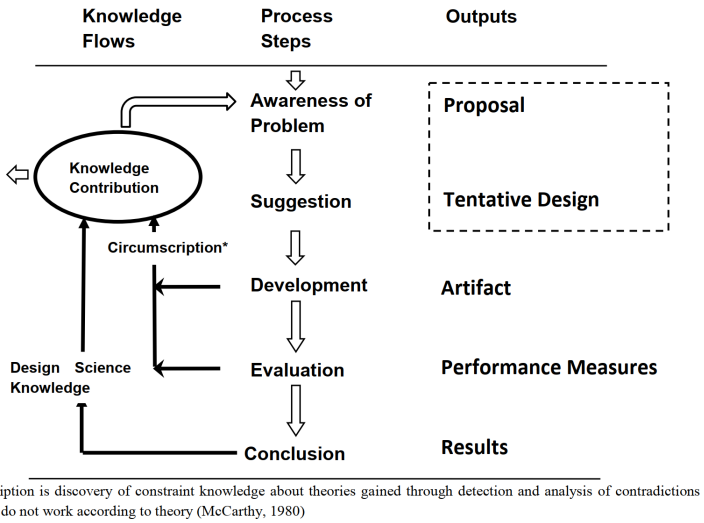

Figure 2.1: Design Science Research Process Model (DSR Cycle) (from [Vaishnavi et al., 2004/19, fig. 3])

The section outline of this thesis is broken down according to these process steps.

AWARENESS OF THE PROBLEM In the introductory section 1, a motivation is given why it seems beneficial to try out alternative means of aircraft control besides the well known conventional control systems.

SUGGESTION Mainly in section 3 and partly in section 4 the candidate algorithm and the necessary adaptations to the problem at hand are described. After an investi-

gation of the existing field of reinforcement learning and the established methods and algorithms, the special needs to apply these algorithms to aircraft control are sketched out. The result of this groundwork is a collection of the building blocks of a tentative design to be embedded in an experimentation framework.

DEVELOPMENT The development phase culminates in a software framework enabling multiple experiments on the application of reinforcement learning to the setting of an aircraft in gliding descent. The architecture and the implementation of the developed software is outlined in section 5. The software being part of this thesis itself is referenced in appendix C. Findings during the development and the following experimentation phase were looped back into section 4 in an iterative manner.

EVALUATION Finally, the developed software framework is utilized to conduct experiments. These experiments serve the purpose to find out about the postulated rationale to apply reinforcement learning methods to aircraft control. The desription and evaluation of the experiments is discussed in section 6.

CONCLUSION In the final section 7, the results are wrapped up and the chosen approach is assessed with respect to the expected outcome. A short discussion of the achieved results and an outlook on future research necessities concludes the work.

3 BASICS OF REINFORCEMENT LEARNING IN CONTINUOUS SPACES – STATE OF THE ART

The problem of low-level aircraft control is characterized by a continuous, multi-dimensional state space and a continuous, multi-dimensional action space. A state of the art algorithm to handle such problems is the *Deep Deterministic Policy Gradient* (DDPG) algorithm introduced in the breakthrough paper [Lillicrap, Hunt, Pritzel, Heess, Erez, Tassa, Silver, and Wierstra, 2015].

Even though there are improvements and other algorithms available which may or may not outperform DDPG (cf. [Duan, Chen, Houthooft, Schulman, and Abbeel, 2016]), this algorithm is used as the basis for the aircraft control experiments in this work as it is widely known and quite understandable. The main focus of this research is not on improving the underlying algorithm, but to find out how the concrete problem of low-level aircraft control can be tackled using RL techniques. In a first step this is algorithm agnostic and different algorithms can be plugged into the developed software later on.

Before going into the peculiarities of RL based aircraft control, the most important basic terms regarding RL and on the principle of the underlying DDPG algorithm shall be sketched out in this introductory section on the state of the art.

Firstly, a recap of the most basic terms of RL is given. This is followed by some general intro into policy gradient based algorithms as opposed to value based algorithms before the *Actor-Critic* principle as a special form of policy gradient algorithms is presented. To handle continuous space problems, function approximation is needed which can be done by utilizing *Artificial Neural Networks* (ANN) for the actor and the critic in DDPG. The different learning objectives for the actor- and the critic-ANN are briefly introduced, before all the parts are assembled together into the DDPG algorithm itself, which is described as the final of this review on the sate of the art.

3.1 Basic Terms

A brief overview over the terms and ideas of *reinforcement learning* will be given in the next paragraphs. This is of course no comprehensive introduction to the field, but merely a quick recap on the basic terms and concepts to establish a common understanding. A thorough introduction can be found in [Sutton and Barto, 2018].

The basic model underlying the idea of reinforcement learning is an *agent* interacting with an *environment*. The agent observes the *state* of the environment, chooses a certain *action* and issues it to the environment which in turn changes into a new state. Alongside with the observed new state, the agent receives some *reward* which rates the desirability to be in the current state. This principle is depicted in figure 3.1

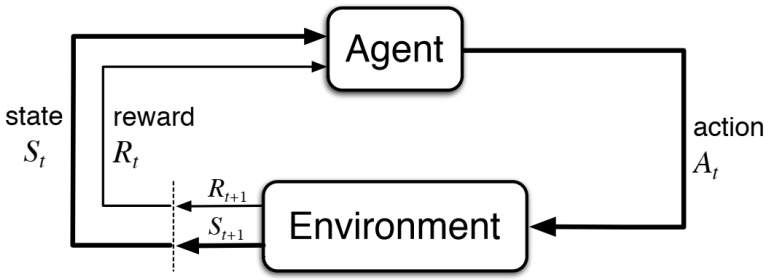

Figure 3.1: The agent-environment interaction in reinforcement learning. (from [Sutton and Barto, 2018, fig. 3.1])

This "State–Action–Reward/new State–Cycle" is repeated over so called *episodes*. Depending on the experiment setup, the episodes may have an arbitrary number of steps which may or may not be known in advance or may even last infinitely. The agent aims to maximize the sum of the received rewards till the end of the episode. To make infinite episodes possible and to express some preference of short term rewards over long term rewards, a *discount factor* $0 \leqslant \gamma \leqslant 1$ is introduced in the summation of rewards. A discount factor $\gamma < 1$ makes early rewards having more impact than future rewards and also makes it possible to deal with infinite episodes which otherwise could lead to infinite rewards.

There are no further restrictions on the reward apart from being a scalar value. The reward in each timestep may be positive, negative or zero. If a reward other than zero is received in each timestep we speak of *dense* rewards opposed to *sparse* rewards which are only non-zero at certain timesteps or at the end of an episode. When selecting actions, the agent shall consider the expected future rewards and shall maximize the overall discounted return over an entire episode.

This leads to the objective of the agent to maximize the return G:

$$\max G = \sum_{t=0}^{T} \gamma^t r_t$$

with T being the final timestep of the episode.

MARKOV DECISION PROCESS (MDP) Mathematically the reinforcement learning problem is modeled as a so called Markov Decision Process (MDP) which is an abstract formalization of the sequential decision making process.

> The Markov decision process model consists of decision epochs, states, actions, rewards, and transition probabilities. Choosing an action in a state generates a

> reward and determines the state at the next decision epoch through a transition probability function." ([Puterman, 1994, p. xv])

I. e. the MDP is defined by a set of states, actions and rewards ($\mathcal{S}$, $\mathcal{A}$ and $\mathcal{R}$) and assigned transition probabilities $p(s_{t+1}, r|s_t, a)$, which reads as the probability to reach a new state $s_{t+1} \in \mathcal{S}$ with its associated reward $r \in \mathcal{R}$ when issuing action $a \in \mathcal{A}$ while being in state $s_T \in \mathcal{S}$.

The interaction between the agent and the environment yields an alternating sequence of states, actions, new states and rewards which is called a *trajectory*.

$S_0, A_0, R_1, S_1, A_1, R2, S_2, A_2, \ldots, R_T, S_T$ with T being the final step in an episode.

The rewards only depend on the new state as a function $r(s_{t+1})$. The transition to this new state is stochastic and is given by the transition probability $p(s_{t+1}, a|s_t, a)$ which is an inherent characteristic of the environment. The reward is independent from the trajectory leading to s_{t+1} which puts a restriction on the state s to "include all information about all aspects in the past agent-environment interaction that make a difference for the future." [Sutton and Barto, 2018, p. 50]. This restriction is known as the *Markov Proptety* which must be satisfied when formulating a problem as an MDP amenable to reinforcement learning.

STATE VALUE AND THE BELLMANN EQUATION The agent's action selection when being in a state s is defined by its *policy* $\pi(s) \to a$. The reinforcement learning problem is basically the problem of finding a (near) optimal policy for the agent to choose the best action a in each state s of the environment. To this end, the agent must have a notion of the rewards it can expect from the states it will visit in the future till the end of the episode. Mathematically this notion can be formulated as the *value* $v(s)$ of each state. Leaving aside the details resulting from the stochasticity of the action selection as well as the state transition,[2] the *state value* is defined recursively as

$$v(s_t) = r + \gamma v(s_{t+1})$$

This equation is called the *Bellman equation* for state values and has the following meaning: The value of a state is determined by the reward r received after issuing an action a and the discounted value $v(s_{t+1})$ of the subsequent state.

As the agent directly influences the future by its action selection policy, the equation is valid only for a certain agent policy and should be interpreted as $v_\pi(s)$. The index π is oftentimes left out to simplify notation.

2 In general, the state transition $s_t \xrightarrow{a} s_{t+1}$ as well as the action selection by the policy $\pi(s) \to a$ are stochastic processes which lead to slightly more complications in the equations. This is detailed out in the pertinent literature, but is left out for this brief overview.
However it is important to know about the stochastic nature of these processes, as in a lot of papers and algorithms, the state values are not given directly, but as expectancies.

The goal of all reinforcement learning is to find an optimal policy π^* that maximizes the value of each state and finds a trajectory of states, actions and rewards to collect this maximum value as the agent's return $G = \sum_{t=0}^{T} \gamma^t r_t$.

3.2 Value Functions and Policy

In a similar way to the state-value function $v(s)$, the action-value $Q(s, a)$ is defined as the expected reward till the end of an episode when starting in state s_t and issuing action a_t:

$$Q_\pi(s_t, a_t) = \sum_{\substack{s_{t+1} \in \mathcal{S} \\ r \in \mathcal{R}}} p(s_{t+1}, r | s_t, a_t) \left[r + \gamma Q_\pi(s_{t+1}, a_{t+1}) \right]$$

While the Bellman equation for the the state-value is immediately obvious, the Bellman equation for the action value is slightly more complicated due to the fact that the state transitions from $s_t \to s_{t+1}$ are a stochastic process.

As said above, the just defined state- and action-value functions depend on the policy π the agent follows. For at least one policy, an optimal policy π^*, the value functions are maximized:

$$v^*(s) = \max_\pi v_\pi(s), \qquad Q^*(s, a) = \max_\pi Q_\pi(s, a)$$

One possibility to tackle RL problems is to learn the optimal action-values assigned to the underlying environment. When these optimal Q-values for all states and actions are known, the optimal agent's policy π^*can be defined greedily to always chose the action with maximum $Q(s, a)$ as

$$a_t = \arg\max_{a \in \mathcal{A}} Q(s_t, a)$$

3.3 Policy Gradient Methods

Presupposed the action-value function in an environment is learned, the greedy policy selection using $\arg\max_{a \in \mathcal{A}}$ is obvious as long as the action space $\mathcal{A}$ is discrete and finite. As soon as however, the action space becomes continuous (or at least sufficiently big), this value based action selection turns into a hard optimization problem of its own to find the arg max in each step.

That's why value based methods collapse in problems with continuous action spaces. But what other methods could be used to select the agent's action? Instead of defining the policy by taking the indirection via the value function, another class of algorithms can deal with a suitable representation of the policy directly. These policy based methods directly optimize the performance of a policy.

Let the policy π be defined in a general stochastic form as the probability distribution $\pi(a|s,\theta) \to [0,1]$ stating the probability of taking action a when being in state s. θ is the parameter vector defining the policy function to be optimized in order to maximize the policy's performance $J(\pi)$ (cf. [van Hasselt, 2012])

This performance is defined as the expected cumulative return when starting in state s_t and following policy π:

$$J(\pi, s_t) = v_\pi(s_t)$$

The class of so called *policy gradient* algorithms aims to maximize this expected return by gradient ascent in the parameter space of θ. This leads to the update rule for the parameters like:

$$\theta_{k+1} = \theta_k + \alpha \nabla_\theta J(\pi) = \theta_k + \alpha \nabla_\theta \mathbb{E}\left[v_\pi(s_t)\right] = \theta_k + \alpha \nabla_\theta \int_{s \in \mathcal{S}} p(s_t = s) v(s) ds$$

with α being a hyper-parameter called the learning rate and $p(s_t = s)$ being the probability of being in state s_t at time t. As the policy itself has an unknown influence on this state distribution $p(s)$ it is unclear how to handle its derivative $\nabla_\theta p(s)$ in the above term. Fortunately, the result of a lengthy derivation in [Sutton and Barto, 2018, §13] yields as a result the *Policy Gradient Theorem* stating that the derivative of the state distribution is not needed but that the policy gradient can be written as:

$$\nabla_\theta J(\pi) = \int_{s \in \mathcal{S}} p(s) \mathbb{E}\left[\nabla_\theta \log \pi(a|s,\theta) Q_\pi(s,a)\right] ds$$

The absence of a term in the form $\nabla_\theta p(s)$ makes this policy gradient amenable to stochastic sampling from experience, learned by using the policy to be optimized. This is elaborated in [Sutton, McAllester, Singh, and Mansour, 1999] where the authors also show that the normally unknown action-value $Q_\pi(s,a)$ may be replaced by a learned approximation. While the mentioned paper proves this result for the case of stochastic policies $\pi(a|s,\theta) \to [0,1]$ yielding a probability for each action and not the action itself, this finding is extended to deterministic policies of the form $a = \mu(s|\theta)$ in [Silver, Lever, Heess, Degris, Wierstra, and Riedmiller, 2014]. This comes in handy in the DDPG algorithm described below.

3.4 Actor-Critic Algorithms

One may argue that this again uses the value function that was supposed to be left behind with policy gradient methods. That's true, but the value function is used in a totally different way now: It's not used any more to define the policy function itself, but rather to determine the direction of maximum ascent (the gradient) in the policy's parameter space.

Nevertheless, the value of $Q(s, a)$ must be estimated somehow. There are basically two different ways of obtaining the Q-values that go into the policy update:

The straight forward way is to play an episode following the current policy π and record the transitions and rewards. At the end of the episode the actual reward values earned from each step can be summed up and be used as the $Q(s, a)$ action-values in the update step of the policy.

Another alternative is to separate the policy calculation from the value calculation. The agent is split into two entities: The *actor* calculating the policy and thus determining the actions, and another entity, the *critic*, which learns to estimate the $Q(s, a)$ action values independent from the currently played episode. The critic assesses the actor's actions by comparing their actual outcome with its estimation of $Q(s, a)$. This information goes into the policy update to encourage the actor's policy to issue beneficial actions.

The critic's update procedure is postponed till section 3.6. The take-away for now is that algorithms with different entities for the policy calculation and the value estimation are called *actor-critic*-algorithms.

3.5 Function Approximation using Artificial Neural Networks

In recent years, the use of *artificial neural networks* (ANN) for multi-dimensional, non-linear function approximation became a standard technology. ANNs are around for decades and have a sound theoretical foundation. Already in the eighties of the last century it was proven that feed forward ANNs with at least one hidden layer and non-linear activation can approximate any continuous function to any degree of accuracy (cf. [Cybenko, 1989] after [Sutton and Barto, 2018, §9.7]). But only in recent years, since the availability of massively parallel compute power –namely Graphics Processing Units (GPUs)– at a reasonable price in conjunction with heavyweight toolsets like Tensorflow (cf. `https://www.tensorflow.org/`) or PyTorch (cf. `https://pytorch.org/`), the use of ANNs for learning tasks really got off the ground and is widely used nowadays.

The standard training of ANNs to learn an approximation for a function $f(x)$ is done by example in a supervised setting: An input value x is fed into the ANN and the output $\hat{f}(x, \mathbf{w})$ is compared with the expected target result $f(x) = y$. The parameters $\mathbf{w}$ of the ANN's forward function are the weight parameters to be adjusted to approximate $f(x)$ as close as possible. As a measure of accurracy of the approximation, the loss $\mathcal{L}$ is computed as the square error from the approximation $\hat{f}(x, \mathbf{w})$ to the target y: $\mathcal{L} = (\hat{f}(x, \mathbf{w}) - y)^2$. Now the network weights $\mathbf{w}$ are adjusted to minimize this loss.

The adjustment process used is known in the literature as *stochastic gradient descent* (SGD) (cf. [Sutton and Barto, 2018, §9.3]): To adjust the weights, the gradient of the loss with respect to the weight vector is calculated and the weights are pushed towards

the negative direction of this gradient. This is the direction in which the loss error falls most quickly. The update rule at timestep t with learning rate[3] $\frac{1}{2}\alpha$ is given as

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \frac{1}{2}\alpha\nabla_{\mathbf{w}}\mathcal{L} = \mathbf{w}_t - \frac{1}{2}\alpha\nabla_{\mathbf{w}}\left[\hat{f}(x, \mathbf{w}_t) - y\right]^2 = \mathbf{w}_t + \alpha\left[\hat{f}(x, \mathbf{w}_t) - y\right]\nabla_{\mathbf{w}}\hat{f}(x, \mathbf{w}_t)$$

Technically, this procedure is not only done with a single example, but with an entire set of examples (a so called minibatch) at once. For good convergence, it is necessary, that these samples are independent and identically distributed (i. i. d.) over the entire space of possible inputs. That's why the minibatch is randomly sampled over a wide range of possible inputs.

The loss is averaged over this minibatch and only small steps, determined by the learning rate α are taken into the direction of the negative gradient. In practice, there are more sophisticated methods available like described in [Ruder, 2017] and [Kingma and Ba, 2017], but all of them build on the shown principle of stochastic gradient descent.

In recent reinforcement learning settings, such ANNs are used to approximate the actor's policy function as well as the critic's state-value estimation.

3.6 Deep Deterministic Policy Gradient Algorithm for continuous Control

All the above mentioned building blocks shall now be brought together to the *Deep Deterministic Policy Gradient* (DDPG) algorithm for continuous control which was introduced in [Lillicrap et al., 2015]. This algorithm is relatively simple to implement and excels in being robust regarding its hyperparameters: In the original paper, it was applied to more than 20 diverse simulated physics tasks with the same hyperparameter settings and yielded a good overall performance for all of them. This makes it the right choice for our experiments on aircraft control, as it sets the stage for investigating the specialties of RL usage in aircraft control without the fear of loosing focus while struggling with hyperparameter tuning.

DDPG is an actor-critic policy gradient algorithm. It utilizes ANNs to learn an optimal policy for continuous control tasks. In contrast to earlier policy gradient algorithms (cf. [van Hasselt, 2012]) which output a probability distribution for the action selection that is then sampled to get the actual action, the DDPG actor deterministically yields an action which is directly fed into the environment as a control signal. The action evaluation by the critic is then used to improve the actor's policy.

The DDPG can be trained off-policy i. e. the examples used for training can be collected using a different policy than the one currently trained. This gives rise to the

3 As the learning rate α is an arbitrarily chosen parameter, the factor $\frac{1}{2}$ was introduced to make the end result looking nicer as the factor falls out after application of the chain rule for the gradient calculation.

idea of explicit exploration during training: As the outputs of the actor in DDPG are deterministic, there is no inherent exploration due to the stochasticity of a sampling process taking place to formerly unseen areas of the state-action space. To mitigate this lack of exploration, the policy during the training phase is explicitly overlayed with a noise signal disturbing the original output of the trained actor. Due to the algorithm's ability of off-policy learning, this added noise is not an unwanted disturbance, but a solution to the exploration/exploitation dilemma.

The DDPG algorithm trains on experience collected by interaction of the agent with the environment. The actor $\mu(s)$ issues actions according to its initially random policy and the resulting experience is stored in a large replay buffer. The i^{th}-experience stored in this buffer is the tuple (s_i, a_i, r_i, s_{i+1}) of state, action, reward and the next state. During training, batches of this buffer are sampled and used as training examples to improve the critic's and the actor's ANNs.

The critic Q is realized as a neural network estimating the state-action-value function inherent to the environment. This network is parameterized by its network parameters θ^Q. For each input combination of state and action, it outputs a single Q-value as an estimation of the true state-action-value for this point in the state space. The training of the critic's ANN follows standard SGD of supervised learning: A minibatch of input is forward processed by the ANN and the resulting output is compared with the desired target y. In a backward step, the gradient $\nabla_{\theta^Q} Q$ of the output wrt. the network parameters is calculated. The loss $\mathcal{L}$, which is the mean square error between ANN output and target, is then used to push the network weights in descending direction to minimize this very loss. The update step at time t towards the target y is like

$$\theta^Q_{t+1} = \theta^Q_t + \alpha_{\text{critic}} \mathcal{L} \nabla_{\theta^Q} Q = \theta^Q_t + \alpha_{\text{critic}} \left[Q(s_i, a_i) - y_i \right] \nabla_{\theta^Q} Q$$

The learning rate α_{critic} is a hyperparameter of the algorithm and y_i is the desired target value for Q.

Still open is the question how this target y_i is determined. In DDPG the target values are estimated through a 1-step unrolling of the Bellman equation using the reward r_i gained in the current experience step and dedicated target ANNs to estimate the Q-value and the action of the "next" step.

$$y_i = r_i + \gamma Q'(s_{i+1}, a_{i+1}) = r + \gamma Q'(s_{i+1}, \mu'(s_{i+1}))$$

So called target networks for the actor μ' and the critic Q' are used to determine the training targets for the actor Q and critic μ under training. These target networks are not trained themselves, but track the trained networks' parameters using a delayed soft-update. This concept of delayed target networks introducing stability into the training process was presented in [Mnih, Kavukcuoglu, Silver, Rusu, Veness, Bellemare, Graves, Riedmiller, Fidjeland, Ostrovski, Petersen, Beattie, Sadik, Antonoglou, King, Kumaran, Wierstra, Legg, and Hassabis, 2015] and is one of the standard procedures in deep RL.

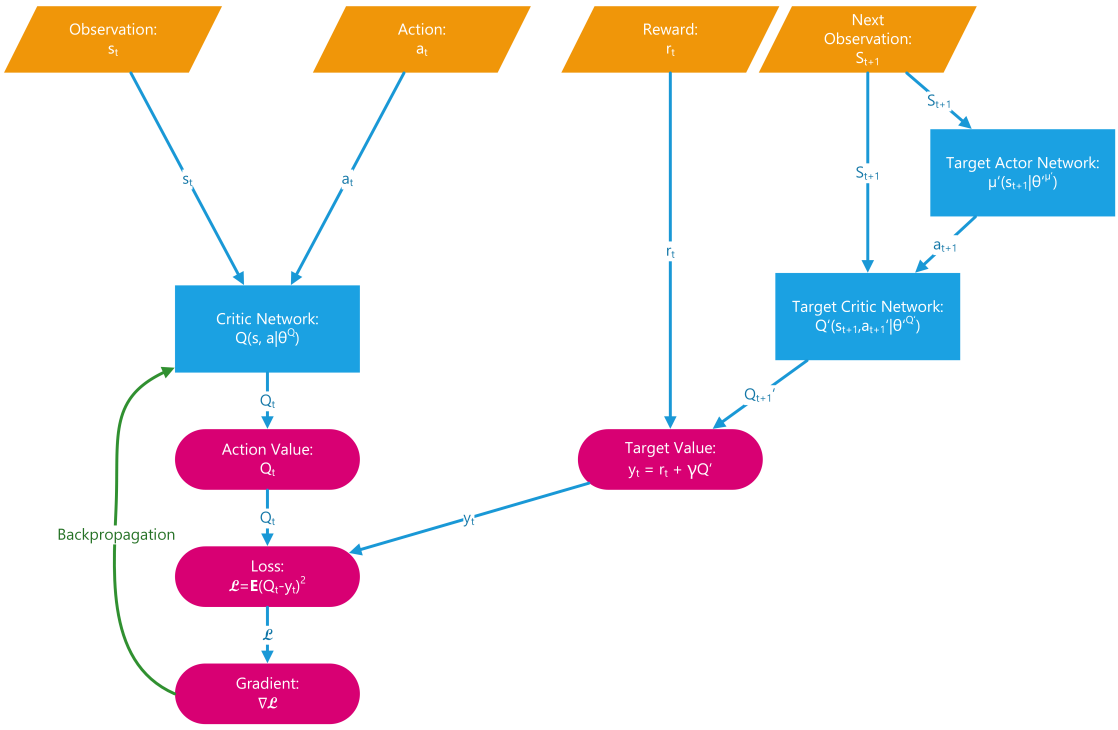

Figure 3.2: Networks involved in the training of the critic in DDPG, (own visualization)

The critic training with the involved networks is depicted in figure 3.2.

The actor μ is also realized as an ANN, directly calculating the desired action output a from the input state s as $a = \mu(s|\theta^{\mu})$. In contrast to the critic, whose network is updated by SGD on the loss, the critic is updated to maximize it's performance $J(\theta^{\mu})$ which is defined as the expectancy of the return when following the policy μ defined by the network's parameters θ^{μ}.

$$J(\theta^{\mu}) = \mathbb{E}\left[Q(s, a)\right] = \mathbb{E}\left[Q(s, \mu_{\theta^{\mu}}(s|\theta^{\mu}))\right]$$

To this end, the gradient $\nabla_{\theta^{\mu}}$ must be climbed upwards in each training step. The policy gradient is given by[4]

$$\nabla_{\theta^{\mu}} J = \mathbb{E}\left[\nabla_{\theta^{\mu}} Q(s, a|\theta^{Q})\right] = \mathbb{E}\left[\nabla_{a} Q(s, a|\theta^{Q}) \nabla_{\theta^{\mu}} \mu(s, |\theta^{\mu})\right]$$

For training the actor, the expectancy $\mathbb{E}$ is replaced by the mean over samples of a minibatch from the experience replay buffer which leads to the update rule

$$\theta^{\mu}_{t+1} = \theta^{\mu}_{t} + \alpha_{\text{actor}} \nabla_{\theta^{\mu}} J$$

The actor training with the involved networks is depicted in figure 3.3.

4 The presented notation is simplified by leaving out the integration over the distributions of the start states. For an extensive write up including the proof for existance of the deterministic policy gradient see [Silver et al., 2014] and [Lillicrap et al., 2015].

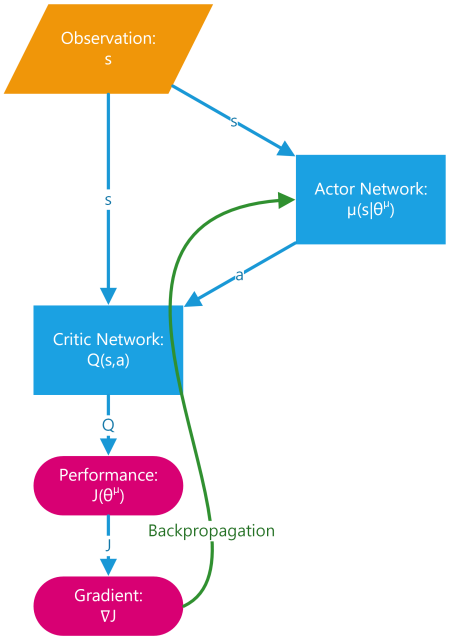

Figure 3.3: Networks involved in the training of the actor in DDPG, (own visualization)

When looking at the equations and the diagrams, it becomes obvious, that the key to optimize the policy is a good approximation of the action values Q by the critic, as this is the defining element in the policy gradient used for updating the actor.

After the training steps of the critic and the actor, the associated target networks are updated by a soft update pulling them a small bit defined by the transfer factor τ –which is another hyper-parameter in the algorithm– towards the direction of the trained networks:

$$\theta^{Q'} \leftarrow \tau\theta^{Q} + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^{\mu} + (1-\tau)\theta^{\mu'}$$

The entire DDPG algorithm, which is applicable to continuous state and action spaces, is given in Appendix A in figure A.1.

4 APPLICATION OF REINFORCEMENT LEARNING TO AIRCRAFT CONTROL

After this general review of a state of the art algorithm on RL in continuous action and state spaces, it's time to reason about the special needs and requirements to apply the described methods to aircraft control. This section starts with an introduction to aircraft flight control in general followed by the formulation of the control tasks in a form amenable to RL. At the end of the section some criteria to assess the quality of a developed control strategy are given.

4.1 Aircraft Flight Control

For the research at hand, the general term *aircraft* shall be identified with its special variation of a *fixed-wing airplane*. In contrast to e. g. helicopters, such fixed-wing airplanes need a relative forward velocity wrt. the surrounding air to produce lift. As this forward velocity induces drag that inhibits the forward movement of the craft, some external energy is needed to compensate the loss resulting from the drag force. This energy supply must generate a forward force counteracting the drag force. The energy is delivered either from some kind of engine or by converting potential energy into kinetic energy in a gliding descent.[5] Forces acting on a fixed-wing airplane are depicted in figure 4.1.

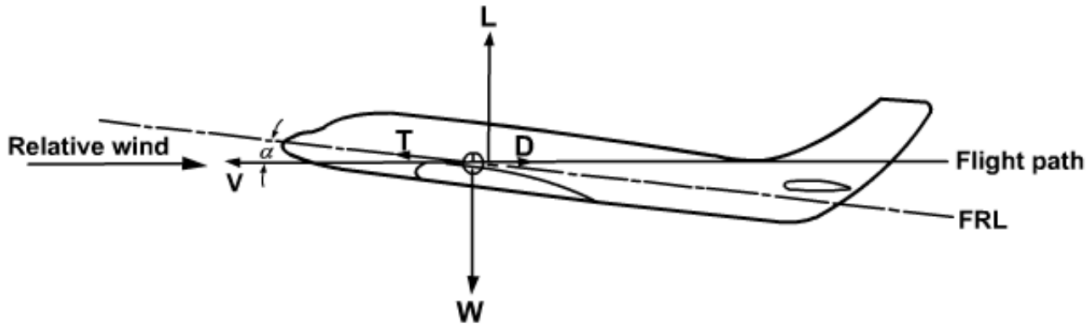

Figure 4.1: Forces on a fixed-wing airplane in level-flight: **L**ift, **W**eight, **D**rag, **T**hrust. Additionally shown are the fuselage reference line, the angle of attack α and the relative velocity. (from [Tulapurkara, 2012, fig. 1.12])

To simplify the problem of aircraft control, the further investigations concentrate on the "engine-off" case of an un-propelled gliding descent of an aircraft, that starts with initial conditions in a stable flight situation. While not being the most general setup

5 Sailplanes, even though un-propelled can of course also be on an ascending flight path. In that case they harvest energy from the surrounding air in the form of thermal lift or from relative up-wind e. g. at a cliff.

by leaving aside thrust control, this situation has special relevance for an emergency-landing approach after engine-failure (cf. [Eckstein, 2018]). The software framework to be developed is flexible enough to include thrust control later on.

Before reasoning about aircraft control, some basic terms and definitions are needed. The terms and definitions used here follow the nomenclature of [Allerton, 2009].

COORDINATE SYSTEM In modeling the attitude of an aircraft, the directions, angles, velocities and accelerations need a common reference point. For flight control a coordinate system which is firmly attached to the center of gravity of the aircraft is used. In this point, a right-handed coordinate system like shown in figure 4.2 is spanned with x being the forward direction, y being the direction along the starboard wing and z pointing downwards. Forces, accelerations and velocities along these directions are defined as positive. Moments and angular accelerations are taken as positive in the anti-clockwise sense of rotation. As this coordinate system moves along with the aircraft body, it is called the *body frame*.

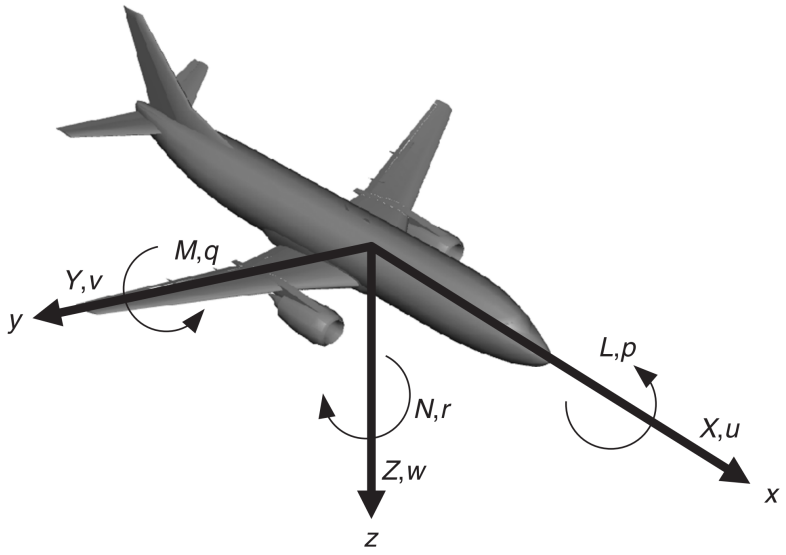

Figure 4.2: Aircraft body axes (from [Allerton, 2009, fig. 3.9])

The rotation of the body frame coordinate system wrt. an earth-centered, *geodetic* coordinate system is given by the angles ϕ (roll around the x-axis), θ (pitch around the y-axis) and ψ (azimuth around the z-axis). To convert from the geodetic system to the body frame, a sequence of Euler angles is applied in the order ψ, θ, ϕ. The correlation between those two systems is shown in figure 4.3.

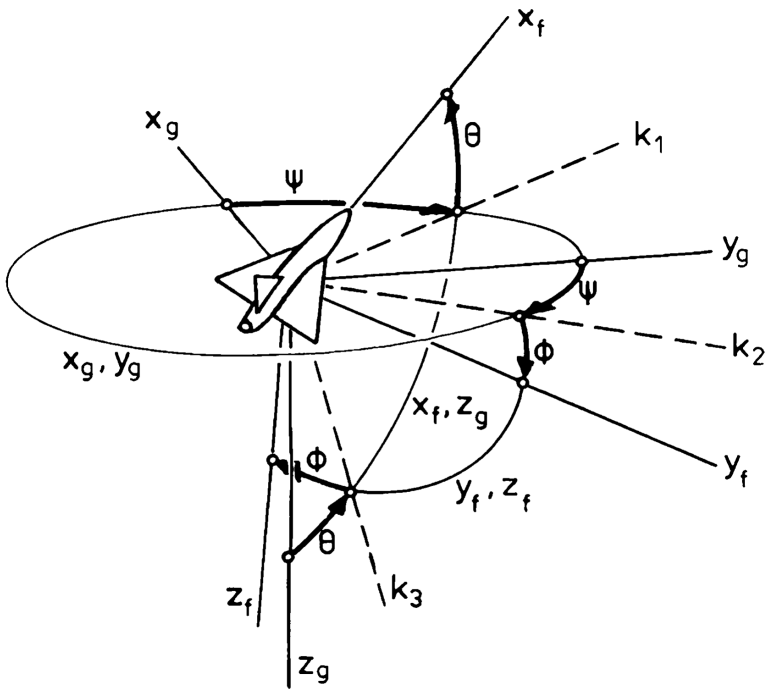

Figure 4.3: Geodetic coordinates (index g) and body frame (index f). The dashed k-lines describe intermediate steps when transforming from the geodetic to the body frame system. (from [Brockhaus et al., 2011, fig. 2.1])

6-DEGREES-OF-FREEDOM As can be seen in figure 4.2, there are three axes and at each axis a force and a moment can be applied. They are denominated with capital letters X, Y, Z for forces and L, M, and N for the moments in the figure. The motion of an aircraft is defined by these six physical values applied to the center of gravity. This coined the term of *6-Degrees-of-Freedom* (6-DoF) to describe systems like an aircraft which may translate and rotate freely in three linear and three angular dimensions.

In steady flight, all forces and all moments shall be neutralized, i. e. they shall sum up to zero.

CONTROL SURFACES To maintain this equilibrium during steady flight under external disturbances or to change the aircraft's attitude at will, forces and moments must be applied resulting in either transversal or angular accelerations till a new steady flight attitude is established. They are denoted by the lowercase letters u, v, w for transversal accelerations and with p, q and r for angular acceleration. To induce moments, so called *control surfaces* can be manipulated by means of actuation devices. A conventional fixed-wing airplane has three control surfaces to induce moments around the body axes like shown in figure 4.4: the *aileron* to induce a roll moment around the x-axis, the *elevator* to induce a pitch around the y-axis and the *rudder* to induce a yaw

around the z-axis. Application of these moments is sufficient to control the gliding descent of an aircraft. In contrast to these actively induced moments, the forces establish themselves as a consequence of the laws of motion and the air resistance. In conventional fixed-wing airplanes, the only force to be actively controlled is the engine-*thrust* in the direction of the x-axis, which is neglected in the engine-off situation chosen for this investigation.

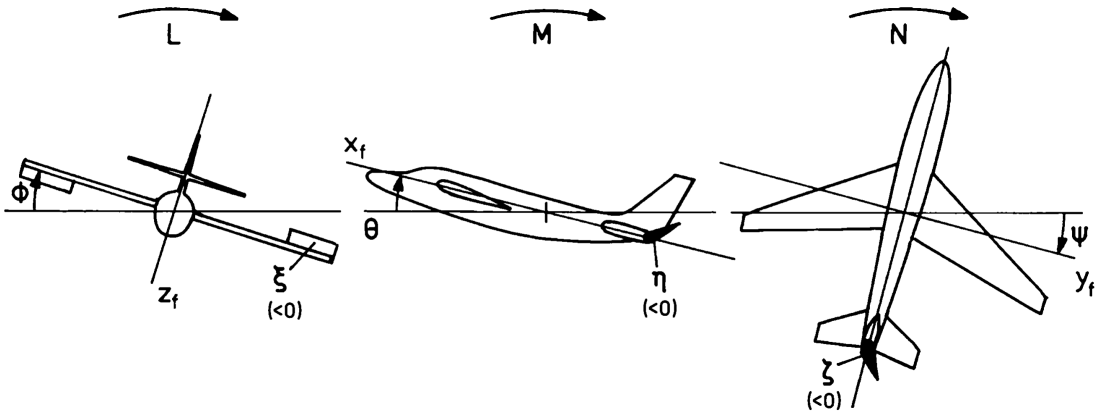

Figure 4.4: Aerodynamic control moment control (ϕ bank angle, θ pitch angle, ψ yaw angle). (from [Brockhaus et al., 2011, fig. 1.8])

The targeted excitation of the control-surface actuators is the task of the flight control system. It shall apply control commands to either maintain stable flight or to perform specific maneuvers. During gliding descent, the most important values to control are the bank angle ϕ and the glide path angle γ. To arrive at a certain point on the earth's surface with a specified heading, the gliding descent follows a trajectory in the form of a Dubins path which is defined by straight and circular segments of a defined overall length (cf. [Klein, Klos, Lenhardt, and Schiffmann, 2018]).

CONVENTIONAL FLIGHT CONTROL To follow along such a trajectory, the banking of the aircraft is controlled by means of the aileron to adjust the radius r of the circular path segments according to $r = \frac{V_{TAS}^2}{g \tan(\phi)}$ with $g \approx 9,81 \frac{m}{s^2}$ and V being the forward velocity of the aircraft ([Allerton, 2009, eq. (4.75)]). The overall range R of the gliding descent is determined by controlling the glide path angle $\gamma = \arctan(\frac{\Delta h}{R})$ by means of the elevator,[6] while respecting the constraint, that Δh must exactly correspond to the altitude to be consumed during the descent.

6 The usage of the elevator to control the glide path distinguishes the engine-off situation of a gliding descent significantly from normal flight with running engine: "The so-called 'elevator' is really the airplane's speed control, the throttle is really its up-and-down control. This is hard to believe but is one of the keys to the art of piloting." (cf. [Langewiesche and Collins, 1972, p. 153])

The rudder is not used to control the flight path, but to reduce the *sideslip* which is otherwise present in banked turning flights, as a component of the gravitational force is acting along the wing axis ([Allerton, 2009, §3.3.2]). The sideslip angle β is the deviation of the aircraft's x-axis to the direction of movement like depicted in figure 4.5. The compensation of sideslip by means of rudder activation is known as *turn coordination*.

As the rudder is not used for heading control of the aircraft's flight path, the control of the rudder can be omitted in the first instance of aircraft flight control. But as occurrent sideslip is generally uncomfortable to the passengers as they are pressed at one side of the fuselage and as it degrades flight efficiency, turn coordination should be included in any more advanced controller.

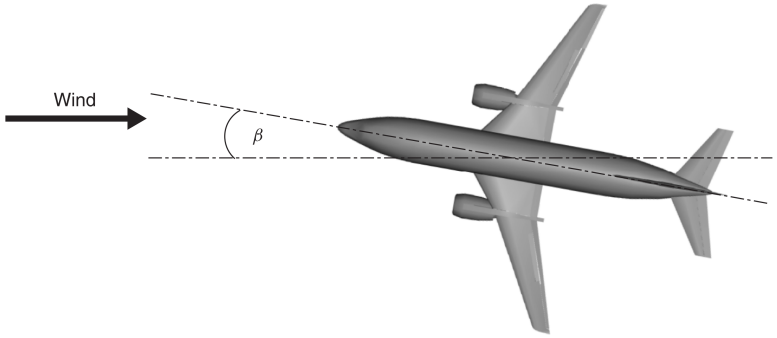

Figure 4.5: Angle of sideslip (from [Allerton, 2009, fig. 3.5])

Actual aircraft control systems are oftentimes implemented as PID controllers (cf. [Allerton, 2009, §4]), well known from systems theory. In figure 4.6 a basic PID control system is depicted with the controller $C(s)$ and the system to be controlled $G(s)$. The difference between the system output to be controlled y and the desired set-point r is the error e which is the input to the PID controller. The controller equation

$$C(s) = K_P + \frac{K_I}{s} + K_D s$$

with s being the Laplace operator is applied to the error e. Division by s translates to integration of the error and multiplication by s to derivation while K_P is a proportional factor. The controller yields a corrective action u as the system's input to the actuators.

PID controller design involves the careful tuning of the three controller parameters K_P, K_I and K_D in a well balanced manner as PID controllers are not inherently stable. The approximate influence of the individual parameters is given in table 4.1. Depending on the concrete application, the K_I or the K_D term can be omitted.

While being relatively simple, PID control suffers from the fact that it is inherently linear, while the aircraft dynamics, when examined over the full fidelity range, also

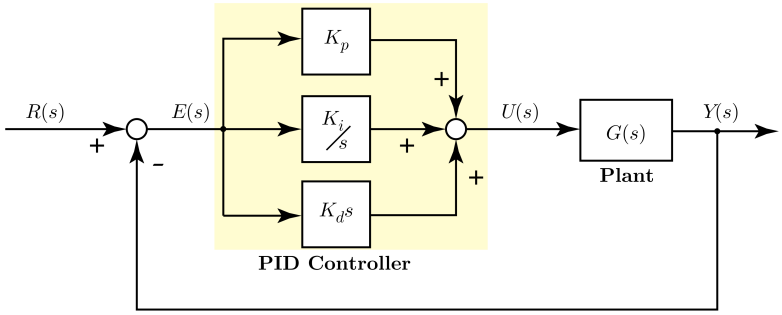

Figure 4.6: A basic PID control system (from [Paz, 2001, fig. 1])

Control term	Rise time	Overshoot	Settling time	Steady-state error
K_p	Decrease	Increase	Small change	Decrease
K_I	Decrease	Increase	Increase	Eliminate
K_d	Small change	Decrease	Decrease	Small change

Table 4.1: Effect of PID control terms (from [Allerton, 2009, tab. 4.1])

incorporate non-linear behavior. This implies, that a linear controller may only work in a limited range around an operating point where linearization is valid.

Besides getting rid of the tedious and error prone process of parameter tuning, one of the most appealing ideas of applying RL techniques to flight control is to better cope with the non-linearities of the controlled system.

4.2 Continuous Control as Markov Decision Process

Before applying RL to the challenge of flight control, the control task must be formulated as a *Markov Decision Process* (MDP), or, as will be shown later in this section, as a set of MDPs. In the upcoming section, this formulation of a continuous control problem into the MDP framework shall be elaborated, as this translation is not directly obvious.

4.2.1 *MDP Basics*

Let's revisit the first glimpse into the definition of an MDP, that was already mentioned in section 3.1 and detail this further:

An MDP is defined by the tuple

$$\text{MDP} \doteq \{\mathcal{S}, \mathcal{A}, p, r, \gamma\}$$

with the set of states $\mathcal{S}$, the set of actions $\mathcal{A}$, a stationary transition dynamics distribution with conditional density $p(s_{t+1}|s_t, a_t) : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$, the reward function

$r(s_t, a_t, s_{t+1}) : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ and γ being the discount factor defining the preference of earlier rewards over later rewards.

On first sight, one might assume, that these components are firmly determined by the underlying system to be controlled. However, on closer examination there's only one function which is naturally given, but still dependent on arbitrarily chosen values: the transition probabilities $p(s_{t+1}|s_t, a_t)$ are not changeable by the engineer but determined by the environment after the set of states $\mathcal{S}$ and the set of actions $\mathcal{A}$ was fixed.

All other components of the MDP's defining tuple can be chosen and defined arbitrarily as long as the process is still "Markov": "The qualifier 'Markov' is used because the transition probability and reward functions depend on the past only through the current state of the system and the action selected by the decision maker in that state." ([Puterman, 1994, p. 20])

In formal terms this Markov property can be stated as $p(s_{t+1}|s_1, a_1, \ldots, s_t, a_t) = p(s_{t+1}|s_t, a_t)$ for any trajectory $s_1, a_1, s_2, a_2, \ldots, s_t, a_t$ (cf. [Silver et al., 2014].

The components $\mathcal{S}$, $\mathcal{A}$, $r(s_t, a_t, a_{t+1})$ and γ can be chosen such that the system to be controlled is presented to an RL agent in a way, that the agent "does the right thing." ([Dewey, 2014]

In the upcoming paragraphs we will reason about the definition of $\mathcal{S}$, $r(s_t, a_t, a_{t+1})$ and $\mathcal{A}$ to enable high quality flight control by reward maximizing RL agents. The parameter γ is put a little bit aside, as the conducted experiments showed only negligible effects on control quality.

4.2.2 *The Notion of State*

The notion of state is a central building block of RL: State is the information the agent receives about the inner conditions of the environment. Implicitly one tends to assume, that the state information is unambiguously defined by the environment, and that it is complete in a sense, that it carries all information necessary to describe the world and its future development wrt. the control task at hand.

Unfortunately, such an assumption usually doesn't hold: Neither is it obvious which data is needed to derive all information to fulfill a certain control task, nor is every needed information generally available as sensory input from the environment.

In practice it's necessary to carefully chose the state variables to be presented to the agent among those, that are available from the environment. Neither is it beneficial to present all available state information to the agent, nor is it guaranteed, that every desirable information is directly available form the environment.

To distinguish between the *state* of the world known by some omniscient instance and the information presented to the agent we call the agent's information an *observation* in cases where this distinction is necessary. Otherwise we use the two terms interchangeably when the context is clear. The aircraft's state in its 6-degrees-of-freedom

like introduced in section 4.1 is not to be confused with the observation presented to the agent to perform some sensible flight control.

While in principle it is possible for the agent to learn which parts of the presented observation are irrelevant for its task, it is still not useful to overwhelm the agent with observations by including each and every information that can be gained form the environment. On one hand, this is due to general computational efficiency and on the other hand reinforcement learning is based on the recognition of similar situations which is less likely with an unnecessarily bloated observation. It takes the same effort for the agent to ignore a certain observation component, that it takes to include it in some RL based control.

During the conducted experiments, a successful strategy was to start with a relatively small set of observation components based on physical insights and engineering judgment, and then to extend this set of observations step by step till satisfactory control was learned. Of course this approach includes the risk of leaving out some valuable state information which could further improve the controller. This risk is however not really different from another potential problem of *hidden* or *non-observable* state. Depending on the available sensors some information might just not be available from the environment. Fortunately the generalizing behavior of the approximation functions used in the RL algorithm like described in section 3.5 can compensate missing information to a certain degree (cf. [Sutton and Barto, 2018, §17.3]). This also encourages the chosen approach of starting with a small observation and incrementally add available information.

The overall guideline is, that the observation shall include all necessary information to fulfill the control task.

In first place, this definitely means that either the set-point and the controlled value or the control deviation (the difference of set-point and controlled value) need to be included in the state. If other properties than the set-point deviation are included into the calculation of the agent's reward, it is always a good idea to include these properties in the observation as well, even though some of them may be approximated implicitly.

Of course knowledge about the underlying physics should always be leveraged: e. g. to control a circle flight it's a good idea to present the bank angle ϕ and the forward velocity V as the circle radius is defined by $r = \frac{V_{TAS}^2}{g \tan(\phi)}$.

As the agent itself has no memory, it may be necessary to explicitly include the current status of the actuators of the control surfaces in the observation to enable the evaluation of actor movement.

Another important requirement for the presented observation is that it must comply with the Markov property. The agent has neither a concept of history nor future. Every decision taken by the agent only emerges from the currently available observation and

its learned behavior. As some control strategies depend on the trajectory, we want the observation "to be a compact summary of history" ([Sutton and Barto, 2018, p. 465]). This can be achieved by different means like e. g. stacking multiple observations on-top of each other in a FIFO manner or by explicitly calculating integrals or derivatives and include them into the observation.

It's an engineering task to define the contents of the observations and "in reinforcement learning, as in other kinds of learning, such representational choices are at present more art than science." [Sutton and Barto, 2018, p. 50]

4.2.3 *Reward Properties*

The agent's ultimate goal in RL as stated in section 3.1 is to maximize the return G: $\max G = \sum_{t=0}^{T} \gamma^t r_t$. This gives rise to another engineering task in RL which is the design of a good reward function which is known as *reward engineering*: "The reinforcement learning agent's goal is not being changed, but the environment is being partially designed so that reward maximization leads to desirable behavior." ([Dewey, 2014]).

The designer's ultimate goal of high-quality flight control must be translated in a reward function so "that desirable outcomes can be recognized and rewarded consistently, and that these reward mechanisms cannot be circumvented or overcome." ([Dewey, 2014])

In this section some reasoning on the design of reward functions shall be introduced.

SPARSE VS. DENSE REWARDS One first distinction between different types of rewards is the moment it is earned by the agent. For some tasks designing a reward consistent with the designer's desire is straightforward, but this reward is earned only very rarely. In the domain of board games this could be winning the game or in flight planning arrival at the target. While it seems natural to assign some positive reward to the agent on those achievements, such a reward only occurs one to few times per episode. These rewards are called *sparse* rewards and impose some problems on the learning agent. In the beginning of learning, the agent with no prior knowledge wanders randomly and receives no guidance on how to collect the reward. It must literally stumble into a first winning situation. No intermediate progress mark is given to the agent and the positive feedback can only be evaluated after long periods with no distinguishable feedback at all.

Opposed to these sparse rewards are *dense* rewards which give feedback at every step of an episode. Dense rewards can give permanent feedback to the agent and effectively guide the learning. Such guidance can even include clues on the preference of certain trajectories over others. In continuous control tasks, rewarding the agent permanently for minimizing the deviation of the controlled value from its set-point, one can effectively nudge the agent towards the desired control behavior. Besides

using the absolute error as the only source of reward, other superordinate design objectives regarding the type of control can be posed to the controller.

POSITIVE AND NEGATIVE REWARDS So far the only requirement for the reward is to be a real number $r \in \mathbb{R}$. Nothing was said on the order of magnitude or the sign of this number. For the agent's objective to maximize the discounted sum of rewards, it is irrelevant whether for each desired outcome of an action a positive number is earned or whether for each unwanted result of an action the reward is negative in the sense of a punishment. In the case of tracking a set-point an obvious reward would be to send the absolute value of the error as a negative reward aka. punishment to the agent. A reward of zero would then be the maximum reward in each step. This however only works if the agent has no possibility to prematurely terminate episodes. Otherwise a tempting strategy for the agent would be to commit early suicide as this would immediately stop the painful sequence of receiving negative rewards. A short episode would then be preferred over achieving the actual goal of minimizing the tracking error.

As a consequence that in flight control there is usually the possibility of early episode ends, be it by crashing or by violating certain safety measures, it is preferred to work with all positive rewards to eliminate any suicidal tendencies in the agent.

REWARD NORMALIZATION Another thought shall be spared on the range of scalars a reward shall be taken from. In principle, the absolute value of a reward doesn't matter to the agent as long as the order imposed on the desirability of different states is consistent with the order of the rewards. A higher reward must only correspond to a higher desirability of the state.

As will be shown later, a reward is oftentimes an amalgamation of several components originating from different physical domains. With arbitrary orders of magnitude and possibly unbounded rewards, it becomes increasingly difficult to weight these components among each other. Therefore it is preferable to normalize all reward components to a defined interval as it makes it easier to express their relative importance.

After those thoughts the rewards in the flight control experiments were chosen as dense rewards for every time-step lying in the range of $r \in [0, \dots 1]$. This however is only the point where the task of reward engineering begins. It is still a challenge and an open question how to adequately translate the designer's preferences into a suitable reward function. "In practice, designing a reward signal is often left to an informal trial-and-error search for a signal that produces acceptable results." ([Sutton and Barto, 2018, p. 469]).[7]

7 There exist some approaches to make this informal search for the right reward function more targeted, but this is still a field of active and ongoing research far from established engineering recipes. Such

All desired behavior of the agent must be encoded in the reward function. No other inducement besides the reward can be put on the agent. A major complication arises from the fact, that humans oftentimes have big difficulties in assigning consistent values to alternatives. According to [Regan and Boutilier, 2012] this difficulty stems mostly from three reasons:

1. People find it extremely difficult to quantify their strength of preferences precisely using utility functions.
2. The necessity to assess rewards for *all* possible states imposes an additional burden.
3. There is a possible conflation of immediate reward with long-term reward as states can be viewed as good or bad depending on whether they make good states available later.

Another problem in *multi-input-multi-output systems* (MIMO) like flight control with aileron, elevator and rudder control is the question on how to relate a single scalar reward feedback to the correct action. It's not always clear how to prioritize several aspects of the agent's behavior and how to include different aspects at the same time. In the end, the reward function is not targeted anymore, but results in a mingle-mangle of excessive components.

4.2.4 *Action and Task Separation*

Reinforcement learning algorithms like e. g. DDPG can cope with MIMO systems. The input state as well as the action outputs by definition of the MDP are vectors of arbitrary dimensionality. In principle, the algorithms for maximizing the actor's performance and for minimizing the critic's loss are amenable for agents with multiple actions as output. However, as just stated in the section before, the feedback to the agent by means of the reward signal remains a single scalar value no matter what the action's dimension is. Hence the reward becomes more and more unspecific which in turn complicates the learning of good overall strategies.

This problem of assigning a fraction of the scalar reward to a fraction of the multi-dimensional action stems from the formulation of the problem as an MDP.

A brief look into the real world immediately shows, that the fiction of a single agent interacting with the environment is kind of ridiculous. In reality a lot of agents with individual interests interact with the world and with each other. This interaction can be either competitive with contradicting interests or cooperative with the individual interests (at least partly) contributing to a common superordinate goal.

In flight control for the gliding descent, the split into sub-tasks for individual agents with specialized reward functions arises quite naturally. These natural sub-tasks with an associated single action agent are

attempts are generally called *inverse reinforcement learning* (cf. [Ng and Russell, 2000]), where the reward function itself is learned by extracting it from some expert's demonstration on the task.

- Glide path angle control with an agent controlling the elevator
- Bank angle control with an agent controlling the aileron
- Turn coordination or sideslip compensation with an agent acting on the rudder.

These agents with individual reward functions shall cooperate to contribute to the common goal of high quality flight control which is quite intractable with a single common reward.

Based on the idea of multiple agents acting in the same environment, the model of so called Markov Games (cf. [Littman, 1994]) as an extension to MDPs was developed.

4.2.5 *Extension of MDPs to Markov-Games*

Markov games are an extension of MDPs to multiple agents acting in the same environment. The agents' actions can be issued to the environment subsequently one action after another in rounds or synchronously with all actions issued at once. For the flight control task synchronous actions of all agents are considered.

The formal definition of a Markov Game in this sense is straight forward and we follow the concise write-up given in [Lowe, Wu, Tamar, Harb, Abbeel, and Mordatch, 2020]:

> A Markov game for N agents is defined by a set of states $\mathcal{S}$ describing the possible configurations of all agents, a set of actions $\mathcal{A}_1, \ldots, \mathcal{A}_N$ and a set of observations $\mathcal{O}_1, \ldots, \mathcal{O}_N$ for each agent. To choose actions, each agent i uses a stochastic policy $\pi_{\theta_i} : \mathcal{O}_1 \times \mathcal{A}_1 \to [0; 1]$, which produces the next state according to the state transition function $T : \mathcal{S} \times \mathcal{A}_1 \times \mathcal{A}_N \to \mathcal{S}$. Each agent i obtains rewards as a function of the state and agent's action $r_i : \mathcal{S} \times \mathcal{A}_i \to \mathbb{R}$, and receives a private observation correlated with the state $o_i : \mathcal{S} \to \mathcal{O}_i$.

In a first guess, this is everything that is needed to solve the problem of non-targeted rewards and the agents contributing to flight control could be trained entirely independently using the DDPG algorithm from section 3.6. This naïve approach would only work without further complications if the sub-tasks were completely independent and would not influence each other at all. But in flight control such an independence assumption does not hold: Though the control surfaces are associated with a certain axis of the aircraft, there are also cross correlations and influences on the other axes.

Maybe this becomes most obvious in turn coordination where a change in the bank angle induced by the aileron evokes a sideslip to be compensated by the rudder (cf. [Langewiesche and Collins, 1972, §11]).

The actions of one agent influencing the observations of another, are perceived like a non explainable change in the environment. The environment becomes non stationary regarding the state transition probabilities $p(s_{t+1}|s_t, a_t)$ and thus one of the

preliminaries of the MDP model is violated. With a non stationary environment the assumptions regarding off-policy learning from experience replay, which is at the heart of the DDPG algorithm, do not hold anymore.

In the case of flight control with cooperative agents all residing in the same electronics system, the problem of non-explainable environment changes could be eliminated by including the other agent's actions into the observations presented to each agent. In that way, each agent would explicitly know the other agent's policies and could include them into its emerging notion of the environment. This however requires explicit communication channels between the agents to exchange their policies or action outputs.

In the very recent paper [Lowe et al., 2020] another approach to mitigate the non-stationarity issues introduced by multiple agents in the same environment is proposed: In an extension to the DDPG algorithm, it is allowed to use augmented state information in the critic's observation during training but only rely on local observations during the action determination by the actor. This yields "a framework of centralized training with decentralized execution" like depicted in figure 4.7. The proposed algorithm is an easy-to-implement extension of DDPG called *multi-agent deep deterministic policy gradient* (MADDPG).

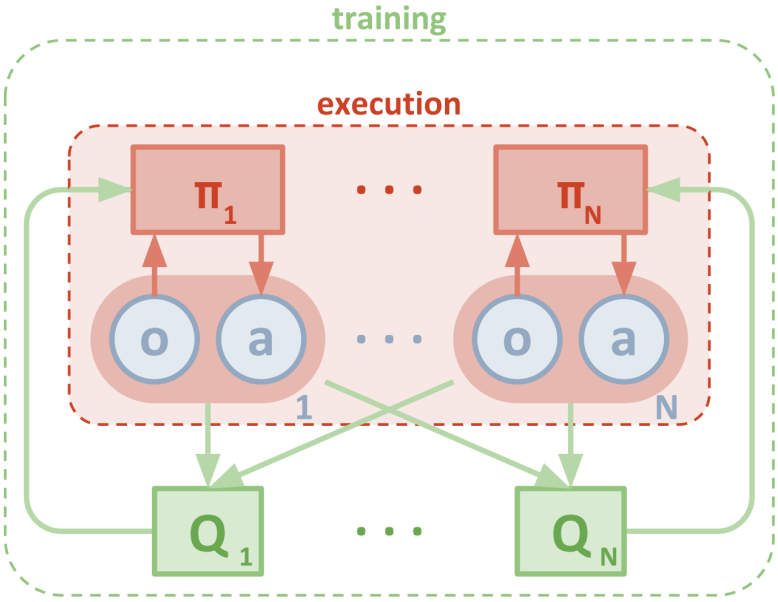

Figure 4.7: Principle of the decentralized actor, centralized critic approach. (from [Lowe et al., 2020, fig. 1])

In the MADDPG setting, the actor is left unchanged wrt. traditional DDPG, while the critic not only receives the agent's own observations, but also the observations and actions of all other agents.

The value function $Q_i^\pi(s, a_1, \ldots, a_N)$ estimated by the critic i for agent i is "a centralized action-value function that takes as input the actions of all agents, $a_1, \ldots, a_N$, in addition to some state information s, and outputs the Q-value for agent i. In the simplest case, s could consist of the observations of all agents, $s = (o_1, \ldots, o_N)$" ([Lowe et al., 2020]).

Each agent learns a different Q_i^π value which makes individual reward structures for each agent possible.

The extension to the traditional DDPG implementation is depicted in fig. 4.8. The entire algorithm from the original paper is presented in Appendix B.

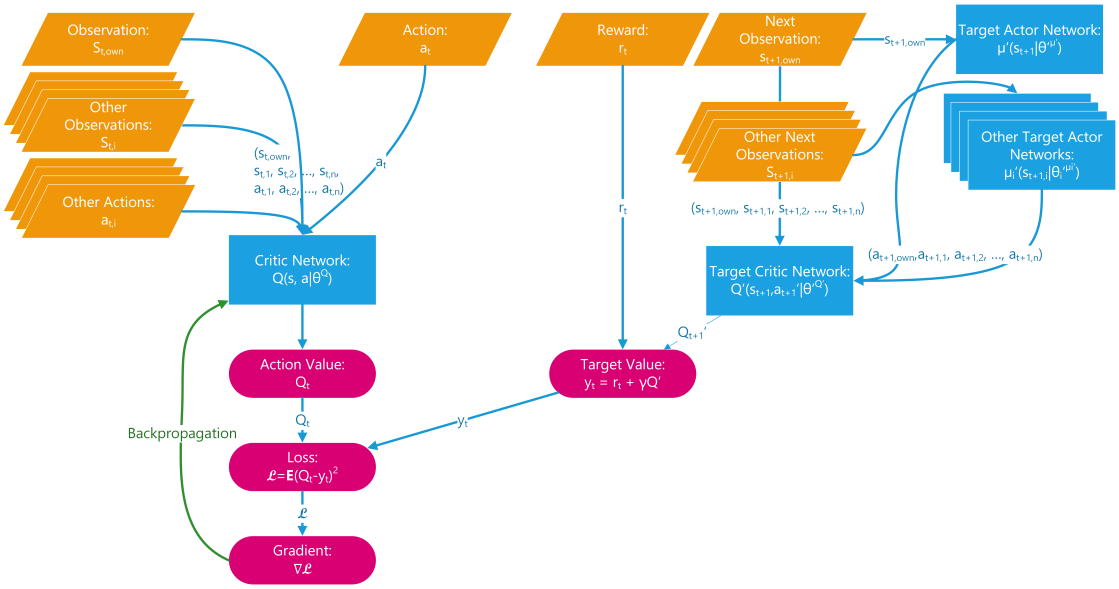

Figure 4.8: Critic's extensions to change DDPG into MADDPG, (own visualization)

By explicitly including the actions of the other agents in the observed state, the environment becomes stationary again since the transition probabilities are not further changed:
$p(s_{t+1}|s_t, a_1, \ldots, a_N, \pi_1, \ldots, \pi_N) = p(s_{t+1}|s_t, a_1, \ldots, a_N) = p(s_{t+1}|s_t, a_1, \ldots, a_N, \pi_1', \ldots, \pi_N')$
for any $\pi_i \neq \pi_i'$ which would not be the case if the other agent's actions would not be included in the observed state. (cf. [Lowe et al., 2020])

[Lowe et al., 2020] claim, that for making the environment stationary again, it does not matter whether the augmented observation is presented to the agent's actor **and** critic in the case of explicitly available communications during execution, or if this observed state augmentation is only available to the critic during training like in MADDPG. This reclaimed stationarity of the environment of course does not relieve the engineer from presenting adequate state information to the actor as discussed in section 4.2.2.

In the experiments in section 6 both variants –just adding the other actions to the agent's observation and an implementation of MADDPG– shall be investigated.

4.3 Evaluation of control Strategies

The resulting control strategies must be evaluated and compared somehow to evaluate their performance and their usefulness. Besides just making an educated guess by having an experienced engineer looking at the resulting outputs, some objective metrics are necessary to quantitatively compare different control laws.

First of all, a standardized test bed is needed to pit different controllers against each other. All controllers shall perform the same task to record their outputs and to derive some metrics for comparison.

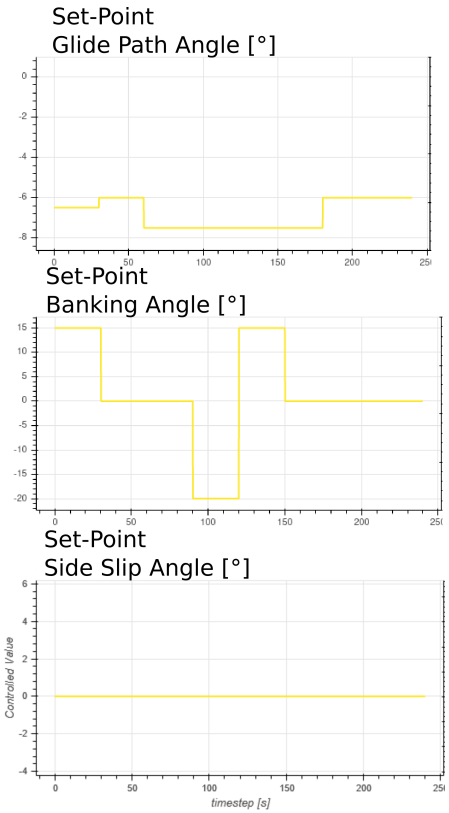

Figure 4.9: Set-point variation during standardized evaluation tests for each controlled axis

TEST SCENARIO The test scenario used in this work is an engine-off gliding descent over 4 minutes. The aircraft used as a guinea pig is the Cessna 172P included in the JSBSim simulation package. The trial starts with the aircraft flying at 90% cruise speed in level flight with 0° glide path angle, 0° bank angle and an angle of attack of 1° at

an altitude of 6000ft. Immediately at $t = 0$ the set-points for glide path γ and banking angle ϕ are set to $\gamma = -6.5°$ and $\phi = 20°$. The desired side-slip angle $\beta \equiv 0°$ over the entire test time. The set-points for γ and ϕ are varied according to table 4.2. Plots of the set-points are shown in figure 4.9.

time t [sec.]	0	30	60	90	120	150	180	240
glide path γ [°]	-6.5	-6.0	-7.5	-7.5	-7.5	-7.5	-6.0	-6.0
banking ϕ [°]	15	0	0	-20	15	0	0	0
sideslip β [°]	0	0	0	0	0	0	0	0

Table 4.2: Set-point variation during standardized evaluation tests

CHARACTERISTIC VALUES In classic linear control theory, the *step response* is one of the most important sources of information regarding the controller's performance (cf. [Tay, Mareels, and Moore, 1998, §4]). The step response is the output of the controlled system when a sudden change in the desired set-point is applied. In fully linear systems with linear controllers, a lot of internal parameters can be derived from certain metrics of the step response to a unit step. However, also in the case of a non-linear system with a non-linear controller, the step response yields a lot of valuable information to compare different control strategies.

In figure 4.10 a step response like used in our standardized test scenario is shown with the characteristic values marked that shall be used for comparison and performance evaluation.

The characteristic values recorded in the experiments are:

- step size: The difference between the old and the new set-point.
- max. overshoot: The maximum absolute value of the response in the direction of the step exceeding the new set-point. Additionally, the relative overshoot wrt. the step size is calculated. ('NaN' is given if no own set-point change caused the disturbance.)
- peak time: The time interval between applying the input step and reaching the max. overshoot.
- settling time: The time interval between the application of the input step and the output being within a maximum error band. Different error bands are used in the tests: $\epsilon_1 = \pm 0.5°$, $\epsilon_2 = \pm 0.1°$, $\epsilon_2 = \pm 0.05°$ and $\epsilon_4 = \pm 0.01°$.
- error integral: The sum of all absolute/quadratic deviations to the set-point over an interval.
- control energy: The absolute sum of all actuator movements over an interval. As a first approximation, this is proportional to the energy needed to achieve the accomplished performance.

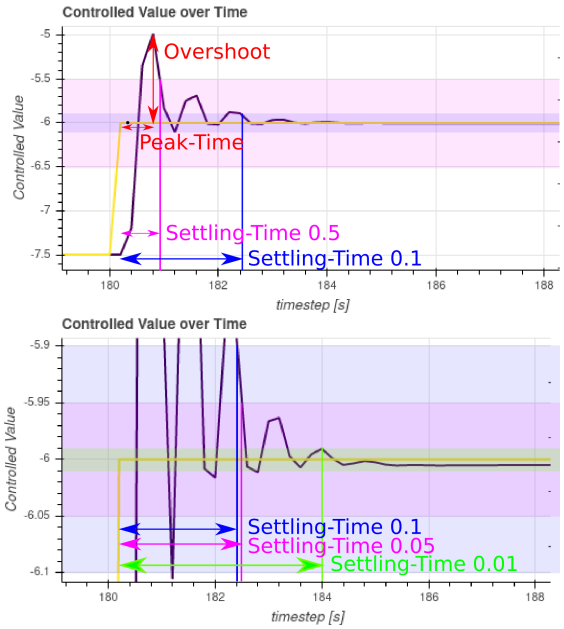

Figure 4.10: Characteristic values to be recorded for comparison of different control laws.

Besides being informally selected, these characteristic values give a good first impression on the control quality. The design goals are to achieve a control with a short settling time and minimum overshoot. The achieved error band after settling shall be minimized for high accuracy control. The overall error integral shall be minimized as well as the actuation energy needed to move the actuators.[8]

Additionally to the standard test scenario from above, a possibility is needed to perform different tests than applying step changes to the set-points. A more realistic scenario is to apply ramps. To estimate the tracking capabilities it's also a good idea to modulate the set-points using a sine function. Such set-point variations however are not covered by the characteristic values mathematically derived from the curves, but are assessed manually.

This leads to the overall most important criterion which is engineering judgment by an experienced engineer looking at the curves. No measured quantity can beat experience, especially if the trained ANN based controllers behave in unforeseen ways.

8 Of course for a final control dimensioning in a real-world application these parameters and design objectives must be explicitly elaborated in the form of unambiguous requirements. This however is out of the scope of this work.

5 DEVELOPMENT OF A SW FRAMEWORK TO APPLY RL TO AIRCRAFT CONTROL

In section 3 the basics of reinforcement learning and the DDPG algorithm in the continuous action space domain were laid out. This was followed in section 4 by considerations on the concrete application to aircraft flight control. The building blocks described in these sections form the components of a tentative design for an experimentation framework in which an actual flight control algorithm can be developed.

In this section, the actual design, architecture and implementation of this experimentation framework followed by some advice on its usage shall be given. All the software developed in the course of this research can be found under `https://github.com/opt12/Markov-Pilot`.

5.1 Overview of the Building Blocks

The generic software (SW) structure for reinforcement learning problems is quite simple and is oriented towards the training cycle shown in figure 3.1. It consists of the two main blocks *environment* and *agent*. Additionally, some overall sequencer is needed to coordinate the training, control the steps and to perform some housekeeping.

The environment must be extended to support multiple tasks with multiple rewards which finds its counterpart in an *agent container* supporting multiple control agents. A diagram of the main SW components is sketched out in figure 5.1.

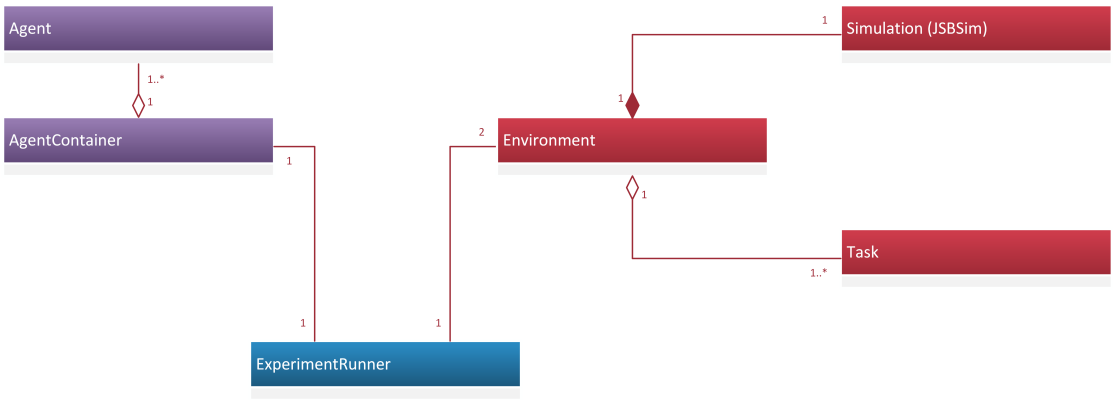

Figure 5.1: The main SW components for an RL-flight-control experimentation framework.

The overall running sequence is roughly outlined in algorithm 1.

More details on the involved classes and components are detailed below in section 5.2 on the actual implementation.

Algorithm 1 Running sequence of training session

```
Specify tasks
Instantiate an environment containing tasks and a simulator
Specify agents
Instantiate an agent container to mediate between tasks and agents
while True do
  initialize new episode, get initial tasks' observations
  present tasks' observations to associated agents
  while Episode is not over do
    get agents' next action
    issue actions to simulation
    for each task determine new observation and calculate task specific reward
    present observations and reward to agents
    for each agent, store experience (s_t, a_t, r_t, s_{t+1}) to replay buffer
    for each agent, perform a training step using minibatches from replay buffer
  end while
  plot episode's outcome
  analyze agents' performance
end while
```

Regarding the environment there exists a quasi-standard interface in the RL community: The *OpenAI gym interface* (cf. [OpenAI]). In the gym interface the interactions with an environment are standardized to enable different agents to interact with different environments. While not being the focus of the work at hand, this allows the comparison of a lot of different RL algorithms in various settings.

Unfortunately, the original gym-interface only supports a single agent performing a single task in the environment with a single reward and observation. While not being formally standardized, there exists an extension to multi-task settings which is also backed by OpenAI. This extension is used throughout the already mentioned MADDPG-paper ([Lowe et al., 2020]) and it shall also be used in this work.

At the heart of the environment implementation lies the simulation engine providing a 6-DoF flight model. For RL experiments it is of utter importance to use a fast model that can speed up beyond real-time providing a convenient interface to issue control commands and to yield internal state information. Of course, the flight dynamics model (FDM) must be sufficiently accurate to generate reliable results.

Eventually, *JSBSim* was chosen as flight dynamics model.[9] This simulation engine is freely available and is used in multiple projects: "The most notable examples of the use of JSBSim are currently seen in the FlightGear (open source), Outerra, BoozSimulator (open source), and OpenEaagles (open source) simulators. JSBSim is also used to drive the motion-base research simulators at the University of Naples, Italy, and in the Institute of Flight System Dynamics and Institute of Aeronautics and Astronautics at RWTH Aachen University in Germany." ([JSBSim]) An overview article featuring some internals and other real-world use-cases of the JSBSim engine can be found in [Berndt and De Marco, 2009].

JSBSim can be used in a headless mode for blazingly fast simulations during training or it can utilize the open-source flight-simulator *FlightGear* as a rendering engine to visualize trained controllers.

Last but not least, the integration into an OpenAI gym environment was already done in a master's thesis at the University of Bath [Rennie, 2018] which served as a jump-start for the research at hand.[10]

Despite targeting in a different direction, the software developed by Gordon Rennie is highly valuable as a basis for a more comprehensive multi-agent experimentation framework. Quite some components could be reused. These components are clearly marked in the implementation description later on in this section.

5.2 Implementation

The experimentation framework developed during this research is called `Markov-Pilot` to stress the interpretation of flight control as a Markov decision process resp. a Markov game to make it amenable to RL methods. While currently used for applying the (MA)DDPG-algorithm to the gliding descent scenario, the framework is designed to be interfaced with other flavors of RL algorithms and to include more tasks including flight planning and e. g. thrust and flap control.

The different software modules shall be described briefly to convey an impression of the responsibilities and the most important implementation details. The description

9 The first experiments in the course of this work were conducted with XPlane, but it became obvious that this simulator is extremely difficult to integrate in the RL-interaction cycle: XPlane runs in real-time with no easy to access possibility to perform discrete steps. It runs in real-time and even slows down the internal simulation time without further notice if the rendering is slow. It is not possible to run XPlane in a headless mode without rendering the scenery, which consumes a lot of GPU power that cannot be used for ANN training anymore.
All in all, the experiments with XPlane were quite disappointing as the mentioned constraints made learning slow and debugging almost infeasible.

10 The goal of that thesis was also to apply RL techniques to flight control, but the approach was different: In that thesis, the focus was merely on low-level flight control, but more on higher level flight planning. Heading hold and turning tasks were attempted to train, but the results regarding a smooth flight stayed relatively poor. In the conclusion the suspicion is formulated, that this struggle results from the combination of learning different goals at once in a single agent setting.

given here is explicitly not an entire SW implementation specification, as all details can be found directly in the sources. The sources are extensively commented and equipped with `pydoc` documentation. Hence the given description is meant to serve as a sound basis to understand the experiments and results presented in section 6 and to know where to hook in extensions for further research.

The SW consists of these six modules:

- `tasks`: Contains the task implementation to be embedded into an environment. Includes functions to define task specific observations and rewards.
- `environment` Implements the multi-agent OpenAI gym interface and controls the JSBSim simulation engine. Serves as a container for multiple tasks.
- `wrappers`: Provides wrapper classes to extend the environment's functionality. In the current framework wrappers for set-point variation and for visualization and controller evaluation are provided.
- `agents`: Implements the agents for different types of flight control. Provides an agent container to mediate between the individual agents and the environment interface. A compatible training loop is also provided within this module.
- `testbed`: Implements the standardized testing sequence to compare different agents.
- `helper`: This is kind of a kitchen-sink module to collect various utilities and helpers providing useful functionality of general interest.

5.2.1 *Tasks Module*

In a *task* a certain control objective is defined. The task is part of the environment and targets one individual aspect of control. The task is responsible to query the relevant aircraft's state and to calculate the corresponding reward as a feedback to the controller agent. To enable control by the agent, the task maintains a list of observations containing suitable state information to be provided. The components of this observation are either directly pulled from the simulation engine or they are maintained under the task's responsibility. The task object also checks if any parameter is out of bounds so that an episode must end.

In the simplified class diagram in figure 5.2, the most important components of the `task`-module are shown.

SINGLECHANNEL_FLIGHTTASK In the `Markov-Pilot`-SW, the base class `FlightTask` is not directly used, but only the derived `SingleChannel_FlightTask` specialization.

At initialization, each `SingleChannel_FlightTask` object is provided with its task specific information. The initialization parameters together with their default values are:

- `name`: Specifies the name of the task. Is used to associate agents and tasks.

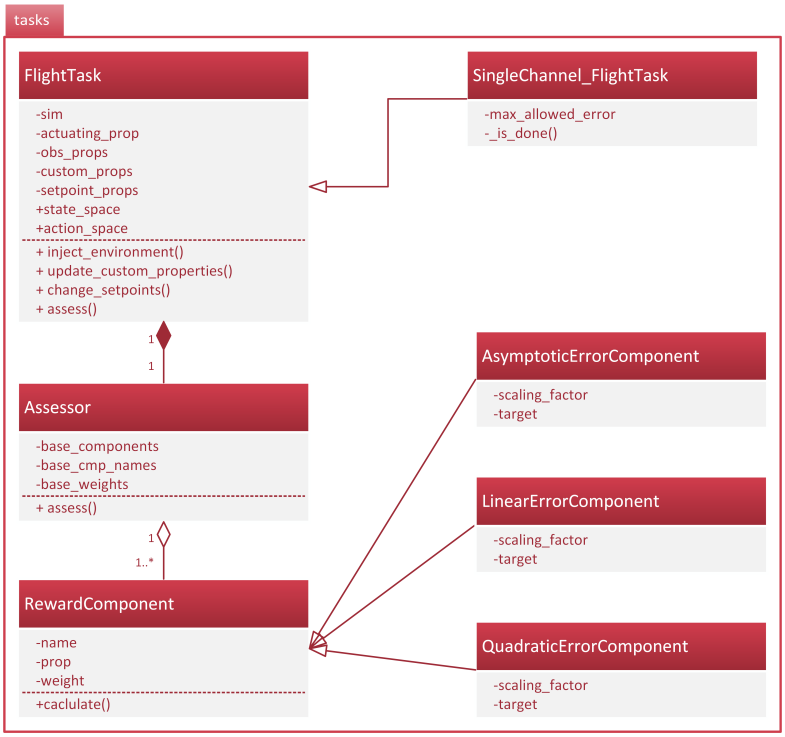

Figure 5.2: Simplified class diagram of `tasks`- module.

- `actuating_prop=None`: (Optional parameter) The property specifying the actuator associated with this task.
- `setpoints=dict()`: (Optional parameter) A dictionary of the set-points to be maintained by the task. The dictionary keys indicate the properties (e. g. banking angle, glide path angle, ...) and the value specifies the desired value. Currently only one set-point per task is supported for the `SingleChannel_FlightTask`. If more set-points shall be tracked at once, multiple tasks may be combined when assigning them to agents.
- `presented_state=[]`: A list of additional properties to be presented in the task's observation.
- `make_base_reward_components()`: Function to prepare a tuple of reward components. This function is injected and bound to the `SingleChannel_FlightTask` object and can therefore access all internal variables and properties. If not specified, a default function is used yielding a constant-Zero-`RewardComponent`.
- `is_done()=None`: Function to determine the end of an episode. If specified, this function is injected and bound to the `SingleChannel_FlightTask` object and can therefore access all internal variables and properties. If not specified, a default function is used checking the maximum allowed deviation from the set-point.

- `measurement_in_degrees=True`: Specifies if the quantities underlying the set-points are given in degrees to correctly handle turn-over.
- `max_allowed_error=`∞: Specifies the maximum allowed set point deviation used by the default `_is_done` function.
- `integral_limit=`∞: Specifies the limit for the calculation of the error integral.
- `integral_decay=1`: Specifies a decay factor which can be used in the calculation of the error integral.

If a set-point is specified for the `SingleChannel_FlightTask`, in each step the deviation of the actual value in the simulation to the desired set-point is calculated. Additionally the integral and the derivative of this error is determined. These three calculated custom properties are added as the first elements to the observation.
To avoid integrator wind-up, the integral can be limited. Optionally, the integral can be calculated using a decay factor.

Each `SingleChannel_FlightTask` can be associated with an actuator. If this assignment is specified, the `FlightTask` updates the actuator movement in every step and adds it to the observation.

A function `make_base_reward_components()` is injected during initialization to specify the reward to be calculated by the task. This function is then bound to the task object. Injecting and binding a function at initialization time has the advantage, that this function has access to all member variables, but can be specified and modified in a different source file. When persisting a task definition, this function is stored as Python-source code which makes it easy to experiment with different reward functions. Each simulation property used in a `RewardComponent` for calculation, is automatically added to the observation of the task.

The `presented_state` list can be passed at initialization time to present even more state information in the task's observation. The list entries specify arbitrary simulation properties which shall be included in the task's observation.

The task's entire observation is maintained in the task object's `obs_props` list which eventually consists of the error wrt. the set-point including integral and derivative, the actuator movement, the properties specified in `presented_state` and the properties referenced in the `RewardComponents`. The `obs_props` list is used to map the observations to the associated agents.

The value ranges for the task's observations and the task's action can be queried via the `state_space` and the `action_space` variables to inform the associated agents about the admissible domains.

Before using a `SingleChannel_FlightTask` object, it must be provided with a handle to the simulation engine to query simulation properties. This handle is injected by calling `inject_environment()` before the simulation is started.

When the task is used within a running environment, in each step the functions `update_custom_properties()` and `assess()` are called to update all calculated components of the observation and to calculate the reward and its contributing components for each new state.

The desired set-point value can be changed anytime by calling `change_setpoints()`.

ASSESSOR To calculate the reward from the current aircraft state, an `Assessor`-object is used. It iterates over its `RewardComponents`, calculates their individual reward contributions and finally aggregates them as a weighted sum into the task specific reward value. As an additional information, the individual contributions from each `RewardComponent` are returned alongside the aggregated reward value.

REWARDCOMPONENTS In the `Markov-Pilot`-SW, three different types of reward components are available. Each of them shall be initialized with the same parameter set, but they differ in the method of reward calculation. The initialization parameters together with their default values are:

- `name`: Specifies the name of the reward. Is used in the epsiode visualization to identify individual reward components.
- `prop`: Specifies the property, the `RewardComponent` is based on. A property mentioned here is added to the task's observation automatically.
- `weight`: When aggregating the `RewardComponents`, this `weight` is used in the weighted summation.
- `scaling_factor`: Specifies the scaling factor k used to normalize the reward.
- `target`: For error based reward components, this is the target value to be achieved by `prop` to gain the full reward.

The reward components used in `Markov-Pilot` provide positive normalized rewards in the range $r \in [0, 1]$. They are all error-based rewards, which means they use the absolute deviation from a property's actual value to a desired target as their calculation input. These base-properties can be either simulation properties contained in JSBSim or custom properties calculated in each step by the task object. The three different available reward components are shown in figure 5.3.

The three different `RewardComponents` use different formulæ to calculate the reward with the error to the target being e and the scaling factor being k:

- `AsymptoticErrorComponent`: $r = 1 - \frac{\frac{|e|}{k}}{1+\frac{|e|}{k}}$. At the point $|e| = k$, the resulting reward is $r = 0.5$.
- `LinearErrorComponent`: $r = 1 - \frac{1}{k}|e|$ clipped to the interval $r \in [0, 1]$ with $r \equiv 0$ if $|e| \geqslant k$.
- `QuadraticErrorComponent`: $r = 1 - \frac{1}{k}e^2$ clipped to the interval $r \in [0, 1]$ with $r \equiv 0$ if $|e| \geqslant \sqrt{k}$.

While the linear and quadratic error components achieve the reward normalization by clipping the reward, the asymptotic component performs a real mapping of the

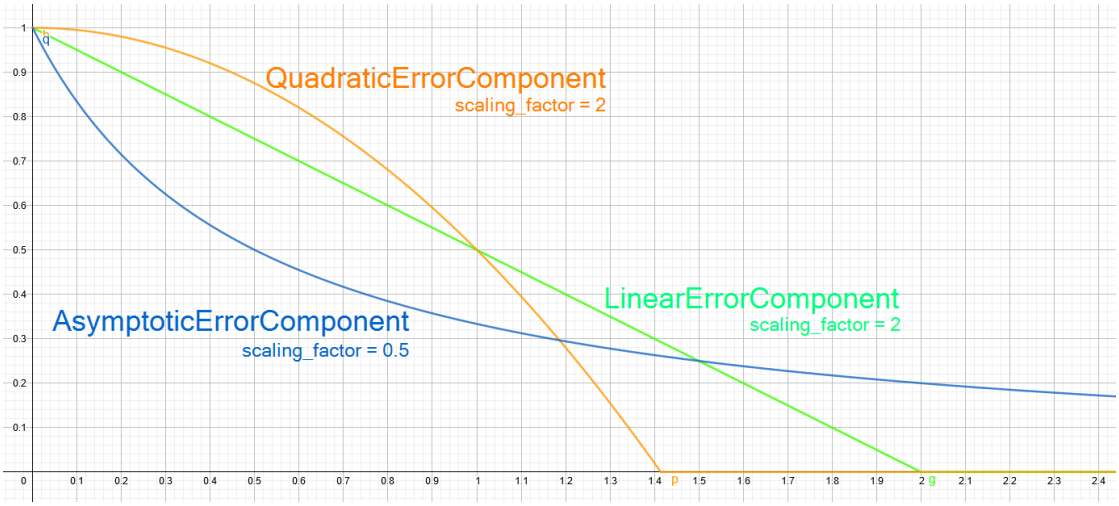

Figure 5.3: Comparison of the different `RewardComponents`.

possible error interval $|e| \in [0, \infty]$ to the reward interval $r \in [0, 1]$. This is especially useful to guide the agent towards small errors, before the linear or quadratic reward components get effective.

The factor k in the asymptotic reward component can "intuitively be interpreted by the designer as 'at what absolute error value do I consider the agent's work to be half complete?'." [Rennie, 2018, p. 38]

An arbitrary combination of `RewardComponents` can be used for the reward calculation by injecting the function `make_base_reward_components()` into the constructor of `SingleChannel_FlightTask`. The `RewardComponents` returned by this function are used to determine the current reward by the `Assessor` on a call to the task's `assess()`-function.

Takeovers from [Rennie, 2019]:

The overall concept of a dedicated `Assessor` object to calculate the rewards is a takeover. The reward calculation algorithm of `AsymptoticErrorComponent` and the `LinearErrorComponent` could be used unchanged, while the `QuadraticErrorComponent` was added. The possibility to use different weights to aggregate the `RewardComponents` was missing in the original. The original `Assessor`'s abilities for reward shaping and defining dependent rewards are not currently used in the `Markov-Pilot-SW`, but are still contained in the sources.

While being based on the original idea of the definition of a single task within the environment, the vast majority of the task implementation was developed from scratch as the original did not support multiple tasks.

5.2.2 *Environment Module*

The *environment* module is responsible for the simulation of the world the agents interact with. It accepts actions, performs a simulation step and, for each embedded task, returns a new observation in conjunction with the reward received.

In the simplified class diagram in figure 5.4, the most important components of the `environment`-module are shown.

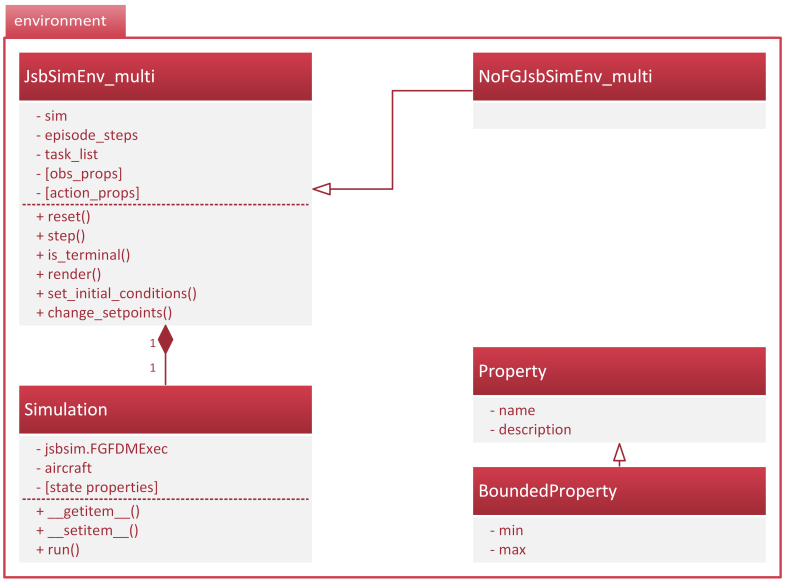

Figure 5.4: Simplified class diagram of `environment`- module.

JSBSIMENV_MULTI The `JSBSimEnv_multi` class provides the aircraft simulation to `Markov-Pilot`. It accepts actions, performs a simulation step and comes back with the new observations collected from the `FlightTask`-objects. A `JSBSimEnv_multi`-object[11] is initialized with these parameters:

- `task_list`: The list of tasks to be embedded in the environment. This list defines the observations and actions maintained by the environment.
- `aircraft=cessna172P`: Specifies the aircraft type to be used by the simulation engine. As a default the Cessna 172P model is used.
- `agent_interaction_freq=5`: Specifies the interaction frequency of the agents with the environment. As a default value 5 steps per second are assumed.
- `episode_time_s`: Defines the maximum length of an episode in seconds.

On instantiating a new environment, a new instance of JSBSim is prepared. A reference to this simulation object is then injected into each task of the provided task list to allow direct access to the simulation properties for the reward calculation and observation preparation. Internally, the environment object's constructor performs

11 The environment is available in two flavors: `JSBSimEnv_multi` and `NoFGJSBSimEnv_multi`. The basic functionality of both classes is exactly the same with the only exception, that the `NoFGJ...` environment is not enabled to render an aircraft in the FlightGear software. This environment variant must be used for training purposes, as due to an underlying error, the FlightGear-enabled variant leaks a file handler in every episode and therefore crashes in a training session with thousands of episodes even if no actual rendering is requested.

some sanity checks (e. g. no double actions nor set-points) and maintains lists of the observation and action contents of the embedded tasks.

The main purpose of the `JSBSimEnv_multi`-object however is to offer the OpenAI gym-like interface for multi-agents. The main functions of the said interface shall be briefly explained:

THE OPENAI GYM-INTERFACE "Gym is a toolkit for developing and comparing reinforcement learning algorithms. It supports teaching agents everything from walking to playing games like Pong or Pinball." [OpenAI]

To facilitate the attachment of agents implementing various RL algorithms to a broad spectrum of environments, a very simple to use interface is provided by gym-compatible environments. The three most important operations to be supported by such environments are:

- `reset() -> observation`: Resets the internal simulation engine to initial conditions and yields a first observation.
- `step([actions]) -> (observation, reward, done, info)`: Issues the specified actions to the simulation engine and advances it by one step in time. Returns a tuple of `(observation, reward, done, info)`.
- `render(mode)`: Depending on the mode, some presentation of the environment shall be rendered.

These simple commands are sufficient to perform the agent-environment interaction cycle shown in figure 3.1. At the start of each episode, the environment is reset and afterwards the agent and the environment are called alternately till the `done` flag indicates the end of an episode. The agent provides a new `action` based on the last `observation`, this `action` is performed in the simulation and feedback information based on the internal simulation state is returned.

Besides the next `observation`, a `reward` value is returned to rate the last action. The contents of the `info`-dictionary is not specified by the OpenAI gym interface, but may be filled with any useful information at the developer's convenience. In `Markov-Pilot`, the `info` field is used to provide information on the individual `RewardComponents` aggregated in the `reward` value.

EXTENSION TO MULTI-AGENTS Following the example given in [Lowe et al., 2020], the interface extension to a multi-task-multi-agent environment is as simple as using lists instead of single values as input parameters and return values. The signatures of the interface functions change to

- `reset() -> [observation_n]`
- `step([actions_n]) -> ([observation_n], [reward_n], [done_n], [info_n])`

The `_n` suffix indicates "per task" values which are internally distributed to and collected from the embedded tasks. An episode ends if any of the `[done_n]` flags is set to true. As there may be terminating conditions for an episode not supervised

by the embedded tasks (e. g. the maximum episode length), an additional function `is_terminal()` is provided by `JSBSimEnv_multi` to explicitly query for the end of an episode as the `[done_n]` flags only cover task specific conditions.

Besides the standardized gym interface, the `JSBSimEnv_multi` supports some more useful functions. The most important ones are `set_initial_conditions()` to change the aircraft's initial flight attitude and the `change_setpoints()` function which is forwarded to the individual tasks to change the set-points within a running episode.

Additionally, there exist functions to persist and restore an existing environment with all its internal task settings.

SIMULATION The other major class in the environment-module is the `Simulation`-class. This class is used to spawn a JSBSim instance performing the 6-DoF simulation of the aircraft.

Internally, the JSBSim engine holds a property tree containing all necessary state information for the simulation of the aircraft. The list of available properties depends on the used aircraft model and can be obtained from JSBSim directly using a shell command.[12]

Additional properties can be created at run time which makes the `Simulator` class also a suitable data storage for custom task properties which may be read and written by using the provided `__getitem__()` and `__setitem__()` methods (cf. [JSBSim Reference]). The values in the property tree can then be accessed like a Python dictionary.

The advancement of the simulation can be entirely controlled by calling the `run()` method. Before `run()` is called, the new command properties shall be set in JSBSim's internal property tree. This is all handled within the `JSBSimEnv_multi.step()` function.

(BOUNDED) PROPERTY The `Property` and `BoundedProperty` classes are used to access the property tree of the simulation object. `Property` boils down to a named tuple with a `'name'` entry specifying the path to a certain property in JSBSim's property tree. In the source file `properties.py` the globally relevant properties are collected and may be accessed by other parts of the SW after importing this file and obtaining a handle to the `Simulation`-object.

With the first write access to a formerly unknown property a corresponding entry is created in the JSBSim property tree. This facilitates the storage of locally defined values in the simulation object as a central data storage. This principle is used e. g. for the custom properties of the `FlightTask`-objects.

In the `BoundedProperty`-class boundaries for the allowed values are given to derive the limits for observation and action spaces needed by the agents.

12 The shell command to be used is like `JSBSim -aircraft=c172p -catalog`

Takeovers from [Rennie, 2019]:

The interface to the JSBSim engine and the concept of the `Property`-dictionary is a takeover. The extension to a multi-task-multi-agent interface and functionality like data persistence, initial conditions and set point changes are a new development for `Markov-Pilot`.

5.2.3 *Wrappers*

In OpenAI gym a special *wrapper* class is defined which is used to extend the functionality of an environment. The `Wrapper`-class is derived from the environment class and takes the environment to be wrapped as its first initialization parameter. The wrapper class can intercept the method calls to the environment to perform additional operations. To add functionality to an existing environment function, intercepted functions shall be overwritten in the wrapper class. In the very most cases, the original method of the wrapped environment is called inside the corresponding wrapper's method.

In `Markov-Pilot` two additional wrappers are provided to extend the functionality of `JSBSimEnv_multi`. These wrappers are briefly shown in figure 5.5.

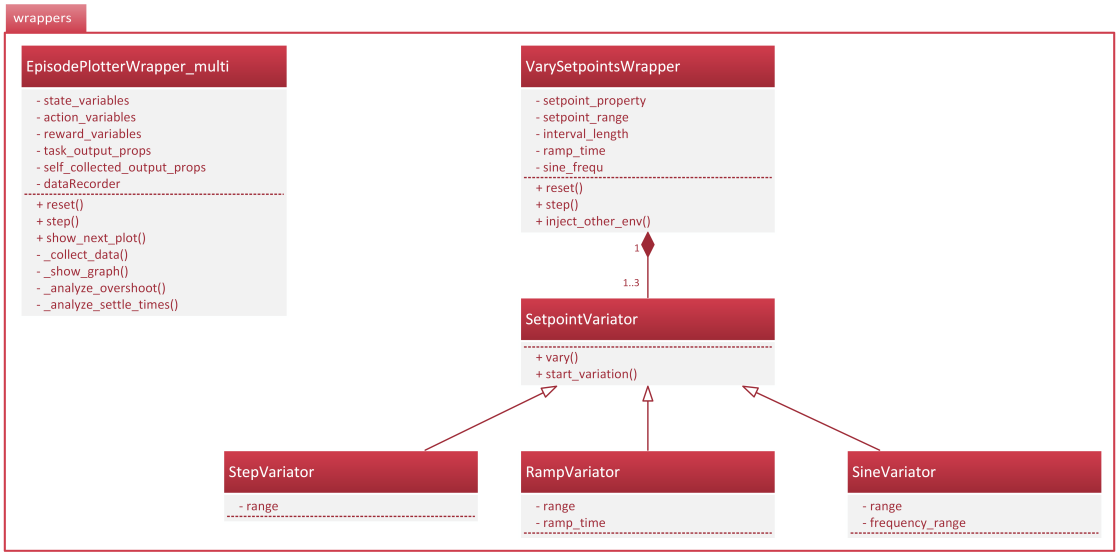

Figure 5.5: Simplified class diagram of `wrappers`- module.

EPISODEPLOTTERWRAPPER_MULTI While the `render()`-method specified in gym's environment is meant to be called after each simulation step, some means for visualizing an entire episode is needed. The `JSBSimEnv_multi` environment itself has no memory and cannot analyze data sets of full episodes. The `EpisodePlotterWrapper_multi` wrapper class provides internal memory to collect episode data and is therefore used for all necessary analysis with an episode scope.

To initialize an `EpisodePlotterWrapper_multi`-object these parameters are given:

- `env`: The environment to be wrapped
- `output_props=[]`: An optional list with properties that shall be plotted, but are not contained in any of the task's observations.

On initialization, the wrapper builds a list of all simulation properties contained within all tasks, the properties given in the `output_props` parameter, all rewards and reward components of all tasks and all available actions.

The `EpisodePlotterWrapper_multi` intercepts the calls to `step()` and `reset()`. On `reset()`, the internal memory is cleared. On each call to `step()`, a step on the wrapped environment is performed and then the properties in the wrapper's list are collected and stored to the memory. When the episode is over, the internal memory contains a full data set of values to be plotted or analyzed.

To control the outputs of `EpisodePlotterWrapper_multi`, there are three flags that can be set by calling `showNextPlot()`:

- `show`: If set, the last episode's data is plotted to an interactive website.
- `export`: If set, the last episode's data is exported to a `PNG` graphics file.
- `save_to_csv`: If set, the last episode's data is exported to a `CSV` data file for further analysis with external tools.

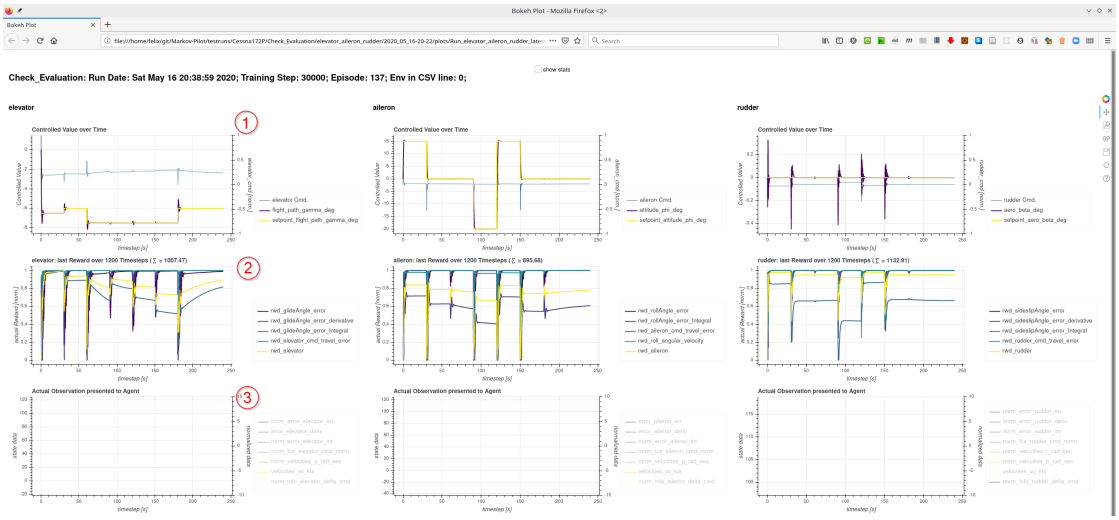

Figure 5.6: Website output produced by the `EpisodePlotterWrapper_multi` at the end of an episode.

If the data is visualized, one column per task is created containing three plot panels each. An example is shown in figure 5.6:

1. The panel contains the desired set-point and the current value. Referenced to the right axis, the applied actuation is shown.
2. The panel contains the reward assessed by the task in each step and all individual reward components contributing to it. All rewards are in the range $r_i \in [0, 1]$. The weights of the component's contribution is not shown in the plot.

3. The panel contains all properties making up the observation of the task. To avoid excessive distraction, all curves are dimmed by default, but can be enabled by a click on the corresponding name in the legend. As properties may span entirely different orders of magnitude, two axes are used for the plot. Smaller values go to the right axis and are marked with the prefix `norm_` in the legend.

All plots are interactive and can be panned and scaled at the user's convenience. The time-scales of all plots are coupled to avoid confusion about the shown cut-out. Statistics tables on the controller's performance can be activated by selecting the `show stats` checkbox.

As the episode data is available in the `EpisodePlotterWrapper_multi` anyhow, the analysis of the controller's performance like described in section 4.3 is also included. At the end of an episode, the characteristic values regarding the control quality and the settling times are determined from the recorded data for every task. Result tables of this analysis can be added to the plotting website on the user's request. Examples for the characteristic values analysis and the settling time analysis are shown in tables 5.1 resp. 5.2.

For the analysis, an episode is split into several segments which mark the intervals in between two events. The characteristic values and the settling times are analyzed for each of these segments separately. As a set-point change in one task usually induces a noticeable disturbance in the other tasks, an event is defined as a set-point change in any of the tasks embedded into the environment. The standardized test-run proposed in table 4.2 contains seven individual events and segments.

Characteristic values of control between events:

event #	0	1	2	3	4	5	6
	RL control	RL	RL	RL	RL	RL	RL
event_time	0	30.2	60.2	90.2	120.2	150.2	180.2
peak_time	1.6	32.2	60.8	93	122.8	152	180.8
delay_secs	1.6	2	0.6	2.8	2.6	1.8	0.6
setpoint	15	0	0	-20	15	0	0
actual_value	15.575	-0.263	0.108	-20.15	15.428	-0.231	-0.117
setpoint_change	15	-15	0	-20	35	-15	0
abs_overshoot	0.575	-0.263	0.108	-0.15	0.428	-0.231	-0.117
rel_overshoot	0.038	0.018	NaN	0.008	0.012	0.015	NaN
abs_mean	0.292	0.294	0.005	0.441	0.997	0.274	0.003
MSE	3.098	2.824	0	5.591	23.566	2.603	0
actuation_energy	0.363	0.57	0	0.56	0.48	0.576	0

Table 5.1: Table containing characteristic values after analyzing the aileron-task in a standard test run (cf. section 4.3). For each section in between events, the characteristic values are given. "NaN" is given for relative overshoot if the event starting the segment was no own set-point change, but a disturbance from another task.

Settling times after events given in seconds:

event #	0	1	2	3	4	5	6
	RL control	RL	RL	RL	RL	RL	RL
setpoints	15.0	0.0	0.0	-20.0	15.0	0.0	0.0
setpoint_changes	15.0	-15.0	0.0	-20.0	35.0	-15.0	0.0
error band: 0.5	1.8	1.2	0.0	1.6	1.8	1.2	0.0
error band: 0.1	2.6	3.0	0.8	3.2	3.6	3.0	0.8
error band: 0.05	3.4	3.0	2.0	3.2	4.2	4.2	2.0
error band: 0.01	5.4	4.6	4.2	4.6	6.2	5.4	4.4

Table 5.2: Table with the settling times analysis after analyzing the aileron-task in a standard test run (cf. section 4.3). For each section in between events, the settling times for different allowed bandwidths are determined. If settling to a certain bandwidth is not achieved within a segment, "NaN" would be stated instead of a settling time.

For more thorough analysis of the episode, the entire data including the additional task-independent properties given in the `output_props`-parameter can be saved to a `CSV`-file.

VARYSETPOINTSWRAPPER An RL agent learns from its experience during training. To train the agent for the broadest range of contingencies, it is essential to provoke a broad range of experiences during training. While it is desirable for testing and comparison to have a standardized setting with the same objective repeated in every test run, this is the opposite of what is needed for an effective training.

The `VarysetpointsWrapper` that randomly varies the set-point for a certain task during training was developed to present as many different situations to the training agent as possible.

While step functions are very well suited to compare different controllers, a step function is very unlikely to occur in real-world flights. To also present more realistic scenarios to the training agent, the `VarysetpointsWrapper` can not only apply step functions to a certain set-point, but can also apply ramp- and sine-shaped modulations.

To apply a `VarysetpointsWrapper` to an existing `JSBSimEnv_multi` environment, these parameters are needed:

- `env`: The environment to be wrapped
- `setpoint_property`: The property specifying the set-point to be varied.
- `setpoint_range`: The maximum allowed range for the set-point. All variations are kept within these bounds.
- `interval_length=(5, 120)`: Specifies the time range in seconds till a new set-point variation gets effective. The interval time is randomly selected within the given limits.
- `ramp_time=(0, 0)`: The range from which the ramp times are selected. If the parameter is left out, no ramps will be applied to the set-point.

- `sine_frequ=(0, 0)`: The range from which sine frequencies are randomly selected to modulate the set-point. If the parameter is left out, no sine modulation will be applied to the set-point.

To apply set-point variations during training, one instance of `VarysetpointsWrapper` must be applied per set-point to be varied. During a running episode, the wrapper chooses a time from the given interval range and applies one of the activated variations after the chosen interval expired.

Slightly abusing the idea of gym environment wrappers, `inject_other_env()` is provided to inject multiple other environments into the wrapper. All additionally injected environments experience the same variations when a step in the master-environment is taken. This feature comes in handy if several agents shall not only be compared in the standardized test bed, but also with randomly varied set-points.

The wrapper can be applied multiple times per environment with different set-point properties. An example for the set-point variation during training is shown in figure 5.7. The set-point variation was found to be very effective during agent training.

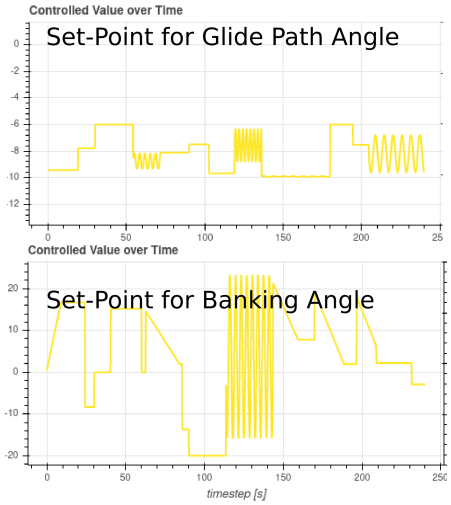

Figure 5.7: Resulting set-point variations when an instance of `VarysetpointsWrapper` is applied to the glide path angle (top) and the banking angle (bottom) in the gliding descent scenario.

5.2.4 *Agents Module*

In the RL interaction cycle from figure 3.1, the agent is the counterpart of the environment. The agent is the controller which shall be trained to perform control tasks in the environment. Following the idea of Markov games like explained in section 4.2.5 the `Markov-Pilot`-SW supports multiple agents acting on the same environment.

The agents themselves called `AgentTrainer` and an `AgentContainer` are the classes implemented in the agents module. Additionally, the module contains a source file `train.py` providing the function `perform_training()` that orchestrates the entire RL interaction cycle together with the necessary training steps.

In the simplified class diagram in figure 5.8, the most important components of the `agents`-module are shown.

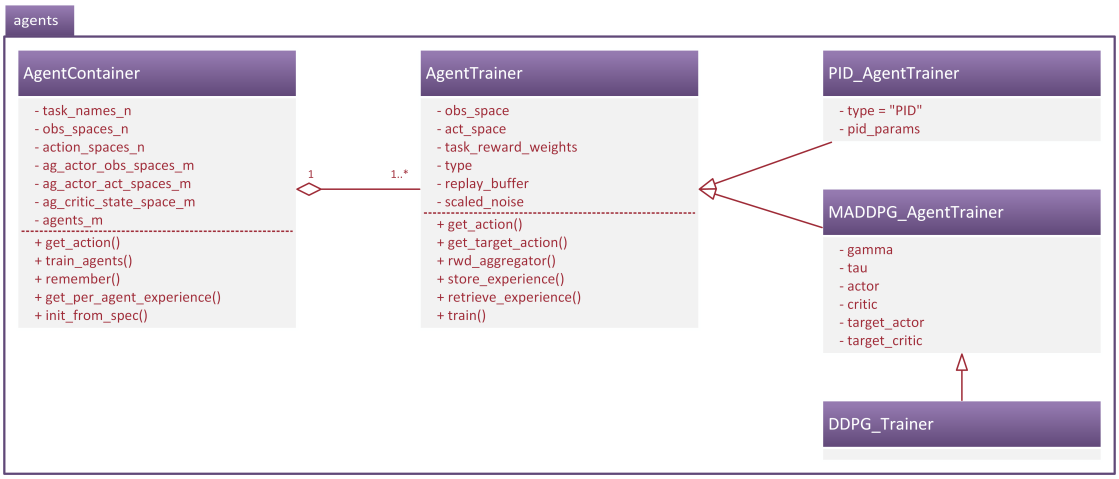

Figure 5.8: Simplified class diagram of `agents`- module.

AGENTTRAINER The `AgentTrainer` abstract base class realizes a generic RL agent for usage in multi-agent Markov-games. It provides a common interface to be used with different specializations of the agents. Before the specializations used in `Markov-Pilot` are detailed, the common interface is described in a generic way.

All classes implementing `AgentTrainer` have some initialization parameters in common:

- `name`: The name of the agent used for mapping agents to tasks.
- `obs_space`: A specification of the shape of the observation space of the agent.
- `act_space`: A specification of the shape of the action space of the agent.
- `buf_len=1000000`: The max. number of entries in the experience replay buffer. When using the MADDPG algorithm, this must be the same for all agents.
- `train_steps=1`: The starting value for counting the training steps. May differ from 1 when restoring a persisted agent to continue training.
- `task_reward_weights=None`: The aggregation weights to be used if an agent is associated with multiple tasks. If not specified, all weights are implicitly set to one.

Additional parameters may be passed to the specializations of the `AgentTrainer`-class and are explained later on.

To enable control and training, a common functional interface for all types of agents is defined:

- `get_action(obs, add_exploration_noise)`: Providing an observation, the controller's action is returned. During the training of an DDPG-like agent, exploration noise can be added like described in section 3.6.
- `get_target_action_t(obs)`: Providing an observation, the controller's target network's action is returned as a `PyTorch`-tensor. This function is used during the training of the critic like depicted in figure 3.2.
- `rwd_aggregator(rwd_list)`: Returns the aggregated reward as a weighted average using the agent's `task_reward_weights`. If multiple tasks are assigned to a single agent, this converts the multiple rewards to a single scalar value needed for training the critic in DDPG-like agents.
- `store_experience(experience)`: Stores an experience tuple (s_t, a_t, r_t, s_{t+1}) to the agent's replay buffer.
- `retrieve_experience(batch_idxs)`: Returns a minibatch of experience tuples specified by `batch_idxs` from the replay buffer.
- `train(agents_m, own_idx)`: Performs a training step on the agent. To support MADDPG agents, a list of all agents in the environment is passed. The `AgentTrainer` is informed about its own position in this list by the parameter `own_idx`.

The control and training loop implemented in the `perform_training()` function of the agents-module relies only on these common functions whatever concrete agents are instantiated in the current setting.

Additional functions to persistently store and retrieve agents including their training state are available.

PID_AGENTTRAINER To benchmark the RL based controllers developed in this research against conventional PID-control, a `PID_AgentTrainer` class was developed. This class implements a conventional PID controller, but uses the `AgentTrainer`-interface. That way, learning RL agents and static PID agents can be used together in the same environment interchangeably.

To instantiate a `PID_AgentTrainer`, an additional parameter must be specified:

- `pid_params`: A tuple with the PID parameters K_P, K_I and K_D (cf. section 4.1).

In a traditional implementation, a PID controller calculates the integral and the derivative of the error value internally and thus is stateful. As a stateful agent contradicts the Markov property which does not allow memory, the `PID_AgentTrainer` was implemented without internal memory. The integral and the derivative of the error must be added to the observation instead of calculating it internally. This is realized by the specification of the associated task like specified in section 5.2.1.

By using this trick to externalize the integral and the derivative calculation into the observation provided by the associated task, a `PID_AgentTrainer` can be used the same way like it would be a DDPG-like RL agent. As PID agents do not learn anything, but are constantly parameterized, the `train`-function is mapped to a no-op. Even though not needed for learning, the `PID_AgentTrainer` maintains a replay buffer so that it can be queried for its policy in a setup containing MADDPG-agents which query the observations and actions from other agents like shown in figure 4.7.

MADDPG_AGENTTRAINER The `MADDPG_AgentTrainer` uses a straight-forward implementation of the MADDPG algorithm introduced in section 4.7.

To instantiate an `MADDPG_AgentTrainer` object, additional parameters may be specified for the agent's internal ANNs, the learning process and the exploration noise. These additional parameters are provided with sane defaults an hence are not given in detail here. They can be looked up in the sources. They follow the schema of the original DDPG-paper [Lillicrap et al., 2015].

The only additional parameter worth mentioning is

- `critic_state_space`: A specification of the shape of the observation space presented to the agent's critic.

In the MADDPG algorithm, the critic is provided with an extended observation space depending on the observations and actions of the other agents populating the shared environment.

Depending on the parameters for `obs_space`, `critic_state_space` and `act_space`, the four internal ANNs for actor, critic, target-actor and target-critic are instantiated. In the current implementation of `Markov-Pilot` the same network topology like proposed in [Lillicrap et al., 2015] is used.

To add exploration noise to the deterministic actions a noise source for Ornstein-Uhlenbeck-noise (OU) is available. The noise values are shifted and scaled according to the shape specified in `act_space`.

The functions `get_action()` and `get_target_action_t()` are implemented as simple forward passes through the actor resp. target-actor networks.

The function `train(agents_m, own_idx)` implements the MADDPG training cycle like described in section 4.7. To train the centralized critic in the MADDPG algorithm, the experience tuples and the target actions a_{i+1} from all other agents are queried using their methods `retrieve_experience()` and `get_target_action_t()`.

DDPG_AGENTTRAINER The `DDPG_AgentTrainer` is inherited from the more general `MADDPG_AgentTrainer` with very little differences.

As `DDPG_AgentTrainer` doesn't have different observations for actor and critic, the parameter `critic_state_space` is omitted.

The `train(agents_m, own_idx)` function is implemented exactly the same way like in `MADDPG_AgentTrainer`, but the step of querying the other agent's experience and target actions is left out.

Applying these two little omissions, the MADDPG algorithm is naturally transformed back into the original DDPG algorithm.

AGENTCONTAINER The `AgentContainer` serves as a mediator between n tasks embedded in an environment and m agents embedded in the `AgentContainer`. Generally, there is a $n : m$ relation between tasks and agents,[13] but some restrictions in the assignment of tasks to agents apply.

As the assignment of tasks to agents and vice versa can become quite complex, this shall be explained before the initialization and the usage of `AgentContainer`-objects is detailed.

In the `Markov-Pilot-SW`, there exist n tasks embedded into an `JSBSimEnv_multi` and m agents embedded into an `AgentContainer`. In each step, a task emits an array of observations and a scalar reward. As input a task accepts $0 \ldots i$ action values which are entered into the simulation engine before a step is taken. An agent emits one array of action values and accepts one scalar reward per associated task.

Each agent can be associated with $1 \ldots n$ tasks. The observations of the associated tasks are concatenated by the `AgentContainer` to build the agent's observation. The rewards of the associated tasks are concatenated to the agent's reward inputs. This reward list is aggregated into a single scalar value within the agent by means of the `rwd_aggregator(rwd_list)` function.

Each agent emits an array of action values. The action space of each agent corresponds to the concatenated action spaces of the associated tasks. If actions for more than one task are emitted, the `AgentContainer` splits up the agent's actions in per-task-slices.

Each task can be associated to more than one agent, but in that case, its action space must be empty to avoid contradicting action inputs. Each task action must be associated to exactly one agent. In most setups there will be a one-on-one assignment between agents and tasks, however more complicated associations may be useful in future research like e. g. a combined glide path angle and thrust control for which the task-agent-setup is shown in figure 5.9 as an example.

All the distribution and aggregation of values to support the specified assignments between agents and tasks is done in the `AgentContainer`.

To initialize an `AgentContainer`, these parameters must be provided:

- `task_list_n`: A list of tasks that shall be associated with agents.

13 That's why parameters referring to the n tasks of the associated environment are mostly suffixed with `_n` and those referring to he m agents in the container are suffixed with `_m`.

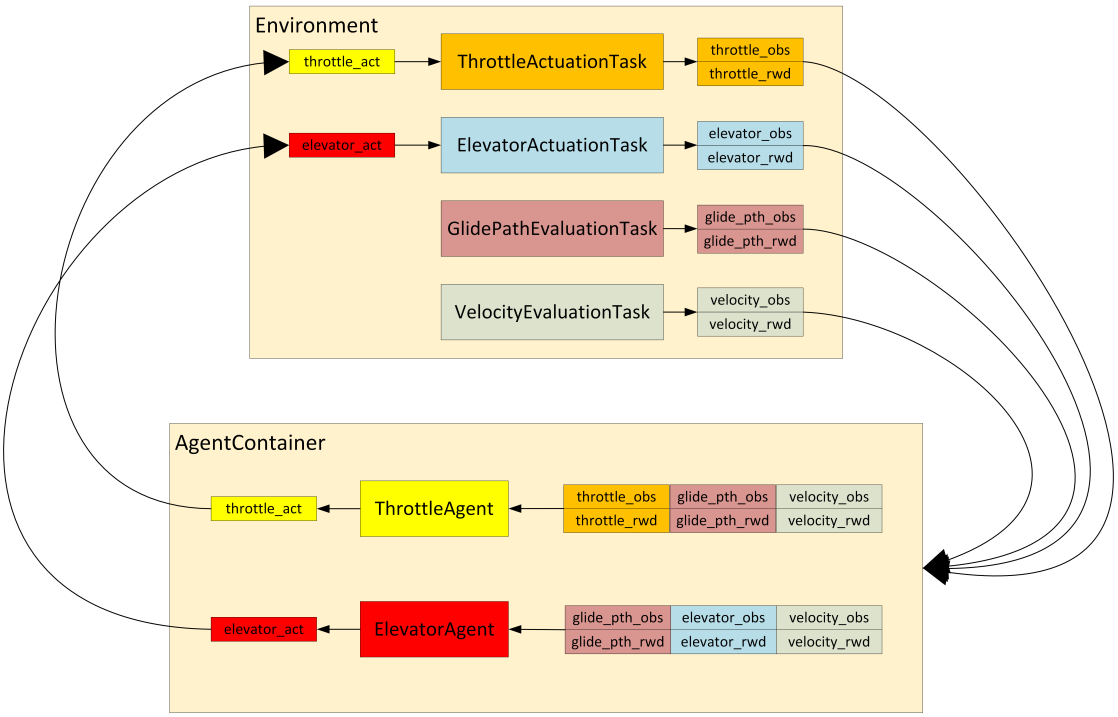

Figure 5.9: Associations of tasks to agents in a combined control task for thrust and elevator control.

- `agents_m`: A list of agents that shall be maintained by the `AgentContainer`.
- `mapping_dict`: A dictionary specifying the mapping between agents and tasks. The dictionary's keys are the names of the agents. For each agent name, a list of associated task names is provided.

On initialization, the `AgentContainer` builds up all internal structures to convey the data for observations, rewards and actions back and forth between tasks and agents. Sanity checks are performed to prevent invalid assignments.

In practice, the `AgentContainer` object is usually not instantiated directly, but via the class-function `AgentContainer.init_from_specs()`. This convenience function accepts a list of tasks and specifications for the agents. It instantiates the agent objects and the `AgentContainer` at once.

While the setup of the `AgentContainer` is a little bit fiddly, its usage is easy and straightforward. The standard operations supporting the RL interaction cycle are performed by this interface:

- `get_action(obs_n, add_exploration_noise)`: Provided with per-task observations, the agents' per-task actions are returned. During the training of an DDPG-like agents, exploration noise can be added like described in section 3.6.

- `remember(obs_n, actions_n, rewards_n, next_obs_n, dones_n)`: Given the per-task output $(s_{t,i}, a_{t,i}, r_{t,i}, s_{t+1,i})$ together with the tasks' done flags, each agent stores its associated per-agent experience to its replay buffer.
- `train_agents()`: On all agents contained in the `AgentContainer` a training step is performed.

Additionally, there exist functions to persist and restore an existing `AgentContainer` with the specifications of its contained agents.

TRAIN.PY In the agents-module, there exists one more source file `train.py` providing the single function `perform_training()`.

This function is used to perform an entire training and testing cycle for a given RL setup within the `Markov-Pilot-SW`.

Given a training environment including tasks and an agent container including RL agents, a given amount of training steps is performed and the agents are trained. Whenever an episode is over, a new episode is started till the maximum training steps are reached.

After a certain amount of training steps, the trained agents perform the standardized tests on the testing environment also provided to the `perform_training()` function. The testing environment is usually identical to the training environment, but uses its own instance of the simulation engine to not destroy the simulators state during training.

Usually, the training environment is wrapped by `VarysetpointsWrappers` to present a broad range of situations during training and the testing environment is wrapped with an `EpisodePlotterWrapper_multi` to visualize the test runs and analyze the characteristic values.

The agents in this module themselves are developed anew for this research, but the implemented interface borrows from the sources complementing the MADDPG paper [Lowe et al., 2020]. The DDPG implementation including the ANNs follows a proposal of the course series "Machine Learning with Phil" (`https://www.youtube.com/channel/UC58v9cLitc8VaCjrcKyAbrw`) but a speed up of $\approx 30\%$ was achieved wrt. his original implementation by optimizing the code.

5.2.5 *Test Bed Module*

The run control for the standardized test scenario described in section 4.3 is located in the test bed module. This module currently consists of a single function.

- `evaluate_training(agent_container, env, lab_journal=None, store_evaluation_experience=True, add_exploration_noise=False, render_mode=None)`

This function accepts an `AgentContainer` containing the agents to be tested and an environment that shall be used for the test run. A different environment than the one used for training shall be used, as the switch from training to testing may happen in

the middle of an episode and such a context switch would destroy the aircraft's state in the simulator.

Additionally, the function accepts a reference to a `LabJournal`-object which is used to record the test-runs and its results into a central `CSV`-file.

If not otherwise specified, the experience generated during the test-run is also stored to the replay buffers of the agents as this is as valuable as the experience from a training session.

The possibility to add exploration noise to the agent's action is only used for debugging as this is the most convenient way to visualize noisy actions.

By passing either `'flightgear'` or `'timeline'` in the render mode parameter, a corresponding rendering is enabled in the evaluation. If FlightGear rendering is requested, the corresponding environment must be chosen.

When called, the function `evaluate_training()` performs a test-run like specified in table 4.2. After the run, the internal state of the agents (i. e. the ANNs' parameters) are persisted to disk. The best test results of each agent are tracked and marked in the saved files.

After the test episode ends, the test run is visualized in an interactive website if the environment is wrapped by an `EpisodePlotterWrapper_multi`.

5.2.6 *Helper Module*

The helper-module is a collection of useful utility functions and classes used throughout the implementation. Most of these utilities are not worth to be mentioned, but at least a brief overview shall be given.

- `LabJournal`: A class to record the training runs with the used parameters and the yielded results in a `CSV`-file that serves as a lab journal. Especially in the experimentation phase it was extremely handy to look up the parameters and results from the trials performed the other day ...
- `load_store.py`: A collection of functions to save parameters of experiments and to restore environments including tasks as well as agent containers by referencing line numbers in the lab journal `CSV`-file.
- `networks.py`: This file contains the PyTorch-implementation of the ANNs used in the DDPG and MADDPG agents.
- `OUNoise`: This class implements the Ornstein-Uhlenbeck (OU) noise used for exploration as per the DDPG-paper ([Lillicrap et al., 2015]).
- `ReplayBuffer`: This class implements the replay buffer used in the agents. Currently there is no provision to use any prioritized experience replay.
- `utils.py`: This file is more or less a random collection of useful functions used throughout the implementation.

- `visualizers.py`: The functions in this file are called from within the environment's `render()`-function. They can render the environment data either to a real-time graph of data or to a full blown 3D-scene in FlightGear.

In the helper module, only the load and store functions and the `LabJournal`-class are entirely new developments. Most other functions and classes in the helper module stem from different sources. Most of them are referenced directly in the source code.

The function to be mentioned separately is the FlightGear rendering function, which is a take over from [Rennie, 2019].

5.3 Software Usage

After this detailed insights into the internals of the `Markov-Pilot`-SW package, a short example of its usage shall be given. In the `main.py` source file of the project's associated GitHub page `https://github.com/opt12/Markov-Pilot`, a working example can be found.

Most parameters for training and testing can be altered using command line arguments. The valid options and their default parameters are listed in the `parse_args()` function in `main.py`.

To specify the experimentation setup, two additional functions are specified in `main.py`.

SETUP_ENV In the `setup_env()` function, the environment and the tasks are specified.

Each task is specified as an instance of the `SingleChannel_FlightAgentTask` class with the needed parameters. While the specification of the action property and the set-point dictionary is obvious, the parameter for `make_base_reward_components` and for `integral_limit` deserve some advice:

The base reward components are specified by a function that yields a tuple of `RewardComponents`. This function is usually defined in an extra file to avoid frequent changes to `main.py`. A concrete examples is given in appendix D.3.

The integral limit specifies the maximum absolute value of the error integral calculated by `SingleChannel_FlightAgentTask`. For PID agents, this limit avoids the phenomenon called *integrator wind-up*. In the experiments described in section 6 it was discovered, that a similar effect is present in (MA)DDPG-like agents and that a small integral limit dramatically improves the control. While all other task parameters can be the same for PID- or (MA)DDPG-like agents, the integral limit is different by several orders of magnitude. It shall be chosen to be much smaller for the (MA)DDPG-like agents.

After the task objects are instantiated, an environment can be instantiated by passing a task list in the constructor parameters. After environment instantiation wrappers

can be applied and the initial conditions shall be set before the environment object is returned by the `setup_env()` function.

SETUP_CONTAINER In the `setup_container()` function, the agent container and the embeddeded agents are specified.

To instantiate the agent container and the agents, the `init_from_specs()` class function is used which needs a task list, a list of agent specifications and a dictionary of agent classes as parameters.

The task list is passed as a parameter to `setup_container()`.

The different task types ('PID', 'DDPG' and 'MADDPG') are identified by name in the agent specification. The mapping from these names to the different agent classes to be instantiated is given in the `agent_classes_dict`.

An agent specification is a named tuple with entries for

- `name`: Specifies the name of the agent.
- `agent_type`: Specifies the agent's type. Used as key in `agent_classes_dict`.
- `task_names`: A list of the task names to be associated with the agent. If multiple tasks are associated with a single agent, the `task_reward_weights` parameter shall be set accordingly in the `parameters`.
- `parameters`: A dictionary of additional init parameters for the agent. Usually all command line options are given here. If multiple tasks are associated with a single agent, an entry for `task_reward_weights` shall be added containing a list of the individual weights of the tasks' rewards when aggregated into a single scalar value within the agent.

For each agent to be included in the `AgentContainer`, one `AgentSpec` shall be specified.

After the agents are specified, the agent container and the agents are instantiated with a single function call to `AgentContainer.init_from_specs()`.

MAIN After the two functions `setup_env()` and `setup_container()` are defined, the only steps to start an experiment are to call them in the correct order:

```
arglist = parse_args()
#specify the lab journal
lab_journal = LabJournal(arglist.base_dir, arglist)
#create the training env and the testing env
training_env = setup_env(arglist)
testing_env = setup_env(arglist)
#apply VarySetpoints to the training to increase the variance of training data
training_env = VarySetpointsWrapper(training_env, prp.roll_deg,
                     (-30, 30), (10, 30), (5, 30), (0.05, 0.5))
training_env = VarySetpointsWrapper(training_env, prp.flight_path_deg,
                     (-10, -5.5), (10, 45), (5, 30), (0.05, 0.5))
training_env = VarySetpointsWrapper(training_env, prp.sideslip_deg,
                     (-3, 3), (10, 45), (5, 30), (0.05, 0.5))
#create the agent_container
agent_container = setup_container(training_env.task_list, arglist)
#save the experiment to the lab journal
save_test_run(testing_env, agent_container, lab_journal, arglist)
#start the training
perform_training(training_env, testing_env, agent_container, lab_journal, arglist)

training_env.close()
testing_env.close()
```

Code Listing 5.1: The setup and start of a new experiment.

If a previous training shall be replayed, the following sequence is used to load an environment and agents from the lab journal. In the given example, the loaded environment is changed from `NoFGJSBSimEnv_multi` to `JSBSimEnv_multi` and rendering to FlightGear is requested.

```
arglist = parse_args()
#specify the lab journal
lab_journal = LabJournal(arglist.base_dir, arglist)
#specify the requested lines from the lab journal
restore_lines = [61, 62, 60]
#load env from disk and change to FlightGear enabled env
testing_env = restore_env_from_journal(lab_journal, restore_lines[0], target_environment='FG')
#load trained agents from disk
agent_container = restore_agent_container_from_journal(lab_journal, restore_lines)
# perform a test-run on the environment using the trained agents and render it to FlightGear
evaluate_training(agent_container, testing_env, lab_journal=None, render_mode = 'flightgear')
```

Code Listing 5.2: The restoring of pre-trained agents from the lab journal with FlightGear rendering.

6 EXPERIMENTATION AND DEVELOPMENT OF A WORKING RL CONTROL

In the just described experimentation framework, literally hundreds of experiments were conducted to learn about possible pitfalls and a working parametrization of the environment with its flight tasks and the agents learning to fly. Of course, during the research, the creation of knowledge did not follow a waterfall like model like the outline of the text might suggest, but a lot of smaller and greater iteration loops were taken.

In this section, the most important findings during the experimentation with the developed SW shall be described. The aim of this section is to not only show the end results and how they work, but to explain why certain parameters were chosen and why they work the way they do.

EXPERIMENTAL SETUP The studied scenario is a gliding descent with engine off. During the descent, the glide path and the banking angle shall be controlled to follow a desired path to eventually arrive at a desired landing field. Turns shall be performed in a coordinated manner, i. e. the sideslip induced by banking the aircraft shall be controlled to zero by means of the rudder.

The path planning and calculation itself is not in the focus of this research. Further information on planning gliding descents along Dubins paths can be found in [Eckstein, 2018] or in [Klein et al., 2018].

Two blocks of experiments were conducted: Firstly, the path towards a sensible reward function for RL in flight control is illustrated. Using the developed reward structure, first attempts were undertaken to train RL agents for 3-axes low-level flight control.

6.1 Reward Engineering

Engineering a reasonable reward function is crucial to all reinforcement learning. The reward defines the ultimate goal of the learning agent and the structure of the reward guides the agent during the learning process.

To gain a thorough understanding of a most suitable reward function, the research on reward engineering started with a single learning agent controlling either the elevator or the aileron. The associated task's reward was adjusted to guide the agent through a successful training.

To enable such an approach, in a preliminary investigation, the parameters for PID agents like described in section 5.2.4 were identified. These PID agents serve as a benchmark for the RL agents. During the experimentation with one single RL agent, a PID agent was used for the *other* axis to allow a stable flight attitude. The rudder was kept at 0-position during these experiments.

In figure 6.1, the performance of the PID controllers on both axes is shown. The code snippets to instantiate such a double PID control as well as the characteristic performance values and settling times can be found in appendix D.1.

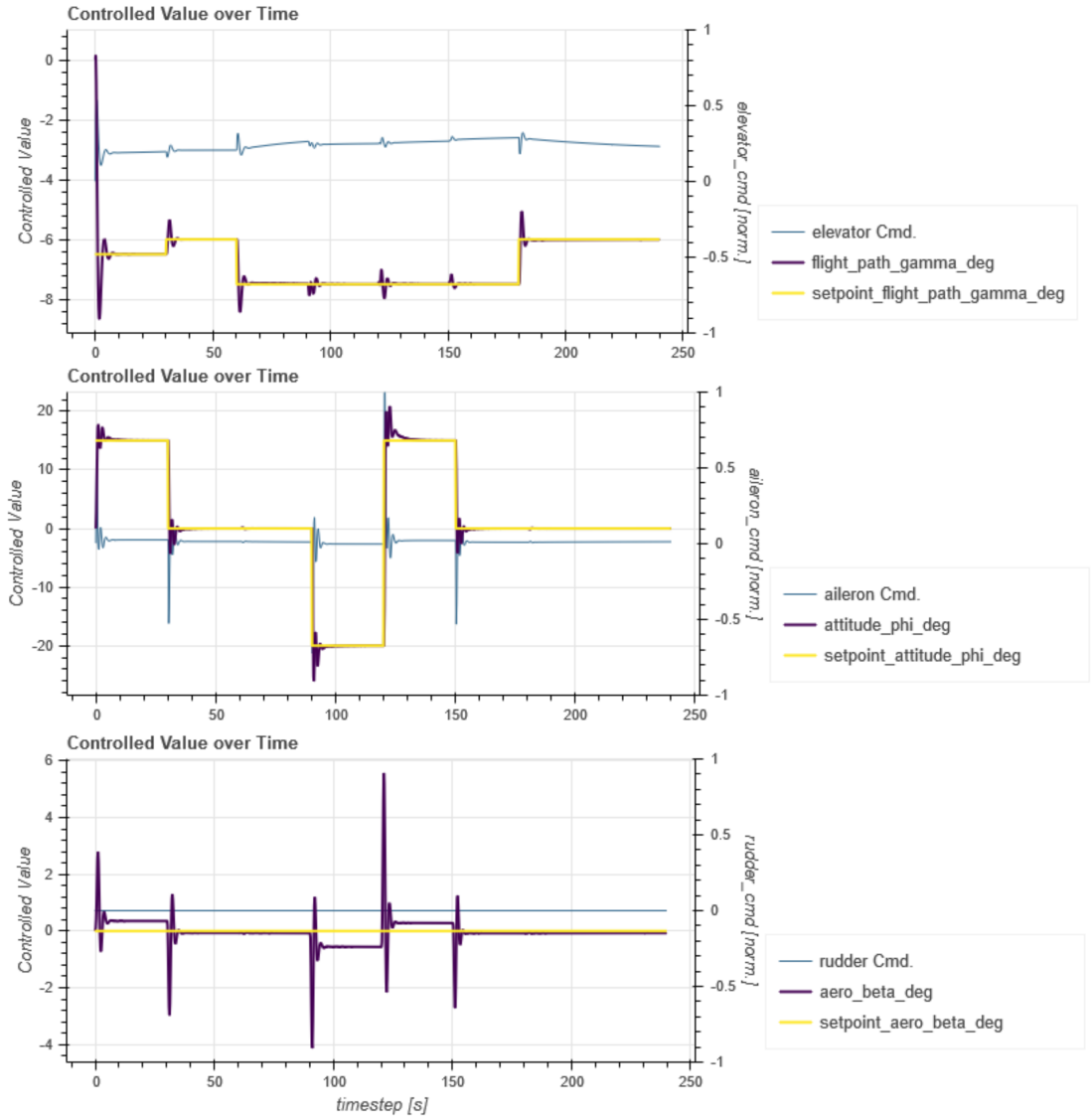

Figure 6.1: Standardized test scenario controlled by two PID controllers and the rudder held in zero-position.

From experience it is known, that the glide path angle control using the elevator is a lot more complicated than the banking angle control using the aileron. For this reason, the reward engineering experiments start with exactly the more complicated glide path angle control task, as it is assumed, that a reward structure covering the more complicated case can also cover the easier case.

If not otherwise noted, for the rest of this section 6.1, the agents used in the experiments are a DDPG-agent for the elevator control and a PID-agent to control the aileron. The rudder is left untouched in neutral position.

Generally, all trainings were performed over 30000 training steps with a standardized test run every 2000 training steps. If not stated differently, the plots show the agent's control after 30000 training steps.

All the training was performed with the `VarySetpointsWrapper` applied to all set-points during training. This random and uncorrelated variation of the set-points was found to be highly effective to speed up the training, as a lot of different aircraft states and attitudes were presented to the learning agent.

REWARD BASED ON ERROR-ONLY As a first naïve idea to minimize the tracking error, it's almost self-evident to reward the agent for a minimum error between desired set-point and the actual glide path angle. The only observations presented to the agent are the error value, its integral and its derivative. The `AsymptoticErrorComponent` is used with a scaling factor of 0.1 to achieve accurate control in the reward calculation.

The result of this first approach is shown in figure 6.2 top left. This control is obviously useless, as the agent only learns some wild bang-bang control. This however is no real surprise, as the agent has no information about the aircraft attitude at all.

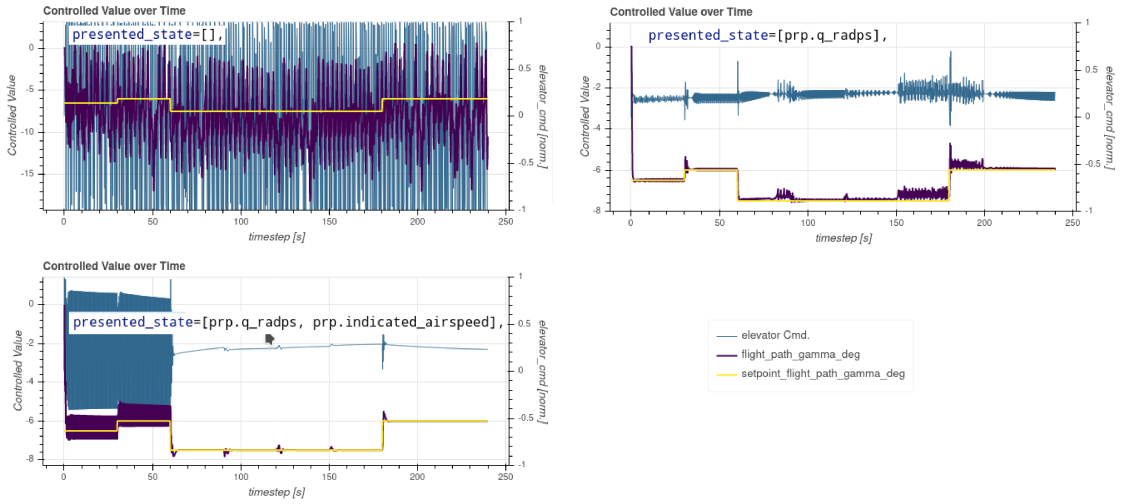

Figure 6.2: Glide path angle control with reward based only on the tracking error. Different additional observations were present in the observation.

From physical considerations, it is mandatory to have information on the aircraft's angular velocity around the controlled axis. In a smooth flight, this angular velocity shall be minimized.

In the case of glide path angle control, the angular velocity around the pitch axis is given in the property `prp.q_radps`. In a next experiment, this information was added.

Even though the value of q is not incorporated into the reward calculation, the agent behaves way better with this additional state information.

The result of additionally presenting the angular velocity q to the agent is shown in figure 6.2 top right.

Now there is recognizable control and the controller actually follows the set-point. However, there is still a problem that the controller tends to flitter[14] on the actuator. Of course such a control strategy is not desirable in real aircraft control.

In a next experiment, another state component, the `prp.indicated_airspeed`, was added to the agent's observation. The idea to do so is the assumption that the effectiveness of the control surfaces changes with the speed of the flowing air. The result is shown in the graph in figure 6.2, lower left. With this additional state information, another step forward to a working control is made. Even though in the beginning, there is still enormous thrashing on the actuator, there is also good control in other times of the test. In the testing segments with smooth control, the accuracy is even good enough to settle within an error band of $\pm 0.01°$.

Obviously the agent needs a certain minimum amount of state information to learn useful assumptions on the causes and effects of its observations and actions. Therefore, in further experiments more state information was added. Also the amount of training steps was tentatively increased. However, there was no way that the agent learned a consistently good control. From time to time, the observed high frequency oscillations on the actuator reappeared with no reproducible reason. Eventually the observation content was frozen containing the error, the error integral and derivative, the angular velocity q and the indicated airspeed.

The main takeaway from these first experiments with a single error-only reward component can be summarized as:

- A certain minimum amount of state information on the aircraft's attitude is needed to train an agent. It's helpful to incorporate physical considerations in the selection of observation properties.
- More state is not automatically better. Even with relatively few observation components the control headed into the right direction. With more state information alone, no consistently good control could be achieved. In some experiments with a lot of state, a lot more training steps were needed till the training started to converge. It takes as long for an agent to learn to ignore some state properties as it takes to learn how to leverage it.
- Whatever combination of state was presented to the agent, the controller sometimes applied high frequency oscillations to the control. The trigger to switch into this flittering-mode is unclear.

14 The verb "to flitter" is used to denominate high frequency oscillations in the action output. "Thrashing" or "flittering" is used to indicate that the agent is in the state of issuing high frequency oscillations on its output.

ACTIVE OSCILLATION-SUPPRESSION As it was impossible to reliably avoid the unwanted oscillations appearing on the actuator, the next step was to develop an active oscillation suppression.

An additional reward component was to be developed that punishes the ocurrence of oscillations and rewards smooth control. As a criterion to detect oscillations, the actuator travel from one step to another was selected. This movement from the current action to the last action is calculated as custom property `prop_delta_cmd` by the `SingleChannel_FlightTask` class used for the task definition. A linear error component based on this criterion was added to the reward calculation. The maximum absolute value of `prop_delta_cmd` is 2 as the action itself is normalized to the range of $[-1, 1]$. This maximum value was selected as the scaling factor which results in a reward of $r = 1$ for no actuator movement and a reward of $r = 0$ if the actuator moves full way within one step.

The first attempt to suppress the tendency for oscillations achieved exactly this objective. In figure 6.3, the actuator is moved smoothly and flittering occurred only for a short period in the starting phase of the training. Unfortunately, the actuator in general became "lazy", i. e. it prefers not to move at all, which results in a horrible tracking accuracy.

In the lower panel of figure 6.3 the reason for this preference of standstill over set-point tracking can be identified. In this first attempt the reward for moving the actuator smoothly is equally weighted to the reward for minimizing the error. This effectively results in a "better-don't-move" strategy that still leads to relatively high rewards even if the tracking error is enormous.

The agent only receives the aggregated reward shown in yellow. The individual reward components are only visible in the plots for analysis, but are not known to the learning agent.

To adjust the balance between oscillation suppression and set-point tracking, in the next experiment the weights from `rwd_Angle_error` to `rwd_cmd_travel_error` were set to 80/20. The result of this readjustment is shown in figure 6.4. With this weight relations, the tendency for flittering is still sufficiently reduced, while the agent follows its main goal of tracking the set-point.

Unfortunately, the accuracy is still far from being satisfactory. Additional experiments were conducted with carefully adjusted weights, but it was impossible to find a setting that accomplishes the objective of a high accuracy control in conjunction with an effective oscillation rejection.

One further improvement was found by reducing the scaling factor from 2 to 0.5 for the `rwd_cmd_travel_error`. This yields better differentiated rewards in the areas of small movements and for actuator movements above 0.5 there is no additional penalty for even stronger moves to minimize the tracking error fast. This positive effect on the tracking accuracy can be seen in figure 6.5.

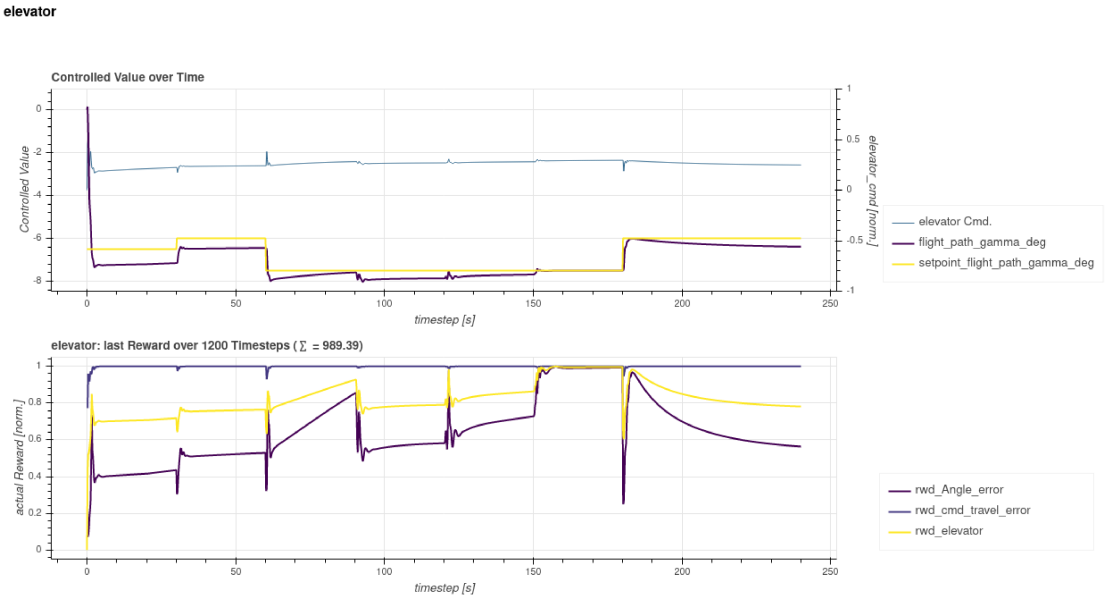

Figure 6.3: First attempt with oscillation suppression. Due to the equal weights of oscillation punishment and error tracking reward, the agent prefers to not move the actuator resulting in extremely poor tracking accuracy. The lower panel shows the reward components for error minimization and actuator movement.

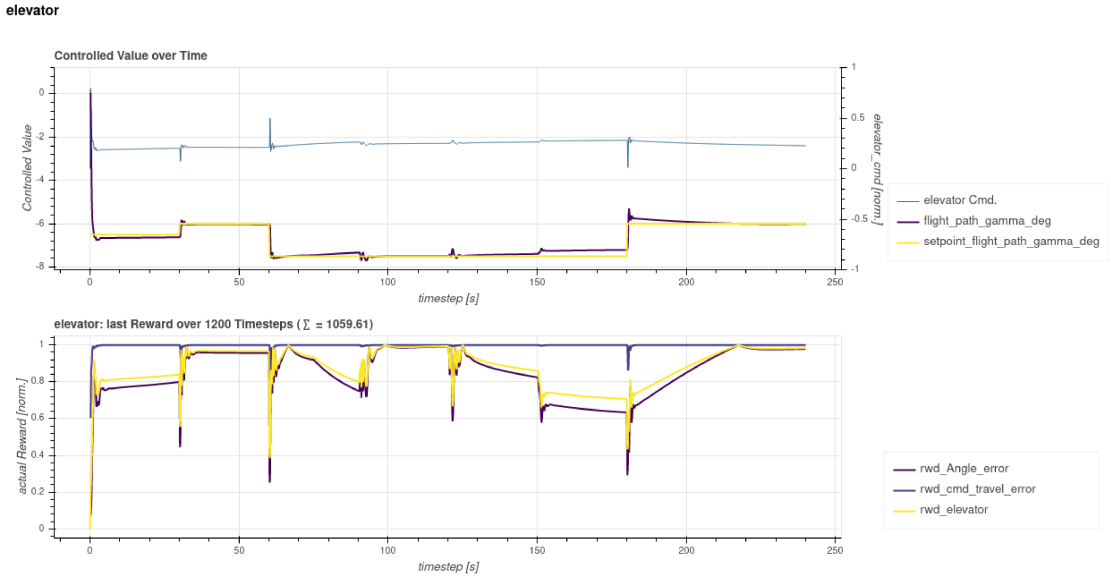

Figure 6.4: After adjusting the weight relations between oscillation punishment and error tracking reward, the agent tries harder to achieve the set-point.

The main takeaway from these experiments with active oscillation suppression can be summarized as:

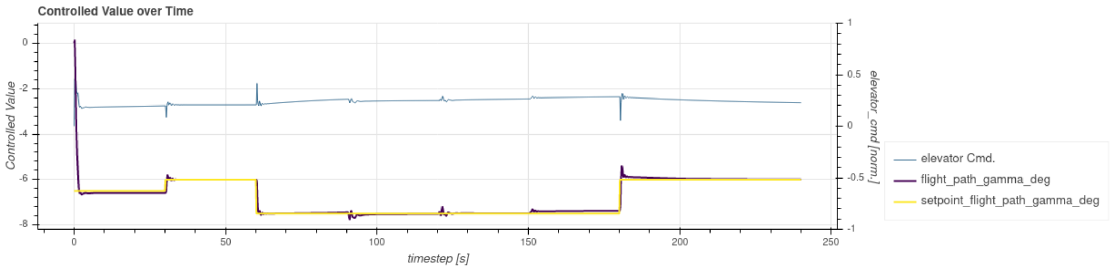

Figure 6.5: By reducing the scaling factor of the `rwd_cmd_travel_error`, the accuracy of set-point tracking was positively affected.

- An additional reward component to punish excessive actuator movements is an effective means to mitigate the agent's tendency for high frequency oscillations.
- Better reward differentiation in the areas of interest can be achieved by adjusting the scaling factors in the reward components. This can lead to better control accuracy.

INTEGRAL COMPONENT TO MINIMIZE TRACKING ERROR So far, the accuracy of the control was not satisfactory. In the best agents so far, the actual value comes near the set-point, but after reaching a certain undefined threshold, it either stops improving or even deviates again. There is obviously not enough pressure on the agent any more to further reduce the steady state error.

Comparing this to an ordinary PID controller, it is striking that the PID-based controller performs way better regarding the steady state error. A direct comparison of the steady state behavior of RL-based and PID-based agents is shown in figure 6.6.

The PID controller tries to minimize the steady state error due to the otherwise increasing integral part of its error term. This observation gives rise to the idea of adding some integral part to the reward components of the RL agent. An integral part that sums up over time would result in an ever decreasing reward if the steady state error would not be reduced over time. Such an integral component is the basis of the next set of experiments.

The effect of the integral component in the reward can be observed even in the very first experiment with an arbitrarily chosen scaling factor of 5 for the integral's linear error component. The steady state error is considerably reduced as can be seen in figure 6.7.

There is however a downside of the integral part like used in this experiment: The overshoot after set-point steps is increased dramatically. This is probably due to the fact, that in the moment of the set-point change a huge value is accumulated in the integrator which is compensated by a fast, but huge overshoot to the other side of

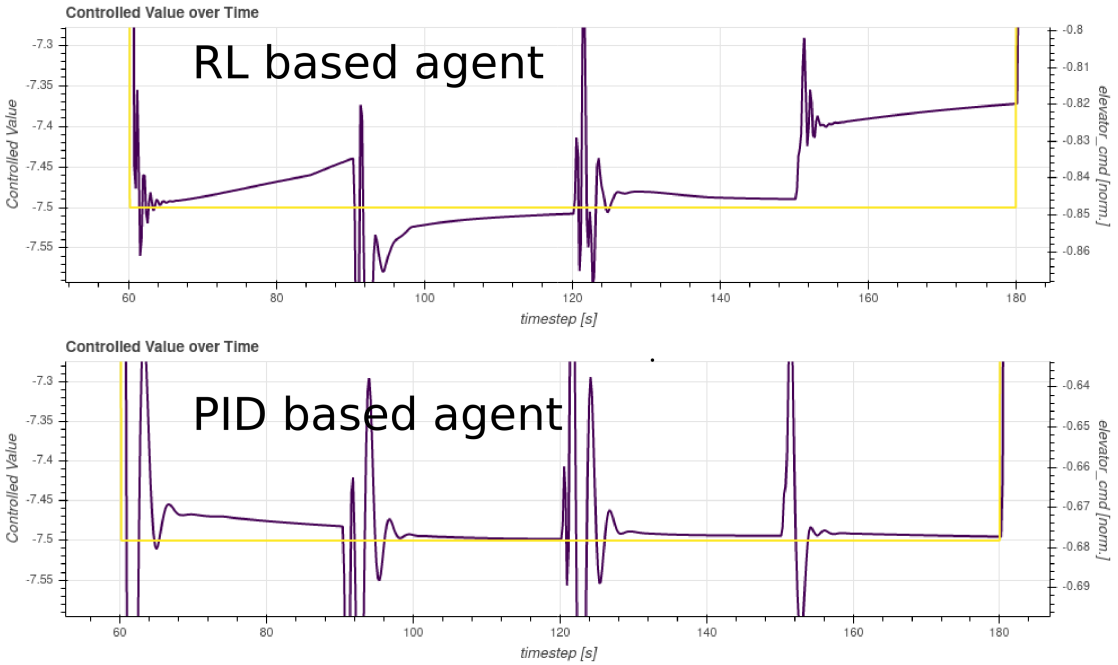

Figure 6.6: Comparison of the steady-state behavior of an RL agent (top) and a PID agent (bottom). The RL agent does not attempt to minimize the error any further.

the set-point. This is very similar to the phenomenon called *integrator-wind-up* known from conventional PID control.

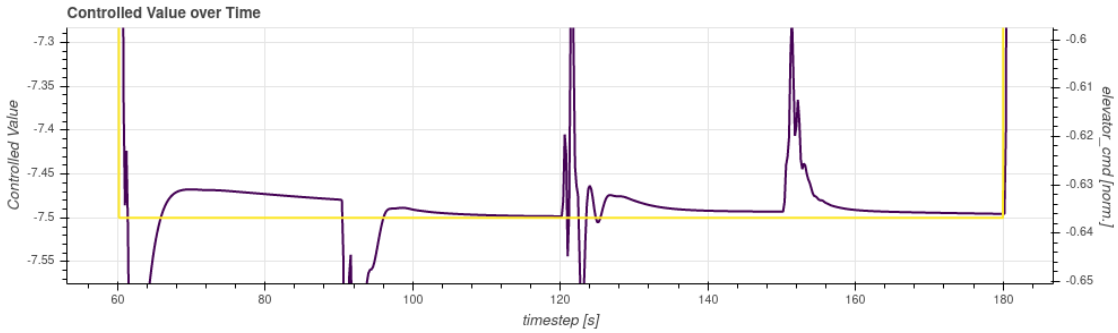

Figure 6.7: RL controller with an integral reward component. The steady-state accuracy is increased considerably compared to fig. 6.6 top, but also the overshoot increases.

Not only the phenomenon is similar to the PID case, but also the countermeasure: To minimize the effect of integrator-wind-up, the integral shall be limited to avoid huge accumulated values. However, there is a big difference in the exploitation of the integral component between PID and RL-based control. In PID control, the integral part must be non-zero to keep the actuator in the correct position by the mediation

of the factor K_I. This works only with a residual value kept in the integrator during steady state operation. If the integral limit is too small, the PID-based controller cannot eliminate the steady state error anymore and accuracy gets poor.

In contrast, the RL-based controller tries to eliminate the integral part to earn a higher reward. In RL-based control, limiting the integral helps reducing the integral near zero level fast, as the disintegration of the integral value starts at a lower level. The integral limit can be lowered to very small values. This has the effect of minimizing overshoot and making the control extremely accurate, as the reward can be differentiated very good in the region of interest when the set-point is already nearly met.

In figure 6.8 the RL-based controller with an integral reward component and limited integrator is shown. Due to the limitation, the overshoot is reduced and at the same time the accuracy is increased. In the close-up in the lower half of the figure, it can be recognized, that the accuracy goes down in the thousands of degrees which is more a theoretical than a useful value for flight control.

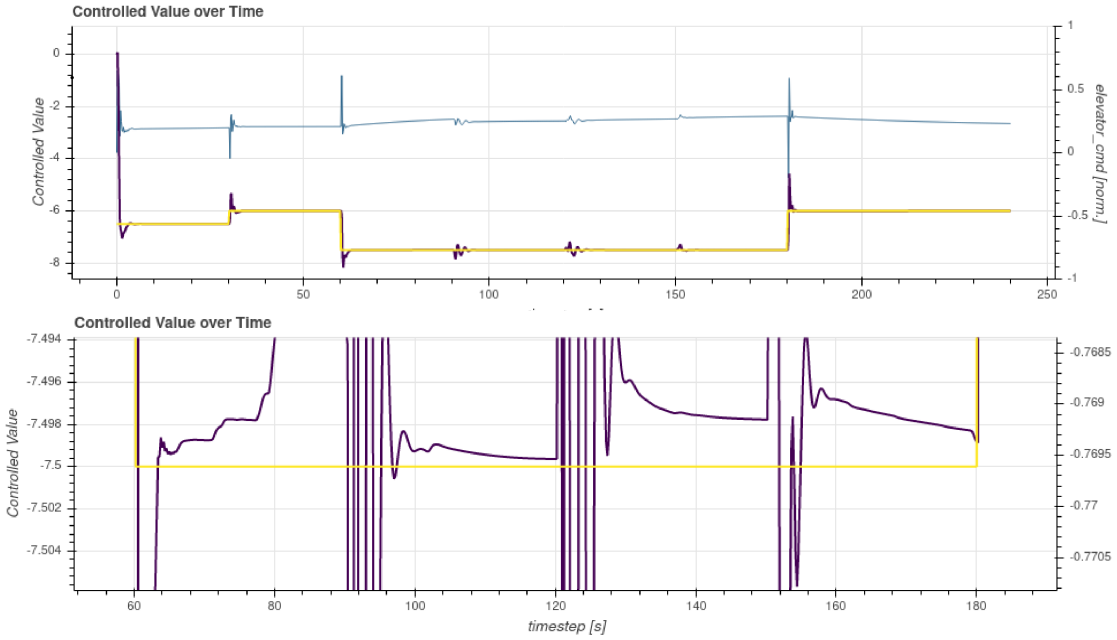

Figure 6.8: RL controller with an integral reward component that is limited to small integral values. The overshoot –especially at the initial set-point change– is reduced while at the same time the accuracy is increased. The lower part is a close-up to show the remaining steady-state error.

At the example of integral- and hence reward-clipping, the effect of this clipping-kind of non-linearity in the reward structure can be nicely seen. If the base value for the reward component is out of range, the reward stays constantly at $r = 0$ which reduces

the overall reward but makes no further differentiation possible. In these phases, the agent obeys the guidance of the other reward components regardless on how far the base value of the clipped reward is away from its own target.

Only if the integral's base value reaches the region of interest, the associated reward kicks in and puts more reward on the agent. Only after reaching this region, the agent learns how to modulate it's action to also increase this reward component.

This effect can be exploited to build up cascading rewards: Some reward components contribute to the overall reward all the time and guide the agent to learn basic control. This basic control only needs to bring the state into the regions, where the so far clipped reward kicks in. From now on, this reward can guide the agent further to increase the accuracy of control.

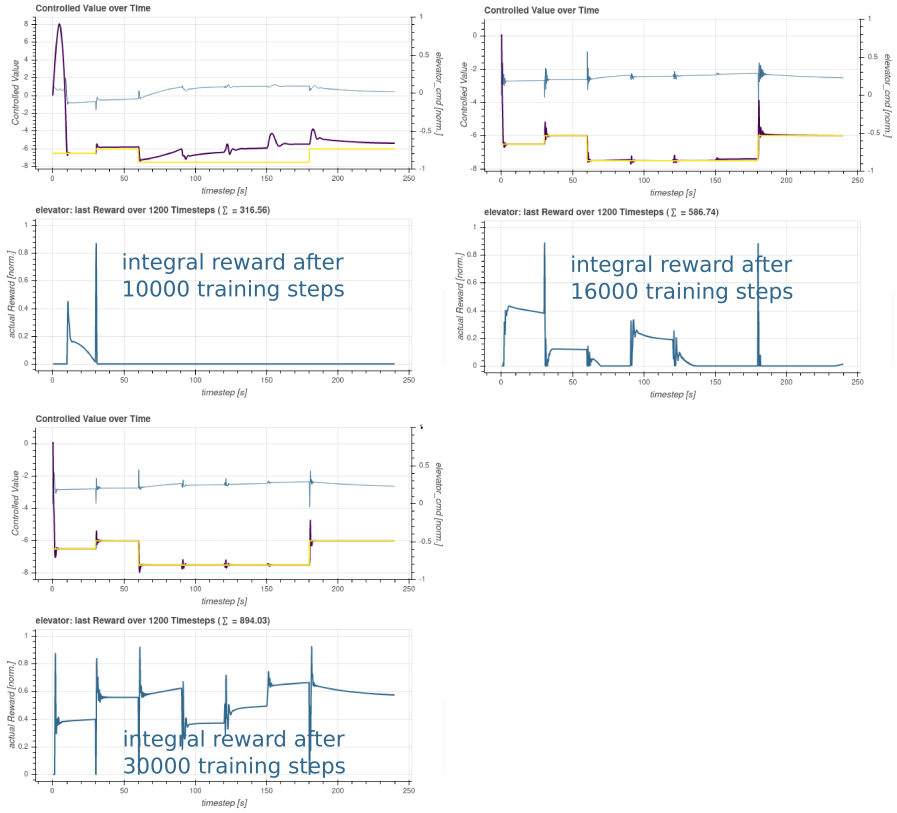

Figure 6.9: Influence of the integral reward component in different phases of training: Earlier in the training (top left), the integral reward stays at $r = 0$ for most of the time. Later the agent manages to gain integral reward at all (top right). Over the long run (bottom left) the agent learns to keep the set-point tracking accurate enough to permanently earn integral reward.

In figure 6.9 this behavior is shown over the course of agent training with integral components. The agent was trained with an integral reward component and the in-

tegral clipped to a small value of 0.25. In earlier training phases, the agent doesn't reach regions where the error integral decreases. So the integral reward component is mostly zero. In the middle of the training it reaches non-zero regions regularly and learns that staying in those regions is desirable to maximize the overall reward. At the end of the training, the agent manages to almost always stay in the regions with active integral reward and tries to maximize it.

This cascaded reward design leads to an overall fast control with extremely high accuracy.

The main takeaway from these experiments with an additional integral reward component can be summarized as:

- A reward component based on the error integral can increase the steady-state accuracy.
- While steady-state performance is increased, the overshoots after set-point changes or disturbances also increase. This is especially true, if the integral value is not limited.
- Limiting the integral value can further increase the accuracy dramatically and at the same time mitigate the tendency to overshoot.
- Introducing non-linearities with clipped reward components into the reward structure enables the construction of cascaded rewards. In the beginning only coarse control is learned till the clipped reward components start contributing and guide the agent towards higher accuracy.

6.2 Generic Reward Structure

With the development of the three reward components in the last section, a construction kit for the definition of sub-tasks in flight control is available. Based on a generic reward structure comprising of these components, the tasks' individual properties can be factored in by adjusting the components' weights and parameters.

The generic reward structure consists of these components:

- Error Component: A pure error-based `AsymptoticErrorComponent`. Guides the agent into the correct direction to follow the set-point. This component is mostly responsible to push the agent towards a coarse control. By adjusting the scaling factor, the attraction strength of the set-point can be adjusted.
- Oscillation Suppression: A `LinearErrorComponent` punishing the excessive movement of the associated actuator. Reduces the tendency of high frequency oscillations in the output actions. The importance of oscillation suppression relative to the necessary actuator movement for set-point tracking must be carefully balanced by adjusting the relative weight of this component.
- Integral Component: A `LinearErrorComponent` based on the clipped error integral. Due to the clipping, this component works in a cascaded way together

with the error component to increase the steady state accuracy and to reduce the settling times. As soon as the set-point tracking error falls below a certain deviation, the integral component starts contributing to the overall reward and pulls the agent towards finalizing the set-point tracking with very high accuracy. The integral limit defines the error bandwidth that must be undercut till the integral kicks in. A small limit also helps in reducing the overshoot introduced by the integral component. The weight of the integral component can be chosen relatively high as it starts contributing only after the agent already learned some basic control and is primarily used to increase the accuracy.

In a final step of the reward engineering experiments, the final task definition for glide path and banking angle control as well as sideslip compensation shall be derived to be used in the upcoming section 6.3 on jointly training a comprehensive 3-axes controller for the gliding descent scenario.

To figure out the parameters and weights defining the individual sub-tasks, for each of them a single-channel DDPG agent was trained while the other channels were held in stable conditions either by conventional PID-control (glide path and banking angle) or by fixing the associated action to zero (sideslip angle). The resulting parameters and weights for the task definitions are given in table 6.13 at the end of this section.

TASK DEFINITION FOR GLIDE PATH ANGLE CONTROL The training of an elevator controller to maintain the desired glide path angle already served as a guinea pig in the development of the generic reward structure. Therefore these former results are just taken over.

The control resulting from isolated training of a DDPG agent on the glide path angle control task definition is shown in figure 6.10. The characteristic values and settling times achieved by this controller are given in table 6.1.[15] For reference a comparison with conventional PID control is included.

TASK DEFINITION FOR BANKING ANGLE CONTROL Exactly the same reward structure was now hooked into the definition of the banking angle control task while the glide path angle control was changed to PID control. Of course, the angular velocity presented to the agent was changed to the roll velocity p instead of the pitch velocity q. In this setup a DDPG agent was trained without further changes to the parameters and already the first conducted experiment was successful.

The control resulting from isolated training of a DDPG agent on the banking angle control task definition is shown in figure 6.11. The characteristic values and settling times achieved by this controller are given in table 6.2. For reference a comparison with conventional PID control is included.

15 'NaN' in the relative overshoot row is stated if there was no own set-point change but a disturbance from another channel.

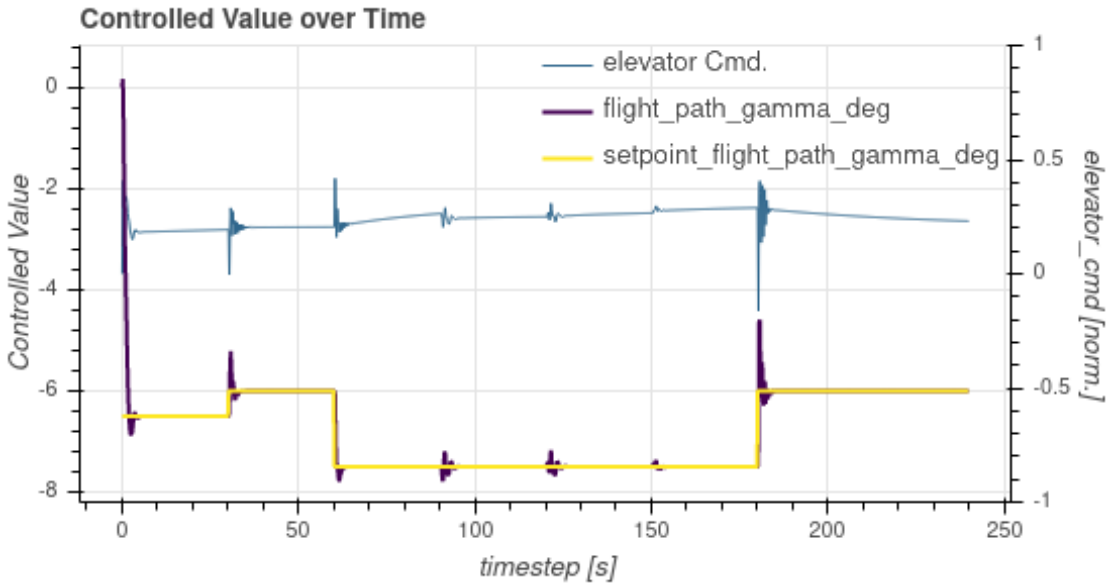

Figure 6.10: Trained agent for glide path angle control.

Characteristic values of control between events:

event #	0			1			2			3			4			5			6		
	RL control	PID control	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID
event_time	0	0		30.2	30.2		60.2	60.2		90.2	90.2		120.2	120.2		150.2	150.2		180.2	180.2	
peak_time	2.6	1.8		30.8	31.6		61.6	61.6		91.4	91		121.6	123		151.4	151.4		180.8	181.4	
delay_secs	2.6	1.8		0.6	1.4		1.4	1.4		1.2	0.8		1.4	2.8		1.2	1.2		0.6	1.2	
setpoint	-6.5	-6.5		-6	-6		-7.5	-7.5		-7.5	-7.5		-7.5	-7.5		-7.5	-7.5		-6	-6	
actual_value	-6.866	-8.669		-5.215	-5.336		-7.787	-8.443		-7.199	-7.894		-7.169	-7.98		-7.373	-7.18		-4.597	-5.048	
setpoint_change	-6.5	-6.5		0.5	0.5		-1.5	-1.5		0	0		0	0		0	0		1.5	1.5	
abs_overshoot	-0.366	-2.169	16.9%	0.785	0.664	118.2%	-0.287	-0.943	30.4%	0.301	-0.394	-76.4%	0.331	-0.48	-69.0%	0.127	0.32	39.7%	1.403	0.952	147.4%
rel_overshoot	0.056	0.334	16.8%	1.569	1.327	118.2%	0.191	0.629	30.4%	NaN	NaN		NaN	NaN		NaN	NaN		0.935	0.635	147.2%
abs_mean	0.24	0.288	83.3%	0.024	0.039	61.5%	0.034	0.094	36.2%	0.015	0.028	53.6%	0.015	0.04	37.5%	0.005	0.02	25.0%	0.029	0.043	67.4%
MSE	1.113	1.096	101.6%	0.01	0.016	62.5%	0.032	0.064	50.0%	0.003	0.006	50.0%	0.002	0.011	18.2%	0	0.003	0.0%	0.027	0.028	96.4%
actuation_energy	0.174	0.207	84.1%	0.125	0.002	6250.0%	0.107	0.013	823.1%	0.007	0.001	700.0%	0.006	0.002	300.0%	0.001	0		0.709	0.014	5064.3%

Settling times after events given in seconds:

event #	0			1			2			3			4			5			6		
	RL control	PID control	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID
setpoints	-6.5	-6.5		-6	-6		-7.5	-7.5		-7.5	-7.5		-7.5	-7.5		-7.5	-7.5		-6	-6	
setpoint_changes	-6.5	-6.5		0.5	0.5		-1.5	-1.5		0	0		0	0		0	0		1.5	1.5	
error band: 0.5	1.8	4	45.0%	0.8	1.8	44.4%	0.6	2	30.0%	0	0		0	0		0	0		1.4	1.8	77.8%
error band: 0.1	3.2	6.2	51.6%	2.8	3.8	73.7%	2	4	50.0%	3.2	4.4	72.7%	3.4	4.6	73.9%	1.4	3	46.7%	3.2	3.4	94.1%
error band: 0.05	3.8	6.4	59.4%	2.8	4	70.0%	2.4	4.2	57.1%	3.4	5.4	63.0%	3.6	5.6	64.3%	1.4	3.4	41.2%	3.8	3.6	105.6%
error band: 0.01	5.2	8.6	60.5%	4.8	5.6	85.7%	3.6	didn't settle		4.6	7.4	62.2%	5.8	10.6	54.7%	3	6.6	45.5%	4.8	46.8	10.3%

Table 6.1: Characteristic values and settling times for the glide path angle controller from figure 6.10. Values of the PID control from figure 6.1 are given for comparison.

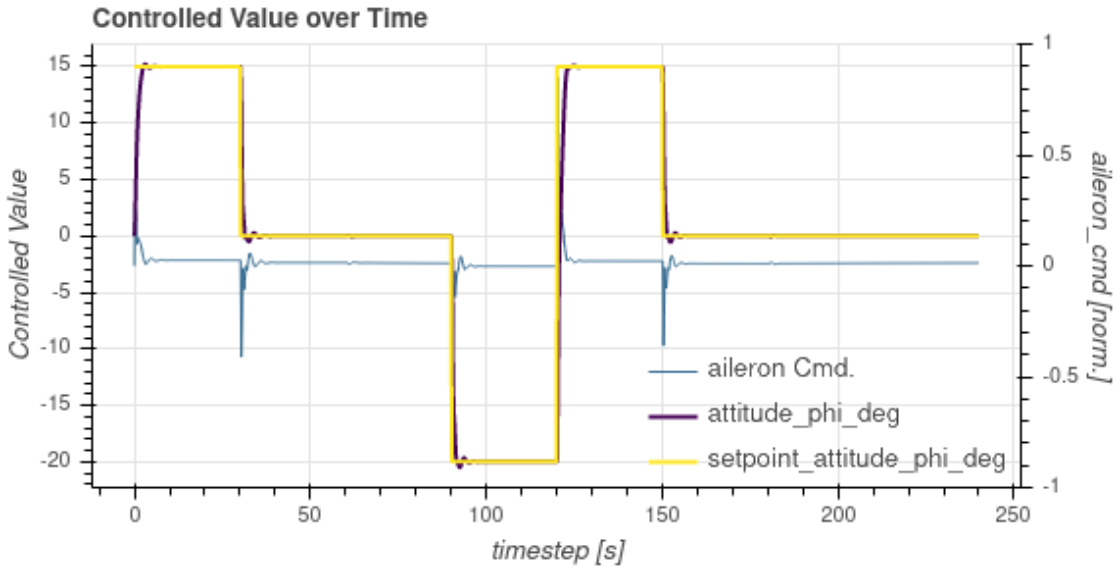

Figure 6.11: Trained agent for banking angle control. The reward structure was not altered wrt. the training of the glide path angle controller.

Characteristic values of control between events:

event #	0			1			2			3			4			5			6		
	RL control	PID control	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID
event_time	0	0		30.2	30.2		60.2	60.2		90.2	90.2		120.2	120.2		150.2	150.2		180.2	180.2	
peak_time	3	1		32.6	31		61	61.2		92.4	91		124.8	122.6		152.4	151		180.8	181	
delay_secs	3	1		2.4	0.8		0.8	1		2.2	0.8		4.6	2.4		2.2	0.8		0.6	0.8	
setpoint	15	15		0	0		0	0		-20	-20		15	15		0	0		0	0	
actual_value	15.213	17.802		-0.523	-4.364		0.064	0.147		-20.511	-26.11		15.102	20.908		-0.568	-4.278		-0.069	-0.134	
setpoint_change	15	15		-15	-15		0	0		-20	-20		35	35		-15	-15		0	0	
abs_overshoot	0.213	2.802	7.6%	-0.523	-4.364	12.0%	0.064	0.147	43.5%	-0.511	-6.11	8.4%	0.102	5.908	1.7%	-0.568	-4.278	13.3%	-0.069	-0.134	51.5%
rel_overshoot	0.014	0.187	7.5%	0.035	0.291	12.0%	NaN	NaN		0.026	0.305	8.5%	0.003	0.169	1.8%	0.038	0.285	13.3%	NaN	NaN	
abs_mean	0.474	0.417	113.7%	0.295	0.415	71.1%	0.004	0.015	26.7%	0.373	0.516	72.3%	1.132	1.031	109.8%	0.289	0.381	75.9%	0.002	0.007	28.6%
MSE	4.527	3.015	150.1%	2.818	2.859	98.6%	0	0.001	0.0%	4.647	4.852	95.8%	23.19	16.673	139.1%	2.781	2.666	104.3%	0	0	
actuation_energy	0.089	0.413	21.5%	0.289	0.466	62.0%	0	0		0.752	0.872	86.2%	1.041	1.399	74.4%	0.222	0.489	45.4%	0	0	

Settling times after events given in seconds:

event #	0			1			2			3			4			5			6		
	RL control	PID control	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID
setpoints	15	15		0	0		0	0		-20	-20		15	15		0	0		0	0	
setpoint_changes	15	15		-15	-15		0	0		-20	-20		35	35		-15	-15		0	0	
error band: 0.5	2.2	5.6	39.3%	2.6	3.4	76.5%	0	0		2.6	4.6	56.5%	2.8	8.2	34.1%	2.4	3	80.0%	0	0	
error band: 0.1	4.6	9.4	48.9%	5.8	8.2	70.7%	0	1.4	0.0%	4.2	8	52.5%	4.8	13.4	35.8%	4.2	7.4	56.8%	0	1.2	0.0%
error band: 0.05	6	11.8	50.8%	7.2	10.2	70.6%	1.8	2.6	69.2%	5.6	10.4	53.8%	5.2	15.8	32.9%	5.4	9.4	57.4%	1.8	2.2	81.8%
error band: 0.01	8	17	47.1%	9.2	15.2	60.5%	4	21.4	18.7%	8.2	15.2	53.9%	8	21.8	36.7%	8	14	57.1%	3.6	4.8	75.0%

Table 6.2: Characteristic values and settling times for the banking angle controller from figure 6.11. Values of the PID control from figure 6.1 are given for comparison.

TASK DEFINITION FOR SIDESLIP COMPENSATION (TURN COORDINATION) Finally the task definition for sideslip is still missing. For these experiments, the first time three agents issue actions on the environment. The elevator and the aileron are controlled by their respective PID controllers, while the rudder is controlled by a DDPG agent.

In the experiments, it became obvious, that the sideslip compensation task is missing some state. As the sideslip is a direct consequence of banking, it became evident during the experiments, that the set-point and the actual value for the banking angle ϕ shall be presented to the agent as additional information. "The rudder is entirely subservient to the aileron and is used because we use ailerons" [Langewiesche and Collins, 1972, p. 184]. Adding this information considerably improved the rudder control.

While the reward structure could be left unchanged, the individual weights and parameters in the sideslip compensation task needed adaptation: The rudder has an extreme preference for flittering and so the weights for the oscillation suppression need a bigger relative weight. It was also necessary to raise the scaling factor for the error component to have a softer reward transition towards small errors.

The control resulting from isolated training of a DDPG agent on the sideslip compensation task definition is shown in figure 6.12. The panel uses the same scale as in figure 6.1 to enable a fair comparison. The characteristic values and settling times achieved by this controller is given in table 6.3. No comparison data to PID control is given, as no PID-based controller is available for sideslip compensation.

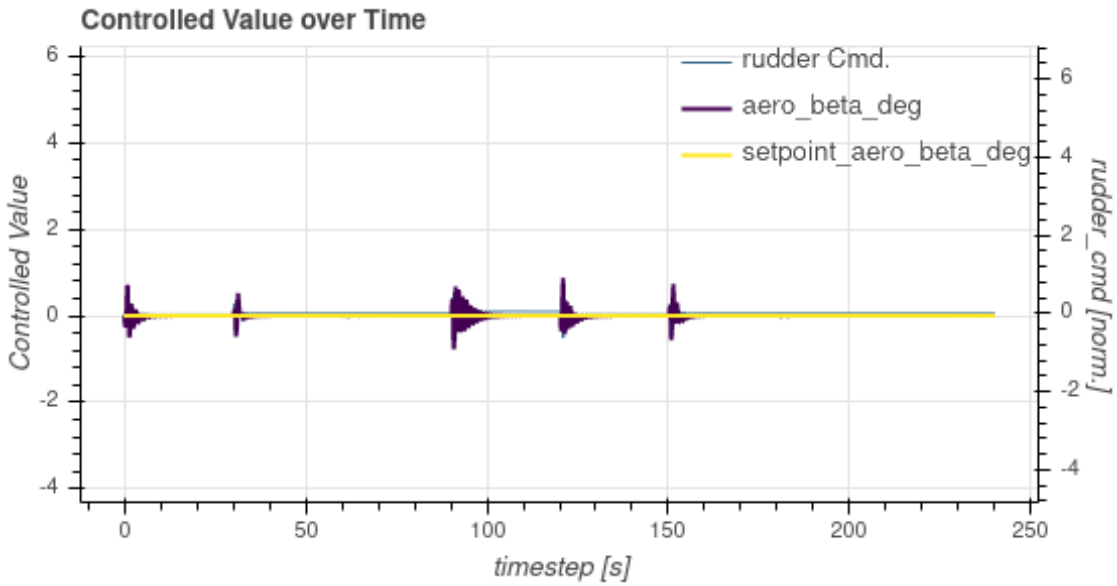

Figure 6.12: Trained agent for sideslip compensation.

Characteristic values of control between events:

event #	0	1	2	3	4	5	6
	RL control	RL	RL	RL	RL	RL	RL
event_time	0	30.2	60.2	90.2	120.2	150.2	180.2
peak_time	0.8	31.4	61.2	90.8	121	151.4	181.6
delay_secs	0.8	1.2	1	0.6	0.8	1.2	1.4
setpoint	0	0	0	0	0	0	0
actual_value	0.695	0.521	0.01	-0.764	0.881	0.734	0.01
setpoint_change	0	0	0	0	0	0	0
abs_overshoot	0.695	0.521	0.01	-0.764	0.881	0.734	0.01
rel_overshoot	NaN	NaN	NaN	NaN	NaN	NaN	NaN
abs_mean	0.037	0.02	0.001	0.072	0.043	0.032	0
MSE	0.013	0.006	0	0.027	0.015	0.011	0
actuation_energy	0.605	0.621	0	2.628	0.855	0.724	0

Settling times after events given in seconds:

event #	0	1	2	3	4	5	6
	RL control	RL	RL	RL	RL	RL	RL
setpoints	0	0	0	0	0	0	0
setpoint_changes	0	0	0	0	0	0	0
error band: 0.5	1.2	1.4	0	2.2	1	1.4	0
error band: 0.1	4.2	1.6	0	8.4	6	4	0
error band: 0.05	5.8	4	0	10.8	7.4	5.8	0
error band: 0.01	8.8	7.2	1.2	15	10.2	7.8	0

Table 6.3: Characteristic values and settling times for the sideslip compensation controller from figure 6.12. No PID control values for comparison are available.

All three sub-tasks needed for a gliding descent are now defined. They all follow the same structure, but are differentiated by their weights, parameters and the presented state. The generic code to define the sub-tasks is given in the listing in appendix D.3. The weights and parameters are summarized in table 6.13.

Task:	glide path angle	banking angle	sideslip compensation
actuating_prop:	`prp.elevator_cmd`	`prp.aileron_cmd`	`prp.rudder_cmd`
setpoints:	`prp.flight_path_deg`	`prp.roll_deg`	`prp.sideslip_deg`
presented_state:	`[prp.q_radps, prp.indicated_airspeed, prp.elevator_cmd]`	`[prp.p_radps, prp.indicated_airspeed, prp.aileron_cmd]`	`[prp.r_radps, prp.indicated_airspeed, prp.rudder_cmd, banking_task.setpoint_value_props[0], banking_task.setpoint_props[0]]`
ordinary error component:	AsymptoticErrorComponent	AsymptoticErrorComponent	AsymptoticErrorComponent
ANGLE_DEG_ERROR_SCALING:	0.1	0.1	0.5
oscillation suppression component:	LinearErrorComponent	LinearErrorComponent	LinearErrorComponent
CMD_TRAVEL_MAX:	0.5	0.5	0.5
integral component:	LinearErrorComponent	LinearErrorComponent	LinearErrorComponent
integral_limit:	0.25	0.25	0.25
ANGLE_INT_DEG_MAX:	`self.integral_limit`	`self.integral_limit`	`self.integral_limit`
weight error:	6	6	4
weight travel error:	4	4	1
weight integral:	10	10	1

Figure 6.13: The weights and parameters for the task definitions for glide path and banking angle as well as sideslip compensation.

6.3 Learning Control for the Gliding Descent

So far, each agent was used and trained separately on its associated task while the aircraft was stabilized on the other axes by conventional PID control.

In a first attempt to apply RL control on all three axes, the three RL agents that were trained during the reward engineering experiments, were just combined together and applied to the gliding descent problem. During reward engineering a single conventional controller was replaced by an RL-based controller acting in the environment it was trained in. Now the controllers for elevator, aileron and rudder were all replaced at once and thus experience a different environment like they were trained in.

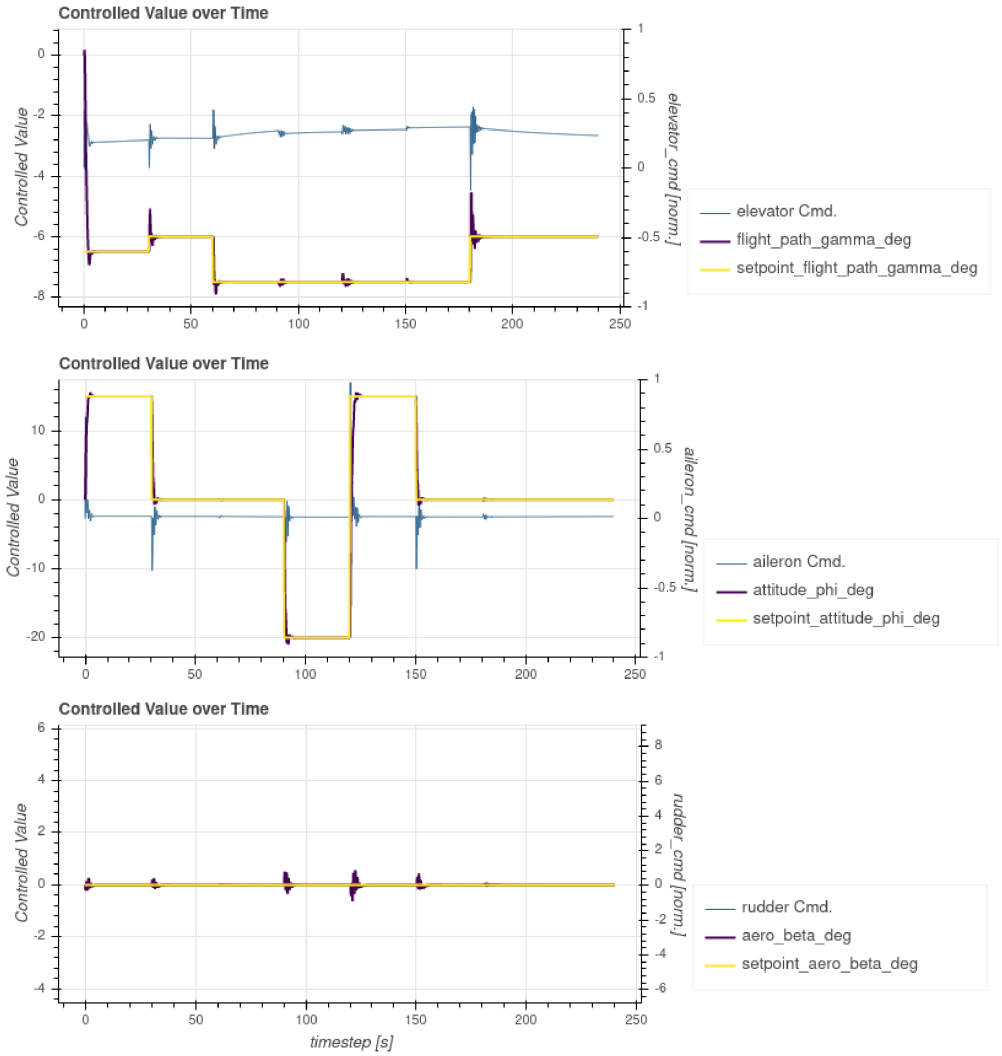

Figure 6.14: Control performance when combining the agents trained isolated from each other in the reward engineering section.

The results of this drop-in-replacement of PID controllers with RL controllers is shown in figure 6.14. The tables comparing the characteristic values from pure PID control to RL control are given in appendix D.4.

This research however is aiming at the development of an RL-based 3-axes flight controller for elevator, aileron and rudder that learns its behavior purely from data gained by own experience in a simulator. The usage of externally developed PID controllers like in the preceding experiment, shall be avoided, as this would defeat the idea of using machine learning without manual parameter tuning for the hard work of finding a good control law.

Therefore, in a last set of experiments, the control of all three axes was jointly trained using different approaches which are all embedded into the `Markov-Pilot`-SW framework.

elevator agent	aileron agent	rudder agent	
isolated training with PID	isolated training with PID	isolated training with PID	combination of separately trained agents settled; Needs manually tuned PID controllers for training;
multi channel agent		constant 0	agent didn't converge; no useful control possible; same result with extended ANN dimensions;
multi channel agent			agent didn't converge; no useful control possible; same result with extended ANN dimensions;
DDPG	DDPG	constant 0	training converged, but elevator and aileron didn't settle to the lowest error band;
DDPG	DDPG	DDPG	training converged, but aileron and rudder didn't settle to the lowest error band;
MADDPG	MADDPG	MADDPG	unstable learning; flittering came back late in training; similar results with different ANN dimensions;
DDPG	MADDPG	MADDPG	aileron unlearned good control as training evolved; no consistently good strategy was learned;
cooperative DDPG	cooperative DDPG	cooperative DDPG	all agents converged; all agents settled within the lowest error band;
cooperative MADDPG	cooperative MADDPG	cooperative MADDPG	aileron unlearned good control as training evolved; flittering came back late in training; no consistently good strategy was learned;

Table 6.4: Overview of the different joint agent training approaches evaluated by experiments.

The tested setups include agents controlling two or three axes simultaneously with aggregated rewards from the associated tasks and single channel agents following the idea of Markov games with multiple agents acting in the same environment. To train multiple agents, the classic DDPG and the extended MADDPG algorithm were used.

While in the MADDPG algorithm, information from the other agents is only available to the critic, in a different set-up, truly cooperating agents were tested. The action

outputs from all agents were added to the presented observation so that the agents always know about the policies of the other agents.

Compared to the experiments on reward engineering, these experiments are even more resource hungry and time consuming. In the work at hand, it was not possible to perform these experiment to a degree of detail like the reward engineering investigations, but more thorough investigations are left for future research. Nevertheless, the performed tests already give a first impression and serve as a starting point.

The different experimental setups are given in table 6.4 together with a brief result summary. If not otherwise stated, all training sessions were performed over 30000 training steps. To assess the training performance, the analysis of characteristic values like described in section 4.3 was interpreted together with a close look at the plots produced over the standardized test-run.

The results stated in table 6.4 shall be detailed further in the next paragraphs.

MULTI-ACTION AGENTS Not very surprisingly, the experiments with a single agent serving multiple sub-tasks at once with an aggregated reward were not successful. As already suspected in section 4.2.5, the different characteristics of the sub-tasks are not amenable for reward aggregation in the agent. Obviously, too much information is lost in the aggregation of the individual task rewards and the agent cannot learn a balanced strategy for its multiple output actions.

One of the best results achieved during the experiments is shown in figure 6.15 with a dual channel DDPG agent and the rudder fixed in neutral position. When adding the rudder actuation to a multi channel agent, the results got even worse and no targeted learning at all could be observed.

Different tests were conducted with different ANN dimensions and more training steps. In no tested case, the agent converged to a usable control.

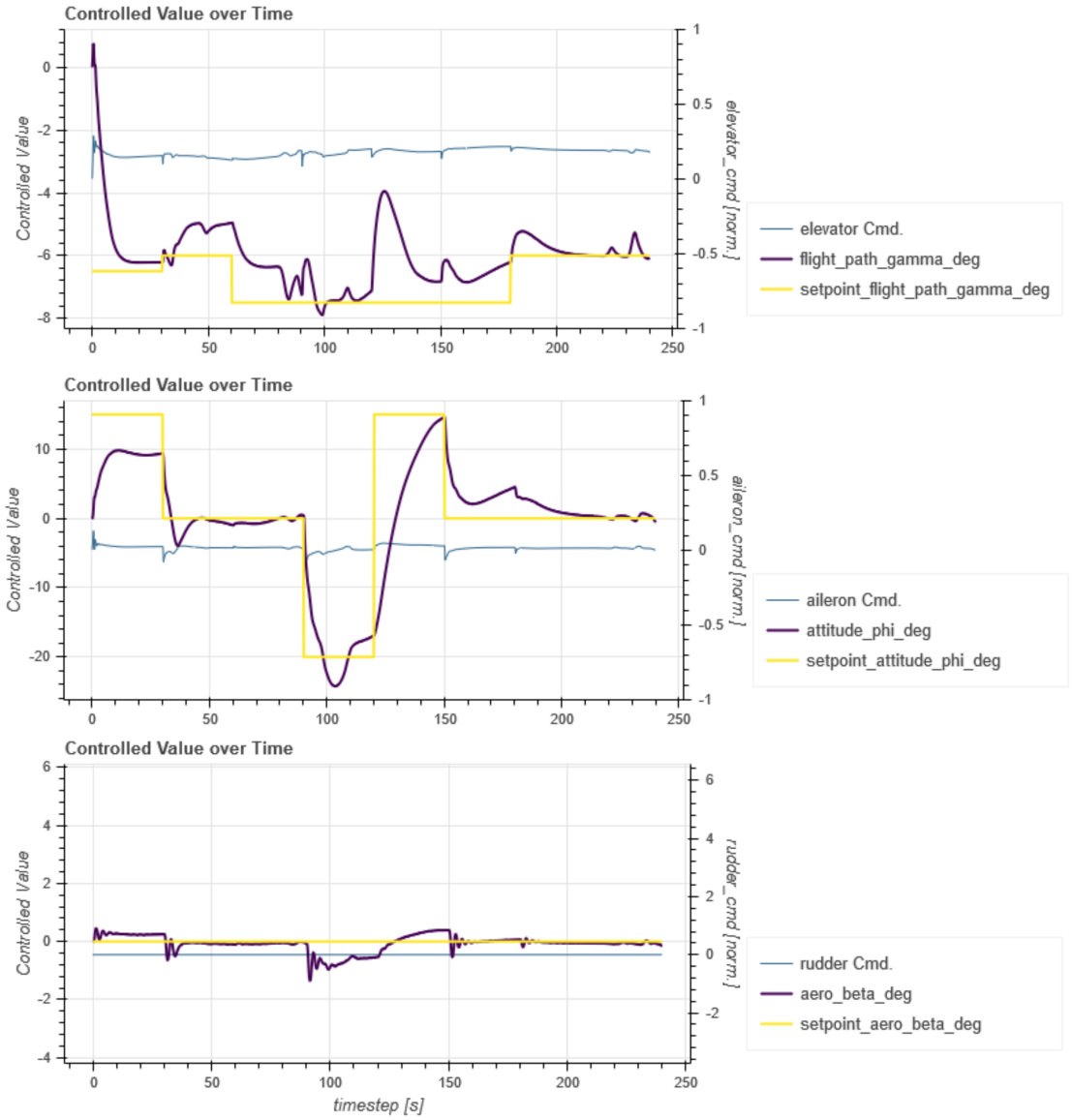

Figure 6.15: Training result of a dual channel DDPG agent with reward aggregation associated to the elevator and aileron task after 50000 training steps.

MULTIPLE AGENTS TRAINED WITH DDPG Interesting results were obtained by training independent agents each associated to a single flight task and trained with the traditional DDPG algorithm. Despite the fact, that the surrounding environment gets non-stationary due to the changing policies of the other agents (cf. [Lowe et al., 2020]), the overall training result was impressively good.

In the case of two DDPG agents and the rudder fixed to neutral position, the RL control of elevator and aileron achieved the result shown in figure 6.16. While looking good in the graph, the analysis of the settling times revealed, that the accuracy of the aileron in steady state did not reach the smallest error band of $\pm 0.01°$.

The same observation was made for the case of three independent DDPG agents with rudder control included. Also in these tests, the steady state error did not settle within the lowest error band.

However, depending on the actual accuracy requirements, this training approach could be an option. Training was stable and reliable and yielded the named results already after 30000 training steps.

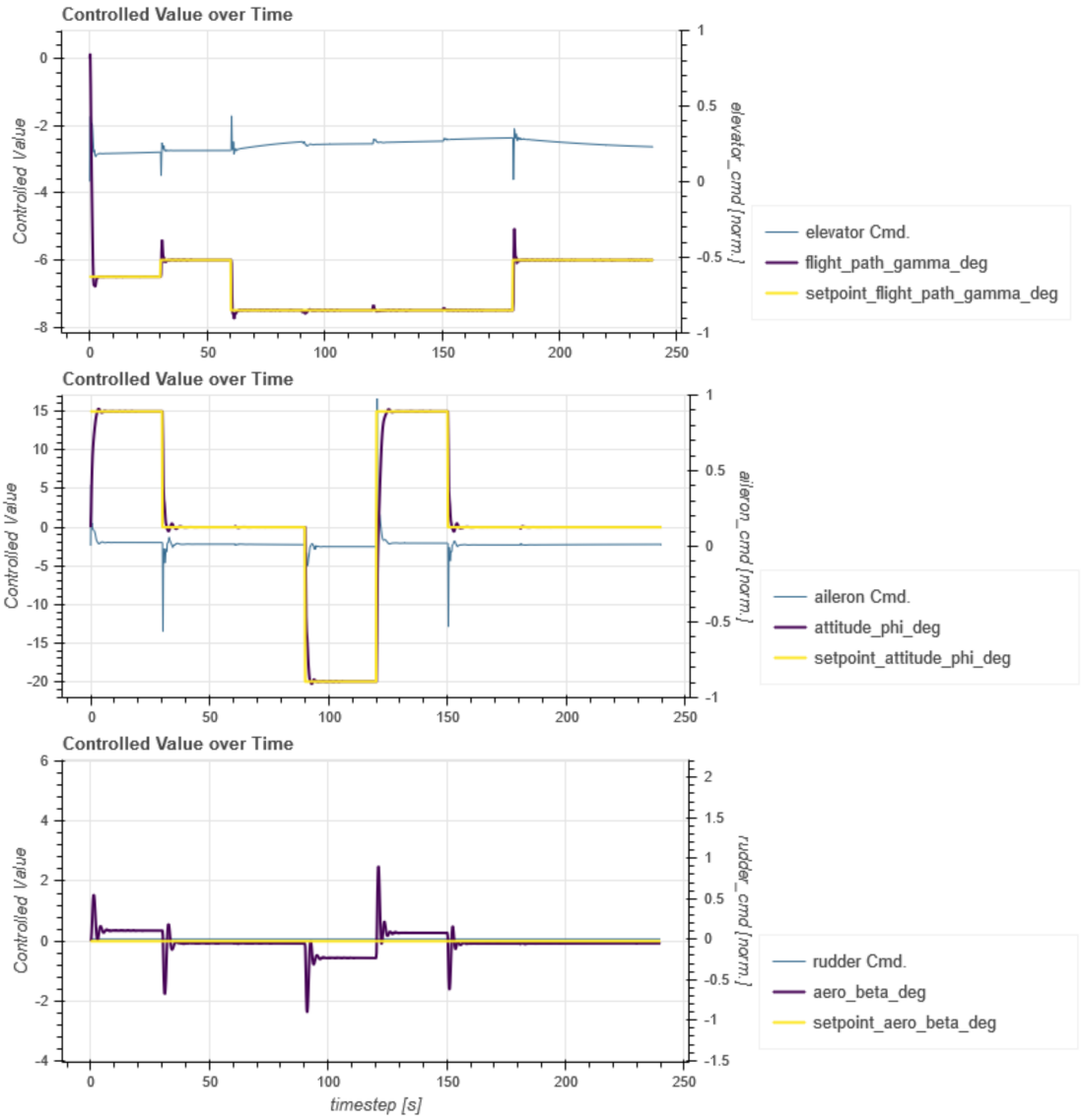

Figure 6.16: Training result of two independent DDPG agents associated to the elevator and aileron task respectively after 30000 training steps. Rudder was held at zero.

MULTIPLE AGENTS TRAINED WITH MADDPG The most surprising and at the same time disappointing results were observed when training three RL agents using the MADDPG approach proposed in [Lowe et al., 2020]. In this setting, the agents' critics are supplied with the observations and actions of all other agents in the environment. This method makes the environment stationary even in the presence of changing policies of the other agents.

A typical test-run after training with MADDPG can be seen in figure 6.17.

In the tests conducted, the inclusion of the extended observations to the agents' critics had adverse effects: It was observed, that the learning became quite unstable. While already looking extremely good in the mid of the training, the agents kept forgetting formerly learned good strategies and degraded considerably again. No recognizable pattern could be identified.

Additionally it was observed, that flittering of actuators came back even late in the training. When investigating the evolution of the MADDPG-trained controllers more closely it looked like the flittering originated from the rudder controller and subsequently infected the other agents especially the elevator controller.

Based on this observation, more tests were performed: The elevator was trained isolated using the DDPG algorithm while MADDPG was only used for aileron and rudder. The network dimensions of the ANNs were increased as the critics in MADDPG have significantly more input parameters. Last but not least, the amount of training steps was increased.

Unfortunately, none of these measures lead to a consistently better training with reliable results. In all settings so far, the performance of the agents over training time was fluctuating and from time to time the control was overlaid with high frequency oscillations on the action output.

Regarding the training with the MADDPG algorithm there is still a lot of potential for further research as this observation contradicts the claims and observations published in [Lowe et al., 2020].

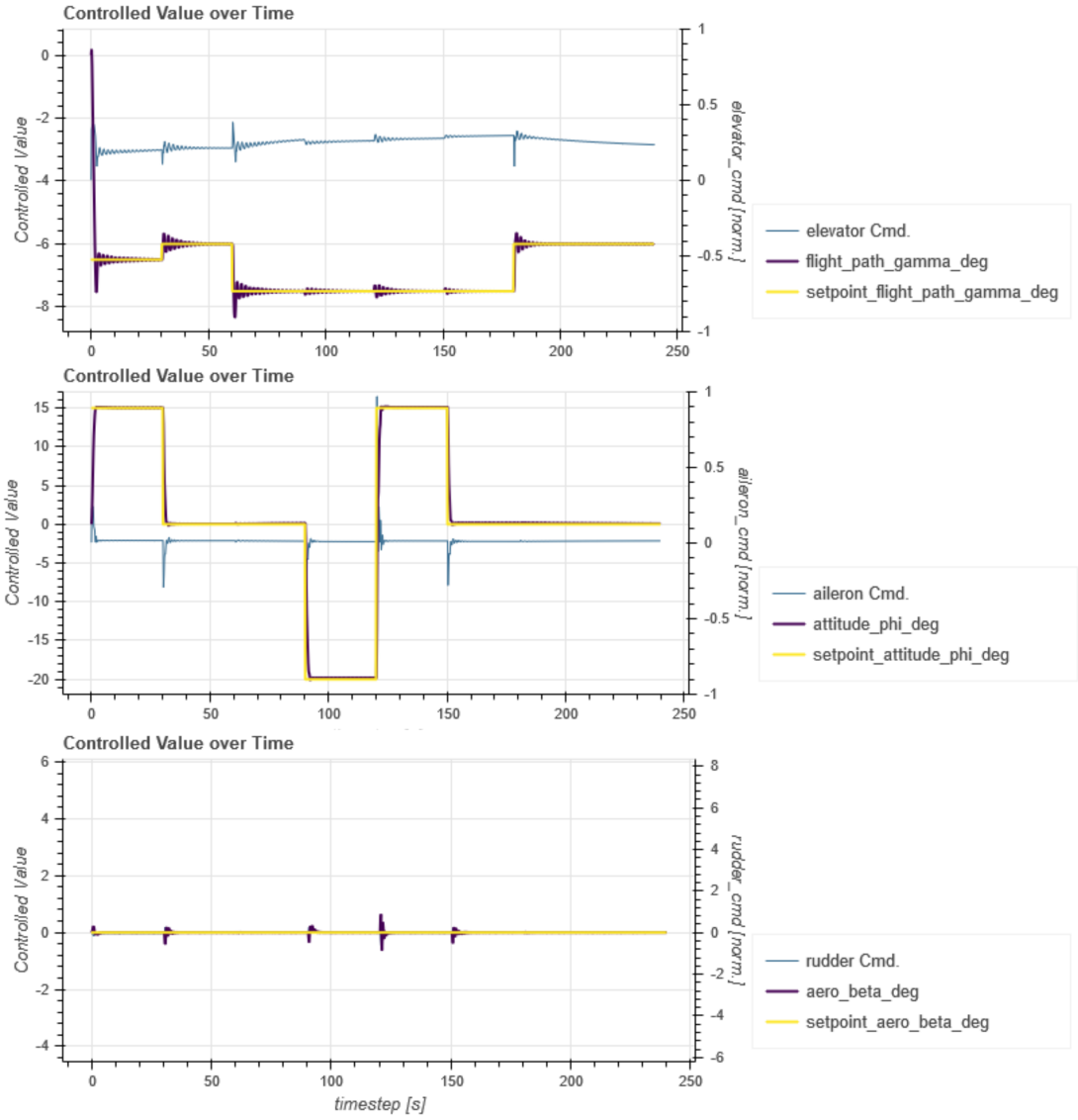

Figure 6.17: Training result of three MADDPG agents after 30000 training steps. The training with the MADDPG algorithm was quite unstable and a tendency for high frequency oscillations was regularly propagated from the rudder control to the other agents.

MULTIPLE COOPERATING AGENTS The last training setting that was investigated was one of *cooperating agents*. Cooperating agents in our setting share their actions and thus their policies with all other agents in the same environment. I. e. the output actions of all agents are added to the observation inputs of all agents. By this means, the environment gets stationary again not only for the critic, but also for the actor of a DDPG agent.

Eventually this strategy was successful when used with the DDPG agents in the gliding descent environment. It yielded controllers that managed to settle within the lowest error range in steady state operation while performing fast control with acceptable overshoot.

The result of training three cooperating DDPG agents over 30000 steps is shown in figure 6.18. As a benchmark for the direct comparison, the control performance of conventional PID control on elevator and aileron with the rudder held in neutral position is repeated in figure 6.19. The characteristic values and settling times of this test-run can be found in appendix D.5. The tables include a comparison to conventional PID control.

When comparing the controller performance of the cooperating RL agents with the conventional PID control, it becomes obvious, that there is definitively an advantage for the RL-based approach. The cooperating RL agents outperform PID control in every concern besides the needed actuation energy. The actuation energy is higher for RL control, which is due to the fact, that the control acts faster and more active counteraction is needed to avoid overshoot.

Two of the main benefits of the trained agents over the manually tuned PID controllers are:

- As RL controllers like described in this work are based on ANNs which can inherently cope with non-linear functions, the quality of control can be stunning when adequately trained.
- When properly set up, the agents learn on their own without further intervention. They can derive control laws from data just from exploiting their experience. This makes adaptations to the controllers easy and fast after the initial invest of setting up the training environment.

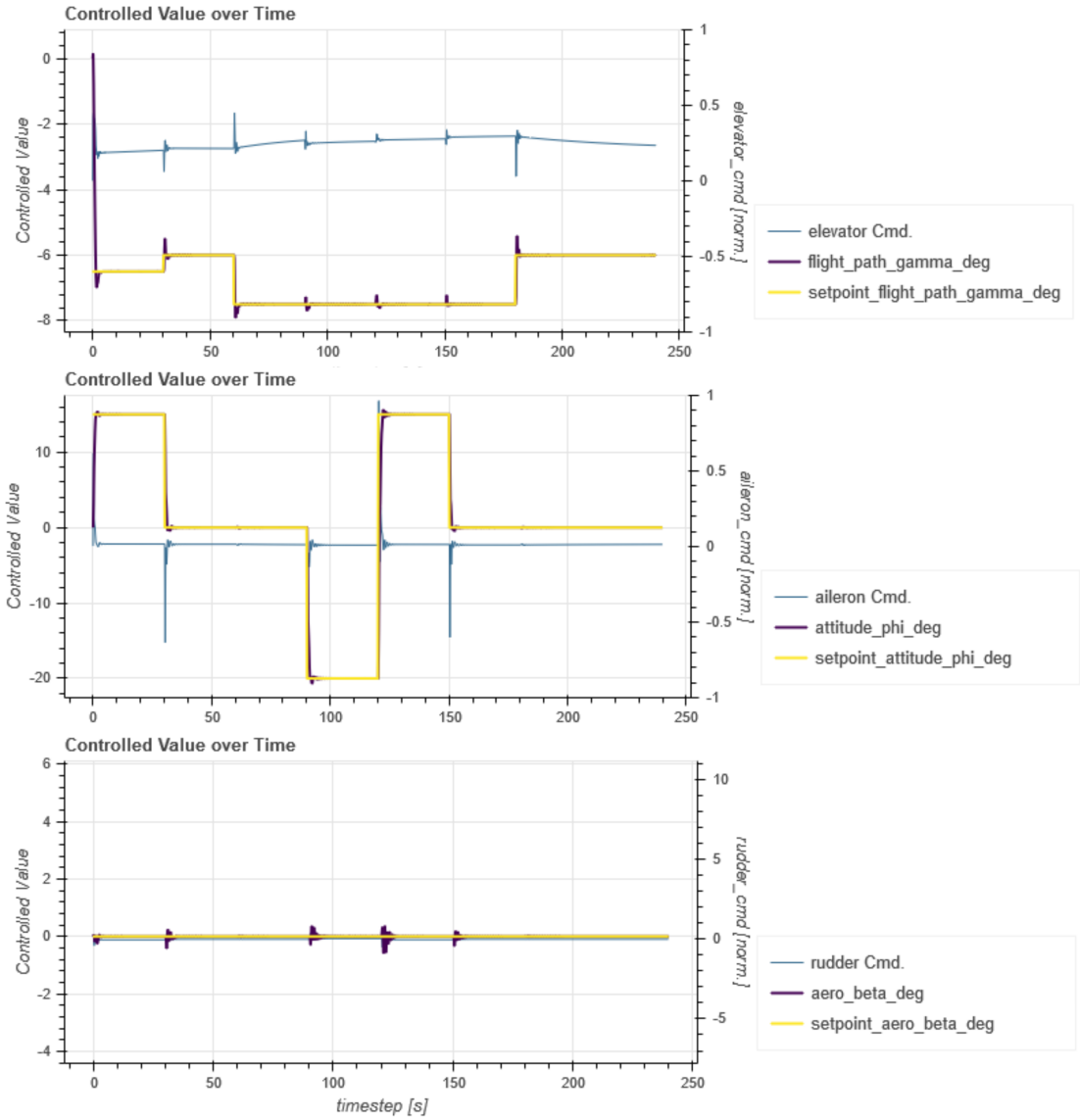

Figure 6.18: Training result of three cooperating DDPG agents after 30000 training steps. The training with cooperating DDPG agents yielded the best results observed during the research at hand.

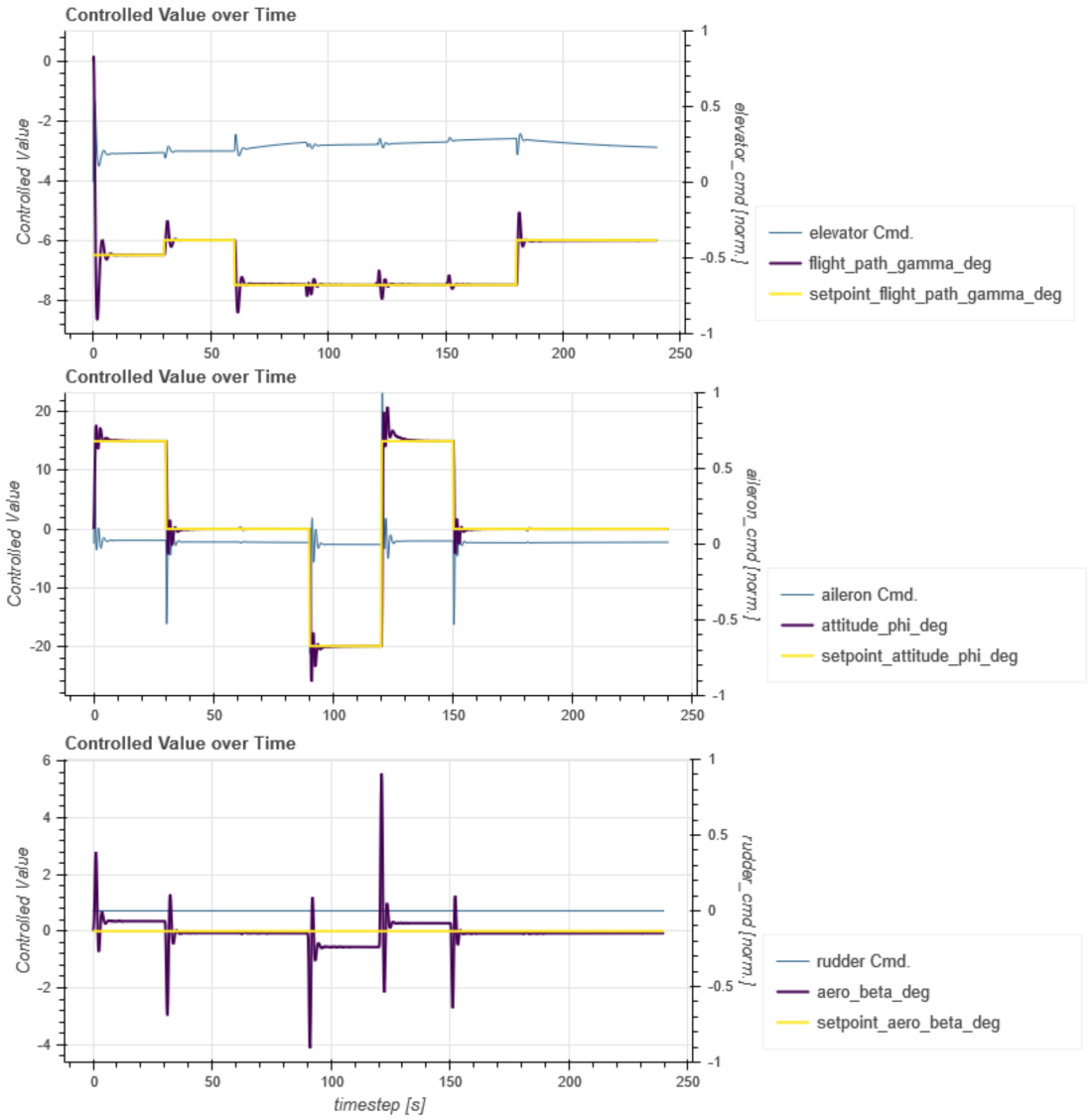

Figure 6.19: PID-based elevator and aileron control with the rudder held in zero position. (Figure given as a comparison to figure 6.18.)

In a last experiment, the 3-axes controller with cooperating RL agents was confronted with a setting different from the standardized test: To visually get an impression on the controller's performance, the successfully trained cooperating agents were applied to an environment with the random set-point variation enabled. In figure 6.20 it becomes obvious, that the controllers can do a lot more than just handle simple step functions in their set-points.

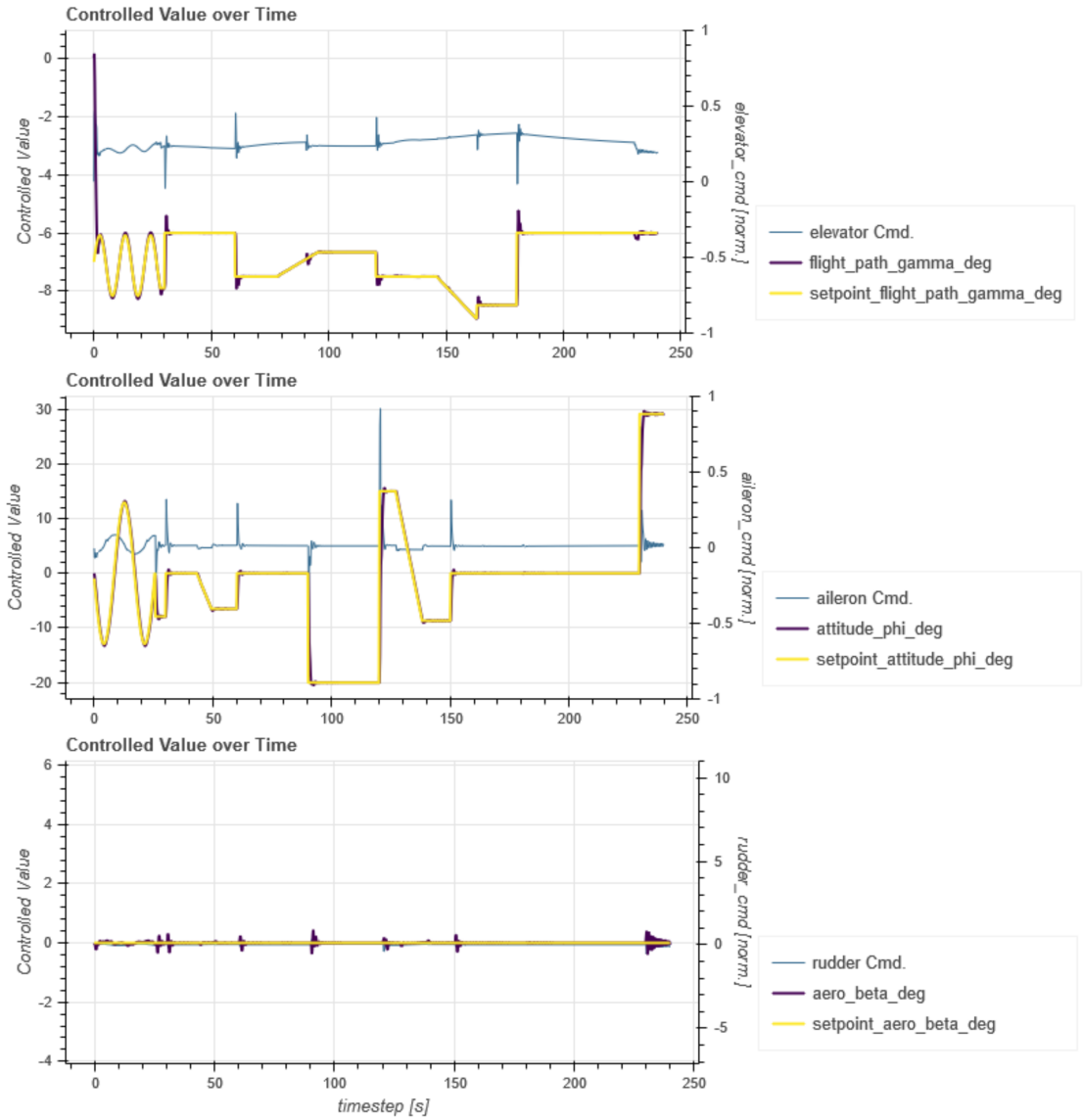

Figure 6.20: Performance of the jointly trained cooperating agents when applied to random set-point variations.

7 RESULTS SUMMARY AND OUTLOOK ON FUTURE RESEARCH

This concluding section shall summarize the main findings and outcomes of the research at hand and give an outlook to the still open points which deserve to be addressed in continuing work and future research.

The main outcome of this research is the `Markov-Pilot` SW-framework for the application of RL methods to aircraft flight control. The design of this software allows a lot of different investigations regarding (multi-)task definitions with flexible reward structures as well as the definition and training of (multiple) RL agents learning to solve these tasks in a common simulated aircraft environment.

The developed SW was successfully used to conduct thorough analysis on a reward structure suitable for flight control tasks. As a result of these investigations, a generic reward structure was found, that can be parameterized for all types of subtasks needed for the gliding descent scenario of an aircraft. It is expected, that the same generic reward structure is also eligible to other flight control tasks like e. g. speed control or flight planning tasks like e. g. heading hold or turning.

As a by-product of the reward engineering experiments, single channel agents were trained, which outperform the manually tuned PID agents for elevator and aileron control in most concerns. These agents, that were trained in an environment stabilized by conventional control during training, can be used as a drop-in replacement for the PID controllers.

Even though these drop-in replacements already outperform the conventional control when hooked into the environment, it's not the ultimate goal to manually tune PID controllers to use them for the training of their own replacements. Consequently, in the last part of experimentation, the so far developed building blocks were put together, to train not only individual controllers for single axes, but to jointly train an entire 3-axes controller from scratch. These experiments were successful in training cooperative agents in a DDPG set-up.

The performance of this 3-axes controller is impressive regarding fast set-point tracking and high accuracy: Despite the fast reaction after steps in the set-point, the overshoot could be kept low. At the same time, the settling in steady state shows highest accuracy and the susceptibility to cross coupling from events in the other axes is minimized.

This superior performance is probably due to the ANNs' non-linear nature that allows non-linear control laws. They cannot only cope with small deviations in a linearized region around a working point, but can also handle big excursions where the linearization assumption used implicitly for PID control doesn't hold. This way, the ANN-based RL control can adapt its behavior depending on the disturbance to be compensated and the overall flight attitude.

The jointly trained 3-axes controller not only proved very good performance in the standardized test-run for evaluation, but also when confronted with extreme random set-point variations.

Surprisingly, the training with the MADDPG algorithm showed unexpected weaknesses. It was not possible to jointly train a controller that consistently settled to the lowest error band on all three axes. To further optimize this behavior and to further understand the mechanics of joint training, excessive hyper-parameter studies should be performed as a follow-up to this research.

The results seen so far give rise to quite some research topics for the continuation of this work:

The joint training of agents was only touched upon briefly. In subsequent steps, more detailed investigations must follow that should include at least:

- Hyper-Parameter Tuning: The algorithms contain some parameters that were so far set to default values without proper justification. These parameters like network dimensions, learning rates, batch sizes for back-propagation, ..., shall be evaluated and optimized.
- Variance and Fluctuation during Training: It was observed, that the agents keep forgetting good strategies they once learned later on in the training. Sometimes they recover and find even better strategies, but sometimes they stay at sub-optimal results. This phenomenon needs further inspection and strategies shall be developed on how to leverage better intermediate results. Is it possible to combine agents on different learning levels without further problems?
- Inclusion of Combined and Hierarchical Tasks: So far the simplified gliding descent was used as the example problem. This should be enhanced to more advanced problems needing more complex cooperation of the agents. Also hierarchical tasks with one controller determining the set-point for another shall be investigated. Such hierarchical tasks may be useful for navigation or automated landing.
- Different Learning Algorithms: So far only DDPG-like algorithms were investigated as DDPG is one of the first an best understood algorithms for control in continuous spaces. Meanwhile there are however more recent algorithms for continuous control available. It's definitely worth to try out different training methods and find out about the specialties of different algorithms.

Such more advanced investigations however are very resource hungry. These further investigations will need broad scans over parameter spaces and a whole lot of comparative experiments need to be performed. While a single training run takes only about half an hour on a decent laptop computer, such broadened experiments will need more powerful hardware. Fortunately, high-performance clusters and cloud comput-

ing resources are available nowadays which allow running multiple experiments in parallel.

The `Markov-Pilot` software is a sound basis for such further experiments as it can run entirely headless and most parameters can be passed by command line arguments which is amenable to scripting. It contains automated performance evaluation regarding the characteristic values and settling times and the generated HTML-pages containing test-run plots can be collected centrally for evaluation.

Results from such broadened experiments will definitely deepen the understanding of the concrete behavior, the interoperability and the evolution of RL agents acting in an aircraft environment. Only with a very deep and thorough understanding more advanced topics like safety and guaranteed control quality can be tackled.

Hopefully, this work serves as a first baby-step towards not only interesting, but truly inspiring new solutions in flight control.

There must be better means of Learning to Fly than what Douglas Adams stated in *Life, the Universe and Everything* [Adams, 1995]:

> The knack of flying is learning how to throw yourself at the ground and miss.

REFERENCES

Douglas Adams. *Life, the Universe and Everything*. Del Rey, New York, reissue edition, September 1995. ISBN 978-0-345-39182-7.

David Allerton. *Principles of Flight Simulation*. AIAA Education Series. American Institute of Aeronautics and Astronautics ; Wiley, Reston, VA : Chichester, U.K, 2009. ISBN 978-1-60086-703-3.

Jon Berndt and Agostino De Marco. Progress On and Usage of the Open Source Flight Dynamics Model Software Library, JSBSim. In *AIAA Modeling and Simulation Technologies Conference*, Chicago, Illinois, August 2009. American Institute of Aeronautics and Astronautics. ISBN 978-1-62410-161-8. doi: 10.2514/6.2009-5699. URL http://arc.aiaa.org/doi/10.2514/6.2009-5699. [Accessed on: 2020-05-03].

Rudolf Brockhaus, Wolfgang Alles, and Robert Luckner. *Flugregelung*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-01442-0 978-3-642-01443-7. doi: 10.1007/978-3-642-01443-7. URL http://link.springer.com/10.1007/978-3-642-01443-7. [Accessed on: 2020-05-09].

G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, 2(4):303–314, December 1989. ISSN 0932-4194, 1435-568X. doi: 10.1007/BF02551274. URL http://link.springer.com/10.1007/BF02551274. [Accessed on: 2020-05-04].

Daniel Dewey. Reinforcement Learning and the Reward Engineering Principle. In *2014 AAAI Spring Symposium Series*, page 4, 2014. URL https://www.aaai.org/ocs/index.php/SSS/SSS14/paper/viewFile/7704/7740. [Accessed on: 2020-05-10].

Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. Benchmarking Deep Reinforcement Learning for Continuous Control. *arXiv:1604.06778 [cs]*, May 2016. URL http://arxiv.org/abs/1604.06778. [Accessed on: 2020-05-05 22:02:59].

Felix Eckstein. *Implementierung eines Notlandeassistenten auf Basis von Dubins-Kurven unter Berücksichtigung des Windes*. Bachelor thesis, FernUniversität in Hagen, Hagen, 2018. URL https://github.com/opt12/DubinsPilot/tree/master/thesis. [Accessed on: 2020-05-09].

Hado van Hasselt. Reinforcement Learning in Continuous State and Action Spaces. In Marco Wiering and Martijn van Otterlo, editors, *Reinforcement Learning: State-of-the-Art*, Adaptation, Learning, and Optimization, pages 207–251. Springer, Berlin, Heidelberg, 2012. ISBN 978-3-642-27645-3. doi: 10.1007/978-3-642-27645-3_7. URL https://doi.org/10.1007/978-3-642-27645-3_7. [Accessed on: 2020-06-01].

JSBSim. JSBSim Open Source Flight Dynamics Model. URL http://www.jsbsim.org. [Accessed on: 2020-05-17].

JSBSim Reference. JSBSimReferenceManual.pdf. URL http://jsbsim.sourceforge.net/JSBSimReferenceManual.pdf. [Accessed on: 2019-10-21].

Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *arXiv:1412.6980 [cs]*, January 2017. URL http://arxiv.org/abs/1412.6980. [Accessed on: 2020-05-03].

Marius Klein, Andreas Klos, Jörg Lenhardt, and Wolfram Schiffmann. Moving target approach for wind-aware flight path generation. *International Journal of Networking and Computing*, 8(2):351–366, July 2018. ISSN 2185-2847. URL http://www.ijnc.org/index.php/ijnc/article/view/189. [Accessed on: 2020-05-09].

Wolfgang Langewiesche and Leighton Holden Collins. *Stick and Rudder: An Explanation of the Art of Flying*. McGraw-Hill, New York, 1972. ISBN 978-0-07-036240-6.

Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv:1509.02971 [cs, stat]*, September 2015. URL http://arxiv.org/abs/1509.02971. [Accessed on: 2019-01-09].

Michael L. Littman. Markov games as a framework for multi-agent reinforcement learning. In *Machine Learning Proceedings 1994*, pages 157–163. Elsevier, 1994. ISBN 978-1-55860-335-6. doi: 10.1016/B978-1-55860-335-6.50027-1. URL https://linkinghub.elsevier.com/retrieve/pii/B9781558603356500271. [Accessed on: 2020-04-03].

Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel, and Igor Mordatch. Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments. *arXiv:1706.02275 [cs]*, March 2020. URL http://arxiv.org/abs/1706.02275. [Accessed on: 2020-03-24].

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015. ISSN 0028-0836, 1476-4687. doi: 10.1038/nature14236. URL http://www.nature.com/articles/nature14236. [Accessed on: 2020-02-19].

Andrew Ng and Stuart Russell. Algorithms for Inverse Reinforcement Learning. *ICML '00 Proceedings of the Seventeenth International Conference on Machine Learning*, May 2000. URL https://www.researchgate.net/publication/2622278_Algorithms_for_Inverse_Reinforcement_Learning. [Accessed on: 2020-05-11].

OpenAI. Gym: A toolkit for developing and comparing reinforcement learning algorithms. URL https://gym.openai.com. [Accessed on: 2019-06-17].

Robert Paz. The Design of the PID Controller. January 2001. URL https://www.researchgate.net/publication/237528809_The_Design_of_the_PID_Controller. [Accessed on: 2020-05-25].

Martin L. Puterman. *Markov Decision Processes : Discrete Stochastic Dynamic Programming*. Wiley Series in Probability and Statistics. Wiley-Interscience, Hoboken, NJ, 1994. ISBN 978-0-471-72782-8.

Kevin Regan and Craig Boutilier. Regret-based Reward Elicitation for Markov Decision Processes. *arXiv:1205.2619 [cs]*, May 2012. URL http://arxiv.org/abs/1205.2619. [Accessed on: 2020-05-11].

Gordon Rennie. *Autonomous Control of Simulated Fixed Wing Aircraft Using Deep Reinforcement Learning*. Thesis, University of Bath, Bath, September 2018. URL https://drive.google.com/file/d/1jPLG-OYcPiffh4ZAWW1N1__4l68jh-G_/view. [Accessed on: 2019-10-11].

Gordon Rennie. Gor-Ren/gym-jsbsim, July 2019. URL https://github.com/Gor-Ren/gym-jsbsim. [Accessed on: 2019-10-11].

Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv:1609.04747 [cs]*, June 2017. URL http://arxiv.org/abs/1609.04747. [Accessed on: 2020-05-03].

David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ICML'14, pages I–387–I–395, Beijing, China, June 2014. JMLR.org.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Adaptive Computation and Machine Learning Series. The MIT Press, Cambridge, MA, second edition edition, 2018. ISBN 978-0-262-03924-6. URL http://incompleteideas.net/book/the-book-2nd.html. [Accessed on: 2019-01-08].

Richard S. Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Proceedings of the 12th International Conference on Neural Information Processing Systems*, NIPS'99, pages 1057–1063, Denver, CO, November 1999. MIT Press.

Teng-Tiow Tay, Ivan Mareels, and John B. Moore. *High Performance Control*. Systems & Control: Foundations & Applications. Birkhäuser Basel, 1998. ISBN 978-0-8176-4004-0. doi: 10.1007/978-1-4612-1786-2. URL https://www.springer.com/gp/book/9780817640040. [Accessed on: 2020-05-15].

E. G. Tulapurkara. Flight dynamics I - Airplane performance, Introduction - 3, Lecture 3. 2012. URL https://nptel.ac.in/courses/101/106/101106041/. [Accessed on: 2018-08-16].

V. Vaishnavi, W. Kuechler, and S. Petter. Design Science Research in Information Systems, 2004/19. URL http://www.desrist.org/design-research-in-information-systems/. [Accessed on: 2020-05-02].

A DDPG ALGORITHM

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

Figure A.1: The DDPG algorithm in pseudo-code like presented in the original paper [Lillicrap et al., 2015]

B MADDPG ALGORITHM

Algorithm 1: Multi-Agent Deep Deterministic Policy Gradient for N agents

for episode $= 1$ to M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial state $\mathbf{x}$
  for $t = 1$ to max-episode-length **do**
    for each agent i, select action $a_i = \boldsymbol{\mu}_{\theta_i}(o_i) + \mathcal{N}_t$ w.r.t. the current policy and exploration
    Execute actions $a = (a_1, \ldots, a_N)$ and observe reward r and new state $\mathbf{x}'$
    Store $(\mathbf{x}, a, r, \mathbf{x}')$ in replay buffer $\mathcal{D}$
    $\mathbf{x} \leftarrow \mathbf{x}'$
    for agent $i = 1$ to N **do**
      Sample a random minibatch of S samples $(\mathbf{x}^j, a^j, r^j, \mathbf{x}'^j)$ from $\mathcal{D}$
      Set $y^j = r_i^j + \gamma\, Q_i^{\boldsymbol{\mu}'}(\mathbf{x}'^j, a'_1, \ldots, a'_N)\big|_{a'_k = \boldsymbol{\mu}'_k(o_k^j)}$
      Update critic by minimizing the loss $\mathcal{L}(\theta_i) = \frac{1}{S}\sum_j \left(y^j - Q_i^{\boldsymbol{\mu}}(\mathbf{x}^j, a_1^j, \ldots, a_N^j)\right)^2$
      Update actor using the sampled policy gradient:

$$\nabla_{\theta_i} J \approx \frac{1}{S}\sum_j \nabla_{\theta_i} \boldsymbol{\mu}_i(o_i^j) \nabla_{a_i} Q_i^{\boldsymbol{\mu}}(\mathbf{x}^j, a_1^j, \ldots, a_i, \ldots, a_N^j)\big|_{a_i = \boldsymbol{\mu}_i(o_i^j)}$$

    end for
    Update target network parameters for each agent i:

$$\theta'_i \leftarrow \tau\theta_i + (1-\tau)\theta'_i$$

  end for
end for

Figure B.1: The MADDPG algorithm in pseudo-code like presented in the original paper [Lowe et al., 2020]

C DEVELOPED SOFTWARE MARKOV-PILOT

The entire software `Markov-Pilot` developed within the course of this thesis is published with an MIT license on GitHub.

`https://github.com/opt12/Markov-Pilot`

> The version being part of this thesis is:
> `https://github.com/opt12/Markov-Pilot/tree/bd4ecb603bbab6bb9272b9c00f2993b79a708d47`
> This is also the version contained on the CD attached to the print edition.

Please feel free to use and enhance this software at your needs.

D EXPERIMENTS' PARAMETERS

In the upcoming section, detailed parameters for some of the experiments described in section 6 are given.

D.1 Double PID Agent

The code snippets show how to define the environment and the agents for a setting with dual PID control for glide path angle and banking angle control. A PID agent with all-zero parameters keeps the rudder in neutral position. As the PID agents do not learn, the reward for the tasks is set to constant zero.

```
def setup_env(arglist) -> NoFGJsbSimEnv_multi_agent:
    agent_interaction_freq = arglist.interaction_frequency
    episode_time_s=arglist.max_episode_len_sec

    glide_path_task_for_PID = SingleChannel_FlightTask('glide_path_angle', prp.elevator_cmd, {prp.flight_path_deg:
          target_path_angle_gamma_deg},
                              make_base_reward_components=_make_base_reward_components,
                              integral_limit = 100)

    banking_task_for_PID = SingleChannel_FlightTask('banking_angle', prp.aileron_cmd, {prp.roll_deg:
          initial_roll_angle_phi_deg},
                              make_base_reward_components=_make_base_reward_components,
                              integral_limit = 100)

    sideslip_task_for_PID = SingleChannel_FlightAgentTask('sideslip', prp.rudder_cmd, {prp.sideslip_deg: 0},
                              make_base_reward_components= _make_base_reward_components,
                              integral_limit = 100)

    task_list = [glide_path_task_for_PID, banking_task_for_PID, sideslip_task_for_PID]
    env = NoFGJsbSimEnv_multi(task_list, agent_interaction_freq = agent_interaction_freq, episode_time_s = episode_time_s)
    return env
```

Code Listing D.1: The environment setup for static PID control with constant zero-reward.

```
def setup_container(task_list, arglist):
    agent_classes_dict = {
        'PID': PID_AgentTrainer,
        'MADDPG': MADDPG_AgentTrainer,
        'DDPG': DDPG_AgentTrainer,
    }
    #for PID controllers we need an elaborated parameter set for each type
    pid_params = {'aileron':  PidParameters(3.5e-2,    1e-2,   0.0),
                  'elevator': PidParameters( -5e-2, -6.5e-2, -1e-3),
                  'rudder':   PidParameters(   0.0,     0.0,   0.0),  #no tuning parameters for rudder available ---> const-0
                  }
    params_aileron_pid_agent = {
        'pid_params': pid_params['aileron'],
        'writer': None,
    }
    params_elevator_pid_agent = {
        'pid_params': pid_params['elevator'],
        'writer': None,
    }
    params_rudder_pid_agent = {
    #with the all-0 PID parameters, the output will also be constant 0
        'pid_params': pid_params['rudder'],
        'writer': None,
    }

    agent_spec_aileron_PID  = AgentSpec('aileron',  'PID', ['banking_angle'],    params_aileron_pid_agent)
    agent_spec_elevator_PID = AgentSpec('elevator', 'PID', ['glide_path_angle'], params_elevator_pid_agent)
    agent_spec_rudder_PID   = AgentSpec('rudder',   'PID', ['sideslip'],         params_rudder_pid_agent)

    agent_spec = [agent_spec_elevator_PID, agent_spec_aileron_PID, agent_spec_rudder_PID]
    agent_container = AgentContainer.init_from_specs(task_list_n, agent_spec, agent_classes_dict, **vars(arglist))
    return agent_container
```

Code Listing D.2: The agent container setup with dual PID agents for elevator and aileron. Rudder is held at zero.

Characteristic values of control between events:

event #	0	1	2	3	4	5	6
	RL control	RL	RL	RL	RL	RL	RL
event_time	0	30.2	60.2	90.2	120.2	150.2	180.2
peak_time	1.8	31.6	61.6	91	123	151.4	181.4
delay_secs	1.8	1.4	1.4	0.8	2.8	1.2	1.2
setpoint	-6.5	-6	-7.5	-7.5	-7.5	-7.5	-6
actual_value	-8.669	-5.336	-8.443	-7.894	-7.98	-7.18	-5.048
setpoint_change	-6.5	0.5	-1.5	0	0	0	1.5
abs_overshoot	-2.169	0.664	-0.943	-0.394	-0.48	0.32	0.952
rel_overshoot	0.334	1.327	0.629	NaN	NaN	NaN	0.635
abs_mean	0.288	0.039	0.094	0.028	0.04	0.02	0.043
MSE	1.096	0.016	0.064	0.006	0.011	0.003	0.028
actuation_energy	0.207	0.002	0.013	0.001	0.002	0	0.014

Settling times after events given in seconds:

event #	0	1	2	3	4	5	6
	RL control	RL	RL	RL	RL	RL	RL
setpoints	-6.5	-6	-7.5	-7.5	-7.5	-7.5	-6
setpoint_changes	-6.5	0.5	-1.5	0	0	0	1.5
error band: 0.5	4	1.8	2	0	0	0	1.8
error band: 0.1	6.2	3.8	4	4.4	4.6	3	3.4
error band: 0.05	6.4	4	4.2	5.4	5.6	3.4	3.6
error band: 0.01	8.6	5.6	didn't settle	7.4	10.6	6.6	46.8

Table D.1: Characteristic values and settling times for the glide path angle controller from figure 6.1 with dual PID agents.

Characteristic values of control between events:

event #	0	1	2	3	4	5	6
	RL control	RL	RL	RL	RL	RL	RL
event_time	0	30.2	60.2	90.2	120.2	150.2	180.2
peak_time	1	31	61.2	91	122.6	151	181
delay_secs	1	0.8	1	0.8	2.4	0.8	0.8
setpoint	15	0	0	-20	15	0	0
actual_value	17.802	-4.364	0.147	-26.11	20.908	-4.278	-0.134
setpoint_change	15	-15	0	-20	35	-15	0
abs_overshoot	2.802	-4.364	0.147	-6.11	5.908	-4.278	-0.134
rel_overshoot	0.187	0.291	NaN	0.305	0.169	0.285	NaN
abs_mean	0.417	0.415	0.015	0.516	1.031	0.381	0.007
MSE	3.015	2.859	0.001	4.852	16.673	2.666	0
actuation_energy	0.413	0.466	0	0.872	1.399	0.489	0

Settling times after events given in seconds:

event #	0	1	2	3	4	5	6
	RL control	RL	RL	RL	RL	RL	RL
setpoints	15	0	0	-20	15	0	0
setpoint_changes	15	-15	0	-20	35	-15	0
error band: 0.5	5.6	3.4	0	4.6	8.2	3	0
error band: 0.1	9.4	8.2	1.4	8	13.4	7.4	1.2
error band: 0.05	11.8	10.2	2.6	10.4	15.8	9.4	2.2
error band: 0.01	17	15.2	21.4	15.2	21.8	14	4.8

Table D.2: Characteristic values and settling times for the banking angle controller from figure 6.1 with dual PID agents.

D.2 Agents used during Reward Engineering

```
def setup_container(task_list, arglist):

    agent_classes_dict = {
        'PID': PID_AgentTrainer,
        'MADDPG': MADDPG_AgentTrainer,
        'DDPG': DDPG_AgentTrainer,
    }

    #for PID controllers we need an elaborated parameter set for each type
    pid_params = {'aileron':  PidParameters(3.5e-2,    1e-2,   0.0),
                  'elevator': PidParameters( -5e-2, -6.5e-2, -1e-3),
                  }

    params_aileron_pid_agent = {
        'pid_params': pid_params['aileron'],
        'writer': None,
    }

        params_DDPG_MADDPG_agent = {
        **vars(arglist),
        'writer': None,
    }


    agent_spec_aileron_PID = AgentSpec('aileron', 'PID', ['aileron'], params_aileron_pid_agent)
    agent_spec_elevator_DDPG = AgentSpec('elevator', 'DDPG', ['elevator'], params_DDPG_MADDPG_agent)

    agent_spec = [agent_spec_elevator_DDPG, agent_spec_aileron_PID]
    agent_container = AgentContainer.init_from_specs(task_list_n, agent_spec, agent_classes_dict, **vars(arglist))

    return agent_container
```

Code Listing D.3: The agent container setup with DDPG control for the elevator task and PID control for the aileron task

D.3 Code for the Definition of Sub-Tasks for a Gliding Descent

```
def make_angular_integral_reward_components_xxxxx(self) -> Tuple[rewards.RewardComponent, ...]:
    ANGLE_DEG_ERROR_SCALING = xxx
    CMD_TRAVEL_MAX = 2/4  # a qarter of the max. absolute value of the delta-cmd;
    ANGLE_INT_DEG_MAX = self.integral_limit
    base_components = (
        rewards.AngularAsymptoticErrorComponent(name='rwd_Angle_error',
                                prop=self.prop_error,
                                state_variables=self.obs_props,
                                target=0.0,
                                potential_difference_based=False,
                                scaling_factor=ANGLE_DEG_ERROR_SCALING,
                                weight=xxx),
        rewards.LinearErrorComponent(name='rwd_cmd_travel_error',
                                prop=self.prop_delta_cmd,
                                state_variables=self.obs_props,
                                target=0.0,
                                potential_difference_based=False,
                                scaling_factor=CMD_TRAVEL_MAX,
                                weight=xxx),
        rewards.LinearErrorComponent(name='rwd_Angle_error_Integral',
                                prop=self.prop_error_integral,
                                state_variables=self.obs_props,
                                target=0.0,
                                potential_difference_based=False,
                                scaling_factor=ANGLE_INT_DEG_MAX,
                                weight=xxx),
    )
    return base_components


def setup_env(arglist) -> NoFGJsbSimEnv_multi:
    agent_interaction_freq = arglist.interaction_frequency
    episode_time_s=arglist.max_episode_len_sec

    glide_path_task = SingleChannel_FlightTask('glide_path', prp.elevator_cmd, {prp.flight_path_deg:
          target_path_angle_gamma_deg},
                                presented_state=[prp.q_radps, prp.indicated_airspeed, prp.elevator_cmd],
                                max_allowed_error= 30,
                                make_base_reward_components= make_angular_integral_reward_components_glide,
                                integral_limit = 0.25)

    banking_task = SingleChannel_FlightTask('banking', prp.aileron_cmd, {prp.roll_deg: target_roll_angle_phi_deg},
                                presented_state=[prp.p_radps, prp.indicated_airspeed, prp.aileron_cmd],
                                max_allowed_error= 60,
                                make_base_reward_components= make_angular_integral_reward_components,
                                integral_limit = 0.25)

    sideslip_task = SingleChannel_FlightTask('sideslip', prp.rudder_cmd, {prp.sideslip_deg: target_sideslip_angle_beta_deg},
                                presented_state=[prp.r_radps, prp.indicated_airspeed, prp.rudder_cmd,
                                aileron_AT_for_PID.setpoint_value_props[0], aileron_AT_for_PID.setpoint_props[0]],   #this
                                      additional state is needed
                                max_allowed_error= 30,
                                make_base_reward_components= make_angular_integral_reward_components_sideslip,
                                integral_limit = 0.25)

    task_list = [glide_path_task, banking_task, sideslip_task]

    env = NoFGJsbSimEnv_multi(task_list, agent_interaction_freq = agent_interaction_freq, episode_time_s = episode_time_s)

    return env
```

Code Listing D.4: The generic code to define the three sub-tasks used in the gliding descent scenario.

D.4 Drop-In Replacement of conventional Control by RL-based Controllers

In the following tables, the characteristic values and settling times of the test-run with three agents independently trained, but applied simultaneously to the gliding descent problem like described in section 6.3 are compared with the performance of the PID controllers used during their training.

Characteristic values of control between events:

event #	0			1			2			3			4			5			6		
	RL control	PID control	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID
event_time	0	0		30.2	30.2		60.2	60.2		90.2	90.2		120.2	120.2		150.2	150.2		180.2	180.2	
peak_time	2.4	1.8		30.8	31.6		61.6	61.6		91.6	91		120.8	123		150.8	151.4		180.6	181.4	
delay_secs	2.4	1.8		0.6	1.4		1.4	1.4		1.4	0.8		0.6	2.8		0.6	1.2		0.4	1.2	
setpoint	-6.5	-6.5		-6	-6		-7.5	-7.5		-7.5	-7.5		-7.5	-7.5		-7.5	-7.5		-6	-6	
actual_value	-6.944	-8.669		-5.076	-5.336		-7.908	-8.443		-7.636	-7.894		-7.215	-7.98		-7.37	-7.18		-4.547	-5.048	
setpoint_change	-6.5	-6.5		0.5	0.5		-1.5	-1.5		0	0		0	0		0	0		1.5	1.5	
abs_overshoot	-0.444	-2.169	20.5%	0.924	0.664	139.2%	-0.408	-0.943	43.3%	-0.136	-0.394	34.5%	0.285	-0.48	-59.4%	0.13	0.32	40.6%	1.453	0.952	152.6%
rel_overshoot	0.068	0.334	20.4%	1.848	1.327	139.3%	0.272	0.629	43.2%	NaN	NaN		NaN	NaN		NaN	NaN		0.969	0.635	152.6%
abs_mean	0.234	0.288	81.3%	0.028	0.039	71.8%	0.037	0.094	39.4%	0.009	0.028	32.1%	0.014	0.04	35.0%	0.003	0.02	15.0%	0.034	0.043	79.1%
MSE	1.097	1.096	100.1%	0.013	0.016	81.3%	0.033	0.064	51.6%	0.001	0.006	16.7%	0.002	0.011	18.2%	0	0.003	0.0%	0.031	0.028	110.7%
actuation_energy	0.173	0.207	83.6%	0.136	0.002	6800.0%	0.118	0.013	907.7%	0.006	0.001	600.0%	0.014	0.002	700.0%	0.001	0		0.877	0.014	6264.3%

Settling times after events given in seconds:

event #	0			1			2			3			4			5			6		
	RL control	PID control	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID
setpoints	-6.5	-6.5		-6	-6		-7.5	-7.5		-7.5	-7.5		-7.5	-7.5		-7.5	-7.5		-6	-6	
setpoint_changes	-6.5	-6.5		0.5	0.5		-1.5	-1.5		0	0		0	0		0	0		1.5	1.5	
error band: 0.5	1.8	4	45.0%	0.8	1.8	44.4%	0.6	2	30.0%	0	0		0	0		0	0		1.6	1.8	88.9%
error band: 0.1	3	6.2	48.4%	3	3.8	78.9%	2.6	4	65.0%	2.8	4.4	63.6%	3.4	4.6	73.9%	0.8	3	26.7%	4	3.4	117.6%
error band: 0.05	4	6.4	62.5%	3	4	75.0%	2.6	4.2	61.9%	3.4	5.4	63.0%	4.8	5.6	85.7%	1	3.4	29.4%	4.8	3.6	133.3%
error band: 0.01	5.6	8.6	65.1%	5.4	5.6	96.4%	6	didn't settle		4.4	7.4	59.5%	6.2	10.6	58.5%	2.6	6.6	39.4%	7.2	46.8	15.4%

Table D.3: Characteristic values and settling times for the glide path angle controller from figure 6.14. Values of the PID control from figure 6.1 are given for comparison.

Characteristic values of control between events:

event #	0			1			2			3			4			5			6		
	RL control	PID control	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID
event_time	0	0		30.2	30.2		60.2	60.2		90.2	90.2		120.2	120.2		150.2	150.2		180.2	180.2	
peak_time	2.2	1		31.6	31		60.8	61.2		92.2	91		123	122.6		151.4	151		181.2	181	
delay_secs	2.2	1		1.4	0.8		0.6	1		2	0.8		2.8	2.4		1.2	0.8		1	0.8	
setpoint	15	15		0	0		0	0		-20	-20		15	15		0	0		0	0	
actual_value	15.433	17.802		-0.675	-4.364		0.092	0.147		-21.015	-26.11		15.567	20.908		-0.801	-4.278		0.176	-0.134	
setpoint_change	15	15		-15	-15		0	0		-20	-20		35	35		-15	-15		0	0	
abs_overshoot	0.433	2.802	15.5%	-0.675	-4.364	15.5%	0.092	0.147	62.6%	-1.015	-6.11	16.6%	0.567	5.908	9.6%	-0.801	-4.278	18.7%	0.176	-0.134	-131.3%
rel_overshoot	0.029	0.187	15.5%	0.045	0.291	15.5%	NaN	NaN		0.051	0.305	16.7%	0.016	0.169	9.5%	0.053	0.285	18.6%	NaN	NaN	
abs_mean	0.303	0.417	72.7%	0.295	0.415	71.1%	0.005	0.015	33.3%	0.392	0.516	76.0%	0.947	1.031	91.9%	0.29	0.381	76.1%	0.006	0.007	85.7%
MSE	2.877	3.015	95.4%	2.971	2.859	103.9%	0	0.001	0.0%	4.986	4.852	102.8%	19.624	16.673	117.7%	2.803	2.666	105.1%	0.001	0	
actuation_energy	0.957	0.413	231.7%	0.293	0.466	62.9%	0	0		0.638	0.872	73.2%	1.46	1.399	104.4%	0.286	0.489	58.5%	0.003	0	

Settling times after events given in seconds:

event #	0			1			2			3			4			5			6		
	RL control	PID control	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID
setpoints	15	15		0	0		0	0		-20	-20		15	15		0	0		0	0	
setpoint_changes	15	15		-15	-15		0	0		-20	-20		35	35		-15	-15		0	0	
error band: 0.5	1.4	5.6	25.0%	1.6	3.4	47.1%	0	0		2.4	4.6	52.2%	3.4	8.2	41.5%	2.2	3	73.3%	0	0	
error band: 0.1	3.6	9.4	38.3%	3.8	8.2	46.3%	0	1.4	0.0%	4.2	8	52.5%	5	13.4	37.3%	3.8	7.4	51.4%	2.2	1.2	183.3%
error band: 0.05	3.8	11.8	32.2%	4.4	10.2	43.1%	2	2.6	76.9%	4.6	10.4	44.2%	5	15.8	31.6%	5	9.4	53.2%	4.2	2.2	190.9%
error band: 0.01	4.4	17	25.9%	7.4	15.2	48.7%	5.4	21.4	25.2%	6	15.2	39.5%	6.8	21.8	31.2%	8.4	14	60.0%	7.4	4.8	154.2%

Table D.4: Characteristic values and settling times for the banking angle controller from figure 6.14. Values of the PID control from figure 6.1 are given for comparison.

Characteristic values of control between events:

event #	0	1	2	3	4	5	6
	RL control	RL	RL	RL	RL	RL	RL
event_time	0	30.2	60.2	90.2	120.2	150.2	180.2
peak_time	1.6	31.2	61.4	90.6	121.2	151.2	181.2
delay_secs	1.6	1	1.2	0.4	1	1	1
setpoint	0	0	0	0	0	0	0
actual_value	0.265	0.255	-0.019	0.493	-0.635	0.427	0.057
setpoint_change	0	0	0	0	0	0	0
abs_overshoot	0.265	0.255	-0.019	0.493	-0.635	0.427	0.057
rel_overshoot	NaN	NaN	NaN	NaN	NaN	NaN	NaN
abs_mean	0.014	0.013	0.001	0.027	0.036	0.022	0.002
MSE	0.002	0.002	0	0.008	0.012	0.004	0
actuation_energy	0.464	0.314	0.001	1.355	1.711	0.537	0.007

Settling times after events given in seconds:

event #	0	1	2	3	4	5	6
	RL control	RL	RL	RL	RL	RL	RL
setpoints	0	0	0	0	0	0	0
setpoint_changes	0	0	0	0	0	0	0
error band: 0.5	0	0	0	0	2.4	0	0
error band: 0.1	2.4	2.6	0	2.8	4.8	3.6	0
error band: 0.05	3.6	3.6	0	4	5.8	4.4	1.6
error band: 0.01	5.4	5.6	1.8	5.2	7.4	7.2	5.8

Table D.5: Characteristic values and settling times for the sideslip compensation controller from figure 6.14. No PID control values for comparison are available.

D.5 Three cooperating DDPG Agents

In the following tables, the characteristic values and settling times of the test-run with three cooperating DDPG agents like described in section 6.3 are given for reference.

Characteristic values of control between events:

event #	0			1			2			3			4			5			6		
	RL control	PID control	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID
event_time	0	0		30.2	30.2		60.2	60.2		90.2	90.2		120.2	120.2		150.2	150.2		180.2	180.2	
peak_time	1.8	1.8		30.8	31.6		60.8	61.6		90.8	91		121	123		150.8	151.4		180.8	181.4	
delay_secs	1.8	1.8		0.6	1.4		0.6	1.4		0.6	0.8		0.8	2.8		0.6	1.2		0.6	1.2	
setpoint	-6.5	-6.5		-6	-6		-7.5	-7.5		-7.5	-7.5		-7.5	-7.5		-7.5	-7.5		-6	-6	
actual_value	-6.975	-8.669		-5.475	-5.336		-7.926	-8.443		-7.262	-7.894		-7.204	-7.98		-7.205	-7.18		-5.392	-5.048	
setpoint_change	-6.5	-6.5		0.5	0.5		-1.5	-1.5		0	0		0	0		0	0		1.5	1.5	
abs_overshoot	-0.475	-2.169	21.9%	0.525	0.664	79.1%	-0.426	-0.943	45.2%	0.238	-0.394	-60.4%	0.296	-0.48	-61.7%	0.295	0.32	92.2%	0.608	0.952	63.9%
rel_overshoot	0.073	0.334	21.9%	1.049	1.327	79.1%	0.284	0.629	45.2%	NaN	NaN		NaN	NaN		NaN	NaN		0.405	0.635	63.8%
abs_mean	0.205	0.288	71.2%	0.016	0.039	41.0%	0.037	0.094	39.4%	0.01	0.028	35.7%	0.009	0.04	22.5%	0.005	0.02	25.0%	0.017	0.043	39.5%
MSE	1.003	1.096	91.5%	0.006	0.016	37.5%	0.033	0.064	51.6%	0.001	0.006	16.7%	0.001	0.011	9.1%	0.001	0.003	33.3%	0.015	0.028	53.6%
actuation_energy	0.204	0.207	98.6%	0.052	0.002	2600.0%	0.101	0.013	776.9%	0.028	0.001	2800.0%	0.004	0.002	200.0%	0.015	0		0.151	0.014	1078.6%

Settling times after events given in seconds:

event #	0			1			2			3			4			5			6		
	RL control	PID control	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID
setpoints	-6.5	-6.5		-6	-6		-7.5	-7.5		-7.5	-7.5		-7.5	-7.5		-7.5	-7.5		-6	-6	
setpoint_changes	-6.5	-6.5		0.5	0.5		-1.5	-1.5		0	0		0	0		0	0		1.5	1.5	
error band: 0.5	1.4	4	35.0%	0.8	1.8	44.4%	0.4	2	20.0%	0	0		0	0		0	0		0.8	1.8	44.4%
error band: 0.1	2.8	6.2	45.2%	2	3.8	52.6%	1.8	4	45.0%	2.2	4.4	50.0%	2.4	4.6	52.2%	0.8	3	26.7%	1.6	3.4	47.1%
error band: 0.05	2.8	6.4	43.8%	2.2	4	55.0%	2.6	4.2	61.9%	2.4	5.4	44.4%	2.4	5.6	42.9%	2	3.4	58.8%	2	3.6	55.6%
error band: 0.01	4	8.6	46.5%	3.2	5.6	57.1%	2.8	didn't settle		4.8	7.4	64.9%	4.4	10.6	41.5%	3	6.6	45.5%	3.6	46.8	7.7%

Table D.6: Characteristic values and settling times for the glide path angle controller from figure 6.3. Values of the PID control from figure 6.1 are given for comparison.

Characteristic values of control between events:

event #	0			1			2			3			4			5			6		
	RL control	PID control	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID
event_time	0	0		30.2	30.2		60.2	60.2		90.2	90.2		120.2	120.2		150.2	150.2		180.2	180.2	
peak_time	1.8	1		32.4	31		60.8	61.2		92.2	91		122.4	122.6		152.2	151		181.2	181	
delay_secs	1.8	1		2.2	0.8		0.6	1		2	0.8		2.2	2.4		2	0.8		1	0.8	
setpoint	15	15		0	0		0	0		-20	-20		15	15		0	0		0	0	
actual_value	15.33	17.802		-0.48	-4.364		0.094	0.147		-20.729	-26.11		15.686	20.908		-0.526	-4.278		0.081	-0.134	
setpoint_change	15	15		-15	-15		0	0		-20	-20		35	35		-15	-15		0	0	
abs_overshoot	0.33	2.802	11.8%	-0.48	-4.364	11.0%	0.094	0.147	63.9%	-0.729	-6.11	11.9%	0.686	5.908	11.6%	-0.526	-4.278	12.3%	0.081	-0.134	-60.4%
rel_overshoot	0.022	0.187	11.8%	0.032	0.291	11.0%	NaN	NaN		0.036	0.305	11.8%	0.02	0.169	11.8%	0.035	0.285	12.3%	NaN	NaN	
abs_mean	0.289	0.417	69.3%	0.259	0.415	62.4%	0.004	0.015	26.7%	0.385	0.516	74.6%	0.857	1.031	83.1%	0.24	0.381	63.0%	0.002	0.007	28.6%
MSE	2.952	3.015	97.9%	2.484	2.859	86.9%	0	0.001	0.0%	4.819	4.852	99.3%	18.582	16.673	111.4%	2.292	2.666	86.0%	0	0	
actuation_energy	0.593	0.413	143.6%	0.856	0.466	183.7%	0	0		0.892	0.872	102.3%	1.783	1.399	127.4%	0.8	0.489	163.6%	0	0	

Settling times after events given in seconds:

event #	0			1			2			3			4			5			6		
	RL control	PID control	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID	RL	PID	ratio RL/PID
setpoints	15	15		0	0		0	0		-20	-20		15	15		0	0		0	0	
setpoint_changes	15	15		-15	-15		0	0		-20	-20		35	35		-15	-15		0	0	
error band: 0.5	1.2	5.6	21.4%	1	3.4	29.4%	0	0		2.4	4.6	52.2%	2.8	8.2	34.1%	2.2	3	73.3%	0	0	
error band: 0.1	3	9.4	31.9%	3.2	8.2	39.0%	0	1.4	0.0%	4.6	8	57.5%	5	13.4	37.3%	3	7.4	40.5%	0	1.2	0.0%
error band: 0.05	3.2	11.8	27.1%	4.8	10.2	47.1%	1.4	2.6	53.8%	6	10.4	57.7%	5.6	15.8	35.4%	4.6	9.4	48.9%	1.4	2.2	63.6%
error band: 0.01	5.2	17	30.6%	6.2	15.2	40.8%	3.2	21.4	15.0%	9.2	15.2	60.5%	7.4	21.8	33.9%	7.6	14	54.3%	2.8	4.8	58.3%

Table D.7: Characteristic values and settling times for the banking angle controller from figure 6.3. Values of the PID control from figure 6.1 are given for comparison.

Characteristic values of control between events:

event #	0	1	2	3	4	5	6
	RL control	RL	RL	RL	RL	RL	RL
event_time	0	30.2	60.2	90.2	120.2	150.2	180.2
peak_time	0.2	30.6	61	91	121	150.6	181.4
delay_secs	0.2	0.4	0.8	0.8	0.8	0.4	1.2
setpoint	0	0	0	0	0	0	0
actual_value	0.078	-0.44	0.016	0.34	-0.603	-0.356	0.021
setpoint_change	0	0	0	0	0	0	0
abs_overshoot	0.078	-0.44	0.016	0.34	-0.603	-0.356	0.021
rel_overshoot	NaN	NaN	NaN	NaN	NaN	NaN	NaN
abs_mean	0.01	0.016	0.001	0.022	0.034	0.016	0.001
MSE	0.001	0.003	0	0.004	0.011	0.002	0
actuation_energy	0.383	0.293	0	0.266	0.939	0.187	0

Settling times after events given in seconds:

event #	0	1	2	3	4	5	6
	RL control	RL	RL	RL	RL	RL	RL
setpoints	0	0	0	0	0	0	0
setpoint_changes	0	0	0	0	0	0	0
error band: 0.5	0	0	0	0	2	0	0
error band: 0.1	2.2	2.8	0	3.6	4.2	2.6	0
error band: 0.05	2.2	4	0	4.8	4.8	3.8	0
error band: 0.01	4.2	5.4	2.2	8.4	6.6	6.8	2.4

Table D.8: Characteristic values and settling times for the sideslip compensation controller from figure 6.3. No PID control values for comparison are available.

ERKLÄRUNG

Name: Felix Eckstein
Matrikel-Nr.: 8161569
Fach: Informatik
Modul: Masterarbeit

Ich erkläre, dass ich die vorliegende Abschlussarbeit mit dem Thema

Learning to Fly - Building an Autopilot System based on Neural Networks and Reinforcement Learning

selbstständig und ohne unzulässige Inanspruchnahme Dritter verfasst habe. Ich habe dabei nur die angegebenen Quellen und Hilfsmittel verwendet und die aus diesen wörtlich, inhaltlich oder sinngemäß entnommenen Stellen als solche den wissenschaftlichen Anforderungen entsprechend kenntlich gemacht. Die Versicherung selbstständiger Arbeit gilt auch für Zeichnungen, Skizzen oder graphische Darstellungen. Die Arbeit wurde bisher in gleicher oder ähnlicher Form weder derselben noch einer anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht. Mit der Abgabe der elektronischen Fassung der endgültigen Version der Arbeit nehme ich zur Kenntnis, dass diese mit Hilfe eines Plagiatserkennungsdienstes auf enthaltene Plagiate überprüft und ausschließlich für Prüfungszwecke gespeichert wird. Außerdem räume ich dem Lehrgebiet das Recht ein, die Arbeit für eigene Lehr- und Forschungstätigkeiten auszuwerten und unter Angabe des Autors geeignet zu publizieren.

Lüneburg, den 2. Juni 2020

Felix Eckstein