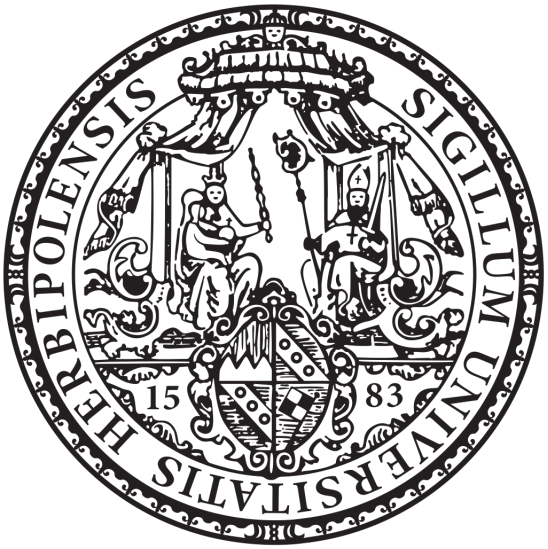
Deep Learning Attitude Control And Simulator For Satellites



Fabian Arzberger
Bachelor thesis
Supervised by: Prof. Dr.-Ing. Sergio Montenegro
Informationstechnik für Luft- und Raumfahrt
Universität Würzburg
fabian.arzberger@stud-mail.uni-wuerzburg.de

16.09.2019

Contents

List of Figures

Abstract

This thesis proposes a method to substituting a conventional PID controller, used for 3-dimensional attitude control of reaction wheel driven satellites, with a deep neural network (DNN) operating in a closed control loop with error based reference trajectory. Initially, a mathematical model of the system is derived, then an attitude control simulation is created to show the success of the substitution. A low-frequency sine-wave disturbance is also simulated, representing axis precession. The simulation shows that even though the commonly used PID controller produces more reliable results, the DNN solution also works very well.

1 Introduction

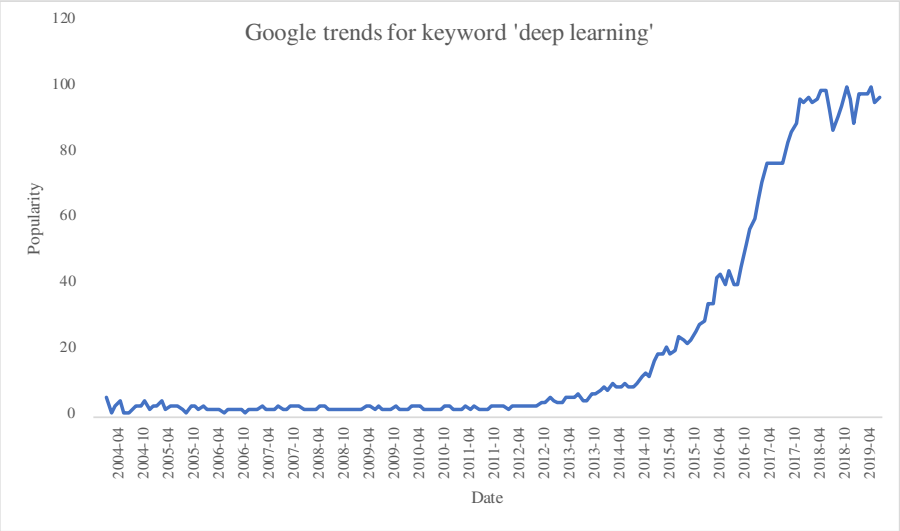


Figure 1: Popularity of 'deep learning' on Google trends

As Figure 1 shows: since records of Google trends began, the search popularity of the term "deep learning" has grown exponentially fast. A value of 100 corresponds to the highest measured search popularity.

This sharp increase in popularity is the result of the excellent performance of new neural network optimizing methods that can be used in many domains of computer science, such as face identification, image and voice recognition, language translation and many more.

Hagan, Demuth, and Jesús (2002) have shown in their paper that neural networks are also suitable for usages in non-linear control systems, such as

a continuous stirred tank reactor, a robot arm, and a magnetic levitation system, due to their nature of being efficient in function approximation, which will be shown in section 4.

However, these applications are single-input-single-output (SISO) systems. Therefore, a neural network can generalize the non-linear behavior of those systems more quickly than the behavior of a multiple-input-multiple-output (MIMO) system, such as the 3-dimensional attitude control of a spacecraft is. A larger amount of training data and an error-based reference trajectory is used as an input to the neural network instead of a feed-forward reference trajectory to ensure training convergence nevertheless. (Carrara & Rios Neto, 1999)

The overall idea of getting the neural network controller to work is first to derive a general mathematical model of the spacecraft system in section 2. The model is then used in order to build a simulation in section 6. The simulation can produce sample data to feed the neural network. Then, the simulation is used to validate the performance of the neural network after its training process and to compare it to the PIDs results in section 7.

2 Satellite model

The rotational inertia Θ of any given spacecraft describes the coherence of torque and angular acceleration about a specific rotational axis. It can be described with a symmetric, positive definite 3x3 matrix, where the diagonal items are called the moment of inertia, whereas the non-diagonal items are called the moment of deviation.

$$\Theta = \begin{pmatrix} \Theta_{11} & \Theta_{12} & \Theta_{13} \\ \Theta_{21} & \Theta_{22} & \Theta_{23} \\ \Theta_{31} & \Theta_{32} & \Theta_{33} \end{pmatrix} \tag{1}$$

A component Θ_{ij} of the rotational inertia can be analytically calculated for any physical object, using a volume integral,

$$\Theta_{ij} = \int_V \rho(\vec{r}) \left[(\vec{r} \cdot \vec{r})\delta_{ij} \; - \; r_i r_j\right] \mathrm{d}V \tag{2}$$

given the mass density $\rho(\vec{r})$ and the Kronecker delta δ_{ij}. (Goldstein, 1981)

$$\delta_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{else} \end{cases} \tag{3}$$

As Goldstein (1981) shows, the volume integral over a homogeneous cube simplifies to the rotational inertia of

$$\Theta^c = \frac{m}{6} d^2 \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{4}$$

where m is the mass and d is the side length.

However, in real-world conditions, a satellite will not be a homogeneous solid. For this reason, when simulating a cube-sat, it makes sense adding little, arbitrary chosen momenta of deviation. For any other satellite that is not a cube-sat, 3D-modelling software can approximate the rotational inertia.

2.1 In-orbit attitude kinematics

The rotational inertia Θ, together with the angular momentum $\vec{L}$ determines the 3-dimensional rotational speed $\vec{\omega}$ of a satellite.

$$\vec{L} = \Theta \cdot \vec{\omega} \tag{5}$$

Since the rotational inertia Θ is a symmetric 3x3 matrix (section 2) and therefore an inverse matrix Θ^{-1} exists, Equation 5 can be converted to

$$\vec{\omega} = \Theta^{-1} \cdot \vec{L} \tag{6}$$

The angular momentum $\vec{L}$ is a conserved quantity. Thus, only external torques like those created by the reaction wheels may change it. (Meschede & Gerthsen, 2005) A reaction wheel is a spinning flywheel that induces a rotation speed to the satellite it is attached to. When the rotation speed of the reaction wheel changes, it causes the satellite to counter-rotate due to the conservation of angular momentum. (NASA, n.d.) In this thesis, a reaction wheel is modelled as a homogeneous cylinder with mass m_w, radius r_w and height h_w. The rotational inertia of such a reaction wheel has no moment of deviation and can be written like

$$\Theta^w = \begin{pmatrix} \Theta_x^w & 0 & 0 \\ 0 & \Theta_y^w & 0 \\ 0 & 0 & \Theta_z^w \end{pmatrix} \tag{7}$$

The z-component of the inertia matrix can be calculated using only the mass m_w and the radius r_w of the cylinder-shaped reaction wheel. (Serway, 1986)

$$\Theta_z^w = \frac{1}{2} \cdot m_w \cdot {r_w}^2 \tag{8}$$

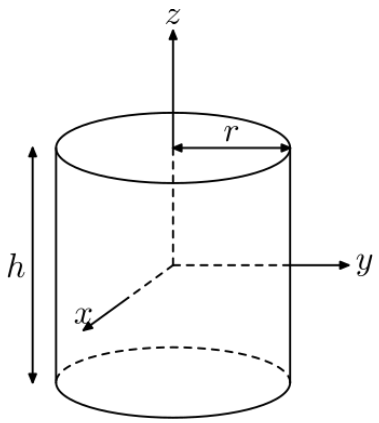


Figure 2: Reaction wheel model in reaction wheel reference frame

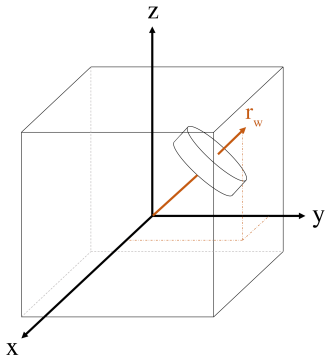


Figure 3: Reaction wheel in satellite reference frame

Since the reaction wheel only rotates around the z-axis, which can be seen in Figure 2, the x and y components of the matrix in Equation 7 do not have any influence on the rotation of the wheel. Therefore, the angular momentum L_w of the reaction wheel, seen from the reaction wheels frame, is determined only by the z-component of the inertia matrix and the rotation speed of the wheel ω_w

$$L_w = \Theta_z^w \cdot \omega_w = \frac{1}{2} \cdot m_w \cdot {r_w}^2 \cdot \omega_w \tag{9}$$

Seen from the satellites reference frame, shown in Figure 3, the angular momentum of the reaction wheel is not a scalar, but a 3-dimensional vector pointing in the direction of $\vec{r}_w$, which is the rotational axis of the reaction wheel. Defining $\vec{r}_w$ as a unit vector, the angular momentum of the reaction wheel in the satellite frame is

$$\vec{L}_w = L_w \cdot \vec{r}_w \tag{10}$$

The overall angular momentum of the satellite $\vec{L}$ is the sum of all the angular momenta of the reaction wheels, due to the conservation of angular momentum.

$$\vec{L} = \sum_w \vec{L}_w \tag{11}$$

Bringing together Equation 11, Equation 10, and Equation 9 with Equation 6, the rotation speed of a satellite $\vec{\omega}$ can be expressed with

$$\vec{\omega} = \Theta^{-1} \cdot \sum_w \frac{1}{2} m_w \omega_w {r_w}^2 \cdot \vec{r}_w \tag{12}$$

2.2 In-orbit attitude dynamics

Considering Equation 12, the rotation speed ω_w of the reaction wheels has to be calculated to fully determine the rotational motion of the satellite, neglecting utter influences such as solar or atmospheric pressure. The rotation speed ω_w of a reaction wheel depends on its angular momentum L_w.

In order to change the angular momentum L_w of a reaction wheel, a torque M has to be established. (Meschede & Gerthsen, 2005)

$$M = \dot{L}_w \tag{13}$$

Using Equation 9 to build the time derivative gives

$$M = \Theta_z^w \cdot \dot{\omega}_w = \Theta_z^w \cdot \alpha \tag{14}$$

where the angular acceleration α can be also expressed like

$$\alpha = \frac{M}{\Theta_z^w} \tag{15}$$

The rotation speed of a reaction wheel ω_w finally is

$$\omega_w = \int \alpha dt = \int \frac{M}{\Theta_z^w} dt \tag{16}$$

3 Fundamentals of a PID controller

A PID controller is one of the most classical approaches when it comes to system control. The purpose of the controller is for any set-point input $r(t)$ to generate an output trajectory $u(t)$ that, when applied to the process being controlled, minimizes the error $e(t)$ between the desired and the measured process value $y(t)$. It does so by using **p**roportional, **i**ntegral and **d**erivate parts of the error function $e(t)$

$$e(t) = r(t) - y(t) \tag{17}$$

The output $u(t)$ is then calculated using

$$u(t) = K_p \cdot e(t) + K_i \cdot \int_{t_0}^{t} e(\tau) d\tau + K_d \cdot \frac{d}{dt} e(t) \tag{18}$$

where K_p, K_i and K_d are coefficients, weighting the proportional, integral and derivative term. (King, 2010)

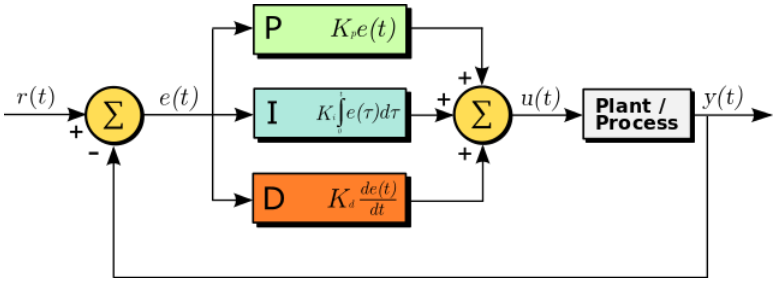


Figure 4: Block diagram of a feedback PID controller

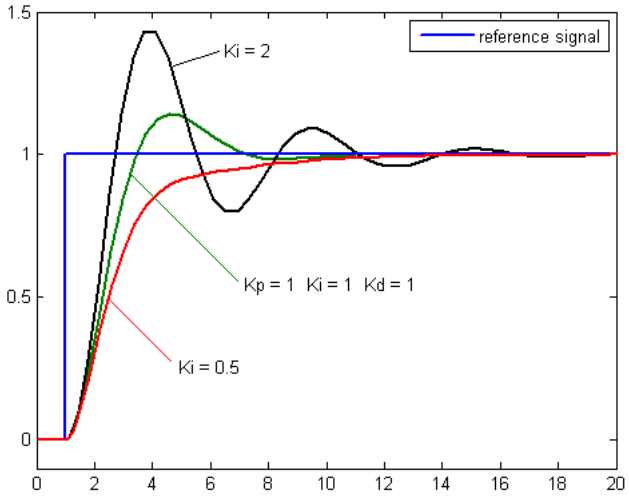


Figure 5: Step response of process value with variation of K_i while holding K_p and K_d const.

When operating in a closed loop, shown in Figure 4, with a known sample time, this controller is able to minimize the error $e(t)$ so that the process value $y(t)$ converges to the desired value $r(t)$. Figure 5 shows exemplary step responses that a PID controller produces for a given desired reference signal. Furthermore, it shows variation of the K_i parameter. The next chapter focuses more on tuning the PID parameters.

3.1 Tuning the PID parameters

Tuning the K_p, K_i and K_d parameters is essential for a well-working PID controller. Many methods can be used for tuning the parameters, e.g. Ziegler-Nichols, Cohen-Coon or Åström-Hägglund. (Atherton, 2014) Manual tuning is also a conventional tuning method that provides better understanding of how the parameters affect the step response.

- The $K_p e(t)$ term is proportional to the error. Therefore, the controller generates a significant control output $u(t)$ if the error is large. However, if the process value converges to the desired value over time, the control output decreases. That is why using a P-controller ($K_i = K_d = 0$) will often lead to steady-state errors that can not be corrected by the proportional term. (King, 2010)

- The $K_i \int e(\tau)d\tau$ term integrates over the error, adding up all the previous errors into the control output. It is for this reason that the integral term strives for correcting steady-state errors. The residual error adds up over time until the control output gets large enough to correct the error. (King, 2010)

- The $K_d \frac{d}{dt} e(t)$ term can not be used for system control alone, because it does not react to the error but rather to its derivative. Paired with a P- or PI-controller, it weakens the control output the larger the change in the error is. Thus, the derivative term (paired with a P- or PI-controller) prevents overshooting. (King, 2010)

With this behavior of parameter tuning in mind, the following table stated by Ang, Chong, and Li (2005) is reasonable to derive. It shows the impact of increasing K_p, K_i or K_d separately:

	Rise Time	Overshoot	Settling time	Steady-state error	Stability
K_p	Decrease	Increase	Small change	Decrease	Degrade
K_i	Decrease	Increase	Increase	Eliminate	Degrade
K_d	Minor change	Decrease	Decrease	No effect in theory	Improve if K_d is small

Table 1: Effects to the step-response of increasing a parameter independently

3.2 Advanced modifications

The pure PID algorithm, as stated in Equation 18, can be modified to perform even better. Beauregard (2011) shows several tweaks that can be applied for any PID controller. However, in the simulation environment, not all of them are required. It is, for example, not necessary to change the PID parameters while the controller is operating.

One improvement to the classical PID controller is to get rid of **derivative kick**. The problem is that since the $K_d \frac{d}{dt} e(t)$ term uses the derivative of the error, a huge spike in the output is measurable whenever the desired value changes, because that is when the change in error is largest.

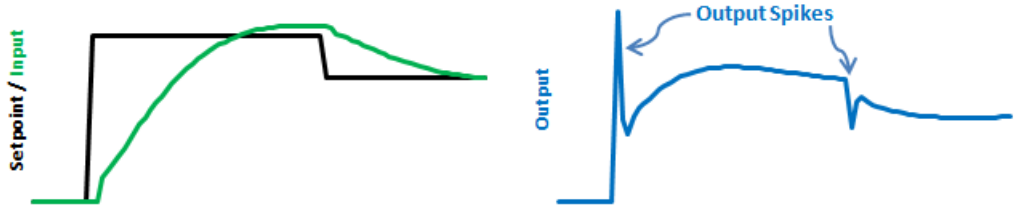


Figure 6: Derivative kick illustrated

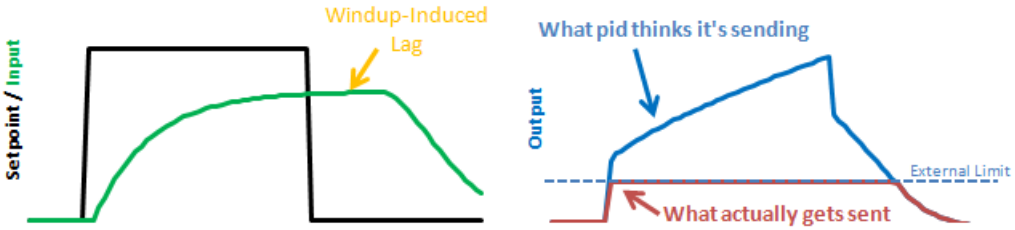


Figure 7: Reset windup illustrated

Figure 6 shows that behavior. The desired value, represented by the black line, almost instantly jumps, resulting in a significant change in the error function. The derivative term accounts that into the output, hence the spikes. The solution to the problem consists in modifying the derivative term. Deriving Equation 17 gives

$$\frac{d}{dt}e(t) = \frac{d}{dt}r(t) - \frac{d}{dt}y(t) \tag{19}$$

However, it appears that the desired value $r(t)$ is constant most of the time, resulting in $\frac{d}{dt}r(t)$ to equal zero. In fact, it is the $\frac{d}{dt}r(t)$ term that causes the output spikes, so it makes perfect sense excluding it. The new PID algorithm therefore becomes

$$u(t) = K_p \cdot e(t) + K_i \cdot \int_{t_0}^{t} e(\tau)d\tau - K_d \cdot \frac{d}{dt}y(t) \tag{20}$$

Another improvement is to get rid of a phenomenon called **reset windup**. It occurs whenever the output of the PID controller grows beyond an empirically meaningful value. Figure 7 shows that when the programmer does not set any limits for the controller's output, it likely tries outputs beyond the system limits even if the plant (see Figure 4) is not able to process them as intended, resulting in a windup effect when the desired value is reset. To solve this issue, the programmer needs to specify sensible limits for the output $u(t)$, but also the integral term $K_i \int e(t)$. (Beauregard, 2011)

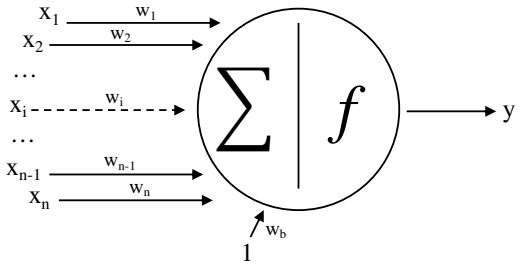


Figure 8: Model of an artificial neuron

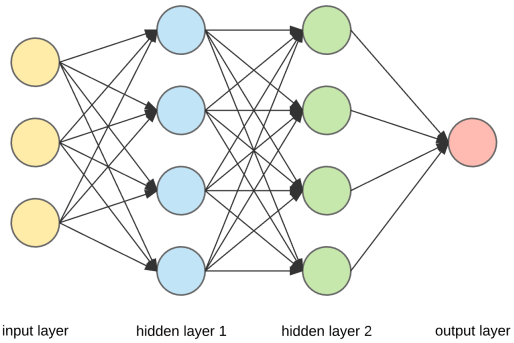


Figure 9: Multiple neurons connected in layers forming a neural network

4 Fundamentals of a neural network

A neural network consists of so-called "neurons," imitating the kind of neurons in the human brain. Just like in the brain, these neurons are connected, some of them having a stronger connection than others. From this perception, a neuron model is derived, shown in Figure 8. A neuron can have multiple inputs x_i, weighted by w_i, that gets summed up together with a bias w_b.

$$z = \sum_i w_i x_i + w_b \tag{21}$$

The sum gets passed to an activation function f to produce an output y.

$$y = f(z) \tag{22}$$

This output can then be used as an input for another neuron. Connecting various neurons in multiple layers creates a feed-forward neural network, as shown in Figure 9. When considering a single layer made of any number of neurons, Equation 22 extends into matrix form, where W is the weight matrix, $\vec{x}$ the input vector, $\vec{y}$ the output vector and $\vec{b}$ the bias vector.

$$\vec{y} = f(W\vec{x} + \vec{b}) = f(\vec{z}) \tag{23}$$

Since the output of one layer is the input for the next layer, Equation 23 can be generalized as

$$\vec{y}^m = f^m(W^m \vec{y}^{m-1} + \vec{b}^m) \quad \text{for m} = 1, 2, \ldots, \text{M} \tag{24}$$

where M is the number of layers in the neural network. This equation is enough to feed data through the network to obtain an output. However,

the weights and biases are chosen randomly at the beginning, so you may not expect good performance from the network. Instead, the output will be unpredictable and meaningless. Consider a set of data with N samples

$$\{\vec{x}_1, \vec{t}_1\}, \{\vec{x}_2, \vec{t}_2\}, ..., \{\vec{x}_N, \vec{t}_N\} \tag{25}$$

where $\vec{x}_n$ is an input to the network and $\vec{t}_n$ is the desired target output. The network's purpose is to generalize the relationship between $\vec{x}_n$ and $\vec{t}_n$ given only the samples. The process of learning from the samples comes down to finding the optimal weights and biases for the neuron connections in the network. Therefore, the cost function C, which is a measure for the performance of the network, has to be minimized. The cost of one sample is

$$C = \sum_{q=1}^{n_M} (t_q - y_q^M)^2 \tag{26}$$

where n_M is the number of neurons in the output layer and t_q is the target output for the q-th neuron in the output layer. Consider a connection between two neurons in layer m, connecting the previous layers i-th neuron to the j-th neuron of layer m. The weight of that connection at iteration k should be updated like the following

$$w_{i,j}^m(k+1) = w_{i,j}^m(k) - \alpha \cdot \frac{\delta C}{\delta w_{i,j}^m} \tag{27}$$

where α is a positive, arbitrarily chosen and small hyper-parameter called learning rate. The update of the i-th neuron's bias in layer m is computed similarly:

$$b_i^m(k+1) = b_i^m(k) - \alpha \cdot \frac{\delta C}{\delta b_i^m} \tag{28}$$

The derivatives of the cost function C seem unintuitive at first. However, the chain rule can be used in order to calculate the derivatives of the cost function with respect to the weights

$$\frac{\delta C}{\delta w_{i,j}^m} = \frac{\delta z_i^l}{\delta w_{i,j}^m} \cdot \frac{\delta y_i^m}{\delta z_i^l} \cdot \frac{\delta C}{\delta y_i^m} \tag{29}$$

and the biases

$$\frac{\delta C}{\delta b_i^m} = \frac{\delta z_i^l}{\delta b_i^m} \cdot \frac{\delta y_i^m}{\delta z_i^l} \cdot \frac{\delta C}{\delta y_i^m} \tag{30}$$

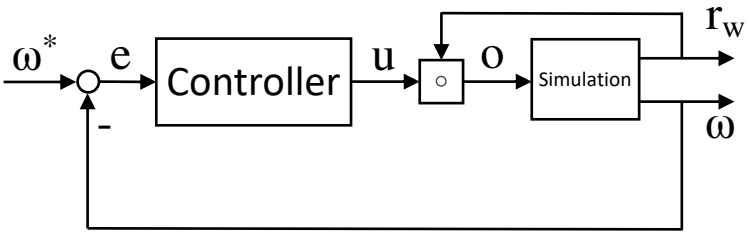


Figure 10: Block diagram of attitude control

Differentiating Equation 21 and Equation 22 leads to

$$\frac{\delta C}{\delta w_{i,j}^m} = y_j^{m-1} \cdot f'(z_i^l) \cdot \frac{\delta C}{\delta y_i^m} \tag{31}$$

and

$$\frac{\delta C}{\delta b_i^m} = f'(z_i^l) \cdot \frac{\delta C}{\delta y_i^m} \tag{32}$$

Building the derivative of the cost C with respect to the output y_i^m of the i-th neuron in the m-th layer is trivial if this neurons output feeds forward in only one connection (e.g. the output layer of the neural network). Differentiating Equation 26 then gives

$$\frac{\delta C}{\delta y_i^m} = 2 \cdot (t_i - y_i^M) \tag{33}$$

Otherwise, if the neurons output feeds forward in multiple paths, the derivative must be calculated differently:

$$\frac{\delta C}{\delta y_i^m} = \sum_{i=1}^{n_{m+1}} (w_{i,j}^{m+1} \cdot f'(z_i^{m+1}) \cdot \frac{\delta C}{\delta y_i^{m+1}}) \tag{34}$$

where n_{m+1} is the number of neurons in the layer $m+1$. (Hagan et al., 2002) (Sanderson, 2017)

5 Attitude control

The control archetypes presented in section 3 and section 4 shall now be used to control the rotational speed ω of a satellite. A simulation is used to evaluate the movements of the satellite, implementing the model stated in section 2. Figure 10 shows the framework structure of the attitude control.

The controller uses the 3D rotation speed error e to produce a 3D output u. The components of the vector u reveal how the rotation speed should change on a certain axis. However, the rotational axes of the reaction wheels will most likely not be parallel to the coordinate axes of the reference frame in which the satellite moves in. In order to find out how much impact a certain wheel has on the reference frame axes, the control output u gets multiplied with the rotation axis r_w of the reaction wheel (which is seen from the reference frame), using a dot product. When this product is calculated for every reaction wheel, the components of the vector o contain values proportional to the power that the wheels need to consume. The simulation then feeds that to the wheels, resulting in rotation. In a small time step Δt, where the rotation speed is considered constant, the Euler-angles roll α, pitch β and yaw γ change about

$$\alpha = \omega_x \cdot \Delta t \tag{35}$$

$$\beta = \omega_y \cdot \Delta t \tag{36}$$

$$\gamma = \omega_z \cdot \Delta t \tag{37}$$

Therefore, the rotational axis of a reaction wheel gets rotated by these angles. The updated rotational axis at iteration $k+1$ can be calculated using the ZY'X" convention

$$r_w(k+1) = \begin{pmatrix} c_\gamma c_\beta & c_\gamma s_\beta s_\alpha - s_\gamma c_\alpha & c_\gamma s_\beta c_\alpha + s_\gamma s_\alpha \\ s_\gamma c_\beta & s_\gamma s_\beta s_\alpha + c_\gamma c_\alpha & s_\gamma s_\beta c_\alpha c_\gamma s_\alpha \\ -s_\beta & c_\beta s_\alpha & c_\beta c_\alpha \end{pmatrix} r_w(k) \tag{38}$$

where c refers to the cosine function, s to the sine function and the subscripts of α, β and γ to the Euler-angles. In conclusion, the first entry of the matrix can be written as $\cos(\gamma) \cdot \cos(\beta)$.

5.1 Attitude control with PID

The rotation speed can be controlled straightforward using the PID algorithm. However, the controller presented in section 3 needs some adjustments.

First, the limits for the integral and the output have to be set. A reasonable value for a limit is the maximum power a wheel can consume. This might be a certain voltage or something proportional to it, depending on the implementation. As will be presented in section 6, the wheels in the simulation accept values between negative one and one, corresponding with the

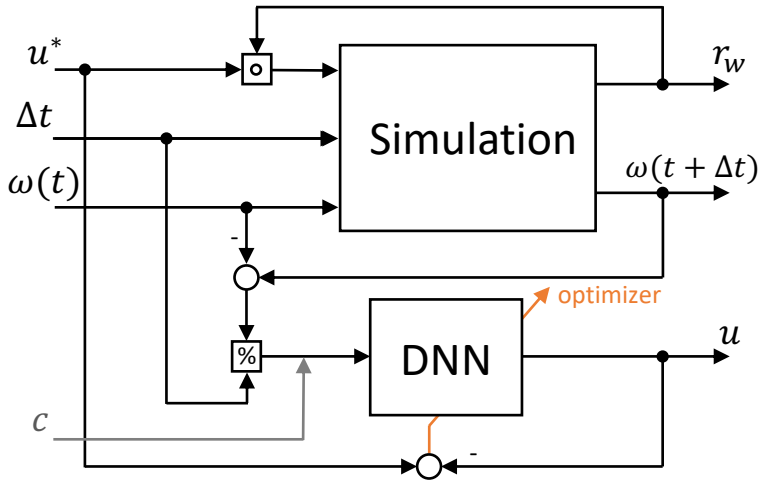


Figure 11: Block diagram of DNN attitude control learning process

maximum torque (negative and positive). The value zero means no power to the wheel at all.

Second, a sample time, which is the time the PID controller waits before it calculates its output again, needs to be set. The simulation can only work in small, yet discrete time steps. The sample time of the PID controller must be chosen bigger than the simulation time steps. In this case, the PID controller operates with a sample time of $t_s = 0.5$.

Third, the K_p, K_i and K_d values have to be set. Since their optimal values vary for every setup of satellite + wheels, they will be shown together with the performance of the controller in section 7.

5.2 Attitude control with DNN

The DNN approach is not as straightforward as the PID algorithm, where just a few hyper-paramters need to be set. Controlling the rotation speed of a spacecraft with a deep neural network requires not only the pure DNN algorithm, but also a learning strategy. Such a strategy is depicted in Figure 11. At time t, the satellite has an initial rotational speed of $\omega(t)$. After a time period Δt has passed while a control output u^* has been applied, the rotation axes r_w of the wheels and the rotation speed $\omega(t+\Delta t)$ has changed. The input $\vec{x}$ to the DNN is

$$\vec{x} = c \cdot \frac{\vec{\omega}(t+\Delta t) - \vec{\omega}(t)}{\Delta t} \tag{39}$$

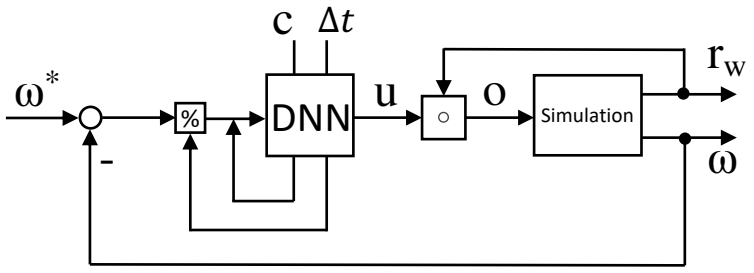


Figure 12: Block diagram of DNN operating in a closed-loop control mode

where c is an input scaling factor. The neural network makes a guess u about the control output that was applied in the first place. An optimizer then uses the error between the guess u and the target u^* to tweak all the weights and biases in order to improve the networks performance.

One optimizing method was presented in section 4, called stochastic gradient descent (SGD). However, there are many other optimizing methods such as AdaGrad or RMSProp, which tweak the SGD algorithm to perform even better in some situations. (John Duchi & Singer, 2011) (Tieleman & Hinton, 2012)

When the training process is finished, the DNN can be used to control the rotation speed of a satellite. However, a tiny adjustment needs to be made to the control framework shown in Figure 10. When operating in control mode, the DNN still needs to know what the value of the input scaling factor c is. Also, a sample time Δt needs to be set and communicated with the DNN, so that the input $\vec{x}$ to the network can be calculated properly with Equation 39. Figure 12 shows a block diagram of this setup in a closed-loop.

6 Implementation of the simulation

The following sections focus more on the implementation of all the sections before and on the actual code written in C++. There are two different types of simulation needed. The first type is a simulation used to validate the performance of the controller. Therefore, in this type of simulation a desired rotation speed is set which the controller aims to hold. The second type is a more direct way of simulating satellite movement behavior, where instructions are passed immediately to the wheels. This second type is needed to create reference sample data for the neural network to learn from.

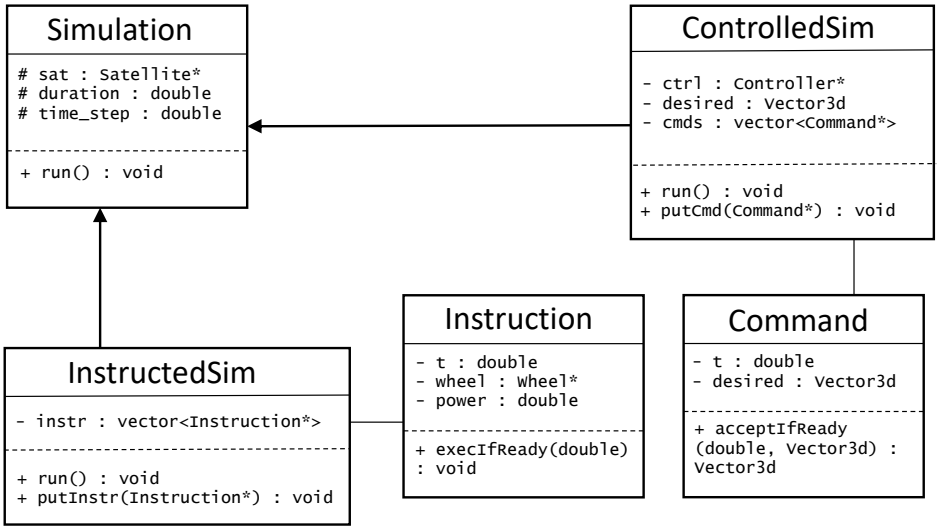


Figure 13: UML diagram of simulation framework

Figure 13 shows a diagram of the structure used to create the two types of simulation. The simulation class holds a satellite object (which will be explained in more detail in subsection 6.3), a duration and a time step variable. Furthermore, it declares the virtual run()-function, which will be overridden by the two different simulation types. The ControlledSim class represents the first type of simulation, used to validate controller performance. It therefore holds a controller object, a current desired 3D rotation speed vector and a list of commands, which holds future desired rotation speeds. The core of the run()-functions implementation in the ControlledSim class looks like this:

```
for(double t = 0.0; t < duration; t += time_step)
{
    //Search for a valid command
    for(int i = 0; i < commands.size(); i++)
        desired = commands.at(i)->acceptIfReady(t, desired);

    //Calculate controller output
    Vector3d present = sat->getRotSpeed();
    out = ctrl->controll(sat, desired, present, t);

    //Feed controller output to wheels
    for(int i = 0; i < sat->getWheels().size(); i++)
        sat->getWheels().at(i)->torqueModeControll(out[i]);

    sat->update(time_step);
}
```

The acceptIfReady()-function of the Command-class is implemented in such a way that if a command is not yet ready at time t, the old desired value gets returned. Otherwise, the new desired vector hold by the command gets returned:

```
Vector3d Command::acceptIfReady(double t, Vector3d oldVal)
{
    if (fabs(this->t - t) < 0.01) return this->desired;
    else return oldVal;
}
```

The InstructedSim class represents the second type of simulation, used to create sample data. It therefore does not need a controller, but only a list of instructions, storing which wheel receives how much power at a certain time. The core of the run()-functions implementation in the InstructedSim class is

```
for(double t = 0.0; t < duration; t += time_step)
{
    //search for a valid instruction, then execute if ready
    for(int i = 0; i < instructions.size(); i++)
        instructions.at(i)->execIfReady(t);

    sat->update(time_step);
}
```

where the execIfReady()-function of the Instruction-class automatically feeds the wheel with the power stored in the instruction object:

```
void Instruction::execIfReady(double t)
{
    if(fabs(this->t - t) < 0.01)
        this->wheel->speedModeControll(power);
}
```

The 0.01 threshold is an arbitrary chosen value. The difference between the speedModeControll()- and the torqueModeControll()-function is explained in subsection 6.2.

6.1 Used libraries

The software uses free, third party C++ libraries. One of which is a fast math library called "Eigen", which ships with an intuitive implementation of vectors and matrices. It also features a solving algorithm for linear systems of equations, which comes in handy for solving Equation 6. The term $I \cdot \vec{L}$ where $I = \Theta^{-1}$ can be considered such a linear system of three equations, which can be solved in Eigen using the following line of code:

```
Vector3d w = I.colPivHouseholderQr().solve(L);
```

Full documentation and many examples of Eigen can be found on their homepage.[1]

Another library is used to construct and train the neural networks, called "MiniDNN". An introducing example on how to use the library can be found on github.[2] The library is built on top of Eigen and provides features like a predefined Network-class, several optimization algorithms, different types of neuron layers and the most common activation functions.

6.2 Reaction wheel implementation

The following parameters are required to construct a wheel:

```
Vector3d initRotAxis; //normalized
double mass, radius; //kg, m
double maxRotSpeed, maxTorque; //rad/s, Nm
```

Some more parameters are needed to keep track of the reaction wheel state:

```
Vector3d rotAxis; //normalized
double rotSpeed, desRotSpeed; //rad/s
double torque, desTorque; //Nm
```

The reaction wheels support two control modes: speed-mode-control and torque-mode-control. The first one is used to hold a desired wheel speed, which finds practice in creating sample data for the neural network. When calling the speedModeControll()-function, the desired wheel speed is set and the control mode of the wheel is set to speed mode. A simple P-controller is used to control wheel speed.

```
void Wheel::controllSpeed(double dt)
{
    //Using a simple P-Controller
    t += dt;
    if (sample_time >= t)
    {
        double err = desRotSpeed - rotSpeed;
        desTorque = Kp * err;
        if (desTorque < -maxTorque) desTorque = -maxTorque;
        else if (desTorque > maxTorque) desTorque = maxTorque;
        t = 0;
    }

    //Controll torque
    controllTorque(dt);
}
```

[1] http://eigen.tuxfamily.org/

[2] http://github.com/yixuan/MiniDNN

The torque-mode-control is not used in the network learning process, but in the control process itself. When calling the torqueModeControll()-function, the desired wheel torque is set and its control mode is set to torque mode. The current torque of the wheel grows linearly towards the desired torque.

```
void Wheel::controllTorque(double dt)
{
    //Linear growth of torque
    if (desTorque > torque)
        torque += maxTorque / T_MAX_TORQUE * dt;
    else if (desTorque < torque)
        torque -= maxTorque / T_MAX_TORQUE * dt;

    //Bounds of torque (max and min (== -max))
    if (torque < -maxTorque) torque = -maxTorque;
    else if (torque > maxTorque) torque = maxTorque;
}
```

The parameter T_MAX_TORQUE corresponds to the time the wheel needs to fully power up from zero to maximum torque and may be manipulated to change torque trajectory slope. Putting things together, the state of the wheel updates in every simulation step using the following function definition

```
void Wheel::update(Vector3d dPhi, double dt)
{
    //Determine controll mode + controll
    switch(ctrl_mode)
    {
        case MODE_TORQUE:
            controllTorque(dt);
            break;
        case MODE_SPEED:
            controllSpeed(dt);
            break;
    }

    //update rotation speed:
    double a = torque / (0.5 * mass * radius * radius);
    rotSpeed += a * dt;

    //rotation speed bounds
    if (rotSpeed > maxRotSpeed) rotSpeed = maxRotSpeed;
    else if (rotSpeed < -maxRotSpeed) rotSpeed = -maxRotSpeed;

    //update orientation of axis:
    rotateVecAround(&rotAxis, dPhi.x(), dPhi.y(), dPhi.z());
}
```

where the vector dPhi is the change of angle in all three dimensions and the

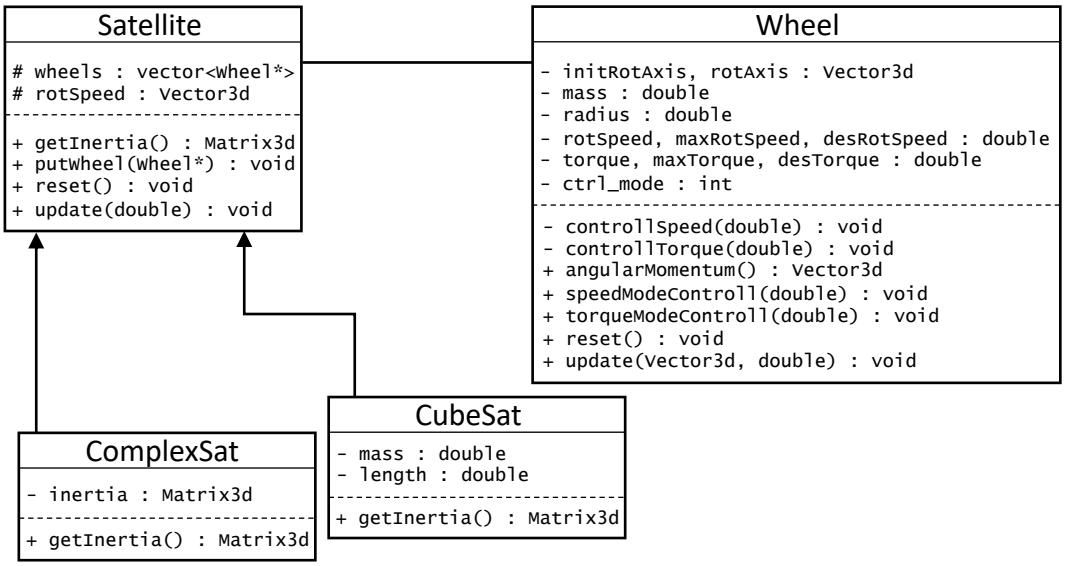


Figure 14: UML diagram of satellite architecture

rotateVecAround()-function is an implementation of Equation 38. Finally, the angularMomentum()-function implements Equation 10:

```
Vector3d Wheel::angularMomentum()
{
    double L_abs = (0.5 * mass * radius * radius) * rotSpeed;
    double L_x = L_abs * rotAxis.x();
    double L_y = L_abs * rotAxis.y();
    double L_z = L_abs * rotAxis.z();
    return Vector3d(L_x, L_y, L_z);
}
```

6.3 Satellite implementation

Figure 14 shows the satellite architecture used in the simulation. The Satellite class holds a list of all wheels, which get updated every simulation step. Using the angular momentum of the wheels and the inertia of the satellite, the rotation speed of the satellite can be calculated every simulation step in the update()-function of the satellite, according to Equation 6. The putWheel()-function can be used to add more wheels to the list of wheels. The reset()-function resets the rotation speed of the satellite, but also the rotational axes and rotation speed of all wheels. This is useful for creating sample data, because this way only one instance of the Satellite class is necessary. The getInertia()-function of the Satellite class is a purely virtual

function. This means that it has to be overridden by using either the ComplexSat or the CubeSat class. When looking at the function defenition of the update()-function

```
void Satellite::update(double dt) {
    t += dt;

    //L is the entire angular momentum of all reaction wheels
    Vector3d L(0, 0, 0);

    //Inertia of the satellite
    Matrix3d I = this->getInertia();

    //sum up all single angular momentums of each reaction wheel
    for(int i = 0; i < wheels.size(); i++)
        L += wheels.at(i)->angularMomentum();

    /*
     *Calculate the rotation speed w of the Satellite
     *L = I * w   ->   w = I^-1 * L
     *that is the same as solving the linear system for w
     */
    Vector3d w = I.colPivHouseholderQr().solve(L);

    //Disturbance
    w = disturb(w, t);

    //Change in the 3D-angle
    Vector3d dPhi = rotSpeed * dt;

    //update the orientation of the reaction wheels
    for(int i = 0; i < wheels.size(); i++)
        wheels.at(i)->update(dPhi, dt);
}
```

a time variable t and a disturb()-function can be spotted, which adds a sine-wave disturbance onto the rotation speed trajectory of the satellite. The t-variable is needed to keep track of the sine-wave disturbance and also gets reset when the reset()-function is called.

```
inline Vector3d disturb(Vector3d w, double t) {
    double s = sin(2 * M_PI * dist_freq * t);
    return w + dist_perc * Vector3d(w.x()*s, w.y()*s, w.z()*s);
}
```

The dist_freq and dist_perc variables correspond to the disturbance frequency (Hz) and the disturbance percentage. They can be manipulated to effect disturbance intensity.

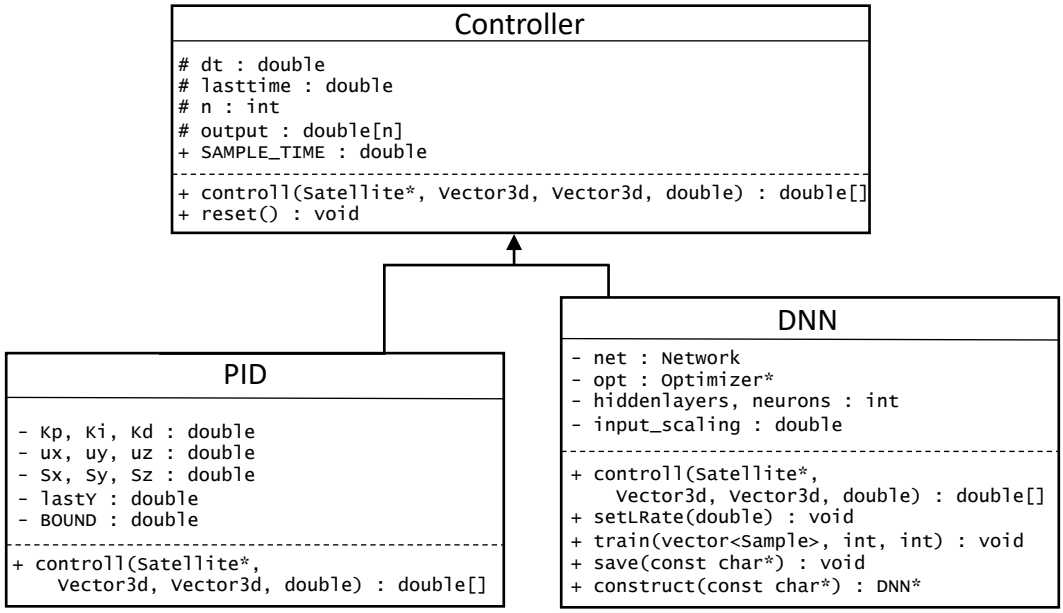


Figure 15: UML diagram of controller architecture

6.4 PID implementation

Figure 15 shows the controller structure. The Controller-superclass provides a virtual controll()-function and some basic variables like the time span dt since the last output was calculated and an output array with the size n which refers to the number of wheels. The PID class has all the variables needed for the algorithm of Equation 20. The output array of size n stores the values that are directly fed to the wheels. The Kp, Ki and Kd variables are the weighting factors of the proportional, integral and derivative term. The ux, uy and uz variables refer to the three dimensional output of the controller and the Sx, Sy and Sz variable are used for integration. The BOUND variable defines the output limit as well as the integral limit, discussed in subsection 3.2. The controll()-function gets overridden by the PID class:

```
double * PID::controll(Satellite* sat,
                       Vector3d des_w,
                       Vector3d w,
                       double t)
{
    dt = t - lastTime;
    if (dt < SAMPLE_TIME) return output;

    else
    {
        //Calc error
```

```
        Vector3d err = des_w - w;

        //Calc integral, watch out for limits, do anti windup
        if (Ki == 0.0) { Sx = 0; Sy = 0; Sz = 0; }
        else
        {
            if (-BOUND < ux && ux < BOUND)
                Sx += Ki * err.x() * dt;
                if (Sx < -BOUND) Sx = -BOUND;
                if(BOUND < Sx) Sx = BOUND;
            if (-BOUND < uy && uy < BOUND)
                Sy += Ki * err.y() * dt;
                if (Sy < -BOUND) Sy = -BOUND;
                if(BOUND < Sy) Sy = BOUND;
            if (-BOUND < uz && uz < BOUND)
                Sz += Ki * err.z() * dt;
                if (Sz < -BOUND) Sz = -BOUND;
                if(BOUND < Sz) Sz = BOUND;
        }

        //Calc "Derivative of Measurement" (lose derivative kick)
        Vector3d dErr = (w - lastY) * (1.0 / dt);

        //Calc control output
        ux = Kp * err.x() + Sx - Kd * dErr.x();
        uy = Kp * err.y() + Sy - Kd * dErr.y();
        uz = Kp * err.z() + Sz - Kd * dErr.z();

        if (ux < -BOUND) ux = -BOUND; if(BOUND < ux) ux = BOUND;
        if (uy < -BOUND) uy = -BOUND; if(BOUND < uy) uy = BOUND;
        if (uz < -BOUND) uz = -BOUND; if(BOUND < uz) uz = BOUND;
        Vector3d u(ux, uy, uz);

        //Dot product to split controller output to the wheels
        for(int i = 0; i < sat->getWheels().size(); i++)
        {
            //Rotation axis of wheel 'r'
            Vector3d r = sat->getWheels().at(i)->getAxis();
            double out = u.dot(r);
            output[i] = out;
        }

        //Remember for next computation
        lastTime = t;
        lastY = w;
        return output;
    }
}
```

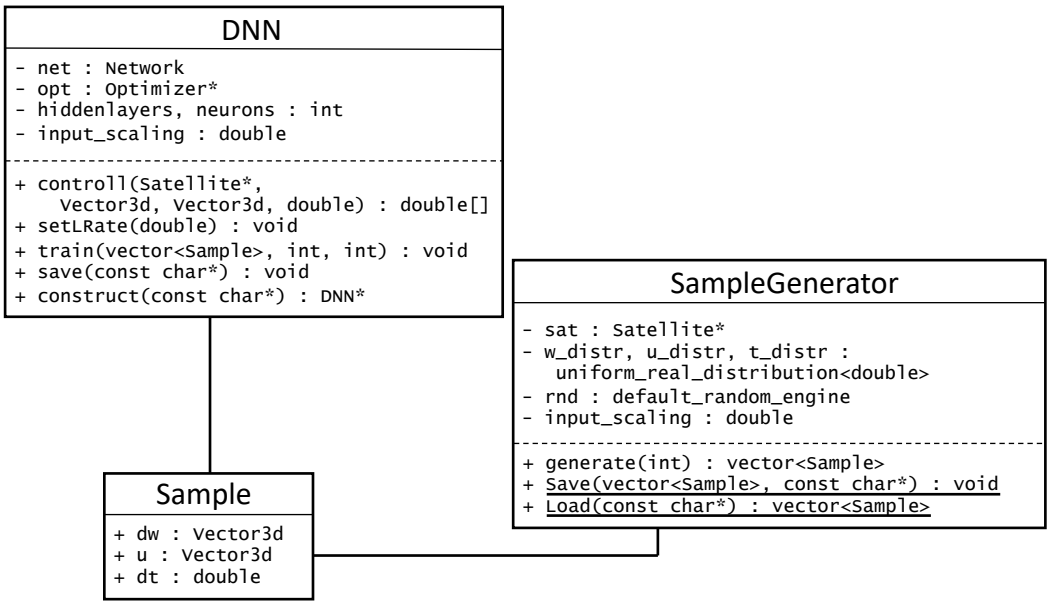


Figure 16: UML diagram of Sample and SampleGenerator classes

6.5 DNN controller implementation

Figure 15 also shows the DNN controller structure. As the figure shows, the DNN class uses a network object and an optimizer, both coming from the MiniDNN library, presented in subsection 6.1. The hidden_layers variable stores the number of hidden layers in the network, whereas the neurons variable stores the number of neurons per hidden layer. The input_scaling parameter is needed to implement Equation 39 and must be the same for the training and control process. Using the setLRate()-function, the hyperparameter α called learning rate of the optimizer stated in Equation 27 and Equation 28 can be changed. The save()- and construct()-functions can be used to save or load an existing DNN once it is trained. In order for the training process to work, the SampleGenerator class, shown in Figure 16, needs to create a set of samples first. A sample consists of the difference in rotation speed dw, the applied control output u and the time span dt it was applied. The SampleGenerator class holds a satellite object, which is used in an InstructedSim object to create random data of the satellite in motion. The continuous uniform distributions w_distr, u_distr and t_distr are used with the default random engine rnd to create random values that fit sensible parameter bounds. For example, the control output should always be a value between negative one and one. However, sample data should not be created fully random. To ensure better learning performance, a fraction of

the data should match particular scenarios that are more likely to encounter in the control process than others. This way, the DNN can have a better understanding of those more likely situations when operating in control mode. The generate()-function can be used to create such a set of samples. About thirty percent of the samples produced this way contain only single axis control outputs. Another 3-4 percent of the data have zero control output on all three axes. The Save()-function is a static function that can be used to save a set of samples in order to load it later, using the also static Load()-function. Once a set of samples was created, the train()-function of the DNN class can be used to start training process. In order to save computation power, the set of samples is divided in smaller batches. The batch-size defines the size of such a mini batch. The benefit in computation power is achieved because the weights and biases of the network get updated not with every sample, but only after one batch. The cost of the network therefore gets averaged over every sample of the mini-batch. The epoch parameter defines how often all of the samples gets processed by the network. In other words: The epoch defines how often a single sample has the opportunity of changing network parameters.

```
void DNN::train(vector<Sample> samples, int batchsize, int epoch)
{
    //Detailed callback
    VerboseCallback call;
    net.set_callback(call);

    //Initialize network
    net.init(0, 0.01, 40);

    //Setup input and output data
    MatrixXd inputs(4, samples.size());
    MatrixXd outputs(3, samples.size());
    Sample s;
    for (int i = 0; i < samples.size(); i++)
    {
        s = samples[i];
        inputs.col(i) << s.t(), s.dwx(), s.dwy(), s.dwz();
        outputs.col(i) << s.ux(), s.uy(), s.uz();
    }

    //Use optimizer to generalize input/output behavior
    net.fit(*opt, inputs, outputs, batchsize, epoch, 40);
}
```

The parameters given to the init()-function of the network object are used to initialize the network with random weights. The first parameter corresponds to the mean of a normal distribution, the second parameter to the

standard deviation of this normal distribution and the third parameter is a seed used by the random generator to create random weight values from that standard deviation. The train()-function uses a VerboseCallback object to give detailed information about the loss to the user.

When constructing a DNN object, a fully connected layer structure is used. Furthermore, the output layer of the network is set to an RegressionMSE object, which calculates the loss of the network using the mean squared error criterion used in Equation 26.

```
//Input Layer (4 inputs: dwx, dwy, dwz, dt)
net.add_layer(new FullyConnected<Identity>(4, neurons));

//Hidden layers
for (int i = 0; i < hiddenLayers; i++)
    net.add_layer(new FullyConnected<Sigmoid>(neurons, neurons));

//Network Output layer (3 outputs: ux, uy, uz)
net.add_layer(new FullyConnected<Identity>(neurons, 3));
net.set_output(new RegressionMSE());
```

The input and output layer both use an identity function $f(x) = x$ as their neuron activation function. The hidden layer neurons use a sigmoid activation function $f(x) = \frac{1}{1+e^{-x}}$, which maps the output of a neuron to values between 0 and 1. The DNN class overrides the control function of the Controller class.

```
double * DNN::controll(Satellite * sat,
                       Vector3d des_w,
                       Vector3d w,
                       double t)
{
    dt = t - lastTime;
    if (dt < SAMPLE_TIME) return output;
    else
    {
        //Calculate the Error
        Vector3d err = des_w - w;

        //Scale the error for network input
        Vector3d scaled_w = input_scaling * err / dt;

        //Feed input to the net, gather output
        MatrixXd input(4, 1);
        input.col(0) << dt,
                        scaled_w.x(),
                        scaled_w.y(),
                        scaled_w.z();
        Vector3d u = net.predict(input);
```

```
        //Split the control output on the wheels
        for(int i = 0; i < sat->getWheels().size(); i++) {
            //Rotation axis of wheel 'r'
            Vector3d r = sat->getWheels().at(i)->getAxis();
            double out =  u.dot(r);
            output[i] = out;
        }

        //Remember for next computation
        lastTime = t;

        return output;
    }
}
```

6.6 User manual

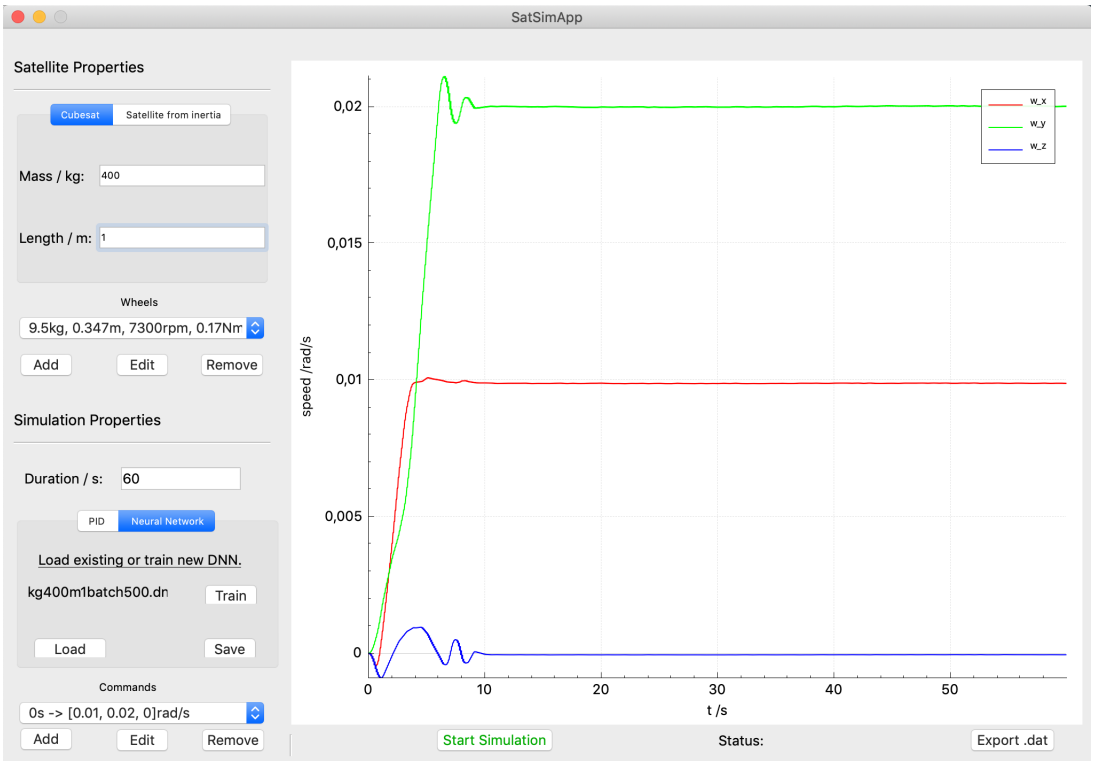


Figure 17: Screenshot of the main dialog

At this point, everything needed to create and run a simulation is set up. Figure 17 shows a screenshot of the graphical user interface of the applica-

tion. The left quarter of the GUI is filled with dialogues that define satellite and simulation properties. The rest of the GUI mainly consist of a plane used for plotting the step response graphs. The following parameters can be manipulated to define any satellite plus reaction wheel setup:

- Inertia tensor Θ of the satellite
- Mass m_w of every reaction wheel
- Radius r_w of every reaction wheel
- Maximum speed $\omega_{w,max}$ of every wheel
- Maximum torque $M_{w,max}$ of every wheel
- (Normalized) rotation axis $\vec{r}_w$ of every wheel

Given those parameters, an approximation of the maximum rotation speed $\vec{\omega}_{max}$ that a satellite is able to reach can be calculated using Equation 12.

$$\vec{\omega}_{max} = \Theta^{-1} \cdot \sum_{w} (\frac{1}{2} m_w \cdot \omega_{w,max} \cdot r_w^2 \cdot \vec{r}_w) \tag{40}$$

When giving commands to the controller, the rotation speed limit should be considered. Setting a rotation speed higher than that value can cause the controller to behave unnaturally.

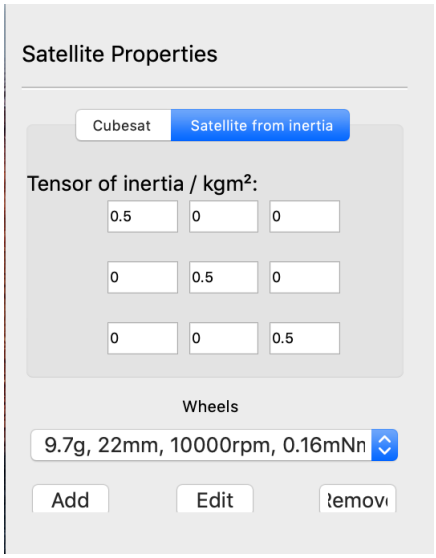


Figure 18: Screenshot of satellite properties dialog

The mentioned parameters can be manipulated using the dialog shown in Figure 18. The inertia of a satellite can be set directly using the "Satellite from inertia" option, or using the "Cubesat" option, where a side-length and a mass has to be given to calculate the inertia. Wheels can be added, removed or their properties can be changed using the "edit" button. Figure 19 shows

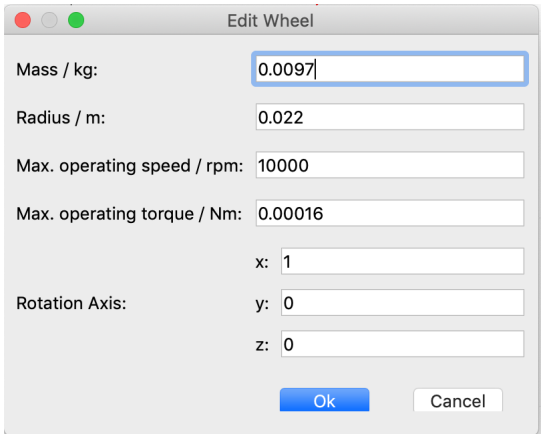


Figure 19: Screenshot of wheel properties dialog

the dialog that is used when defining wheel properties through adding or editing a wheel.

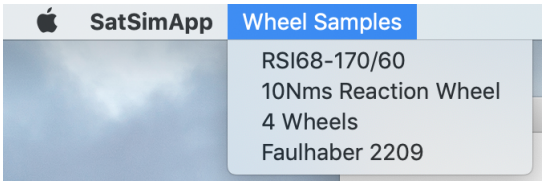


Figure 20: Screenshot of wheel samples in menu bar

Figure 20 shows that there are also predefined wheels which can be found in the menu bar. When one of those predefined wheel setups is chosen, all the existing wheels get overridden by the new wheels.

Once the setup of satellite plus wheels is defined, a controller has to be chosen that processes the given commands. The controller can be chosen by switching between the "PID" and the "Neural Network" option. Command parameters can be manipulated using the "add", "edit", and "delete" button, which can be found in the bottom left corner of the GUI. Furthermore, the duration of the simulation has to be set.

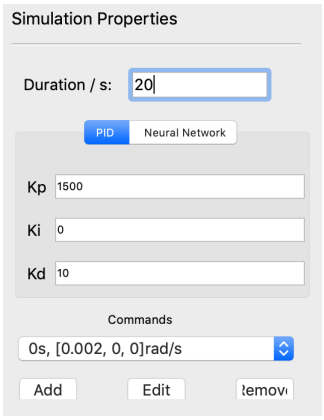


Figure 21: Screenshot of the simulation properties dialog

When choosing the PID controller, the essential parameters can be entered directly into the dialog that is shown in Figure 21. When adding or editing a command, a dialog shows where a desired rotation speed and the corresponding control time can be entered.

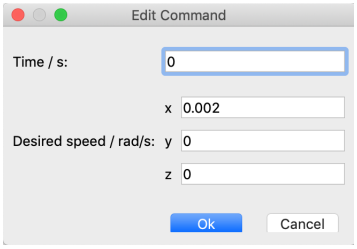


Figure 22: Screenshot of command dialog

Figure 22 shows that dialog. When entering speed values into the dialog, speed saturation defined by Equation 40 should be considered.

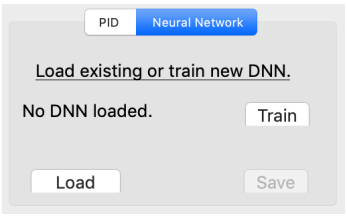


Figure 23: Screenshot of the DNN dialog

Figure 23 shows the deep neural network controller dialog. From here, a new DNN can be trained using the "Train" button, or an existing DNN can be saved or loaded from the hard disk.

Creating a neural network controller requires setting some more parameters for the training process:

- Number of samples
- Batchsize
- Number of epoches
- Number of hidden layers
- Number of neurons per hidden layer
- Optimizer (SGD, AdaGrad, RMSProp)
- Learning rate α
- Input scaling factor c

These parameters must be set when hitting the "Train" button in the "Neural Network" dialog.

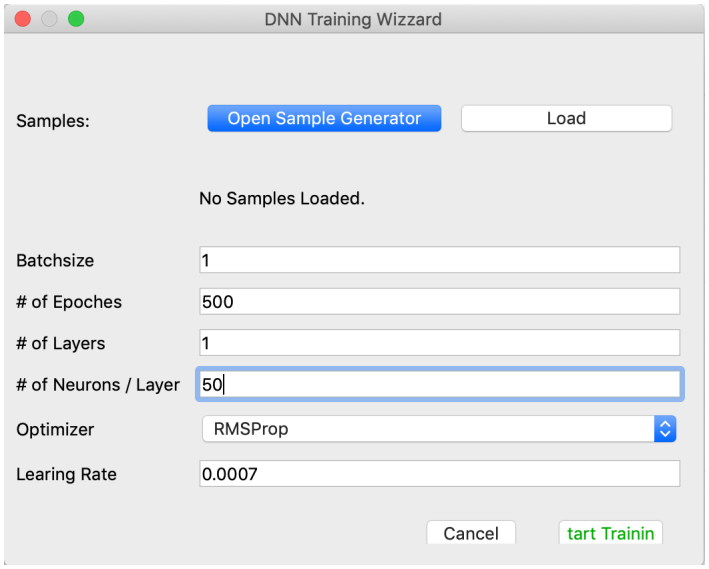


Figure 24: Screenshot of DNN training parameter dialog

Figure 24 shows the DNN training parameter dialog. At the top of the dialog new samples can be created, or existing samples can be loaded from the hard disk.

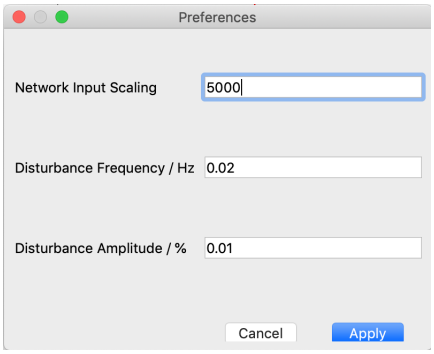


Figure 25: Screenshot of the preferences dialog

The input scaling factor can be also manipulated in the preferences dialog, which can be found in the menu bar. Figure 25 shows the preferences dialog, where not only the input scaling factor, but also the frequency and the amplitude of the sine wave disturbance can be changed.

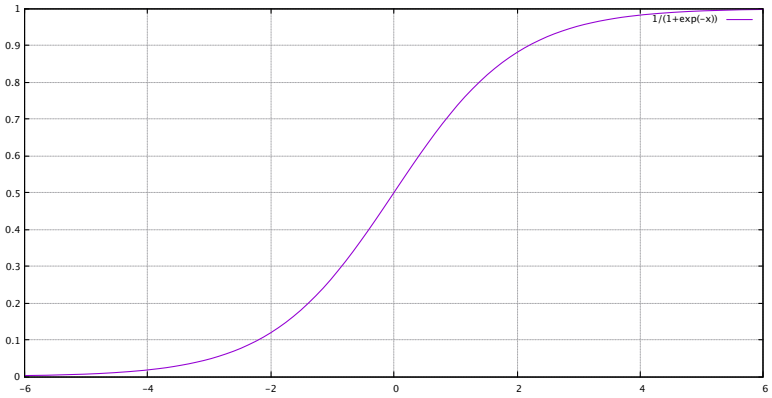


Figure 26: Sigmoid function $S(x) = \frac{1}{1+e^{-x}}$ of any hidden layer neuron

The input scaling factor should be chosen in such a way that all the samples contain inputs with numerical values between -6 and 6. Figure 26 shows that beyond those values the output of a neuron will be approximately just 0 or 1 respectively, which means that the training performance will be worse because the network needs to use neurons for up-scaling in order to differentiate between values very close to each other.

Finding optimal values for the other variables is not a trivial task. However, subsection 7.3 shows some connections between training performance and different training parameters. Detailed information about the installation of the software on any platform can be found on the github page.[3]

[3] `http://github.com/fallow24/neuro-sat`

7 Simulation results

In this chapter, the implemented simulation is used to test the performance of the two different controller (PID and DNN) archetypes. The examples used in the simulation were picked according to existing reaction wheel data sheets. Also, both the orthogonal setup (subsection 7.1) of three wheels and the pyramid setup (subsection 7.2) of four wheels are quite popular in the space industry.

7.1 Three reaction wheels orthogonal setup

This setup uses a cubesat with mass $m = 10\,\text{kg}$ and side length $d = 0.1\,\text{m}$, resulting in an inertia tensor of

$$\Theta = 0.167\,\text{kg}\,\text{m}^2 \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{41}$$

according to Equation 4. Three "Faulhaber 2209" wheels are attached to the satellite in an orthogonal setup with initial rotational axes of $\vec{r}_1 = (1, 0, 0)^\tau$, $\vec{r}_2 = (0, 1, 0)^\tau$ and $\vec{r}_3 = (0, 0, 1)^\tau$. The rest of the wheels specifications were taken from the "Thin Profile Brushless Motors with integrated Speed Controller and 12 Bit Encoder/Interface" (2008) data sheet:

- $m_w = 0.0097\,\text{kg}$
- $r_w = 0.022\,\text{m}$
- $\omega_{w,max} = 10\,000\,\text{rpm} (= 1047.2\,\text{rad/s})$
- $M_w = 0.000\,16\,\text{N}\,\text{m}$

According to Equation 40, the maximum speed of the whole satellite that can be achieved with only the torques created by the wheels is
$\vec{\omega}_{max} = 0.0147\,\text{rad/s} \cdot (1, 1, 1)^\tau$.

7.1.1 PID step responses

The PID values for this setup were adjusted by hand. The controller operates with $K_p = 150$, $K_i = 0.5$, and $K_d = 0.1$. Depending on the asked requirements, those values could be changed to make the controller perform even better in certain situations. However, for step response testing, these values result in good stability and fast reaction time. The controller must speed up to $\vec{\omega}_{des,1} = (0.1, 0, 0)^\tau rad/s$, then $\vec{\omega}_{des,2} = (0.1, -0.08, 0)^\tau rad/s$, and finally $\vec{\omega}_{des,3} = (0.1, -0.08, -0.02)^\tau rad/s$ before slowing down to no rotation at all.

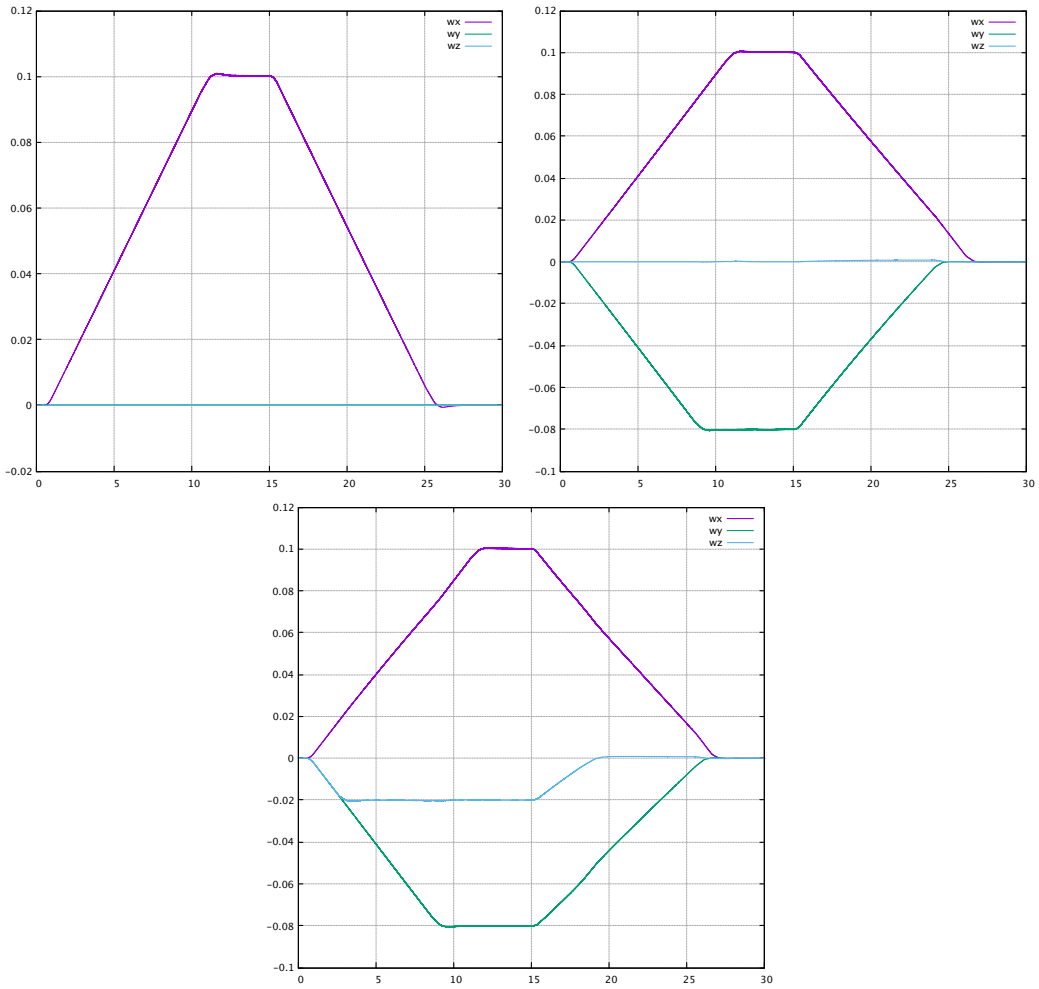


Figure 27: PID step responses with $K_p = 150$, $K_i = 0.5$, and $K_d = 0.1$ on an orthogonal setup satellite. x-axis is time in seconds, y-axis is speed in rad/s

Figure 27 shows the step responses produced by the PID controller on single, double and all three axes.

7.1.2 DNN step responses

The training parameters used for the DNN training process were found through trial and error. The network was trained with 1000 samples, 250 batch-size, 5000 epoches, 2 hidden layers with 50 neurons each, AdaGrad optimization, learning rate of $\alpha = 0.0007$ and an input scaling factor of $c = 5000$. For comparison, the DNN controller must speed up to the same reference values as the PID controller does in subsubsection 7.1.1.

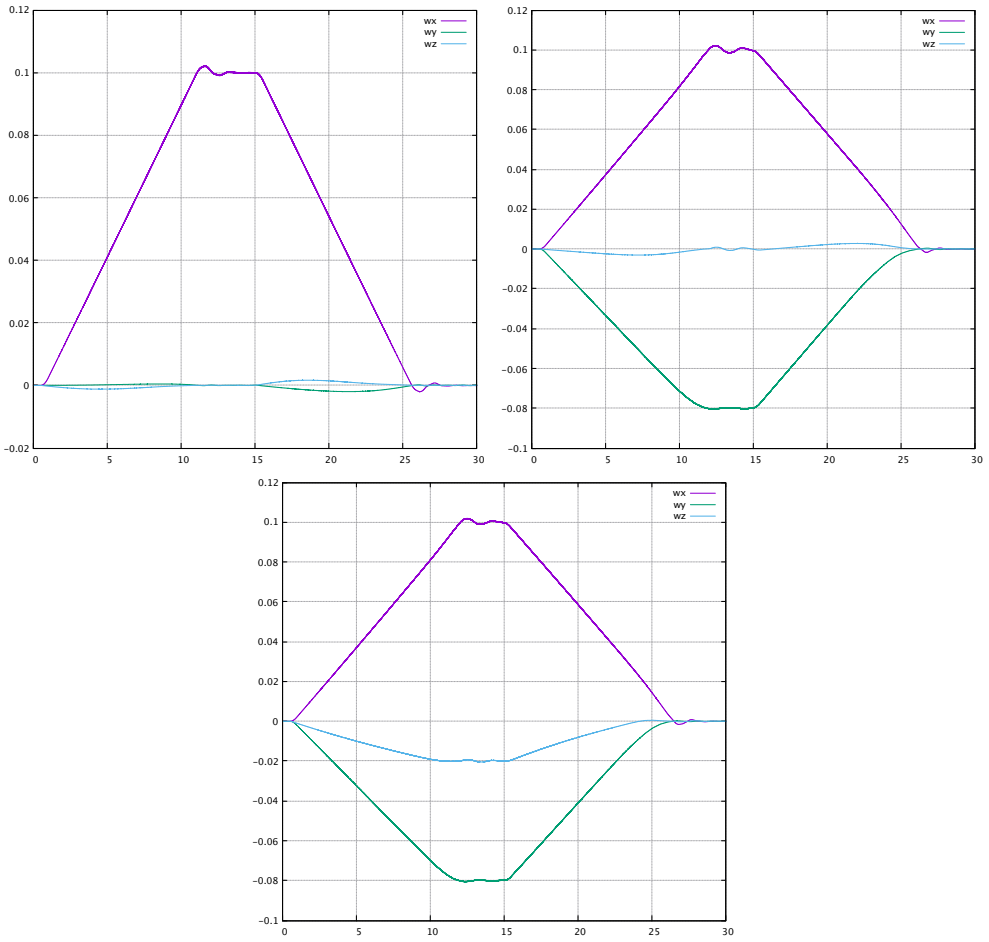


Figure 28: DNN step responses on an orthogonal setup satellite with 2 hidden layers, 50 neurons per layer, 250 batchsize, 5000 epoches, 0.0007 learning rate and 1000 samples. x-axis is time in seconds, y-axis is speed in rad/s

Figure 28 shows the step responses produced by the DNN controller on single, double and all three axes.

7.2 4 reaction wheels pyramid setup

The four-wheel-pyramid-setup is a more complex type of situation to control not only because the wheels are no longer orthogonal, but also because small, arbitrary chosen momenta of deviation are also included as suggested in section 2.

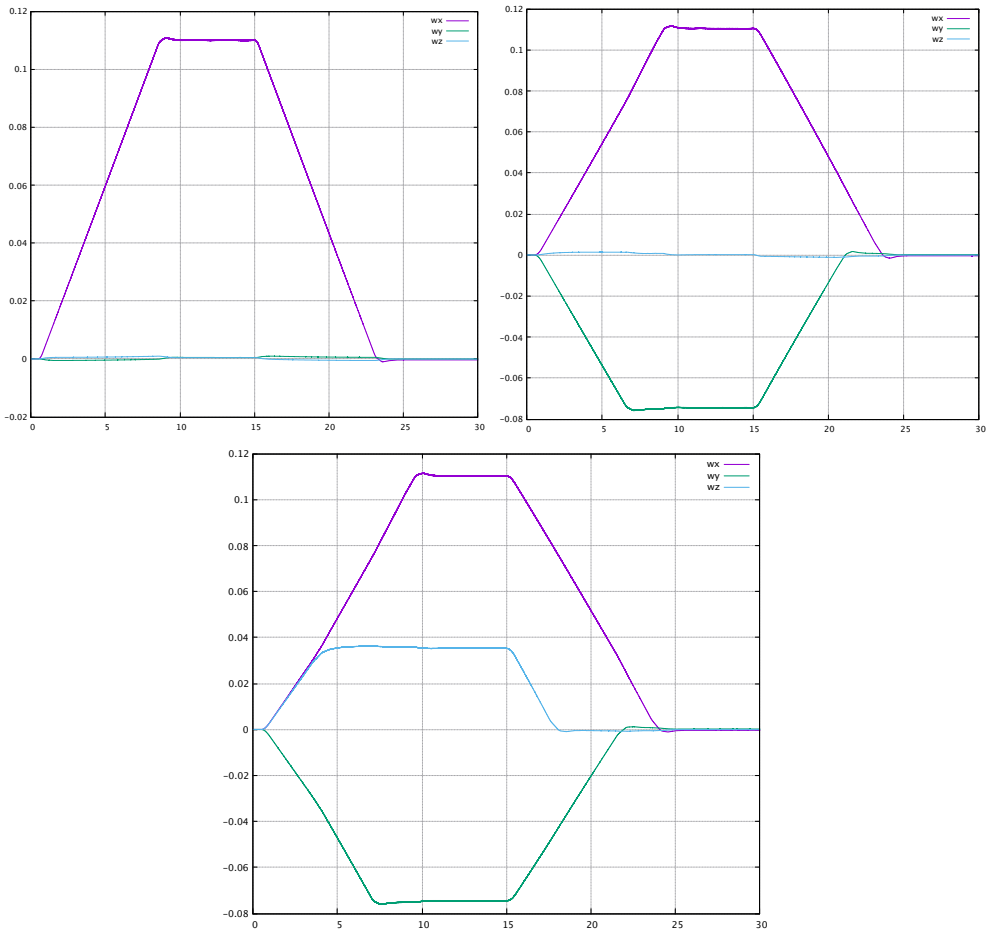


Figure 29: PID step responses with $K_p = 180$, $K_i = 0.8$, and $K_d = 0.5$ on an pyramid setup satellite. x-axis is time in seconds, y-axis is speed in rad/s

The inertia of the satellite used in this simulation is

$$\Theta = \begin{pmatrix} 16.66 & 1.13 & -1.04 \\ 1.13 & 16.67 & 0.52 \\ -1.04 & 0.52 & 16.65 \end{pmatrix} \text{kg m}^2 \tag{42}$$

The structure of this satellite can be imagined as an approximate cube with mass of about 100kg and side length of 1m, but with some irregularities in shape or weight distribution. The initial rotation axes of the wheels are $\vec{r}_1 = \frac{1}{\sqrt{3}}(1,1,1)^\tau$, $\vec{r}_2 = \frac{1}{\sqrt{3}}(-1,1,1)^\tau$, $\vec{r}_3 = \frac{1}{\sqrt{3}}(1,-1,1)^\tau$, and $\vec{r}_4 = \frac{1}{\sqrt{3}}(-1,-1,1)^\tau$, which correspond to the edges of a four-sided pyramid, represented by unit vectors. The parameters of the wheels used come from the RSI68-170/60 data sheet. (*High Motor Torque Momentum and Reaction Wheels*, 2019).

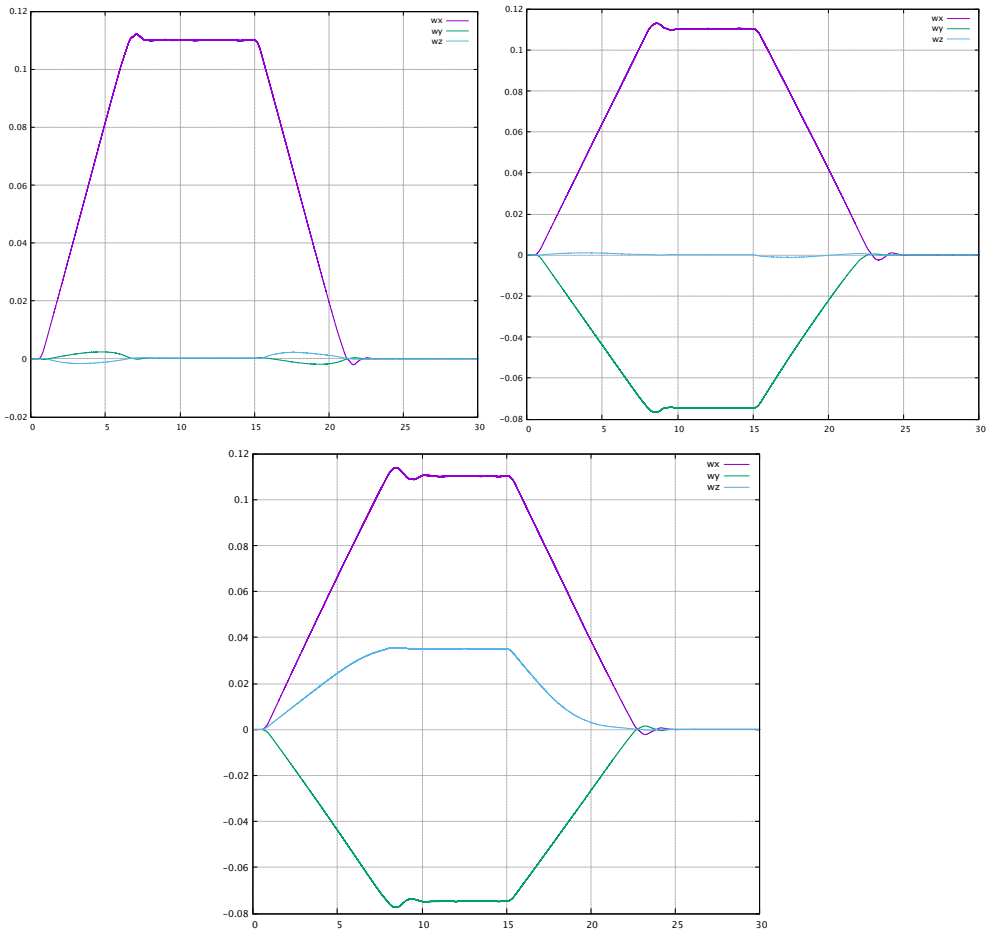


Figure 30: DNN step responses on a pyramid setup satellite with 1 hidden layer containing 50 neurons, 2500 batchsize, 500 epoches, 0.0004 learning rate and 10000 samples. x-axis is time in seconds, y-axis is speed in rad/s

- $m_w = 9.5\,\mathrm{kg}$
- $r_w = 0.347\,\mathrm{m}$
- $\omega_{w,max} = 7300\,\mathrm{rpm} (= 764.5\,\mathrm{rad/s})$
- $M_w = 0.17\,\mathrm{N\,m}$

7.2.1 PID step responses

As in subsubsection 7.1.1, the K_p, K_i and K_d values used by the PID controller were adjusted by hand. The values are $K_p = 120$, $K_i = 0.8$,

and $K_d = 0.5$. The set-point reference values given to the controller are $\vec{\omega}_{des,1} = (0.11, 0, 0)^\tau rad/s$, $\vec{\omega}_{des,2} = (0.11, -0.075, 0)^\tau rad/s$, and $\vec{\omega}_{des,3} = (0.11, -0.075, 0.035)^\tau rad/s$. At $t = 15\,\text{s}$, the reference signal becomes $\vec{\omega}_0 = (0, 0, 0)^\tau rad/s$. Figure 29 shows the step responses produced by the PID controller on single, double and all three axes.

7.2.2 DNN step responses

The training parameters for the DNN learning process were found through trial and error. The network was trained using 10000 samples, 2500 batch-size, 500 epochs, 1 hidden layer with 50 neurons, RMSProp optimization, learning rate of $\alpha = 0.0004$ and an input scaling factor of $c = 5000$. The set-point reference signals used by the DNN controller are the same as in subsubsection 7.2.1. Figure 30 shows the step responses produced by the DNN controller on single, double and all three axes.

7.3 Variation of DNN learning parameters

The purpose of varying learning parameters is to get an idea of how the performance of the neural network correlates with certain training variables in a unique situation. However, some parameters will not be considered in the next chapters, such as the number of neurons or hidden layers, because changes in those parameters lead to a predictable outcome. Adding more neurons or layers makes the network more complex. Therefore, the network will be able to generalize more complex system behaviors, but it will also require more computation power and a longer time to train. That is also the case for sample size. Generally, more samples only improve network training quality to the cost of longer training time. Finding good values for those variables depends on the trade-off you can afford.

There are yet some parameters left that can be tuned to tweak training performance, such as batch-size, epochs, and learning rate. The effects of tweaking these are not as predictable as the effects of tweaking the trade-off parameters. Furthermore, their behavior changes for every setup of satellite plus wheels, so a lot of trial and error is required to get the network to fit perfectly. In the following three chapters all but one of those variables are constant values. These constant values come from the good fit networks, which subsubsection 7.1.2 and subsubsection 7.2.2 show. The value of the non-constant variable is then replaced with something bigger or smaller to see the effect on training performance.

The setup used in subsubsection 7.3.1, subsubsection 7.3.2, and subsubsection 7.3.3 is the same setup used in subsection 7.2: A four-sided pyramid

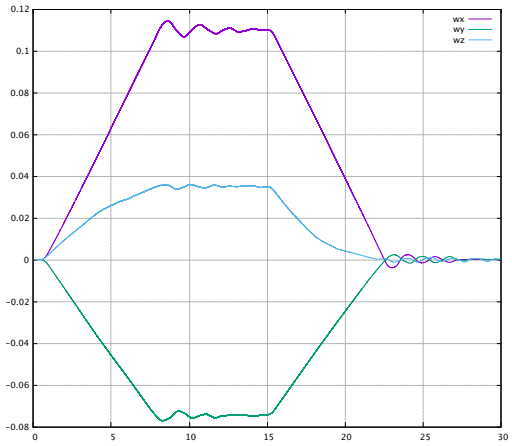


Figure 31: Step response of the same DNN controller used in Figure 30, but with batch-size 1

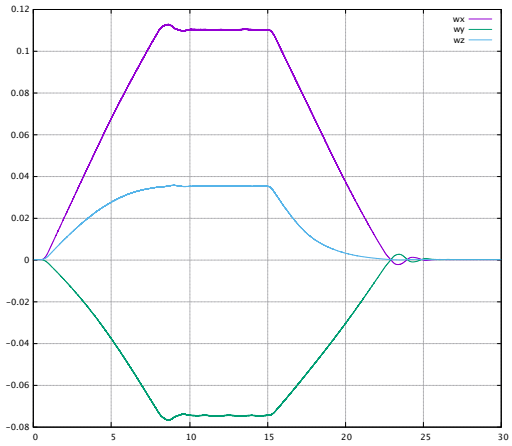


Figure 32: Step response of the same DNN controller used in Figure 30, but with batch-size 100000

setup of wheels and a moment of inertia tensor described by Equation 42. The trained network must then perform the same step response as as in subsection 7.2. Therefore, the bottom picture of Figure 30 is a best case reference for the results of the next sections. However, there is no guaranty that it actually is the best case since the values for the training parameters were found through trial and error.

7.3.1 Batch-size comparison

Figure 31 and Figure 32 show the step responses of two different networks, one of which was trained with a batch-size of 1, the other one with a batch-size of 100000. The rest of the training parameters stay the same for both networks: 1 hidden layer containing 50 neurons, 500 epochs, 0.0004 learning rate and 10000 samples.

For this setup and when every other training parameter stays constant, a smaller batch-size seems to correlate with larger oscillations in the step response whereas a larger batch-size seems to correlate with less oscillation.

7.3.2 Epochs comparison

Since the number of epochs directly affects the training duration, more epochs mean more training and therefore, a better network performance. However, in some cases the opposite is true. At a certain point, too much training decreases the network performance due to over-fitting. The term ”over-fitting” means that the network performs almost perfectly when given a sample from

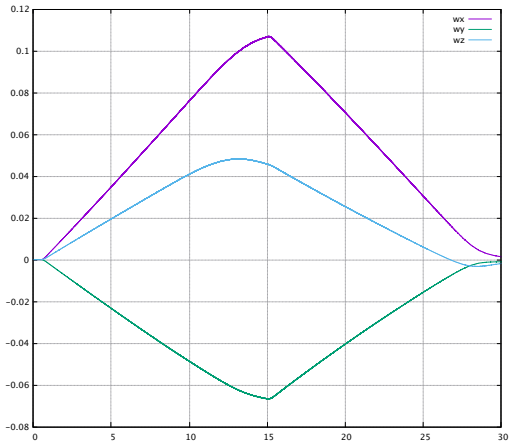


Figure 33: Step response of the same DNN controller used in Figure 30, but with 10 epochs

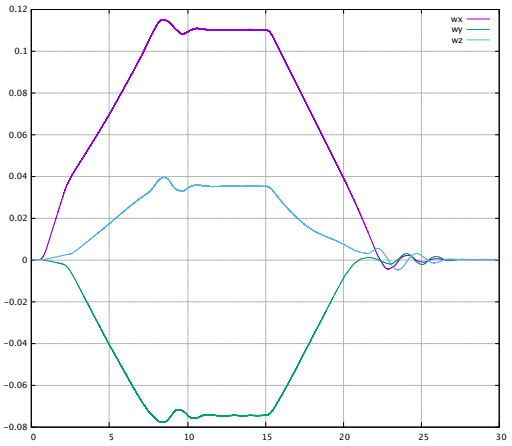


Figure 34: Step response of the same DNN controller used in Figure 30, but with 40000 epochs

the training data set, but struggles when given a sample that it has never processed before. This happens because the network memorizes all the training data but was not able to accurately generalize the relationship between the given system variables. Over-fitting does not only happen because of too much training, it can also happen for many other reasons like giving the network additional, yet unnecessary input parameters. (Mitchell, 1997).

Figure 33 and Figure 34 show the step response of two differently trained networks. Figure 33 shows a DNN that was trained with only 10 epochs. Figure 34 shows a DNN that was trained with 40000 epochs. The rest of training parameters stay the same for both networks: 1 hidden layer containing 50 neurons, 2500 batch-size, 0.0004 learning rate and 10000 samples.

For this setup and if every other training parameter stays constant, more epochs seem to lead to over-fitting, whereas a small number of epochs means that the network does not get enough training iterations to decrease its cost.

7.3.3 Learning rate comparison

Finally, the last parameter to consider is the learning rate α. Just as with epochs, a higher learning rate increases training convergence, because according to Equation 27 and Equation 28 the steps taken in the direction of the cost gradient get bigger. So, in theory, a higher learning rate should increase network performance. However, if the number of epochs does not decrease, a learning rate that is too high can also lead to over-fitting. (Mitchell, 1997)

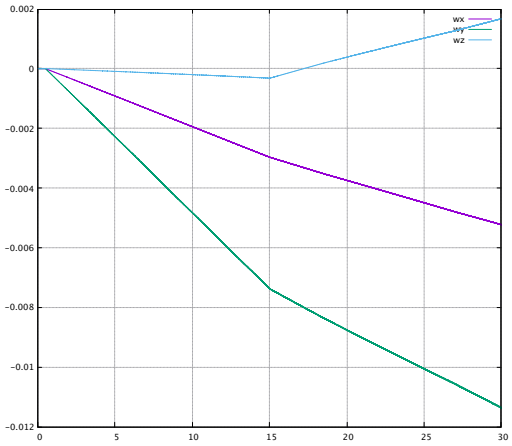


Figure 35: Step response of the same DNN controller used in Figure 30, but with $\alpha = 0.000000001$

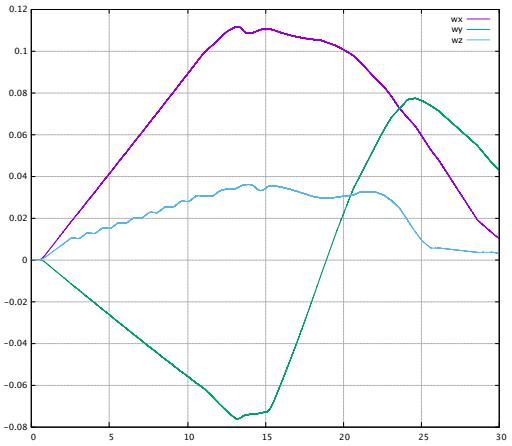


Figure 36: Step response of the same DNN controller used in Figure 30, but with $\alpha = 0.05$

Figure 35 and Figure 36 show the step responses of two different networks. The first one was trained with a small learning rate of $\alpha = 0.000000001$, the other one with a bigger learning rate of $\alpha = 0.05$. The rest of the training parameters stay the same for both networks: 1 hidden layer containing 50 neurons, 2500 batchsize, 500 epochs and a sample size of 10000.

Figure 35 shows that decreasing the learning rate too much while holding constant all other training parameters prevents training convergence. On the contrary, Figure 36 shows that increasing the learning rate leads to a worse network performance due to over-fitting.

However, there is not a perfect learning rate that can be used for any network. Instead, the optimal learning rate depends on the other training parameters and must be found again for every setup.

7.4 Quality comparison between PID and DNN controller

First, the performance of the two controllers used on the 3-axes orthogonal wheels setup should be compared on single, double and all three axes. Figure 27 shows the PID step responses, which are smoother than any of the DNN step responses shown in Figure 28. On single axis, the PID controller manages to reach its target with a small overshoot, then holds the desired value without oscillation. The DNN reaches its single axis target with a bigger overshoot than the PID. It is also able to hold the single axis target without oscillation, but fails to hold the other two rotation speeds down to

zero when the rotation speed is changing on one axis. The same things are true for double axis control. Furthermore, on double axis, the PID controller reaches the desired rotation speed faster than the DNN does. All these effects combine for three axes control, making the PID controller perform better in every aspect for this setup.

Then the performance of the two controllers used on the 4 wheels pyramid setup should be compared on single, double and three axes. Again, the PID step responses shown in Figure 29 are smoother than the DNN step responses shown in Figure 30. On single axis, the PID controller reaches its target with a smaller overshoot than the DNN controller does. The oscillations on the other two axes while the rotation speed increases on one axis are bigger for the DNN. When operating on multiple axes, the PID controller reaches its target faster than the DNN controller. What this all amounts to is that the PID controller performs better than the DNN controller in every aspect for this setup.

8 Conclusion

As subsection 7.4 shows, the commonly used PID controller produces better results than the implementation of the DNN controller in terms of overshooting, oscillation, and general stability. However, the PID controller does not perform remarkably better than the DNN controller, but the difference between the two is rather shallow.

Before bringing the neural network onto a real satellite, more testing with a more accurate system model is required. The simulation used in this thesis ignores utter influences like atmospheric pressure, solar pressure, and other disturbances like gravity or the earths magnetic field. Furthermore, the torque model for the reaction wheels is linearly simplified.

The PID controller is easier to tweak and does not need any exemplary training data, which are straightforward to create in a simulation environment but difficult to collect in the real world. However, as a proof of concept, section 7 shows that substituting a closed-loop PID controller with a deep neural network works beyond doubt in a simulation.

References

Ang, K., Chong, G., & Li, Y. (2005). Pid control system analysis, design, and technology. *IEEE Transactions on Control Systems Technology*.

Atherton, D. P. (2014). Almost six decades in control engineering [historical perspectives]. *IEEE Control Systems Magazine*.

Beauregard, B. (2011, 4). *Improving the beginner's pid.* Retrieved 9.7.2019, from `http://brettbeauregard.com/blog/2011/04/improving-the-beginners-pid-introduction/`

Carrara, V., & Rios Neto, A. (1999). A neural network satellite attitude controller with error based reference trajectory. *XIV International Symposium on Space Flight Dynamics*.

Goldstein, H. (1981). *Klassische mechanik*. Akademische Verlagsgesellschaft.

Hagan, M. T., Demuth, H. B., & Jesús, O. D. (2002). An introduction to the use of neural networks in control systems. *International Journal of Robust and Nonlinear Control*.

High motor torque momentum and reaction wheels. (2019). Retrieved 09.08.2019, from `https://www.rockwellcollins.com/Products-and-Services/Defense/Platforms/Space/High-Motor-Torque-Momentum-and-Reaction-Wheels.aspx`

John Duchi, E. H., & Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research 12*.

King, M. (2010). *Process control: A practical approach*. UK: John Wiley and Sons Ltd.

Meschede, D., & Gerthsen, C. (2005). *Gerthsen physik*. Springer Spektrum.

Mitchell, T. M. (1997). *Machine learning*. McGraw-Hill Companies, Inc.

NASA. (n.d.). *Reaction/momentum wheel.* Retrieved 3.7.2019, from `https://spinoff.nasa.gov/spinoff1997/t3.html`

Sanderson, G. (2017, 11). *Backpropagation calculus.* Retrieved 26.07.2019, from `https://www.youtube.com/watch?v=tIeHLnjs5U8`

Serway, R. A. (1986). *Physics for scientists and engineers*. Saunders College Publishing.

Thin profile brushless motors with integrated speed controller and 12 bit encoder/interface [Computer software manual]. (2008).

Tieleman, T., & Hinton, G. (2012). *Divide the gradient by a running average of its recent magnitude.* Retrieved 31.07.2019, from `COURSERA: NeuralNetworksforMachineLearning,Lecture6.5`