Accurate white-box simulation model of the Crazyflie dynamics with the possibility of sim-to-real transfer

Pieter-Jan van Dolderen, Fabian Gebben, Ewout Schokker and Christiaan Theunisse

***Abstract*— To be able to quickly and cheaply develop complex quadcopter controllers, a good simulation model of the quadcopter is needed. Previous quadcopter simulation models had limitations such as being black-box and being hard to switch between simulation and reality. We have developed a white-box simulation model for a quadcopter, namely the Crazyflie 2.0. The implementation of the simulation model in EAGERx allows us to easily switch between simulation and reality. We validated our model with an accurate black-box model by flying different trajectories in 2D. The highest mean errors for all the trajectories are 0.172 m in x-dimension, 0.0468 m in z-dimension and 6.84 deg for the pitch. Our white-box model is developed for the Crazyflie, but it could also be used as an inspiration for others simulating quadcopters similar to the Crazyflie.**

***Keywords*— Quadcopter, Dynamics, Modeling, Parameter Estimation, Simulation, Machine Learning, Python, Crazyflie 2.0, EAGERx**

I. Introduction

Quadcopters have seen a tremendous rise in popularity, because of their relatively low cost and their many different applications in e.g. agriculture, insurance and construction. Due to this rise in popularity, the demand for more complex quadcopter control has also increased. To be able to develop these more complex controllers as quickly and cheaply as possible a good simulation model of the quadcopter is needed. With an accurate quadcopter simulation, reinforcement learning can be used to learn control policies for complex tasks. The user should be able to switch between using the controller on the real-life and simulated quadcopter seamlessly to verify these control policies.

A lot of quadcopter simulation models have already been developed for different purposes like platooning [7], landing on an inclined surface [5] or for understanding the quadcopter's behaviour [8]. However, none of these simulation models could be used to train control policies and then easily use these policies to control the Crazyflie in the real world.

[5] made an ODE[1] model of the Crazyflie (a small developer quadcopter) dynamics. However, it has some limitations: it is only useful within an angle range of 30 degrees, hard to switch between simulation and reality and it is a partly black-box model. A black-box model provides no insight into the quadcopter's internal workings. Changing some parameters of the real-life quadcopter, like mass or a gain constant for example, renders the black-box model invalid.

To counter this problem, this paper will focus on implementing a white-box simulation model of a Crazyflie 2.0 quadcopter. The white-box simulation model will be based on the Crazyflie firmware. It will be transparent and explicitly model the internal control loops and flight dynamics. Consequently, changing some parameters of the quadcopter does not render the simulation model invalid, because these changes can also be modified in the white-box model itself. Since the white-box model will be based on physics and not on experiments, the model may also be valid outside the angle range of 30 degrees

The white-box model will be implemented inside the EAGERx framework [9]. The implementation in EAGERx makes it possible to easily switch between the simulation and reality, because it splits the simulator into two parts: an engine-specific part and an agnostic part. An engine will in this paper be used to indicate a piece of software that approximates the movement of the quadcopter based on the applied forces.

The engine-specific part consists of three different objects with all the simulator-related components. The three objects in our engine-specific part are: (1) a partly black-box model with an ODE model engine based on [5], (2) a white-box model with a physics engine and finally (3) the real world (Fig. 1). All the objects have the same in- and outputs. This allows the user to easily switch between the three different objects.

The agnostic part encapsulates all other components, like the control policies that have to be trained. The outputs of the agnostic part are connected to the inputs of the engine-specific part. This makes it easy to use the trained control policies on the real world quadcopter by changing the object to the real Crazyflie.

In summary, our contributions are:

- We present a white-box Crazyflie 2.0 model as an object in the EAGERx framework[2] by combining parts of different existing Crazyflie simulation models and the Crazyflie firmware. This model can be used for simulating the behaviour of the real Crazyflie. The implementation in EAGERx allows the simulation trained control policies to be easily applied on the real Crazyflie. The model can also be used as an inspiration for modelling other quadcopters inside of EAGERx.
- We implement the partly black-box ODE model for the Crazyflie's dynamics from [5] as an object in the EAGERx framework, which can be used for validation of or comparison with other objects.

[1] Ordinary Differential Equations

[2] https://github.com/PietDol/Crazyflie_Simulation

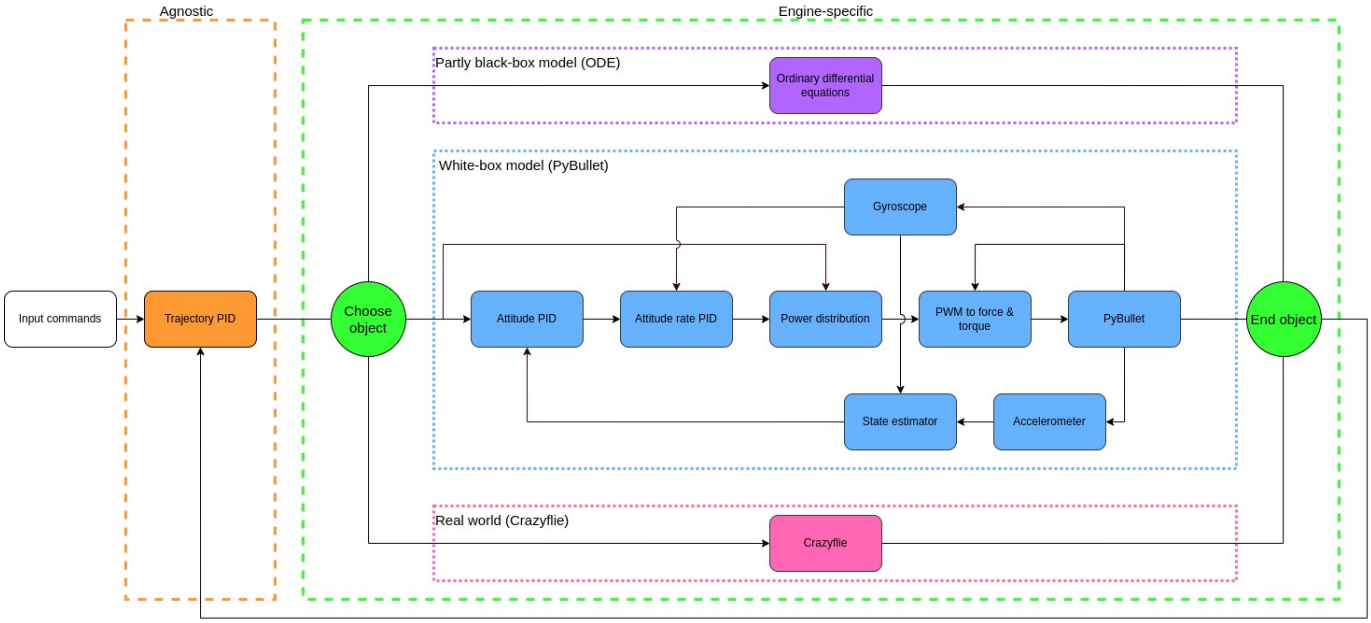


Fig. 1: Block diagram showing the structure of our simulation model.

- We test our white-box Crazyflie model's accuracy with the ODE Crazyflie model by comparing various closed-loop trajectories.

The remainder of the paper is structured as follows. Section II will briefly introduce related work. In section III, the developed simulation framework will be discussed in detail. This simulation will then be validated, which will be explained and shown in section IV. Finally, in section V, the conclusion will be given, and some possible future work will be addressed.

II. Related work

Various different quadcopter simulators have already been developed. A simulation was developed to help understand the behaviour of the flight control system of the Crazyflie using the Gazebo physics engine [8]. Another simulator also contained reinforcement learning interfaces. This simulator also modelled collisions and aerodynamic effects using Pybullet as a physics engine [7]. It also contained a lot of flight examples, like PID trajectory tracking. The last related simulation model was a partly black box model of the Crazyflie dynamics by [5]. This model was compared to the real world and is already mentioned in the introduction.

III. Simulation

The structure of the simulation is displayed in a block diagram (Fig. 1). The layout in EAGERx also contains the exact same blocks (nodes) and inputs/outputs as the block diagram, so having this visual diagram makes it more easy to implement in EAGERx. The arrangement of these blocks is based on the internal controllers and sensors of the Crazyflie, in order to make it as explicit and transparent as possible. The inputs of the three different objects in Figure 1 are the desired attitude (roll, pitch and yaw) in degrees and the desired thrust (a value between 0 and 65535) defining the average PWM signal to be sent to the motors. The outputs consist of the position and orientation of the Crazyflie.

Our agnostic part only consists of one PID controller, used to follow a trajectory. This controller compares the current position with a desired position along a predefined trajectory. Based on the error it produces a desired attitude and thrust to send to the object. The current agnostic part can also be replaced with, for example, a controller that needs to be trained. This can be done with reinforcement learning in the simulation framework. After validating the controller in simulation, the same framework can be used to validate it in the real world by changing the object to the real Crazyflie. Ideally the controller works equally well on the real quadcopter as it did in simulation.

For the simulation we used the coordinate system East, North, Up (ENU) convention adopted from Bitcraze [1] shown in Figure 2. Roll (φ) and yaw (Ψ) are clockwise while the pitch (θ) is defined counter-clockwise. In the next paragraphs the developed white-box simulation will be discussed in further detail.

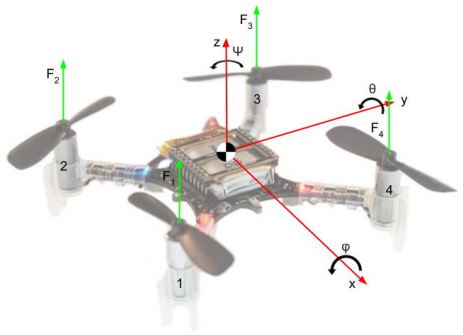


Fig. 2: Coordinate system used in the simulation [1] [10].

A. Attitude PID

The `Attitude PID` block takes the object input, i.e. the desired attitude, and the current attitude and calculates the desired attitude rate (i.e. angular velocity in deg/s). This is done by a discrete time PID controller, that is specified in the Crazyflie firmware. The equation is given in equation 1 where $e[k]$ is the error on time step k and $T_s = 1/rate$ is the sample time. However, the rates at which the code in the Crazyflie runs, differs from the simulation's rate, because running the `Attitude PID` block at a high rate only slows down the simulation and does not make it more accurate. This is why it runs at a lower rate. This influences the PID constants (k_p, k_i and k_d), so the values had to be changed to make the simulation accurate. For equation 1 follows, that when the rate is divided by 5, T_s gets 5 times larger, so k_d has to be multiplied by 5 and and k_i has to be divided by 5. The used rates and PID constants are given in table I.

$$u[k] = k_p e[k] + k_d \frac{e[k] - e[k-1]}{T_s} + k_i e[k] T_s \tag{1}$$

B. Attitude rate PID

The quadcopter attitude changes by creating a difference between the rotor speeds. The desired attitude rates are passed on to the `Attitude rate PID` block, which calculates the desired differences between the different motor PWM values based on the actual and desired attitude rate. The desired differences in PWM to cause roll, pitch and yaw are respectivily: ΔPWM_{roll}, ΔPWM_{pitch} and ΔPWM_{yaw} which are values between -32767 and 32767, half of the maximum motor PWM. This is done by a PID controller, which is adopted from the Crazyflie. However this PID runs at a different rate as well and therefore needs different PID constants. The used rates and PID constants are given in table I as well and are calculated in same way as for the `Attitude PID` (III-A).

Controller	Rate	kp	ki	kd
`Attitude PID`	100	6	0.6	0
`Attitude rate PID`	100	250	100	12.5

TABLE I: Constants of the PID controllers.

C. Power distribution

The `Power distribution` block calculates the PWM signal per motor (expressed as a command between 0 and 65535) based on the desired differences between the motor PWM values (ΔPWM_{roll}, ΔPWM_{pitch} and ΔPWM_{yaw}) and the desired average thrust. This is done by the formulas in equation 2 from Bitcraze [2], where the $desired\,thrust = PWM_{avg}$ and the calculated effects for the roll, pitch and yaw from the `Attitude rate PID` block are respectively ΔPWM_{roll}, ΔPWM_{pitch} and ΔPWM_{yaw}:

$$\begin{aligned}
PWM_1 =& PWM_{avg} - \frac{\Delta PWM_{roll}}{2} + \frac{\Delta PWM_{pitch}}{2} \\
&+ \Delta PWM_{yaw} \\
PWM_2 =& PWM_{avg} - \frac{\Delta PWM_{roll}}{2} - \frac{\Delta PWM_{pitch}}{2} \\
&- \Delta PWM_{yaw} \\
PWM_3 =& PWM_{avg} + \frac{\Delta PWM_{roll}}{2} - \frac{\Delta PWM_{pitch}}{2} \\
&+ \Delta PWM_{yaw} \\
PWM_4 =& PWM_{avg} + \frac{\Delta PWM_{roll}}{2} + \frac{\Delta PWM_{pitch}}{2} \\
&- \Delta PWM_{yaw}
\end{aligned} \tag{2}$$

In the simulation framework these PWM values are then passed on to the next block, the `PWM to force & torque` block. In the real world however, these values are passed on to the Embedded Speed Controllers, which power the rotors.

D. PWM to force and torque

The `PWM to force & torque` block calculates the force and the torque that act on the quadcopter. Currently, three forces on the quadcopter are considered: the thrust produced by the rotors, the drag force induced by the air resistance from the rotating rotors, and the gravitational force.

First, we consider the rotor thrust force. Equation 3 shows the relation between the PWM and thrust based on [4], where F_i is the thrust force per motor in Newton and PWM_i are the output values from equation 2 (Fig. 2). We added the PWM_{factor} to make the hover PWM the same as the hover PWM Jakob Kooi used in his model. The PWM_{factor} in such a way that the hover PWM values are equal. This could also be done with a offset, but the error became smaller when a factor was used. The thrust is only applied in the local z-dimension, which results in $\vec{F_i} = [0, 0, F_i]^T$.

$$\begin{aligned}
F_i =\ & 2.130295 \cdot 10^{-11} (PWM_{factor} PWM_i)^2 \\
+\ & 1.032633 \cdot 10^{-6} (PWM_{factor} PWM_i) \\
+\ & 5.484560 \cdot 10^{-4}
\end{aligned} \tag{3}$$

Next, the drag force is considered. Equation 4 shows the relation between the PWM and the rotor speed based on [4], where ω_i is the angular velocity of the rotor in rad/s and PWM_i is the output value for the rotor from equation 2. Equation 5 then calculates the rotor drag force from the drag coefficient matrix K_d, the angular velocities of the rotors ω_i and the body velocity $\dot{x}$, which is also based on [4]. The drag force then gets rotated from the world frame onto the quadcopter body frame with rotation matrix R_{drag}.

$$\omega_i = 0.04076521 PWM_i + 380.8359 \tag{4}$$

$$\vec{F}_{drag,rotors} = \mathbf{R}_{drag} \cdot \mathbf{K}_d \left(\sum_{i=0}^{3} 2\omega_i \right) \vec{\dot{x}} \tag{5}$$

Finally, the gravitational force has to be considered. This force is applied by default in the PyBullet engine and does not have to be applied manually.

By taking the sum of the forces, the total force that should be applied on the quadcopter's center of mass in the local coordinate system can be calculated. This is shown in equation 6, where $\vec{F_i}$ is the force from equation 3 and $\vec{F}_{drag,rotors}$ is the rotor's drag force (eq. 5). As said, the gravitational and body drag force will be automatically applied by PyBullet.

$$\vec{F}_{tot} = \left(\sum_{i=1}^{4} \vec{F_i}\right) - \vec{F}_{drag,rotors} \tag{6}$$

When summing the forces as shown in equation 6, the forces $\vec{F_i}$ on the rotors also result in torques. By taking the sum of the cross product between the distance vector and the force vector the total torque can be calculated (eq. 9), where the distance vector ($\vec{r_i}$) is the distance from the centre of mass to the point where the thrust is applied, i.e. the length and direction of the quadcopter's arms. Since the gravity and the $\vec{F}_{drag,rotors}$ apply in the centre of mass, they do not add extra torque to equation 7.

$$\vec{T}_{thrust} = \sum_{i=1}^{4} \vec{r_i} \times \vec{F_i} \tag{7}$$

The rotation of the rotors also cause yaw in the x,y-plane of the quadcopter. The relation between the torque per motor (τ_i) and the thrust force per motor (f_i) is given in equation 8 [4]. The direction of the torque depends on the rotational direction of the rotor. When all motors have the same speed, the torques cancel out each other.

$$\tau_i = 0.005964552 f_i + 1.563383 \cdot 10^{-5} \tag{8}$$

The total resulting torque to be applied will then be the sum of eq. 7 and the sum of all torques in eq. 8. The factor d_i represents the direction of rotation of that rotor.

$$\vec{T}_{tot} = \vec{T}_{thrust} + \left(\sum_{i=1}^{4} \tau_i d_i\right) \tag{9}$$

E. PyBullet

Finally the calculated force and torque are applied in the `PyBullet` block [3]. This is a Bullet physics engine implemented in Python, which makes it possible to simulate the Crazyflie in a 3D environment. This physics engine is the main difference between the three different objects shown in Figure 1. At this point the new state is calculated. In the black-box model this is done by integrating the ODE for one time step using Runge Kutta 4 [5]. The white-box model calculates the new position for the next step by applying the total force (eq. 6) and torque (eq. 9). PyBullet also applies the gravitational force by default. As outputs, it returns the orientation, position, angular velocity and linear velocity. In the real world, the Crazyflie is controlled by using its onboard controllers, sensors and motors.

F. Accelerometer

In the `Accelerometer` block, the inputs orientation and linear velocity from the PyBullet engine are converted into the quadcopter's accelerometer measurement, since the PyBullet engine does not output this measurement by default.

Calculating the acceleration can be done by differentiating the last two known velocities, so $\vec{a} = \frac{d\vec{v}}{dt}$. This body acceleration however does not include the gravitational acceleration, while the real accelerometer does measure this. Simply adding the dot product of the gravitational vector and the current rotation matrix (eq. 10) to the body's acceleration adjusts for that (eq. 11). In equation 11 the s stands for $\sin$ and the c for $\cos$. The pitch is θ and the roll is φ.

$$\mathbf{R}_{gravity} = \begin{bmatrix} c(\theta) & 0 & s(\theta) \\ -s(\varphi)s(\theta) & c(\varphi) & s(\varphi)c(\theta) \\ -c(\varphi)s(\theta) & -s(\varphi) & c(\varphi)c(\theta) \end{bmatrix} \tag{10}$$

$$\vec{a} = \frac{d\vec{v}}{dt} + \mathbf{R}_{gravity} \cdot [0, 0, 9.81] \tag{11}$$

This acceleration vector $\vec{a}$ is then passed through to the `state estimator` block.

G. Gyroscope

The `Gyroscope` block takes the angular velocity from PyBullet and directly forwards it to the output.

H. State estimator

The `state estimator` block used in the Crazyflie is based on the algorithm developed by Madgwick [6]. It uses the measurements of the (simulated) gyroscope and the (simulated) accelerometer, to calculate the attitude (roll, pitch and yaw) of the quadcopter. The calculations are taken from the Bitcraze Crazyflie firmware [2] and copied into the `state estimator` block to estimate the state of the quadcopter. The calculations of the attitude also use a PI controller. Which constants had to be changed and are given in table II. These constants are changed by the same calculation as in subsection III-A.

Controller	Rate	kp	ki
`State estimator`	240	0.4	0.0005

TABLE II: Constants of the PI controller in the state estimator.

IV. Results

For the validation of the Crazyflie simulation model we used the ODE model made by [5]. Because this ODE model is implemented as a second object in EAGERx, running the two simulation models consecutively is very simple. The ODE model was assumed to be very similar to the real world Crazyflie, as can be concluded from the results in [5]. Therefore, we will validate our simulation by comparing our results with that of the ODE model. Hence, the ODE simulation is used as a reference for the real world. In order to prevent accumulating errors after a certain amount of time,

a closed-loop trajectory controller was used to fly the models in a certain trajectory.

A trajectory PID controller (`Trajectory PID` block in Fig. 1) was developed based on the position PID controller in [7]. Four different trajectories were simulated to compare the simulation to the ODE model: a lemniscate, a line, a triangle and a rectangle. This can be seen in Figure 3, where the trajectories and attitudes of both models are plotted on top of each other, next to the commanded trajectory. Both models do not exactly follow the commanded trajectory. Nonetheless, these images imply that the simulation is quite accurate, because the two different trajectories are visually similar to each other. The highest mean error for all the trajectories are 0.172 m in x-dimension, 0.0468 m in z-dimension and 6.84 deg for the pitch.

It is worth noting that Jacob's model is only available in 2D, so the validation will be in 2D as well even though our simulation is in 3D. However, this should not be a problem since the pitch and roll are modelled identically.

For the validation, the timestamp, the positions and orientations of the two different objects are logged in a CSV file for every timestep. The agnostic part of EAGERx always runs at the same rate, but the rates at which the objects run differ. The simulation mostly runs at a rate of 240 Hz because of Pybullet, but the PID controllers run at a rate of 100 Hz to save computational power. The ODE model runs at 50 Hz, because the model is validated at this rate. The data is analysed and the error is calculated for every time step. The error is the absolute value of the difference between the simulation's and the ODE model's value for the following quantities: x-position (x [m]), z-position (z [m]) and pitch (θ [deg]) (eq. 12). So the error for x-, z-position and pitch are respectively Δx, Δz and $\Delta\theta$.

$$\begin{aligned} \Delta x &= |x_{simulator} - x_{ODE\ model}| \\ \Delta z &= |z_{simulator} - z_{ODE\ model}| \\ \Delta\theta &= |\theta_{simulator} - \theta_{ODE\ model}| \end{aligned} \tag{12}$$

The maximum error is the highest value for the error in a quantity. The mean error (μ) is the mean of the error in a quantity at every time step. The standard deviation (σ) is calculated with the formula in equation 13. The error in the x-position (Δx) is used as an example, but the calculation is the same for every quantity. N is the total number of time steps.

$$\sigma_x = \sqrt{\frac{\sum(\Delta x - \mu_x)^2}{N}} \tag{13}$$

The trajectories and results will be further discussed in the following subsections.

A. Lemniscate trajectory

The maximum pitch in the ODE model for the lemniscate trajectory (Fig. 3a) is $13.86°$, so it is the trajectory with the smallest maximum pitch angle. This means it is well within the limits of the ODE model and a relatively simple trajectory with no big direction changes. A new set point is sent every time step, slowly moving along the lemniscate.

The results of the lemniscate trajectory are shown in table III.

Trajectory	Max error	Mean error	Standard deviation
`Position x [m]`	0.243	0.0837	0.0800
`Position z [m]`	0.0757	0.0346	0.0195
`Orientation [deg]`	9.12	2.40	1.83

TABLE III: Results of the lemniscate trajectory.

B. Line trajectory

The maximum pitch in the ODE model for the line trajectory (Fig. 3b) is $31.93°$, therefore it is the trajectory with the highest maximum pitch angle. This means the results from the ODE model might deviate a bit from reality. This is because of the fact that the ODE model was determined for angles up to $30°$, but accuracy outside of this range was not that great [5].

The line trajectory is added to the validation to show the performance of the simulation model at the boundaries of the ODE model, to verify the simulation model over the widest range possible. The results are shown in table IV. The maximum and mean error are the highest for the line trajectory, which was expected because the quadcopter is flown at the highest speed.

In this case, the set points do not move along the line, but lie at one end of the line for a specific period of time and than change to the other end. Therefore, in figure 3b the set points are given by numbers, instead of by a line. This results in a big difference between the current position and the set point, thus does the trajectory controller make the pitch angle go so high.

Trajectory	Max error	Mean error	Standard deviation
`Position x [m]`	0.353	0.172	0.114
`Position z [m]`	0.0919	0.0238	0.0200
`Orientation [deg]`	24.7	6.84	4.36

TABLE IV: Results of the line trajectory.

C. Triangle trajectory

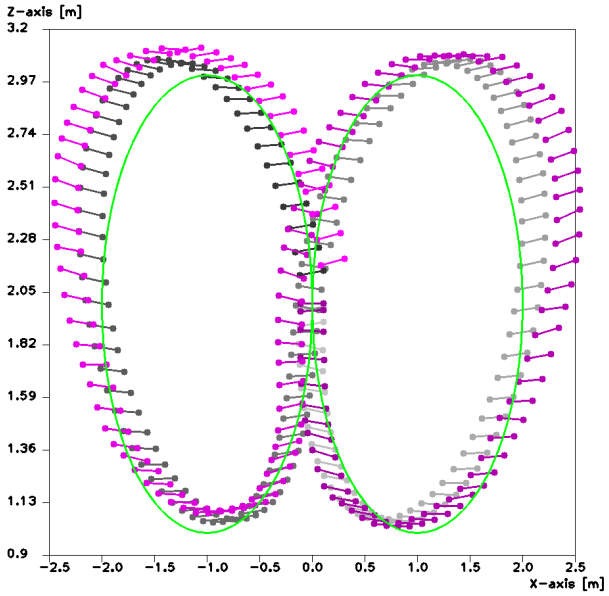
The maximum pitch in the ODE model for the triangle trajectory (Fig. 3c) is $18.91°$. The results on this trajectory (tab. V) are about the same as the other trajectories with the same maximum pitch angle. This trajectory is added, because of the sharp angles in the trajectory. It can be seen that the results begin to differ more and more after every angle. The set points are sent in the same way as the lemniscate, so moving along the triangle.

Trajectory	Max error	Mean error	Standard deviation
`Position x [m]`	0.349	0.130	0.0990
`Position z [m]`	0.0746	0.0468	0.0369
`Orientation [deg]`	13.6	3.70	3.12

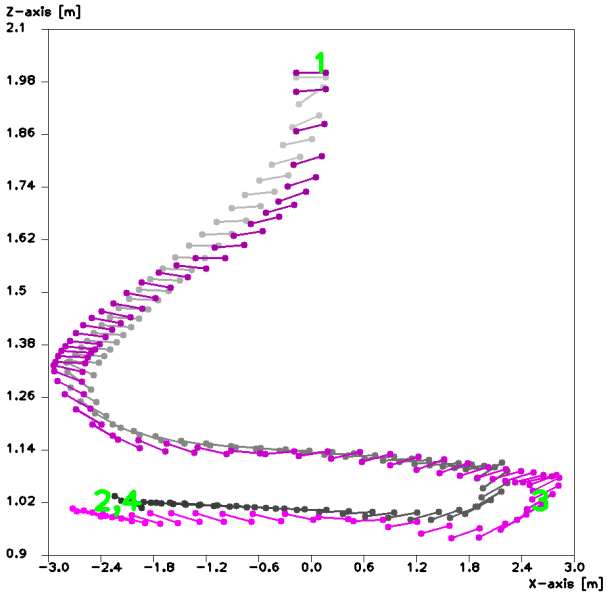
TABLE V: Results of the triangle trajectory.

D. Rectangle trajectory

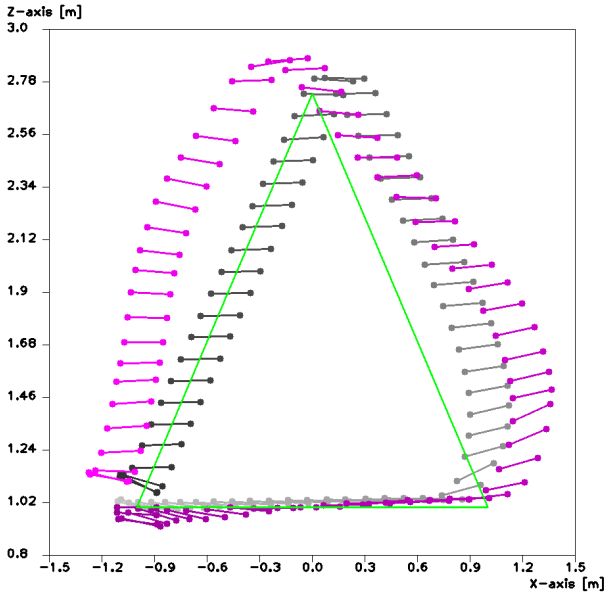
The maximum pitch in the ODE model for the rectangle trajectory (Fig. 3d) is $15.61°$. The errors for this trajectory are



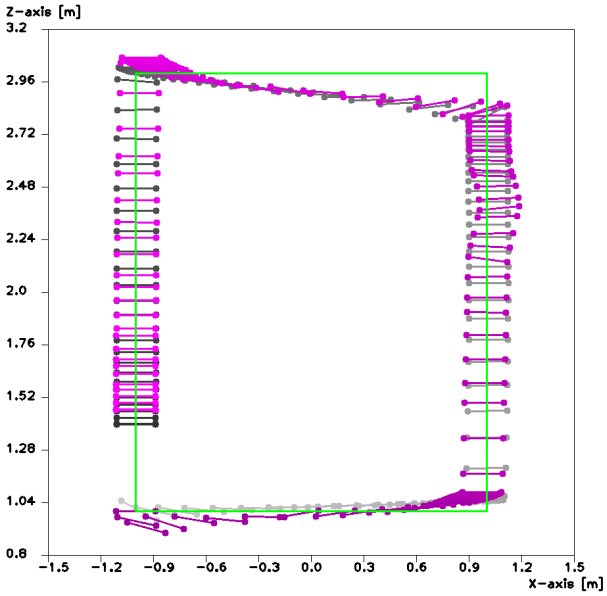
(a) Lemniscate trajectory



(b) Line trajectory



(c) Triangle trajectory



(d) Rectangle trajectory

Fig. 3: The four different trajectories.

the smallest of all. This is because the movement is solely in x- or z-direction, which means a constant x- and z-value half of the time. The set points also move along the rectangle, similar to the lemniscate and the triangle.

Trajectory	Max error	Mean error	Standard deviation
Position x [m]	0.101	0.0345	0.0246
Position z [m]	0.0751	0.0335	0.0210
Orientation [deg]	12.6	1.37	2.17

TABLE VI: Results of the rectangle trajectory.

E. Discussion

A cause for the errors could be the use of a different mass for the Crazyflie. The ODE model uses a mass of 33.03 grams while the mass assumed in the simulation is 27 grams. The weight provided by the manufacturer of the quadcopter is 27 grams, therefore this value is used in the simulation. However, the ODE model is based on a customised Crazyflie with OptiTracks and thus has a different mass. This different mass effects the moments of inertia as well, because the OptiTracks are placed away from the centre of mass. We tried running our model with a mass of 33.03 grams and increased the PWM_{factor} from equation 3 at the same time. In Figure 4 is

the plot shown with the error results given in table VII. The main difference between table III and VII is the mean error in x-dimension. This is probably due to the different moments of inertia.

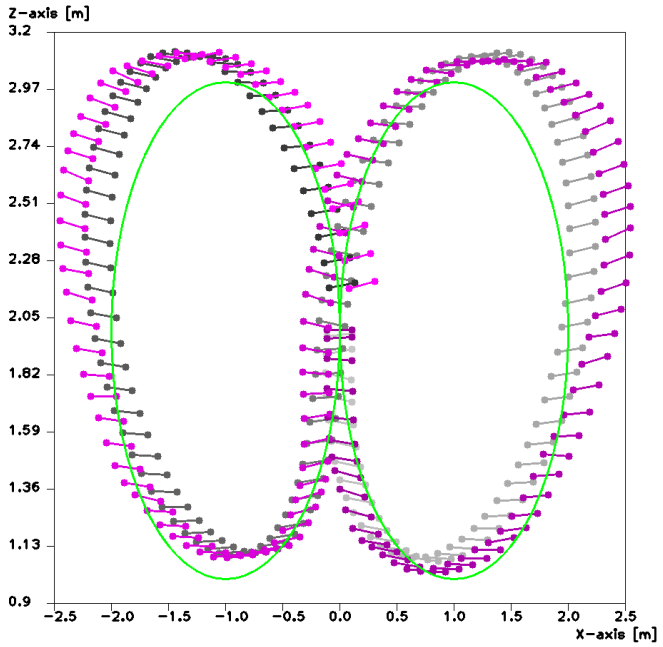


Fig. 4: Lemniscate trajectory with a mass of 33.03 grams and an increased PWM_{factor} for the simulation. The legend is the same as in Figure 3b.

Trajectory	Max error	Mean error	Standard deviation
`Position x [m]`	0.235	0.123	0.0856
`Position z [m]`	0.0827	0.0274	0.0195
`Orientation [deg]`	8.49	2.54	1.82

TABLE VII: Results of the lemniscate trajectory with a mass of 33.03 grams and an increased PWM_{factor} for the simulation.

Another possible cause might be the constants of the PID controllers. These work at a different rate in the simulation then they do in the Crazyflie. The PID constants were tuned for the lower rates, but they might deviate from the values that the closest represent the real quadcopter. This slightly changes the PID controller behaviour, which makes the simulation differ from reality and thus from the ODE model.

Furthermore, the different trajectories are all tested with the same PID controller: `Trajectory PID` (Fig. 1), which probably influences the behaviour of the models. It is possible that a different controller, results in a different behaviour of the simulation and the ODE model. For example, a faster reacting PID controller will send a stronger fluctuating desired attitude and desired thrust to the simulation and the ODE model, which could react different.

Other differences between the simulation and reality, that can induce errors, are the delays and the sensors. The delays in the real Crazyflie aren't simulated in the simulation. The sensor values returned by the physics engine are perfect unlike the real Crazyflie sensor values, which are clouded by noise.

Finally, it is important to realise that the ODE model is also a model and therefore not perfect. So errors of the simulation might be masked or enlarged by errors in the ODE model.

V. Conclusion and Future Work

We have presented a simulation model of the onboard controllers and dynamics of the Crazyflie 2.0 and implemented it in EAGERx. This model can be used to, for example, train quadcopter controllers using reinforcement learning. The implementation in EAGERx allows these trained controllers to be easily used in the real world. We have also implemented an ODE model [5] in EAGERx to validate our own simulation model. Both models seemed to perform similar.

However, there are still possible improvements for our simulation model. Our simulation is validated in 2D, but it is made in 3D. So our model is not validated for 3D. This can be done by flying a trajectory in 3D with the real Crazyflie and compare it to our model.

Furthermore, ground effects when flying close to the ground or the effect the downwash of a quadcopter has on another quadcopter are currently not implemented. This did not have an effect on our simulation, because we flew far above the ground and only with one quadcopter, but it could cause issues when this is not the case. So this is also something that could be added in the future.

Additionally, the differences in mass give some interesting results that need to be researched further in the future.

Also, the constants of our controllers could be tuned to get a more accurate model. Other sensors could also be implemented with a more realistic output. For example, our simulated sensors get their input from PyBullet while in reality the output of the sensors have noise. With the use of filters this noise is filtered [2] to estimate the real signal, so simulated sensors with noise could be modelled and added to the simulation. Also, since the 2D ODE model lacked the implementation of yaw, this axis of rotation has yet to be validated.

VI. Acknowledgements

We want to thank our supervisors Bas van der Heijden, Dennis Benders and Robert Babuška for their help on this project.

References

[1] Bitcraze. (2022, May 31). *The coordinate system of the crazyflie 2.x*. Retrieved May 31, 2022, from https://www.bitcraze.io/documentation/system/platform/cf2-coordinate-system/

[2] Bitcraze AB. (2022, May 30). *Bitcraze information*. Retrieved May 30, 2022, from https://www.bitcraze.io/documentation/repository/crazyflie-firmware/master/

[3] Coumans, E., & Bai, Y. (2016). Pybullet, a python module for physics simulation for games, robotics and machine learning.

[4] Förster, J. (2015, August). *System identification of the crazyflie 2.0 nano quadrocopter* (Master's thesis). ETH Zurich. Zurich, ETH Zurich.

[5] Kooi, J. E., & Babuška, R. (2021). Inclined quadrotor landing using deep reinforcement learning. *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2361–2368.

[6] Madgwick (Ed.). (2014). *Ahrs algorithms and calibration solutions to facilitate new applications using low-cost mems.* (Doctoral dissertation). University of Bristol.

[7] Panerati, J., Zheng, H., Zhou, S., Xu, J., Prorok, A., & Schoellig, A. P. (2021). Learning to fly - a gym environment with pybullet physics for reinforcement learning of multi-agent quadcopter control. *CoRR*, *abs/2103.02142*.

[8] Silano, G., Aucone, E., & Iannelli, L. (2018). CrazyS: A Software-In-The-Loop Platform for the Crazyflie 2.0 Nano-Quadcopter. *2018 26th Mediterranean Conference on Control and Automation (MED)*, 352–357.

[9] Van der Heijden, B., & Luijkx, J. (2022). *What is EAGERx*. Retrieved May 30, 2022, from https://eagerx.readthedocs.io/en/master/

[10] Zimmerman, N. (2021, January). *Embedded implementation of reactive end-to-end visual controller for nano-drones* (Master's thesis). Università della Svizzera Italiana. Lugano, Università della Svizzera Italiana.