RotorS – A Modular Gazebo MAV Simulator Framework

Chapter *in* Studies in Computational Intelligence · January 2016

DOI: 10.1007/978-3-319-26054-9_23

4 authors:

Fadri Furrer
ETH Zurich
18 PUBLICATIONS **340** CITATIONS
SEE PROFILE

Michael Burri
ETH Zurich
30 PUBLICATIONS **1,883** CITATIONS
SEE PROFILE

Markus Wilhelm Achtelik
ETH Zurich
30 PUBLICATIONS **2,832** CITATIONS
SEE PROFILE

Roland Siegwart
ETH Zurich
797 PUBLICATIONS **35,187** CITATIONS
SEE PROFILE

RotorS – A Modular Gazebo MAV Simulator Framework

Fadri Furrer, Michael Burri, Markus Achtelik, and Roland Siegwart

Autonomous Systems Lab, ETH Zurich LEE J,
Leonhardstrasse 21, 8092 Zurich, Switzerland
{fadri.furrer,michael.burri,markus.achtelik,
rsiegwart}@mavt.ethz.ch
http://www.asl.ethz.ch

Abstract. In this chapter we present a modular Micro Aerial Vehicle (MAV) simulation framework, which enables a quick start to perform research on MAVs. After reading this chapter, the reader will have a ready to use MAV simulator, including control and state estimation. The simulator was designed in a modular way, such that different controllers and state estimators can be used interchangeably, while incorporating new MAVs is reduced to a few steps. The provided controllers can be adapted to a custom vehicle by only changing a parameter file. Different controllers and state estimators can be compared with the provided evaluation framework. The simulation framework is a good starting point to tackle higher level tasks, such as collision avoidance, path planning, and vision based problems, like Simultaneous Localization and Mapping (SLAM), on MAVs. All components were designed to be analogous to its real world counterparts. This allows the usage of the same controllers and state estimators, including their parameters, in the simulation as on the real MAV.

Keywords: ROS, Gazebo, Micro Aerial Vehicles, Benchmarking

1 Introduction

To test algorithms on MAVs, one needs access to expensive hardware and field tests usually consume a considerable amount of time and require a trained safety-pilot. Most of the errors, occurring on real platforms, are hard to reproduce, and often result in damaging the MAV. The RotorS simulation framework was developed to reduce field testing times and to separate problems for testing, making debugging easier, and finally reducing crashes of real MAVs. This is also convenient for student projects, where access to an expensive and complex real platform cannot always be granted. To solve higher-level tasks such as path-planning, the provided simulated MAVs can be used without any additional modification to the models. Besides a simulation of MAVs, the framework also includes a position controller and a state estimator, that work with the provided models.

The focus of this chapter will be to describe in detail the steps required to set up the RotorS simulator, depicted in Fig. 1, including the Robot Operating System (ROS) and Gazebo, and to present an evaluation framework. Once this chapter has been read, the reader will be able to set up the simulator, attach basic sensors to a MAV, and get it to fly autonomously in the simulation. The reader will also be able to compare different (custom) algorithms using the evaluation scripts. All the aspects learned and methods developed in this chapter can then be applied to a real MAV.

Fig. 1. A screenshot of the RotorS simulator. The scene is built up from Gazebo default models and a Firefly hex-rotor helicopter.

An overview of the main components of the RotorS simulator are shown in Fig. 1. In this chapter, the focus is put on the simulation part, shown on the left side in Fig. 1, but a lot of effort was put into keeping the structure of the simulator analogous to the real system. Ideally, all components used in the simulated environment can be run on the real platform without any changes.

All components, found on real MAVs, are simulated by Gazebo plugins and by the Gazebo physics engine. In the simulator we created a modular way of assembling MAVs. A MAV consists of a body, a fixed number of rotors, which can be placed at user specified locations, and some sensors attached to the body. Each rotor has motor dynamics, and accounts for the most dominant aerodynamic effects according to [7]. The parameters were identified on a real MAV, a "Firefly" from Ascending Technology[1] using recorded flight data. Several sensors, such as an Inertial Measurement Unit (IMU), a generic odometry sensor and a visual inertial sensor, consisting of a stereo camera and an IMU, and sensors developed by the user can be attached to the body. To simulate realistic conditions, we implemented noise models for the applied sensors. All the simulation data can be logged using rosbags [2], by having direct access to sim-

[1] Ascending Technologies `http://www.asctec.de/`

ulation data. Hence, ground truth data is available for each sensor, which eases debugging and evaluation of new algorithms.

To facilitate the development of different control strategies, a simple interface is provided. We present an implementation of a geometric controller [5], which gives access on various levels of commands, such as angular rates, attitude, or position control. Additionally, this acts as an ideal starting point to implement more advanced control strategies.

The last building block is the state estimation, used to obtain information about the state of the MAV at a high rate. While state estimation is crucial on real MAVs, in the simulation this part can be replaced by a generic (ideal) odometry sensor. This means position, orientation, linear and angular velocity of the MAV are directly provided by a Gazebo plugin. In the first tutorial section, Section 3.3, we are showing how to get the MAV airborne, using this ideal sensor. In the following section, we show how the Multi Sensor Fusion (MSF) framework, an open source framework based on an Extended Kalman Filter (EKF) for 6 Degrees of Freedom (DoF) pose estimation [6], can be used to handle noisy, low-rate, and high-latency sensor data.

Once the MAV is flying, higher level tasks can be tackled and tested in the simulation environment, such as obstacle avoidance and path planning. The reader will be provided with representative simulation environments including the widely used octomap representation of the 3 D occupancy grid [4].

This book chapter starts with a short summary of the underlying theory and references for further reading in Section 2. In particular it covers modeling, control and state estimation of an MAV. In the following section, Section 3, the basics of the simulator are explained, which are needed to get the MAV hovering. Also, the available sensors are explained in detail and how to add them to the Gazebo simulator. In Section 4 advanced topics are covered. This includes a deeper insight in the XML Macros (Xacro) scripting language, with an example of developing a custom asymmetric quad-rotor helicopter and a controller to enable hovering. Additionally, we show examples for solving higher level tasks using the simulator, such as collision avoidance or path planning. Finally, thoughts into the transition to real MAVs are given.

2 Background

In this section, we are presenting the theoretical background, needed to understand the following tutorials and how the simulator works. Throughout this chapter we refer to MAVs as multi-rotor helicopters, while many of the concepts are not limited to multi-rotor helicopters only.

The most important part is how to model the dynamics of an MAV. This allows to develop control strategies on different command levels such as attitude or position, which is explained in the second part of this section. Furthermore, we give a short introduction to state estimation, which is used to provide state information to the controller. For higher level tasks, such as path planning or

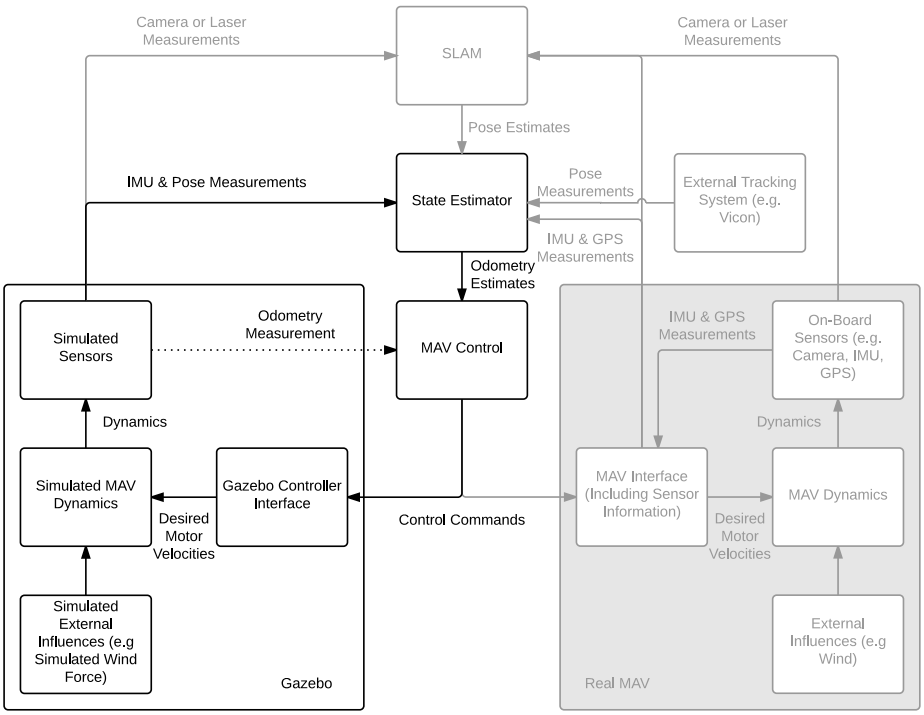

Fig. 2. Necessary building blocks to get an MAV airborne. This book chapter has a focus on the simulation part on the left side. All parts in black are covered in this book chapter and are available open source. The structure of the simulator is designed to match the real MAVs as close as possible and can be seen by comparing to the right.

local collision avoidance, we give an overview of the octree representation of a 3 D occupancy grid.

2.1 MAV Modeling

The forces and moments that are acting on an MAV can be split up into the forces and moments acting on each rotor, and the gravitational force acting on the Center of Gravity (CoG) of an MAV. All these forces combined together describe the full dynamics of an MAV.

Single Rotor Forces and Moments As depicted in Fig. 2.1, we will analyze the thrust force $\boldsymbol{F_T}$, the drag force $\boldsymbol{F_D}$, the rolling moment $\boldsymbol{M_R}$, and the moment

originating from the drag of a rotor blade, $\boldsymbol{M_D}$, from [7].

$$\boldsymbol{F_T} = \omega^2 C_T \cdot \boldsymbol{e_{z_B}} \tag{1}$$
$$\boldsymbol{F_D} = -\omega C_D \cdot \boldsymbol{v_A^\perp} \tag{2}$$
$$\boldsymbol{M_R} = \omega C_R \cdot \boldsymbol{v_A^\perp} \tag{3}$$
$$\boldsymbol{M_D} = -\epsilon C_M \cdot \boldsymbol{F_T} \tag{4}$$

where ω is the positive angular velocity of the rotor blade, C_T is the rotor thrust constant, C_D is the rotor drag constant, C_R is the rolling moment constant, and C_M is the rotor moment constant. All of these constants are positive. ϵ denotes the turning direction of the rotor, namely $+1$ (counter clockwise) or -1 (clockwise). $\boldsymbol{e_{z_B}}$ is the unit vector pointing in the z-direction in the rotor's body frame. ($\boldsymbol{u}^\perp$ denotes the projection of a vector $\boldsymbol{u}$ onto the rotor plane as can be seen in Fig. 2.1. It can be calculated as:

$$\boldsymbol{u}^\perp = \boldsymbol{e_{z_B}} \times (\boldsymbol{u} \times \boldsymbol{e_{z_B}}) = \boldsymbol{u} - (\boldsymbol{u} \cdot \boldsymbol{e_{z_B}}) \cdot \boldsymbol{e_{z_B}} \tag{5}$$

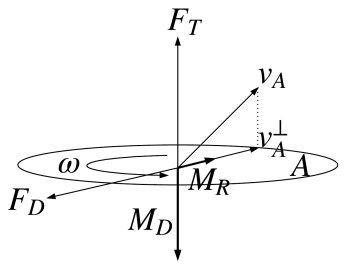

Fig. 3. Forces and moments acting on the center of a single rotor.

MAV Dynamics Fig. 2.1 shows a quad-rotor helicopter with numbered rotors and the forces acting on it. Looking at the whole MAV, we can write down the equations of motion from Newton's law, and Euler's equation as:

$$\boldsymbol{F} = m \cdot \boldsymbol{a} \tag{6}$$
$$\boldsymbol{\tau} = \boldsymbol{J} \cdot \dot{\boldsymbol{\omega}} + \boldsymbol{\omega} \times \boldsymbol{J} \cdot \boldsymbol{\omega} \tag{7}$$

with m, the mass of the MAV, $\boldsymbol{a}$ its acceleration, $\boldsymbol{J}$ its inertia matrix, and $\boldsymbol{\omega}$ its angular velocities. The linear part in (6) is expressed in the world coordinate system, while the rotational part in (7) is expressed in the rotating body frame.

For a multi-rotor helicopter with n motors, (6) and (7) can be written as:

$$\sum_{i=0}^{n-1}(\boldsymbol{R_{WB}}\underbrace{(\boldsymbol{F_{T,i}}+\boldsymbol{F_{D,i}})}_{\boldsymbol{F_i}})+\boldsymbol{F_G}=m\cdot\boldsymbol{a} \tag{8}$$

$$\sum_{i=0}^{n-1}(\boldsymbol{M_{R,i}}+\boldsymbol{M_{D,i}}+\boldsymbol{F_i}\times\boldsymbol{r_i})=\boldsymbol{J}\cdot\dot{\boldsymbol{\omega}}+\boldsymbol{\omega}\times\boldsymbol{J}\cdot\boldsymbol{\omega} \tag{9}$$

where $\boldsymbol{R_{WB}}$ is the rotation matrix rotating vectors from the body frame B to the world frame W, and $\boldsymbol{r_i}$ denotes the vector from the CoG of the MAV to the center of the i-th rotor.

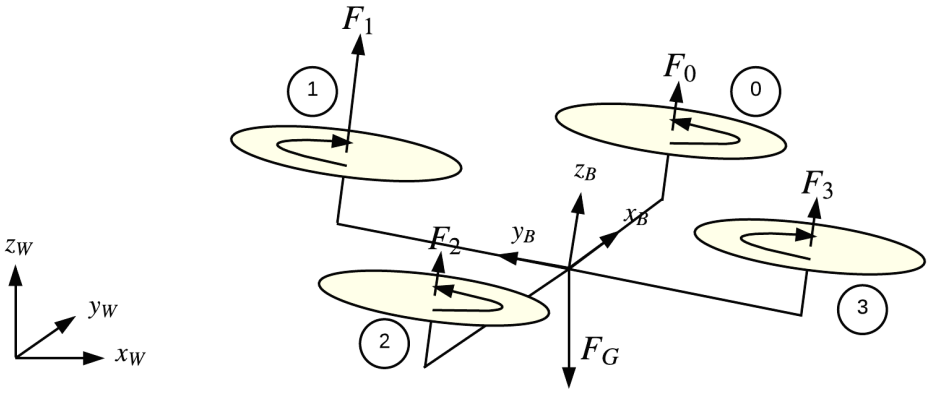

Fig. 4. Sketch of a quadrotor with the body centered body frame B and the global world frame W. The main forces from the individual rotors F_i and gravity F_G are acting on the main body.

2.2 Control

In this section, we only cover the control of multi-rotor helicopters. Most of the current multi-rotor helicopters have rotor configurations, where the rotor axes normals point in the z-axis of the body frame z_B, therefore we will look at these configuration in particular.

In order to control an MAV, we first need to find a mapping between the system's output, which is the resulting thrust T, the accumulated thrust of all rotors, and torque $\boldsymbol{\tau}$ acting on the CoG of the helicopter, and the system's input, which are the angular velocities of each rotor ω_i. We are only considering the thrust forces of each rotor, its resulting moments, and the drag moments for

now. Hence we can formulate the following equation:

$$\begin{pmatrix} T \\ \boldsymbol{\tau} \end{pmatrix} = \boldsymbol{A} \cdot \begin{pmatrix} \omega_0^2 \\ \omega_1^2 \\ \vdots \\ \omega_n^2 \end{pmatrix} \tag{10}$$

We refer to this mapping matrix $\boldsymbol{A}$ as the allocation matrix, which is of size $\boldsymbol{A} \in \mathbb{R}^{4\times n}$. For a quad-rotor helicopter, as in Fig. 2.1, such an allocation matrix is given by:

$$\boldsymbol{A} = \begin{pmatrix} C_T & C_T & C_T & C_T \\ 0 & lC_T & 0 & -lC_T \\ -lC_T & 0 & lC_T & 0 \\ -C_T C_M & C_T C_M & -C_T C_M & C_T C_M \end{pmatrix} \tag{11}$$

By looking at a multi-rotor helicopter with all rotor axes pointing in the same direction, we can only generate a thrust T pointing in the direction of the rotor blade's normal vector, we assume here, that this coincides with z_B. Hence, only the thrust T, and the moments around all three body axes x_B, y_B, and z_B can be controlled directly. To be able to navigate in 3 D space, the vehicle has to be tilted towards a setpoint. Therefore we want to control the overall thrust of the vehicle, the direction of z_B (by controlling the roll- and pitch angle), and the yaw rate ω_z. This is usually referred to as the attitude controller. Because the dynamics of the attitude are usually much faster than the translational dynamics, often a cascaded control approach is chosen. The attitude control loop usually runs onboard a micro controller at high rate, while the position loop, which calculates the desired attitude and thrust, runs at a lower rate. This yields a control structure as depicted in Fig. 2.2. In this simulator, we are using the geometric controller proposed in [5], which has the same structure, but directly calculates the resulting thrust and moments at the same rate.

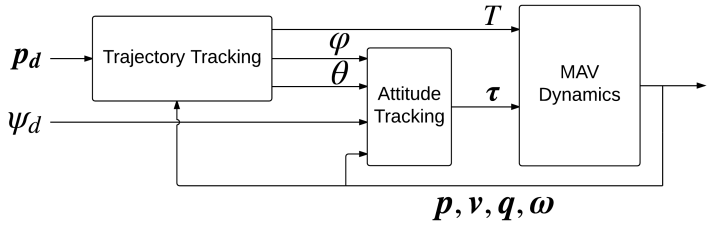

Fig. 5. Controller sketch, with the desired position $\boldsymbol{p_d}$ and the desired yaw angle ψ_d. Usually, position control is split into two parts: An outer trajectory tracking controller calculates attitude and thrust references, that are tracked by an inner attitude tracking controller.

2.3 State Estimation

One of the key elements for enabling stable and robust MAV flights is accurate knowledge of the state of the MAV. In this section, we give a brief overview of how to fuse IMU measurements with a generic 6 DoF pose measurement in order to obtain estimates of the states that are commonly needed for a position controller for an MAV. Such a 6 DoF pose measurement could be obtained for instance from visual-odometry or visual SLAM systems, laser rangefinder-based techniques or marker-based localization. Another very common technique to get 6 DoF pose measurements is to use an external tracking system, which provides highly accurate measurements at high rate. This method however has the disadvantage of being tied to external infrastructure. There are many other sensor types available, like pressure, optical flow, or ultrasonic sensors, but an in-depth explanation would be beyond the scope of this chapter. A good reference and overview of the properties of these sensors can be found in [11].

While IMU measurements are available on all of today's MAVs, the pose can be acquired by different methods as outlined above. IMU measurements and pose measurements have very complementary properties: Measurements from IMUs, typically used onboard MAVs, are available at high rate and with low delay, but are corrupted by noise and a time-varying bias. As a result, solely time-discrete integration (dead-reckoning) of these sensors makes a steadily accurate estimation of the absolute pose of the vehicle nearly impossible. In contrast, the methods for estimation of the 6 DoF pose usually exhibit no drift or only very low drift, but their measurements usually arrive at a much lower rate, and with high delay, due to their computational complexity. Combining both types of measurements yields a (almost) drift-free estimate of the state, at high rate and with low delay. We describe the essentials of how to do this using an EKF formulation, as shown below, and refer to [1] for a detailed description.

IMU Sensor model: A common IMU model is given by

$$\boldsymbol{\omega}_m = \boldsymbol{\omega} + \boldsymbol{b}_\omega + \boldsymbol{n}_\omega \tag{12}$$

$$\boldsymbol{a}_m = \boldsymbol{a} + \boldsymbol{b}_a + \boldsymbol{n}_a \tag{13}$$

where the measured quantities are denoted with subscript m. $\boldsymbol{b}_\omega$ and $\boldsymbol{b}_a$ are biases on the measured angular velocities, and accelerations respectively. These biases are modeled as random walk, having zero mean white Gaussian noise as time-derivative:

$$\dot{\boldsymbol{b}}_\omega = \boldsymbol{n}_{b_\omega} \tag{14}$$

$$\dot{\boldsymbol{b}}_a = \boldsymbol{n}_{b_a} \tag{15}$$

where $\boldsymbol{n}_{b_\omega}$ and $\boldsymbol{n}_{b_a}$ are noise levels.

State Representation: Most controllers are separated into a position loop and an attitude loop. This works well under the assumption, that the rotational

dynamics are faster than the translational motion. For the outer loop, the position $\boldsymbol{p}$ and the velocity $\boldsymbol{v}$ are needed in world coordinates. The inner attitude loop needs the orientation $\bar{\boldsymbol{q}}$ and the angular velocity $\boldsymbol{\omega}$, which is provided by the IMU. Together with the bias states from the IMU model, this leads to the following state vector:

$$\boldsymbol{x} = \left[\boldsymbol{p}^T\ \boldsymbol{v}^T\ \bar{\boldsymbol{q}}^T\ \boldsymbol{b}_a^T\ \boldsymbol{b}_\omega^T\right]^T \tag{16}$$

Thanks to the low complexity of time-discrete integration of IMU measurements, IMU measurements are commonly used as input for the time-update phase ('prediction') of the EKF. This has also the advantage of being independent from specific vehicle dynamics and their model parameters, and furthermore avoids having the angular rate in the state. This leads to the following dynamic model:

$$\begin{aligned}
\dot{\boldsymbol{p}} &= \boldsymbol{v} \\
\dot{\boldsymbol{v}} &= \boldsymbol{C} \cdot (\boldsymbol{a}_m - \boldsymbol{b}_a - \boldsymbol{n}_a) + \boldsymbol{g} \\
\dot{\bar{\boldsymbol{q}}} &= \frac{1}{2}\bar{\boldsymbol{q}} \otimes \begin{bmatrix} 0 \\ \boldsymbol{\omega}_m - \boldsymbol{b}_\omega - \boldsymbol{n}_\omega \end{bmatrix} \\
\dot{\boldsymbol{b}}_a &= \boldsymbol{n}_{b_a} \\
\dot{\boldsymbol{b}}_\omega &= \boldsymbol{n}_{b_\omega}
\end{aligned}$$

For the derivation of the time-discrete integration, the error-state dynamics, the system propagation matrix, and the noise covariance matrix for this particular problem, we refer to [1]. A very good tutorial on the background is given in [10][2].

Measurement Model: The measurement equations can be separated into position $\boldsymbol{p}_m$ and attitude measurements $\bar{q}_m$, which express the measured pose of the IMU with respect to the world frame.

$$\boldsymbol{p}_m = \boldsymbol{p} + \boldsymbol{n}_p \tag{17}$$

$$\bar{q}_m = \bar{q} \otimes \delta\bar{q}_n \tag{18}$$

where $\delta\bar{q}_n$ represents a small error rotation and $\boldsymbol{n}_p$ is zero mean, white Gaussian noise. This simple model assumes that the origin of the pose-sensor coincides with the IMU, which is usually not the case. Also, the frame of reference of the pose-sensor may not align with the world frame. However, one can compensate for these misalignments, since they are often observable, and thus do not need to be calibrated beforehand. For the derivations and an observability analysis, we refer to [1, 11].

2.4 Octree Representation

For 3 D collision avoidance and path planning, it is important to have an efficient representation of obstacles (necessary for collision checking). In recent work,

[2] Be aware that in [10], the JPL quaternion notation is used, while [1] and many libraries, including ROS and Eigen, use the Hamilton notation.

octree representations are widely used for that purpose. An octree is a tree where every node has exactly eight children, which makes it well suited for efficient storage in memory. It is often used as a representation of whether a 3 D space is occupied or not. Every node represents a certain part of a 3 D space, this part can get subdivided into eight parts (of equal size), denoted as octants, of this subspace. This can recursively be applied until a leave has the desired resolution of the represented space, see Fig. 2.4. If every octant of a node has

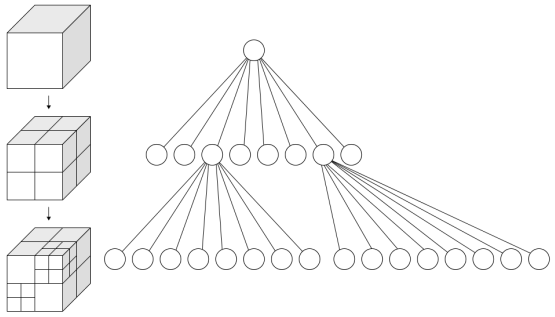

Fig. 6. Left: Recursive subdivision of a cube into octants. Right: The corresponding octree.[4]

the same value, the node value can be set to this value, and the octant nodes can be omitted in the tree.

3 Tutorials

This section explains how to use the RotorS simulator with its main components. It demonstrates how to get the MAV into hovering mode, and how to attach sensors to it. The setup of the simulator, for OS X and Ubuntu, is shown in Section 3.1. An overview of the different components of the simulator is given in Section 3.2. Then, we explain how to use our controllers to get the MAV into hovering mode, by assuming that we have a sensor available, which delivers a full *Odometry* message [5], in Section 3.3. Followed by Section 3.4, which covers the same scenario, but with a setup closer to real world applications, where only a pose sensor and an IMU are available. The state of the MAV is estimated from these measurements. We explain how sensors can be added in Section 3.5, and how to use our evaluation scripts in Section 3.6. For more advanced tutorials on how to build a custom model, controller, or sensor plugin, see Section 4. Directions on how to solve high level tasks with the simulator are given in Section 5.

[4] Octree2 by WhiteTimberwolf, PNG version: Nü - Own work. Licensed under CC BY-SA 3.0 via Wikimedia Commons - `http://commons.wikimedia.org/wiki/File:Octree2.svg#/media/File:Octree2.svg`

[5] As described on: `http://docs.ros.org/api/nav_msgs/html/msg/Odometry.html`

3.1 Simulator Setup

Before installing the MAV simulator RotorS, the following steps are necessary: First, install ROS according to the official wiki-page `http://wiki.ros.org/indigo/Installation`. Second, the following external packages are needed.

Ubuntu is the recommended OS to run ROS and the package manager should be used to install the necessary dependencies.

```
$ sudo apt-get install ros-indigo-joy ros-indigo-octomap-ros
    python-wstool python-catkin-tools
```

OS X can run ROS native, using homebrew, and we recommend using pip to install the necessary dependencies.

```
$ pip install wstool
$ pip install catkin-tools
```

Installing RotorS [6] is independent of the operating system and can be done using wstool, assuming the catkin workspace is located at `~/catkin_ws/src`. Make sure that wstool is initialized before running the following commands.

```
$ cd ~/catkin_ws/src
$ wstool init
$ wstool set rotors_simulator https://github.com/ethz-asl/
    rotors_simulator.git --git -y
$ wstool set mav_comm https://github.com/ethz-asl/mav_comm.git
    --git -y
$ wstool update
$ cd ~/catkin_ws/
$ catkin build
$ source ~/catkin_ws/devel/setup.bash
```

If you want use the demos with an external state estimator, the following additional packages are needed:

```
$ cd ~/catkin_ws/src
$ wstool set ethzasl_msf https://github.com/ethz-asl/
    ethzasl_msf.git --git -y
$ wstool set rotors_simulator_demos https://github.com/ethz-
    asl/rotors_simulator_demos.git --git -y
$ wstool set glog_catkin https://github.com/ethz-asl/
    glog_catkin.git --git -y
$ wstool set catkin_simple https://github.com/catkin/
    catkin_simple.git --git -y
```

[6] For Indigo and Jade there are debian packges available, which can be installed with: `sudo apt-get install ros-<version>-rotors-simulator ros-<version>-mav-comm`.

```
$ wstool update
$ cd ~/catkin_ws/
$ catkin build
$ source ~/catkin_ws/devel/setup.bash
```

3.2 Simulator Overview

The RotorS simulator is split up in various packages as shown in Fig. 3.2. The general way to incorporate an already existing robot into the simulation is by describing its geometry, and kinematic properties. This procedure is outlined in Section 4.3. As a second part, an arbitrary number of sensors can be added, which is described in Section 3.5. Once at this stage, the robot can be moved, in our case the MAV will be put into hovering mode, as described in Section 3.3 without state estimation, and in Section 3.4 with a state estimator. The whole

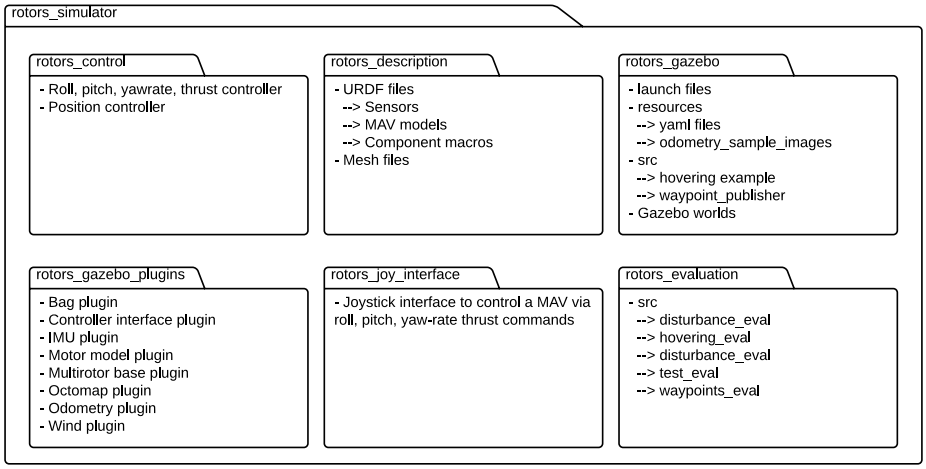

Fig. 7. Structure of the packages contained in the RotorS simulator.

procedure is sketched below:

1. Select your model
   (a) Pick one of the models provided by RotorS, or
   (b) Build your own model as described below in Section 4.3
2. Attach sensors to the MAV
   (a) Use one of the Xacro for the sensors shipped with RotorS, or
   (b) Create your own sensor (look at our plugins in `rotors_gazebo_plugins` for reference[7]), and attach them directly, or create a Xacro, analogously to the Xacros in `component_snippets.xacro`

[7] The official Gazebo sensor tutorials can be accessed on: http://gazebosim.org/tutorials?cat=sensors

3. Add a controller to your MAV
   (a) Start one of the controllers shipped with RotorS, or
   (b) Write your own controller, and launch it
4. Use a state estimator
   (a) Do not run a state estimator, and use the output of the ideal odometry sensor directly as an input to your controller, or
   (b) Use MSF, see Section 3.4 on how to use it in the simulator, or
   (c) Use your own state estimator

3.3 Hovering Example

To check if the setup is working, we want to start with a short example. We will run the simulator with an AscTec Firefly hex-rotor helicopter, running an implementation of a position controller by Lee et al. [5]. Here, we assume that we have a sensor available, which publishes odometry data that can directly be used by a controller. To start the simulation run:

```
$ roslaunch rotors_gazebo mav_hovering_example.launch
```

You will see the hex-rotor helicopter taking off after 5 s, and flying to the point $\boldsymbol{p} = \begin{pmatrix} 0 & 0 & 1 \end{pmatrix}^T$.

In Fig. 3.3 you can see a re-drawn output of `rqt_graph`. This tool gives an overview of all ROS nodes that are running, and the topics on which the nodes are communicating, which is very helpful for debugging.

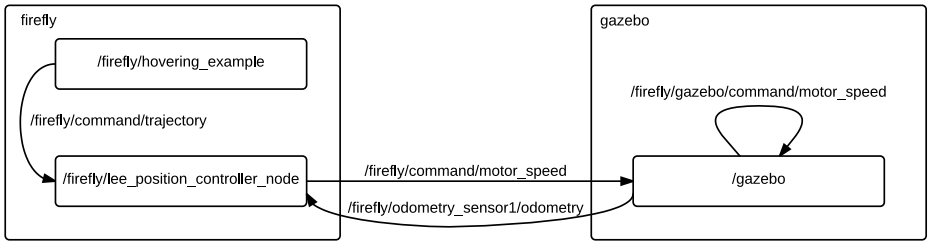

Fig. 8. Graph of ROS nodes and topics of the minimal hovering example with a Firefly. This example consists of a node that sends the initial waypoint, a controller node and the Gazebo simulator (including plugins)

Gazebo is only shown as one ROS node, but internally all the Gazebo plugins are running, such as IMU and individual motors that are mounted on the frame. In this example, a generic odometry sensor is mounted on the Firefly, which publishes the following messages in the `/firefly/odometry_sensor1` namespace:

- *nav_msgs/Odometry* message on the `odometry` topic.
- *geometry_msgs/PoseStamped* message on the `pose` topic.
- *geometry_msgs/PoseWithCovarianceStamped* message on the `pose_with_covariance` topic.

– *geometry_msgs/PointStamped* message on the `position` topic.
– *geometry_msgs/TransformStamped* message on the `transform` topic.

The sensor was added to the Firefly model in the `firefly_generic_odometry_sensor.xacro`-file, which describes the model and gets assigned to the `robot_description` parameter in the launch file.

All the states needed by the position controller, depicted in Fig. 2.2, are contained in the *Odometry* message. Namely, these (states) are the position, orientation, and linear and angular velocity of the MAV. Hence, the position controller running alongside the simulation can directly subscribe to the *Odometry* message. The controller publishes *Actuators* messages, which are read by the Gazebo controller interface and forwarded to the individual motor model plugins.

Commands for the controller are read from *MultiDOFJointTrajectory* messages, which get published by the `hovering_example` node. These messages contain references of poses, and its derivatives. In this example only the position and yaw values of the messages are set. The `hovering_example` node publishes such a message to initiate the take-off maneuver. In addition this node un-pauses the Gazebo physics. *MultiDOFJointTrajectory* messages are used to be compatible with planners, such as MoveIt! [9]. But waypoints can also be published as *PoseStamped* messages.

You can change the position reference of the MAV by sending a *MultiDOFJointTrajectory* message. Usually this is done by a planner or waypoint publisher. For this tutorial, we implemented a simple ROS node, that reads the position and yaw from the command line, translates it into a *MultiDOFJointTrajectory* message and sends it to the controller.

```
$ rosrun rotors_gazebo waypoint_publisher 1 0 1 0 __ns:=
    firefly
```

The first three parameters are the position, followed by the yaw angle in degrees. Because all our nodes are running in the `firefly` namespace, we also need to run the waypoint publisher in that namespace, which is done by the last argument[8].

All the topics that are published in this configuration can be listed with:

```
$ rostopic list
```

and to look at the published data:

```
$ rostopic echo <the_desired_topic>
```

You can exchange the vehicle by setting the `mav_name`-argument, for example, you could launch the simulation with a *Pelican* quad-rotor helicopter.

```
$ roslaunch rotors_gazebo mav_hovering_example.launch mav_name
    :=pelican
```

[8] More on namespaces can be found on: `http://wiki.ros.org/Names`

Currently, these MAVs are available: `asymmetric_quadrotor`, `firefly`, `hummingbird`, and `pelican`. All the control parameters are set in the `lee_controller_<mav_name>.yaml`, and the vehicle parameters used by the controller in `<mav_name>.yaml`.

Note 1. These values do not have to be identical with the values in the model's description. On real systems, the vehicle parameters are usually unknown, and only approximate values can be identified.
Both files are located in the `rotors_gazebo/resources` folder.

3.4 Hovering with State Estimation

On real MAVs there is usually no direct odometry sensor, that gives information about all the states. Instead there is a broad variety of sensors, like GPS and magnetometer, cameras or lasers to do SLAM, or external tracking systems that provide a full 6 DoF pose. In this section, we want to show how to use the MSF package [6] to get the full state from a pose sensor and the IMU.

To run this example you need the `rotors_simulator_demos` package and its dependencies, see Section 3.1. The following command will start all the needed ROS nodes.

```
$ roslaunch rotors_simulator_demos mav_hovering_example_msf.
    launch
```

This time, the MAV is not as stable as in the previous example, and has a small offset at the beginning. The wobbling comes from the simulated noise on the pose sensor and the offset from the IMU biases. After a while the offset will disappear, because the EKF estimates the biases correctly.

Having a look at the re-drawn `rqt_graph` in Fig. 3.4, the following changes from the previous example can be observed: First, the Gazebo odometry plugin only publishes the pose of the MAV and not the full odometry message as before. On real systems, this could be an external tracking system. Second, an additional node is started, the MSF `pose_sensor`, but more on this node later. The controller node now subscribes to the odometry topic from the MSF instead of the odometry from Gazebo. This is done in the launch file, by a topic remapping with the following line:

```
<remap from="odometry" to="msf_core/odometry" />
```

As explained in the previous section, the following node can be used to move the MAV.

```
$ rosrun rotors_gazebo waypoint_publisher <x> <y> <z> <yaw>
    __ns:=firefly
```

The MSF package consists of a core that performs the state propagation based on IMU measurements. For the state update many predefined sensor combinations are available, from which we use the `pose_sensor` in this example. The parameters are loaded from the `msf_simulator.yaml` located in the `rotors_simulator_demos/resources` folder.

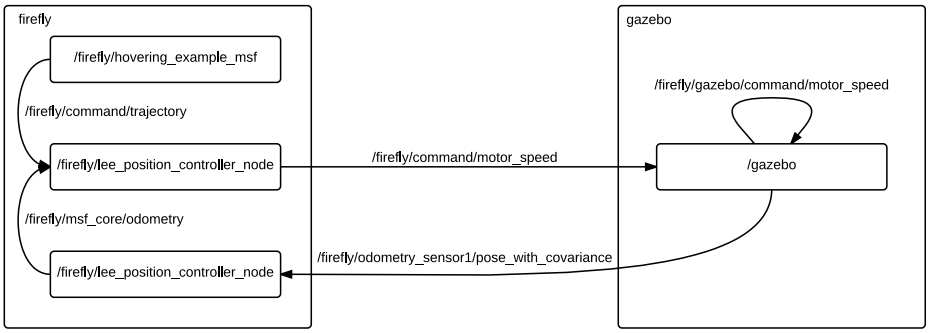

Fig. 9. Graph with ROS nodes and topics of the hovering example with state estimation. This example consists of a node that sends the initial waypoint, a state estimator node, a controller node and the Gazebo simulator (including plugins)

3.5 Mount Sensors

This section explains the different sensors, and how these are mounted on the MAVs. Any sensor can be attached to the vehicle in the corresponding Xacro files. This is done by calling one of the macros, located in the `rotors_description/urdf/component_snippets.xacro` file. The `mav_hoverin_example.launch` file, for instance loads the robot description from the `firefly_generic_odometry_sensor.gazebo` file, which adds an odometry sensor to the Firefly, by calling:

```
<xacro:odometry_plugin_macro (here go the macro properties)>
  <inertia (with its properties) />
  <origin (with its properties) />
</xacro:odometry_plugin_macro>
```

The macro properties are explained in the `component_snippets.xacro`-file. Properties in macros that are preceded by a `*` have to be set as tag-blocks within the opening and closing tags of the macro. The `inertia` and `origin` properties of the odometry macro are such blocks. An inertia block is written as:

```
<inertia ixx="1.0" ixy="0.0" ixz="0.0" iyy="1.0" iyz="0.0" izz
    ="1.0" />
```

Properties present in all of RotorS' sensor macros are:

namespace assigns a ROS namespace to this sensor
parent_link the sensor gets attached to this link (with an offset, relative to this link, specified in the `origin`-block)
some_topic topic name on which the sensor publishes messages

Macro properties can be set, as illustrated for the `parent_link` property below:

```
  <xacro:odometry_plugin_macro
      ...
```

```
      parent_link="base_link"
      ...
      >
    ...
</xacro:odometry_plugin_macro>
```

Note 2. All values of properties are passed as strings, hence, they must be enclosed by quotation marks.

Note 3. In Gazebo links must have a certain weight (at least 10^{-5} kg), otherwise they get omitted by the physics engine. This will result in not finding the links by the plugins.

An overview of the sensors currently used in RotorS, for which Xacros are provided, is given in Table 1[9]. In Fig. 3, a VI-Sensor was added to the Firefly hexacopter.

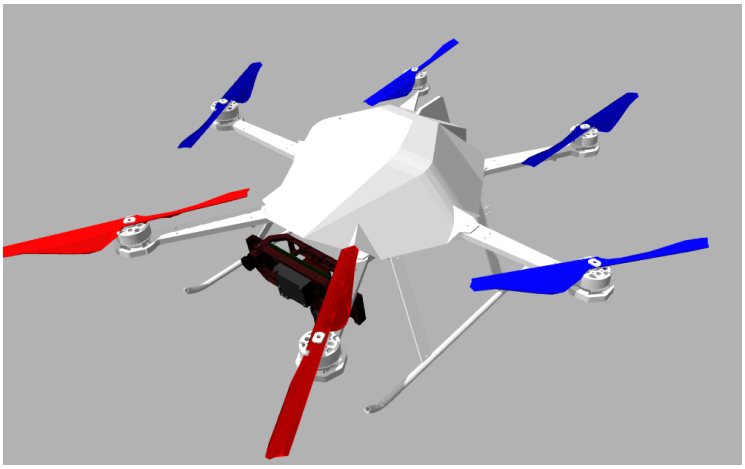

Fig. 10. The Firefly hexacopter with a VI-Sensor mounted on it pointing to the front, and slightly downwards.

3.6 Evaluation

We are particularly interested in tracking the MAV's location to compare it with a user specified position, and how it reacts to disturbances, such as a wind gust. Great interest lies also in checking how quickly an MAV can fly to a certain location, and again how accurately it stays around the specified setpoint. As a first test, we can generate a bag file of a flight with a *Firefly*, by running the hovering example with logging:

[9] Additional Gazebo plugins are listed on: `http://gazebosim.org/tutorials?tut=ros_gzplugins`.

[10] For more information about the VI-sensor: `http://www.skybotix.com/`

Sensor	Description	Xacro-name Gazebo plugin(s)
Camera	A configurable standard ROS camera.	`camera_macro` `libgazebo_ros_camera.so`
IMU	This is an IMU implementation, with zero mean white Gaussian noise and a random walk on all measurements as described in (12) and (13)	`imu_plugin_macro` `librotors_gazebo_imu_plugin.so`
Odometry	Simulate position, pose, and odometry sensors. A binary image can be attached to specify where the sensor works in the world, in order to simulate sensor outages. White pixel: sensor works, black: sensor does not work. It mimics any generic on- or off-board tracking system such as Global Positioning System (GPS), Vicon, etc.	`odometry_plugin_macro` `librotors_gazebo_odometry_plugin.so`

VI-Sensor	This is a stereo camera with an IMU, featuring embedded synchronisation and time-stamping developed at Autonomous Systems Lab (ASL)[10]. In simulation, one can choose to enable images, depth images and point clouds, and ground truth odometry	`vi_sensor_macro`
		`libgazebo_ros_camera.so`
		`librotors_gazebo_odometry_plugin.so`
		`librotors_gazebo_imu_plugin.so`
		`libgazebo_ros_openni_kinect.so`

Table 1. This table gives an overview of the different sensor Xacros provided in the RotorS simulator.

```
$ roslaunch rotors_gazebo mav_hovering_example.launch
    enable_logging:=true
```

The bag files will be stored in the `.ros`-folder in the user's home directory by default. To run the evaluation of the controller:

```
$ rosrun rotors_evaluation hovering_eval.py -b ~/.ros/<
    your_firefly_bag_file>.bag --save_plots True --mav_name
    firefly
```

Note 4. If there is an error about an unindexed bag file, reindex it with:

```
rosbag reindex <bagfile>
```

We get the following output by the script (the evaluation measures the differences to a hovering setpoint of $\boldsymbol{p} = \begin{pmatrix} 0 & 0 & 1 \end{pmatrix}^T$, from 10 s to 20 s):

```
Position RMS error: 0.040 m
Angular velocity RMS error: 0.000 rad/s
```

Additionally, the script will produce three plots showing the position, the position error, and the angular velocities. We evaluate angular velocities to expose overly aggressive-tuned controllers. The plots show that we have a static offset in the z-axis. This is the case since there is no integral part in the controller, and the mass value in the controller is not exactly set to the correct value. As there is no noise on the sensors, the angular velocity RMS error is zero. More realistic conditions would be achieved by adding some noise on the odometry sensor, and then runningh the simulation and evaluation again. The noise parameters of the odometry sensor can be set in the `firefly_generic_odometry_sensor.gazebo`-file. You will get plots like the ones shown in Fig. 4 for the position error, and for the angular velocities in Fig. 4 (these plots might differ, depending on the magnitude of the noise that was set). There are evaluation scripts available

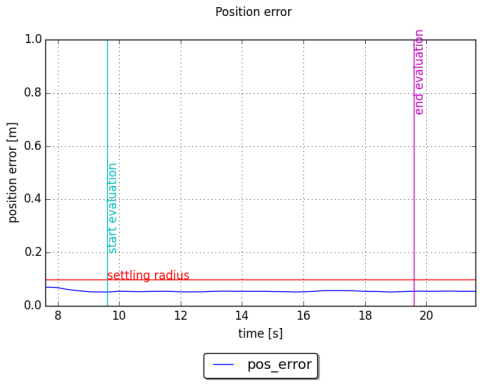

Fig. 11. Position error plot of a hovering sequence with noise on the odometry sensor. Because the controller does not contain an integrator or disturbance estimator, there is a constant offset.

for multiple waypoints: `waypoint_eval.py`, and for evaluating the reaction to external disturbances: `disturbance_eval.py`.

4 Advanced Tutorials

Up to this point we learned how to use the the components provided by RotorS. If the reader is not only interested in using this set of MAVs with some standard sensors and controllers, this section will give more insight on how to develop a custom controller in Section 4.1. In Section 4.3, it is described how to integrate a new MAV, how to write new sensors, and how to work on a state estimator.

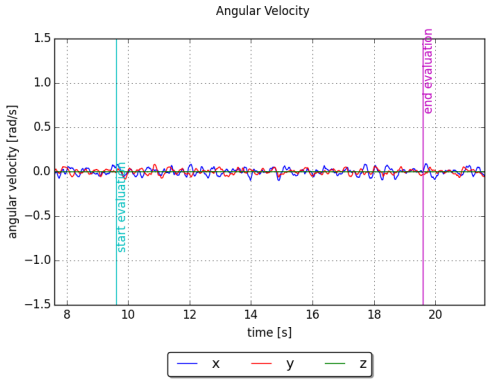

Fig. 12. Angular velocities plot of a hovering sequence with noise on the odometry sensor.

4.1 Developing a Custom Controller

Here, we show how you build a controller for an arbitrary MAV. The message passing is handled by the `gazebo_controller_interface`, and the motor dynamics are incorporated in `gazebo_motor_model`. Hence, the task of developing a controller can be reduced to subscribing to sensor or state estimator messages, reference commands, and publishing *Actuators* messages on the `command/motor_speed` topic. Another option is to run the *RollPitchYawrateThrustController* and build a position controller that publishes *MultiDOFJointTrajectory* messages.

Note 5. Our position controller listens to *MultiDOFJointTrajectory* messages as control input, whereas the roll-pitch-yawrate-thrust controller listens to *RollPitchYawrateThrust* messages. Both listen to odometry messages either from a sensor directly, or from a state estimator.

There are two parts in our design of the controller, the first part handles the parameters and the message passing, this is an executable, a ROS-node. The second part of the controller is a library, which gets loaded by the executable, and does all the computations. We encourage to use one of our controller ROS-nodes as a template, located in the `rotors_control/src/nodes`-folder. It reads the controller parameters from the ROS-parameter server. The parameters are usually set in `yaml`-files, and split up into controller specific parameters and vehicle specific parameters. Through a *launch*-file, the parameters from a `yaml`-file are passed to the ROS-parameter server, by adding the following line between your `<node>`-tags.

```
<rosparam command="load" file="$(find rotors_gazebo)/resource/
    your_parameters_file.yaml" />
```

The vehicle parameter file includes the mass of the MAV, its inertia and rotor configuration. Controller gains are specified in the controller parameters file.

With all the parameters in your controller and the callback methods for the *Odometry* and *MultiDOFJointTrajectory* messages, we can now start writing the actual controller, the controller library. These libraries work without ROS, and hence would be able to run on an MAV directly.

Our controller libraries are located in the `rotors_control/src/library`-folder. Again, we encourage to use one of the libraries as a template to develop your controller.

We have a method, `CalculateRotorVelocities`, which gets called every control iteration, that calculates the required rotor velocities $\boldsymbol{\omega}$ from the controller's input, and information about the current MAV state. As mentioned before, the reference consists of the desired position $\boldsymbol{p_d}$ and its derivatives, and the yaw angle ψ and its derivatives. The input is the content of a *MultiDOFJointTrajectory* message. In our implementation of [5], this method calls other methods, which do the calculations of the desired accelerations $\ddot{\boldsymbol{x}}$, and the angular accelerations $\dot{\boldsymbol{\omega}}$. To wrap it up, if you want to implement your own controller, all that needs to be done is to compute the rotor velocities in the `CalculateRotorVelocities` method, based on the state information of the MAV. Or use cascaded controllers, that is, run our roll-pitch-yawrate-thrust controller and publish *RollPitchYawrateThrust* messages in your controller.

4.2 Tutorial on Xacros

This section should give a short overview on what Xacro[3] is, it can be skipped if the reader is already familiar with the concepts of Xacro. Xacro is an Extensible Markup Language (XML) macro language, which is used to generate more readable and often shorter XML code. In our `rotors_description`-package we use it extensively. First, a very quick introduction to XML, which is built out of tags. A tag is dividing parts of XML code in blocks. These block have opening tags `<tagName>` and closing tags `</tagName>`. Each tag can have any number of properties, which are set as:

```
<someTag property1="value1" property2="all values are passed
    as strings" property3="4.0" />
```

As seen in the line above, if there is no further content between the opening and closing tag, a tag can be closed directly as `<tag />`.

Since XML is solely text based, there are a number of reasons, as can be seen below, for using Xacro. The `xmlns:xacro="http://ros.org/wiki/xacro"` (tag) property needs to be set to the first XML tag, such that the converter recognizes that the current file is of Xacro type. Here we will start with a minimal example on how to set and use Xacro properties. These are particularly useful for repeating values, such as constants or parameters.

```
<foo xmlns:xacro="http://ros.org/wiki/xacro">
  <xacro:property name="pi" value="3.14159265359" />
  <xacro:property name="moment_constant" value="0.016" />
  <bar property1="${moment_constant}">${pi}</bar>
</foo>
```

The code above can be stored to a text file, `test.xacro`, for instance. Then it can be converted to a regular XML file by:

```
$ rosrun xacro xacro.py test.xacro
```

Properties surrounded by $-brackets (`${}`) get replaced in this step, as you can see in the output of the line above. This is usually written to a file, by appending `-o test.xml` to the example above, or piped to the ROS parameter server in a launch file:

```
<param name="robot_description" command="
  $(find xacro)/xacro.py 'test.xacro'"
/>
```

Below the remaining basic Xacro functionalities are explained:

Property Blocks If there is content between the opening and the closing tag of Xacro properties, we call it property blocks. It can be inserted as shown below in the macros part.

```
<xacro:property name="an_origin">
  <origin xyz="0.0 0.0 0.05" rpy="0 0 0" />
</xacro:property>
```

Math Expressions can be used to do basic arithmetic operations, such as: `${(30 + 1) * pi/180}`.

Conditional Blocks can be used to conditionally include code in the output:

```
<xacro:if value="${add_camera}"><...some extra code...></
    xacro:if>
<xacro:unless value="${moment_constant}"><...some extra
    code...></xacro:unless>
```

Rospack commands such as `find`, can be used within $-parentheses `$(find rotors_gazebo)`.

Macros specify your own snippets such as the one shown below to calculate the inertia of a box:

```
<xacro:macro name="box_inertial" params="x y z mass *origin
    ">
  <inertial>
    <mass value="${mass_box}" />
    <xacro:insert_block name="origin" />
    <inertia ixx="${0.0833333 * mass * (y*y + z*z)}" ixy="
    0.0" ixz="0.0"
      iyy="${0.0833333 * mass * (x*x + z*z)}" iyz="0.0"
      izz="${0.0833333 * mass * (x*x + y*y)}" />
  </inertial>
</xacro:macro>
```

To define a macro, the `<xacro:macro>` tag is used, where you specify its name and an arbitrary number of parameters. Parameters starting with an asterisk (*), denote property blocks, which are read as the content between

the opening and closing tag of the macro. The above just defines a macro, the code gets only placed in your output file if you call the macro with its parameters as:

```
<xacro:box_inertial x="0.2" y="0.4" z="0.1">
  <origin xyz="0.1 0.2 0.05" rpy="0 0 0" />
</xacro:box_inertial>
```

Include other Xacro files with:

```
<xacro:include filename="../other.xacro" />
```

4.3 Assembling a Model

As you are able to fly with the MAVs that are shipped with RotorS, we want to encourage you to bring your own MAV into RotorS. In this section, we describe the procedure of how to get the geometry of your robot, and the locations of the actuators into RotorS. In the next section, we describe how custom sensors can be written as a Gazebo plugin. For the description of the robot we are using the Unified Robot Description Format (URDF), with Xacro, which is briefly explained in the previous section.

Note 6. Gazebo has its own format to describe robots, objects, and the environment, called Simulation Description Format (SDF). We are sticking with URDF here, as this format can be displayed in the ROS 3D Robot Visualizer (rviz), and all the SDF-specific properties can be added by putting them in a `<gazebo>`-block. Internally Gazebo converts the URDF-files to SDF-files.

If you have a specific robot that you want to use in your simulation, this procedure is quite straight forward. You have to check which parts of your robot are fixed, or rigid bodies, in URDF these parts are called *links*, and connect these links; connections are called *joints*. In Fig. 6, the different links and joints of an example quad-rotor helicopter are shown. There are a number of different joints as can be seen on `http://wiki.ros.org/urdf/XML/joint`. Each joint has a *parent* and a *child* link. For our application we only use three types:

continuous joints are hinge joints without any mechanical limits, such as our rotors.

fixed joints are connecting two links rigidly, such that there is no movement possible. These joints get removed by Gazebo's physics engine, that means, no information can be gathered about these joints and its child links in Gazebo (plugins).

revolute joints are the same as continuous joints, but have a lower and upper limit, which limits their turning angle. We are using these joints with both limits set to zero, as fixed joints, if the child link needs to be accessible from a plugin.

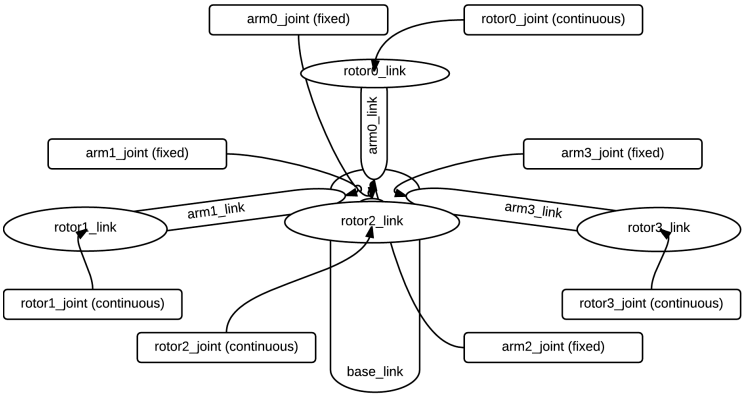

Fig. 13. A draft of a multirotor helicopter with four non-symmetrical aligned rotors. All the joints and links are named to get an overview of the necessary components in the Xacro files.

As we go on with this section, we want to develop a model of a conceptual quad-rotor helicopter, as depicted in Fig. 6. This quad-rotor helicopter model has four rotors, of which three are distributed equally on the edge of a circle with radius l, while the remaining rotor is placed in the back of the vehicle with an arm of length $l/3$. All the rotors are of the same dimension.

We can now start to build our model in a URDF file. Every robot that is built with URDF needs a `<robot>`-tag in the beginning of the file, with a unique name. Within the opening- and the closing tag, the robot will be assembled. We start by adding the base of the robot, there is a macro for doing this.

```
<robot name="asymmetric_quadrotor" xmlns:xacro="http://ros.org
    /wiki/xacro">
  <xacro:multirotor_base_macro
    robot_namespace="${namespace}"
    mass="${mass}"
    body_width="${body_width}"
    body_height="${body_height}"
    use_mesh_file="${use_mesh_file}"
    mesh_file="">
     <xacro:insert_block name="body_inertia" />
  </xacro:multirotor_base_macro>
</robot>
```

The listing above needs some Xacro-properties, such as `namespace` and `mass` to be set, this is explained in Section 4.2.

One would continue adding the arms and the rotors to the base link by connecting them with joints – we will show this for one arm and rotor. If you have a mesh file of your multi-rotor helicopter with arms, and know its inertia (from the body including the arms), you can omit this step, and attach the rotors directly to the base link. For the other MAVs that are provided with RotorS, the rotors are directly attached to the base. The reader is encouraged to use the macros provided in the `component_snippets.xacro` and `multirotor_base.xacro` files.

```
<link name="arm1_link">
  <xacro:box_inertial x="${arm_length}" y="0.03" z="0.01"
    mass_box="${mass_arm}" />
  <visual>
    <origin xyz="${arm_length/2} 0 0" rpy="0 0 0" />
    <geometry>
      <box size="${arm_length} 0.03 0.01" />
    </geometry>
  </visual>
  <collision>
    <origin xyz="${arm_length/2} 0 0" rpy="0 0 0" />
    <geometry>
      <box size="${arm_length} 0.03 0.01" />
    </geometry>
  </collision>
</link>
<joint name="arm1_joint" type="fixed">
  <origin xyz="0 0 0" rpy="0 0 ${2*pi/3}" />
  <parent link="base_link" />
  <child link="arm1_link" />
</joint>
<xacro:vertical_rotor
  robot_namespace="${namespace}"
  suffix="1"
  direction="cw"
  motor_constant="${motor_constant}"
  moment_constant="${moment_constant}"
  parent="arm1_link"
  mass_rotor="${mass_rotor}"
  radius_rotor="${radius_rotor}"
  time_constant_up="${time_constant_up}"
  time_constant_down="${time_constant_down}"
  max_rot_velocity="${max_rot_velocity}"
  motor_number="1"
  rotor_drag_coefficient="${rotor_drag_coefficient}"
  rolling_moment_coefficient="${rolling_moment_coefficient}"
  color="Blue"
  use_own_mesh="false"
  mesh="">
  <origin xyz="${arm_length} 0 ${rotor_offset_top}" rpy="0 0 0
    " />
  <xacro:insert_block name="rotor_inertia" />
</xacro:vertical_rotor>
```

Note 7. Since you are using a very similar snippet for all four arms, it is a good idea to create a macro, which could be called `arm_with_rotor`. It would take the arm length, an angle, the motor number, and the mass as parameters.

You can test your current configuration by either just looking at the output of:

```
$ rosrun xacro xacro.py <xacro_file> > <urdf_file>
$ check_urdf <urdf_file> # You need liburdfdom-tools installed
    for this.
```

or by running our `mav_empty_world.launch` file, which spawns your robot in an empty Gazebo world:

```
$ roslaunch rotors_gazebo mav_empty_world.launch mav_name:=
   asymmetric_quadrotor
```

Note 8. To use this command your file should be called `asymmetric_quadrotor.xacro` and be placed in the `rotors_description/urdf`-folder. Additionally you need to have a file called `asymmetric_quadrotor_base.xacro`, which includes the above file. We seperated the geometrical properties from the sensors and controllers. Sensors and controllers should be placed in the second file.

4.4 Creating Custom Sensors

To make your robot perceive the environment and its own motion with respect to the inertial frame, you need to add sensors to your model. We already discussed how to add available sensors in Section 3.5. In this section, we focus on how to write a Gazebo plugin for a new sensor.

Note 9. Before you create your own plugin, you should make sure that there is no sensor available, which satisfies your requirements. For instance, a laser sensor can be used to mimic an ultrasonic sensor, by just using one ray. This of course is not very accurate, as ultrasonic sensors usually measure the shortest distance to an object in a cone-like shape. But, depending on your application, a measurement along one single ray might also be enough.

In the remainder of this section, we show how one would conceptionally proceed with building a wind sensor. A wind sensor usually measures the airspeed $\boldsymbol{v}_{air}$ on an MAV. This airspeed is the difference between the wind speed $\boldsymbol{v}_{wind}$ and the current velocity $\boldsymbol{v}$ of the MAV.

$$\boldsymbol{v}_{air} = \boldsymbol{v}_{wind} - \boldsymbol{v} \tag{19}$$

To bring this sensor into the simulation, you would start by creating a *Model-Plugin*, as described on: `http://gazebosim.org/tutorials?tut=plugins_model&cat=write_plugin`. You would create a subscriber to the `wind`-topic, and store the pointer to the link, to which the plugin got attached to. Additionally, you would create a publisher on your desired topic, `air_speed`, for instance. All of this can be done in the `Load` method of the plugin. In the `OnUpdate` method, which gets called every single simulation iteration, you are then getting the link's velocity in the world frame by

```
link_->GetWorldLinearVel()
```

and perform the calculation described in (19). You can either publish the calculated value directly, or add some additional noise. Take a look at the *GazeboImuPlugin* and the *GazeboOdometryPlugin* to get an idea how to add noise to your calculated value. Of course, you can also reduce the publishing frequency.

Note 10. Since on a lot of aircraft, this sensor is measuring the difference between the static pressure and the stagnation pressure due to inflowing air in a pitot tube [12], it is arguable if the above sensor design makes sense. To make this airspeed sensor more realistic, the example above could be modified by, for example, only taking the wind and MAV velocities in a single direction in the MAV's body frame, and then transforming them back into the world frame.

5 Using RotorS for Higher Level Tasks

One of the biggest advantages of the RotorS simulator is, that it comes with a fully functional trajectory tracking controller. This allows implementing higher level tasks like collision avoidance or path planning without having to implement state estimation or control first. Also in simulation, you have access to perfect ground truth, which is usually hard to get on real systems. Once the algorithms are working in simulation, the changes necessary when transitioning to real world systems are typically small. In this section we want to give some ideas, how two example problems can be solved using the simulator.

5.1 Collision Avoidance

A commonly used strategy to solve collision avoidance on MAVs is to down-project the environment onto a 2 D ground plane and use the ros navigation stack. To tackle this problem, an appropriate sensor needs to be mounted on the MAV, to get an estimate of the surrounding. Gazebo provides plugins for 2 D laser scanners like the Hokuyo, that could potentially be mounted on a real MAV. This approach has the big disadvantage of reducing the operating space of the MAV to a plane at a constant height.

For full 3 D collision avoidance, front looking depth cameras are a good starting point, like the Kinect-sensor which is already implemented in Gazebo. These sensors are light-weight and give rich information about the surrounding. Another possibility would be to mount cameras and use structure from motion in the monocular case, or a stereo camera to calculate a disparity image. One possible approach to perform 3 D collision avoidance is to do it directly on the disparity images as proposed in [8].

5.2 Path Planning

In the RotorS simulator, we prepared an example environment with a power-plant and a waypoint publisher to test different planning algorithms. The waypoints are chosen such that the straight line solution is in collision, and the MAV

is going to crash into the wall without a planner. A file with the octree representation, as described in Section 2.4, of the power-plant can be found in the `rotors_gazebo/resources` folder and can be used with the `octomap_server` package. The octree representation allows for efficient collision checking in the planner and is well suited for 3 D environments. To run the example, use the following command.

```
$ roslaunch rotors_gazebo
    mav_powerplant_with_waypoint_publisher.launch
```

ROS provides a big collection of planning algorithms in the MoveIt! package [9], that can be adapted to 3 D path planning in static environments.

6 Transfer to a Real MAV

Having a simulation of an MAV is of course nice, but it raises the question, how well it represents the real world and how easy the transition from simulation to real MAVs is. During the development of this simulator, a lot of effort was put into keeping the structure of the simulator as close as possible to the real system. In the best case, this means just switching the simulation environment with a ROS node that communicates with the hardware. The simulator should be a tool that enables the development of algorithms, to be deployed on a real MAV later on. To make the transition of the code from the simulator to the code running on the actual hardware as simple as possible, the interface is designed in a way which mimics most of the interfaces on the real systems. Not all messages are replicated in the simulator, but one can easily extend the simulator with models of a battery or other parts which are currently not implemented. One of the biggest challenges on real platforms are the changing delays and the resulting non deterministic order of measurements. This leads to complex message queues, tedious time delay estimation and delay compensation.

To give an impression on how well the simulator replicates the real MAV, we are using the same controller gains in the simulation as on the real Hummingbird and Firefly MAVs[11]. Here we have to mention that in general we send attitude commands to the low level controller of the real helicopters, and use the manufacturer's low level attitude controller to compute motor commands.

7 Conclusion

In this chapter, we presented an overview of the necessary background to understand the basics for autonomous flights of MAVs. We then explained step-by-step how to run the simulation with existing models, with ideal conditions first, and then how to use it with more realistic sensors, and a state estimator. As advanced tutorials, we presented instructions on how to implement a custom quad-rotor

[11] A video of a Firefly following a path in real world and in the RotorS simulator can be seen on `http://youtu.be/3cGFmssjNy8`.

helicopter, including guidance to create own controllers and sensors. We believe that this framework will be very helpful for both research and teaching, as it gives an easy start into the topic. Interfaces are compatible, and the simulated dynamics are close to the real platforms, such that an easy transfer from the simulation to the real world is possible.

8 Authors' Biographies

Fadri Furrer is a PhD student at the ASL at ETH Zurich since 2015. He received his MSc degree in electrical engineering from the Swiss Federal Institute of Technology, Zurich, in 2011. His research interests are in simulation, object reconstruction and recognition from images and point clouds.

Michael Burri is a PhD student at the ASL at ETH Zurich since 2013. He received his MSc degree in robotics and control from the Swiss Federal Institute of Technology, Zurich, in 2011. His research interests are in control, state estimation, and planning for micro aerial vehicles.

Markus Achtelik received his Diploma degree in Electrical Engineering and Information Technology from TU München in 2009. He finished his PhD in 2014 at the ASL at ETH Zurich, and currently works as Postdoc at ASL on control, state estimation and planning, with the goal of enabling autonomous maneuvers for MAVs, using an IMU and onboard camera(s) as main sensors. Since 2014, he is challenge leader of the "plant servicing and inspection challenge" within the European Robotics Challenges.

Roland Siegwart (born in 1959) is a professor for autonomous mobile robots at ETH Zurich. He studied mechanical engineering at ETH, brought up a spin-off company, spent ten years as professor at EPFL, was vice president of ETH Zurich and held visiting positions at Stanford University and NASA Ames. He is and was the coordinator of multiple European projects and co-founder of half a dozen spin-off companies. He is recipient of the IEEE RAS Inaba Technical Award, IEEE Fellow and officer of the International Federation of Robotics Research (IFRR). He is in the editorial board of multiple journals in robotics and was a general chair of several conferences in robotics including IROS 2002, AIM 2007, FSR 2007 and ISRR 2009. His interests are in the design and navigation of wheeled, walking and flying robots operating in complex and highly dynamical environments.

Bibliography

[1] Markus W. Achtelik. *Advanced Closed Loop Visual Navigation for Micro Aerial Vehicles*. PhD thesis, ETH Zurich, 2014.

[2] Tim Field, Jeremy Leibs, and James Bowman. Rosbag, 2015. URL `http://wiki.ros.org/rosbag`. [Online; accessed 27-March-2015].

[3] Stuart Glaser and William Woodall. Xacro, 2015. URL `http://wiki.ros.org/xacro`. [Online; accessed 27-March-2015].

[4] Armin Hornung, Kai M. Wurm, Maren Bennewitz, Cyrill Stachniss, and Wolfram Burgard. OctoMap: An efficient probabilistic 3D mapping framework based on octrees. *Autonomous Robots*, 2013. Software available at `http://octomap.github.com`.

[5] Taeyoung Lee, M Leoky, and N Harris McClamroch. Geometric tracking control of a quadrotor UAV on SE (3). In *Decision and Control (CDC), 2010 49th IEEE Conference on*, pages 5420–5425. IEEE, 2010.

[6] Simon Lynen, Markus W Achtelik, Stephan Weiss, Margarita Chli, and Roland Siegwart. A robust and modular multi-sensor fusion approach applied to mav navigation. In *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, pages 3923–3929. IEEE, 2013.

[7] P. Martin and E. Salaun. The true role of accelerometer feedback in quadrotor control. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 1623–1629, May 2010.

[8] Larry Matthies, Roland Brockers, Yoshiaki Kuwata, and Stephan Weiss. Stereo vision-based obstacle avoidance for micro air vehicles using disparity space. In *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, pages 3242–3249. IEEE, 2014.

[9] SIoan A. Sucan and Sachin Chitta. Moveit!, 2015. URL `http://moveit.ros.org`. [Online; accessed 10-August-2015].

[10] N. Trawny and S. I. Roumeliotis. Indirect Kalman filter for 3D attitude estimation. Technical Report 2005-002, University of Minnesota, Dept. of Computer Science and Engineering, 2005.

[11] Stephan Weiss. *Vision based navigation for micro helicopters*. PhD thesis, ETH Zurich, 2012.

[12] Wikipedia. Airspeed indicator, 2015. URL `http://en.wikipedia.org/wiki/Airspeed_indicator`. [Online; accessed 27-March-2015].