Dynamic Planning Pipeline for Indoor Inspection Flights

Raphael Hagmanns

Vision and Fusion Laboratory
Institute for Anthropomatics
Karlsruhe Institute of Technology (KIT), Germany
raphael.hagmanns@kit.edu

Abstract

In this report a basic pipeline for planning and operating an indoor drone flight is presented and evaluated in detail. We introduce the structure and interface considerations of a Planner Manager enabling autonomous indoor flights. The interaction routines of different planners are introduced in detail before we evaluate the system in both simulation and real test flights. We show that the system is capable of managing the typical building blocks of a mobile robotics system. Most of the components can be swapped easily to allow for rapid prototyping without the need to rework the whole pipeline.

1 Introduction

UAVs (Unmanned Aerial Vehicles) have become increasingly important in the last couple of years. Both industrial and consumer sectors require UAVs to be equipped with more and more intelligent and autonomous behavior. This includes automatic obstacle avoidance, efficient path planning as well as an environment sensing with varying imaging sensors. While some of these

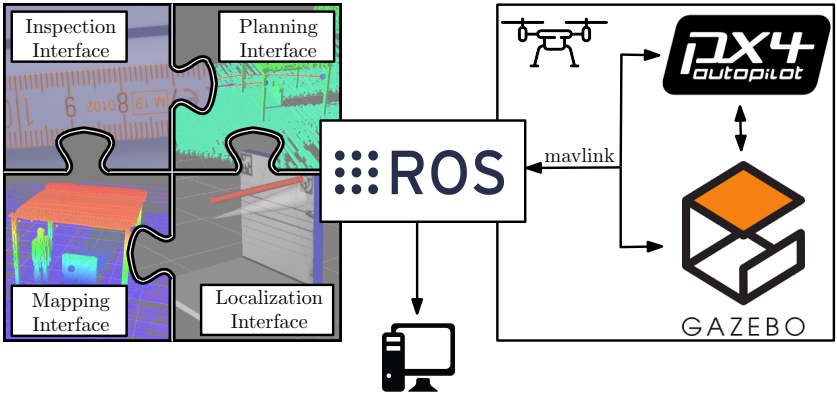


Figure 1.1: Overview of different pipeline building blocks.

functionalities are included in flight control software [1],[2], they rarely meet the requirements of the research community. Prototype driven development of new algorithms, reproducible testing in simulation as well as in practical experiments and fine graded control over all system variables often require researchers to create their own entire software stack in order to run a specific algorithm. Due to the novelty of the research area, many of the basic requirements as *feedback control*, *trajectory generation* or *state estimation* are still very active research topics [2]. Without access to the whole deployment stack, it is often difficult to run a specific module or to reproduce the results. On the other hand, if the stack is available, extensive modifications are often required in order to test it against alternative implementations.

Therefore, we propose a framework which allows for a loose coupling of several main building blocks of a mobile robot autonomy stack. These building blocks are depicted in 1.1. We target autonomous indoor inspection flights with some specific inspection target, which comes with the additional difficulty of accurate state estimation. Usually some kind of visual inertial odometry must be provided

[1] `https://www.dji.com/de/guidance`
[2] `https://docs.px4.io/v1.9.0/en/advanced_features/`

to allow a drone to fly in such GPS-denied environments. However, to support different environments, we don't want to restrict the platform to a specific source of odometry as explained in the localization interface (see Section 3.1). The generation of a obstacle free trajectory requires a planner to interact with some kind of map representation, this interaction is described in further detail in Section 3.1. We then continue to describe the main control routine in further detail in Section 3.2. Finally, we include some practical considerations in Section 4 and present an evaluation of the controller pipeline and the interfaced algorithms.

The Robot Operating System (ROS) serves as main link between the different components in our system as visualized in Figure 1.1. It is widespread in the robotic community and comes with a powerful toolset consisting of sensor drivers, simulation frameworks (Gazebo) and a selection of state-of-the-art perception algorithms. [3],[4] The node based architecture together with a publisher / subscriber information scheme allows for a semantic abstraction of single building blocks into reusable, modular software packages. Our platform is intended to be deployed on drones running the px4 software stack.[5] Px4 is an open source autopilot working on various hardware platforms. It comes with *SITL* and *HITL* features and supports standardized communication via the mavlink communication protocol. Additionally it allows to publish and retrieve odometry information to and from ros nodes via *mavlink*[6], which makes it the most popular platform in the research community.

2 Related Work

There exist few software stacks containing a high level controller which is able to plan and execute a given waypoint trajectory. Most of the available dynamic UAV planners solely rely on a specific underlying map representation and require additional work to run in a real environment.

[3] `https://ros.org/`
[4] `https://gazebosim.org/`
[5] `https://px4.io/`
[6] `https://mavlink.io/`

In [12], Schmid et al. present a generic planning framework with the focus on *active* planning. While the planning module itself is highly configurable, the underlying map representation must be Voxblox [9] and the calculation of the exploration gain of specific viewpoints is deeply interlinked with the raycasting for volumetric map building. The planner is working in the *RotorS* [4] environment, which on of the most common simulation frameworks for UAVs. However, the framework is not meant to be used for flights in practice. *Fuel* [15] is another explorative path planner based on a *frontier information structure* maintaining a tree of already explored paths. They use a hierarchical planner structure which is able to perform global and local planning. It however lacks the possibility to replace single parts of the planner and does not come with integrated controlling capabilities.

The flight controller software stack *px4* itself provides different modes suitable for simple waypoint navigation. This comes with the advantage of a strong link to the flight controlling mechanisms. Theses modes are only provided for GPS based flights and the support for local planning is limited. Within the px4-universe, the recently open-sourced frameworks XTDrone [14] and MRS UAV System [2] offer a variety of functions, platforms and sensors and are highly extensible by providing a plugin based architecture. The MRS UAV System even allows to swap between different odometry sources, trackers and controllers mid-flight making it an advanced platform for carrying out research experiments in simulation and real world scenarios. Due to the sheer size and complexity of the platforms, it is not a trivial task to adapt or exchange single components in the systems.

Our proposed architecture does not aim on providing a planner which is on pair with state-of-the-art planners but rather allow the utilization of different modules with unified interfaces and minimal overhead.

3 Pipeline

In the following, we briefly describe our pipeline architecture and discuss the planning routines in greater detail. The main building blocks of the pipeline are depicted in Figure 3.2. Figure 3.1 introduces the notation of the drone's reference

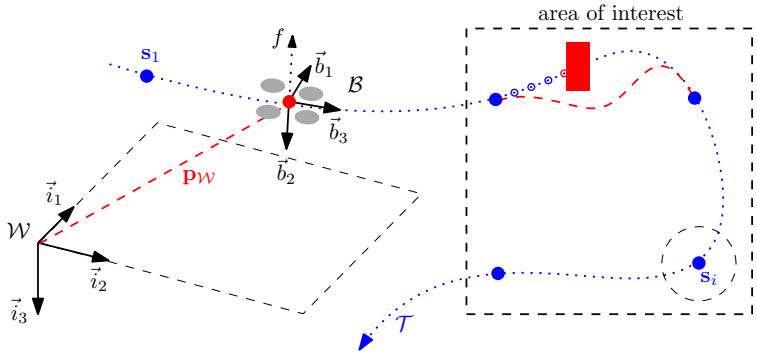


Figure 3.1: UAV System Overview. Similar to [7], the UAV is represented through its body frame $\mathcal{B} = \{\vec{b}_1, \vec{b}_2, \vec{b}_3\}$ originated in the center of mass (red dot) with the combined thrust of the four rotors pointing in $-\vec{b}_2$ direction. The UAVs position estimate is given through $\mathbf{p}_\mathcal{W}$, w.r.t. the reference world frame $\mathcal{W} = \{\vec{i}_1, \vec{i}_2, \vec{i}_3\}$. The drone is tracking the given trajectory $\mathcal{T} = \mathbf{s}_1, \ldots, \mathbf{s}_n$, while dynamically avoiding the obstacle marked in red (cf. Section 3.1). $\mathcal{T}$ is calculated to be a dynamically feasible trajectory, guaranteed to contain the blue inspection points unless they lie within an obstacle. The circle around $\mathbf{s}_i$ depicts the *acceptance radius* for the respective waypoint.

pose and frames. As input we take a list of setpoints $\mathcal{T} = \mathbf{s}_1, \ldots, \mathbf{s}_n$ given as coordinates in a specific reference system $\mathcal{W}$, where each setpoint $\mathbf{s}_i = [\mathbf{t}, \psi]^\top$ consists of a position $\mathbf{t} = (x, y, z)^\top$ and a heading ψ. This input can either be user defined or the result of some static planning routine as explained in Section 3.1. Final output of the pipeline are the lowlevel *setpoint* commands which then serve as input for the flight controller.

3.1 Architecture

The controller handles different components which now will be explained in further detail. Most of the building blocks follow a plugin architecture so that they can easily replaced with alternatives as long as they are implementing the same interface.

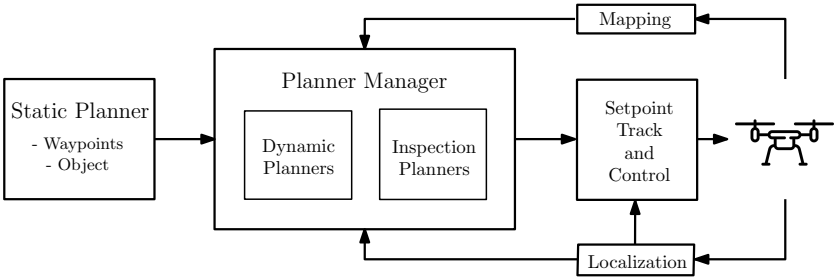


Figure 3.2: Planning Blocks for a Pipeline. An initial trajectory can be provided by a static planner or a waypoint list. The Planner Manager handles different kinds of subplanners, which all use an interface to an underlying map representation and the current position estimate. The "Setpoint Track & Control" block receives the output from the localization system and provides inputs to the low-level flight controller. For a more detailed description of the building blocks, see Section 3.1.

Static Planner For some inspection tasks, it can be useful to pre-generate an inspection trajectory with the goal of optimal inspection coverage while reducing path length and uncertainty at the targeted measuring points at the same time. To support such a pre-computation using a triangle mesh as input, we adopted a version of the Structural Inspection Planner [1] and abstracted it into a single "Static Planner" node. Output of this procedure is a set of optimized viewpoints optimizing the criteria stated above. For details we refer to the original publication in [1].

Localization Interface We don't target a specific localization system within our pipeline. We require some source of odometry to provide an estimate of the pose $\mathbf{p}_{\mathcal{W}} = (\mathbf{t}, \mathbf{R})$ of the vehicle. The typical source of such odometry is the GPS / IMU fusion provided by the px4 flight controller utilizing an Extended Kalman Filter. Other sources of position estimates such as Visual Odometry (cf. Section 4) system or a full SLAM approaches might be used as well. The position estimate is provided to the high-level planner manager as well as to the *Setpoint Control* block.

Setpoint Track and Control This building block generates the commands for the low level flight controller as output. It is setup to either use the flight controller internal position controller by providing the current target

setpoint $\mathbf{s} = [\mathbf{t}, \psi]$ at a rate of 50hz. It additionally supports the utilization of a $\mathbf{SE}(3)$ geometric controller as described in [7] to support more aggressive maneovers. We refer to that publication [7] for a detailed description of the assumed UAV dynamic model. In that mode, output will be provided as *attitude rate* commands right into the *attitude controller* of the flight controller. However, using a geometric controller requires a smooth and feasible input trajectory [7]. Following the approach by Richter, Bry, and Roy [11], we consider the construction of such a piecewise polynomial trajectory out of a set of waypoints to be a linear optimization problem and solve it using an unconstraint linear solver. The implementation in [3] provides additional methods to check for the trajectory feasibility.

Mapping Interface Typically, planners work on an underlaying map representation to account for obstacles. This representation may either be derived from an existing environment or it can be constructed dynamically using the current position estimate $\mathbf{p}_{\mathcal{W}}$ together with some sort of depth sensor information. The pipeline does not rely on a specific type of such representation as long as it supports obstacle-queries for a bounding volume $\mathbf{b} = (\mathbf{t}, \mathbf{d}) \in \mathbb{R}^{2\times 3}$ of the size $\mathbf{d}$. Most of the available volumetric map representations allow for such an operation, in particular we implemented the interface for some of the most popular representations: OctoMap [6] as efficient Octree based volumetric map; Voxblox, a *Truncated Signed Distance Field* based representation [9] and EWOK [13], a highly efficient voxel representation based on a local ringbuffer.

Constraint Planner Sometimes it is useful to constrain the UAV trajectory to a specific operation area. While this could be modeled similar to obstacles in the underlying map representation, we decided to abstract this operation into a different planner, which can reject specific points outside of the defined areas and re-trigger the trajectory generation process with additional constraints. It can also be used to provide warnings or trigger *safety actions* if a no-fly zone is approached.

Dynamic Planner With an underlaying map representation in place, a dynamic planner can now be put into use to dynamically avoid obstacles during

execution time. We leverage the *Open Motion Planning Library* (OMPL) [7] as default framework for local planning as it enables us to compare different kinds of planners using the same interface. In our default configuration, we use a *Informed* RRT* Planner [5], which has proven to outperform the classical RRT* in terms of both convergence rate and solution quality. Each state in the planners search space is being checked for potential collision using an approximate *hit bounding volume* of the vehicle. The resulting trajectory is post-processed using a path simplification and smoothing routine.

Inspection Planner As we are targeting object inspection we introduce another module, which is solely responsible for planning low-level inspection routines. This allows us, for example, to trigger a hover action when we reach a point of interest. We can also use it to perform a static maneuver in order to generate multiple views of a target inspection point. When carrying out an object measurement task, these measures help to a reduce the measurement uncertainty, which otherwise would be mainly influenced by shutter and motion blur effects. When activated during hovering the Inspection Planner may also target a specific distance to an inspection object and overrule the safety distances maintained by the Dynamic Planner. This can prove useful if an inspection unit needs a specific working distance to perform a specific task.

3.2 Planning Procedure

We now describe the interaction of the modules introduced in Section 3.1 in further detail. To ensure a high robustness of the planning system, we employ a multithreaded structure. The *Setpoint Track and Control* block is publishing commands with a high availability while the replanning procedure is running at a lower frequency and the cascading planner structure is invoked on demand. Algorithm 3.1 explains the concept of the *horizon*-based dynamic planning procedure. Multiple planners may be used in a *Planner Manager* $\mathcal{P}$, their

[7] https://ompl.kavrakilab.org/

configurations or even the whole planner may be replaced at runtime. For each update, the previously generated smooth trajectory $\mathcal{T}$ is being evaluated for feature waypoints up to the horizon h (line 7) and stored in a planning queue Q (lines 1-3). If the evaluated position lies within the *acceptance radius* of the currently targeted inspection point, we mark it as reached (line 8) and set a new goal. All registered planners now *check* the generated points (lines 9-13). The *check* depends on the nature of the planner, it typically is an obstacle query against the *Mapping Interface*. Within the check, planners can change the *state* of the input points to mark them for *replanning, inspection*, etc. When all points are marked, the actions are derived (line 14) using an asynchronous function call. For unplanned points, the configured planners are then invoked to generate the path changes with respect to their planning target (lines 17-24). A list of constraints is shared between all the planners. Finally, the waypoint list Q is updated (line 25).

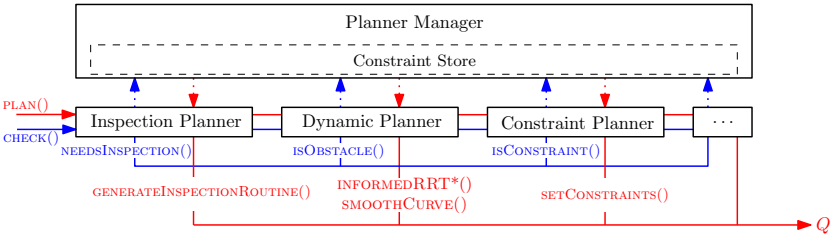


Figure 3.3: Scheme of planner invokations in Planner Manager. Multiple planners can be attached to the Planner Manager, they need to implement *plan* and *check* interfaces. The blue flow illustrates the check functionality for a point. A dynamic Planner would check for free space necessary to approach the point, while a constraint planner would check for user defined constraints preventing this point to be a target. The red flow illustrates the generation of actions to resolve points which need replanning. A dynamic planner might invoke some RRT* routine while an inspection planner generates a custom inspection trajectory.

Since the order of invocation within the Planner Manager is crucial, different priorities can be assigned to the planners. Typically, a low priority planner such as the Inspection Planner should be invoked first, followed by the Dynamic Planner with collision avoidance capabilities. Finally, a Constraint Planner may reject some of the generated points and trigger another iteration of planning.

Algorithm 3.1 Structure of the Planner Controller Routine

Input: Waypoints $\{s_1, \ldots, s_n\}$

$\mathcal{T} \leftarrow$ smooth trajectory through $s_1, \ldots, s_n$ by solving unconstraint QP [11]
$\mathcal{P} \leftarrow$ Manager{ConstraintPlanner, RRTPlanner, InspectionPlanner}
$Q \leftarrow$ sampled waypoints from $\mathcal{T}$ up to horizon h
function UPDATEHORIZON // 5hz rate
    $t \leftarrow$ current time
    Q.prune()
    Q.update($\mathcal{T}(t), \mathcal{T}(t + \Delta), \ldots, \mathcal{T}(t + h\Delta)$)
    if $\mathcal{T}(t) \in$ acceptanceRadius(s_t) **then** s_t.reached $\leftarrow$ **true**
    for all $q \in Q$ with q.state $=$ wp_state::future **do**
        for all $p : \mathcal{P}$ **do**
            q.planning_state $\leftarrow$ p.check(q, t)
        end for
    end for
    generateActions() **async**
end function

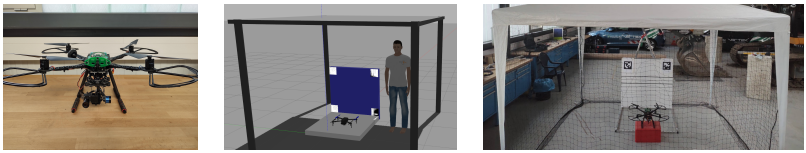
function GENERATEACTIONS
    V $\leftarrow$ Q.rangeView($q =>$ q.state $\neq$ planning_state::planned)
    $\mathcal{C} \leftarrow$ collectConstrains(V)
    for all $p : \mathcal{P}$ **do**
        if V contains unplanned points **then**
            $V_p \leftarrow$ p.plan(Q.begin(), V.begin(), $s_t, \mathcal{C}$)
            V.update(V_p)
        end if
    end for
    Q.update(V)
end function

Output: Collision free setpoints in Q

Safety Actions may be triggered in the *updateHorizon* procedure. The invocation tree of three exemplary planners managed by a common manager is visualized in 3.3.

4 Experimental Evaluation

We verify the functionality of the proposed system in both simulation and practical demonstrations. We develop the pipeline with focus on cross-platform operability in order to support different *companion computers*. We equip both the virtual model and the drone used for the practical experiments with a stereo setup (720px $\times$ 480px per lens) for depth perception as well as a $4k$-RGB sensor and a forward facing distance sensor. In addition, we use a 9-axis IMU (3-axis accelerometer, 3-axis gyroscope and 3-axis compass) as well as barometer. We allow to set user-defined values for IMU noise parameters as well as camera calibration parameters and forward the values to the respective modules.



(a) UAV (b) Simulation (c) Real Environment

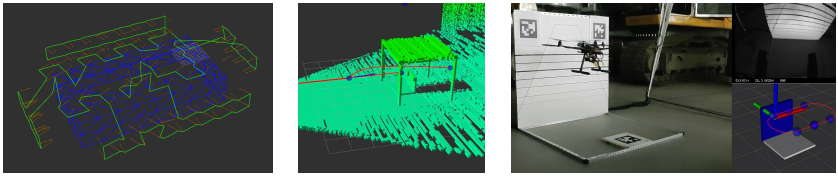
Figure 4.1: Planning tests can be performed in simulation as well as in the real environment.

4.1 In Simulation

We used the px4 *Software-In-The-Loop* component to run experiments with Gazebo as simulator. We recreated the real environment in simulation to test the localization, planning and mapping procedures (cf. 4.1(b)). Even if Gazebo is not capable of rendering photorealistic environments, it allows for a realistic dynamic simulation and connects well with ROS-based system. Sensor and environment data can be read programmatically using standardized interfaces. Checks, such as disabling the planning framework mid-flight, can be performed safely in the simulation environment.

Figure 4.2 shows some of the experiments performed in the simulation environment. In 4.2(a) we calculate inspection points for a simulated building using the static planner module. We tested the dynamic obstacle avoidance

as visualized in 4.2(b). The module conveniently allows to test and evaluate different dynamic planners leveraging the OMPL library. Additional safety checks can be performed to ensure that the replanned trajectory meets specific constraints such as a maximal path length. In the depicted scenario, we used the InformedRRT* planner as default choice. We assumed a $60 \times 60 \times 40\text{cm}^3$ hit volume for the planner's collision check. As admissible heuristic for the planner, we chose the *CostToGoal* heuristic, which is basically the euclidean distance to the target position.



(a) Static Planner

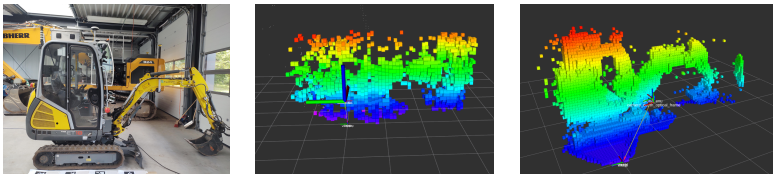
(b) Dynamic Planner in Simulation

(c) Inspection Planner

Figure 4.2: In (a), we invoked a static planning procedure (derived from [1]) resulting in a set of viewpoints. In (b) we can see the result of a dynamic RRT* Planner avoiding an obstacle on the flight path, whereas in (c) an actual short inspection routine is carried out during a real flight.

4.2 On a Real Drone

Figure 4.1(a) shows our hardware and environment setup. We used a standard *Holybro S500* frame with a *Pixhawk 4* as lowlevel flight controller. We designed different custom plug-in mounts to allow for different companion computers and sensor setups. Planning and localization tests have been performed using the Snapdragon Flight module equipped with a quad-core *Snapdragon820* processor using the ARM-v8 architecture. To enable the usage of modern cameras, we also deployed the pipeline onto the *Jetson Xavier NX* as companion computer and connected a *Realsense L515* solid-state LiDAR camera. This setup allows for a more detailed indoor map generation than the stereo setup described above. Figure 4.3 shows a comparison of the two hardware setup in terms of mapping performance.



(a) Scene (b) Stereo Setup (c) Solid-State LiDAR

Figure 4.3: The scene in (a) was captured into a map representation using different sensor setups. (b) shows an exemplary result using a stereo setup while the LiDAR setup used in (c) allows for a more detailed scene representation.

The Localization module (cf. 3.2) allows to run different localization algorithms. Figure 4.4 compares two different approaches. The *Snapdragon Flight* board comes with a basic visual-inertial algorithm based on an Extended Kalman Filter. [8]. We referenced the local pose within the global frame $\mathcal{W}$ by mounting AprilTags [10] at different locations and passing the resulting camera poses in $\mathcal{W}$ back to the EKF. The resulting poses (grey line) are used as baseline as they provide centimeter accurate pose estimates at measured reference points. OrbSlam2 [8] uses only the monocular camera to estimate the pose. The blue line in Figure 4.4 shows that while loop-closure is successfully performed on the right part of the trajectory, on the left part not enough visual features are available to provide an accurate state estimate.

We evaluate the waypoint following capabilities of the pipeline by flying trajectories through manually defined inspection points. Figure 4.2(c) shows an excerpt of such an inspection flight. We notice that the flight performance highly depends on the stability of the localization module. Frequent divergence of the visual inertial system causes the drone to fallback to its safety hovering action preventing a smooth flight.

[7] `https://michaelgrupp.github.io/evo/`
[8] `https://developer.qualcomm.com/software/machine-vision-sdk`

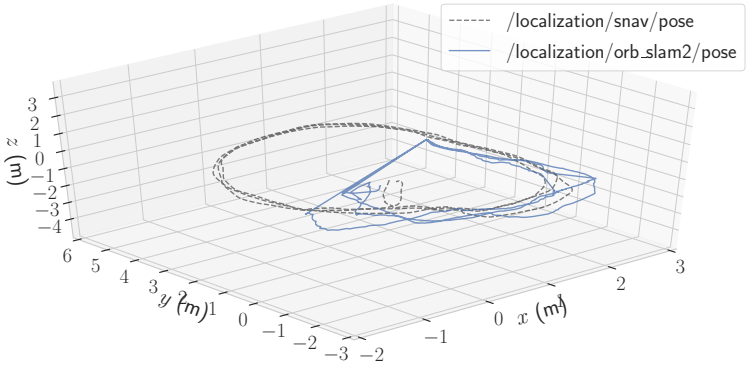


Figure 4.4: Trajectory estimates for repeated circle flight using two different localization systems. The grey line depicts the estimated pose of the reference system, which is a VISLAM approach for the custom *Snapdragon Flight* platform. The tracking camera and the onboard IMU are fused in an EKF fashion to provide a state estimate. In addition, AprilTags have been integrated to define a precise reference frame and enhance the state estimation. The blue line shows the state estimate provided by OrbSlam2 [8] using the monocular tracking camera only. The evaluation can be conveniently performed using EVO [9]as part of the pipeline.

5 Conclusion and Future Work

In this report, a pipeline for performing UAV flights in indoor environments has been introduced. The construction of such a pipeline is motivated by the lack of controlling software for research oriented flight experiments as presented in Section 2. The presented interfaces allow the dynamic usage of different modules for localization, mapping and planning. The required building blocks and their interface definitions have been established, before the cascaded planner structure was presented in detail. Finally, different experiments have been conducted in simulation as well as in real-world scenarios to verify the practicability of the pipeline. The experiments showed that the pipeline is capable to safely perform and operate simple indoor flights.

In order to enhance the robustness of the pipeline further, additional failchecks and fallbacks need to be implemented in future works. This should allow for a

smooth trajectory tracking even with inaccurate localization results. In addition, the dynamic planner module needs additional robustification for application in practice. In the current setup, the feasibility of a planned route is not rechecked after an evasive maneuver. Therefore, such a maneuver can only be performed slowly as the transition to the next planned waypoint may not be smooth. So far, the setup has only been tested with static obstacles. Moving obstacles have higher requirements regarding the real-time capability of the Mapping module which should be investigated in the future.

References

[1] A. Bircher, K. Alexis, M. Burri, P. Oettershagen, S. Omari, T. Mantel and R. Siegwart. "Structural Inspection Path Planning via Iterative Viewpoint Resampling with Application to Aerial Robotics". In: *Robotics and Automation (ICRA), 2015 IEEE International Conference on*. May 2015, pp. 6423–6430.

[2] Tomás Báca et al. "The MRS UAV System: Pushing the Frontiers of Reproducible Research, Real-world Deployment, and Education with Autonomous Unmanned Aerial Vehicles". In: *CoRR* abs/2008.08050 (2020). arXiv: 2008.08050. URL: https://arxiv.org/abs/2008.08050.

[3] M. Burri et al. "Real-time visual-inertial mapping, re-localization and planning onboard MAVs in unknown environments". In: *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2015, pp. 1872–1878.

[4] Fadri Furrer et al. "Robot Operating System (ROS): The Complete Reference (Volume 1)". In: ed. by Anis Koubaa. Cham: Springer International Publishing, 2016. Chap. RotorS—A Modular Gazebo MAV Simulator Framework, pp. 595–625. ISBN: 978-3-319-26054-9. DOI: 10.1007/978-3-319-26054-9_23. URL: http://dx.doi.org/10.1007/978-3-319-26054-9_23.

[5] Jonathan D. Gammell, Siddhartha S. Srinivasa, and Timothy D. Barfoot. "Informed RRT*: Optimal Incremental Path Planning Focused through an Admissible Ellipsoidal Heuristic". In: *CoRR* abs/1404.2334 (2014). arXiv: 1404.2334. URL: http://arxiv.org/abs/1404.2334.

[6] Armin Hornung et al. "OctoMap: An Efficient Probabilistic 3D Mapping Framework Based on Octrees". In: *Autonomous Robots* (2013). Software available at http://octomap.github.com. DOI: 10.1007/s10514-012-9321-0. URL: http://octomap.github.com.

[7] Taeyoung Lee, Melvin Leok, and N. Harris McClamroch. "Geometric tracking control of a quadrotor UAV on SE(3)". In: *49th IEEE Conference on Decision and Control (CDC)*. 2010, pp. 5420–5425. DOI: 10.1109/CDC.2010.5717652.

[8] Raúl Mur-Artal and Juan D. Tardós. "ORB-SLAM2: an Open-Source SLAM System for Monocular, Stereo and RGB-D Cameras". In: *IEEE Transactions on Robotics* 33.5 (2017), pp. 1255–1262. DOI: 10.1109/TRO.2017.2705103.

[9] Helen Oleynikova et al. "Voxblox: Incremental 3D Euclidean Signed Distance Fields for On-Board MAV Planning". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2017.

[10] Edwin Olson. "AprilTag: A robust and flexible visual fiducial system". In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, May 2011, pp. 3400–3407.

[11] Charles Richter, Adam Bry, and Nicholas Roy. "Polynomial trajectory planning for aggressive quadrotor flight in dense indoor environments". In: *Robotics Research*. Springer, 2016, pp. 649–666.

[12] L. Schmid et al. "An Efficient Sampling-Based Method for Online Informative Path Planning in Unknown Environments". In: *IEEE Robotics and Automation Letters* 5.2 (Apr. 2020), pp. 1500–1507. ISSN: 2377-3774. DOI: 10.1109/LRA.2020.2969191.

[13] V. Usenko et al. "Real-Time Trajectory Replanning for MAVs using Uniform B-splines and a 3D Circular Buffer". In: Vancouver, Canada, Sept. 2017.

[14] Kun Xiao et al. “XTDrone: A Customizable Multi-Rotor UAVs Simulation Platform”. In: *CoRR* abs/2003.09700 (2020). arXiv: 2003.09700. URL: https://arxiv.org/abs/2003.09700.

[15] Boyu Zhou et al. “FUEL: Fast UAV Exploration using Incremental Frontier Structure and Hierarchical Planning”. In: *CoRR* abs/2010.11561 (2020). arXiv: 2010.11561. URL: https://arxiv.org/abs/2010.11561.