An Egocubemap Based Algorithm for Quadrotors Obstacle Avoidance Using a Single Depth Camera

Thibaut Tezenas Du Montcel, Amaury Nègre, Matthieu Muschinowski, Jose-Ernesto Gomez-Balderas, Nicolas Marchand

An Egocubemap Based Algorithm for Quadrotors Obstacle Avoidance Using a Single Depth Camera

T. Tezenas Du Montcel*, A. Nègre*, M. Muschinowski*, E. Gomez-Balderas* and N. Marchand*
*Univ. Grenoble Alpes, CNRS, Grenoble INP, GIPSA-lab, 38000 Grenoble, France

***Abstract*—A fast obstacle avoidance algorithm is a necessary condition to enable safe flights of Unmanned Aerial Vehicles (UAVs) eventually at high-speed. Large UAVs usually have a lot of sensors and available computational resources which allow complex algorithms to run fast enough to navigate safely. On the contrary, small UAVs gather many difficulties, like computation and sensors limitations, forcing algorithms to retain only a few keys points of their environment. This paper proposes an obstacle avoidance algorithm for quadrotor using a single depth camera. Taking advantage of the possibilities offered by embedded GPUs, a cubic world representation centered on the robot - called Egocubemap - is used while the whole obstacle detection and avoidance algorithm is light enough to run at $10\,Hz$ on-board. Numerical and experimental validations are provided.**

I. INTRODUCTION

For some years now, the increase of interest for autonomous navigation of Unmanned Aerial Vehicles (UAVs) has led to the development of many autopilot systems. In order to make a safer one, an avoidance layer is needed under its global navigation control scheme. This is especially true in crowded environments, where the dangerousness of these systems is a big obstacle to concrete applications.

Autonomous obstacle avoidance for UAVs, generally, and quadrotors, especially, is not particularly new [1, 2]. For its simplicity and its low cost, lots of existing works, including ours, focus on a single looking-ahead passive depth sensor [3, 4] which raises three major issues. First, the range of the depth information of those sensors is limited to, typically, distances of 10 meters. Secondly, the angular region of space in which we have depth information is limited to the Field Of View (FOV) of the sensor. Thirdly, whether we use an RGBD camera or whether we apply a stereo-vision algorithm to a pair of stereo-vision images, the extracted depth map is more noisy than a LIDAR point cloud. Those three issues need to be addressed somehow.

The main way to solve the range issue is to have a high frequency obstacle avoidance algorithm to check for obstacles as soon as they enter the perception range. This solution also allows to fly fast with a limited input depth range [3]. We consider that reaching a frequency of $10\,Hz$ is the minimum to allow avoidance at high speed and to give ourselves a chance when considering moving obstacles.

The reduced FOV issue is usually addressed with some kind of onboard memory of the environment. Closely following the idea of Brockers et al. in [5] who presented an Egocylinder type of vision, a cubic world representation centered on the drone - called Egocubemap - is used in this paper to map the environment.

Finally, the lack of precision of the depth map is addressed by implementing a Configuration Space expansion (C-Space) [6] and adding a margin to it. The idea of a C-space expansion is to enlarge any object of the environment by a volume V. At any position in the C-Space expanded environment, it becomes possible to check if there are collisions between the volume V and the environment only by verifying if the center of the volume is occupied by an obstacle. If the chosen volume includes a body, it is therefore possible to check for collisions between this body and the environment in a single operation. Adding a margin in the volume V of the C-Space and making it include the body plus some extra-space will ensure that, if there is no collision in the C-Space, the body is at least at this margin from any obstacle. By adding a margin linked to the error in the depth map, the lack of precision of the depth map will have a reduced impact.

The rest of the paper is organized as follows. In Section II a short review of existing works is given. Section III presents the main contribution of the paper, that is the proposed algorithm. In Section IV, the results obtained during more than $100\,km$ of simulated flight are given. Section V presents the experimental validation in the context of an hardware in the loop implementation. A real UAV is flying and its position obtained by a motion capture system enables us to make it fly in a virtual simulated environment with fictive obstacles. The obstacle avoidance algorithm then runs based on the simulated images. The paper ends with some conclusions and perspectives.

II. RELATED WORKS

Obstacle avoidance consists in a set of functions. If those functions can change depending on the algorithm, two of them are always necessary: creating a world representation from the depth inputs and generating trajectories or direct control commands of the robot. We briefly spoked about other existing avoidance algorithms in the introduction, in this part we will focus on those two important functions of an obstacle avoidance algorithm.

For world representation, there are multiple possibilities. From the simplest one, taking only the raw depth sensor data at the current frame, to the most complex, 3D mapping with SLAM techniques, there is the possibility to keep in memory only a few key points [3, 4], to build a 2D 360° depth representation [5] or to use odometry techniques. There is obviously a tradeoff to find between the computational cost of

a spatial representation and its precision. This is why most of the fastest obstacle avoidance algorithms are keeping only a few key points in memory. In [4], Barry even reduced its stereo algorithm to a single distance pushbroom stereo-vision in order to reduce his processing time. Our goal is to have at least a $360°$ representation of the world while still running above $10\,Hz$ onboard a quadrotor which weights less than $1\,kg$.

For the trajectory generation, there are two main possibilities which are to generate the trajectories on-line or to generate off-line a library of trajectories and to pick one among them on-line. In both cases, trajectories can be linked to a corresponding control to apply on the robot either in open-loop or by implementing some closed-loop control to track it.

In order to generate some trajectories beforehand, it is necessary to discretize the state-space of the quadrotor, and then to generate trajectories for each of the discretized states that allow the quadrotor to go in different directions [7]. Relying on a precise model of their quadrotor, some works even focused on the uncertainty of the generated trajectories [8].

Generating trajectories on-line has one main drawback which is that the generation must be very fast (typically less than a millisecond) in order to be able to generate more than one trajectory and to leave time for the rest of the process. With a precise model, one of the fastest method still uses $2\,ms$ to generate a $500\,ms$ trajectory [9]. It therefore means that it is mandatory to use a simplified model to generate the trajectories. Historically, on-line trajectories have consisted on steering commands or geometric trajectories. Then, Mellinger & Kumar in [10] introduced a method using the differential flatness of the quadrotors which has become the new standard for on-line trajectory generation. Recently, [11] added to a flatness based generation a few efficient tests to check the compatibility of the generated trajectory with a quadrotor dynamic. Due to the constraints of our project which aims at developping a generic, 'plug and play' and working for a wide range of quadrotor, algorithm, we have to generate our trajectories online which is why we adpoted the method from [11].

III. DESCRIPTION OF THE ALGORITHM

The goal is to reach $W_i \in \mathbb{R}^{3\times n},\ i \in [1,n]$ an ordered list of n high level way-points according to the sequence of the ordered list. We consider a way-point W_i reached when the distance between it and $X(t) \in \mathbb{R}^3$, the cartesian position of the quadrotor at time t, is inferior to $\mu_i \in \mathbb{R}^+$, a distance parameter depending on the precision needed by the high level navigation. If no path to a way-point is found, the avoidance algorithm is expected to search for one for 10s before asking the navigation layer for a new high level way-point.

A. Overview of the algorithm

Figure 1 represents the main steps of the proposed algorithm which runs at each new depth acquisition. Its starting point is the depth map acquisition. Many kind of sensors can provide data that are or can be transformed into depth maps. With

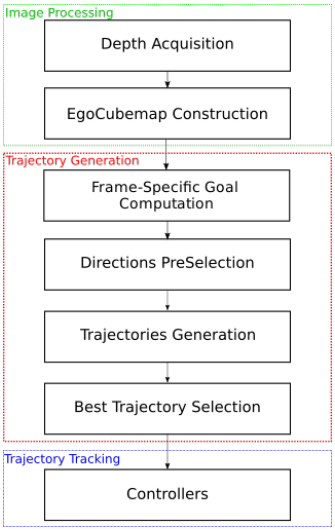


Fig. 1. Overview of the algorithm

stereo-vision cameras, this transformation step consists in running a stereo-vision algorithm, whereas with a direct depth sensor like laser sensors, this step is straightforward. The second step of the algorithm is the construction of the Egocubemap. It starts by the estimation of the displacement between the last two frames. Then an ego centric cubic representation of the environment, the Egocubemap itself, is computed. Finally, it ends with the C-space expansion, which consists in increasing the volume of each depth measure to face unprecise and non dense depth maps. This representation of the environment is the first step to figure out how the next way-point can be reached without colliding with the obstacles. For that purpose, a new frame-specific goal, that may not coincide with the position of the way-point in the frame, is computed. In that frame, trajectories reaching that goal or a neighbourhood of it are generated. Among them, we select the "*best*" one that will be given as input to the closed loop controller of the quadrotor. The following subsections explain those steps in more detail.

B. Egocubemap Construction

The Egocubemap is a world representation shaped in a cube and centered on the quadrotor. Each pixel of the Egocubemap stores the distance from the quadrotor to the closest obstacle in its direction. It is a light dense 360Â° representation of the world. Its construction is as follows:

At each new frame, the ego motion between the last used frame and the new one is estimated. To do so, we use a keyframe-based dense visual odometry mixing the real-time visual odometry from dense RGB-D images detailed in [12] while adding the keyframe feature proposed in [13]. Once the motion is estimated, we move the old Egocubemap according to it. Each pixel from the previous Egocubemap is back projected, displaced from the reverse motion and reprojected to the new estimated Egocubemap. In this step, each pixel is considered as the rectangular area between its corner coordinates in order to have a dense output. There is necessarily some overlapping during the reprojection on the new Egocubemap, which is why

a Z buffer test mechanism is used to keep only the closest depth per pixel.

The new depth data coming from the sensor is then added to the estimated Egocubemap by overwriting the old data with the newly acquired one.

Finally a spheric C-space expansion is applied which means that all the pixels, considered as obstacles, are enlarged by a sphere. We chose to use a spheric C-Space despite the nearly planar volume occupied by a quadrotor considering that it will tilt in space during the flight. Using a sphere allows to check for collisions without the need to recompute the quadrotor angle at each point of the trajectory. To build the C-Space, each pixel is considered as a single point, back projected and enlarged to a sphere. The smallest rectangle overlapping the sphere in the direction of its center is computed (see Figure 2) and reprojected.

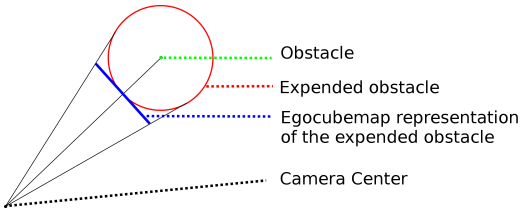
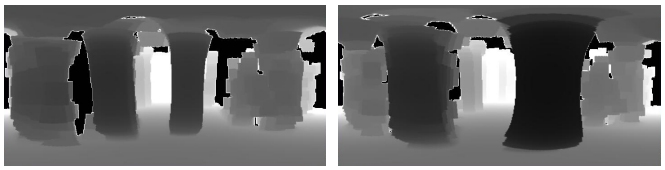


Fig. 2. C-Space reconstruction

The idea behind the C-Space is to check the quadrotor trajectories as its center trajectory instead of the sum of the trajectories of all its components. By reducing the needed checks to only a point instead of a volume at each point of each tested trajectories, we will be able to fasten the checks and therefore to check more trajectories at a reduced cost. 3.a and 3.b are two consecutive planar projections of the Egocubemap enlarged by the C-space.



(a) (b)

Fig. 3. (a) and (b) show two spherical projections of two successive Egocubemaps with Configuration Space

C. Frame-Specific Goal Computation

At this point, we have a representation of our environment, the quadrotor corresponding state and the high level way-point W_i we currently want to reach. We need to define more precisely where we want to go at this specific frame. To do so, we define a new frame-specific goal G. If a trajectory has been defined on the previous frame, the frame-specific goal G is the mean point between the end position of the trajectory and the way-point W_i. By using the previous trajectory in the new goal definition, we stabilize the direction in which we are going and avoid some instabilities in this direction. If no previous trajectory was defined, the frame-specific goal G is simply the high level way-point W_i.

D. Directions Preselection

Now knowing this frame goal G, we select some directions which could potentially lead closer to this goal. To do so, we first truncate the Egocubemap to the distance between our quadrotor and G. We then compute the distance between each point of this truncated Egocubemap and G and finally, among the closest points of the truncated Egocubemap to G, we randomly pick a hundred points E_l. The severity of the restriction on the closeness to G creates a trade off between converging to the frame specific goal and obtaining some diversity in the directions to find at least one path compatible with the flight dynamic.

E. Trajectory Generation

The previously selected points give a hundred different directions that are potentially interesting to reach the goal G and the high level way-point W_i, but we now need to verify how far it is possible to reach in those directions without colliding with the environment. By generating quadrotor feasible trajectories at multiple distances on those directions and projecting them on the Egocubemap, we will be able to check for collisions. To generate the trajectories, we are using the motion primitive generation proposed by Mueller et al. [11]. This method is a trade off between purely geometric methods and dynamically very accurate methods. It creates quadrotors feasible trajectories with the assumption that angular rate commands are tracked perfectly, an assumption which is obviously not exact but from which we are not very far considering the low angular inertia of a quadrotor.

Mueller et al. choose the trajectory as the one that minimizes their cost function, which is the integral of the squared jerk on the trajectory for a given input state, output state and time between those two states. To guarantee the feasibility, it is checked that the required thrust and angular velocity to follow the trajectory are reachable by the quadrotor. Since Mueller et al. found an expression of the minimum of this cost function, this method is really fast and allows the generation of almost a million primitives per second.

It is also worth noting that the cost function can be seen as an upper bound on the average of a product between the thrust and the angular velocity and that it reflects the dynamic difficulty of the trajectory. Therefore, it is interesting to define trajectories that have the same cost because we can expect the precision of the control on those trajectories to be pretty similar. Even if it is not possible to define a cost directly using Mueller et al.'s method, it is still possible to find trajectories with a precise cost using a binary search on the time since the cost depends solely on it for defined input and output states.

In order to avoid obstacles for each of the 100 preselected directions, we try to reach E_l for a few different costs C_m. The lowest cost corresponds to the targeted flight dynamics and the highest cost is chosen at the limit of the quadrotor dynamics in case of emergency. If we can reach E_l at C_m without collision, we generate the next trajectory which is either the same goal E_l with a lower cost or a next goal. If we noticed a collision during the projection of the trajectory on the EgoCubemap, we try to reach a new point E_{l_n} at the same cost C_i. This point is in the direction of the point E_l but at

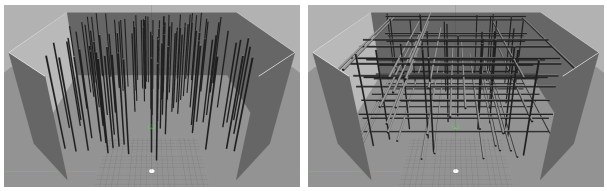


Fig. 4. View of a 2D (100 vertical cylinders) and a 3D (90 vertical or horizontal cylinders) simulated test

a distance reduced by a 0.75 factor. We repeat this operation until $\|E_{l_n} - X(t_k) < min\_dist\|$, with min_dist being the minimum distance of forward progress which depends on the distance between the quadrotor and the frame specific goal.

Finally, all the generated trajectories include a null velocity and acceleration in their final state. This ensures that the last trajectory given to the control will always be a safe one if a hardware failure was to happen. Adding this and the perception limit of 10m creates a velocity limit for the quadrotor for a defined cost. This limit is the one that allows the quadrotor to stop from this initial velocity to a null velocity in 10m for a particular cost. In the following, those velocity limits will be used to characterize the different targeted trajectory cost.

F. Best trajectory Selection

For all obstacle-free generated trajectories, we select the best trajectory $Traj_{Best}$ as a trade off between the trajectory cost $CTraj_l$ and the distance from its final state position $FTraj_l$ to the local goal G_k with more emphasis on the distance :

$$\forall\, l \neq Best, \frac{CTraj_{Best}}{CTraj_l} * \left(\frac{\|G_k - FTraj_{Best}\|}{\|G_k - FTraj_l\|} \right)^2 < 1 \quad (1)$$

In a few cases, we might not find any valid trajectory. This can mainly happen for 2 reasons : an obstacle is closer than min_dist or it is impossible to avoid the impact with a newly detected object (in case of a dynamic object for example). If no valid trajectory exists, we issue a stop command which is treated in a specific way by the controller in order to stop the UAV as fast as possible.

G. Trajectory Tracking

In order to keep our system "plug and play", we use the most common control scheme for quadrotors: a cascade controller. From the trajectory, we use the desired position, velocity and yaw to feed a first PID in position which outputs an acceleration command on the X, Y, Z axes and the desired yaw rate. Using the quadrotor dynamics, the accelerations are converted into the desired thrust, pitch and roll. Those are then fed to a second PID in angle which outputs a pitch rate and a roll rate. The command, which now consists of a thrust, a pitch rate, a roll rate and a yaw rate, can then be mathematically converted into the power needed in each rotor. This control scheme has not been designed to track trajectories and creates errors during the tracking, but it is the most common one. In our project, we wanted to show that even using this control scheme, our algorithm allows to efficiently avoid obstacles. We also keep this control scheme when we issue a stop command but we nullify the proportional term of the PID in position.

H. Computing Time

We implemented all the image processing steps on an embedded GPU card using Cuda and OpenGL. The motion estimation takes $15\,ms$ on an NVIDIA Jetson-TX1, our processing card for onboard computations, while the rest of the process, for a $128\times128\times6$ cubemap, takes $35\,ms$ on the same card. All the trajectory related steps take less than $20\,ms$ on a single CPU core@3.0GHz and less than $55\,ms$ on an NVIDIA Jetson-TX1. The control scheme takes less than $1\,ms$ on both device. Hence in total the algorithm takes less than $70\,ms$ on a ground station and less than $105\,ms$ on an NVIDIA Jetson-TX1. Since we worked only using a ground station, and since we were already above $10\,Hz$ on our ground station, there has been no emphasis on improving the performances of the algorithm, especially for all the trajectory related steps. By parallelizing the trajectory related steps, whether on the GPU or on the CPU, we feel confident about reaching $10\,Hz$ performances on an NVIDIA Jetson-TX1, thus enabling real-time aggressive motion planning.

IV. SIMULATION

A. The Setup

We decided to work with ROS since it is very widely used and allows easily to exchange packages in the same "plug and play" spirit that we followed. The use of the Gazebo simulator was then pretty straight forward considering that it had all the features we needed and the quality of its ROS integration.

We are using a simulator based on the fcu_sim ROS package from BYU-MAGICC. This package offers Gazebo/ROS sensors plug-ins and a quadrotor dynamics simulator. The idea of this simulator is to define the quadrotor as a simple 6DOF rigid body on Gazebo and then to add the forces, torques and saturations that differ between a quadrotor and a 6DOF rigid body of same mass and inertia as described in [14].

Our simulated quadrotor has a radius of $0.5\,m$, weights $3.52\,kg$. This quadrotor is around five times heavier and two times larger than the one we will use when doing real hardware experimentations, but our algorithm is supposed to be scalable and we did not want to change the native model which would have probably led to more approximation on the modeling.

The tests have been designed so that the previously defined quadrotor travels 100m in an unknown environment filled with randomly positioned obstacles. The obstacles consist in 16cm radius cylinders which cross the whole scene in specific axes. The tests are run 100 times with different maximum velocities and obstacle number which means that the quadrotor flies at least $10\,km$ in each configuration. Taking into account the fact that, in the earth frame, the quadrotor dynamics on the Z axis are different from the dynamics on the X and Y axes, which are similar due to symmetries, we designed two different tests. The first one involves only the X and Y axes (2D test) while the other one involves all three axes (3D test). The only difference between both tests is the direction of those cylinders which is only along the Z-axis in the 2D case and which is along the X, Y or Z axes in the 3D test. On Figure 4 are represented specific configurations of a 2D and a 3D test with respectively 100 and 90 cylinders.

Test directions	2D	2D	2D	3D	3D	3D
Obstacle Number	50	100	150	30	90	150
Max Velocity: 3,3 m/s						
Collisions/km	0.0	0.1	0.8	0.0	0.1	0.3
Max Velocity: 5 m/s						
Collisions/km	0.1	0.4	0.7	0.0	0.1	1.0

TABLE I
RESULTS OF THE TESTS IN SIMULATION USING THE SIMPLE MODEL.

B. Results

Figure 5 is a view from above of the quadrotor trajectory from a typical case of a test with 100 vertical (2D test) cylinders.

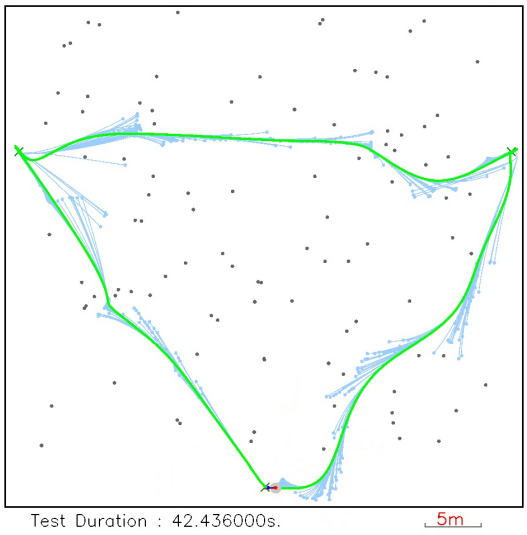


Fig. 5. A view from above of the quadrotor trajectory during a simulated 2D test with 100 cylinders. In green, the quadrotor trajectory. In blue, the trajectories generated each time the avoidance algorithm is called.

Table I gives the results of the different configurations tested in our simulated environment. The different maximum velocities have been defined depending on the time horizon of the perception of the quadrotor. With a $3.33\,ms^{-1}$ maximum velocity, given a $10\,m$ max depth perception, the time horizon is at least $3\,s$. That duration is reduced to $2\,s$ with a $5\,ms^{-1}$maximum velocity. The C-Space radius was 70cm which constitutes a margin corresponding to 40% of the quadrotor radius.

There are two main results :

- Reaching very low or null errors in multiple cases, even in very crowded environments, validates our choices while building the algorithm.
- In the very crowded 2D 150 obstacles environments, our algorithm get stuck in local minimums. In all cases, the quadrotor stops in front of the obstacles during 10s, as expected, after which it could be tasked to land.

V. EXPERIMENTAL VALIDATION

A. The Setup

The experimental validation was carried out on a homemade quadrotor represented on Fig. 6. It weights 303g and has a 33cm diameter including its 5" blades. It embarks brushless motors, a NAZE32 flight controller flashed with ROSflight [15] and it is powered by a 7.4V LIPO battery. The NAZE32 IMU was used to feed the ROSflight attitude controller and we used a motion capture system (MOCAP) to feed the avoidance algorithm and the position controller on the quadrotor state.

Fig. 6. A photo of our custom quadrotor

For the tests, all the avoidance related computations are done on a ground station. Embedding a NVIDIA Jetson-like processing card for onboard computations would necessitate a bigger frame, which is impossible in our motion capture room which useful volume is a 3m×2.5m×2m cuboid. The quadrotor is not equipped with any depth sensor. To provide such measurements to the quadrotor, a virtual clone of the quadrotor, with the same position and orientation as the real one in the MOCAP room, evolves in a Gazebo world. A depth image can then be created in this virtual environment. Virtual obstacles or clones of the real ones can be added to this world. Due to the limited size of the MOCAP volume compared to the quadrotor size, we could only create a scene with 6 or less cylinders. As in simulation, the cylinders are randomly spawned.

B. Results

Figure 7 is a summary from above of an hardware-in-the-loop test with 6 cylinders.

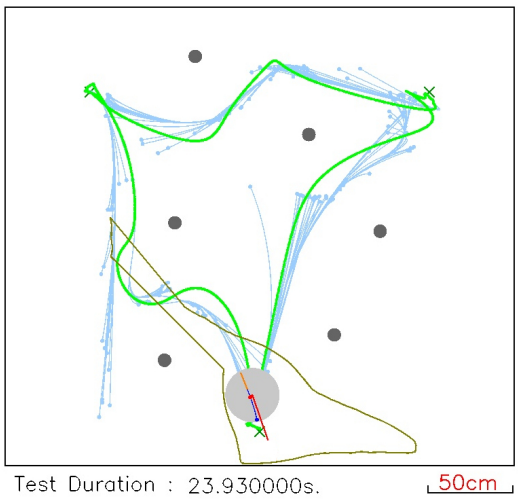


Fig. 7. A view from above of the quadrotor trajectory from a hardware-in-the-loop test with 6 vertical cylindrical obstacles. In green, the quadrotor trajectory. In blue, the trajectories generated each time the avoidance algorithm is called

Table II gives the results of the hardware in the loop tests.

Test directions	2D	2D	2D
Max Theorical Velocity	3.3 m/s	5 m/s	3.3 m/s
Number of obstacles	4	4	6
Number of tests	10	10	10
Successful tests	10	10	10
of which local minimum stops	1	2	3

TABLE II
RESULTS OF THE HARDWARE IN THE LOOP TESTS

Due to the size limitation of the MOCAP room, we were only able to test with a maximum of 6 vertical obstacles. Even with this few obstacles, 20% of the tests ended in a local minimum but it's worth noticing that the quadrotor correctly hovered, waiting for a new high-level waypoint. Also, since the algorithm has a zero velocity and acceleration constraint at the end of each generated trajectory, because of the size limitation of the MOCAP room and the short trajectories, the maximum velocity of the quadrotor was never above 1m/s despite theoretically being able to go above it in the tested configurations. Testing it in larger environment will increase the velocity of the quadrotor and is clearly the next step we have to take. Despite this limitation, the algorithm reacted as expected during those flights and there have been only a few differences between the simulated results and the hardware-in-the-loop ones. The main difference resides in the quality of the trajectory tracking. Even at those low velocity, the cascade PID scheme shows some of its limit due to an 80ms latency in the loop. We expect this delay to be reduced in a fully embedded scenario and therefore an improvement of the control.

VI. CONCLUSIONS

We presented an obstacle avoidance algorithm solely based on a single facing ahead depth input with a field of view corresponding to what is expected from a pair of stereo vision cameras. Using an Egocubemap world representation, we successfully and consistently avoided obstacles whether in simulation or in a hardware-in-the-loop setup and in differently crowded environments. Our next step will be to attempt outdoor and fully embedded flights with our algorithm.

Due to the uniqueness of the tests of each published paper, comparing our results to other existing works is also really complicated at the moment. In order to make this easier, we are currently creating an avoidance benchmark. We believe that it could help highlighting on the strengths and weaknesses of each approach.

VII. ACKNOWLEDGMENT

This work is founded by the CAP2018 project.

REFERENCES

[1] S. Scherer, S. Singh, L. Chamberlain, and M. Elgersma, "Flying fast and low among obstacles: Methodology and experiments," *The International Journal of Robotics Research*, vol. 27, no. 5, pp. 549–574, 2008. [Online]. Available: https://doi.org/10.1177/0278364908090949

[2] S. Hrabar, "3D path planning and stereo-based obstacle avoidance for rotorcraft UAVs," in *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Sept 2008, pp. 807–814.

[3] H. Oleynikova, D. Honegger, and M. Pollefeys, "Reactive avoidance using embedded stereo vision for mav flight," in *2015 IEEE International Conference on Robotics and Automation (ICRA)*, May 2015, pp. 50–56.

[4] A. J. Barry, "High-speed autonomous obstacle avoidance with pushbroom stereo," Ph.D. dissertation, Massachusetts Institute of Technology, 2016.

[5] R. Brockers, A. Fragoso, B. Rothrock, C. Lee, and L. Matthies, "Vision-based obstacle avoidance for micro air vehicles using an egocylindrical depth map," in *International Symposium on Experimental Robotics*. Springer, 2016, pp. 505–514.

[6] T. Lozano-Perez, "Spatial planning: A configuration space approach," *IEEE transactions on computers*, no. 2, pp. 108–120, 1983.

[7] S. Daftry, S. Zeng, A. Khan, D. Dey, N. Melik-Barkhudarov, J. A. Bagnell, and M. Hebert, "Robust monocular flight in cluttered outdoor environments," *arXiv preprint arXiv:1604.04779*, 2016. [Online]. Available: https://arxiv.org/pdf/1604.04779.pdf

[8] A. Majumdar and R. Tedrake, "Funnel libraries for real-time robust feedback motion planning," *The International Journal of Robotics Research*, vol. 36, no. 8, pp. 947–982, 2017. [Online]. Available: https://doi.org/10.1177/0278364917712421

[9] M. Neunert, C. de Crousaz, F. Furrer, M. Kamel, F. Farshidian, R. Siegwart, and J. Buchli, "Fast nonlinear model predictive control for unified trajectory optimization and tracking," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, May 2016, pp. 1398–1404.

[10] D. Mellinger and V. Kumar, "Minimum snap trajectory generation and control for quadrotors," in *2011 IEEE International Conference on Robotics and Automation*, May 2011, pp. 2520–2525.

[11] M. W. Mueller, M. Hehn, and R. D'Andrea, "A computationally efficient motion primitive for quadrocopter trajectory generation," *IEEE Transactions on Robotics*, vol. 31, no. 6, pp. 1294–1310, Dec 2015.

[12] F. Steinbrücker, J. Sturm, and D. Cremers, "Real-time visual odometry from dense RGB-D images," in *2011 IEEE International Conference on Computer Vision Workshops (ICCV Workshops)*, Nov 2011, pp. 719–722.

[13] C. Kerl, J. Sturm, and D. Cremers, "Dense visual SLAM for RGB-D cameras," in *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Nov 2013, pp. 2100–2106.

[14] R. C. Leishman, J. C. Macdonald, R. W. Beard, and T. W. McLain, "Quadrotors and accelerometers: State estimation with an improved dynamic model," *IEEE Control Systems*, vol. 34, no. 1, pp. 28–41, Feb 2014.

[15] J. Jackson and D. Koch, "ROSflight," 2016. [Online]. Available: http://rosflight.org/