Collision-free Path Planning in Indoor Environment using a Quadrotor

Cosmin Copot*†, Andres Hernandez†, Thi Thoa Mac† and Robin De Keyser†
* Department of Electromechanics, Op3Mech, University of Antwerp, Groenenborgerlaan 171, 2020 Antwerp, Belgium
† Department of Electrical energy, Systems and Automation, DySC, Ghent University, 9052 Ghent-Zwijnaarde, Belgium
Email: cosmin.copot@uantwerpen.be, {cosmin.copot, andres.hernandez, thoa.macthi, robain.dekeyser}@ugent.be

***Abstract*—This paper presents a new approach of a path planing algorithm applied in an indoor environment. Path planning or motion planning is an essential part of navigating mobile agents. The goal of path planning is to specify a collision-free trajectory between the start state and the goal state for a mobile agent. The estimation of the absolute position inside the testing environment is performed based on visual ground patterns and image processing technique. A graph search algorithm, A^*, find the shortest path considering not only a distance cost but also an additional cost function for controlling the safety. An illustrative example of a quadrotor is used to evaluate the robustness of algorithm.**

I. Introduction

Unmanned Areal Vehicles (UAVs) were initially put in practice by the military for a considerable amount of spying operations. Recently, they are getting more and more popular for private companies because of their potential use for a wide range of applications. Despite its controversies concerning privacy, many uses are already proven to be very useful [1]. Given the ability to perform dangerous and repetitive tasks in remote and hazardous environments, UAVs are very promising to play challenging roles such as flight maneuverer [2], ball throwing and catching [1], formation control of UGVs using an UAV [3], illustration in the field of precision agriculture using UAV as smart flying sensor [4], collaborative construction tasks [5].

Path planning or motion planning is an essential part of navigating mobile agents and has to determine a collision-free path from a starting point to a goal point. Moreover, optimization of parameters such as: distance, time or energy are being considered. Most commonly adopted parameter is the distance. Based on the availability of information about environment, there are two categories of path planning algorithms, namely off-line (Global path planning) and on-line (Local path planning) [6]. A global path planner usually generates a low-resolution high-level path based on a known environmental map or its current and past perceptive information of the environment. The method has the ability of producing an optimized path; however, it is inadequate reacting to unknown or dynamic obstacles. On the other hand, local path planning algorithm does not need a priori information about the environment. It usually gives a high resolution low-level path only over a fragment of global path based on information from on-board sensors. It works effectively in dynamic environments. However, the method is inefficient when the target is in a long distance or the environment is cluttered. Usually, the combination of both methods is advised in order to enhance their advantages and to eliminate part of their weaknesses [7], [8]. For different applications specific constraints may be imposed on the layout of the path. For example, when transporting a delicate payload, a safer path further away from obstacles may be preferred over a shorter but riskier path. These demands for an optimal path can be met by manipulating parameters of a path planner.

Several path planning techniques have already been conceived and are implemented in a number of fields/areas. However, their applicability the use depends on variables such as application, environment, degrees of freedom (DOF) and the specification of the optimal path. In this study, a graph-based algorithm is used to compute the path. A first approach to solve the graph problem is by converting the working area to a discrete graph presentation (grid-graph, or visibility graph). In this case, the problem is reduced to applying a shortest path traversal algorithm, based on Dijkstra, to find the best path to go from the initial position to the target. This algorithm can be slow when new obstacles appear because it will restart the path planning from scratch without using the previously calculated results. Therefore, several methods [9], [10], [11], [12], [13] to optimize this algorithm for a dynamic environment were proposed. In this paper, the A^* algorithm is used to calculate the path. In comparison with the Dijkstra algorithm, the A^* algorithm [9] considers an extra heuristic function in order to converge faster to a solution. A multi-objective cost function consisting two components: the shortest path and the safer path was designed in order to obtain the optimal path. To be able to follow the path, the quadrotor should know, at any time, his relative position with respect to the reference trajectory. For the indoor environmental conditions, a method based on ground patterns and image processing is used to get the drone pose (position and orientation).

The paper is structured as follows: Section II describes the hardware and the software of the system. Section III is dedicated to the robot localization and obstacle detection. Section IV investigates the path planning while the control design and the experimental results are presented in Section V. The conclusions of the paper are given in the last section.

II. Hardware and software description

A. Hardware

The Ar.Drone is a commercial and low-cost micro UAV. The quadrotor comes with internal in-flight controllers and emergency features making it stable and safe to fly [14]. There are 4 brushless DC motors powered with 14.5 W each from the

3 element 1000 mA/H LiPo rechargeable battery which gives an approximate flight autonomy of 10-15 minutes. There are four basic motions of this quadrotor: pitch, roll, throttle and yaw as shown in Figure 1. Two video cameras are mounted on the central hull. The front camera resolution is 1280x720 with a video stream rate of 60 FPS and the bottom one is 640x360 with 30 FPS. The sensors are located below the central hull and consist of a 3-axis accelerometer, a 2-axis gyroscope and 1-axis gyroscope which together form the Inertial Measurement Unit (IMU). There is one ultrasonic sensor and one pressure sensor for altitude estimation. A 3-axis magnetometer gives the orientation of the quad-rotor with respect to the command station. Communication between quadrotor and command station is done via Wi-Fi connection within a range of 30 m to 100 m for indoor and outdoor environment, respectively. The Ar.Drone creates a Wi-Fi network, self-allocates a free IP address to grant access to client devices that wish to connect. More details about internal structure of this quadrotor can be found in [14]. A Visual Studio application in C++ establishes access to all Ar.Drone communication channels, enabling functions to send commands or set configurations and also receive and store data from sensors and video stream. Thus, data can be interpreted off-line or on-line for the purpose of identification, modeling and control of the quadrotor.

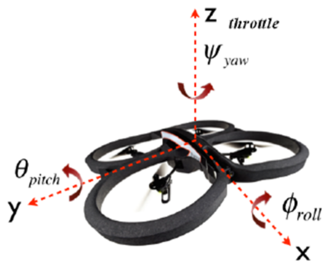


Fig. 1. Parrot Ar.Drone 2.0 and its movements

B. Software

The Ar.Drone has a *Software Development Kit* (SDK) available which makes it easy to get started with sending commands and receiving sensor data. However, in order to perform the path planing extra functionalities are required. An experimental platform is built based on a basic open source platform 'CvDrone' which combines the Ar.Drone SDK with OpenCV (computer vision library) and has already a few examples for controlling the quadrobot and for processing the images coming from the cameras.

Every functionality of the developed software architecture is divided into modules, this makes it easier to transfer and reuse some functionalities on other platforms:

- **CvDrone:** Communication with the Ar.Drone + OpenCV library for image processing.
- **Agent:** Main module, represents the autonomous entity of the quadrotor.
- **Controller:** Contains all functionalities for the controller.
- **Localization:** For determining the location of the quadrotor in the map.
- **Detection:** Responsible for detecting obstacles.
- **Planning:** Contains the planning algorithm which generate the path to a certain goal or waypoint.
- **UI:** Visualizations for showing the current state and progress of the quadrotor on the map using OpenCV.
- **Tasks:** Scriptable tasks for different tests without polluting the main code.

This C++ program can run standalone and can also be exported as a dynamic library accompanied by an API which can be used by a GUI platform that is for example build in C# Windows Forms (see Fig. 2). An additional Java simulator was created in order to perform different simulations and to analyze the behavior and the performance of the quadrotor. This simulator will also allow us to see what should we expected from the real results.

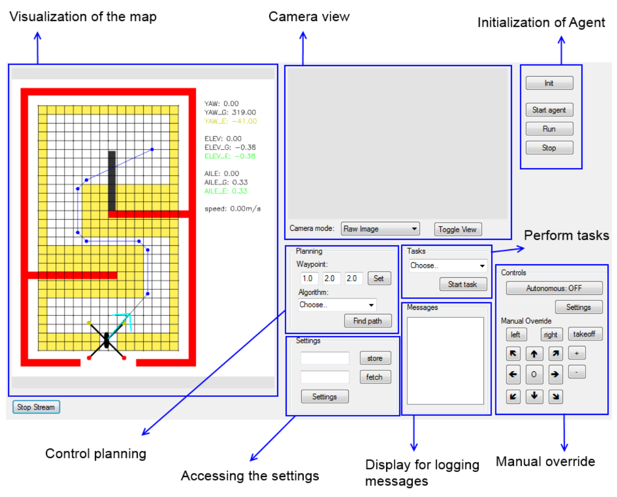
III. Localization and Obstacle Detection

A. Quadrotor Localization

In order to find the optimal path to reach a certain location, it is required that the drone knows its pose (position and orientation) relative to its goal accurately enough at any time. In the case of outside environments this problem can be solved by using GPS which is considered accurate enough.

For indoor environments however other techniques are necessary. A first approach is to use the data from the IMU chip of the Ar.Drone to determine its position. The IMU chip delivers the velocities of each axis of the quadrotor, making possible to obtain position by integrating them. However, small errors in the sensors and numerical integration can propagate to large errors in position estimation over time.

To overcome the limitations, a second approach based on vision and ground patterns is used. This is a more accurate technique, but has also some shortcomings (e.g. computation time). First, the velocity data of the drone in the local coordinate system is transformed into the world coordinate system,


YAW: 0.00
YAW_G: 319.00
YAW_E: −41.00
ELEV: 0.00
ELEV_G: −0.38
ELEV_E: −0.38
AILE: 0.00
AILE_G: 0.33
AILE_E: 0.33
speed: 0.00m/s

Camera mode: Raw Image
Waypoint: 1.0 2.0 2.0
Algorithm: Choose..
Tasks: Choose..
Autonomous: OFF

Fig. 2. GUI using the Platform Library

then the position is retrieved by using Euler integration:

$$velocity_{world} = velocity_{local} * \begin{bmatrix} cos(yaw) & -sin(yaw) \\ sin(yaw) & cos(yaw) \end{bmatrix} \tag{1}$$

$$pos_{world} = prevpos_{world} + velocity_{world} * \Delta time \tag{2}$$

This technique uses of the bottom camera and a grid of ground patterns to estimate the pose of the drone. In Fig. 3 a ground pattern is shown, each pattern represents a certain XY coordinate inside this grid. The square at the right bottom is used for detecting orientation. The top row represents the x position in binary form, and the bottom row represents the y position in binary form of the pattern:

$$x_{pos} = 2^0 x_0 + 2^1 x_1 + 2^2 x_2; \quad y_{pos} = 2^0 y_0 + 2^1 y_1 + 2^2 y_2 \tag{3}$$

Since the quadrotor has different pitch angles during flight, the center of the image from the bottom camera is not always pointing perpendicular to the ground (see Fig. 4(a)). Therefore, a correction to this center needs to be made using the information from the on board pitch sensors. This offset to the center is dependent on the pitch/roll angles of the quadrotor and the field of view (FOV) of the camera which can be calculated using trigonometry rules as illustrated in Fig. 4(b). To correct the offset for x and y, the following relationships can be used:

$$offset_x = \frac{2 * tan(roll) * tan(FOV_X/2)}{image_{width}} \tag{4}$$

$$offset_y = \frac{2 * tan(pitch) * tan(FOV_Y/2)}{image_{height}} \tag{5}$$

The third approach is based on sensor fusion techniques which combines the information from the two methods in order to have a more robust localization system, necessary to do the experiments within path planning applications. Over time the error of the odometry propagates to larger errors. So whenever a ground pattern is visible, the position calculated from odometry can be calibrated with the real position derived from the ground patterns.

B. Obstacle Detection

In order to perform path planning, it is essential to interact with the environment by sensing and avoiding obstacles. There are different possibilities of obstacle detection on a quadrotor. Ultrasonic sensors are good candidates for raw estimations of nearby obstacles while long range laser sensors and depth cameras can give even more detailed information.

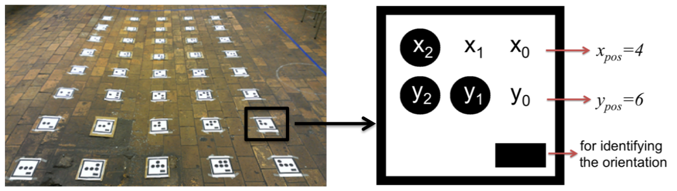


Fig. 3. Grid of patterns resembling a test area of 2x3.5m, (*left*); A Ground pattern representing position (x=4, y=6), (*right*). Note that in this case $x = \overline{0,4}$ and $y = \overline{0,7}$

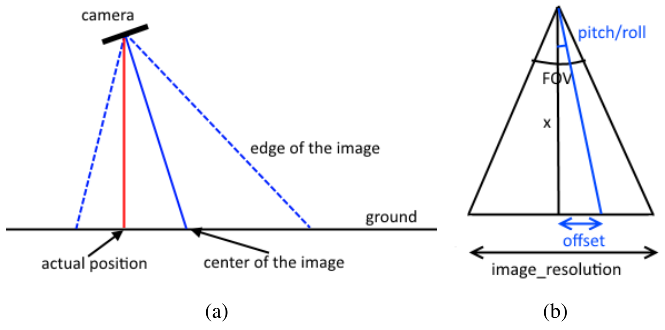


Fig. 4. (a) Center of the image versus the actual position of the quadrotor; (b) Calculation of the offset to the center

The evaluation method proposed in this paper considers both unknown and known obstacles, where the known obstacles are predefined from the beginning. This information could be extracted from a map of the building or from previous flights through that environment. Unknown obstacles become only visible upon detection, which may or may not force the algorithm to adjust the previous trajectory.

The corners on the obstacle are marked by colored patterns in order to distinct them from the rest of the room. The patterns are chosen so that each one represents a corner of the obstacle as it can be seen in Fig. 5. This way the geometry of the obstacle can be derived. The orientation of the triangles also gives information about the obstacle itself because the hypotenuse is always in the direction of the inner part of the obstacle. This was done so that the obstacle and its location can still be detected when is only partially visible to the frontal camera.

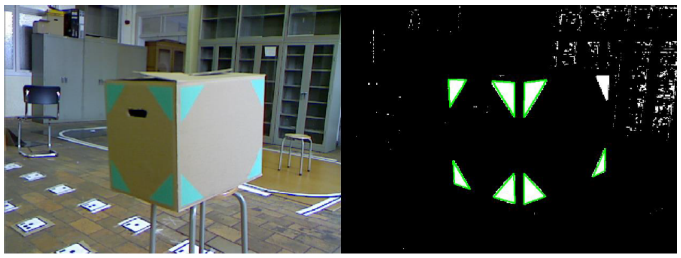


Fig. 5. Obstacle detection

In order to detect the distance to the obstacle, the size of the triangles can be used since they have a fixed size. The larger they appear, the closer they are and vice-versa. Since the shape of the triangles can change drastically based on view point and since for these tests the drone will mostly view the obstacle at the same height, the length of the vertical edges of the triangles were used to make a raw estimation of the distance.

IV. Path Planning

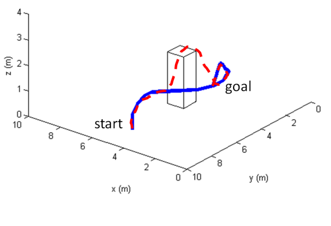
The goal of path planning is to specify a collision-free trajectory between the start state and the goal state for a mobile agent. In practice, the start state is the initial position of the agent and the goal state is a waypoint. For any application, this waypoint can be a location that needs to be visited or scanned. Depending on the type of application, multiple waypoints can be specified alongside that trajectory. The configuration space is the total set of possible states of a mobile agent within an environment. A state is specified by its pose (position (X, Y, Z) and orientation $(pitch, roll, yaw)$). States that overlap with obstacles are marked as inaccessible. The path between these states, or waypoints, can be generated in many ways based on a preferences or specification for an *optimal path*. A shorter path may be faster, but a longer path can be safer when it maintains more distance to the obstacles. This is in analogy with highway vs. shortcuts for ground vehicles. A highway can be much longer, but there is much less variation in velocity and is therefore often a more efficient route. While a shortcut through the city can require more effort from the motors of the vehicle and is thus often less efficient and longer.

There are several possibilities to define the cost function in the graph. However, most of the methods consider only the effort of moving from one node to another. Hence, the novelty of this paper is that a multi-objective cost function for the path planning is considered. Thus, instead of using only the distance as a cost, other costs will also be introduced for controlling the safety and efficiency of the generated path. These costs are then weighed so that different preferences can result in alternate path layouts.

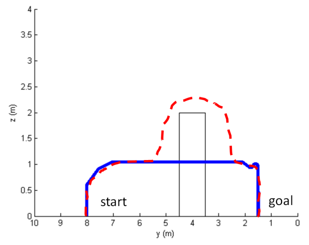
A few ideas for possible costs are introduced:

- **Move Cost**
  This cost is a measurement of distance, which will result in finding a shorter path.
- **Obstacle Distance Cost**
  Moving to a node that is closer to an obstacle can also have an additional cost. This way, if possible, a path that is further away from obstacles is chosen over a short path that is too close to an obstacle.
- **Turn Cost**
  While using a quadrotor, it is often preferable to choose a path that has less turns because it uses less effort from the controller and is thus more efficient. In some cases this can result in a longer smoother path but this is sometimes favorable.
- **Uncertainty Cost**
  In a real case scenario, the robot is often uncertain of its location and the location of nearby obstacles. Therefore, a cost for nodes that are inside an area that is not entirely visible at the time can also be introduced. This reduces the risk of going to locations that are not accessible in the hope to find a shorter path.
- **Height Cost**
  A huge advantage of a flying robot is that it is capable of flying over obstacles.
  In order to do so, the altitude needs to be changed often which is sometimes less efficient then going around an obstacle. Some applications can also require the robot to maintain a ce altitude. Hence, an additional cost can ensure that a certain height will be preferred unless there is no other option.

In this study, multiple experiments were performed to monitor the energy efficiency of the quadrotor while flying over and/or around the obstacles, Figure 6. As a result of these experiments, plus the fact that adding an extra degree of freedom (the height), more expensive algorithm is obtained. Thus, a preferred height of 1m is set for the rest of the experiments.



(a)



(b)

Fig. 6. A path planning in 3D space: (a) Angled view; (b) Side view

Once all the costs are known, the optimal path on the graph can be calculated. In this paper, the A^* algorithm is used to obtain the least-cost path from a given initial node to the goal node. In order to determine the path, the cost function of the algorithm contains the distance traveled from the initial node to the current node and a heuristic function which estimates the necessary cost to reach the goal from its current position. However, the generated path may be too close to obstacles. Therefore, in this study, an additional cost function for controlling the safety is taken into account. The designed cost function is given by:

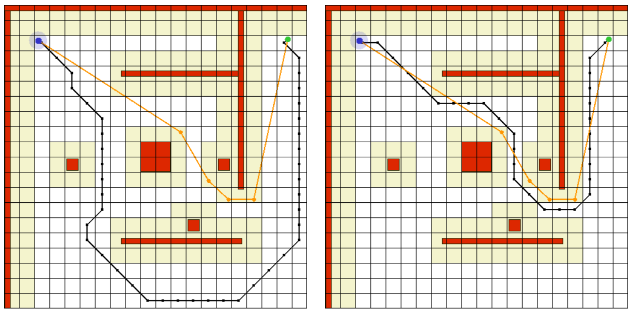
$$F(n) = w_1 * \{G(n) + H(n)\} + w_2 * S(n) \tag{6}$$

where w_1 and w_2 are positive weights of path length and path safe respectively; $G(n)$ is the known cost of getting from the initial node to node n; $H(n)$ is a heuristic estimate of the cost to get from node n to the goal node and $S(n)$ is the designed safety cost function. In this study, $G(n)$ and $H(n)$ are Euclidean distances between nodes while $S(n)$ is defined as follows:

$$S(n) = min\left\{\frac{1}{\sum d_i}\right\} \tag{7}$$

where: d_i is the shortest distance between node i and the obstacles. In figures 7(a) and 7(b) the influence of the obstacle distance cost is shown by performing A* on a grid graph. The path obtained using A^* when the cost function is defined using only the path length is illustrated in Figure 7(a)-orange line, while if only the safety cost function is considered, the obtained path is depicted in Figure 7(a)-black line. By combining the path length with the safety in the cost function, the generated path is presented in Figure 7(b). The rough layout can be smoothened by fine-tuning the path using *B-splines*. Another way is to use *Field D** which will generate more sensible paths anyways by interpolating the neighbors, instead of choosing between neighbors.

In many application it is not possible that all obstacles are known beforehand. A path planner should be able to update and re-plan the trajectory for new obstacles that are on the currently planned path and will cause a collision. Such algorithms are often described to be *dynamic* where the focus lies on efficient re-planning. The algorithm is developed to appropriately work in unknown complex environment. First, the global agent path planning is generated based on the A^* algorithm with equation (6) for known obstacles then this



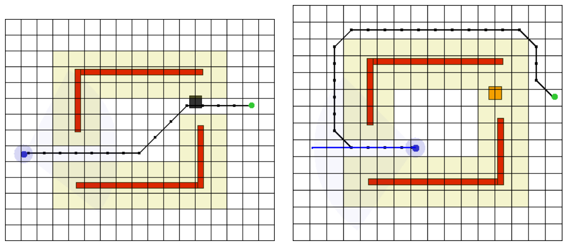
(a) (b)

Fig. 7. (a)A*: Safer path further from obstacles (black), 59.36% longer than shortest path (orange); (b) Multi-objective cost function (shortest path + safer path) is 12.48% longer than shortest path (orange)

path is renovated when the agent senses a new obstacle until reaching the target.

When should the path be updated? Many strategies are possible here. If a new obstacle appears to be on the current generated trajectory, the path needs to be re-planned or it can wait until there is more certainty about the new obstacle's position. Therefore, we can refer to dynamic approaches, where updates occur only when absolutely needed, or highly dynamic approaches, where updates occur as much as possible in order to find the optimal path.

Figure 8 illustrates how the path can be adjusted while the test is running. The known obstacles are presented in red while the unknown obstacle is presented in black. Since agent has no information about the black obstacle in advance, the trajectory is generated to avoid only red obstacles. In Fig. 8(a) the initial path is calculated and a route through the room is chosen because it is the shortest. After 3 seconds the robot detects that an obstacle is obstructing the exit from the room (the obstacle changes its color to orange), so a new path needs to be calculated. The total traveled distance is in this case longer than it could have been, but the robot was unaware of the obstacle.



(a) (b)

Fig. 8. (a) A^*: Initial path, black box $\Rightarrow$ unknown obstacle; (b) A^*: Obstacle becomes visible, path is re-planned, orange box $\Rightarrow$ detected obstacle

V. Experimental validation

The generated paths are defined by a set of waypoints. In order to perform real experiments with the quadrotor, a controller that is able to follow these waypoints is required. Also, an additional higher layer is needed (adaptive supervisory controller) which changes the target reference based on the current pose of the quadrotor and the next waypoint.

Due to the internal control, the quadrotor behaves as a set of single-input single-output (SISO) systems. The outputs of the controller represents the speed for x and y direction $(\dot{x}_{sp}, \dot{y}_{sp})$. These were considered to be the only degrees of freedom during the experiments since a fixed height of 1m has been imposed. The transfer functions obtained for the 2 DOF used in this experiment are given by:

$$\frac{v_x}{v_x^*} = \frac{7.27}{1.05s+1}e^{-0.1}; \quad \frac{v_y}{v_y^*} = \frac{7.32}{1.1s+1}e^{-0.1} \tag{8}$$

To control the movements of the quadrotor, two PD controllers with the following form were designed:

$$PD(s) = K_p\left(1 + T_d s\right) \tag{9}$$

By using the FRtool toolbox [15], the next tuning values for $K_p = 0.01$ and $T_d = 10$ were obtained for both directions.

The speeds given to the internal controllers are computed as:

$$\dot{y}_{sp} = -cos(\theta) * target_{velocity} \tag{10}$$

$$\dot{x}_{sp} = sin(\theta) * target_{velocity} \tag{11}$$

where the target velocity vector is $target_{velocity} = \|\overrightarrow{target}\|_2$ and

$$\theta = yaw_{goal} - yaw_{current} \tag{12}$$

with yaw_{goal} the orientation of the $\overrightarrow{target}$ and $yaw_{current}$ the orientation of the robot inside the world coordinate system.

When to mark a waypoint as reached? If we require the flying robot to be exactly on his waypoint before going to the next waypoint it may occur that it will miss and overshoot the waypoint. Thus, he needs to return to the waypoint and possibly keep missing the target in an oscillating behavior. Therefore a certain margin of error or *reach zone* around the waypoint is introduced.

In a first instance, the target reference is calculated only by looking at the next waypoint, while ignoring the actual line and layout of the path. This leads to unwanted behavior once the drone has drifted away from the defined trajectory. In order to solve this issue, new definitions for the target reference are introduced (see Fig. 9(a)). Let $\overrightarrow{target_{WP}}$ be the target which is pointing directly towards the next waypoint (the same as the previous $\overrightarrow{target}$), while $\overrightarrow{target_{pathline}}$ is now a new target reference directly/perpendicular towards the pathline. In short, $\overrightarrow{target_{WP}}$ (or main target) is used to make progress on the trajectory, $\overrightarrow{target_{pathline}}$ is used to stay to the path line as close as possible.

How to combine them? An intuitive approach could be to sum these two targets and use this as the final target (Fig. 9(b)). Though, whenever the drone has drifted too far from the path, it may be a better strategy to firstly fix the large path error and ignoring the main target for a while.

To have a smoother conversion between these two targets, the final $\overrightarrow{target}$ can now be defined by a weighted average between $\overrightarrow{target_{WP}}$ and $\overrightarrow{target_{pathline}}$ where the weight is dependent on the length of the path error:

$$\overrightarrow{target} = \lambda * \overrightarrow{target_{WP}} + (1-\lambda) * \overrightarrow{target_{pathline}} \tag{13}$$

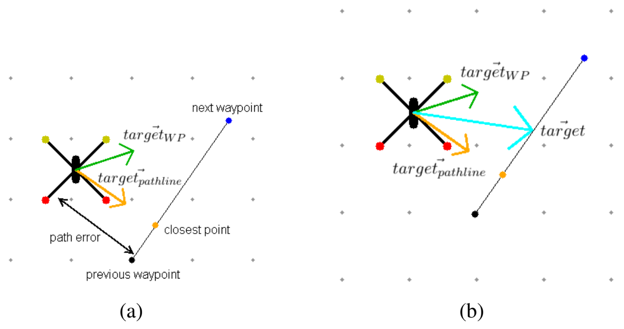


Fig. 9. (a) Calculating $\overrightarrow{target}$: Green arrow: $\overrightarrow{target_{WP}}$, Orange dot: closest distance to pathline, Orange arrow: $\overrightarrow{target_{pathline}}$; (b) Target definitions: Blue arrow: $\overrightarrow{target} = \overrightarrow{target_{WP}} + \overrightarrow{target_{pathline}}$

where λ is defined in $[0, 1]$ and measures the significance of the main target ($\overrightarrow{target_{WP}}$) dependent on the path error and can be defined as a polynomial function:

$$\lambda = \begin{cases} 1 - (path_{error}/threshold)^p & : path_{error} < threshold \\ 0 & : path_{error} \geq threshold \end{cases} \tag{14}$$

with $p > 0$ and $threshold > 0$.

If p is larger it will only react to the path error if the path error is approaching the threshold. If $threshold$ is larger the final target will also react slower to path errors. For our experiment the next values were chosen $n = 4$ and $threshold = 0.5m$, but this is off course dependent on the environment and margin of errors.

So far, the generated paths appeared to be very rough in corners and are not always very realistic. When flying a quadrotor it is often preferred to have a smoother path especially when a less aggressive controller is used. Therefore *B-splines* are considered in order to guarantee a smooth continuous trajectory along the path. Due to constraints and problems with localization the configuration space was limited to 2D. The goal is here to validate that the results obtained in simulations can be reproduced on a real quadrotor in a indoor environment. Figure 10 shows that the drone first starts by choosing a path through the narrow corridor without knowing the obstacle at the beginning. These results in the drone having to make a sharp U-turn which introduces drifting against the wall. This is also due to the controller which is not responsive enough for this type of scenario.

VI. Conclusions

In this paper a new path planning approach was implemented and analyzed, i.e. a graph-based path planing algorithm. Next, the proposed algorithm has been applied to a quadrotor for evaluation. Graph-based search is a suitable approach for the cases when the problem consists of more complex indoor environments. Therefore, a grid graph is preferred when more freedom is favorable over the path specification. An experimental platform was developed to simulate the motion of a real quadrotor in a virtual environment. This platform can also be used to test and validate the algorithms before applying it in real experiments. Multiple experiments were performed and the results show that the algorithm can generate feasible paths which can be successfully implemented in real life.

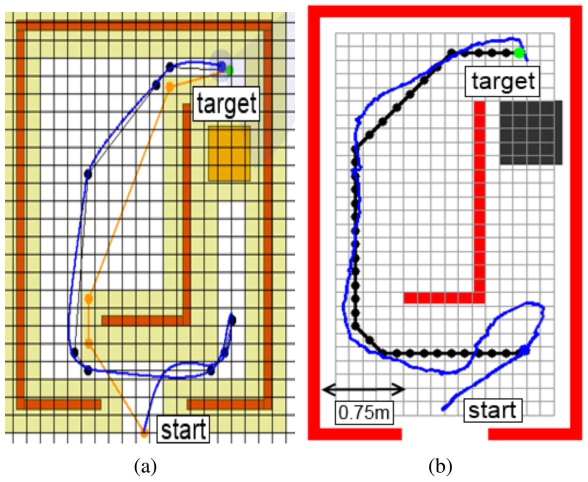


Fig. 10. (a) Simulation; and (b) Real experiment using Grid-graph search algorithm. *Orange line:* shortest path, *Black line:* defined path, *Blue line:* traveled path.

References

[1] R. Ritz, M. Mueller, and R. DAndrea, “Cooperative quadrocopter ball throwing and catching,” *IEEE Intl. Conf. on Intelligent Robots and Systems (IROS)*, 2012.

[2] D. Mellinger and V. Kumar, “Minimum snap trajectory generation and control for quadrotors,” *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, 2011.

[3] A. Hernandez, C. Copot, J. Cerquera, H. Murcia, and R. D. Keyser, “Formation control of UGVs using an UAV as remote vision sensor,” *19th IFAC World Congress*, pp. 618–623, 2014.

[4] A. Hernandez, H. Murcia, C. Copot, and R. De Keyser, “Towards the development of a smart flying sensor: Illustration in the field of precision agriculture,” *Sensors*, vol. 15, pp. 16 688–16 709, 2015.

[5] Q. Lindsey, D. Mellinger, and V. Kumar, “Construction of cubic structures with quadrotor teams,” *Proc. of Robotics: Science and Systems (RSS)*, 2011.

[6] P. Raja and S. Pugazhenthi, “Optimal path planning of mobile robots: A review,” *International Journal of Physical Sciences*, vol. 7(9), pp. 1314–1320, 2012.

[7] H. Zhang, J. Butzke, and M. Likhachev, “Combining global and local planning with guarantees on completeness,” *IEEE International Conference on Robotics and Automation*, pp. 4500–4506, 2012.

[8] Z.Bi, Y.Yimin, and Y. Wei, “Hierarchical path planning approach for mobile robot navigation under the dynamic environment,” *IEEE Conference on Industrial Informatics*, pp. 372–376, 2008.

[9] N. Nilsson, “Principles of artificial intelligence,” *Springer-Verlag*, 1982.

[10] A. Stentz, “Optimal and efficient path planning for partially-known environments,” *IEEE International Conference on Robotics and Automation*, pp. 207–222, 1994.

[11] S. Koening and M. Likhachev, “D* lite,” *American Association for Artificial Intelligence*, pp. 476–483, 2002.

[12] M. Likhachev, D. Ferguson, G. Gordon, A. Stentz, and S. Thrun, “Anytime dynamic $A*$: An anytime, replanning algorithm,” *Autonomous Robots*, vol. 13, pp. 207–222, 2005.

[13] D. Ferguson and A. Stentz, “Field $D*$: An interpolation-based path planner and replanner,” *Springer Tracts in Advanced Robotics*, vol. 28, pp. 239–253, 2007.

[14] P. Bristeau, F. Callou, D. Vissiere, and N. Petit, “The navigation and control technology inside the AR.Drone micro UAV,” *18th IFAC World Congress of Automatic Control*, p. 14771484, 2011.

[15] R. De Keyser and C. M. Ionescu, “Frtool: A frequency response tool for cacsd in matlab,” *IEEE International Symposium on Computer Aided Control Systems Design*, pp. 2275–2280, 2006.