Fine 3D path following of a quadcopter

Manuel Zafra Villar[1] and Jose María Cañas Plaza[2]

Universidad Rey Juan Carlos, Madrid, Spain

Abstract. This paper addresses the design and implementation of a path following controlling system for a drone which relies on 3D localization by visual markers. The program is designed only for indoor flight, special attention is paid to accuracy of the position estimation algorithm, robustness of the path following controller and real time operation. The system is composed of two components, one responsible of the image analysis and 3D pose estimation and another responsible of the drone navigation. The system has been experimentally validated both in Gazebo simulator and in a real drone.

1 Introduction

In recent years, the rapid development of new technologies and the decreasing price of the hardware have allowed people an easier access to robotics. There has also been an emerging interest in the use of *Unmaned Aerial Vehicles* (UAVs) on applications such as 3D mapping, military tasks, security, inspection [6] or agriculture. Maybe quadrotors are the most popularized aerial vehicles now. Currently there are several lines of research and projects with UAVs, like *Project Wing* from *Google* or *Prime Air* from *Amazon*.

In order to accomplish some of the above mentioned taks, the vehicle must work autonomously without the constant supervision of human operators. One approach to achieve this autonomy the UAV is following of predefined paths in 3D [9, 5]. For this the UAV should continuously know its 3D position. Outdoor this can be achieved with the help of sensors such as GPS, Inertial Measurement Unit (IMU) or altimeter. For indoor environments higher accuracy is needed as there may be multiple obstacles within a few meters.

Solutions to 3D localization may come from external motion capture systems [8, 7] of from onboard computer vision [1]. Typically motion capture systems monitor the position of the vehicles at a frequency of 100 Hz. The procedures may vary depending on the camera used (RGB, RGBD, ToF...) and whether the camera is on the scenario or onboard the drone. Three main techniques can be distinguished for onboard cameras. First, *Visual Odometry* in which the position is estimated by calculating the incremental movement between separate pictures extracting the characteristic points of an image. Even though this method can provide good short-term accuracy, the error rapidly increases with movement. Second, *Visual SLAM* family of algorithms allows the mapping of the area observed and simultaneously the localization of the camera. MonoSLAM, PTAM, DSO [4], LSD-SLAM and SVO [3] algorithms belong to this family. And third,

Localization based on Markers, fiducial systems, which are based on the previous knwoledge of the environment's map [2, 10]. This information is given by a series of visual markers so when the camera detects any of these markers the relative position to it can be obtanied, therefore the global position of the camera.

The goal of this project is to develop a vision-based autonomous navigation system for a quadcopter in indoor environments. The navigation will be based on a path tracking method, ensuring a robust position control. In order to accomplish that, self location will depend on computer vision algorithms relying on visual markers. Given the characteristics of the environment, the system must function with minimal error in order to avoid obstacles in close spaces.

2 Infrastructure

Several hardware and software pieces have been used in this work.

2.1 ArDrone2 quadrotor

The platform used is *Parrot*'s *ArDrone2.0*. The quadcopter was developed in 2012 as an enhanced product of its precessor, *ArDrone1.0*. The onboard computer runs a *Linux* OS, and communicates with the pilot through a self-generated Wi-Fi spot. The onboard sensors include an ultrasonic altimeter enhanced with an air pressure sensor, as well as 3-axis gyroscope, accelerometer and magnetometer, which are used to provide stabilisation. It is also equipped with a 720p front camera and a ventral QVGA sensor.

Table 1. ArDrone2.0 Characteristics

Characteristic	Value
Size	58.4 x 1.3 x 58.4 cm.
Weight	436 g.
Wi-fi range	50 m.
Max. speed	18 km/h
Camera	1280 x 720 pixels
Max. height	100 m.

2.2 AprilTags beacons

AprilTags is a visual fiducial system widely used for tasks including robotics, augmented reality and camera calibration. This system was developed in 2011 by Ed Olson [11]. The concept is similar to QR Codes in that they are two-dimensional bar codes (Fig. 1). However, AprilTags are designted to encode far smaller data payloads (between 4 and 12 bits) and it introduces a new encoding

system addressing some specific problems with 2D bar codes, enhancing robustness for rotation angles detection and against false positives, and allowing them to be detected from longer ranges.

The AprilTags detection software detects any AprilTags in a given image, providing the unique ID of the tag as well as its location in the image. It can also provide the the relative transformation between tag and camera. For this system, AprilTags has only been used for the detection of tags in an image, obtaining the tag ID and the location (height and width pixel) in the given image. The positon estimation given by this library is not used because we wanted to obtain the postion given solely by our own developed component.

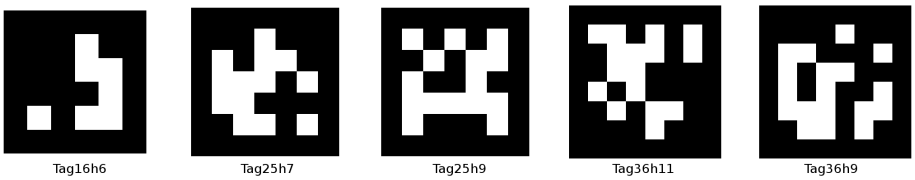

Fig. 1. Some examples of AprilTags markers

2.3 JdeRobot framework

The system has been developed using the JdeRobot software framework, a robotics and computer vision development platform. It provides support to many sensors and actuators, while providing several libraries and drivers. It is mainly developed in C++, also having some parts developed in Python. In JdeRobot a robotic application typically is composed of several interacting components and the communication between them is done through *Internet Comunication Engine* (ICE) over TCP/IP. ICE is a server-client protocol, based on *Remote Procedure Call* (RPC), which allows remote execution of software without worrying about communications.

The components may extract information from different robot sensors and provide it for other components. Within JdeRobot there are many drivers developed to support several physical sensors and adapters. This simplifies hardware access from applications providing an abstraction layer over the manufacturer's software such as *Ar.Drone SDK* in *ArDrone*.

Progeo Library JdeRobot also contains several libraries such as *progeo*, *jderobotutil* or *geometry*. Progeo is a projective geometry library that offers functions that relate 2D and 3D points. Having the camera extrinsic and intrinsic parameters, obtained by calibration, we can obtain the projected 3D point of a 2D image point. With the *project* function, we can project a 3D point in space into

a 2D image. With the *backproject* function the exact 3D point corresponding to a pixel can not be determined, but provides a ray in 3D space that contains the 3D point that projects on that pixel.

CameraCalibrator CameraCalibrator is a JdeRobot tool which obtains the intrinsic parameters of a camera. CameraCalibrator recieves camera images by ICE interface, and offers a simple user interface which simplify the calibration process. In its configuration file, settings can be modified such as calibration pattern, number of images taken or delay between images. The component uses OpenCV and its camera calibration methods and writes the output in a *.yml* text file. This tools was used to obtain the camera parameters for the real ArDrone camera and the simulated camera.

Fig. 2. Real drone ArDrone2 and calibration of its onboard camera

Ardrone_server Ardrone_server is one of the drivers included in JdeRobot that allows cammunication with the ArDrone device using ICE interfaces. The driver encloses the ArDrone SDK libraries provided by the manufacturer. There are six interfaces offered by the driver, three of them are designated for the information obtained by the sensors, two correspond to the actuators and the last one is for real-time configuration of the drone. The most important interfaces for this matter are *CMDVel*, used to send velocity commands; *Ardrone_extra*, that allows execution of complex maneuvers methods like taking off or landing; and *Camera*, where images captured by the camera are received.

Uav_viewer Uav_viewer is an application that allows the user the control of an ArDrone. It has a graphic user interface in which the information received by the quadcopter's sensors is displayed. It also has two control pads for the control of the drone's movement including vertical and horizontal speed as well as angular speed along the Z axis. The application can be use either with a real ArDrone as with a simulated quadrotor on Gazebo simulator.

Quadrotor Gazebo plugin Gazebo is a multi-robot simulator for both outdoor and indoor environments. It is able to simulate serveral robots with their sensors in a 3D world. This world can be created an personalized by the user, thanks to the tools provided for the integration with 3D models designed with SDF and the models included in Gazebo (*TurtleBot* or *Pioneer2*, from others). It is also able to simulate physical interactions between the objects in the 3D world thanks to its integration with physics libraries, such as *Bullet* or *Ogre*. The simulator is under constant development, under the *Open Source Robotics Foundation*. Gazebo provides us with an extensive API, which is very well documented, to allow de users to develop their own plugins for their personalized robots and sensors. The simulator has been used to simulate the behaviour of the drone on the first iterations of the development stages. JdeRobot includes one plugin for a quadrotor similar to ArDrone2 from Parrot (Fig. 4).

3 3D path tracking system

3.1 System design

The developed application is composed of two components, one of them is tasked with image analysis and position estimation and the other one manages drone's movement and control its position given the estimated location. Communication between components is done through ICE, same with communication between the application and the cuadricopter.

First, *VisualLoc* component receives the image taken by the quadrotor's camera. This component performs an analysis of the image looking for the presence of *AprilTag* markers. Once the markers are located, if any, the component applies projective geometry algorithms in order to estimate the relative position of the camera to each of the markers found. Next operation is the application of spatial fusion by a weighted average filter that depends on the distance between the markers and the camera, giving higher weight to those closer to the camera. Finally, a temporal fusion is carried out by a *Kalman Filter* and the estimation is sent to clients.

Second, *Navigator* component recieves the positions and generates a combination of velocities that control the movement of the quadcopter. The position control is based on a predefined path that the drone must follow. Given the horizontal characteristics of the indoor environments, the algorithm rests heavily on the rotation angle along the Z axis. The algorithm predicts the future position of the vehicle and adjust the steering angle so the future position error in relation to the path is minimized.

3.2 Beacon based visual 3D localization

The self-location task is carried out by the component *VisualLoc*. The software architecture of this component consists of a main module *MainWindow* that interconnects the rest of the modules and implements an user graphic interface. One of those modules is *Sensors*, which manages ICE interfaces and shared

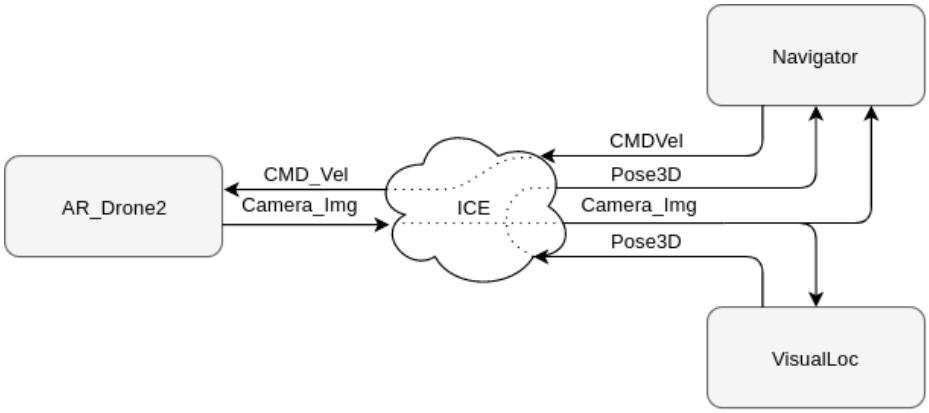

Fig. 3. Global system design

memory between the rest of the modules. MainWindow also contains instances of the *CameraManager* and *World* classes. *CameraManager* is in charge for the computer vision algorithms and *World* implements a 3D rendered world showing position estimation results. Following JdeRobot's standards, each of those modules are running on different threads that call the corresponding `update()` method. Calling this method causes the modules to carry out their main tasks in an iteration.

A *GeometryUtils* class has been developed in order to define a series of geometric calculus methods. Those methods include operations such as calculation of the intersection between planes and lines, generation of rotation matrices, or conversions between quaternions and euler angles. The structure that defines the camera is *TPinHoleCamera* model, defined in *JdeRobot*'s *progeo* library. This structurel is based on a modelling where the projection is conical, meaning that all the light rays, at some point, go past one unique point, the camera focus. Pin Hole model is very useful thanks to its simplicity and precision in camera modelling. In order to instance a *TPinHoleCamera* object, camera parameters must be obtained by previous calibration of the real camera. Information regarding markers position and camera parameters are loaded from text files, *markers.txt* and *camera.yml* correspondingly.

The method *ProcessImage* contained in *CameraManager* has the task to process the 2D image captured by the camera looking for markers, as well as estimating the 3D position of the camera. The markers found are instanced by *MarkerInfo* class, which contains the information about each marker: id, size and position. Position is stored in two matrices, one with the position of the marker regarding the world and another one with the position of the world regarding the marker. The *AprilTags* detection method is applied to a greyscale converted image, this method generates an array containing all markers detected. The

detected markers are highlighted on the original image and shown on the user interface.

Once the detection array is generated, each marker is located on the original image and a series of geometric operations are applied. Those operations start with the application of *OpenCV*'s function *SolvePnP*, which returns the relative position of the camera given a reference system composed of the correspondence between the 2D image points and the referred world's 3D points. Thus the translation and rotation vectors that determine the position of the marker in relation to the camera are obtained. In order to acquire the full RT matrix, *OpenCV*'s function *Rodrigues* is used. The matrix containing the position of the camera referred to the world is obtained by multiplying the calculated matrix and the matrix of the position of the marker in relation to the world.

The position estimations for each marker are stored in an array, then, a *spatial fusion* is applied by a weight filter. This filter asign a different weight to each estimation based on the distance to the marker, so closer markers get a higher weight. Weight values are asigned by experimental testing, analysing the distances where the system lost accuracy. The filter extracts the weighted average of the estimations in the array and obtains a ratio for each estimation (1). Then, the final estimation is calculated applying that ratio to its corresponding estimation and summing all the resulting values(2). There is a special operation with rotation angles, as they cannot be summed as the linear coordinates are (3).

$$ratio_i = \frac{weight_i}{weight_{total}} \tag{1}$$

$$[x, y, z]_{fusion} = \sum_{i=1}^{n} ([x_i, y_i, z_i] \cdot ratio_i) \tag{2}$$

$$\alpha_{fusion} = \arctan\left(\frac{\sum(\sin\alpha_i \cdot ratio_i)}{\sum(\cos\alpha_i \cdot ratio_i)}\right) \tag{3}$$

Apart from the spatial fusion, a *temporal fusion* is applied using a Kalman filter. With this method smooth results are obtained, while spike errors are eliminated. The variation of a single pixel may result in a sudden change in the position estimation, that is when the *Kalman Filter* mitigates the sudden varations that may occur. The values of the noise covariance matrices must be obtained with experimental testing in order to obtain desirable results. The final position estimation obtained through this filter is sent to *Pilot* component. Additionally, the 3D world shows the particular estimation for each marker and the filtered estimation.

3.3 3D position control

Position control is accomplished by the component *Navigator*. It consists of three modules, *Interfaces*, *Gui* and *Pilot*. System inputs are *Camera_Img*, ArDrone's camera images to be shown in the user interface; and *Pose3D*, position estimation

given by *VisualLoc*. The outputs are *CMDVel*, composed of a combination of linear and angular velocities; and *ArDrone_extra*, where additional commands such as taking off or toggling cameras are sent to the quadcopter. Additionally, in simulation, the absolute position given by the plugin can be added as an input so the position estimation error can be calculated. The three modules posses an `update()` method that is called periodically by their corresponding thread asynchronically. With this method, each of the modules run an iteration carrying out their main tasks.

Interfaces is the module where ICE interfaces are created and managed. The parameters needed, such as IP address and port, are included in a text file that is loaded by ICE. This module also manages shared memory between the rest of the modules, administrating critical sections with a mutex. The next module is *Gui*, which implements a graphical user interface. *PyQt* is the library used for the implementation, following the standards of JdeRobot. The interface shows the images captured by the quadcopter's camera, a real-time graph, in which figures the error between the vehicle's position and the path, and a 3D world rendered with *OpenGL*. This world shows the quadcopter's position in relation to a coordinate axis and the path to follow, as well as the vehicle's trail. Additionally, a series of buttons are implemented allowing the user to manipulate the quadcopter with actions such as pausing or resuming movement, taking off, landing and toggling cameras.

The module *Pilot* is where all the position information is processed and the velocity commands are generated. Initially, a simple navigation system was developed which only relied on direction vectors from the vehicle to the corresponding path point. After some experimental testing, we reached to the conclusion that steering angle along Z axis (yaw) is a key factor for an accurate navigation, whereas lineal velocity can be constant while not making a big impact. Additionally, the only linear velocity that matters are vertical velocity along Z axis and velocity along quadcopter's X axis, being velocity along quadcopter's Y axis irrelevant. Thus, developing a control system around the steering angle was the most suitable option. The algorithm is based on position prediction, so that steering angle adapts to the predicted error minimizing it. Only horizontal components are considered in the error prediction given the vertical components are minimal and don't affect steering control.

The algorithm starts calculating the direction vector between the current position and the desired postion, that is the corresponding path point (4). Then, unit vector is computed (5). In order to obtain the magnitude of the vertical and horizontal velocities while keeping linear velocity constant, unit vector must be decomposed. Vertical veloctiy is computed directly from the Z component of the unit vector and the predefined constant linear velocity v_k (7). Horizontal velocity is obtained by calculating the modulus of the X and Y component of the unit vector (6).

$$\mathbf{V} = \mathbf{P_{ath}} - \mathbf{P_{ose}} \tag{4}$$

$$\mathbf{u}_v = \frac{\mathbf{V}}{|\mathbf{V}|} \tag{5}$$

$$v_x = |\mathbf{u}_{vxy}| \cdot v_k \tag{6}$$

$$v_z = |\mathbf{u}_{vz}| \cdot v_k \tag{7}$$

Next step is obtaining the horizontal distance the vehicle will travel until the next algorithm iteration. That distance is calculated from the horizontal velocity previously obtained and the lapse of time between iterations (8). In order to compute the future position regarding the current steering angle must be calculated. Considering that the received position rotation angles are expressed in quaternions, a transformation to euler angles must be done (9). Once those values are calculated, future position point is obtained (10)(11).

$$d_\tau = v_x \cdot \Delta_t \tag{8}$$

$$\theta_z = \arctan^2\left(\frac{2 \cdot (q_0 \cdot q_3 + q_1 \cdot q_2)}{1 - 2 \cdot (q_2^2 + q_3^2)}\right) \tag{9}$$

$$X_f = d_\tau \cdot \cos\theta_z + x_{pose} \tag{10}$$

$$Y_f = d_\tau \cdot \sin\theta_z + y_{pose} \tag{11}$$

In order to acquire the desired steering angle another factor must be obtained. That is the future lateral error, computed from the difference between the predicted future position and the desired path point(12). Finally, the steering angle is composed of the steering angle needed, δe, plus a steering angle gain that depends on the predicted future error (13). The steering angle needed is obtained by calculating the yaw angle needed by the vehicle to face the navigation point. The factor K_g is the gain rate of the steering adjustment and must be obtained experimentally. This adjustment ensures a minimization of the error by correcting vehicle's trajectory and smoothing its movement. The velocity commands sent to the quadricopter through the method `CMDVel()` are v_x, v_z and δ_θ.

$$L_{fe} = -\sin\theta_z \cdot (X_{path} - X_f) + \sin\theta_z \cdot (Y_{path} - Y_f) \tag{12}$$

$$\delta_\theta = \sin\delta_e + K_g \cdot (L_{fe}/v_x) \tag{13}$$

4 Experiments

Experimental tests have been performed on simulated and real environments. The first tests were performed in *Gazebo* simulator, creating a custom 3D world representing a flat where several AprilTags markers were placed. First experiments were conducted to both of the components inependently. Then, once validated, the full working system was tested first on simulated environments and then on a real scenario.

Early self-localization experiments showed high levels of noise. Through analysis, the conclusion was that the algorithm was working properly but the parameters given to it were not. *Kalman Filter* noise covariance matrices needed to be adjusted so the noise were modelled accordingly. Another problem was a faulty calibration of the camera used, which was easily solved by a more accurate camera calibration. Finally, one of the most important conclusions was that the markers needed to be bigger due to the difficulty in the markers detection process caused by the high mobility of the vehicle. The final marker size chosen was 25cm.

On the control side, experiments showed good results. We had to run several iterations in order to find the proper parameters of the control algorithm. Vehicle's optimal speed is 0.1-0.4 m/s, and steering angle gain rate, K_g showed stable results within the range [0.1, 0.3]. The gain rate must be proportional to vehicle's speed in order to have a stable postion control. Lower gain rates are not enough to minimize the position error, while higher rates cause an erratic movement.

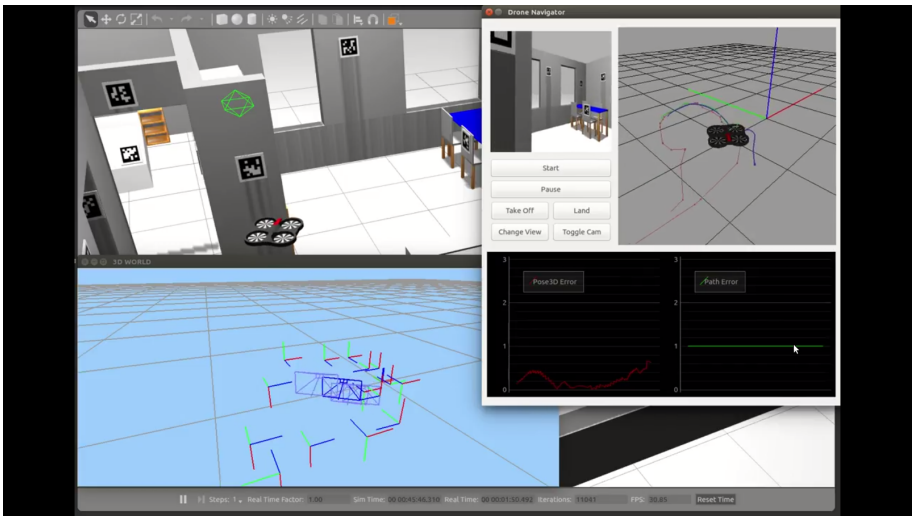

Fig. 4. Position based control of a simulated drone

The first real scenario experiments showed an unstable system behavior, so several iterations had to be run in order to find the correct parameters. Starting with self-localization algorithm, noise matrixes had to be re-adjusted so they could reflect the new noise model. Several marker sizes were tested (17cm., 23cm. and 33cm.), obtaining the best results with a 33cm. marker size. Also, quadcopter camera needed to be calibrated. The position control component also showed an erratic behaviour due to the natural drift of the vehicle and the magnitude of the velocity commands. Even though the drift caused by the slow movement of the vehicle could not be completeley countered, speed parameters were adjusted so its impact was minimal. Furthermore, the movement of the vehicle caused a blurry effect on cameras, being angular speed a critical factor for this incidence.

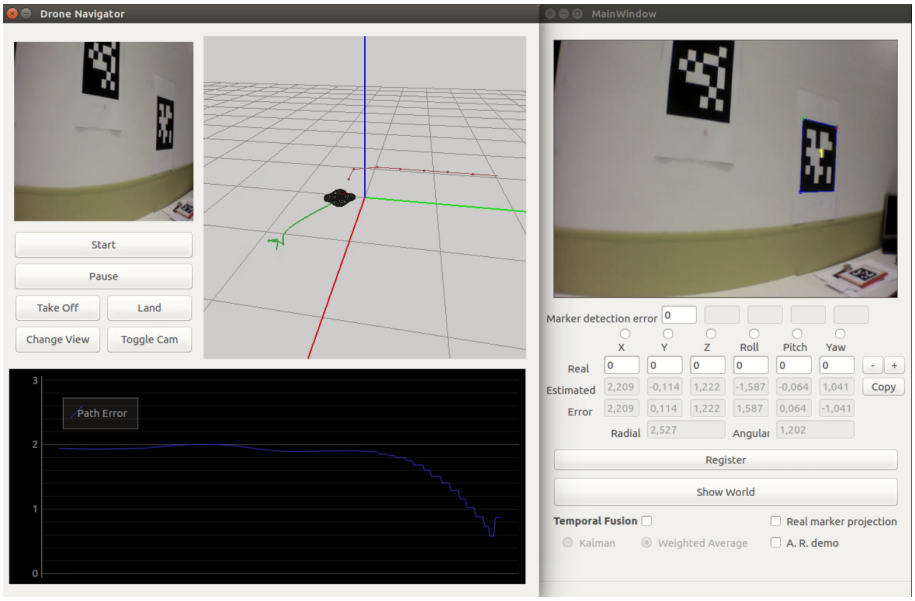

Fig. 5. Position based control of the real drone

5 Conclusions

We have designed and developed an autonomous quadricopter indoor flight system based on a fiducial auto-localization technique. In order to achieve that, we made use of a wide range of different technologies that had to be comprehended and sometimes adapted for our purposes. The system is validated on simulated environments as well as on real environments. Even though noisy data can affect the performance, it has been proven to be stable enough to fit the proposed goals.

Experimental results show us that the system is limited in velocity due to the blurriness present in the images taken at high speed. Another limiting key factor to having a robust system is the size of the markers. Acknowledging those limitations, the final prototype is robust enough to satisfy the initial requirements of the project. Future working lines may involve the addition and combination of other visual auto-location techniques such as visual odometry. Furthermore, more exhaustive position control methods could be added in order to counter the quadricopter's drift.

Acknowledgements

References

1. Wu, A., Johnson, E.N., Kaess, M., Dellaert, F., Chowdhary, G.: Autonomous Flight in GPS-Denied Environments Using Monocular Vision and Inertial Sensors. J. Aerospace Inf. Sys.. 2013 Apr 1;10(4):172-86.
2. Apvrille, L., Dugelay, J.L., Ranft, B.: Indoor autonomous navigation of low-cost mavs using landmarks and 3d perception. Proc. Ocean and Coastal Observation, Sensors and Observing Systems. 2013
3. Forster, C., Pizzoli, M., Scaramuzza, D.: SVO: Fast semi-direct monocular visual odometry, in Robotics and Automation (ICRA), 2014 IEEE International Conference on. IEEE, 2014, pp. 15–22.
4. J., Engel, Koltun, V., Cremers, D.,: Direct sparse odometry, IEEE Transactions on Pattern Analysis and Machine Intelligence, 2017
5. Beul, M., Krombach, N., Zhong, Y., Droeschel, D., Nieuwenhuisen, M., Behnke, S.: A High-performance MAV for Autonomous Navigation in Complex 3D Environments, International Conference on Unmanned Aircraft Systems (ICUAS) June 2015 DOI: 10.1109/ICUAS.2015.7152417
6. Nikolic, J., Leutenegger, S., Burri, M., Huerzeler, C., Rehder, J., Siegwart, R.: A UAV System for Inspection of Industrial Facilities, IEEE Aerospace Conference, 2013 10.1109/AERO.2013.6496959
7. Lupashin, S., Hehn, M., Mueller, M., Schoellig, A., Sherback, M., D'Andrea, R.: A platform for aerial robotics research and demonstration: The Flying Machine Arena, Mechatronics 24 (2014) 41–54
8. Jiménez, J., Zell, A.: Framework for Autonomous On-board Navigation with the AR.Drone, Journal of Intelligent & Robotic Systems, January 2014, Volume 73, Issue 1–4, pp 401–412
9. Hehn, M., D'Andrea, R.: Real-Time Trajectory Generation for Quadrocopters, IEEE TRANSACTIONS ON ROBOTICS, 31(4), 2015
10. López-Cerón, A., Cañas, J.M.: Accuracy analysis of marker-based 3D visual localization XXXVII Jornadas de Automática, Madrid, 2016
11. Olson, E.: A robust and flexible visual fiducial system, IEEE International Conference on Robotics and Automation (ICRA), 3400-2407, 2011.