On Autonomous Landing of AR.Drone: Hands-on Experience

Roman Barták, Andrej Hraško, David Obdržálek
Charles University in Prague, Faculty of Mathematics and Physics, Malostranské nám. 25, Praha, Czech Republic
{roman.bartak, david.obdrzalek}@mff.cuni.cz, andrej@hrasko.eu

Abstract

Autonomous landing is one of the must-have operations of unmanned aerial vehicles such as drones. In this paper we describe an approach for fully autonomous landing of AR.Drones. This approach consists of two components – identifying and recognizing a proper pattern used to mark the landing area and reaching the landing area (landing) even if small disturbances occur. We discuss possible patterns to identify the landing area, justify why we selected two coloured circles as the landing pattern, and describe an algorithm to recognize this pattern. In the second part we explain the PID controller used to control the drone during landing. The presented techniques were implemented and are available as an extension of Control Tower – software for controlling AR.Drone from a computer.

Introduction

Artificial Intelligence (AI) reached a significant advance in its subareas in the last twenty years but the trade-off was its fragmentation. Although we now understand much deeply individual subareas of AI, particular results are less visible in practice as these results are rarely applicable separately. One of the recent trends in AI is integration. Its goal is solving real-life problems via careful combination of existing techniques developed in various subareas of AI. This approach increases visibility of AI again but it also brings new challenges. Autonomous agents behaving in a real world impose several requirements on the technology. In particular, we see three major problems when developing AI software for real-life agents: the software must run in real time, the environment is not always as the agent perceives it, and finally, the things do not always work as the agent intends.

The significant progress in robotics in recent years brought inexpensive hardware that is frequently presented as robotic toys. One of such robotic toys is an AR.Drone (Parrot 2013) – a robotic quadricopter equipped with an extensive set of sensors including cameras. This flying "toy" can be controlled via WiFi from Android and iOS devices and also from a computer. AR.Drones are acting in a real-world environment with all the difficulties it brings and hence controlling them is definitely not a toy problem. Even if the hardware is now widely available and affordable; the major limitation seems to be good software to control these agents. Autonomous landing is a must-have feature both for fully autonomous flight as well as for safe landing by less experienced human "pilots".

In this paper we present an integrated approach to fully autonomous landing of an AR.Drone. The AR.Drone is equipped with a processing unit, but pattern recognition is computationally demanding so we decided to control the drone from a computer. Nevertheless we are not using other external devices and only the sensors and actuators available in the drone are exploited. The landing software consists of two integrated components: one for recognizing a landing pattern and identifying the goal position and the other one for controlling the drone to reach this position.

In the paper we will first introduce the AR.Drone platform by describing its capabilities. Then we will discuss various pattern recognition techniques and explain our choice of the landing pattern and the technique to recognize it. After that we will describe the PID controller used to control the drone during landing. Finally, we will give some details of our implementation and the user interface, and present the preliminary experimental results.

AR.Drone Platform

AR.Drone by Parrot Inc. (Parrot 2013) is a high-tech flying toy. Technically, it is a quadricopter with sensors and a controller. As this device is very light (360-400 grams) and therefore quite prone to weather disturbances, it is better suited for indoor environments. To a certain extent, it can operate outdoors as well.

The AR.Drone is controlled by a 32-bit 468 MHz ARM9 RISC processor with 128MB DDR-RAM running at 200MHz. This processing architecture controls the basic operations of the drone including stabilization. It is possible to install own programs there but due to a limited computing power we have decided to use an external computer.

The core set of sensors consists of a 3-axis accelerometer, a 1-axis gyroscope, a 2-axis gyroscope, an ultrasound sensor (sonar), and a vertical downside camera. The 1-axis gyroscope measures yaw with the error 12° per minute during the flight or 4° per minute in the steady state. The 2-axis gyroscope measures pitch and roll with the error 0.2° per minute (see Figure 1 for the explanation of the yaw, roll, and pitch notions).

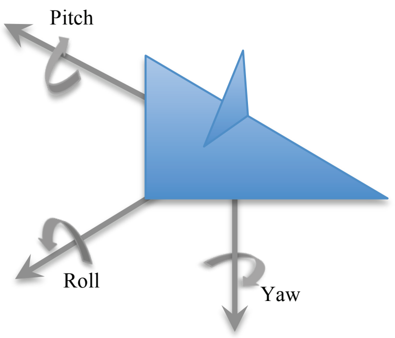

Figure 1. Yaw, Roll, and Pitch.

The ultrasonic sensor measures the altitude of the drone in the range 20-600 cm. The CMOS vertical camera directs downside; it has a resolution 176 × 144 pixels corresponding to viewing angles 45° × 35° and provides a video-stream with 60 frames per second. The system uses this camera to estimate the speed of the drone via measuring the optical flow. It is also used for calibration of other sensors. There is a higher resolution horizontal camera too, but we will not use it for landing.

The quadricopter can be controlled via WiFi from an external device such as a computer. All the communication is done using three UDP channels. A *Command channel* allows sending commands to the drone. The device accepts the following commands with frequency 30Hz: takeoff, land, set limits (maximal declination, speed, and altitude), calibrate sensors, swap cameras (see below), set rotation speeds of rotors, set pitch, roll, yaw, and vertical speed. A *NavData channel* provides information about the drone state again with frequency 30Hz. The following information is provided: drone state (flying, steady, landing, takeoff, calibration, booting), sensor data (current pitch, roll, yaw, altitude, battery level, and speed in all axes). Finally, a *Stream channel* provides visual information from the cameras. Due to a limited bandwidth, only view from a single camera is available (or *picture-in-picture*).

Identifying the Target Area

The first critical decision for autonomous landing is identifying the landing area. This identification must be fast and reliable with low sensitivity to noise. One can use a beamer as commercial aircrafts do, which is a very reliable and fast method, but it is also expensive and restricted to properly equipped landing areas. Hence, it is inappropriate for low-cost drones. We decided for a more flexible but also more computationally demanding approach of visual identification of landing patterns using computer vision techniques. We looked for a reasonably reliable and fast method that can recognize a simple visual pattern used to identify the landing area. We will now discuss several such pattern recognition methods and visual patterns to explain our final decision.

Pattern Recognition Methods

We considered three methods for recognizing visual patterns. Firstly, we considered the *template matching*. It is a digital image processing technique for finding small parts of an image, which match the template image (Brunelli 2009). This is a fast and reliable method but in its pure form it is sensitive to rotation and scale of the template. Trying different rotations and scales of the pattern solves this problem, but then the method becomes slow.

Secondly, we probed a widely used technique called *feature detection and matching* (Szeliski 2010). This method is based on identifying interesting points in the image and describing them by vectors of numbers charactering the features of these points. There exist many feature detection methods such as SURF, SIFT, ORB, BRIS, and FREAK with various levels of invariance to rotation, color changes, luminosity etc. The second step of the algorithm is finding a matching between the corresponding locations in the template and in the current image, which is then used to define the transformation between the images and hence to find the exact location of the template in the current image. This is a very general method but it may fail in feature-poor images as Figure 2 shows.

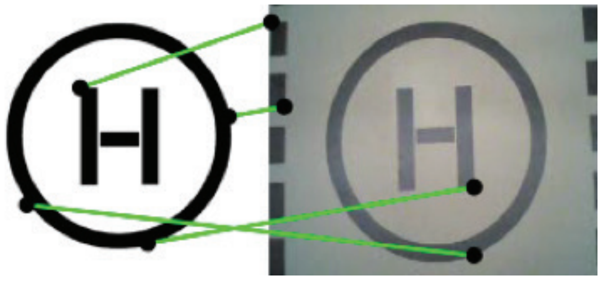

Figure 2. Failed matching of a template image using ORB descriptor and FlannBasedMatcher.

The last method that we considered is the *blob detection*. A blob is a region of a digital image in which some properties such as brightness, convexity, circularity or size are constant or vary within a prescribed range of values. For each blob we can identify its size and its center location. This can be used to identify the template position, where the template consists of several different blobs. This method can be seen as a special version of feature detection and matching, where the blobs describe particular features. The disadvantage of this method is that it is not possible to define a blob color.

Visual Landing Patterns

The success of a particular pattern recognition method is highly dependent on the used visual pattern. Obviously, the used pattern is closely connected to the pattern recognition method but the pattern must have some general properties. Specifically, the method must be able to identify a specific point in the area and also a specific orientation. We explored several possible patterns.

The most straightforward choice is the classical *heliport symbol H* (Figure 3 left). We already showed that it might not be easy to properly recognize this pattern (Figure 2), but the major problem of this pattern is that it does not provide a unique orientation.

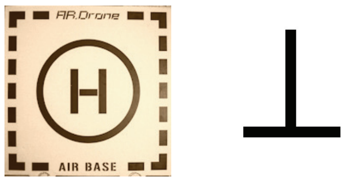

Figure 3. Heliport pattern (left) and reverted T pattern (right).

Then we tried the *reverted T pattern* (Figure 3 right) which is also quite popular. It resolves the problem with orientation but the problem with pattern recognition persists as the "ends" (instrokes and outstrokes of the letter shape) and the "edges" are locally very similar.

To exploit the blob detection method we first considered a pattern in the form of *two rectangles of the same color but different sizes* (Figure 4 left). The blob detection algorithms can easily recognize such blobs and therefore we can find a specific location as well as the orientation of the pattern. The problem is that existing objects in the scene can easily confuse the pattern recognition at all. Figure 4 (right) shows such a situation: the blob detection algorithm recognized a different pair of rectangles than expected (the red circle is supposed to be between the two rectangles forming the pattern).

The pattern must be easy to distinguish from the environment (from other objects) and it should not be sensitive to the varying illumination. We can add color to the pattern to distinguish between the blobs, that the pattern is composed of, and we can also define the orientation. We propose to use two equally sized discs (circles) of different colors, see Figure 5. The colors can be chosen to differentiate the discs from the surrounding environment and from each other. Such pattern is invariant to scale and rotation and to some extent also to the illumination conditions.

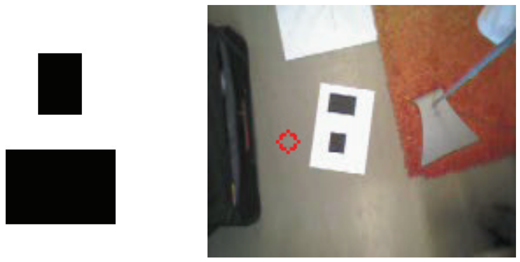

Figure 4. A 2-rectangle pattern (left) and its confusion (right).

A Landing Pattern and a Method to Identify it

Our goal was to find a simple visual landing pattern that can be easily and reliably identified. It should be also easy for the user to specify it. We decided for two colored discs being recognized by blob detection with some additional preprocessing. The user can use any two colors; it is just necessary to specify appropriate ranges for hue, saturation, and value/brightness (HSV) for both discs to distinguish the disc from the other disc and from the environment by performing the HSV band-pass filtering. This makes the image binary (two colors) which is appropriate for blob detection, but there still may be image noise introducing small ghost blobs (see Figure 6, top right). These small points can be removed by *image erosion* that takes larger subareas and uses the darkest point in the area for all points in the area. But as a result, the grain in the blob may grow (Figure 6, bottom left) which might actually cause the blob detection algorithm to incorrectly find more blobs. This grain can be removed by *image dilatation* that is similar to erosion; just the lightest point is used to define the color of the subarea. After this preprocessing (Figure 6), the classical blog detection method can be used – we use the `SimpleBlobDetector` function from the *OpenCV library* (2013).

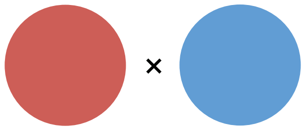

Figure 5. The landing pattern consisting of two colored discs (the cross identifies the target point that we use for landing).

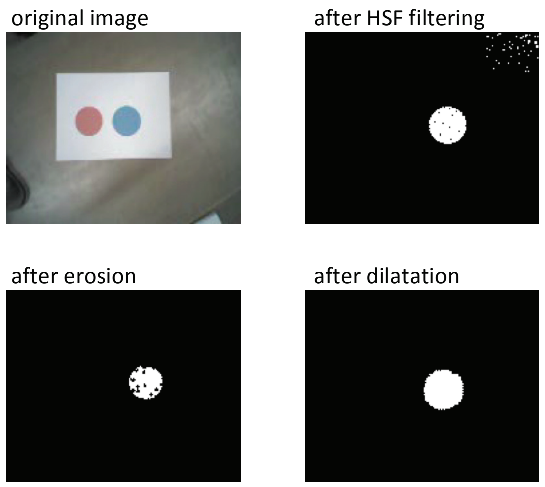

Figure 6. Preprocessing of image to identify one part of the pattern using HSV filtering, image erosion, and image dilatation.

Landing Control

When a particular landing point and orientation in space is detected, the next step is directing the drone to that point. There are basically two ways how to land the drone:

1. navigating the drone right above the landing point and then decreasing the altitude while maintaining the position above the landing point,
2. navigating the drone directly to the landing point by simultaneously decreasing the altitude and orienting the drone towards the landing point.

We decided for the second landing approach as the controller for changing altitude and the one for navigating above the landing point are independent and therefore they can run in parallel.

In the next sections we will discuss how to translate the location of the target point from the camera view to the real 3D space and how to control the drone to reach that point.

Computing the Distance to Target

Controlling the drone basically means decreasing its "error" – the distance between the current and the target locations. There is a horizontal distance that is decreased by moving forward/backward and left/right, a vertical distance that is decreased by changing the altitude, and a rotational distance that is decreased by rotating the drone. The vertical and rotational distances are measured directly via the ultrasound sensor (altitude) and the gyroscope (yaw) respectively, so we will focus now on measuring the horizontal distances. Figure 7 shows the scheme of the situation that is used to compute the distances. Drone camera is at point D right above the point V, T is the target point, C_1 and C_2 are the edges of the area visible by the camera and x is the altitude measured by the ultrasound sensor. First, we use x to approximate the real altitude v as the angle α is usually small and the ultrasound sensor has some small error anyway. Note however, that the camera does not provide the other measures directly; it gives a picture as shown in Figure 7 (top left).

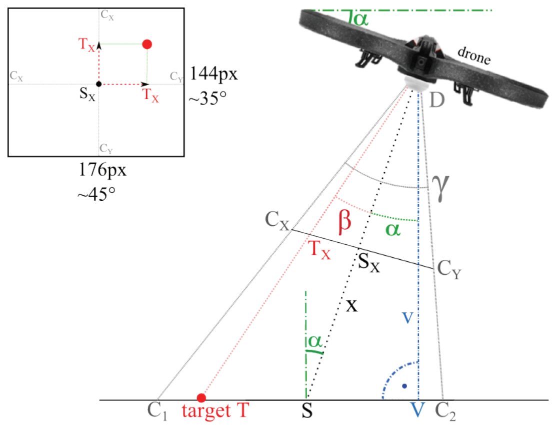

Figure 7. A schematic drone position (a side view) and the view from the bottom camera (top left). Points T,S,C_1,C_2 are seen as T_X,S_X,C_X,C_Y.

We need to compute the distance between T and V, which can be done using the formula:

$$dist(T,V) = v \,.\, \tan(\alpha+\beta).$$

The angle α is known from the sensors – it is either pitch for the forward/backward distance or roll for the left/right distance. The angle β proportionally corresponds to the distance between T and S. We measure this distance in pixels from the camera picture. This is not the real distance of points but the distance of their projections in the camera picture. Notice that we already know the pixel distance between C_Y and S_X and the angle γ from the camera properties (±72 pixels and ±17.5° for the longitudinal axis, ±88 pixels and ±22.5° for the lateral axis). Hence we can compute the angle β using the formula:

$$\beta = \arctan(\tan(\gamma/2) \,.\, \text{pixel\_dist}(T_X,S_X) / \text{pixel\_dist}(C_X,S_X)).$$

Drone Controller

We control the drone by repeating the following steps:

1. identifying the landing point in a video frame,
2. finding the error - the distance of the current position from the target position,
3. feeding the error to the controller.

As the main control mechanism, the *proportional-integral-derivative (PID) controller* is used. It is a generic control loop feedback mechanism widely used in industrial control systems (King 2010). Simply speaking, its goal is to dynamically minimize the error of the controlled system.

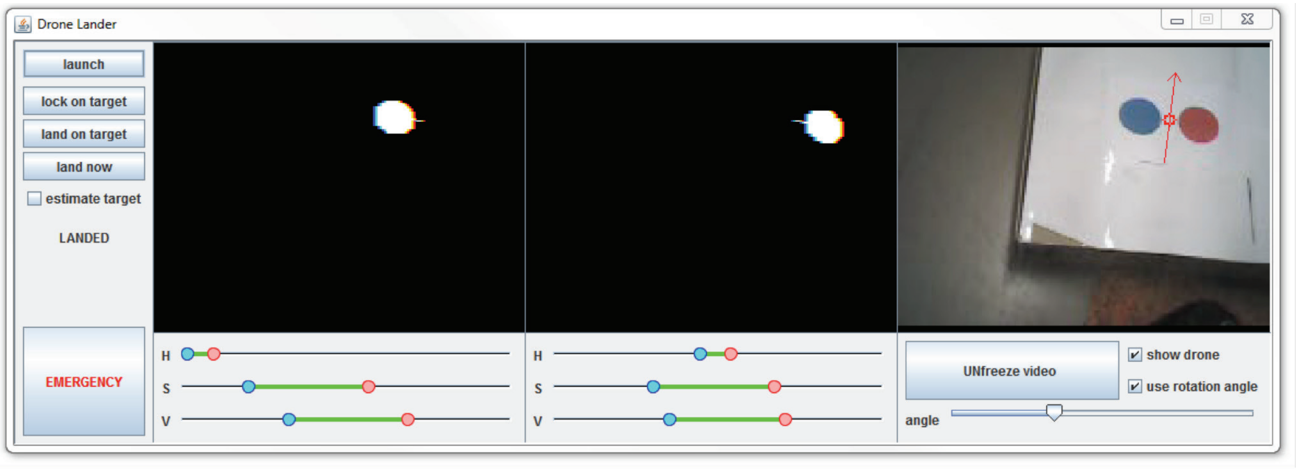

Figure 8. DroneLander user interface.

We use four PID controllers to control the drone, namely for forward/backward movement, for left/right movement, for rotation, and for altitude. In the previous section, we described how to compute the errors for all these controllers. The computation procedures behind all these controllers are identical; the only difference is in the setting of proportional, integral and derivative gain constants for the four different controllers. These constants were tuned experimentally (but with sufficient effect on the behavior of the controllers). Note also that we are not directly controlling the rotation speeds of propellers; instead, the controllers suggest the pitch, roll, yaw, and the vertical speed according to the errors on their input which results in the drone moving in the requested direction.

As mentioned above, the minimal altitude recognized by the ultrasonic sensor is about 22 cm. At this altitude the landing target is also hardly visible by the bottom camera as the target pattern is too close to the camera. Therefore we end the landing procedure at the altitude of 30 cm above the landing area where we switch off the motors. This final stage of landing resembles a controlled fall. That is typical for light drones as when they are close to surface, there is a lot of turbulence caused by their own propellers and the drones become very hard to control at low altitudes. Switching the motors off does not usually impose any danger as the drone does not fall from high altitude and their structure is usually shockproof enough to withstand it.

Implementation and User Interaction

We decided to implement our software on top of an existing application *ControlTower* (2013), which means that our software communicates with the AR.Drone through the JavaDrone API. ControlTower is a Java application providing a user interface to control an AR.Drone via a computer keyboard or a game controller. It also provides a visual view of basic sensor data including the camera view. We use *OpenCV* (2013) to do all graphical operations, for picture processing as well as for blob detection. OpenCV is a C++ library implementing many computer vision algorithms and general picture manipulation procedures.

We have implemented the *DroneLander* application module which is started from *ControlTower* and which provides its own user interface for defining the landing pattern and for controlling the drone (Figure 8). During the start maneuver, this software switches to the bottom camera, which is the only camera used for landing. The user can freeze a picture from the camera and set the HSV parameters to identify two color blobs in the picture (left and center panels at Figure 8). Anytime when the software recognizes these two blobs in the video stream, it displays a red circle between them that marks the landing point (see the right most panel in Figure 8). The user can also optionally select the horizontal angle (yaw) of the drone marked by an arrow for landing with respect to these two blobs (if not activated then the drone simply lands with any orientation).

The flight of a drone can be controlled manually through the keyboard or a game controller via ControlTower. When the user (pilot) spots the landing pattern in the picture from the bottom camera, it is possible to start the landing process. We provide two options for using the landing pattern. Either the drone lands onto the pattern (*land on target*) or the drone stays steady directly above the pattern at current height and at the required angle (if activated) (*lock on target*). The unexpected drone movements caused by external forces (wind, turbulence, user interaction etc.) are corrected by the landing software via the implemented controller. When the landing pattern disappears from the camera view, the system has a simple function of trying to find the pattern by going in the direction where the pattern was lost (*estimate target*).

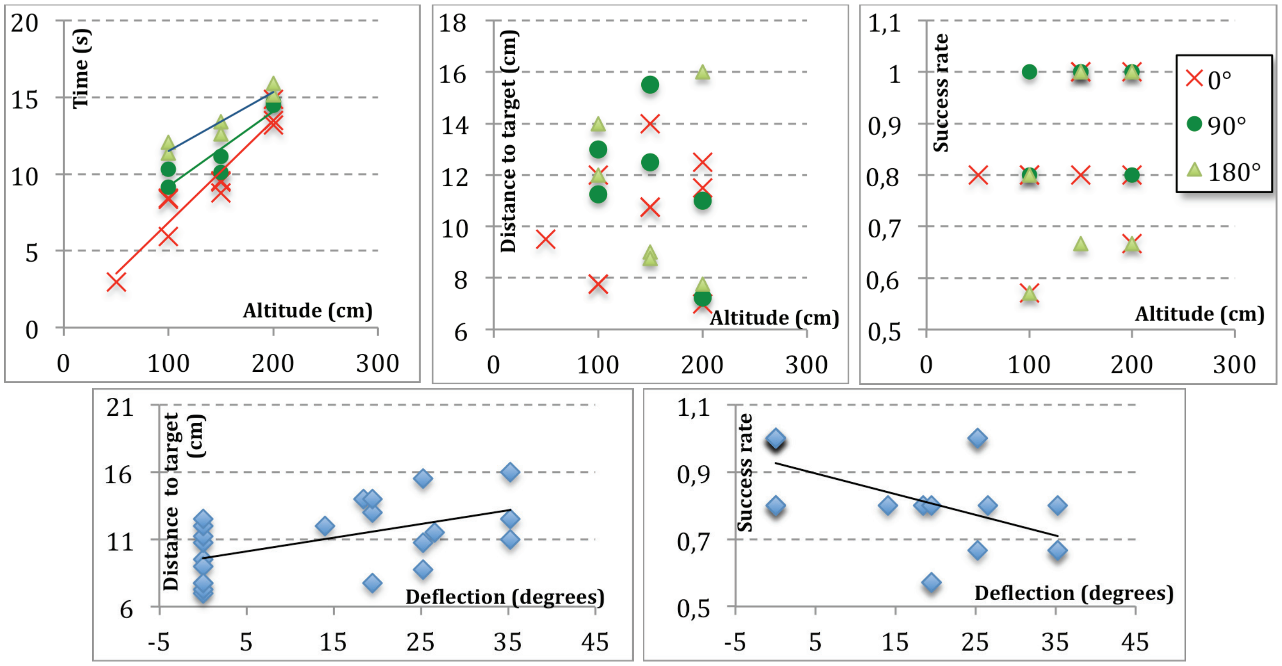

Figure 9. Dependence of landing time (left), precision (middle), and reliability (right) on the initial altitude (50, 100, 150, 200 cm) and orientation relatively to the target (0°, 90°, 180°). Dependence of precision and reliability on horizontal deflection (bottom).

Experimental Results

We did several experiments with the goal to measure reliability, precision, and speed of landing. We started the landing process at different altitudes and at different rotations relatively to the required landing orientation. We also tried different positions of the drone above the landing point. For each set of parameters, we repeated the landing process until we reached four successful landings. By a successful landing we mean that the drone landed at most 30 cm from the target point. Then we measured the *success rate* as 4/#landings. For the successful landings we measured the *time to land* (speed) and the *distance* of the drone (its bottom camera) from the landing point. The experiment was done indoors.

Figure 9 (top) shows the results as a function of starting altitude and orientation (some points overlap). The time to land depends linearly on the altitude and it was slightly longer for larger initial rotations. This is an expected result as the time depends on the distance to the target. Regarding reliability and precision there does not seem to be any dependence on the initial altitude and relative orientation.

We started the drone right above the landing pattern, but we also tried to shift it a bit so the pattern is on the side of the visibility area (measured as β – see Figure 7). The experiment confirmed that reliability and precision depends on this deflection as the target can be easily lost if placed at the border of the visibility area (Figure 9 bottom).

Conclusions

The paper presents an integration of several techniques from different areas, namely computer vision and control theory, with the goal to implement autonomous landing of an AR.Drone. Though these techniques are already known; the novel contribution here is the selection of the proper technique for a particular task and integrating those techniques to work in a real-life environment.

Acknowledgments

Research is supported by the Czech Science Foundation under the projects P103/10/1287 and P202/12/G061.

References

Brunelli, R. 2009. *Template Matching Techniques in Computer Vision: Theory and Practice*, Wiley.

ControlTower. 2013. https://code.google.com/p/javadrone/wiki/ControlTower. Accessed November 23, 2013.

King, M. 2010. *Process Control: A Practical Approach*. Chichester, UK: John Wiley & Sons Ltd.

OpenCV. 2013. http://opencv.org/. Accessed November 23, 2013.

Parrot SA. 2013. *AR.Drone 2.0*. http://ardrone2.parrot.com/. Accessed November 23, 2013.

Szeliski, R. 2010. Computer Vision: Algorithms and Applications. Springer Verlag.