BioMAV: the package-inspecting robot

Robin Wellner and Roland Meertens

May 1, 2013

1 Introduction

BioMAV (which stands for "Biologically inspired Micro Air Vehicles" is a recurring project of the department of Artificial Intelligence at the Radboud University in Nijmegen. This report contains an overview of this years project, the approaches taken, materials used and the result.

1.1 Drone

The robot of this project is the Parrot AR.Drone 1.0. It is able to communicate with a laptop using USB or Wi-Fi. The drone has two cameras, one ground-facing and one forward-facing. However, the driver software used was only able to access the image data from the forward-facing camera.

1.2 ROS

ROS (for Robot Operating System) is an open-source framework for robot controller software and is especially intended for the re-use of libraries (called "nodes") between different projects and for different types of robots. This allowed us to make use of existing solutions for computer vision, drone control and inter-node communication.

1.3 Vision

For the processing of camera images the CMVision library was used. This library contains several computer vision techniques which apply low level image processing techniques to segregate blobs from images. More about the vision component can be found in section 3.1

2 Task

The goals of this project were:

1. To establish a platform, based on the old BioMAV project, on which future BioMAV projects can build.
2. To let the drone perform a task autonomously.

3. To demonstrate the drone performing that task.

The task that the drone had to perform was a search and deploy task. In this task the robot searches for a goal where it has to perform a simple task. This led to task where the drone inspects a package by flying towards this package.

Initially a fly-to-object behaviour was created. As this does not yet give way to very advanced behaviour an implementation for a breadcrumbs finding navigation task was chosen as task.

In this breadcrumbs navigation task the drone is able to locate a goal object by following other objects. Every time a temporary goal has been reached it will start searching for the next goal.

3 Method

3.1 Blob detection

3.1.1 Using CMVision

We found a ROS package called CMVision that could find rectangular blobs of roughly the same color in images, and could do that in real time. We used it for every task to both find targets and recognise corridors (see **??**).

CMVision comes with a tool, called colorgui, which allows one to calibrate the range of values used to identify blobs. The dramatic influence of environment lights on the colors meant that each time the environment changed the colorgui tool had to be used again to recalibrate the colors.

In our initial setup (with ROS Electric), CMVision did not specify correctly which color each blob was. That meant a few functions had to be implemented that manually checked the colors of the camera pixel values. (See xxxx).

3.2 Heat map

3.2.1 Idea

At some point, it became clear that the blob detection was not consistent. Small blobs were often found where none should be, and the blob corresponding to the target fell away for a frame every so often. This confused the drone, making navigation less than optimal.

As a solution a heat map was implemented which would only activate after blobs appeared consistently on a certain position. At the same time, a single missed frame wouldn't mean that the drone had lost its target.

3.2.2 How it works

The heat map is basically a matrix, each cell corresponding to a square of pixels from the drone camera. Every blob detected would then increase the values in the heat map inside that blob.

Each time step, the matrix is multiplied by a constant $c_{cooldown}$, where $0 < c_{cooldown} < 1$, reducing the overall activation.

Only if the maximal value in the matrix is more than a certain constant, the drone considers doing anything based on the heat map that time step. Otherwise, it will turn around slowly, looking for the target.

If the sum of all values in the matrix is more than another constant, that means a large portion of the camera's field of vision is taken up by the target, which means we're close to it. That means the drone has found the target, and as such can continue to its next target.

In any case, each column in the matrix is summed, such that we have an array of values. Each of these values is the total activation in a specific column of the heat map. A high activation at a certain position means it is likely that the target is in that position, and that means the drone should fly in that direction. To figure out the "centre of gravity" of the activation, we calculate the weighted average of the indexes of the columns, with the array of values as weights. The result can be normalised, giving us the direction the drone should fly.

3.2.3 Implementation

We used NumPy to deal with the matrix calculations.

The implementation has several constants that were abstracted from in the previous section.

The multiplication constant $c_{cooldown}$ was taken to be 0.5. In our experience, larger values caused "after images" to persist longer than needed, and smaller values reduced the positive effects of the heat map too much, making jitters and missing blobs too strong.

The "downsize factor" by which the matrix was scaled down from the drone camera was 4, meaning that each cell in the heat map represented 16 pixels. This was chosen because a smaller downsize factor would make the algorithm that activated the heat map too slow, and with a downsize factor of 4, the resolution preserved was still sufficient for subtle steering decisions.

The activation added for each blob depends on the blob area A. This was another way to reduce the effect of jitters. The formula that worked best for us was

$$activation = 50 + \frac{\sqrt{A}}{10}$$

The threshold that decided whether the drone has seen *anything* interesting was set to 16, fairly arbitrarily.

The threshold for deciding the target has been found and the drone was finished or needed to find another target, is strongly dependent on the real life size of the target. Our targets were about $\frac{1}{4}m^2$ and the constant that worked best was 30000.

3.3 Control

3.4 Flying towards objects

```
direction = XvalueOfAverageOfActivation
```

```
turnTowards(direction)
flyForward
```

XvalueOfAverageOfActivation is determined by using this formula: (as explained in 3.2.2)

$$\texttt{XvalueOfAverageOfActivation} = 0.5 - \frac{Avg}{i_{max}}$$

$$Avg = weighted - average([0, ..., i_{max} - 1], weights = weights)$$

$$weights_i = \sum_{j=0}^{j_{max}-1} H_{ij}$$

H = (heat map matrix, i_{max} columns, j_{max} rows)

3.5 Package inspection

```
if SumActivation > threshold
        currentTarget = getNewTarget(currentTarget)
    else
        startLanding
```

The sum of activation if determined by using this formula: (as explained in 3.2.2)

$$\texttt{SumActivation} = \sum_{i=0}^{i_{max}-1} \sum_{j=0}^{j_{max}-1} H_{ij}$$

H = (heat map matrix, i_{max} columns, j_{max} rows)

4 Results

4.1 Github

The source code to the ROSMAV project (as well as this report) are accessible on Github. This is because it made working with multiple team members reasonably convenient. Additionally, it makes ROSMAV easily accessible to others, as they can simply clone or fork the repository from Github.

References

[1] International micro air vehicle conference and flight competition URL: *http://www.imav2011.org/*.

Appendix

5 Background

5.1 IMAV

IMAV is a series of conferences and competitions with as main objective "to provide an effective and established forum for dissemination and demonstration of original and recent advances in MAV technology."[1] MAV stands for Micro Air Vehicles and IMAV stands for International Micro Air Vehicles.

5.2 BioMAV

The previous and first BioMAV project started in 2010. It was the entry of Radboud University Artificial Intelligence department. BioMAV was a large success and obtained the third price in the IMAV 2011 pylon challenge.

5.3 Biological inspiration

This project's biological inspirations are mostly related to vision. The corridor-following task described in section **??** was inspired by how flying insects such as moths follow the light of the moon for navigation. The packet inspection task described in section 3.5 evolved from animals hunting their prey.

In this section several of the implemented algorithms are discussed. Every subsection starts with the code in pseudo code and ends with an explanation of the algorithm.

6 Tutorial

6.1 Installation

To install ROSMAV, follow the following steps:

1. Install Ubuntu (we used Ubuntu 10.10).
2. Install ROS (we used Electric) by following this guide:
   http://www.ros.org/wiki/electric/Installation/Ubuntu
3. Make sure that you performed the "Environment setup" step during installation.
4. Download our repository from https://github.com/dutchcheesehead/ROSMAV
5. Navigate to ROSMAV.
6. Run **./install.sh**.
7. Close your terminal window.

Figure 1: A picture of a drone.

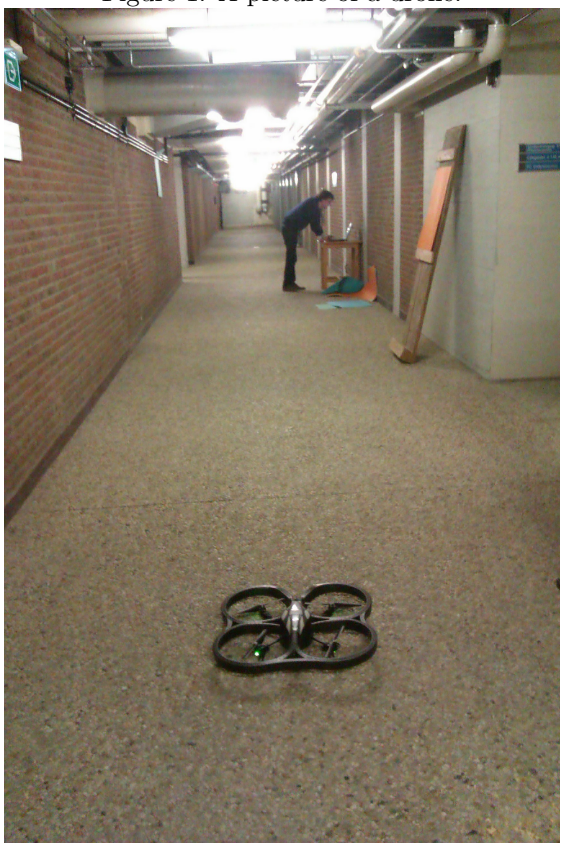
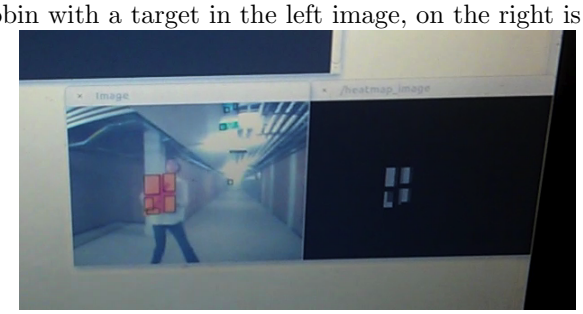


Figure 2: Robin with a target in the left image, on the right is the activation

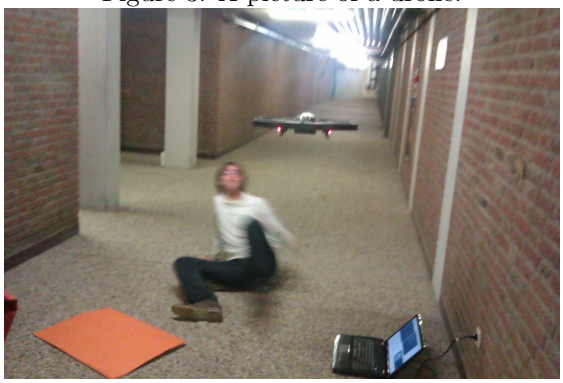


6.2 Starting ROSMAV

To start our software, run the following commands all in different terminal windows:

1. **roscore**
   This starts roscore.
2. **rosrun ardrone_brown ardrone_driver**
   This starts the driver for the AR-Drone.

Figure 3: A picture of a drone.



3. **roslaunch cmvision blobs.launch**
   This starts the blob detection process.
4. **drone_teleop.py**
   This is needed to let the drone take off and land.
5. **rosrun image_view image_view image:=/heatmap_image**
   This allows you to see the heat map activation for the "inspect presents" task, and thus see through the eyes of the drone.
6. **rosrun ROSMAV inspectPresents.py** *or*
   rosrun ROSMAV followLights.py
   This actually starts the task.

6.3 Adding different colors to inspect

Suppose you want to add a different color to inspect.

1. Run **roscore**
2. In another terminal window, start the AR-Drone driver: **rosrun ardrone_brown ardrone_driver**
3. Start the ColorGUI: **rosrun cmvision colorgui image:=/ardrone/image_raw**
   You will see a window with the camera images from the AR-Drone.
4. Resize the window so you can see the text fields.
5. Keep the object you want to inspect in the view of the drone.
6. Click on the image of that object in the ColorGUI on a few different places.
7. Move both the object and the drone a bit around and click some more, to account for changes in lighting.
8. Repeat until you are confident it recognizes the object correctly in different circumstances.
9. If you made a mistake, close the ColorGUI and go back to step 3.
10. If it recognizes the object correctly, open "cmvision/colors.txt". Add both the color and the threshold. For the color, you can copy the last color and change the first bit into

what the ColorGUI said, and change the name of the color. For the threshold, paste the threshold after the last threshold.

11. Edit “inspectPresents.py”. You might need to add a function like “isRed” if it’s not already in there. Then modify the “nextTarget” dictionary to the sequence you want to inspect the presents.

Figure 4: A picture of a drone.