BioMAV: the package-inspecting robot

Robin Wellner and Roland Meertens

May 1, 2013

1 Introduction

BioMAV is a recurring project of the department of Artificial Intelligence at the Radboud University in Nijmegen which stands for "Biologically inspired Micro Air Vehicles". In this report an overview will be given about what the project goals were this year.

1.1 Drone

The robot used in the BioMAV is the Parrot AR.Drone 1.0. It is able to communicate via USB and Wi-Fi. The drone has two cameras, one ground-facing and one forward-facing. However, the driver software we 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, especially intended for 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 blob detection (used in section 10.2 and inter-node communication.

1.3 Vision

For vision the usage of CMVision turned out to be very handy for the detecting of blobs of colours. This was the reason that we used this for the detection of our goals.

2 Task

2.1 Initial task

The initial 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. Which task that would be was to be determined.
3. To demonstrate the drone performing that task.

2.2 Final goal

Our final goal was to have the drone perform the packet-inspection task and light-following task.

Initially we managed to create a fly-to-object behaviour. As this does not yet give way to a very advanced behaviour it was decided to add an implementation for a breadcrumbs finding navigation 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 Vision

3.1.1 Blob detection

3.1.2 HeatMap

3.2 Control

Insert behaviours here (but short...)

4 Artificial intelligence

What kind of AI do we use??

5 Results

5.1 Github

5.2 Use case

6 Appendix

7 Introduction

8 Background

8.1 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.

8.2 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.

8.3 Biological inspiration

This project's biological inspirations are mostly related to vision. The corridor-following task described in section 10.2 was inspired by how flying insects such as moths follow the light of the moon for navigation. The packet inspection task described in section 10.4 evolved from animals hunting their prey.

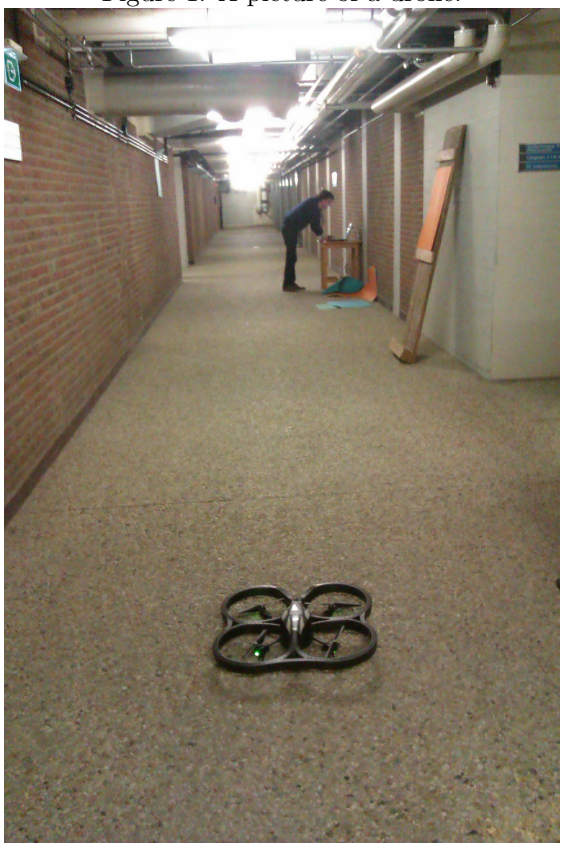
9 Motivation

One of the wishes of the group was a code implementation using ROS, this was another software part that both teams had to figure out how to use. Around June the first implementation of a control program using ROS was finished by Roland. This program consisted of a server extension of the original software written by the first BioMAV team. The ROS component connected to a socket that listened to calls coming from the ROS component of the software. Using a simple syntax to parse fly-commands it was possible to fly with the ARDrone using ROS.

After that, we started to look on how to perform tasks, and we decided to use a library for computer vision, which eventually became blob-detection, and a "driver" ROS node that turned information from the camera, processed with blob-detection, into actions, to perform a task.

10 Pseudocode

Figure 1: A picture of a drone.



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

10.1 Detection of objects

```
targets = getAllBlobsThatMatchOurTarget
coolDownHeatMap
activateHeatMap(targets)
if dot in heat map is high enough
    if sum of heat map is high enough
        inspect package
```

```
    else
        go toward average of activation
```

Our program first determines that targets we have by acquiring all blobs that match our target. After this is done our heat map is cooled down multiplying every activation value by 0.5. The heat map then receives activation again by raising the value of all the locations of the detected blobs. When a dot in our heat map is activated enough it is assumed that this is because our target is at this position. When the total sum of the heat map is activated enough we "inspect" the package (see section 10.4). When there is not enough activation the drone flies towards the average of the activation (see section 10.3).

10.2 Corridor following

```
targets = getAllBlobsThatMatchOurTarget
myTarget = getTargetLowestOnCamera
if isVeryMuchToTheLeft(myTarget)
    flyVeryHardToTheleft
else if isVeryMuchToTheRight(myTarget)
    flyVeryHardToTheRight
else if isSlightlyToTheLeft(myTarget)
    flySlowlyToTheLeft
else if isSlightlyToTheRight(myTarget)
    flySlowlyToTheRight
else
    flyForward
```

We found out that by always flying towards the target that was the "lowest" on the camera (in the y-space) our drone could smoothly follow the lights in the hallway. By implementing this function we were able to get a corridor-following behaviour. Our program first gets all blobs that are matching our target colour (the color of the lights). The blob with the lowest y value is chosen as our primary target. When this blob is very much to the left or right the drone will turn with a great speed in this direction. When this blob is only slightly to the left or right the drone will turn slowly in this direction. Otherwise the drone will fly forwards this blob.

10.3 Flying towards objects

```
direction = XvalueOfAverageOfActivation
turnTowards(direction)
flyForward
```

Flying towards a target was first solved by only flying towards a specific blob. As soon as we started working with objects that were not detected as a "whole" blob and we had an activation value we thought of another function. In the new fly-towards function the X value of the average of the activation is taken. This value is divided by the length of the camera image and mapped to our drone in such a way that we acquire a directional value. We let the drone "spin" in this direction and fly forwards.

10.4 Package inspection

```
if sum(activation) > threshold
    if getNewTarget(currentTarget)
        currentTarget = getNewTarget(currentTarget)
    else
        startLanding
```

For the "inspection" of the packages there is not a really special behaviour. As soon as the total activation in the heat map for the packages exceeds the set threshold the next target is chosen. When there is no new target specified in our goals-array the drone will land.

10.5 Heat map

10.5.1 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.

We came with the idea of a heat map, 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.

10.5.2 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.

10.5.3 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 $50 + \frac{\sqrt{A}}{10}$.

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.

10.6 Color detection

(why we have to use isRed, isGreen, isBlue... and how it works)

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

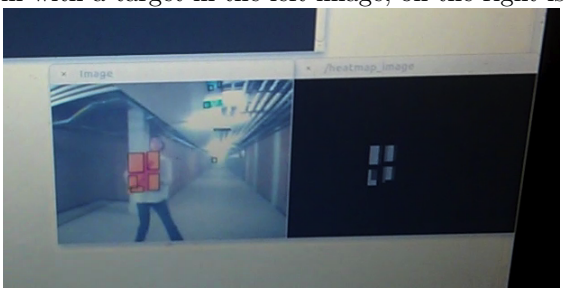
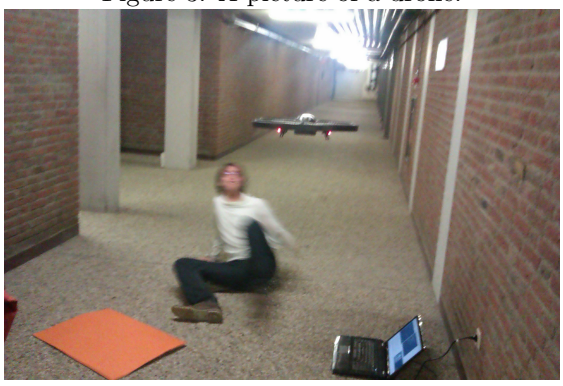


Figure 3: A picture of a drone.



11 Tutorial

11.1 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.

11.2 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.
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.

11.3 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.

References

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