Localization of Drone using IMU and Video Stream

Guryash Bahra, Hauke Kaulbersch, Julian Pape-Lange,
Kolja Thorman and Marc Zöller

September 30, 2016

1 Introduction and Goals

For our project, we worked with the Parrot AR.Drone 2.0. To control it, the PS-Drone API was used [1]. The main goal of the project was to create a program for localizing the drone to enable simple navigation. Therefore, the internal sensors of the drone were used in combination with the detection of ArUco markers [2].

At the beginning, we set milestones to be accomplished in the three weeks we had for the project. First of all, the drone should use its internal sensors to detect its own movement, so that it can estimate its flight path. Furthermore, the drone should detect ArUco markers and determine its relative position to those markers.

Once this was accomplished, these two functionalities were to be combined. The drone should be able to measure its position by detecting markers. Then, it had to use this information to correct its current position created by dead reckoning. We decided to apply a Kalman filter to fuse the dead reckoning with detected markers. After this localization was possible, the drone should be able to fly over a known map and localize itself.

Regarding the complete path, we aimed to apply an algebraic method for smoothing the whole trajectory. Our code is publicly available on GitHub [3].

2 Movement Detection

One part of the localization should consist of the internal movement measurements of the drone for dead reckoning. The API provided several data packages from the drone, which we used.

First, we looked at the raw accelerometer and gyrometer data. The problem here was that the acceleration measurement yielded high values, even when the drone was not moving. In addition, slight tilts already created strong deviations. However, the drone's data package "demo" included a better solution, namely the estimated speed, which was calculated using the accelerometer and gyrometer values.

Using this solution was only possible with the drone in flight, otherwise no velocities were provided by the drone. Therefore, we implemented a way to control the drone. To simplify the control, we reduced the speed from the default 20% to 5%. To calculate the position based on the estimated speed, we multiplied the x and y velocities by the time difference between the receipt of the current and the last data package. Here, we found that the x, y-velocities of the drone were aligned with the drone's main axis. We also defined the starting

[1] See `www.playsheep.de/drone/`

[2] See `www.uco.es/investiga/grupos/ava/node/26`

[3] `https://github.com/Ennosigaeon/sensor-data-fusion.git`

position of the drone as (0, 0) for simplicity (see Figure 1). Thus, the velocities are in drone coordinates and must be translated into real world coordinates.

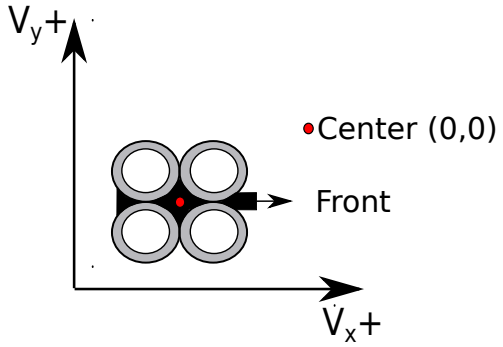


Figure 1: Drone coordinate system

To solve this problem, we used the drone's yaw value (rotation angle of the z-axis) provided by the navigation data package. By printing this value while turning the drone in different directions, we determined that the yaw is 0 where the drone points when initialized and that it shrinks to -180 when turning rightwards and grows to 180 when turning leftwards. Consequently, the yaw values can only be evaluated relatively to the initial drone orientation and not to a global coordinate system. To make the yaw 0 when aligned with the x-axis of the map, we saved the yaw after we positioned the drone and then substracted that value from the measured yaw. Using the yaw in combination with the velocities, we were now able to calculate the drones position in relation to its starting point (see Figure 2). Yet, turning the drone around the yaw axis increased the measurement error significantly, thus we decided to move the drone sideways instead of turning it.

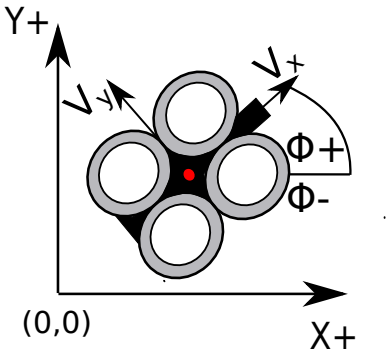
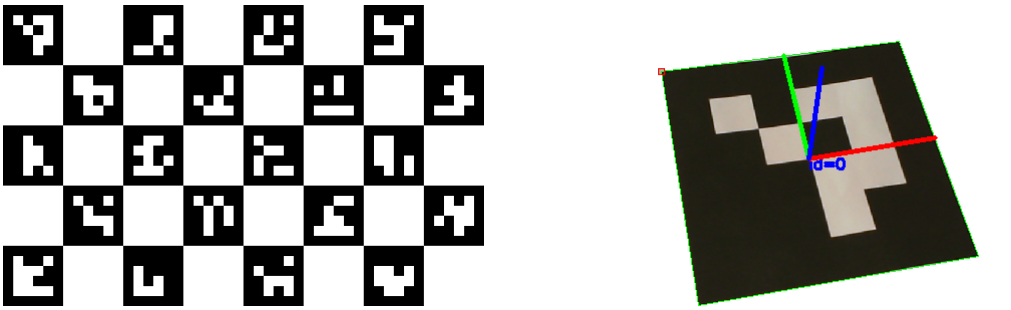


Figure 2: Drone in real world coordinates

After this was accomplished, we implemented a functionality to plot the drones movement. For this, we used the Python package *matplotlib.pyplot*, since it did provide a way to plot the path in realtime. We also saved the data for later offline testing and visualization.

3 Marker Detection

To localize the drone in a map, landmark detection, in the form of markers, was necessary. We first tried the marker detection with the markers delivered with the drone. This worked, however, for building a map with different landmarks, we decided that ArUco markers (See: Figure 3 would be a more suitable option. For marker detection we used OpenCV's ArUco detection module.



(a) Calibration board (b) Augmented ArUco marker

Figure 3: ArUco markers

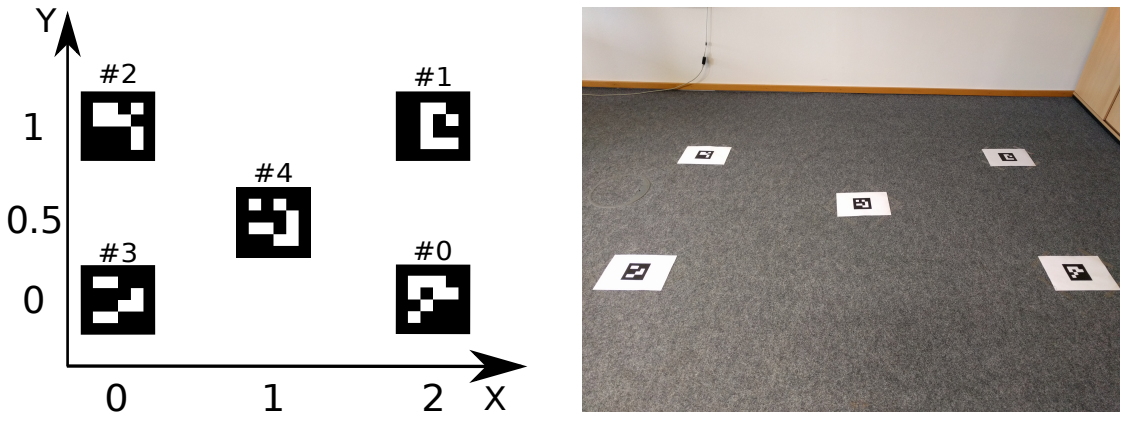
The basic idea behind the detection is to convert the images received from the video stream to gray-scale. Then, edge detection is used. Finally, the resulting grid is searched for patterns corresponding to the ArUco markers. That way, it is also possible to mark them on the video stream. Furthermore, for each marker, the distance of the camera to the marker in world coordinates is provided by a translation and rotation matrix. Using this information and knowledge about the marker size, it is possible to get the offset between the camera center and marker center in world coordinates.

4 Localization in known Map

Our next step was to combine what we did up until now. The drone should measure and plot its own movement using dead reckoning and when detecting a marker in a known map, it should be able to determine its actual position and correct its prior flight path estimation.

First, we needed to create a map. It consisted of five markers. Four of them were positioned in the corners of a $2m \times 1m$ rectangle, the fifth was put in the center of it (see Figure 4).

When started, the drone assumed to be at position $(0,\ 0)$ with the current orientation aligned with map. This was an assumption that was made to allow for an early maker detection. Using its movement detection, it calculated and saved its flight path. Each time it detected a marker, the drone used the marker's position on the known map combined with the rotation and translation matrix between itself and the marker to determine its own position, using



(a) Scales (b) Map

Figure 4: Map

Rodrigues' rotation formula [4]. In case multiple markers were detected, it uses the average position as a measurement. Following that, it used the difference between its estimated position based on the movement and its estimated position based on the marker detection to correct its saved positions of its prior flight path.

We first tested the scripts with test data and later with actual drone data. A problem we encountered here was the strong delay of the video stream. We determined that this was the result of path plotting during dead reckoning. We resolved this issue by adjusting the plotting commands.

For later replication of the drone flights and to be independent of the drone, we implemented methods to save all of the estimated positions of the drone and all detected marker positions.

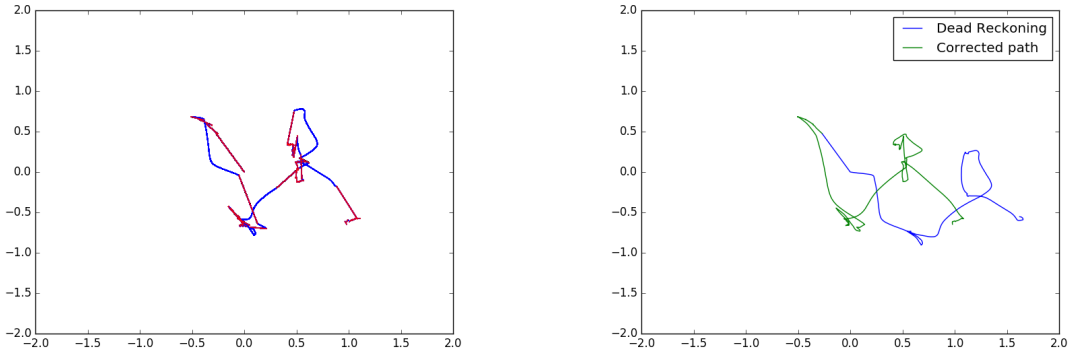
Another problem we encountered was a permanent drift in the orientation. This resulted in the drone assuming to rotate around the yaw axis, even though no movement was made. We solved this by first measuring the drift for five minutes, while the drone was on the ground. To calculate the drift rate, we used linear regression. By modifying the raw yaw value by 0.007 05 degree per second, we were able to limit the drift to roughly 1 degree per minute.

Furthermore, the ArUco markers used a different coordinate system than the drone. That means that for example the direction the drone assumed to be negative y was positive x for the markers. We solved this by inverting the axis to make the drone's and the ArUco marker's vector system compatible.

After these problems were solved, we implemented a Kalman filter [5] to fuse dead reckoning with measured positions. For this, we needed to determine the process and measurement noise. We measured the scatter of the recorded movement while the drone was in air without moving. We combined this with the deviation from the estimated position after moving the

[4] See `https://en.wikipedia.org/wiki/Rodrigues%27_rotation_formula`
[5] See `https://en.wikipedia.org/wiki/Kalman_filter`

drone to get the estimated process noise (0.09metre) per prediction step, in both dimensions). We also measured the scatter of the camera's marker detection, in a similar fashion, to determine the measurement noise (0.5metre) per update step, in both dimensions). With that, we were able to use the Kalman filter on our input to get a better localization (see Figure 5).



(a) Path using dead reckoning and marker detection

(b) Smoothed path and path with only dead reckoning

Figure 5: Kalman implementation

Now that the filter worked, we decided to use the drone controls to fly it. A problem we encountered here was that the drone was hard to stop, resulting in it moving passed markers very often. To make sure it would see markers almost all the time while navigating it, we printed 25 more of them, creating a 2metre $\times$ 2.5metre map made out of 30 markers (see Figure 6. On this map, we could use the drone controls.

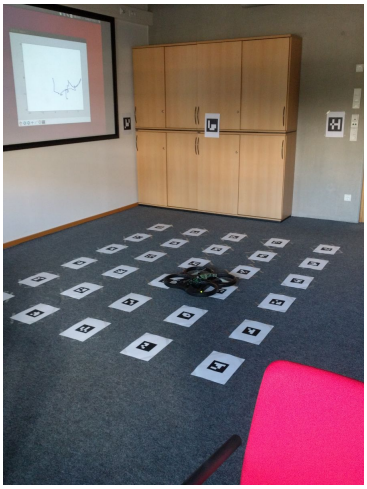


Figure 6: New map

We discovered another problem. Due to uncertainties or false detection, the drone's estimated distance to a marker was sometimes much higher than it should have been, resulting in some outliers in the position estimation. We solved this by implementing a threshold to exclude

distance vectors longer than 0.25metre. Due to these changes, we were able to reduce the influence of false detections.

Since there were still jumps when a marker was detected, we used the following algebraic algorithm to smooth the measured flight path: The assumption to start at (0, 0) is likely to be wrong. Therefore, the first marker being detected with high confidence is used to calculate the starting position. This is done by calculating the distance between the marker and the expected position and shifting the old values by this distance. For all following marker detections the path between the two most recent markers is adjusted by calculating the distance between the new marker and the expected position, and assuming that this error increased linearly over time (see Figure 7). This algorithm works good if the marker detection works well and the expected path is not curved. For curved paths the direction orthogonal to the direction from the second newest marker and the expected position can not be corrected.

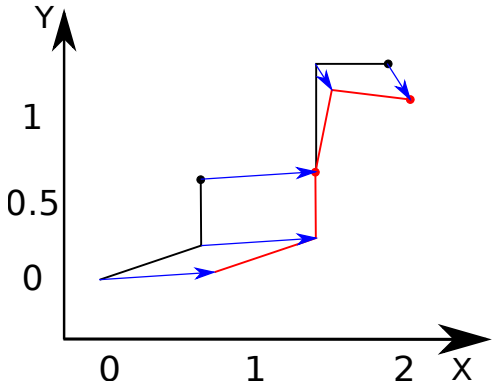


Figure 7: Path correction

5 Conclusion

In this project, we used marker detection, dead reckoning and Kalman filter to enable a drone to determine its position and flight path in a known map. We did so by fusing the drone's internal velocity estimation with the detection of ArUco markers.

Problems we encountered were that the dead reckoning alone drifts significantly. Furthermore, the trajectory could be improved by using more markers.Triangulation could have been used to improve the position estimation. However, the small camera viewport made detection of multiple markers difficult. In future projects, an extension by a SLAM implementation could theoretically be possible.