First Session
Visual Navigation for Flying Robots Workshop

Jürgen Sturm, Jakob Engel, Daniel Cremers
Computer Vision Group, Technical University of Munich

Bergendal Meetings
17.06.2013

Introduction

We split the first hands-on session in two parts:

1. In the first part, we set up the AR.Drone, perform a manual flight with the joystick, and record a bag file (logfile). For this task you will require a Parrot AR.Drone, a PS3 joystick and some space to fly around.
2. In the second part, we provide a quick intro on how to inspect and process logged sensor data in ROS, and how to integrate the sensor observations of the AR.Drone in a Kalman filter to estimate the position of the quadrocopter. This task does not require access to an AR.Drone or a joystick.

As the two parts are independent of each other, we encourage half of the workshop's participants to start with the second exercise, such that not all groups fly at the same time. This sheet assumes that all preparatory steps from the "installation instructions" sheet have been done (i.e., a working ROS environment, pre-compiled AR.Drone drivers, etc).

Part 1: Manual Flight

(a) Connect the AR.Drone battery.

(b) Connect to the AR.Drone WLAN.

(c) Start the ROS master, the AR.Drone driver and rviz (each in a separate terminal window).

```
$ roscore
$ rosrun ardrone_autonomy ardrone_driver
$ rosrun rviz rviz
```

(d) Add an "Image" display to RVIZ and change the "Image Topic" to `/ardrone/image_raw`. You should see the live-camera stream from the quadrocopter's frontal camera.

(e) Plug in the joystick, and start the joystick driver using `rosrun joy joy_node` (maybe add the parameter `_dev:=/dev/input/jsX`, where X is the number of the joystick device). Inspect the topic `/joy`, and verify that the joystick works - you might have to press the P button once to initialize it.

(f) The `ardrone_joystick` node we provided translates the raw joystick messages to the correct control commands sent to the quadrocopter. Start it using `rosrun ardrone_joystick ardrone_teleop`. The axes and buttons are assigned as follows:

- The **R1 button** toggles the emergency state of the robot. Pressing R1 while flying will stop the rotors immediately. If the LED beneath the rotors are red (for example, after a crash), press R1 to reset the quadrocopter.
- The **L1 button** starts the motors of the quadrocopter. It also works as a deadman switch so that the robot will land if you release it during flight. The quadrocopter will ascend to one meter above ground and tries to hold this position.
- The **left stick** can be used to control the roll and pitch angle of the drone. Keep in mind that these velocities are given in the local frame of the quadrocopter!
- The **right stick** controls the yaw-rate and the altitude.
- The **select button** can be used to switch between the two cameras. This can also be done by executing `rosservice call /ardrone/togglecam`.
- The **triangle button** can be used to switch off the on-board position stabilization: Per default, the quadrocopter hovers (i.e. stabilizes it's horizontal position by keeping v_x and v_y at zero) when you do not touch the left control stick. It even actively decelerates when letting go of the left control stick.

  Pressing and holding the triangle button will switch this off, i.e. give you direct control over roll and pitch at all times – note how this leads to rapid drift in horizontal position.

(g) **First Flight:** You are all set up to fly the quadrocopter! Fly a short round through the lab to practice your flying skills. You can try to take a group picture of your team while flying!. *Tip: If the quadrocopter does not take-off vertically but immediately shoots away, you need to calibrate the IMU. This is done by placing the quadrocopter on a horizontal surface, and calling* `rosservice call /ardrone/flattrim`.

(h) Use `rxgraph` to visualize the running nodes and used topics. Check which nodes are there and over which channels do they communicate.

(i) Finally, we record a bag file that we can use for the second part: First, put the visual marker on the ground and fixate it using duct tape. Switch to the bottom camera (the AR.Drone only streams one of the two camera images to the PC), and use rosbag to record approximately a 30s-flight. The marker should regularly be visible in the camera image, but should leave it temporarily. You will need to record at least the camera image topic, the camera info topic, and the ardrone navdata topic, i.e.:

```
$ rosbag record /ardrone/navdata /ardrone/image_raw /ardrone/camera_info
-O flight.bag
```

Part 2: Extended Kalman Filter

In this exercise, you will examine and test the extended Kalman filter which estimates the pose of the robot from a bag file or live data. You can either use the `flight.bag` file we provided (which you downloaded as part of the preparation sheet):

`http://vision.in.tum.de/~engelj/flight.tar.gz`

or your own bag file recorded in part 1. Exit all running nodes still running from part 1, and disconnect the quadrocopter and joystick, you will not need them for this exercise.

From the quadrocopter, we receive (1) relative motion estimates from the optical flow sensor and (2) absolute pose estimates from the marker detector. The EKF prediction step integrates the relative motion estimates. As the motion estimates are noisy, small errors will accumulate and lead to drift. In the EKF update step, we can correct for this drift by integrating the absolute pose observations from the marker detector.

C++ Framework For this exercise, we provide a C++ framework that implements an extended Kalman filter (EKF) in the `visnav_exercise` package. The Kalman filter estimates the quadrocopter's pose from the onboard sensor readings, as well as visual observations of a marker on the floor using the down-facing camera. For simplicity, we model the quadrocopter only in the 2D plane, i.e., its state at time t is described by $\mathbf{x}_t = \begin{pmatrix} x_t & y_t & \psi_t \end{pmatrix}^\top$.

Using the Eclipse IDE If you like to use Eclipse for editing and viewing the C++ source code, `cd` into the `visnav_exercise` folder, and type `make eclipse-project`. Afterwards start Eclipse from the shell, and use *File→Import*. Select *General→Existing Projects into Workspace*, select the *visnav_exercise* as root folder, and click finish.

EKF Prediction (pure odometry)

(a) Replay one of the provided bag files and inspect the topics (`rosbag play, rostopic list, rostopic echo`). Which topics are there? What data does the `/cmd_vel` and the `/ardrone/navdata` topics contain?

(b) Use `rxplot` to visualize vx and vy from `/cmd_vel` and from `/ardrone/navdata`. What is the relation between `/cmd_vel/linear/x` and `/ardrone/navdata/vx`? Estimate (roughly) the delay between steering commands and the onset of the motion.

(c) Visualize the camera images using rviz by adding an image display and selecting the proper topic `/ardrone/image_raw`.

(d) Run the pose-estimation EKF using `rosrun visnav_exercise ekf`. Have a look at the state prediction function (`ExtendedKalmanFilter::predictionStep`), which is implemented in `EKF.cpp`.

(e) The node publishes the trajectory taken by the quadrocopter using `MarkerArray` and `Marker` messages. Visualize these messages in rviz using a marker display. Also add a tf and a grid display. You have to change the base coordinate system to `/world`. You should now see the trajectory of the quadrocopter, as well as the current pose-estimate and covariance ellipse.

(f) Visualize the full estimated two-dimensional trajectory from the bag file using

```
rxplot -p 48 /ardrone/filtered_pose/linear/x:y
```

i.e., restart the playback after starting `rxplot`. Take a screenshot of the whole trajectory for later comparison.

EKF Update (visual observations)

(a) The framework detects markers in the environment and provides (in the case of a detection) an observation $\mathbf{z} = \begin{pmatrix} x & y & \psi \end{pmatrix}^\top$ *relative* to the frame of the quadrocopter. Have a look at the observation function, which is also implemented in `EKF.cpp` and switch it on (i.e., remove the pre-emptive return).

(b) Use `rxplot` again to visualize the - now corrected - estimated two-dimensional trajectory from the bag file, take another screenshot, and compare it to the previous one. Can you see the EKF's drift without visual observations?

(c) From the screenshots, estimate roughly how far the pose-estimate drifted without visual markers over the 48s flight (in meters).

(d) Determine the effect of changing the observation noise variance R (in the EKF's init function) - increase / decrease it by a factor of 100 and observe the effect on the estimated trajectory.