Visual Navigation for Flying Robots
D. Cremers, J. Sturm, J. Engel, C. Kerl
Summer Term 2013

Computer Vision Group
Department of Informatics
Technical University of Munich

Sheet 3

Topic: Robot Control

Submission deadline: Tue, 11.06.2013, 10:15 a.m.
Hand-in via email to visnav2013@vision.in.tum.de

Software Framework: Installation

(a) Download the ROS package for this exercise

```
$ cd ~/fuerte_workspace
$ git clone git://github.com/tum-vision/visnav2013_exercise3.git
$ rosmake visnav2013_exercise3
```

(b) Update the `ardrone_joystick` package

```
$ roscd ardrone_joystick
$ git pull origin
$ rosmake ardrone_joystick
```

Software Framework: Usage

To reduce the number of terminal windows we provide a ROS launch file, which starts the marker tracking, joystick drivers, EKF, RVIZ and `dynamic_reconfigure`. For this exercise you have to run at least the following commands (each in a different terminal):

```
$ roscore
$ rosrun ardrone_autonomy ardrone_driver
$ roslaunch visnav2013_exercise3 bringup.launch
$ rosrun visnav2013_exercise3 ardrone_controller
```

The `ardrone_controller` node implements automatic position control of the AR.Drone using a PID controller, which you will implement in this exercise. The `ardrone_joystick` node has been updated to easily switch between joystick and automatic control. When you press the **start** button the automatic controller will be enabled. Pressing the **start** button again disables the automatic controller. When you move the left or right stick the automatic controller will be disabled as well. As in the previous exercise, you have to hold the **L1** button all the time otherwise the AR.Drone lands.

Exercise 1: Implementation of Position Control

In this exercise, you will derive the PID controller equations and implement them.

(a) **Specify the control law $u(t) = f(e(t), \dot{e}(t))$ of a PID controller**, i.e., specify how the control command u at time t is computed from the error $e = x_{\text{desired}} - x_{\text{current}}$.

(b) **Define how the error integral $e_{I,t}$ can be computed in the discrete case** from its last value $e_{I,t-1}$ and the current error e_t with $e_{I,0} = 0$.

(c) **Define a formula for the discrete error derivative** $e_{D,t}$ given e_t and e_{t-1}.

(d) **Specify the discrete PID control law**, i.e., $u_t = f(e_t, e_{I,t}, e_{D,t})$.

(e) Implement the PID controller in the `PidController::getCommand` functions in `src/controller.cpp`. There are two version of this function one requiring the error derivative and one computing the derivative from the current and the last error. Implement the derivative computation in

```
float PidController::getCommand(const ros::Time\& t, float error)
```

which calls the second version. Implement the integral computation and the control law in

```
float PidController::getCommand(const ros::Time\& t, float error,
    float derror)
```

Use the member variables `c_proportional`, `c_integral` and `c_derivative` for the respective gains.

(f) Now we want to control the x-position, y-position, and yaw-angle of the quadrocopter, using three independent PID controllers. **Specify how the error signals $\mathbf{e} = (x_e, y_e, \psi_e)^\top$ for each of these controllers can be computed from the current pose $\mathbf{x} = (x_x, y_x, \psi_x)^\top$ and the goal pose $\mathbf{g} = (x_g, y_g, \psi_g)^\top$.**

(g) Implement the function `calculateContolCommand` in `src/controller.cpp`. This function has to compute the error in x, y and yaw. The current pose is stored in the `state` member variable, i.e., $\mathbf{x} =$ (`state.x`, `state.y`, `state.yaw`)$^\top$. The desired pose $\mathbf{g}$ is stored in the `goal_x`, `goal_y`, `goal_yaw` member variables. Use the `getCommand` method of the three `PidController` instances `pid_x`, `pid_y` and `pid_yaw` to compute the control commands and store them in the `linear.x`, `linear.y` and `angular.z` part of the `twist` member variable. Note that the errors are in world coordinates, but the control commands have to be in local coordinates.

(h) Start all required ROS nodes as described above, but instead of starting the `ardrone_driver` play back the provided bag file `control_flight.bag`. A green and a blue arrow should appear in RVIZ. The green arrow indicates the direction and magnitude of the control command in x and y direction. The blue arrow shows the yaw command. Check whether these arrows point in the right direction. The default goal position is above the `/marker_95` facing towars `/marker_1`. **Make a screen shot and attach it to your report.**

(i) Run `rxplot` to visualize the current pose estimate and the current command. Replay the bag file again, **make a screen shot of rxplot, and attach it to your report.** Note that you have to enable the controller using the **start** button on the joystick otherwise it does not publish commands on the `/cmd_vel` topic. You can use the following line for accomplishing this for the xy-pose and velocity commands

```
rxplot -p 50 /ardrone/filtered_pose/x:y /cmd_vel/linear/x:y
```

(j) Instead of using the numerical differentiation of the PID controller to determine $\dot{e}_x(t)$ and $\dot{e}_y(t)$ we can use the negative velocities. Adapt the calls to `pid_x.getCommand` and `pid_y.getCommand` in `calculateContolCommand` accordingly. Note the velocities are stored in `state.vx` and `state.vy`. Plot the x and y position and the x and y commands again. **Make another screen shot and attach it as well.** What is the problem with numeric differentiation (**explain in 1-2 sentences**)?

Exercise 2: Autonomous Flight

In the second exercise, you will try out your PID controller from the previous exercise and tune its gains.

(a) Select `/ardrone_controller` in the Reconfigure window to inspect and change the coefficients of your controllers. Play around with different values to understand their effect on the control commands.

(b) Now try your controller on the quadrocopter. Connect to the quadrocopter via wifi and start the `ardrone_driver`. Start with a P-gain of 0.15 for the translational and yaw controllers, and set the I- and D-gains (initially) to zero. Modify the coefficients until you are satisfied with the resulting behavior. Start with changes of 0.05, for fine tuning use 0.01 increments. First tune the P-gain, then the D-gain and optionally the I-gain. **Include the final gain values in your report**.

Submission instructions

A complete submission consists both of a PDF file with the solutions/answers to the questions on the exercise sheet and a TGZ (or ZIP) file containing the source code that you used to solve the given problems. Make sure that your TGZ file contains all files necessary to compile and run your code, but it should not contain any build files or binaries (`make clean`, `rm -rf bin`). Please submit your solution via email to `visnav2013@vision.in.tum.de`.