Ver. 31.05.2022

ITMO University | BE²R Lab

**If you can't do something easily using the instructions in this document and videos of lectures, please let me know. Perhaps there is a mistake or something like that.*

Robot Programming

Kirill Artemov
e-mail: kaartemov@itmo.ru
https://t.me/kartemov

Saint Petersburg, 2022

PREREQUISITES

(nice to know)

1. *Linux* filesystem
   https://www.linux.com/training-tutorials/linux-filesystem-explained
2. *The Bash* shell
   https://en.wikipedia.org/wiki/Bash_(Unix_shell)
3. basic *build*, *test* and *package* software (cmake, catkin)
   http://wiki.ros.org/catkin/conceptual_overview
4. basic *Python*
   https://docs.python.org/2/tutorial/
5. [optional] basic *C++*
6. [optional] basic *Control systems* (PID control)
7. [optional] basic *Robotics*

HOME INSTALLATION

1. Setup your laptop with *Linux* ***Ubuntu*** *18.04+*
   a. one or dual boot (highly recommended)
      Warning: Make sure you know what you are doing and you have a backup plan to restore everything.
   b. using virtual box, vmware etc.
2. Read about ***bash***
3. Read and install ***Robot Operating System*** ***(ROS)***
   a. read: https://www.ros.org/
   b. install: https://wiki.ros.org/noetic/Installation/Ubuntu (desktop version)
4. Read about ***Docker*** https://www.docker.com/
5. Read about ***Gazebo*** http://gazebosim.org/

★ Recommended laptop with
- 8 CPU
- 8GB RAM
- 100 Gb hard disk space
- nvidia/radeon graphics

★ If you install it in a virtual machine, make sure you give it around 6GB of RAM and give about 50GB hard disk space, and disable 3D acceleration

COURSE OVERVIEW

The course includes:

1. Lectures 1-5
2. Practices 1-11
3. Course project = Ex. 1 + Ex. 2 + Ex. 3 + Ex. 4

The evolution of course is sum of

1. Course project **60 points**
    a. Ex. 1 20 p.
    b. Ex. 2 15 p.
    c. Ex. 3 15 p.
    d. Ex. 4 10 p.
2. Exam **40 points**
    a. The format of the exam is
        i. a challenge, where your drones will fly acceptably
        ii. a presentation, where you will present your solution

COURSE PROJECT OVERVIEW

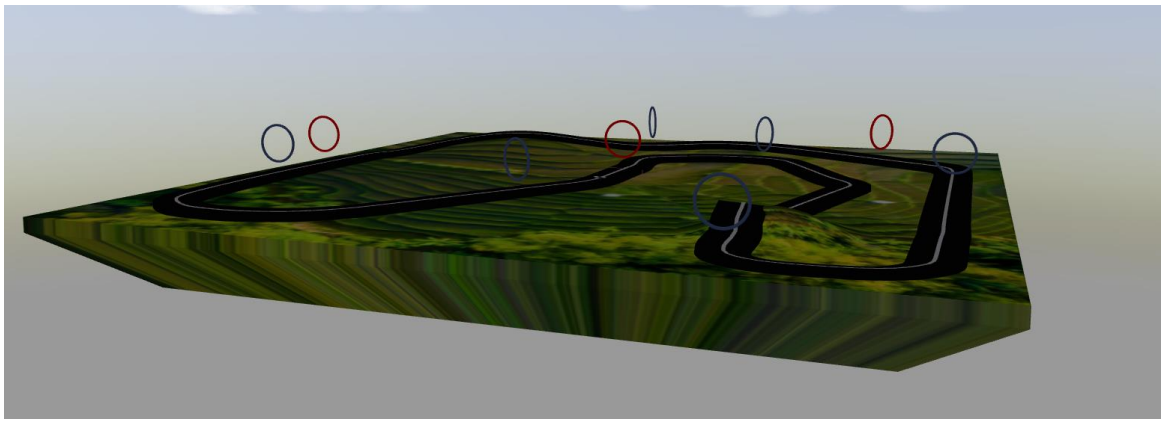

Fig. The virtual world you will create

During the course you will solve the following problem step by step:

"There is a closed line on a surface. The quadcopter must follow the Line. Following the line, the quadcopter may meet Rings on its path. Depending on the color of the ring, flying through it may result in bonus points or penalties."

See the figure above to get a feel!

The solution was decomposed to four parts, which were called ***Exercises.*** Solving each ***exercise*** brings you closer to solving the problem above. Thus, all the exercises are consistent and interconnected.

During the course project you will become familiar with the following technologies and tools:

- ★ Linux Ubuntu
- ★ bash
- ★ git
- ★ docker
- ★ Robot Operating System (ROS)
- ★ gazebo simulator
- ★ Python/C++ (depends what you prefer :))

EXERCISES OVERVIEW

- ★ Exercise 1
  - Gitlab account preparing
  - Software install and setup
  - Drone manual control
- ★ Exercise 2
  - Gazebo world creation
  - ROS packages preparing
  - Drone “takeoff” with Python
- ★ Exercise 3
  - Line detector with OpenCV and Python
  - “Line following” drone controller with Python
- ★ Exercise 4
  - Circle detection with Python
  - Circle drone controller
  - Simple Finite State Machine (FSM) implementation

In the course there are following notation:

- `$ <command>` – a console command executed in terminal with **user** role
- `# <command>` – a console command executed in terminal **inside docker** container
- (text text text) – a comment for a better understanding of what is happening

EXERCISE 1

Here you will do following things:

1. Git and Gitlab setup
2. ROS and Docker setup
3. Drone manual control

Git and GitLab setup

1. Install git on your laptop; run the command:
   `$ sudo apt-get install git`
   enter your password and wait a few moments!
2. Create an account on https://gitlab.com/users/sign_up
3. Inside your gitlab account, create two **empty** and **private** repositories (projects):
   a. Title the first one: **drone_scene**
   b. Title the second one: **drone_solution**
4. Using the hint below, invite teacher to contributors of **each** project as a *Developer.*
   The teacher's gitlab is `kirillin`.
   Hint! https://docs.gitlab.com/ee/user/project/members/#add-users-to-a-project
5. Add a key of Secure Shell (ssh) to remote access your private repositories from your computer:
   a. In your linux open a terminal and run the command:
      `$ ssh-keygen`
      and just press enter many times!
   b. Next, copy the public key using following command:
      `$ cat ~/.ssh/id_rsa.pub`
   c. Copy the command output
      Hint! To copy/paste text from/to terminal use the hotkeys *Ctrl+Shift + C* / *Ctrl+Shift + V*
   d. Open https://gitlab.com and go to *Preferences* (look on the top-right icon-menu) -> *SSH keys* (loot on the left menu)
   e. Paste here your ssh-key and press button *Add key*
6. Now, create two folders with names like in step. 2, for example, you can do it as following:
   a. Open the terminal
   b. Create some folder, using for example following command
      i. `$ mkdir ~/robot_programming_course/` pgi
   c. Change directory and create another two directories for repositories

i. $ cd ~/robot_programming_course/
ii. $ mkdir drone_scene drone_solution

d. Next, inside **each directory** (**repository**) execute (this is a general workflow with git)
   i. $ git init (Doing once to initialize the git repository using working directory)
   ii. $ touch readme.md (To add some changes to the working directory, create new empty file with name readme.md)
   iii. $ git add . (adds a change in the working directory to the staging area)
   iv. $ git commit -m "My first commit" (commits changes)
   v. $ git remote add origin git@gitlab.com:<your_account_name>/<repository_name>.git (links the local repository and the remote one on gitlab)
   vi. $ git push origin master (uploads the working directory)
      1. The name of a branch 'master' or 'main' can be used without difference. And in general, any word can be used here.

e. Now you can check if previously created repositories on the gitlab now contain a *readme.md* file

7. Git and GitLab are now setup!

ROS and Docker setup

1. Clone the repository with Dockerfile and useful scripts
   $ cd robot_programming_course
   $ git clone https://gitlab.com/beerlab/robot_programming_course/docker.git
2. Install *docker* by calling the script
   $ bash ~/robot_programming_course/docker/install_docker.bash
   Hint! if you have nvidia graphics on your laptop, run with key *-n*
3. Build docker image by calling the script
   $ bash ~/robot_programming_course/docker/build_docker.sh
   It took a little bit of time and about 5 Gb internet traffic.
4. Test it
   $ bash ~/robot_programming_course/docker/run_docker.sh
   you should see bash-session inside docker-container.
5. Test if rosmaster is works:
   # roscore
   is it works?
6. Press Ctrl+C, write command exit and press enter to close the container

Drone manual control

1. Create catkin workspace
   `$ mkdir -p ~/robot_programming_course/catkin_ws/src`
   and change directory
   `$ cd ~/robot_programming_course/catkin_ws/src`
2. Clone the repository with hector_quadrotor
   `$ git clone https://gitlab.com/beerlab/robot_programming_course/hector_quadrotor.git`
3. Now, you need to build all packages in the catkin workspace. To do this, you should use a docker container!
   a. Open new terminal and run docker container
      `$ bash ~/robot_programming_course/docker/run_docker.sh`
   b. Change directory in docker bash-session and build catkin workspace
      `# cd /catkin_ws`
      `# catkin build`
      Now wait several minutes, and after the building will completed initialize new ros workspace
      `# source /catkin_ws/devel/setup.bash`
   c. Next, start a simulator with a quadrotor robot
      `# roslaunch hector_quadrotor_gazebo quadrotor_empty_world.launch`
      The simulator window with the drone should open.
   d. Now, you can try to manually control the quadrotor.
      i. Firstly you should enable motors by calling ros-service. To do this, you need another one bash-session. To open it use the command `$ bash ~/robot_programming_course/docker/exec_docker.sh` on your host, and then do
         `# rosservice call /enable_motors "enable: true"`
      ii. Secondly you should set a velocity command throughout the copic */cmd_vel*.
         Hint! To easily enter commands to the terminal, you should use the double pressing tab for autofill commands. For example, if you start to type the command `ros` you can press the Tab key two times, and bash-shell will offer you all possible commands starting from `ros`. It is the same for any other commands.
         Now enter the following command and press two times Tab key
         `# rostopic pub /cmd_vel`
      iii. Next, modify the linear components of Twist and see what happens in the simulator.
4. That's all

a. Write to the teacher that you are ready to present results. You need demonstrate your
    i. gitlab account with two repositories
    ii. start the drone on your personal laptop using docker-container
b. Next, you can move on to the next exercise!

EXERCISE 2

Here you will do following things:

1. Creates ros-packages for the course problem
2. Creates world in *gazebo*
3. Creates simple drone control Python script

Download additional materials

Run the following commands:

1. cd ~/robot_programming_course
2. git clone https://gitlab.com/beerlab/robot_programming_course/materials.git

Creates ros-packages for the course problem

After completing the previous exercise, you have the following file and directory structure

```
~/robot_programming_course
├── catkin_ws
│   └── src
│       ├── drone_scene
│       ├── drone_solution
│       └── hector_quadrotor
├── docker
└── materials
```

The solution of the course problem will include only two ros-packages. Let's do it!

1. To create any ros-package you need to use the console command *catkin_create_pkg*. Let's *create drone_scene* ros-package.
   $ cd ~/robot_programming_course/catkin_ws/src
   $ catkin_create_pkg drone_scene rospy roscpp
   this will create the following file structure

   ```
   ./drone_scene/
   ├── CMakeLists.txt
   ├── include
   │   └── drone_scene
   ├── package.xml
   └── src
   ```

2. Do the same for the drone solution package
3. Using git, upload changes to your gitlab! (see Ex. 1 (general workflow with git))

Creates world in gazebo

A world is the overall simulation scene containing models. A model is something that is simulated in the world. Gazebo uses the SDF format to describe worlds and models. SDF files allow you to describe a robot's links, joints, actuators and sensors. Both SDF and URDF files use XML, so they can interoperate with each other. The <gazebo> tag allows you to associate snippets of SDF with the links and joints in your model. When gazebo starts, it loads a world file. The world file can contain models and you can also spawn additional models at runtime.

Let's make the world with the model of Surface, Line and Circles.

1. Explore the files in *~/robot_programming_course/materials/exercises/ex2* directory. You can see here files for creating world in ros-package *drone_scene*
    a. launch/start_scene.launch
    b. models/*
    c. worlds/line.world
        i. **Note:** you need repair the models pathes inside the file. You need to remove "profi…".
2. Explore these files! Try to figure out what files are needed for what. Then, copy these files to your ros-package drone_scene.
3. Run the world! You can do it using *roslaunch.*
    a. Run the docker
    ```
    $ bash ~/robot_programming_course/docker/run_docker.sh
    ```
    b. Build ros-package *drone_scene*
    ```
    # cd /catkin_ws
    # source devel/setup.bash
    # catkin build drone_scene
    ```
    c. Start the world!
    ```
    # roslaunch drone_scene start_scene.launch
    ```
4. Now you will see *gazebo* simulation with models Line, circles, and drone!

Creates simple drone control Python script

Now, let's write a simple Python script to take off and start moving the drone forward.

1. Copy the file ~/robot_programming_course/materials/exercises/ex2/simple_move.py to drone_solution/src directory and open it to edit. You can use any text editor.
   For example, to open a text file, you can use a default ubuntu text editor *gedit.* To use it, you can use following console syntax:
   ```
   $ gedit ~/robot_programming_course/catkin_ws/src/drone_solution/src/simple_move.py
   ```
2. Next, follow the instruction in comments in the simple_move.py file and write a little bit code

3. Add execution permissions for python script
   ```
   $ chmod +x ~/robot_programming_course/catkin_ws/src/drone_solution/src/simple_move.py
   ```
4. After you completed, you can start it all together
   a. Start the docker
      ```
      $ bash ~/robot_programming_course/docker/run_docker.bash
      ```
   b. Build workspace
      ```
      # cd /catkin_ws
      # source devel/setup.bash
      # catkin build drone_scene
      ```
   c. Add workspace to ROS environment
      ```
      # source /catkin_ws/devel/setup.bash
      ```
   d. Start the scene
      ```
      # roslaunch drone_scene start_scene.launch
      ```
   e. Start the solution
      ```
      # rosrun drone_solution simple_move.py
      ```
   f. Enjoy!
5. Upload your results to gitlab!
6. You can modify the script to control drone more smartly. Share your results in a Telegram group with other students!
7. That's all
   a. Write to the teacher that you are ready to present results. You need demonstrate your
      i. gitlab account with two updated repositories
      ii. run all the materials developed in this exercise and show the teacher
   b. Now, you can move on to the next exercise!

EXERCISE 3

Here you will do following things:

1. Implement PD controller for altitude stabilization
2. Implement computer vision algorithm to detect *Line*
3. Implement controller to *Line* following

Some information

Copy from materials directory new template script for this exercise line_follower.py to your solution rospackage drone_solution/src/

a. There are a set of methods of the class *LineFollower*.
   i. __init__(self)
      Hint: here you can initialize *publishers*, *subscribers*, *services*, and "global" *variables* (variables of an instance of the LineFollower class). All variables must start on *"self."*
   ii. odom_callback(self)
      Hint: here you can receive and process the pose of a robot. You can save the pose to a variable initialized inside __init__() and use it anywhere inside LineFollower class.
   iii. camera_callback(self)
      Hint: here you can receive the image of any camera fixed on a robot. A callback must be implemented for each Subscriber.
   iv. enable_motors(self)
      Why do you think this one?
   v. spin(self)
      Hint: here you can implement all controllers to control a robot.
   vi. shutdown(self)
      Why do you think this one?

Altitude control

1. Implement the controller inside the method spin()
   b. Feedback data you can get inside odom_callback()
   c. PD controller for altitude stabilization can be written as:

$$u_z = K\big(z_{des} - z(t)\big) - B\dot{z}(t)$$

   where K and B are constant proportional gains, z_{des} is constant desires altitude, z and $\dot{z}$ are current position and velocity of a robot, and u_z is control (velocity through z-axis).
2. Tune it!

a. Set K = 1 and B = 0
b. Run the drone and look to it behavior
    i. Decrease K to make take off slower
    ii. Increase K to make take off faster
c. When you are satisfied with the takeoff speed, increase B step-by-spet while the transient of take off will be smooth without overshoot

3. Enjoy the result by changing the desired takeoff altitude
4. How can the takeoff algorithm be improved? Do it on your own!

Computer vision stuff

How to find a line and extract information from it for the controller?

The idea is as follows: based on the assumption that the tilt of the drone during the flight is insignificant, we will use a camera that looks down. Moreover, only the upper half of the image, which sees what is in front of the drone.

So the workflow with camera is as following:

1. Subscribe image from topic /cam_i/camera/image
2. Convert it to OpenCV framework convention
3. Convert the image to grayscale, because it will speed up the processing of the image, since it is not necessary to process all the colors
4. Next, binarize the image to divide it on black and white fields

The result you can see on seria of images below.

Now let's answer the question of how to extract some quantitative features from an image? Let’s try to do it in image space!

Let’s decompose robot control into two parts: (I) steering control and (II) offset control.

(I) steering control

1. Divide the image into 4 parts (by vertical and by horizontal)
2. Find a point at the center of the image
3. Find a point on the Line at the top part of the image
4. Now you can use this information to compute steering. As one of the options, it can be a simple points difference.

(II) offset control

1. Divide the image into 4 parts (by vertical and by horizontal)

2. Find a point at the center of the image
3. Find a point of average of the mask of the image
4. Compute the offset between this point and middle point of the image

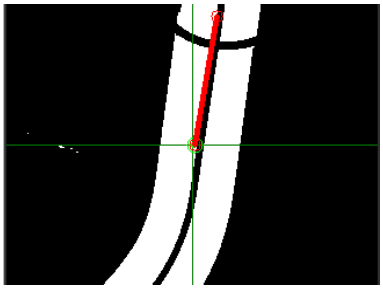

Fig. The graphical interpretation of the idea

From this part you have two control signals:

1. For desired velocity through y-axis
2. For desired angular velocity for a robot orientation $omega_z$

Now, look for this implementation at the line_follower.py script and try to understand all symbols.

Simplest Line following controller

Implement controller inside line_folloing.py script in spin() method. You can find small hints there.

1. The steering control have the following expression

$$u_{\omega_z}(k) = K_s e_{\omega_z}(k) - B_s\big(e_{\omega_z}(k) - e_{\omega_z}(k-1)\big)/\Delta t$$

2. The y-offset control have the following expression

$$u_y(k) = K_o e_y(k) - B_o\big(e_y(k) - e_y(k-1)\big)/\Delta t$$

That's all

1. Write to the teacher that you are ready to present results. You need demonstrate your
    a. gitlab account with developed line_follower.py script
    b. start the drone on your personal laptop using docker-container
2. Next, you can move on to the next exercise when it will be available!

EXERCISE 4

Here you will do following things:

1. Implement computer vision algorithm to detect colored Rings
2. Implement controller to flying through blue Rings
3. Implement controller to avoid red Rings

Before you start

Implement second Subscriber with its own callback for the second camera of a robot. The topic name is /cam_2/camera/image. Next, to show both images for each camera, do following:

1. Add following global variables under the class initialization and after all libraries imports

```
RING_AVOIDANCE_TIME = 5 # [seconds]
DEFAULT_ALTITUDE = 3    # [meters]
```

2. Change `z_des` variable in altitude controller to `self.z_des` and initialized it with `DEFAULT_ALTITUDE` in __init__()
3. Create two class variables in (__init__()) for each image. For example, self.image_1, self.image_2
4. Next, assign each variable the image in callbacks for each camera topic. For example, self.image_1 = cv_image and so on.
5. Now, inside spin() in main loop paste this code:

```
# Display augmented images from both cameras
if len(self.image_1) > 0 and len(self.image_2) > 0:
   self.show_image(self.image_1, title='Line')
   self.show_image(self.image_2, title='Rings')
```

Colored Rings Detector

Create new method and fill it with following content:

```
def ring_detector(self, image, lower, upper, color):
    color_mask = cv2.inRange(image, lower, upper)
    color_contours, _ = cv2.findContours(color_mask,cv2.RETR_TREE,cv2.CHAIN_APPROX_SIMPLE)
    if color_contours:
        max_len_c = 0
        c = color_contours[0]
        for i in range(0, len(color_contours)):
            if len(color_contours[i]) > max_len_c:
                c = color_contours[i]
                max_len_c = len(color_contours[i])
        self.color_distance = max_len_c
        M = cv2.moments(c)
```

```
            if M['m00'] != 0:
                cx = int(M['m10']/M['m00'])
                cy = int(M['m01']/M['m00'])
            else:
                cx = 0
                cy = 0
            (x1,y1), color_r = cv2.minEnclosingCircle(c)
            if color_r > 10:
                image = cv2.circle(image, (cx, cy), radius=5, color=color, thickness=-1)
                cv2.drawContours(color_r, c, -1, (0,255,0), 1)
                color_r = cv2.circle(color_r, (int(x1), int(y1)), radius=int(color_r), color=color,
thickness=4)
                return image, (x1,y1), color_r[0]
        return image, (0,0), 0
```

Next, your second camera callback should looks like following code:

```
    def camera_rings_callback(self, msg):
        # """ Computer vision stuff for Rings"""
        try:
            cv_image = self.cv_bridge.imgmsg_to_cv2(msg, "bgr8")
        except CvBridgeError as e:
            rospy.logerr("CvBridge Error: {0}".format(e))

        # red
        lower = np.uint8([0, 0, 90])
        upper = np.uint8([30, 30, 120])
        cv_image, red_pose, red_radius  = self.ring_detector(cv_image, lower, upper, (0,0,255))

        # blue
        lower = np.uint8([40, 20, 20])
        upper = np.uint8([80, 50, 50])
        cv_image, blue_pose, blue_radius = self.ring_detector(cv_image, lower, upper, (255,0,0))

        # print(red_radius, blue_radius)

        if 50 < red_radius < 70 or 50 < blue_radius < 80:
            if red_radius > blue_radius:
                self.blue_ring_detected = False
                self.red_ring_detected = True
            else:
                self.red_ring_detected = False
                self.blue_ring_detected = True

                # offset in ring xy-plane to fly through center of a ring
                # error = <center of image> - <center of ring>
                self.e_x_blue = 160 - blue_pose[0]
                self.e_y_blue = 120 - blue_pose[1]
        else:
            self.blue_ring_detected = False
            self.red_ring_detected = False

        # save results
        self.image_2 = cv_image
```

A Finite State Machine (FSM) to switch control sceneries

There are four (4) scenarios for our course problem.

1. **State:** Free flight with Line following
2. **State:** Blue Ring alignment to fly through the Ring
3. Red Ring avoiding
    a. **State:** Change the flight altitude to higher than the ring
    b. **State:** Change the flight altitude to any previous

To implement a Finite State Machine, you need:

1. Add variables to class constructor (__init__())

```
self.state = "free_flight"
self.red_ring_detected = False
self.blue_ring_detected = False
self.time_start_up = 0
self.avoidance_time = 0
self.e_x_blue, self.e_y_blue = 0, 0
```

2. Add class method with FSM updater

```
def fsm_update(self):
    if self.red_ring_detected:
        self.state = "drone_up"
    elif RING_AVOIDANCE_TIME <  self.avoidance_time < RING_AVOIDANCE_TIME + 4:
        self.state = "drone_down"
        self.time_start_up = 0
    elif self.blue_ring_detected:
        self.state = "drone_blue_ring"
    else:
        self.state = "free_flight"
```

3. Add handlers for switching scenarios to spin()

```
self.fsm_update()
if self.state == "drone_up":
    self.z_des = 5
    if self.time_start_up == 0:
        self.time_start_up = rospy.get_time()
elif self.state == "drone_down":
    self.z_des = DEFAULT_ALTITUDE
elif self.state == "drone_blue_ring":
    self.z_des += 0.001 * self.e_y_blue
    pass
elif self.state == "free_flight":
    pass
else:
    rospy.logerr("Error: state name error!")

self.avoidance_time = rospy.get_time() - self.time_start_up

print(self.state, self.z_des)
```

It's Debug time

Now, you should debug the resulting code yourself. Initialize all necessary variables and tune the controllers and coefficients. Good luck!

CHALLENGE (optional)

Here you will do following things:

1. Prepare your packages for automatic *Referee*
2. Ask teacher for *an Access code* to use Referee web-interface
3. Using *the Access code* check your solution

Preparing packages

.

Web-interface usage guide

.

Ranking table

.