ROVIO PROGRAMMING IN C#

1. INTRODUCTION

This document presents a set of practical tutorials for programming Rovio robots in C#, based on materials originally used for teaching Robotics at University of Lincoln, UK. The development environment, set of tasks and assumed programming skills might therefore not suit everyone. The tutorial is directed at readers who have basic programming skills and no previous experience with robot programming. Instead of providing step-by-step instructions, the tutorial requires solving different problems (denoted by ➢) and discovering the fundamental concepts of robot programming in a practical way. The vision-based robot control part of the tutorial (Tasks 5-7) assumes some basic understanding of Image Processing techniques.

1.1. ROVIO SET-UP

Rovio is a mobile platform equipped with a colour camera and other sensors, accompanied by a charging dock which also acts as a home base. It is a good practice to keep the robot on charge, parked in the dock, while developing the code. Before you start, read the user manual and familiarise yourself with the robot.

1.2. WEB INTERFACE

Rovio can be accessed through the wireless network and the simplest way of controlling the robot is through the Internet browser. To access the robot, type the IP address of your Rovio in the address bar and alternatively provide user/password credentials. The web interface should appear in the browser window and you should see the live camera image and various controls. Familiarise yourself with the interface. Drive the robot around but try not to hit other objects. The web interface is going to be useful as a visual debugger and additional controller when developing and running your own programs. However, bear in mind that a running web interface might affect the way robot reacts to your code.

1.3. CGI COMMANDS

To be able to program Rovio, you should familiarise yourself with the API document. This document lists all commands that can be sent to and received from the robot. The protocol is based on Common Gateway Interface (CGI) requests. For example, to move the robot one step forward type the following string in the browser: `http://your_rovio_ip/rev.cgi?Cmd=nav&action=18&drive=1&speed=5`

The robot should perform the movement and send back a response to the request that should be visible in the browser window. As you can see, a single request contains many parameters specific for different commands. In our example, the parameters are as follows:

- `Cmd=nav`: choose Movement Control command type;
- `Action=18`: perform Manual Drive command;
- `Drive=1`: drive forward;
- `Speed=5`: set speed to 5 (1 fastest – 10 slowest).

You will get more familiar with these commands over time while working with the robot.

- Change the value of the `Drive` parameter to move the robot backward, left and right with different values of the `Speed` parameter.

2. PROGRAMMING

Our main goal is to program the robot in C#. Open the source code accompanying this tutorial, unpack the file and open the solution in Visual Studio. The solution consists of two projects: a console-based one and a form-based one that will be useful later for vision-based robot control. Let us start with the console project first; you might need to set this project as a default one by right clicking on the 'Rovio Console App' project in the Solution Explorer and selecting 'Set as StartUp Project'.

Let us have a look at the main function. First, a robot object is initialised with the respective IP address and credentials. The next few lines check if we can actually connect to the robot. The program quits if there was a problem with the connection or continues if everything went fine, until the ESC key is pressed.

2.1. SENDING COMMANDS

The basic way of programming Rovio is to send raw CGI requests by using the `Robot.Request(string request)` method. For a quick example, uncomment the part of the code that sends the drive forward command in the CGI format, compile the program and see the results. Modify your code and implement the commands you tried in the previous task.

The included wrapper library implements the most important commands from the API document as a set of classes and methods. For consistency, the library follows the structure of the API document (not necessarily the most intuitive one). There are also convenience classes that allow for an easy access to robot resources (Camera, IRSensor, Odometry, etc.). Uncomment the second part of the code to issue the same drive command as previously but this time using the wrapper library. Try other wrapper methods and parameters.

2.2. COMMAND SEQUENCING

- Familiarise yourself with the `Robot` class; then issue a sequence of commands that will drive the robot in the following way:
    - drive 10 steps forward;
    - point the camera head up;
    - wait 5 seconds;
    - rotate 10 steps right;
    - lower the camera head;
    - drive 10 steps forward.

You might need to introduce delays between some of the commands to ensure that they are executed smoothly. The pause can be realised by issuing the `System.Threading.Thread.Sleep(int milliseconds)` method.

2.3. CONTINUOUS OPERATION

In many robotics applications, it is desirable that a robot operates in a continuous fashion rather than by following a sequence of commands. Thus the robot can execute different actions

depending on certain conditions such as key strokes, sensor values, time intervals, etc. This should be done in the main loop of the program.

- Add the code that will drive the robot one step forward, backward and sideways in response to key presses.
- Add the code that will continuously display the value of the IR sensor. Manually drive the robot around (for example using the web interface or your own keyboard handling routine) and observe the value of the IR sensor (`robot.IRSensor.Detection`). Prior to this, you have to switch the sensor on before the main loop by using the following line: `robot.IRSensor.PowerOn = true;`
- Write a program that will continuously issue the drive forward command until an obstacle is detected in front of the robot.

2.4. THE EXPLORER

- Implement the following behaviour: Rovio drives forward until reaching an obstacle, and then backs a little, rotates by some angle and continues to drive forward until reaching the next obstacle and so on. Experiment with different angle values.

3. ROBOT MOVEMENT

Further development will involve implementing additional methods for the `Robot` class. The most convenient way to achieve that is by creating your own robot class outside the library code. In your project, create a new class file called `MyRobot.cs` and in there create a new class `MyRobot` that inherits from the `Robot` class. Remember to specify the right constructor!

Change the main program such it uses your freshly created `MyRobot` class instead of the `Robot` class from the wrapper library. From now on, you can call all methods specified in the library and also the new ones that you will develop throughout this tutorial.

3.1. DRIVING LONGER DISTANCES

Rovio's drive commands perform a single step movement only. Therefore to move longer distances a series of commands needs to be sent.

- Develop two methods `DriveStraight(int velocity, int repetitions)` and `Spin(int velocity, int repetitions)` that will drive the robot in a straight line or spin it in place with the specified number of command repetitions. The velocity parameter should be from the range of -10 and 10 where a value of 10 indicates movement forward with the highest speed and -10 movement backwards with the highest speed. Value 0 should be interpreted as a stop command.

3.2. BACTERIA-LIKE BEHAVIOUR

- Implement some random behaviour by repeating the `DriveStraight` command followed by the `Spin` command with a random number of repetitions every time. Use the `System.Random` class to generate an integer value from a specified range. Vary the range of the generated random values and observe the changes in the robot's behaviour.

3.3. DRIVE A SPECIFIED DISTANCE/ANGLE

- Calculate the number of repetitions required by the `DriveStraight` command to drive the robot one meter (assume a fixed speed value). Using this information, implement a method `DriveDistance(double distance)` that will accept distance in meters as a parameter. Repeat a similar procedure for spinning by an angle value specified in degrees.
- Program the robot so it drives in a 1x1 m square and ends up at the same position and orientation as at the start. Did the robot do what you expected? How big is the difference between the final and expected position? Try the same task on a different surface and compare the results.
- Check how the speed parameter affects the distance travelled. Implement a method `DriveDistance(double distance, int velocity)` that will drive the robot a specified distance at a given velocity value. To achieve this task, you will need to tune the right number of repetitions for each speed value. Repeat the same task for spinning.

4. ODOMETRY

4.1. ENCODERS

Rovio's wheel encoders provide information about how much each of the wheels has rotated. Each wheel encoder keeps counting until the next time it is read and then its value is reset to 0. It is important to read the encoders frequently; otherwise the read values might be cleared automatically by the robot's microcontroller.

- Write a program that will continuously read the encoder values, accumulate the results for each wheel and display the total value on the screen. Drive the robot one meter and write down the total number of encoder ticks. Repeat the same task for rotation by 90 degrees. Rovio's encoder values are provided by the `Odometry` convenience class.

4.2. ODOMETRY

It is possible to derive a geometric model (kinematics) that relates wheel speeds to the relative speed of the robot. We will use a simplified model for two cases:

- *Robot driving in a straight line:* we can assume that both front wheels rotate with the same speed but opposite direction. Therefore we can take an average of both accumulated encoder readings as the distance travelled (keeping in mind the correct direction of each wheel).
- *Robot rotating in place:* we can assume that all three wheels rotate with the same speed in the same direction. Therefore we can take an average of three encoder values as the angle rotated.

- Implement `DriveStraightOdo(double distance, int velocity)` and `SpinOdo(double angle, int velocity)` methods that will use the simplified odometry model to stop the robot after reaching the desired distance/angle. You may use the information collected in the previous task to relate the number of encoder ticks to the distance/angle. Test these new methods for different speeds and note any differences in the performance. Which of these two methods produces a greater error?

- Now repeat the task of driving the robot in a 1x1 m square. Compare the odometry based method to the approach based on repetitions. Which method is more accurate?

4.3. GOTO POSITION

- Implement a method `DriveToXY(int velocity, double x, double y)` that will drive the robot to a desired position, assuming the robot always starts from point (x=0, y=0) pointing along the Y axis. Decompose the problem into two parts: first correct the heading (orientation), then correct the position (distance).

5. IMAGE PROCESSING

The following tasks will introduce the basics of vision-based robot control. You will learn how to collect and process images from Rovio's camera, detect coloured objects and use this information to control the robot.

We are going to use the AForge.NET library for image processing. The library and on-line documentation can be found at http://www.aforgenet.com/framework/. The provided tutorial files contain a subset of pre-compiled library files only (placed in a solution folder called 'AForge.NET...') but you are encouraged to download and install the full release of the framework for your own projects. The full release contains the source code, code examples and documentation which might be useful if you want to learn more about the AForge framework.

5.1. THE BITMAP CLASS

We are going to use the .NET `Bitmap` class as a basic image structure. First, familiarise yourself with the class: http://msdn.microsoft.com/en-us/library/system.drawing.bitmap.aspx.

The Rovio wrapper library comes with the `Camera` class that implements the functionality related to the camera sensor.

- Grab a single image from Rovio (`Rovio.Camera.Image`) and save it on disk using the `Bitmap.Save()` method.

To save the image, you can choose from a variety of image formats by specifying the appropriate file extension. In our scenario, it is better to use the lossless format like PNG, for example, which does not introduce compression artefacts. Locate the saved image on the disk and inspect its quality. You can also load the saved image by specifying its name to one of the `Bitmap` class constructors.

There are several camera parameters that can be set by the wrapper library (methods in the `Rovio.Camera` class) including image size, quality, brightness, etc. Change these parameters and see the effects on the acquired image.

5.2. IMAGE PROCESSING

Some of the AForge functions accept greyscale images only. To perform image conversion to greyscale implement the following code:

```
Bitmap grayscale_image = AForge.Imaging.Filters.Grayscale.
CommonAlgorithms.Y.Apply(image);
```

- Grab a single image, convert it to greyscale, save it on disk and inspect the results.

AForge implements many of the popular image processing filters. To use the filters you have to create an instance of the specific filter and then use the `Filter.Apply(Bitmap image)` method to perform the filtering. The following example demonstrates how to use the `Threshold` filter:

```
AForge.Imaging.Filters.Threshold filter =
new AForge.Imaging.Filters.Threshold(threshold_value);

Bitmap result = filter.Apply(grayscale_image);
```

- Implement the `Threshold(Bitmap image, int value)` method (a member of `MyRobot`) using AForge filters. Then continuously capture images from the robot (in the main loop) and process them using the newly created `Threshold` method. Measure, how long it takes to collect and process a single image in ms. You can use the `Stopwatch` class to measure time intervals between different steps of your algorithm. To get more realistic results, you can accumulate the results from several steps and calculate averages. Check your results for different image resolution.
- Implement a method that will perform binary erosion (`AForge.Imaging.Filters.Erosion`) on the thresholded image and compare the images before and after filtering.

5.3. Object Detection

- Implement a method `Rectangle DetectObject(Bitmap image)` that will detect a coloured object (e.g. a ball) and return its rectangular boundaries on the provided image.

To implement this method, use the `AForge.Imaging.Filters.ColorFiltering` filter to segment out pixels corresponding to the object. Before you call the `Apply` method, you should specify the colour range for each channel. For example to set the min and max values for red channel use the following code: `filter.Red = new IntRange(100, 255)`. Convert the resulting image to greyscale and use the `AForge.Imaging.BlobCounter` class to detect all connected components in the image (i.e. blobs). You can specify different parameters for this class including sorting order, maximum object count, etc. Remember to set them before issuing the `Apply` method. You can retrieve a list of rectangles that specify the location and size of all detected blobs using the `BlobCounter.GetObjectsRectangles()` function.

To test you method, print the location and size of the largest object on screen or adopt the following code to draw a corresponding rectangle on the image:

```
Graphics g = Graphics.FromImage(image);

g.DrawRectangle(new Pen(Color.FromArgb(255, 0, 0)), your_rectangle);
```

6. Visualising the Results

It might be quite tiresome to manually inspect the results of your image processing algorithm. To enable real-time visualisation of the processed images we need to switch to the window form-based project. To do so, right click on the 'Rovio Forms App' project in the Solution Explorer and select 'Set as StartUp Project'.

You might note several differences to the console based project that were necessary in order to adapt our program to the form-based version. The main thread of the program is responsible

for handling the windows forms and therefore it is a good idea to run all robot related functionality in a separate thread (`robot_thread`). The robot thread simply executes a custom made method `MyRobot.ProcessImages()` where you should develop all your robot related code.

The communication between threads is realised through events. You should see an example event defined in the `MyRobot` class called `SourceImage`. By calling this event, `MyRobot` broadcasts the specified image to other objects in the program. The remaining step is to attach an appropriate object/method that will be called when this event is broadcasted. You should see an example of that in the MainForm.cs file:

```
robot.SourceImage += source_view.UpdateImage;
```

In this example, the `SourceImage` event is attached to the `source_view` form which through `UpdateImage` method will update the content of the displayed image.

- Run the provided example and observe the results.

You have to repeat the above procedure if you need to visualise more than a single step of your algorithm. It should be straightforward to tailor the provided examples to your needs.

- Repeat Tasks 5.2 and 5.3 but this time with real-time visualisation enabled. Visualise different steps of the algorithms.

Since there is no console window available, you need to think about a new way of displaying all relevant information from your program. For example you can try `System.Diagnostics.Debug` class, `System.Windows.Forms.MessageBox`, etc.

If you want to carry on with the visual interface you will need to transfer all methods developed so far in your `MyRobot` class to the new `MyRobot` class included in the form-based project.

7. VISION-BASED CONTROL

7.1. BANG-BANG CONTROLLER

- Implement the following controller using the information provided by the `DetectObject()` method:
  - issue `RotateLeft` command if the object is placed on the left side of the image;
  - issue `RotateRight` command if the object is placed on the right side of the image.

You might want to introduce a neutral zone: no action taken if the object is around the image centre. Test the behaviour of the robot for different values of the speed parameter.

7.2. PROPORTIONAL CONTROLLER

- Implement the following controller using the information provided by the `DetectObject()` method:
  - calculate the displacement between the centre of the object and the image centre along the horizontal axis (in pixels).
  - issue the `Spin` method with the speed parameter equal to: `displacement*gain`.

Set the initial gain value to 0.1. Test the behaviour of the robot for different values of the gain parameter.

7.3. OBJECT FOLLOWING BEHAVIOUR

- Implement the following controller using the information provided by the `DetectObject()` method:
  - a proportional controller to keep the object in the image centre along the horizontal axis (see Task 7.2);
  - a proportional controller to keep the object at the desired size. You would need to modify the controller from Task 7.2 and adjust the speed parameter of the `DriveStraight` command according to the size of the object.

To combine these two controllers, use a simple state machine that will alternate between two actions (i.e. `Spin` or `DriveStraight`).

- Add a search procedure and a corresponding extra state in the state machine to the controller when the object is lost.