Assignment 1

Objective

The instructions given here are for you to get a clearer overview of the assignment, but you are free to implement the solution in your own way.

This programming exercise will help you to get acquainted with ROS and several development tools that you will use throughout the course.

The assignment is based on a simulated Kobuki robot from the Yujin robotics company (Links to an external site.)Links to an external site..

The final goal of the programming assignment consists in controlling a simulated two-wheeled differential drive robot Kobuki (Links to an external site.)Links to an external site. and make it follow a wall by using readings from distance sensors mounted on the robot as shown below

This assignment is divided in several sections to help guide you and so that you can get familiarized with the ROS and Linux environment that you will use throughout the course.

Preliminary assignments:

- Installing Ubuntu 16.04 (64 bit) and ROS Kinetic on your computer (either via Virtualbox or via a dual-partition) or use the computers in the computer rooms

- Ubuntu/Linux tutorials for those of you who are not very experienced using

Ubuntu/Linux

- ROS (Kinetic) tutorials

- Description of the robot setup -- the Kobuki robot + distance sensors + DC motors & encoders

For more information on the preliminary steps.

Setting up your Linux environment

Installing Ubuntu 16.04 + ROS Kinetic on your computer

For this course we will use the **Ubuntu 16.04** ***64 bit*** operating system and **ROS Kinetic**.

You have two options of installing the system or you can use the computer in the computer rooms:

- **Virtual machine**: install virtualbox in your current OS and run a virtual ubuntu machine from there. *This will be the easiest option for most cases*.
- **Separate Ubuntu OS partition**: partition your hard drive to install Ubuntu alongside your current operating system. *This is harder to do than the previous option but probably more worthwhile in the long run*. In this case you will also need to install ROS Kinetic by yourself, although we supply easy-to-follow instructions on how to do that.

Optionally you can also download the virtual machine or ubuntu .iso installation file from these links:

- Ubuntu 16.04 64 bit virtual machine (Links to an external site.)Links to an external site.
- Ubuntu 16.04 64 bit.iso installation file (Links to an external site.)Links to an external site.

Setting up a Virtualbox (easiest option)

For this exercise session we will provide you with Virtualbox images with **Ubuntu 16.04** and **ROS Kinetic** already installed and configured and with the necessary ROS packages you need for the assignment.

To set up your system for the session please do the following:

1. Install **Virtualbox** in your computer (follow this link (Links to an external site.)Links to an external site.).
2. (Optional) Install the **Virtualbox** extension pack for USB 2.0 support (You can download it from here (Links to an external site.)Links to an external site.).
3. Use the VB image (you can follow these instructions). You can download the virtual machine from here: Ubuntu 16.04 64 bit virtual machine (Links to an external site.)Links to an external site..

You can login to the system using the password **ras2018**

TROUBLESHOOTING

- **OBS**: make sure the extension pack you download is compatible with your virtualbox version. Check this page (Links to an external site.)Links to an external site.
- If the virtual machine does not work, try to go into your **BIOS settings** and **enable virtualization**.

Using the computers in the computer rooms (also easy)

The Ubuntu computers in the E building are now running **Ubuntu 16.04** and have **ROS Kinetic** already installed on them.

What you have to do in order to start working on the lab is, open up a terminal window (by typing "Ctrl+Alt+T") and run the following commands:

```
wget  https://raw.githubusercontent.com/danielduberg/KTH-RAS/master/lab1_basic_install.sh
chmod  +x  lab1_basic_install.sh
./lab1_basic_install.sh
```

Installing an Ubuntu 16.04 partition in your hard drive

WARNING!!! BACK UP ALL YOUR DATA FROM YOUR HARD DRIVE FIRST IF YOU CHOOSE TO DO THIS!!!

Installing Ubuntu 16.04

To do this you will need a USB stick which you will convert into a bootable disk.

If you are using Windows you might consider shrinking first your windows partition before running the ubuntu installation file (look at these instructions (Links to an external site.)Links to an external site. for examples on how to do it). Mac users should try to find an equivalent procedure on the internet. **Make sure you allocate at least some 15-20 GB for Ubuntu**.

1. Download **Ubuntu 16.04** here (Links to an external site.)Links to an external site..
2. Create a **bootable ubuntu installation USB stick**. You can do this by following the instructions at this page (Links to an external site.)Links to an external site., for Windows, or this page (Links to an external site.)Links to an external site., for Mac.
3. Reboot your computer and **boot using the USB stick**. You will have to choose the USB stick as your boot disk instead of your hard drive.
4. Follow the installation instructions from the GUI, its pretty simple and straightforward.

Installing ROS Kinetic and libraries necessary for this lab assignment

Once you have started Ubuntu and logged into your account, open up a terminal window (by typing "Ctrl+Alt+T") and run the following commands:

```
wget  https://raw.githubusercontent.com/danielduberg/KTH-RAS/master/install.sh
chmod  +x  install.sh
./install.sh
```

The script might prompt you for your **sudo password** several times, make sure you type in the password and hit ''enter''. It can take fair bit of time to download and install all the packages so please be patient.

To verify that ROS has been correctly installed **open a NEW terminal window** and run the following command:

```
source ~/.bashrc
rosversion -d
```

You should get 'kinetic' as the output.

Linux tutorials

If you are not very experienced using Linux/Ubuntu, we recommend you to take a look at some tutorials available on the internet before you start working with ROS. Focus on how to use the linux **terminal**, i.e., how to use basic linux commands such as e.g. changing/creating directory, listing the contents of a directory, copying/moving files and directories, etc.

You can take a look for example at:

Introduction to Linux and Basic Linux Commands for Beginners (Links to an external site.)Links to an external site.

http://www.ee.surrey.ac.uk/Teaching/Unix/ (Links to an external site.)Links to an external site.

http://linuxcommand.org/lc3_learning_the_shell.php (Links to an external site.)Links to an external site.

You can find many more on the internet. If you find a useful one that you would like to share with your classmates please let us know so that we can post it here!

NB: do not spend TOO much time in the tutorials, this is just to get you familiarized a bit with Linux... you will learn the commands with time & practice.

ROS website tutorials

Please go through the beginner level ROS tutorials (Links to an external site.)Links to an external site..

Skip anything that has to do with the **rosbuild** system for ROS versions Groovy and earlier. In the course we will use the ROS **Kinetic** distribution and we will use the **catkin** build system.

The beginner tutorials which are crucial for the exercise are numbers: **2-10, 12-13. Make sure to select catkin in each of the tutorials**.

Extra: it is also good to look at tutorial **1** of the intermediate level ROS tutorials (Links to an external site.)Links to an external site..

Software setup for this lab

The provided virtualbox image already contains a catkin workspace (located at **~/catkin_ws**) with the packages that you will need to run this lab.

The packages/metapackages are:

1. **ras_lab1** this metapackage contains most of the source code for running the simulations. It contains the following packages:
   1. **ras_lab1_distance_sensor**: contains nodes for simulating the distance sensors
   2. **ras_lab1_motors**: contains nodes for simulating the motors that control the motion of the Kobuki robot.
   3. **ras_lab1_world**: sets up the environment, places a virtual wall in the vicinity of the robot.
   4. **ras_lab1_launch**: contains launch files for running the whole simulation environment in one go.

These packages are also available at our Github repository (Links to an external site.)Links to an external site.

The actual lab:

- Coding an open-loop motor controller for the Kobuki

- Coding a closed-loop motor controller using encoder feedback

- Coding a wall-following controller using also measurements from the distance sensors

Deadline and grading

You can talk to each other and help each other as much as you want, but **you must code the assignment individually** and **you will be graded on an individual basis**.

During grading you will run the assignments for us. We will also ask you to show us and explain parts of your code. You do not need to write any report.

You have a very limited amount of time to show us that you succeeded in implementing your solution and that you understand the code, so **do not** try to implement a fancy solution to a simple problem.

The deadline to get the bonus points for this assignment is September 7. However, you can still present it during a help session after that.

ROS website tutorials

Please go through the beginner level ROS tutorials (Links to an external site.)Links to an external site..

Skip anything that has to do with the **rosbuild** system for ROS versions Groovy and earlier. In the course we will use the ROS **Kinetic** distribution and we will use the **catkin** build system.

The beginner tutorials which are crucial for the exercise are numbers: **2-10, 12-13. Make sure to select catkin in each of the tutorials**.

Extra: it is also good to look at tutorial **1** of the intermediate level ROS tutorials (Links to an external site.)Links to an external site..

Software setup for this lab

The provided virtualbox image already contains a catkin workspace (located at **~/catkin_ws**) with the packages that you will need to run this lab.

The packages/metapackages are:

1. **ras_lab1** this metapackage contains most of the source code for running the simulations. It contains the following packages:
    1. **ras_lab1_distance_sensor**: contains nodes for simulating the distance sensors
    2. **ras_lab1_motors**: contains nodes for simulating the motors that control the motion of the Kobuki robot.
    3. **ras_lab1_world**: sets up the environment, places a virtual wall in the vicinity of the robot.
    4. **ras_lab1_launch**: contains launch files for running the whole simulation environment in one go.

These packages are also available at our Github repository (Links to an external site.)Links to an external site.

The Kobuki robot setup

Assignment 1: Robot setup

The Kobuki robot

The Kobuki robot is a 2 wheeled mobile robot. You can find information about the robot's packages that are available in ROS at the Kobuki ROS wiki page (Links to an external site.)Links to an external site. (make sure to select 'KINETIC'!).

Launching Kobuki in ROS

Follow this tutorial (Links to an external site.)Links to an external site. for instructions on how to launch a simulated Kobuki, visualize it using *Rviz* and controlling it using your keyboard. **NB: Follow the tutorial until the teleoperation section, no need to do section 2 (Navigation demo)**.

To visualize the Kobuki in Rviz, run

```
rosrun  rviz  rviz
```

From a terminal after you have launched the robot simulation node and in the rviz window set the fixed frame to **odom** and add a **robot model** as shown below in the figure.

To understand what nodes and topics are being used to run the keyboard teleop, run

```
rosrun  rqt_graph  rqt_graph
```

To see the commands that are being sent from the keyboard teleop node to the robot through the **/mobile_base/commands/velocity** topic execute the following command on a terminal window while you are driving the robot around:

```
rostopic  echo  /mobile_base/commands/velocity
```

If you want to plot the velocity commands that are being sent to the robot while you are driving the robot forward, you can do the following for instance:

```
rosrun  rqt_plot  rqt_plot  /mobile_base/commands/velocity/linear/x
```

As you can see, the keyboard teleop node sends **linear and angular velocity commands** to the Kobuki robot. In this lab assignment you will not control the Kobuki directly through these linear and angular velocity inputs but rather by controlling the power sent to each of the motors on the wheels of the robot. This way you will deal with a more realistic scenario in this assignment that will be more similar to what you will be doing when you build your own robot for the course.

Launching the Kobuki robot setup for this lab

To launch the Kobuki setup that you will use for this lab, with simulated DC motors that control the motion of each of the wheels, distance sensors and a virtual wall, we have provided a simple launch file where you can run everything in one go. Run the following command from a terminal window:

```
roslaunch  ras_lab1_launch  kobuki_lab1.launch
```

An **rviz** window with the following configuration should pop up:

As you can see there is a red virtual wall and some light blue lines which go from the distance sensors mounted on the robot to the wall. The red arrow indicates the odometry of the robot. You can launch the keyboard teleoperation node to drive the robot around.

Inspect the contents of the launch file that you just ran. Run the following in the terminal:

```
rosed ras_lab1_launch kobuki_lab1.launch
```

As you will see, this launch file runs a number of nodes including the kobuki simulation node, the distance sensors, the 'world' node which sets up the virtual wall, the motors node and rviz.

To see all the published/subscribed topics do a rostopic list from a terminal window:

```
rostopic list
```

The most important ones for this assignment are:

1. **/kobuki/adc**: (msg type: *ras_lab1_msgs/ADConverter*) contains the (digitized) measurements from the distance sensors
2. **/kobuki/encoders**: (msg type: *ras_lab1_msgs/Encoders*) contains the encoder values of each motor on each of the Kobuki's two wheels.
3. **/kobuki/pwm**: (msg type: *ras_lab1_msgs/PWM*) publish to this topic to set the pwm value of each motor and control the motion of the robot.

You can e.g. see the values published by the distance sensors by running:

```
rostopic echo /kobuki/adc
```

If you want to check what is the message type for a particular topic you can run for instance:

```
rostopic info /kobuki/adc
```

The Wall

Everytime you launch the launch file the virtual wall will be placed at a different (random) angle. This is to ensure that you properly test your controller in different situations and to make sure that simple open-loop heuristic controllers do not work :)

The distance sensors

Distance sensors are analog electronic sensors that are used to measure the distance between the sensor and whatever is in front of the sensor.

The Kobuki setup that you will use in this assignment will include two sensors that will help you measure the alignment with respect to a simulated wall.

The sensors are separated by a distance of 20 cm.

The simulated distance sensor that you will use for this assignment resembles *somewhat* the short range Sharp IR sensors (ir_gp_2d120_ss_datasheet.pdf).

The sensor has an operational range from **10 to 80 cm**, in which the voltage output of the sensor decreases as the measured distance increases (look at the sensor datasheet for more info, **although the simulated sensor response is not 100% like the real one!!!**). There is some gaussian white noise added to the sensor output as well.

The code for running the simulated distance sensor is located in the **ras_lab1_distance_sensor** package. The sensors will provide a simulated adc value that must be converted into a distance before being used. You can use *d = 1.114e^(-0.004adc)* as the sensor function, where x is the adc value and d the distance in meters.

Distance sensor topic /kobuki/adc

Run

```
rosmsg  show  ras_lab1_msgs/ADConverter
```

to see the message definition for this topic.

When launching the ''kobuki_lab1.launch'' file as described above, the **front distance sensor** will be published under **ch1** of the **/kobuki/adc** topic while the **back distance sensor** will be published under **ch2** of the **/kobuki/adc** topic.

The values you see under **/kobuki/adc** do not correspond to raw voltage values (i.e. decimal numbers) of the sensor but rather quantized **integer values** that can range from 0 to 1023. This is because this topic simulates a microcontroller's *ADC* (analog to digital converter) which converts linearly voltage values on the range 0-5V to 10 bit (0 to 1023) integer values.

The DC motors

In order to control the motion of the robot, you will have to control the DC motors that move each of the wheels of the robot. If you are not familiar with the operation of (differential) DC motors, please refer to the course book/lectures or the internet and read up a bit. (Links to an external site.)Links to an external site.

What you basically need to know is that DC motors are controlled through **PWM** signals which can be modulated to regulate the amount of power fed to the motor. This in turn affects the *angular* velocity at which the wheel rotates.

However, a PWM signal alone is *not enough* to fully control the speed at which the wheel attached to the motor rotates. External torques applied to the wheel lower the output angular velocity. Moreover, motors usually have different responses even if they come from the same manufacturer. This is why DC motors normally include **encoders**, which provide a feedback signal that allow us to determine the angular position and infer how fast the motor is actually rotating.

The simulated DC motors of this assignment get updated at **10 Hz**. If 2 seconds pass without receiving a pwm signal message, the motors shut down and the robot stops.

The /kobuki/pwm topic

Run

```
rosmsg  show  ras_lab1_msgs/PWM
```

from a terminal to see the message definition for this topic.

This topic allows you to control the pwm signal sent to each of the motors of the robot. **/kobuki/pwm/PWM1** corresponds to the **left wheel** while **/kobuki/pwm/PWM2** corresponds to the **right wheel**. The pwm signal is an **integer** that can range between *-255 and 255*. A pwm signal of **255** will drive the wheel forward at max speed while **-255** will drive the wheel at max speed in the opposite direction.

The /kobuki/encoders topic

Run

```
rosmsg  show  ras_lab1_msgs/Encoders
```

to see the message definition for this topic.

The **encoder1** and **delta_encoder1** fields of the message correspond to the **left wheel** while the **encoder2** and **delta_econder2** fields correspond to the **right wheel**.

The **encoder1** and **encoder2** fields are absolute encoder values. However, for control purposes we are usually more interested in *differential* encoder values (**delta_encoder1** and **delta_encoder2**) which indicate how much the encoder has changed since the last control cycle, which is directly related to angular velocity of the wheel.

Encoder parameters

The encoders for this lab have **360 ticks** per revolution. This means that a **change of 1 in the differential encoder signal (delta_encoder)** corresponds to a **change of 1 degree** in the angular position.

Kinematic parameters of the robot

The kinematic parameters for the differential configuration are:

- **Wheel radius (r)**: 0.0352 m
- **Base (b)**: 0.23 m

Tasks

If you find any conflicting statements on these pages compared to earlier instructions, then the earlier instructions are the correct ones.

Task 1: Send a command to the robot

Send a command to the robot

The robot receives a pwm signal on the topic **/kobuki/pwm**

Your task is to send a command on that topic such that the robot moves.

Optional create a ras_lab1_open_loop_control package

Create a package named **ras_lab1_open_loop_control** in the catkin workspace (**~/catkin_ws/src**) folder.

Create an open loop control node

Write a Python node called **open_loop_controller** which publishes full power commands (PWM = 255) to the DC motors through the **/kobuki/pwm** topic for both of the wheels **every 100 milliseconds**.

You will notice that even though you are sending the same power to both motors the robot drifts towards the left because the left motor is actually slower than the right one.

This will also happen with real robots, given that two DC motors will never have exactly the same response. This is why we need some form of feedback of the output angular velocity/position of the motors to regulate the power that we send to the motors so that they rotate at the speed that we desire.

Play around with your open loop node and try to send different PWM values (from -255 to 255) to the motors to see what happens to the trajectory of the robot.

Task 2: Cartesian controller

Theoretical background: DC motors and differential encoders

As we saw in the previous (optional) task it is necessary to have a sensor mounted on the motors which provide some sort of measurement of the output position/velocity of the motor to be able to control it.

In practice, DC motors are fabricated with encoders to accomplish that. There are different kinds of encoders, but the majority of them are basically digital sensors which indicate the change in angle of the motor shaft. (Links to an external site.)Links to an external site.

As described before, the DC motors simulated in this lab have **360 ticks per revolution** (1 revolution = 360 degrees of rotation = 2*pi radians of rotation).

This means that a value of **1** on the **/kobuki/encoders/delta_encoder1** topic indicates that the motor has rotated **1 degree** since the last control cycle. Each control cycle is **100 ms long** (10 Hz control), which means that a value of **1** on the differential encoder topic indicates that the wheel is spinning at (roughly, up to quantization error) **1 degree/100 ms = 10 degrees/s**

Create the ras_lab1_cartesian_controller package

Create a package named **ras_lab1_cartesian_controller** in the catkin workspace (**~/catkin_ws/src**) folder. In this package you will code your cartesian control node.

Write a closed loop motor controller node with encoder feedback

Code a Python node called **cartesian_controller**. This controller takes as input the **encoder feedback** from the motors and produces **pwm** signals for controlling the motor speeds.

Below is an illustration of the high-level view of the controller:

The motor controller takes as input a **twist** (**linear and angular velocity**; v and w respectively) of the robot and controls the power of the motors through the **pwm** signal such that each wheel spins at the **desired angular velocity**. To estimate the **error** between the **desired angular velocity** for each wheel and the **actual angular velocity** the controller must use **feedback from the motor encoders**. Your controller should run at **10 Hz** (every 100 ms).

For this assignment it is enough for you to implement a P-controller (proportional controller) for each of the motors. A PI-controller (proportional and integral) should work much better.

ROS interface

Your **cartesian_controller** node should **subscribe** to the following topics:

- **/motor_controller/twist** *(message type: geometry_msgs/Twist)*: in this topic the controller will receive the linear and angular velocity at which we wish to move the robot (expressed in the base frame of the robot). The message is a **6D twist** (3D for linear velocity and 3D for angular velocity) but since we are controlling the robot in a 2D plane you only need to use one component of the linear velocity (**the x-component**) and one component of the angular velocity (**the z-component**). Using the kinematics equations for differential mobile robot configurations, one can then calculate the individual contributions of each wheel (in terms of angular velocity) to achieve the desired twist. To view the complete twist message definition run in a terminal:

```
rosmsg  show  geometry_msgs/Twist
```

- **/kobuki/encoders** *(message type: ras_lab1_msgs/Encoders)*: through this topic the controller will receive the encoder feedback signals for estimating the angular velocity of the wheels.

Your node should **publish** to the following topic:

- **/kobuki/pwm** *(message type: ras_lab1_msgs/PWM)*: pwm signal for controlling the power fed to the motors.

For more details on the **encoder and pwm** topics refer back to the robot description page.

Calculating the wheel angular velocities

You can find information about the robot kinematics on page 23 in the assignment description:

ROS_presentation.pdf

There you will find the kinematic equations for differential configurations and learn how to calculate the angular velocity of each wheel given a twist (linear and angular velocity) of the robot frame.

A simple controller

The pseudo-code for a simple motor controller looks like this:

```
error = desired_w - estimated_w
int_error = int_error + error*dt
pwm = alpha*error + beta*int_error
```

Where **desired_w - estimated_w** is the *angular velocity error*. **desired_w** is the desired angular velocity at which we want the wheel to rotate (commonly expressed in radians/second), while **estimated_w** is the angular velocity at which the wheel is actually rotating. **alpha** is a positive control gain that you have to tune yourself by running the controller and seeing how the robot moves. dt is the time difference between two consecutive iterations of the controller.

The desired angular velocity for each wheel depends on the input twist (linear + angular velocity) and can be calculated using the kinematic equations described previously.

TIP for tuning the controller gain: start with LOW gains, and slowly increase until you have a decent control performance or before the system becomes unstable. Keep in mind that the gains for each of the motors could be different.

Task 3: Wall follower

The goal of the final wall following controller is to have the robot move in a straight line parallel to the wall. The alignment of the robot with respect to the wall will be corrected using the distance sensors through a cartesian control scheme.

Code a wall-following cartesian controller

Create a package called **wall_following_controller** in the catkin workspace (**~/catkin_ws/src**) folder.

The wall following controller + cartesian controller will follow this structure:

Code a Python node called **wall_following_controller**. This node should publish a **geometry_msgs/Twist** type message for the robot so that it **aligns to the wall by rotating with an angular velocity that depends on the readings of the distance sensors**. The linear velocity will be set constant for simplicity.

The pseudocode for a simple wall following cartesian controller is:

```
linear_vel = < some constant >
angular_vel = alpha*( distance_sensor1 - distance_sensor2)
```

The angular velocity is a P-control that generates angular velocities **proportional to the difference of the distance sensor ADC values**. Remember to place the linear velocity in the **x-component** and the angular velocity in the **z-component** in the twist message.

Notice that for doing the controller we do not really need to convert the distance measurements to meters, if designed properly the controller can just operate on raw ADC signal values.

This control scheme **decouples** the control in two parts: one cartesian control that controls the alignment of the robot with respect to the wall and the motor control which does the lower level control of the wheels' angular velocities.

There are many ways to do a wall following control, the decoupled scheme presented here is one that is easy enough to understand and code. We have not considered e.g. a minimum distance that the robot has to keep to the wall, we simply align as we drive forward from the initial position. You are welcome to try any other control scheme if you want, the recipe presented here is enough to get a passing grade.

Testing the controller

For testing the wall-following controller, you can first publish to the simulation nodes (**/mobile_base/commands/velocity** topic) at a **10 Hertz rate**.

Once you are happy with the results, stop publishing to the **/mobile_base/commands/velocity** topic and publish the twist message to the **/motor_controller/twist** topic of your **cartesian controller** that you coded in **task 2**. You can compare the performance of your cartesian controller vs. the simulation by observing the output trajectory.

Common pitfalls

- **Use radians not degrees** when doing the kinematic calculations!