

ELE719

Laboratory Manual

Fall 2016 Edition

Y.C. Chen

Department of Electrical and Computer Engineering
Ryerson University

September 2016

ELE719
Fundamentals of Robotics
Laboratory Manual

Fall 2016 Edition

Y.C. Chen
Department of Electrical and Computer Engineering
Ryerson University

Copyright ©2016 by Y.C. Chen

All rights reserved. No part of this publication may be reproduced or distributed in any form or by any means, or stored in a data base or retrieval system, without the prior written permission of the author.

Contents

Introduction	1
1 Introductory Python Programming	4
1.1 Introduction	4
1.2 Preparatory Work	5
1.3 Getting Started	5
1.4 Variables, Data Types and Operations	5
1.5 Program Control Structures	8
1.6 Functions and Modules	9
1.7 Objects, Properties and Methods	11
1.8 Input and Output	14
1.9 Laboratory Demonstration	15
1.10 What to Submit	15
2 Inverse Kinematics and Simple Motions	16
2.1 Preparatory Work	17
2.2 Laboratory Session	21
2.3 Laboratory Demonstration	27
2.4 What to Submit	28
3 Forward Kinematics and Kinematic Control	29
3.1 The Motion Control Problem	29
3.2 Preparatory Work	31
3.3 Laboratory Demonstration	35
3.4 What to Submit	36
3.5 Reference	36
4 Obstacle Avoidance Control	37
4.1 Obstacle Avoidance Controller Design	37
4.1.1 Step 1: Collision Avoidance	39

4.1.2	Exponential Stabilization	42
4.2	Preparatory Work	42
4.3	Laboratory Work	45
4.4	What to Submit	45
4.5	Reference	46
A	Report Requirements	47

List of Figures

2.1	3WD robot	17
2.2	ssh problem.	22
2.3	Connect to Server Window	23
2.4	The Robot Home Window.	24
3.1	Kinematic control	30
3.2	Straight-line motion	33
4.1	3WD Robot and reference frames.	38
4.2	IR sensors locations.	40
4.3	A typical sensor frame.	41
4.4	Sensor experiment.	44
4.5	d versus V_{raw}	45

List of Tables

1.1	Python reserved words	7
4.1	Sensor frame locations (with respect to robot frame \mathcal{F}^R)	43
4.2	Distance versus raw sensor values	44

Acknowledgments

The development of the three-wheel-drive mobile robot used in the experiments has benefited from the work of several individuals. The architecture of the robot was inspired by the work of Rowland O’Flaherty of the GRITS Lab at the Georgia Institute of Technology. The real-time ADC capture Python module for the Beaglebone embedded controller board was developed by Mike Kroutikov. The robots were built with the help of the technical support staff of Department of Electrical and Computer Engineering.

Introduction

Course Contents

This course presents a comprehensive coverage of the fundamental concepts in autonomous mobile robotics as well as robotic manipulators. The motion control problem for autonomous mobile robots is also discussed.

Students are required to design and implement motion controllers for a three-wheel drive mobile robot in the laboratory. The Python computer language will be used to implement these controllers.

Laboratory Facilities

The experiments will be conducted in the Control Systems Laboratory in Room ENG413. There are 12 work stations available in ENG413. Each work station is equipped with a desktop computer running the LINUX operating system. The desktop computers in the laboratory are connected to the departmental computer network, so all the software available on the network disks such as MATLAB and MAPLE are available.

The main objective of the laboratory experiments is to study the motion control problem of mobile robots. Students are required to test their controllers on three-wheel drive mobile robots available in Room ENG413. Each robot is equipped with:

1. Three omni-directional wheels with each wheel driven independently by an armature-controlled DC motor. A quadrature encoder is attached to each wheel to provide angular displacement feedback.
2. A Beaglebone Green embedded controller board.

3. An inertia measurement unit (IMU) capable of providing the absolute orientation of the robot.
4. Six Sharp infrared (IR) analog sensors for distance measurements in the range of 4 to 30cm.
5. H-bridge motor drivers, and rechargeable battery pack for power supply.
6. A WiFi dongle to provide communication between the work station and the robot.

The main CPU of the Beaglebone Green controller board is a 32-bit TI AM335x 1 GHz Cortex-A8 chip. This controller board is quite suitable for mobile robot applications because:

- The TI AM335x Cortex-A8 CPU has built-in hardware support, called eQEP (enhanced quadrature encoder pulse), for up to three quadrature encoders.
- It has two 32-bit Programmable Real-Time Units (PRU) for time critical real-time processing.
- It has eight 12-bit ADC (analog-to-digital converter) that can be used to interface with analog devices, such as the IR sensors here.
- The CPU has hardware support to generate PMW signals for motor control.
- It runs on an embedded version of LINUX.

Rules and Regulations

The following rules and regulations must be observed in the laboratory:

1. Students are required to work in groups. The maximum number of students in a group is 2.
2. Preparatory work and reports are group responsibility. In other words, only one submission is required from the group.
3. Preparatory work must be handed in at the beginning of the first session of each new experiment. Students will not be allowed to begin an experiment if preparatory work is not submitted. Due dates for the preparatory work will be announced in class.
4. All experiments must be demonstrated in order to receive full credit and must be completed within the allotted time. **No extension will be granted.**

5. Unless otherwise specified, laboratory reports (if required) are due one week after the last session of an experiment. **No late reports will be accepted.**
6. Attendance of laboratory sessions are compulsory. You will not receive full credit for experiments in which you did not fully participate. **Students are expected to be punctual.**
7. Students must return all equipment to their proper locations before leaving the laboratory.
8. Students must not use any software unrelated to the experiments on the laboratory computers.
9. No food or drinks are allowed in the laboratory.

Comments and Suggestions

Your comments and suggestions on this laboratory manual are welcomed. This will help us provide future students a better manual. Please fill out the “Comments and Suggestions” page at the end of the manual and return it (anonymously) to the laboratory instructor by the end of the course.

Experiment 1

Introductory Python Programming

Time to Complete: To be announced in class

Marks: 10 (prelab 0, lab 10)

Purpose

To provide an introduction to the Python programming language and to familiarize with the software development environment of the Control Systems Laboratory at ENG413.

Objective

By the end of this experiment, you should be able to write simple object-oriented programs using Python.

1.1 Introduction

Python is an open source, cross-platform, high-level, dynamically typed, object-oriented, interpreted programming language. It is simple yet elegant, compact yet powerful, and is easy to learn and use. In some ways, it is similar to MATLAB but is available free of charge. Python is supported by a vast amount of *modules* (similar to MATLAB's toolboxes) and that makes it even more versatile and powerful. A comprehensive coverage of Python is beyond the scope of this course. Instead, only a small subset of Python essential to this course will be introduced here.

1.2 Preparatory Work

Go through the rest of this experiment, try out the provided examples and answer the Exercises.

1.3 Getting Started

To begin, you will need to logon to the EE Department's computer network. If you do not already have an account, get one at ENG439. The 3WD robot is not required in this experiment.

The LINUX operating system is used in all the work stations of the Control Systems Laboratory. It is strongly recommended that you consult the "EE Network User's Guide" (available at: <http://www.ee.ryerson.ca/guides/user>) if you are not familiar with the LINUX environment.

Now, go to your home directory (i.e. folder) and create a work directory for ELE719 and make it your current directory as follows:

```
% cd
% mkdir ele719
% cd ele719
```

The "%" symbol above is assumed to be the command prompt (your's may be different) and does not need to be entered by you. After that, start up Python by typing "python" at the command prompt, and you will see an output similar to the following on the screen:

```
% python
Python 2.6.6 (r266:84292, Jun 18 2012, 09:57:52)
[GCC 4.4.6 20110731 (Red Hat 4.4.6-3)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The ">>>" symbol above is the python interpreter's prompt and is an indication that it is ready for user's input.

1.4 Variables, Data Types and Operations

It was mentioned earlier that Python is dynamically typed. That means variables do not have a pre-defined data type. Instead, they are typed according to the type of data assigned to them. As an example, enter the following at the Python prompt:

```
>>> a = 1
>>> b = 1.0
```

Both of these statements assign a value of 1 to the variables `a` and `b`. However, `a` is typed to be an integer since it is assigned the integer number 1, and `b` is typed as a floating point number because the floating number 1.0 is assigned to it.

Exercise 1.1 Suppose that a new variable `c` is defined as `c = a + b`, where `a` and `b` are defined as above. What data type do you think `c` will have?

To avoid the kind of unnecessary problems shown in the following example, it is strongly advised that all numeric constants be assigned as floating point numbers in your program.

```
>>> a = 1
>>> b = 2
>>> half = a/b
>>> print half
0
>>> one_third = 1/3
>>> print one_third
0
```

Exercise 1.2 Correct the above example so that it produces the right answers.

As another example, enter the following at the Python prompt:

```
>>> d = 'efg'
>>> e = 'ijk'
```

Exercise 1.3 Suppose that a new variable is define as `g = d + e`. What do you think `f` is? How about `g = a + d` and `h = b + d`?

Variable names in Python are case sensitive and cannot begin with a number. They must begin with a letter of the alphabet but the rest may contain letters, numbers and underscores. However, the following Python reserved words cannot be used as names of variables:

Table 1.1: Python reserved words

and	assert	break	class	continue	def
del	elif	else	except	exec	finally
for	from	global	if	import	in
is	lambda	not	or	pass	print
raise	return	try	while	True	False

The previous examples involved the use of three basic Python data types: integer, float and string. Another useful Python data type is the Boolean type. Variables of this type have value `True` or `False` only. For example,

```
>>> a = True
>>> b = not a
>>> c = (a and b) or (not a)
```

Python supports many other useful basic data types such as long, complex, list, set, tuple, etc., but we will not be needing them in this course.

So far, we have only considered scalar variables. Matrices can be used in Python with the help of the `numpy` module. For example, let A , B and s be defined as

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}, \quad B = \begin{bmatrix} 7 \\ 8 \\ 9 \end{bmatrix}, \quad s = 3.0.$$

Then we can get $C = AB$ and $D = sA$ as follows:

```
>>> from numpy import array, dot
>>> A = array([[1, 2, 3], [4, 5, 6]])
>>> B = array([7, 8, 9]).T
>>> s = 3
>>> C = dot(A, B)
>>> D = s*A
```

The first line in the above example is used to load the `array` and `dot` methods from the `numpy` module into the workspace. The second line defines the matrix A , row by row, starting with the first row. The third line defines the column vector B by transposing the row vector $[7, 8, 9]$. The `dot` method is then used to do the *matrix* multiplication of A and B to get C . Note that the “`*`” operator cannot be used here because it is reserved by Python to represent *element-by-element* multiplication. This is why it can be used to get $D = sA$ because s is

scalar.

Exercise 1.4 Suppose that you entered $C=A*B$ by mistake, what do you think the result will be? How about $C=A+B$ and $C=A-B$?

1.5 Program Control Structures

Similar to other high-level programming languages, Python supports the usual **if-then-else**, **for** loop and **while** loop program control structures. These control structures allow a block of code to be executed if certain condition is **True**. Unlike most programming languages (such as C or C++) which use braces to enclose a block of code, Python use *indentation* instead. Specifically, code within the same block must have the same level of indentation.

Consider the following example which determines all the prime numbers in the range 2 to 20:

```
1. for k in range(2, 21):
2.     for x in range(2, k):
3.         if (k%x == 0):
4.             print k, 'is ', x, '*', k/x, 'so it is not a prime number'
5.             break
6.     else:
7.         print k, 'is a prime number'
```

The line numbers in the above code are for reference only and are not part of the code. This example has two **for** loops and one **if** statement. The code from lines 2 to 7 is the code block of the **for** statement at line 1. The built-in function **range(a,b)** produces the list of numbers $[a, a+1, a+2, \dots, b-2, b-1]$. Hence this **for** loop will be executed for the values of **k** from 2 to 20 (not 21). The “:” character at the end of the **for** statement is used to indicate the beginning of a code block. The code from lines 4 to 5 is the code block of the **if** statement at line 3. They will be executed when the condition $k\%x == 0$ becomes **True** (i.e. when **k** is evenly divided by **x**). The code from lines 2 to 7 make up a **for-else** control structure that works as follows: The **for** part of the code will be executed for the value of **x** from 2 to **k-1**. The **else** part of the code will be executed when the **for** part has exhausted iterating its range (2 to **k-1** in this case).

In general, the **if-then-else** control structure has the following form:

```
if (condition_1):  
    <code block 1>  
elif (condition_2):  
    <code block 2>  
elif (condition_3):  
    ...  
    ...  
else:  
    <code block N>
```

The **elif** and **else** parts are optional.

The **for-else** control structure has the following form:

```
for k in range(k1, k2):  
    <code block 1>  
else:  
    <code block 2>
```

The code in **<code block 1>** will be executed for the value of **k** from **k1** to **k2-1**, and the **else** part is optional.

Finally, the **while-else** loop control structure has the following form:

```
while (condition):  
    <code block 1>  
else:  
    <code block 2>
```

The code in **<code block 1>** will be executed until **condition** becomes **False**, at which point the **while** loop is exited and **<code block 2>** will be executed. Again, the **else** part is optional.

Exercise 1.5 Rewrite the example program for finding prime numbers using **while-else** control structures instead of **for-else** control structures.

1.6 Functions and Modules

A function is defined in Python using the **def** reserved word followed by the name of the function and a list of arguments to be passed to the function:

```
def name_of_function(arg1, arg2, etc):  
    <code block for the function>  
    return <results>
```

The “:” character at the end of the `def` statement signifies the beginning of the code block for the function which must be indented below the `def` statement. The `return` statement is not needed if the function does not return results back to the calling program. Consider the following function:

```
def add(x, y):  
    z = x + y  
    return z
```

Exercise 1.6 What are the results of (a) `add(1, 2)`, (b) `add('hello ', 'there.')`, (c) `add([1, 2, 3], [4, 5, 6, 7])`, and (d) `add(1, 'two')`?

A collections of functions can be put together in a single file as a *module*. To do so, create a file, say `mymodule.py`, and put the code for all the functions in it. Suppose that the file `mymodule.py` contains the following:

```
def func1(arg1, arg2):  
    <code for func1>  
  
def func2(arg1)  
    <code for func2>  
    return result  
  
def func3()  
    <code for func3>  
    return result
```

Then to use these function, do the following:

```
from mymodule import *  
  
# To call func1  
mymodule.func1(a1, a2)
```

```
# To call func2
r2 = mymodule.func2(a3)
```

```
# to call func3
r3 = mymodule.func3()
```

The `import *` statement at the beginning of the code instructs Python to load everything in the `mymodule` module into the workspace. The statement `mymodule.func1` is then used to call the function `func1` defined in the file `mymodule.py`. Statements that begin with “#” are comments. If `func1` is a unique function name, then we can do the following:

```
from mymodule import func1
```

```
# To call func1
func1(a1, a2)
```

As mentioned at the beginning, a vast amount of Python modules are available. We have already used the `numpy` module which is part of the SciPy Python-based ecosystem of open-source software for mathematics, science and engineering. Basic mathematical functions such as the trigonometric functions are defined in the `math` module. For example,

```
from math import sin, cos, sqrt

a = sin(theta)
b = sqrt(c)
```

The `dir` command can be used to list the contents of a particular module once it has been imported into the workspace. For example:

```
>>> import(math)
>>> dir(math)
```

Detailed documentation of a function can be obtained using the `help` command. For example:

```
>>> import(numpy)
>>> help(numpy.array)
```

1.7 Objects, Properties and Methods

Python has been designed as an object-oriented language from the outset. In fact, consider the following assignment statement:

```
>>> a = 1.0
```

This assignment statement creates an object named “a” which is an instance of the object type `float`. Similarly, the following code:

```
>>> from numpy import array
>>> b = array([1.0, 2.0, 3.0])
```

creates an object called “b” which is an instance of an object type `array` defined in the module `numpy`.

The `type()` function can be used to determine the type of an object. For example:

```
>>> a = 1.0
>>> type(a)
```

returns

```
<type 'float'>
```

Python allows the creation of new object types called *classes*. These new classes can then be used to create new objects in the same way as the built-in object types.

Enter the following example which defines a new class called `Rectangle` into a file and name it `rect.py`:

```
class Rectangle:
    def __init__(self, length, width, color):
        self._length = length
        self._width = width
        self._color = color

    def area(self):
        return (self._length)*(self._width)

    def perimeter(self):
        return 2.0*(self._length + self._width)

    def color(self):
        return self._color

    def paint(self, color):
        self._color = color
```

```

@property
def properties(self):
    return [self._length, self._width, self._color]

```

As shown in the above example, the definition of a new object type begins with the reserved word `class` followed by the name of the class. The “:” character after the class name signifies the beginning of the code block for the new class. Inside the code block, functions that can be used with the new object type are defined. These functions are called *class methods* since they are applicable only to their own class object. Similar to ordinary functions, a `def` statement is used to signify the beginning of a class method. It is followed by the name of the method and then a list of parameters enclosed in a pair of round brackets. All class methods must have at least one parameter named `self`, and it can be used by the class object to refer back to itself.

The first class method is a special method known as the *constructor* of the class. This method has a special name `__init__` and it is called whenever an object of this class is created. In the above example, the `__init__` method has 3 additional parameters: `length`, `width` and `color` that allow the user to define the size and color of the `Rectangle` object to be created. Other class methods can be defined the same way as discussed earlier for ordinary functions, with the exception that the parameter list must begin with the parameter `self`. However, the `self` parameter is only required when class methods are *defined*. They are *not* required when class methods are *called*.

The following code shows how the class `Rectangle` can be utilized:

```

>>> from rect import Rectangle    # Import the class definition into workspace
>>> a = Rectangle(2, 3, 'red')    # Create a 2 by 3 Rectangle with color red
>>> a.area()                      # Display the area of 'a'
6
>>> a.perimeter()                 # Display the perimeter of 'a'
10.0
>>> a.color()                     # Display the color of 'a'
'red'
>>> a.properties                  # Display the properties (length, width, color) of 'a'
[2, 3, 'red']
>>> a.paint('blue')               # Change the color of 'a'
>>> a.properties
[2, 3, 'blue']

```

Note that the `@property` statement before `def properties(self)` allows us to omit the brackets `()` when calling the `properties` method.

Exercise 1.7 Extend the `Rectangle` class above by adding 2 class methods, `change_length` and `change_width`, to change the length and width of the `Rectangle` object respectively. Test the new methods by first changing the length to 5 and then the width to 4. Verify the results with the `area` and `perimeter` methods after each change.

1.8 Input and Output

The `raw_input()` function can be used to prompt for a user's input. For example,

```
>>> a = raw_input('Please enter the value of a: ')
```

The result returned by `raw_input()` is a *string*, but it can be converted to either an integer or floating number as follows:

```
>>> a = int(a)          # convert to an integer
```

```
>>> a = float(a)        # convert to a floating number
```

Outputs can be printed using the `print()` function. For example,

```
>>> a = 1
```

```
>>> b = 2.0
```

```
>>> print 'The value of a is ', a, ' and the value of b is ', b
```

The output can also be formatted using C-style format specifiers:

```
>>> print 'The value of a is %3d and the value of b is %5.2f\n' % (a, b)
```

It is also possible to save results into a file for later processing. For example,

```
>>> f = open('output.csv', 'w')
```

```
>>> f.write('%3d %5.2f\n' % (a,b))
```

```
>>> f.close()
```

A file object named `f` is first created using the Python built-in method `open()`. This method takes 2 parameters with the first one being the name of the file. The second one specifies what to do with the opened file: `r` for reading, `w` for writing data to a new file (i.e. if a file with the same name already exists then it will be overwritten), or `a` for appending data to an existing file. The `write` method is then used to write data to the file. Again, the output can be either formatted or unformatted. The file should be closed using the `close` method once it is no longer needed.

1.9 Laboratory Demonstration

Show the results of Exercises 1.5 and 1.7 to the laboratory instructor.

1.10 What to Submit

A report is not required for this experiment.

Experiment 2

Inverse Kinematics and Simple Motions

Time to Complete: To be announced in class

Marks: 30 (prelab 25, lab 5)

Purpose

In this experiment, the three-wheel drive (3WD) mobile robot to be used through out this course is introduced. The ELE719 Python module `Robot3WD` for controlling the motion of this robot is also introduced.

Objectives

By the end of this experiment, you will be able to:

1. Obtain the inverse kinematics equation for this robot.
2. Extend the `Robot` object class in the `Robot3WD` module with additional class methods that implement the inverse kinematics equation and simple motions for the robot.
3. Program the robot to perform simple motions using the extended object class.

2.1 Preparatory Work

Consider the kinematics model of the 3WD robot in Figure 2.1, where the **fixed** base frame is $\mathcal{F}^0 = o_0x_0y_0$, and the robot frame is $\mathcal{F}^R = o_Rx_Ry_R$. The robot frame \mathcal{F}^R is always fixed on the robot, but its position and orientation will change w.r.t. the base frame \mathcal{F}^0 as the robot moves around. The position of the robot with respect to \mathcal{F}^0 is given by (x, y) and its orientation by the angle θ . The vector $p = [x, y, \theta]^T$ represents the current *posture* of the robot. The *linear* velocities of the right, left and back wheels are denoted by the vectors v_1 , v_2 and v_3 respectively. These vectors point to the direction of the *linear* motion produced by *counter-clockwise* rotation of the respective wheel. Let the radius of the wheel be r , the angular displacement of wheel i be ϕ_i , then the angular velocities of the wheels are $\dot{\phi}_i = v_i/r$, $i = 1, 2, 3$.

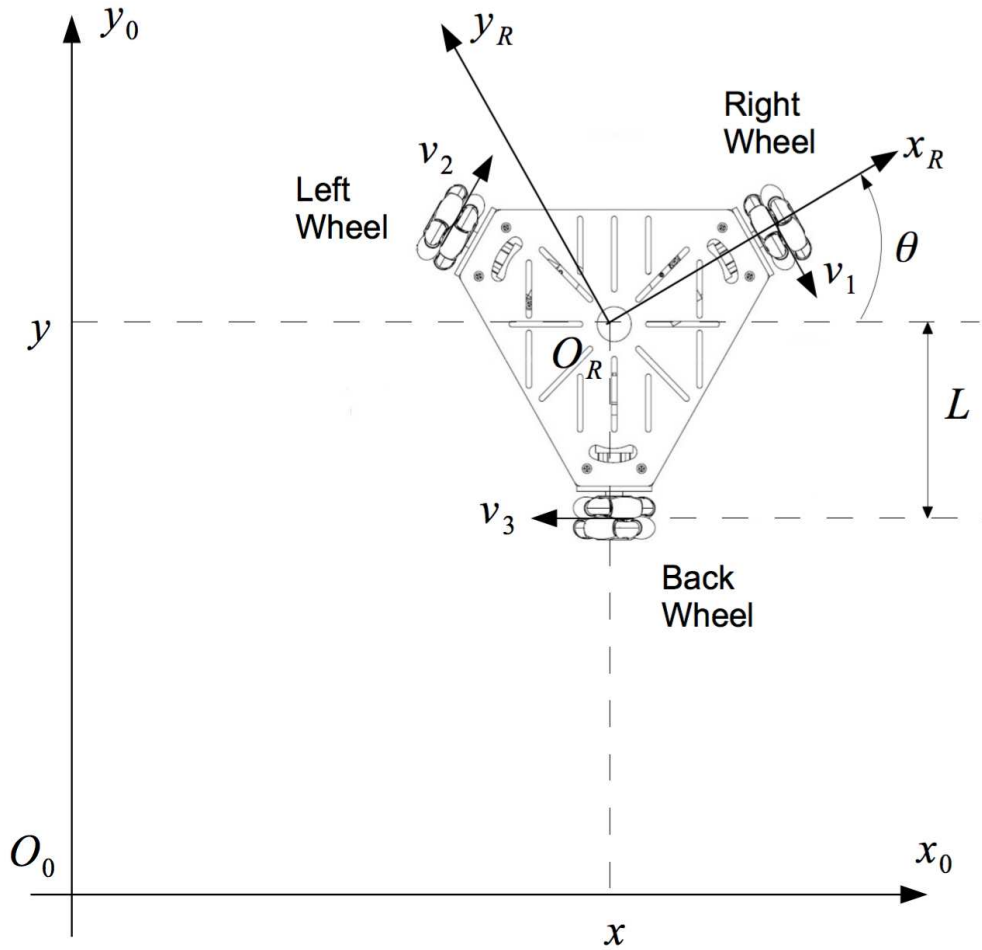


Figure 2.1: 3WD robot

Using the above notations,

1. Show that the inverse kinematics equation for the 3WD robot is given by:

$$\begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} r\dot{\phi}_1 \\ r\dot{\phi}_2 \\ r\dot{\phi}_3 \end{bmatrix} = \begin{bmatrix} \sin(\theta) & -\cos(\theta) & -L \\ \cos(\frac{\pi}{6} + \theta) & \sin(\frac{\pi}{6} + \theta) & -L \\ -\cos(\frac{\pi}{6} - \theta) & \sin(\frac{\pi}{6} - \theta) & -L \end{bmatrix} \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} \quad (2.1)$$

or,

$$\dot{\Phi} = M(\theta)^{-1}\dot{p} \quad (2.2)$$

where $\dot{\Phi} := [\dot{\phi}_1, \dot{\phi}_2, \dot{\phi}_3]^T$, $\dot{p} := [\dot{x}, \dot{y}, \dot{\theta}]^T$, and

$$M(\theta)^{-1} := \frac{1}{r} \begin{bmatrix} \sin(\theta) & -\cos(\theta) & -L \\ \cos(\frac{\pi}{6} + \theta) & \sin(\frac{\pi}{6} + \theta) & -L \\ -\cos(\frac{\pi}{6} - \theta) & \sin(\frac{\pi}{6} - \theta) & -L \end{bmatrix}.$$

2. Extend the `Robot` class in the `Robot3WD` module to provide an additional class method `inverse_kinematics()` that implements the inverse kinematics equation (2.1). At this point, you do not need to know the details of the `Robot3WD` module. All you need to do here is to follow the instructions below to implement Eq. (2.1) with Python code.

Start by opening the file `myRobot.py` (available from the course web page). You will find the following Python code at the top of the file:

```
from math import pi, sin, cos, sqrt, exp
from numpy import array, dot

from Robot3WD import Robot

class myRobot(Robot):
    def __init__(self, sampling_period):
        Robot.__init__(self, sampling_period)
```

The above code simply defines a new object class called `myRobot` which *inherits* the properties and methods of the object class `Robot` defined in the `Robot3WD` modules. In other words, the new object class `myRobot` will have access to all the properties and methods of the `Robot` class. To extend the `Robot` class, simply add the code for the `inverse_kinematics()` method to implement Eq. (2.1) into the file `myRobot.py`. A template for this method has been included in `myRobot.py`:

```
def inverse_kinematics(self, p_dot, theta):
    L = self._L
    wheel_radius = self._wheel_radius
    #... (Fill in rest of the code here) ...
    return wheel_angular_velocities
```

3. Further extend the `Robot` class by providing additional methods to perform the following simple motions (defined with respect to the base frame \mathcal{F}^0): (a) move forward with linear velocity v , (b) move backward with linear velocity v , (c) move left with linear velocity v , (d) move right with linear velocity v , (e) rotate counter-clockwise (CCW) with angular velocity ω , and (f) rotate clockwise (CW) with angular velocity ω .

As an example, consider the motion: move left with linear velocity v . Since the motions are defined with respect to \mathcal{F}^0 , moving left with velocity v simply means moving in the negative x direction of \mathcal{F}^0 at velocity v , or in the positive x direction with velocity $-v$ (see Figure 2.1). Thus, the required velocity vector of the robot is $\dot{p} = [-v, 0, 0]^T$ and the required angular velocities of the wheels to execute this motion can be obtained using the inverse kinematics equation (2.1). Therefore, this motion can be easily implemented using the following method:

```
def move_left(self, v, theta):
    p_dot = [-v, 0.0, 0.0]
    w = self.inverse_kinematics(p_dot, theta)
    self.set_angular_velocities(w)
```

The above code first defines the velocity vector \dot{p} , uses the `inverse_kinematics()` method defined earlier to determine the required angular velocities of the wheels ω , and then uses the `set_angular_velocities()` method from the `Robot3WD` module to move the robot.

Using the above code as a guide, implement the remaining simple motions with the following methods in `myRobot.py`:

```
def move_forward(self, v, theta):
    #... (Fill in rest of the code here) ...

def move_backward(self, v, theta):
    #... (Fill in rest of the code here) ...

def move_right(self, v, theta):
```

```

        #... (Fill in rest of the code here) ...

    def rotate_CCW(self, w, theta):
        #... (Fill in rest of the code here) ...

    def rotate_CW(self, w, theta):
        #... (Fill in rest of the code here) ...

```

4. Using the sample program `lab2.py` in Section 2.2 as a guide, develop a Python program, `lab2demo.py`, for the robot to execute the following motions *in the given order*:
- (a) Move forward with velocity of 0.2 m/sec. for 2 sec., followed by
 - (b) Move right with velocity of 0.2 m/sec. for 2 sec., followed by
 - (c) Move backward with velocity of 0.2 m/sec. for 2 sec., followed by
 - (d) Move left with velocity of 0.2 m/sec. for 2 sec., followed by
 - (e) Rotate counter-clockwise with velocity of 8 rad/sec. for 2 sec., followed by
 - (f) Rotate clockwise with velocity of 8 rad/sec for 2 sec..

Your program should be very similar to `lab2.py`, with the exception that the following statements in `lab2.py`:

```

from Robot3WD import Robot
...
myrobot = Robot(sampling_period)

```

replaced by

```

from myRobot import myRobot
...
myrobot = myRobot(sampling_period)

```

A template for this program, `lab2demo.py`, can be found on the course web page. You may wish to test the method for each motion separately before using all of them together.

2.2 Laboratory Session

The robot to be used in the laboratory experiments for this course is a three-wheel drive robot with 3 omni-directional wheels. Each wheel of the robot is actuated by its own DC motor with an encoder providing feedback on the angular displacement of the wheel. This robot also has 6 infrared sensors capable of measuring distances in the range of 4 to 30 cm. These sensors will be used in later experiments for simple obstacle avoidance path planning. An inertia measurement unit (IMU) is also available to provide information on the orientation of the robot.

A Python module, Robot3WD, has been developed for this course to provide transparent access to the hardware of the robot. Using this and other supporting Python modules, the robot can be controlled using very simple and straight forward code. The entire Robot3WD Python module contains hundreds lines of Python, C and assembly language code. However, from a user's point of view, only several special functions (called class methods) in the library are relevant. In particular, the hardware of the 3WD robot can be accessed through the Python object class called `Robot` and its associated methods to be described in a moment.

Connecting to the 3WD Robot

In order to gain access to the hardware of the 3WD robot, the Beaglebone Green controller board has to be booted up by following the steps below:

1. Turn on the power of the battery located at the bottom level of the robot.
2. Turn the main power switch located on the top plate of the robot to the “ON” position.

A series of blue LED lights on the controller board will begin to flash intermittently. Wait until a flashing green light is observed on the WiFi dongle. This should take no longer than a minute.

Next, follow the steps below to connect to the 3WD robot:

1. Login to the EE Department's computer network through the workstation in the laboratory.
2. Once logged in, open up a new terminal window by clicking on the “Terminal” icon.
3. Identify the robot ID labeled on top of the 3WD robot and go to the new terminal window to login to the 3WD robot from the work station as follows:

```
% ssh user@robot15
```

The name `robot15` is used as an example only, you will need to replace it with the actual ID of your robot.

4. You will then be prompted for the login password for the robot. Enter the password provided by the laboratory instructor and you should see something similar to the following:

```
Debian GNU/Linux 7
```

```
BeagleBoard.org Debian Image 2015-11-03
```

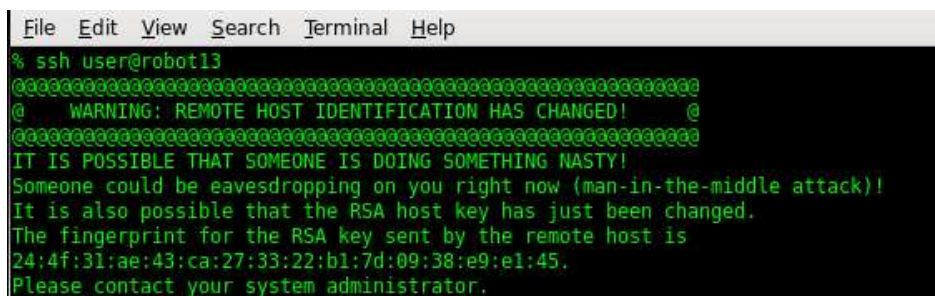
```
Control Systems Laboratory
```

```
Department of Electrical and Computer Engineering, Ryerson University.
```

You can now access the robot through this terminal window.

If your `ssh` attempt to the robot failed due to the authentication problem shown in Figure 2.2, then enter the following command before trying again:

```
% rm ~/.ssh/known_hosts
```



```
File Edit View Search Terminal Help
% ssh user@robot13
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@    WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!    @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle attack)!
It is also possible that the RSA host key has just been changed.
The fingerprint for the RSA key sent by the remote host is
24:4f:31:ae:43:ca:27:33:22:b1:7d:09:38:e9:e1:45.
Please contact your system administrator.
```

Figure 2.2: ssh problem.

Important Notes: Any files created on the robot will be removed after the robot is powered down or rebooted. It is important that you follow the work flow below to ensure your work is not lost.

Recommended Work Flow

It is recommended that you prepare all your program files ahead of the laboratory session and save them in **ele719** folder created using your EE network account in the previous experiment. Files in this folder can then be transferred to the robot as follows:

1. Move the mouse to the menu bar at top of the Desktop and click on “**Places**” and then “**Home Folder**”. A new window would pop up displaying all your files and folder in your home directory on the EE network drive.
2. Locate the “**ele719**” folder created in the previous experiment and double-click on it.
3. Another new window would pop up displaying your files in the “**ele719**” folder. This is the folder you should use to store your files for this course. Hereafter, this window will be referred to as the “**ELE719 Home Window**”. Keep this window open through out the entire laboratory session.
4. Go back to the menu at the top of the Desktop and click on “**Places**” and then “**Connect to Server**”. A window similar to the one in Figure 2.3 would appear. Go



Figure 2.3: Connect to Server Window

to this window and do the following:

- (a) Select “**SSH**” for “**Service type**”.
- (b) Enter the ID of the robot (**robot15** is used as an example here) for “**Server**”.

- (c) Enter `"/home/user"` for "Folder".
- (d) Enter `"user"` for "User Name".
- (e) Click on "Connect".
- (f) A new window will appear and you will be prompted for the password. Enter the login password provided by the laboratory instructor.

A new window showing the home directory of the robot, `/home/user`, would appear. (See Figure 2.4). Hereafter, this window will be referred to as the "Robot Home Window". Any new files created on the robot in `/home/user` will automatically appear in this window. Keep this window open through out the entire laboratory.

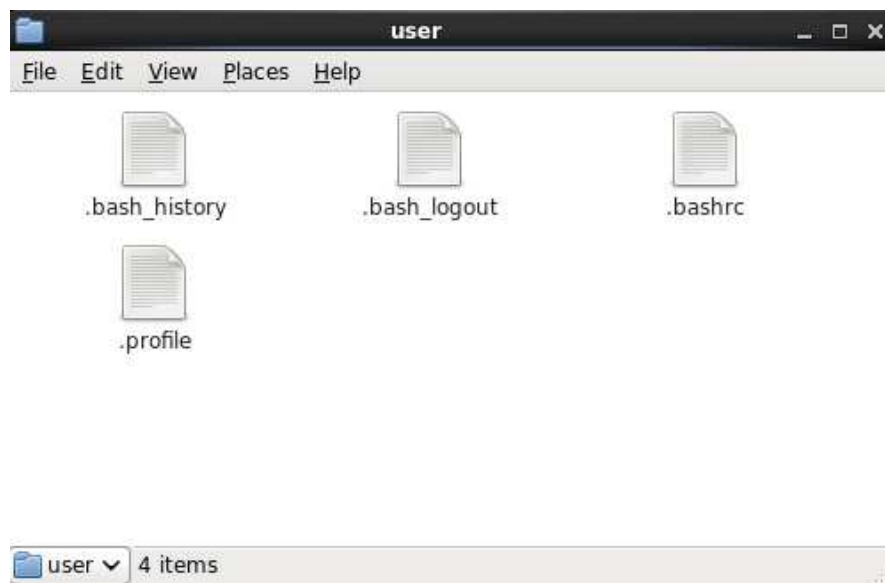


Figure 2.4: The Robot Home Window.

- 5. You can now transfer files from the `ele719` folder in the EE network drive to the robot by simply dragging them from the "ELE719 HOME Window" and dropping them into the "Robot Home Window", and vice versa.

The Robot3WD Python Module

You are now ready to explore the Robot3WD module. To do so, go to the terminal window connected to the robot and start up the Python interpreter by entering:

```
% python
```

at the command prompt. The system will respond by loading the Python interpreter and the Python command prompt “>>>” will appear. Now enter the following Python command to load the class definition of the `Robot` class from the ELE719 `Robot3WD` module:

```
>>> from Robot3WD import Robot
```

Once this is done, an *instance* of the 3WD robot can be defined using:

```
>>> sampling_period = 0.02
>>> myrobot = Robot(sampling_period)
```

These commands tell Python to create an instance of the `Robot` object, assign it to `myrobot` and set the sampling period of the robot controller to 0.02 second (i.e. a sampling frequency of 50 Hz).

To initialize the 3WD robot, the `Robot` class method `initialize()` is required:

```
>>> from math import pi
>>> theta_0 = 30.0*pi/180.0
>>> myrobot.initialize(theta_0)
```

Note that the `initialize()` method requires the initial orientation of the robot, θ_0 , as an argument. (Recall that, from Figure 2.1, the orientation of the robot is the angle of rotation θ from the base frame \mathcal{F}^0 to robot frame \mathcal{F}^R .) In this case the initial orientation is assumed to be 30° , but it has to be converted to radian before passing to the `initialize()` method.

Once this is entered, the motors and IMU will be calibrated and several threads will begin to run in the background to provide readings from the wheel encoders, IR sensors and the IMU. To retrieve these readings, the `get_readings_update()` method must be executed first:

```
>>> myrobot.get_readings_update()
```

The angular velocities of the wheels, readings of the IR sensors and orientation of the robot are defined as *properties* of the `Robot` class and can be accessed as follows:

```
>>> ang_vel = myrobot.angular_velocities
```

The above command obtains the angular velocities of the 3 wheels and assign them to the an array, `ang_vel`, of 3 elements as follows: the angular velocity of the right wheel is `ang_vel[0]`, the left wheel is `ang_vel[1]` and the back wheel is `ang_vel[2]`.

The raw values of the IR sensor readings are obtained using:

```
>>> ir_values = myrobot.ir_sensors_raw_values
```

The raw reading of sensor 0 is returned in `ir_values[0]`, sensor 1 in `ir_values[1]`, etc. These raw readings can then be converted into distance measurements for the purpose of avoiding obstacles. This will be done in a later experiment.

The orientation of the robot (i.e. the angle θ in Figure 2.1) is measured by the IMU and is obtained using:

```
>>> theta = myrobot.orientation
```

which assigns the current orientation (*in radian*) of the robot to the variable `theta`.

As discussed in the lecture, the linear and angular velocities of the robot can be controlled by setting the angular velocities of its wheels to the required values. Once the required wheel velocities are determined, they can be sent to the robot for execution using the `set_angular_velocities()` method as follows:

```
>>> myrobot.set_angular_velocities([wR, wL, wB])
```

where `wR`, `wL` and `wB` are the required angular velocities of the right, left and back wheels respectively. To stop the robot, one can set all its angular velocities to 0, or simply use the `stop()` method:

```
>>> myrobot.stop()
```

Finally, the `close()` method should be used to terminate the robot control program:

```
>>> myrobot.close()
```

The following program, available as `lab2.py`, shows how to program the 3WD robot to spin around by setting the angular velocities of the wheel at a constant value (in this case, 8 rad/sec):

```
from Robot3WD import Robot
from math import pi
from time import time

sampling_period = 0.02

run_time = 3.0

w = 8.0

myrobot = Robot(sampling_period)
```

```
theta_0 = 30.0*pi/180.0
myrobot.initialize(theta_0)

elapsed_time = 0.0
start_time = time()

while (elapsed_time < run_time):
    myrobot.get_readings_update()
    print 'angular velocities of wheels = ', myrobot.angular_velocities
    myrobot.setAngularVelocities([w, w, w])
    print 'ir sensors values = ', myrobot.ir_sensors_raw_values
    print 'robot orientation = ', myrobot.orientation*180.0/math.pi
    print ' '
    elapsed_time = time() - start_time

myrobot.stop()
myrobot.close()
```

The code inside the `while` loop will run until the elapsed time is longer than the 3 seconds specified by the variable `run_time`. The Python method `time()` from the `time` module is used to determine the current time.

Place the robot on the floor and enter the following command at the LINUX prompt:

```
% python lab2.py
```

The robot will execute the Python code in `lab2.py` and the robot should start spinning in the counter-clockwise direction at 8 rad/sec.

2.3 Laboratory Demonstration

Now that you are familiar with the `Robot3WD` module, review the program `lab2demo.py` developed in in Part 4 of the Preparatory Work to check if changes are required.

Demo 2.1 *Simple Motions*

Use an initial orientation of $\theta_0 = 30^\circ$ (or $\frac{\pi}{6}$ radian), run the program `lab2demo.py` and show the results to the laboratory instructor. To be successful, the robot should return close to its starting position and orientation.

Important Note: After the program finished execution, copy any files you wish to keep by dragging them from the “Robot Home Window” and dropping them into the “ELE719 Home Window”.

Now, enter the following commands to power down the robot:

```
% sync; sync;  
% shutdown -h now
```

A shutdown message will then appear. Wait until all the blue LED lights on the controller board are off, and then turn OFF the battery power switch and then the main power switch on the top plate of the robot.

2.4 What to Submit

A report is not required for this experiment. Archive the Python code developed as a .tar file (using your last name as the file name) and email it as an attachment to `ychen@ee.ryerson.ca`. For example,

```
tar cvf lastname.tar lab2demo.py myRobot.py
```

The .tar file must be sent by the end of the laboratory session.

Experiment 3

Forward Kinematics and Kinematic Control

Time to Complete: To be announced in class

Marks: 20 (prelab 15, lab 5)

Purpose

The motion control problem for the 3WD robot is solved using Kinematic Control in this experiment.

Objectives

By the end of this experiment, you will be able to:

1. Obtain the forward kinematics equation for this robot.
2. Implement a motion controller for the 3WD robot using the method of Kinematic Control.

3.1 The Motion Control Problem

The previous experiment considered several simple motions for the 3WD robot. These simple motions were achieved by first solving for the desired angular velocities of the 3 wheels (using inverse kinematics) and then setting them using the `set_angular_velocities()` method.

The `set_angular_velocities()` method then passes these velocities to the internal PI controllers for the wheels as their reference inputs $\dot{\Phi}^d = [\dot{\phi}_1^d, \dot{\phi}_2^d, \dot{\phi}_3^d]^T$. This is shown as the low-level controller in Figure 3.1.

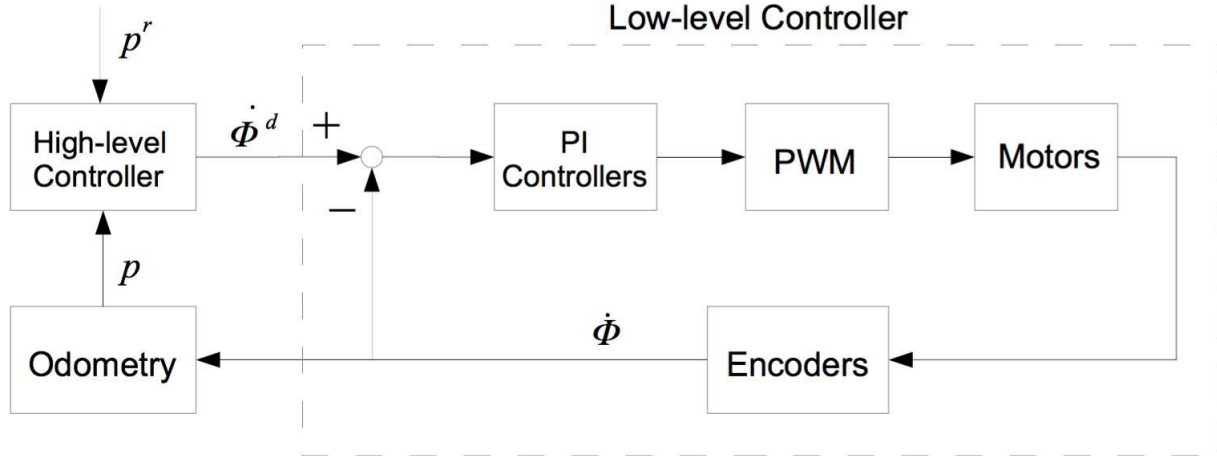


Figure 3.1: Kinematic control

The low-level controller in Figure 3.1 is only responsible for driving the wheels of the robot so that they reach the desired angular velocities. In a general motion control problem, the desired angular velocities of the wheels are not known *a priori*. In most cases, the robot is only required to go from an initial posture p_0 to a final posture p_f . (Recall that a posture p is defined to be the vector $[x, y, \theta]^T$, where (x, y) is the position and θ the orientation of the robot's frame \mathcal{F}^R w.r.t. the base reference frame \mathcal{F}^0 (see Figure 2.1).) So, in this case, the only information the robot has are $p_0 = [x_0, y_0, \theta_0]^T$ and $p_f = [x_f, y_f, \theta_f]^T$.

Similarly, if the robot is required to follow a required path, then a function of time, $p^r(t) = [x^r(t), y^r(t), \theta^r(t)]^T$, $0 \leq t \leq T_f$, describing the path would be provided instead. Thus, in a general motion control problem, one must determine the desired angular velocities based on the available information on either p_0 and p_f or $p^r(t)$.

The approach taken by Kinematic Control [1] is as follows:

1. First, obtain the actual angular velocities of the wheels $\dot{\Phi}$ from the encoder readings and determine the actual velocity of the robot $\dot{p} = [\dot{x}, \dot{y}, \dot{\theta}]^T$ using *forward kinematics* (see Eq. (2.2)):

$$\dot{p} = M(\theta)\dot{\Phi},$$

2. Determine the actual posture of the robot p by integrating (numerically) \dot{p} .
3. Compute the difference between the required posture and actual posture (i.e. the tracking error), $e = p^r - p$.

4. Determine the desired linear and angular velocities for the robot, $\dot{p}^d = [\dot{x}^d, \dot{y}^d, \dot{\theta}^d]^T$, to drive the tracking error e to zero using certain control law.

For example, suppose that the robot is required to move from an initial posture $p_0 = [x_0, y_0, \theta_0]^T$ to a final posture $p_f = [x_f, y_f, \theta_f]^T$ in a linear fashion within T_f seconds, then the reference path $p^r(t)$ can be defined as:

$$p^r(t) = p_0 + \Delta_p t, \quad 0 \leq t \leq T_f,$$

where

$$\Delta_p := \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta \theta \end{bmatrix} = \begin{bmatrix} (x_f - x_0)/T_f \\ (y_f - y_0)/T_f \\ (\theta_f - \theta_0)/T_f \end{bmatrix}.$$

Let the tracking error be

$$e = p^r(t) - p(t),$$

then the desired linear velocities of the robot, $\dot{p}^d(t)$ to correct this error can be determined using, for example, a PID control law:

$$\dot{p}^d(t) = K_p e(t) + K_i \int e(t) + K_d \frac{de(t)}{dt}, \quad (3.1)$$

and the desired angular velocities of the wheels, $\dot{\Phi}^d$, is then determined using inverse kinematics:

$$\dot{\Phi}^d = M(\theta) \dot{p}^d.$$

5. Set the angular velocities of the robot determined in the previous step using the `set_angular_velocities()` method to execute the motion.

Steps 1 and 2 correspond to the “Odometry” block, and Step 3 and 4 correspond to the “High-level Controller” block in Figure 3.1. This approach is known as *Kinematic Control* because only the kinematics model for the robot is required. The dynamics model for the robot is not required.

3.2 Preparatory Work

1. Let

$$A = \begin{bmatrix} 0 & -1 & -L \\ \frac{\sqrt{3}}{2} & \frac{1}{2} & -L \\ -\frac{\sqrt{3}}{2} & \frac{1}{2} & -L \end{bmatrix}, \quad R = \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Verify that the matrix M^{-1} in the inverse kinematics equation, Eq. (2.2), is $M^{-1} = AR$, and use this fact to show that the forward kinematics equation is given by:

$$\dot{p} = M(\theta)\dot{\Phi}, \quad (3.2)$$

where $\dot{p} = [\dot{x}, \dot{y}, \dot{\theta}]^T$, $\dot{\Phi} = [\dot{\phi}_1, \dot{\phi}_2, \dot{\phi}_3]^T$, and

$$M(\theta) = r \begin{bmatrix} \frac{2}{3} \sin(\theta) & \frac{\sqrt{3}}{3} \cos(\theta) - \frac{1}{3} \sin(\theta) & -\frac{\sqrt{3}}{3} \cos(\theta) - \frac{1}{3} \sin(\theta) \\ -\frac{2}{3} \cos(\theta) & \frac{\sqrt{3}}{3} \sin(\theta) + \frac{1}{3} \cos(\theta) & -\frac{\sqrt{3}}{3} \sin(\theta) + \frac{1}{3} \cos(\theta) \\ -\frac{1}{3L} & -\frac{1}{3L} & -\frac{1}{3L} \end{bmatrix}. \quad (3.3)$$

You may use MAPLE or MATLAB symbolic toolbox for the computation. (Hints: Use the matrix identity $M = (M^{-1})^{-1} = (AR)^{-1} = R^{-1}A^{-1}$).

2. Extend the object class `myRobot` from the previous experiment by adding the method `forward_kinematics()` to implement the forward kinematics equation in Eq. (3.2). The following template for this method can be found in `myRobot.py`:

```
def forward_kinematics(self, wheel_angular_velocities, theta):
    L = self._L
    wheel_radius = self._wheel_radius
    #... (Fill in rest of the code here) ...
    return p_dot
```

3. One of the fundamental problems in the mobile robot navigation is to determine its current posture, $p = [x, y, \theta]^T$. This is known as the “Dead Reckoning” or “Localization” problem. A vast amount of research has been done in this area and significant progress has been made. Here, we will only consider the most basic method for localization, namely, localization using odometry information only.

As discussed in Steps 1 and 2 of the Kinematic Control approach earlier, the angular velocities of the robot’s wheels, $\dot{\Phi}$, are available from the readings of the wheel encoders. Hence, the velocities of the robot w.r.t. \mathcal{F}^0 , $\dot{p} = [\dot{x}, \dot{y}, \dot{\theta}]^T$, can be determined using forward kinematics:

$$\dot{p} = M(\theta)\dot{\Phi}.$$

Once \dot{p} is known, the posture of the robot w.r.t. \mathcal{F}^0 , p , can be obtained by integration:

$$p(t) = \int_0^t \dot{p}(\tau) d\tau.$$

Since the controller is implemented digitally, the above integral is approximated numerically instead. If the Euler method is used, then the estimate of the posture at

time t_k , $p(t_k)$, can be determined using the following *odometry update equation*:

$$p(t_k) = p(t_{k-1}) + \dot{p}(t_{k-1})T_s, \quad (3.4)$$

where $p(t_{k-1})$ is the previous estimate of the posture p , and $T_s = t_k - t_{k-1}$ is the sampling period used by the digital controller.

Follow Steps 1 to 5 of Section 3.1 to write a program `lab3demo.py` that implements the Kinematic Control Scheme to move the robot from an initial posture of $p_0 = [0.0, 0.0, \frac{\pi}{6}]^T$ to a final posture of $p_f = [0.5, 0.5, \frac{\pi}{2}]^T$ in a linear fashion in 6 seconds (see Figure 3.2). Use Eq. (3.4) to estimate the posture of the robot.

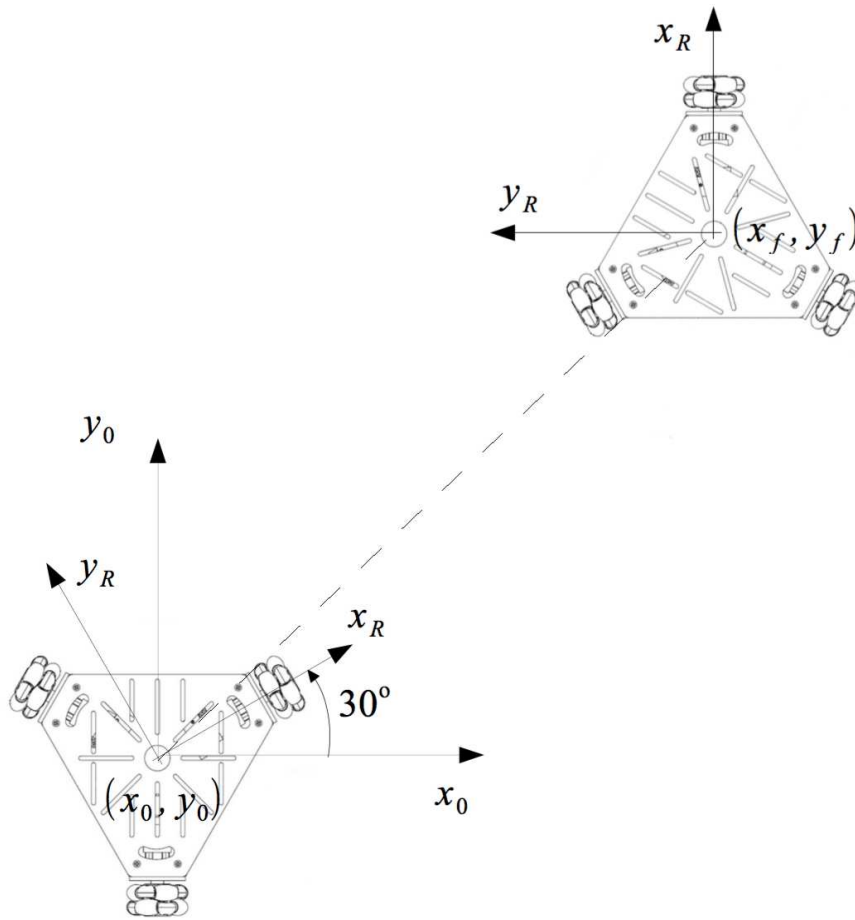


Figure 3.2: Straight-line motion

An object class, `PIDControllers()`, that implements the PID controller equation (3.1) is available in the `Robot3WD` module. The `PIDControllers()` object class requires the controller gains K_p , K_i and K_d at initialization. Each of these controller gains must

be passed as arrays of 3 elements, corresponding to the gain values to be used for the right, left and back wheels respectively. Once initialized, it can then be used to determine the control values to be applied to the wheels. Again, the control values are returned as an array of 3 elements, `[uR, uL, uB]`, corresponding to the control values to be applied to the right, left and back wheels respectively. The following example shows how `PIDControllers` class is used:

```
from myRobot import myRobot
from Robot3WD.PID import PIDControllers
from numpy import array, dot

...
T_s = 0.02
robot = myRobot(T_s)

...
# define the controller gains Kp, Ki and Kd
Kp = array([1.0, 1.0, 1.0]).T
Ki = array([1.0, 1.0, 1.0]).T
Kd = array([0.01, 0.01, 0.01]).T

# create an instance of the PID controllers object and initialize it
pid = PIDControllers(Kp, Ki, Kd, T_s)

...
# compute desired linear velocities of robot using the PID control law
# p_r = reference posture at time t_k
# p = actual posture of robot
pr_dot = pid(p_r, p)
```

3.3 Laboratory Demonstration

Demo 3.1 *Kinematics Control*

Use the controller gains K_p , K_i and K_d defined above and the following values to test the controller program `lab3demo.py`: $(x_0, y_0, \theta_0) = (0.0, 0.0, \frac{\pi}{6})$, $(x_f, y_f, \theta_f) = (0.7, 1.0, \frac{\pi}{2})$, $T_f = 8.0$ and $T_s = 0.02$. Run the controller program and show it to the laboratory instructor. To be successful, the norm of the tracking error at the final posture determined with `np.linalg.norm(p_f - p)` should be less than 0.036. This corresponds to a maximum of 0.5 cm error in both the final x and y positions and a 2° error in the orientation θ . Your program should print out the value of the final error norm for verification by the laboratory instructor. You should also measure the robot's final posture to see if it agrees with the required value. Plot the resulting path as y against x , and also θ against time and show them to the laboratory instructor.

The odometry update equation (3.4) provides estimates of the robot's posture $p = [x, y, \theta]^T$ based on previous estimates. Since this equation is a numerical approximation of the integration operation, errors in the estimates will occur and they will accumulate over time. Instead of using the estimate of θ obtained from the odometry equation, one can replace it with the *measured* value provided by the IMU to improve on the estimate. In general, the process of using sensor measurements for localization improvement is known as *sensor fusion*. In our case, this can be done by replacing the calculated update value of `theta` based on the odometry equation (3.4) with the measured value from the IMU. This can be done by adding the statement

```
p[2] = robot.orientation
```

after the odometry update equation.

Demo 3.2 *Kinematics Control with Simple Sensor Fusion*

Repeat Demo 3.1 with the above modification and see if there is any improvement on the robot's performance.

3.4 What to Submit

A report is not required for this experiment. Archive the Python code developed as a .tar file (using your last name as the file name) and email it as an attachment to `ychen@ee.ryerson.ca`. For example,

```
tar cvf lastname.tar lab3demo.py myRobot.py
```

The .tar file must be sent by the end of the laboratory session.

3.5 Reference

1. C. Canudas de Wit, *et al.*, *Theory of Robot Control*, New York, Springer-Verlag, 1996.

Experiment 4

Obstacle Avoidance Control

Time to Complete: To be announced in class

Marks: 20 (prelab 15, lab 5)

Purpose

The problem of navigating the 3WD robot to a goal position while avoiding collision with obstacles is considered in this experiment.

Objectives

By the end of this experiment, you will be able to:

1. Convert the raw readings of the IR sensors into distance measurements.
2. Implement a control strategy for the 3WD robot to do simple obstacle avoidance while navigating to a goal position.

4.1 Obstacle Avoidance Controller Design

We will begin with the following obstacle avoidance strategy [1]:

- Move the robot towards its final goal position until an obstacle is detected by the IR sensors.
- If an obstacle is detected, change the goal position temporarily until the obstacle is cleared.

- Restore the goal position once the obstacle is cleared and repeat the cycle until the final goal position is reached.

For simplicity, the final orientation of the robot will not be considered here.

Consider Figure 4.1, where it is assumed that the robot is moving from the current position (x, y) towards a final goal position of (x_f, y_f) with linear velocity $V = [\dot{x}, \dot{y}]^T$ and angular velocity $\omega = \dot{\theta}$.

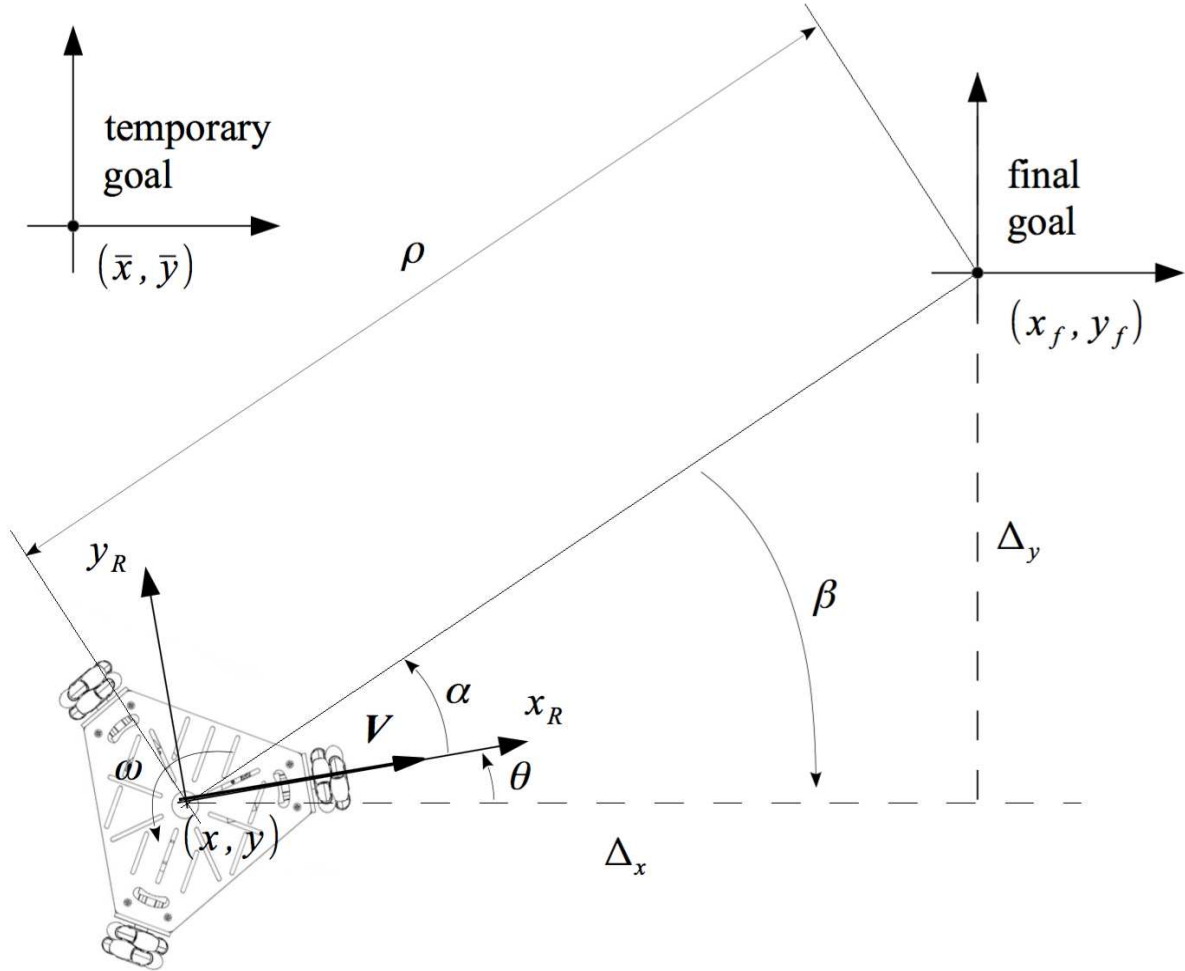


Figure 4.1: 3WD Robot and reference frames.

Using the notations in Figure 4.1, define the *distance error*, ρ , as the distance of the robot from its current position to the final goal position, i.e.

$$\rho = \sqrt{\Delta_x^2 + \Delta_y^2} = \sqrt{(x_f - x)^2 + (y_f - y)^2} \quad (4.1)$$

and the *heading error*, α , is defined as

$$\alpha = \text{atan2}(\Delta_y, \Delta_x) - \theta. \quad (4.2)$$

Also, the angle β is

$$\beta = -\theta - \alpha. \quad (4.3)$$

If an obstacle is detected then the final goal position (x_f, y_f) in the above equation will be temporarily replaced by (\bar{x}, \bar{y}) .

The obstacle avoidance controller will be designed using the following steps:

1. Determine the temporary goal position once an obstacle is detected, and to switch between the final and temporary goal positions.
2. Design a controller to drive ρ and α to 0 given any goal position (final or temporary).

4.1.1 Step 1: Collision Avoidance

The distance measurements from the IR sensors will be used to determine the temporary goal position and heading error. The 3WD robot is equipped with 6 IR sensors, S_0, \dots, S_5 . These sensors are mounted on a circular plate on top of the robot as shown by the sensor frames $\mathcal{F}^{S_i} = o_{S_i} x_{S_i} y_{S_i}$, ($i = 0, \dots, 5$) in Figure 4.2:

Let the distance measurement returned by sensors be d_i ($i = 0, \dots, 5$), then we assume the robot is at risk of collision if

$$\min_i d_i \leq d_{min}, \quad (4.4)$$

where d_{min} is a pre-defined *minimum safe distance* from obstacle. The value of d_{min} depends on several factors, such as the range of the sensors, how fast the robot can move, how fast the sensor readings are processed, etc. In our case, the value of d_{min} is set to 0.08m or 8cm. Similarly, we assume that the robot is free from collision if

$$\min_i d_i \geq d_{free}, \quad (4.5)$$

where d_{free} is a pre-defined *collision-free distance*. In our case, the value of d_{free} is set to be $1.25 \times d_{min}$.

Consider a typical sensor frame \mathcal{F}^{S_i} in Figure 4.3 where δ_i is defined to be a vector of length d_i along the x_{S_i} axis and is represented by the following homogeneous coordinate:

$$\delta_i = \begin{bmatrix} d_i \\ 0 \\ 1 \end{bmatrix} \quad (\text{w.r.t. the sensor frame } \mathcal{F}^{S_i}).$$

The z component of the homogeneous coordinate is omitted here because the motion of the robot is planar only. Now, if $d_i \leq d_{min}$ then the vector δ_i is regarded as a direction for which

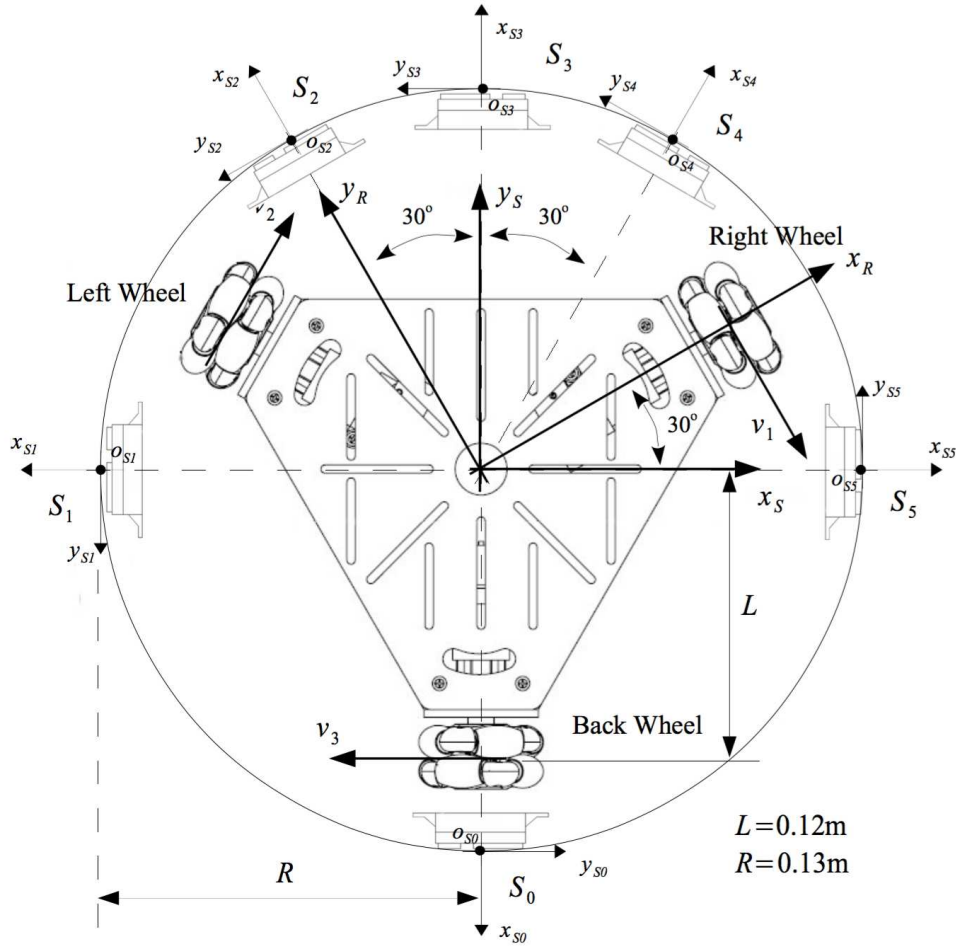


Figure 4.2: IR sensors locations.

an obstacle is present and the robot should not move along that direction. On the other hand, if $d_i \geq d_{free}$ then the vector δ_i is regarded as a collision-free direction for the robot. Therefore, if an obstacle has been detected then a *temporary goal vector* δ_{goal}^0 (w.r.t. the base frame \mathcal{F}^0) can be determined by combining all the collision-free direction vectors δ_i together as follows:

$$\delta_{goal}^0 := \begin{bmatrix} \bar{x} \\ \bar{y} \\ 1 \end{bmatrix} = \sum_{\{\delta_i^0 \text{ s.t. } d_i \geq d_{free}\}} \delta_i^0, \quad (4.6)$$

where δ_i^0 are δ_i w.r.t. \mathcal{F}^0 and can be determined from

$$\delta_i^0 = H_{S_i}^0 \delta_i = H_R^0 H_{S_i}^R \delta_i = H_R^0 H_{S_i}^R \begin{bmatrix} d_i \\ 0 \\ 1 \end{bmatrix}, \quad (4.7)$$

where $H_{S_i}^R$ is the homogeneous transformation matrix (or H-matrix) from the sensor frame

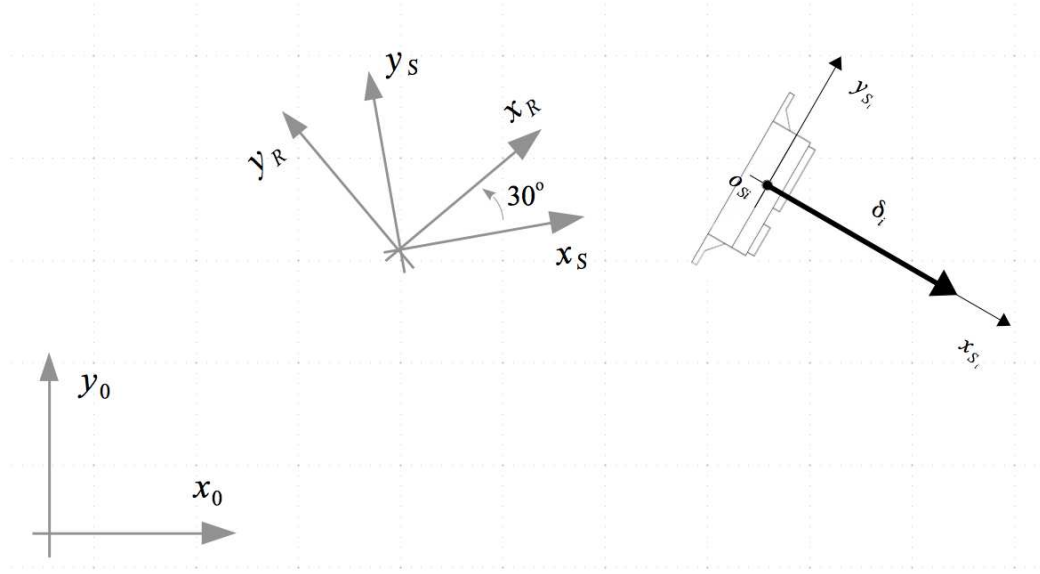


Figure 4.3: A typical sensor frame.

\mathcal{F}^{S_i} to the robot frame \mathcal{F}^R . Since the sensors are fixed on the robot, the matrices $H_{S_i}^R$ ($i = 0, \dots, 5$) are constant and can be pre-determined. The matrix H_R^0 is simply the H-matrix from the robot frame \mathcal{F}^R to the base frame \mathcal{F}^0 and can be determined from the robot's current posture $p = [x, y, \theta]^T$ as follows:

$$H_R^0 = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & x \\ \sin(\theta) & \cos(\theta) & y \\ 0 & 0 & 1 \end{bmatrix}. \quad (4.8)$$

The collision avoidance algorithm is now summarized:

1. Obtain the sensor measurements d_i , $i = 0, \dots, 5$.
2. Use Eq. (4.4) to determine if there is risk of collision.
3. If there is risk of collision, then determine a temporary goal position (\bar{x}, \bar{y}) using Eq. (4.6) and then the temporary heading error $\bar{\alpha}$ using

$$\bar{\alpha} = \text{atan2}(\bar{y}, \bar{x}) - \theta, \quad (4.9)$$

as well as the temporary distance error $\bar{\rho}$ using

$$\bar{\rho} = \sqrt{\bar{\Delta}_x^2 + \bar{\Delta}_y^2} = \sqrt{(x_f - \bar{x})^2 + (y_f - \bar{y})^2} \quad (4.10)$$

4. Use $\bar{\rho}$ and $\bar{\alpha}$ to determine the control values to move the robot until the obstacle is clear.

4.1.2 Exponential Stabilization

In this section a controller is introduced to drive the robot to its goal position (final or temporary). This controller has the property that it makes the closed-loop system to be *exponentially stable* [2]. Details on this controller have been discussed in the lecture and will not be repeated here.

Using the results in [2], the desired angular velocities to drive the robot to its goal position are determined as follows:

1. The desired linear and angular velocities *of the robot* are first determined using the following linear control law:

$$v = k_\rho \rho \quad (4.11)$$

$$\omega = k_\alpha \alpha + k_\beta \beta \quad (4.12)$$

where ρ , α are defined as before, and $\beta = -\theta - \alpha$. The constants k_ρ , k_α and k_β are scalar controller gains such that

$$k_\rho > 0, \quad k_\beta < 0, \quad k_\alpha > \frac{2}{\pi}k_\rho - \frac{5}{3}k_\beta.$$

If an obstacle is detected, then $\bar{\rho}$ and $\bar{\alpha}$ are used instead to determine v and ω .

2. Once v and ω are determined, the desired wheel angular velocities $\dot{\Phi}$ are determined from the inverse kinematics equation (2.1) with \dot{p} determined from:

$$\dot{p} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} v \cos(\theta) \\ v \sin(\theta) \\ \omega \end{bmatrix}.$$

3. The desired wheel angular velocities $\dot{\Phi}$ are then executed using the `set_angular_velocities()` method.

4.2 Preparatory Work

1. The collision avoidance algorithm discussed earlier requires the computation of the temporary goal vector δ_{goal}^0 using Eq. (4.6). In order to do this, the vectors δ_i must be transformed to the base frame \mathcal{F}^0 using Eq. (4.7) which requires knowledge of the H-matrices H_R^0 and $H_{S_i}^R$. The H-matrix H_R^0 can be determined during run time using Eq. (4.8). The matrices $H_{S_i}^R$ are constant and depend only on how the sensors are mounted on the robot. Use the sensor locations in Figure 4.2 as a guide, show that the origin of the sensor frame \mathcal{F}^{S_i} , i.e. o_{S_i} and the angle from x_R to x_{S_i} , namely θ_{S_i} , with respect to the robot frame \mathcal{F}^R are given by the following Table:

Table 4.1: Sensor frame locations (with respect to robot frame \mathcal{F}^R)

Sensor S_i	x coord. of o_{S_i}	y coord. of o_{S_i}	θ_{S_i}
	$x_{o_{S_i}}$	$y_{o_{S_i}}$	
S_0	$-\frac{1}{2}R$	$-\frac{\sqrt{3}}{2}R$	$-\frac{2}{3}\pi$
S_1	$-\frac{\sqrt{3}}{2}R$	$\frac{1}{2}R$	$\frac{5}{6}\pi$
S_2	0	R	$\frac{1}{2}\pi$
S_3	$\frac{1}{2}R$	$\frac{\sqrt{3}}{2}R$	$\frac{1}{3}\pi$
S_4	$\frac{\sqrt{3}}{2}R$	$\frac{1}{2}R$	$\frac{1}{6}\pi$
S_5	$\frac{\sqrt{3}}{2}R$	$-\frac{1}{2}R$	$-\frac{1}{6}\pi$

2. The homogeneous transformation matrix for a planar motion of a rotation of amount $\Delta\theta$ about the z -axis and a translation of Δx along the x -axis and a translation of Δy along the y -axis can be represented by the following 3×3 matrix:

$$H := \begin{bmatrix} \cos(\Delta\theta) & -\sin(\Delta\theta) & \Delta x \\ \sin(\Delta\theta) & \cos(\Delta\theta) & \Delta y \\ 0 & 0 & 1 \end{bmatrix}. \quad (4.13)$$

Let $v = [\Delta x, \Delta y, \Delta\theta]^T$, extend the `myRobot` module by adding a function `HMatrix(v)` that returns the H-matrix in Eq. (4.13) given v . This function can then be used to obtain the matrices H_R^0 and $H_{S_i}^R$ required to determine δ_i^0 in Eq. (4.7). For example,

```
HRO = HMatrix([delta_x, delta_y, delta_theta])
```

3. The IR sensors on the robot are analog distance measurement sensors. These sensors provide an analog voltage output based on the measured distance. Unfortunately, the relationship between the sensor's output voltage and the measured distance is nonlinear. In addition, the analog voltage reading V_S from the sensor must also be converted using an analog to digital converter (ADC) before it can be used by the digital controller.

In order to determine the relationship between the sensor's output voltage and measured distance d , an experiment using the setup in Figure 4.4 was carried out. The distance d was varied from 3 cm to 40 cm and the corresponding values of V_{raw} were obtained in Table 4.2 and plotted in Figure 4.5.

Figure 4.5 suggests that d is some kind of exponential function of V_{raw} . In fact, the following function was found to be an excellent fit to the data:

$$d = c_1 e^{-c_2 \sqrt{V_{\text{raw}}}}. \quad (4.14)$$

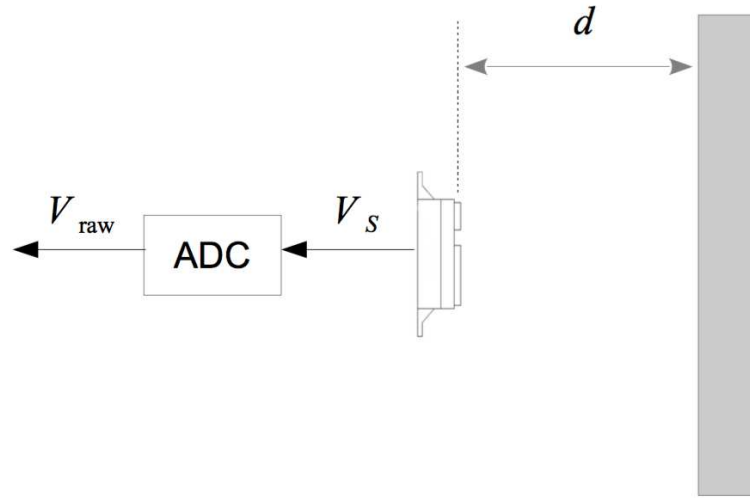


Figure 4.4: Sensor experiment.

Table 4.2: Distance versus raw sensor values

V_{raw}	3106	2727	1622	1084	662	425	284	186	121	76
d (m)	0.03	0.04	0.07	0.10	0.15	0.20	0.25	0.30	0.35	0.40

Now, use the data in Table 4.2 and the Method of Least Squares, determine the constants c_1 and c_2 in Eq. (4.14).

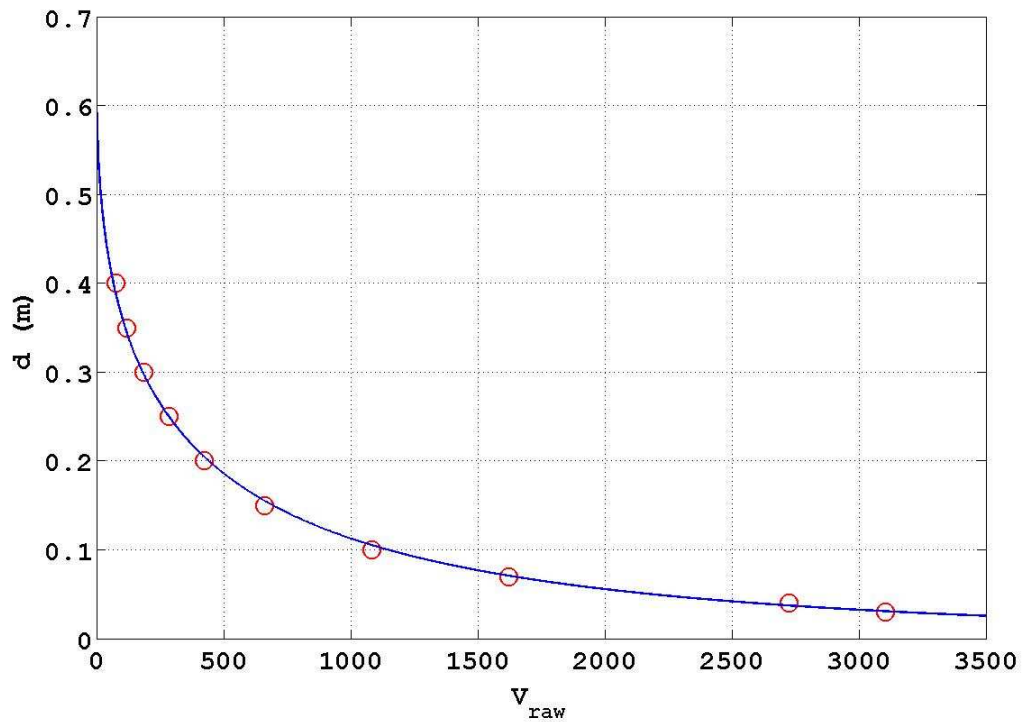
4. Extend the `myRobot` module by implementing Eq. (4.14) as the function

```
distance = Vraw_to_distance(Vraw)
```

Once this function is defined, it can then be used to obtain distance measurements from the IR sensors as follows:

```
for i in range(0,6):
    distance[i] = Vraw_to_distance(myRobot.ir_sensors_raw_values[i])
```

5. Implement the exponentially stabilizing controller and the obstacle avoidance strategy using the program template `lab4demo.py` available from the course web page as a guide.

Figure 4.5: d versus V_{raw}

4.3 Laboratory Work

Demo 4.1 *Obstacle Avoidance with Exponentially Stabilizing Controller*

Using $k_\rho = 1.0$, $k_\beta = -1.0$, run your controller program to move the robot to the final goal position of $(0.7, 1.0)$. Run the program first with no obstacle present and record the path taken by the robot. Next put an obstacle at the location specified by the laboratory instructor and run the program again. To be successful, the robot must not collide with the obstacle and must reach the final goal position within ± 1 cm in the final x and y position.

4.4 What to Submit

A report is required for this experiment. Requirements for the report are specified in Appendix A. In addition, archive the Python code developed as a .tar file (using your last name

as the file name) and email it as an attachment to `ychen@ee.ryerson.ca`. For example,

```
tar cvf lastname.tar lab4demo.py myRobot.py
```

The .tar file must be sent by the end of the laboratory session.

4.5 Reference

1. M. Egerstedt, “Control of Autonomous Mobile Robots,” in *Handbook of Networked and Embedded Control*, D. Hristu and B. Levine, Eds., Birkhauser, Boston, MA, 2005.
2. C. Canudas de Wit and O.J. Sordalen, “Exponential Stabilization of Mobile Robots with Nonholonomic Constraints,” in *IEEE Transactions on Automatic Control*, 37(11), 1992.

Appendix A

Report Requirements

Work completed on all parts of experiments, exercises, prelab work, and all program listings is to be reported. Reports can be hand written *neatly in ink*, or typed, or prepared on a word processor. Reports are graded according to the correctness of the prelab work, experimental results, and answers to exercises. Reports are also assessed based on the communication skills of the authors. However, reports are judged by their contents and not their appearance. Reports must be handed in either to the course professor or lab instructor on the specific deadline announced in class. No late report will be accepted.

All reports are to be prepared according to the following format:

Report Format

Table of contents

Introduction – background, purpose

Theory

Preparatory work

Experimental method, apparatus

Results, observations, figures and graphs, answers, comments

Conclusion, recommendations

Answers to exercises

Commented program listings

All figures, graphs and program listings must be captioned and numbered.

Comments and Suggestions

1. Did you find errors in the manual? Please give page number and a brief description of each error.
2. Did you find the manual clear and easy to understand? Please comment on specific sections that you feel need improvement.

3. Which sections of the manual do you consider most important/least important?

4. Which experiments do you consider most important/least important?