Machine Learning with Robotics: CSCI 4155/6505 2012

Thomas P. Trappenberg
Dalhousie University

Acknowledgements

These lecture notes have been inspired by several great sources, which I recommend as further readings. In particular, Andrew Ng from Stanford University has several lecture notes on Machine Learning (CS229) and Artificial Intelligence: Principles and Techniques (CS221). His lecture notes and video links to his lectures are available on his web site (`http://robotics.stanford.edu/~ang`) and he is also teaching online courses. Excellent books on the theory of machine learning are *Introduction to Machine Learning* by Ethem Alpaydin, 2nd edition, MIT Press 2010, and *Pattern Recognition and Machine Learning* by Christopher Bishop, Springer 2006. A wonderful book on Robotics with a focus of Bayesian models is *Probabilistic Robotics* by S. Thrun, W. Burgard, and D. Fox, MIT Press 2005, and the standard book on RL is *Reinforcement Learning: An Introduction* by Richard Sutton and Andrew Barto, MIT press, 1998. The popular AI, *Artificial Intelligence: A Modern Approach* by Stuart Russell and Peter Norvig, 2nd edition, Prentice Hall, 2003, does also include some chapters on Machine Learning and Robotics.

Several people have contributed considerably to the development of examples in this book. In particular, Leah Brown and Ian Graven have created early examples of Lego Mindstorm implementations in Matlab and André Reis de Geus and Jerome Verney helped considerably with the Python migration. I also thank Paul Hollensen and Patrick Connor for many useful comments, and Chris Maxwell for helped with some implementation issues.

Preface

Only a few years ago it seemed like science fiction that cars would drive safely through San Francisco by themselves, that computers could search photographs for content, or that game consoles could 'see' your posture. The progress has been so rapid in recent years that large companies like Google and Microsoft are not only taking notice but are actively advancing and employing this field. Engineers have worked on these dreams for decades, so what has made this enormous progress possible?

Traditional science and engineering, so successful in flying men to the moon and building ever taller buildings, has long recognized that dynamic environments like city traffic or human interactions are challenging for traditional methods. In a traditional way of designing solutions, smart engineers use the laws of physics and mathematics to predict future states so that the machine can execute manipulations as desired. But the problem with dynamic and complex environments is that future states are difficult to predict for several reasons. For example, our sensors or information processors might be too slow or inaccurate to be able to predict even measure environment sufficiently, and the sheer number of possible states is another common problem.

Two major ingredients have been contributing to the recent success. The first is the acknowledgment that the world is uncertainty. Making this the premise from the outset has drastic consequences. For example, rather than following only the most likely explanations for a given situation, keeping an open mind and considering also other possible explanations has proven to be essential in systems that have to work in a real world environment in contrast to very controlled lab environment. The language of describing uncertainty, that of probability theory, has proven to be elegant and tremendously simplifies arguing in such worlds. This book is an introduction to the probabilistic formulation of machine learning.

The second ingredient for the recent breakthroughs is building system that adapt to unforeseen events. In other words, we must build learning machines since the traditional method of encoding appropriate responses to all future situations is impossible. Like humans, machines should not be static entities that only blindly follow orders which might be outdated by the time real situations are encountered. Although learning machines have been studied for at least half a century, often inspired by human capabilities, the field has matured considerably in recent years through more rigorous formulations of early learning machines and the realization of the importance of predicting previously unseen events rather than only memorizing previous events. Machine learning is now a well established discipline within artificial intelligence.

The power of the probabilistic approach to machine learning can best be acknowledged and experienced in the real world. We therefore chose to demonstrate and illuminate the theoretical constructs with robotics environments. The Lego system was chosen as it is not only one of the cheapest systems around, but the limitations of the system compared to more elaborate robots provide provide a nice challenge for smart solutions over gloomy hardware. We also choose a high-level programming language

so that we can concentrate on algorithmic programming and minimize system-level hacking.

Contents

1 Background

1.1 Some history of AI and machine learning

Artificial Intelligence(AI) has many sub-discipline such as knowledge representation, expert systems, search, etc. This course will focus on how machines can learn to improve their performance or learn how to solve problems. **Machine learning** is now widely respected scientific area with a growing number of applications. Machine learning approaches have enabled new approaches to information processing that made possible new consumer electronics as well as breakthroughs in robotics.

While the field has matured considerably in the last decade, its roots and early breakthroughs are firmly grounded in the 1950s. The history of AI is tightly interwoven with the history of machine learning. For example, Arthur Samuel's checkers program from the 1950s, which has been celebrated as an early AI hallmark, was able to learn from experience and thereby was able to outperform its creator. Basic Neural Networks, such as Bernhard Widrow's ADALINE (1959), Karl Steinbuch's Lernmatrix (around 1960), and Frank Rosenblatt's Perceptron (late 1960s), have sparked the imaginations of researcher about brain-like information processing systems. And Richard Bellman's Dynamic Programming (1953) has created a lot of excitement during the 1950s and 60s, and is now considered the foundation of reinforcement learning.

Biological systems have often been an inspiration for understanding learning systems and visa versa. Donald Hebb's book *The Organization of Behavior* (1943) has been very influential not only in the cognitive science community, but has been marvelled in the early computer science community. While Hebb speculated how self-organizing mechanisms can enable human behavior, it was Eduardo Caianiello's influential paper *Outline of a theory of thought-processes and thinking machines* (1961) who quantified these ideas into two important equations, the neuronic equation, describing how the functional elements of the networks behave, and the memnonic equation that describe how these systems learn. This opened the doors to more theoretical investigations. But such investigations came to a sudden halt after Marvin Minsky and Seymore Papert published their book *Perceptrons* in 1969 with a proof that simple perceptrons can not learn all problems. At this time, likely somewhat triggered by the vacuum in AI research by the departure of learning networks, mainstream AI shifted towards expert systems.

Neural Networks became again popular in the mid 1980 after the backpropagation algorithms was popularized by David Rumelhart, Geoffrey Hinton and Ronald Williams (1986). There was a brief period of extreme hype with claims that neural networks can now forecast the stock-market and how these learning systems will quickly become little brains. The hype backslashed somewhat. Neural Networks predictive power in scientific explanations became questions as they seem to always fit any data, and early claims of the future progress has not substantiated. But the understanding

Fig. 1.1 Some AI pioneers AI.From top left to bottom right: Alan Turing, Frank Rosenblatt, Geoffrey Hinton, Arthur Lee Samuel, Richard Bellman, Judea Pearl.

of both these problems, which are related to **generalizability** and **scalability** have matured this field considerably since.

Recent progress was made possible through several factors. A major contributed is certainly a better grounding in statistical learning theory with more rigorous mathematical insights. A good example is the work by Vladimir Vapnik as published in his book *The Nature of Statistical Learning Theory* in 1995. And more generally, Bayesian methods and graphical models (Judea Pearl, 1985) have clarified and transformed the field, with many important advancements outlined in this book.

1.2 Why machines should learn

Traditional AI provides many useful approaches for solving specific problems. For example, search algorithms can be used to navigate mazes or to find scheduling solutions. And expert systems can manage large databases of expert knowledge that can be used to argue (infer) about specific situations. While such strategies might be well suited for some applications, learning systems are usually more general and can be applied to situations for which closed solutions are not known. Also, a major challenge in many AI applications has been that systems change over time and that systems encounter situations for which they were not designed. For example, robotics systems are often helpless when employed outside their common environment. Some form of adaptation to changing situations and generalizations to unseen environments or unforeseen circumstances are important for may systems. While many AI systems

have adaptive components, we are specifically concentrating on the theory of learning machines in this course.

Learning machines are suppose to learn from the environment, either through instructions, by reward or punishment, or just by exploring the environment and learning about typical objects or spatio-temporal relations. For the systematic discussion of learning systems it is useful to distinguish three types of learning circumstances, namely supervised learning, reinforcement learning, and unsupervised learning. **Supervised learning** is characterized by using explicit examples with labels that the system should use to learn to predict labels of previously unseen data. The training labels can be seen as being supplied by a teacher who tells the system exactly the desired answer. **Reinforcement learning** is somewhat similar in that the system receives some feedback from the environment, but this feedback is only reward or punishment for previously taken actions and thus typically delay in time. The goal of reinforcement learning is to discover the action, or a sequence of actions, which maximize reward over time. Finally, the aim of **unsupervised learning** is to find useful representations of data based on regularities of data without labels. Smart representation of data is often the key of smart solutions, and this type of learning is often more applicable since no labels are required. While it is useful to distinguish such classical learning systems, they can also augment each other. Our ultimate goal is to model the world and to use such models to make 'smart' decisions in order to maximize reward.

Learning systems can help to solve problems for which more direct solutions are not known, another common problem in AI applications is that systems are generally **unreliable** and that environments are **uncertain**. For example, a program might read data in a specific format, but some user might supply corrupted files. Indeed, productions software is often lengthy only to consider all kind of situations that could occur, and we now realize that considering all possibilities is often impossible. Unreliable inputs is very apparent in robotics where sensors are often noisy or have severe limitations. Also, the state of a system might be unclear due to limited data or the inability to process data sufficiently in time with limited resources.

Major progress in many AI areas, in particular in robotics and machine learning, has been made by using concepts of (Bayesian) probability theory. This language of probability theory is appropriate since it acknowledges the fundamental limitations we have in real world applications (such as limited computational researches or inaccurate sensors). The language of probability theory has certainly helped to unify much of related areas and improved communication between researchers. Furthermore, we will see that the representation of uncertain states as a probabilistic map will be very useful. Probability theory will also provide us with the solution for a basic computational need, that of combining prior knowledge with new evidence.

1.3 Scientific computing with Python

While some of the material is theoretical in nature, it is fun and educational to get it to work in either a simulated environment or with the help of physical robots. The brain of our devises will be little programs that control a Lego robot or analyses data given to them. We will use a Python programming environment for this work. This high level language is has a lot of support for the kind of things we want to do, such

as computer vision and Lego control. It is also freely available and can be installed on several systems. We will be using it in a Windows environment, but similar systems can be build on Linux, and with some restrictions, on Mac computers. One of the problems of Python compared to other high level languages such as Matlab or R is that there seems to be and endless pool of packages. In this section we will introduce the basic python environment that we use in this course.

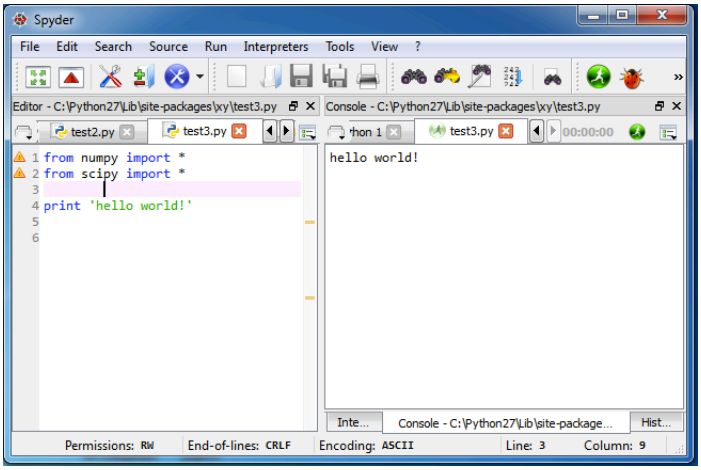


Fig. 1.2 The Spyder programming environment with editor window with command window.

Our environment is based on the Python(x,y) distributions from
`http://code.google.com/p/pythonxy/wiki/Welcome`,
which includes the basic Python interpreter, several common library packages, some of which we will use below, and a scientific programming environment called Spyder. Figure **??** shows a basic Spyder window configuration with an editor window and an interpreter window in which the program is executed. Since Python is an interpreted language, no compilation step is required and you can type the following examples right into the interpreter window at the command prompt. Alternatively, you can write the commands in a file, conveniently using the editor window of Spyder and execute it with F5 or the run button.

While the Python(x,y) distributions for windows installed most of the packages we need in this course, we still have to import specific libraries in each program. To simplify this we will use the PyLab library which itself is a collection of core packages for scientific computing including NumPy, SciPy, Matplotlib. Thus, we will be using the command

```
from pylab import *
```

at the beginning of our script. This will import the methods to the name space of our current environment. An alternative method to import a library is to use a statement like

```
import pylab as pl
```

This has the advantage that the names of the methods will be uniquely defines with the prescript `pl.`. However, since our programs are generally small, and to keep the

script as simple as possible, we follow here the first notation.

1.3.1 Numpy array

We are mainly working on data collections with data of the same type. An appropriate fundamental data type for this is an array. We will be using the array construct from NumPy, for which a longer tutorial is given at
`http://www.scipy.org/Tentative_NumPy_Tutorial`

Most of the following examples show two-dimensional arrays, but arrays can have different dimensions and generalize easily. An array can be created with explicitly specified values using the `array()` method

```
a = array( [[1,2,3],[3,4,5]] )
```

although we are usually initializing an array with the results of other functions or read its values from data files. For example, we might want to generate an array with uniformly distributed values,

```
b = random((2,3))
```

or an array of zeros

```
a = zeros((2,3))
```

or create a one-dimensional array of numbers and reshaping it to the desired form of an array like

```
a = arange(1,7).reshape((2,3))
```

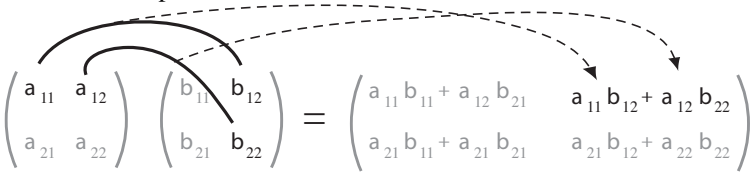
The reshape function is itself quite useful. Finally, it is quite common to set array elements from data in a file. This can be done with the function

```
a=loadtxt('data1')
```

where the file `data1` contains floating points with whitespace between columns.
Usually operators act on arrays element-wise, such as

```
a*b
a+b
exp(a)
```

and there is a large list of useful functions of which the ones we will be using are listed in Table **??**. In addition to the basic use of arrays, we will also be using NumPy arrays to do linear algebra. Although there is a matrix class in NumPy, we can still use the array class which has much more richer in associated functions. The mathematical difference between an array and a matrix is quite suttle. As already mentioned, an **array** is a collection of data with the same data type, as opposed to the more general python **list** (or list-of-lists) that also supports collections of different types. A **matrix** is usually an array of numbers and has additional operations defined on them. This includes a dot product like



This can be written in Python as

```
dot(a,b)
```

There is also an auger product, `outer(a,b)` and corresponding matrix divisions. The definition of the dot product stems from the way to represent equations systems. A linear equation systems such as

$$a_{1,1}x_1 + a_{1,2}x_2 = 1 \tag{1.1}$$
$$a_{2,1}x_1 + a_{2,2}x_2 = 1 \tag{1.2}$$

The (2×1) matrix $\mathbf{x}$, also called a vector, is here an unknown, and the matrix $\mathbf{a}$ describes the constants $a_{1,1}$ etc. This equation system can be solved by multiplying the inverse of matrix $\mathbf{x}^{-1}$ from the left, which gives us

$$\mathbf{ax} = \mathbf{1} \quad \rightarrow \quad \mathbf{x} = \mathbf{a}^{-1} \tag{1.3}$$

Thus, the solution to the above linear equation system can be calculated with Python as

```
a.inv()
```

This shows the peer of high level scientific programming languages like Python. There is no need to write a lengthy program that solves the linear equation system. Such things have been done over and over again, and there is no need to repeat this work. Indeed, the underlying implementations are often more efficient than writing your own solver. We will focus our energy instead on the higher level learning algorithm.

Finally, we might want to access specific values from the array or a range of values. A specific value at position (i, j) is accessed with

```
a[i,j]
```

The first index specifies the row of the array, and the second the column. A range of values can be specified like

```
a[istart:istop:istep,jstart:jstop:jstep]
```

If we omit the start or stopping values the beginning or end of this dimension is assumed. Thus, we could iterate through a vector v in reverse like

```
v[::-1]
```

It is always advisable to use operations on whole arrays rather than writing loop that iterate over all indices.

1.3.2 Loops, control flow, and functions

The rest of the programming constructs are quite similar to other programming languages. In order to write useful programs we need only a few more programming constructs, namely loops, control flow and functions.

Loops are specified with either

```
while {logical expression}:
    block of code
```

or

```
for i in a:
    block of code
```

Important is hereby that the block of code that is inside the loop is indented. There is no need for an end statement as in other programming languages, but the horizontal

position of the code become important. The `for` iteration is really specifying to loop over all elements in the list or array `a`. This is usually a specific set of numbers, such as specified by a vector

```
arange(start,end,step}
```

or by using `nstep` equidistant steps between values `v1` and `v2` as in

```
linspace(v1,v2,nstep}
```

Conditional blocks are similarly specified with indented code like

```
if {logical expression}
    block of code
elif {logical expression}
    block of code
else
    block of code
```

A final construct to organize code is to build functions in which we can encapsulate certain operations. A Python function has physically to proceed any code that uses it, so it is natural to define these at the beginning of a program.

1.3.3 Scipy

Scipy is a collection of methods that are common in science. This includes methods for integration, optimization, signal processing, linear algebra, file IO, statistics and some image processing. It also includes common constants ssh as π. Most basic SciPy methods are included in PyLab though not all.

We will later encounter several methods from this library, but as an example we show here numerical integration. In particular, let us consider a basic system of ordinary differential equations that is so central to science. This has generally the form of

$$\frac{\partial \mathbf{x}}{\partial t} = f(\mathbf{x}, t) \tag{1.4}$$

where $\mathbf{x}$ is a vector. For example, the function $f = 1/t^2$ can be integrated like

```
from pylab import *
from scipy.integrate import odeint

x0 = 0

def func(x, t):
    return [1/t**2]

t = arange(1,4.0, 0.01)
y = odeint(func, x0, t)
```

Another important area that is central to many learning algorithms is optimization, and Scipy provides some advanced support for this. We will discuss this later. See `http://docs.scipy.org/doc/scipy/reference` for a list of tutorials of the different areas supported by Scipy.

1.3.4 Plotting

Plots are very useful to visualize scientific results. The python matplotlib library provided support for scientific plotting similar to Matlab plotting routines. The basic tools for 2-dimensional plotting support is given in the `pyplot` submodule which is already part of PyLab. It can also be imported separately with

```
from matplotlib.pyplot import *
```

A basic line plot can be done like

```
x = linspace(0,2*pi,100)
plot(x,sin(x))
plot(x,cos(x),'r--')
```

It is also important to give axis some labels as in

```
xlabel('text for x axis')
ylabel('text for y axis')
```

Finally, to create an image window with the plot that will be shown on screen we need to use the command

```
show()
```

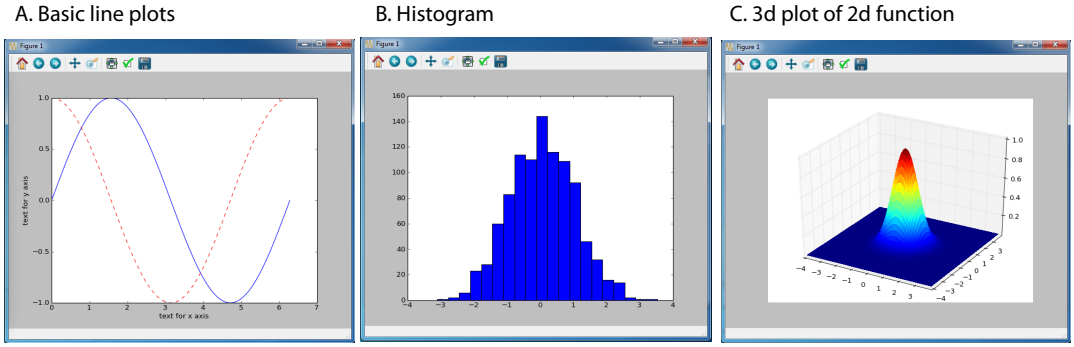


Fig. 1.3 Graphics produced with the matplotlib library.

We will also need to plot histograms which are bar grabs that found how many times an item occurs within one bin. for example. let's great an array with 1000 samples of a Gaussian random variable and plot the corresponding histogram of this sample with 20 bins,

```
x=randn(1000)
hist(x,20)
```

The function also returns the values of the bins and the bins can also be set in different ways if necessary.

Finally, we need to plot 3-dimensional graphs, specifically surface plots of 2-dimensional functions. Support for 3d plotting is provided by the `mplot3d` module of the **matplotlib** library (see `http://matplotlib.sourceforge.net/mpl_toolkits/mplot3d/tutorial.html` for a more detailed tutorial). This is currently not part of PyLab and has therefore to imported separately.

```
from pylab import *
```

```
from mpl_toolkits.mplot3d import Axes3D

X,Y = mgrid[-4:4:.1, -4:4:.1]
Z = exp(-(X**2 + Y**2))

ax = gca(projection='3d')
ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap=cm.jet, linewidth=0, antialiased=False)
show()
```

The `mgrid()` generates two 2-dimensional arrays that spans all the combination of the specified values in each dimension. For three dimensional plotting we need to call the current axis (gca = get current axis) and set the '3d' parameter. We have chosen parameters for the surface plot so that no lines are shown and we use a popular colour scheme.

Exercise

- Write a Python program to print 'Hello World!'.
- Write a Python program that reads and inverts a string.
- Write a Python program to plot a multivariate Gaussian with a covariance matrix that has off-diagonal elements which are half the value of the on-diagonal elements.
- Write a Python program with a function that takes two arrays, a and b, and calculates the matrix $a * b'^2 + b$.

1.4 Basic calculus and minimization

1.4.1 Differences and sums

We are often interested how a variable change with time. Let us consider the quantity $x(t)$ where we indicated that this quantity depends on time. The change of this variable from time t to time $t' = t + \Delta t$ is then

$$\Delta x = x(t + \Delta t) - x(t). \tag{1.5}$$

The quantity Δt is the finite difference in time. For a continuously changing quantity we could also think about the instantaneous change value, $\mathrm{d}x$, by considering an infinitesimally small time step. Formally,

$$\mathrm{d}x = \lim_{\Delta t \to 0} \Delta x = \lim_{\Delta t \to 0} (x(t + \Delta t) - x(t)). \tag{1.6}$$

The infinitesimally small time step is often written as $\mathrm{d}t$. Calculating with such infinitesimal quantities is covered in the mathematical discipline of calculus, but on the computer we have always finite differences and we need to consider very small time steps to approximate continuous formulation. With discrete time steps, differential become differences and integrals become sums

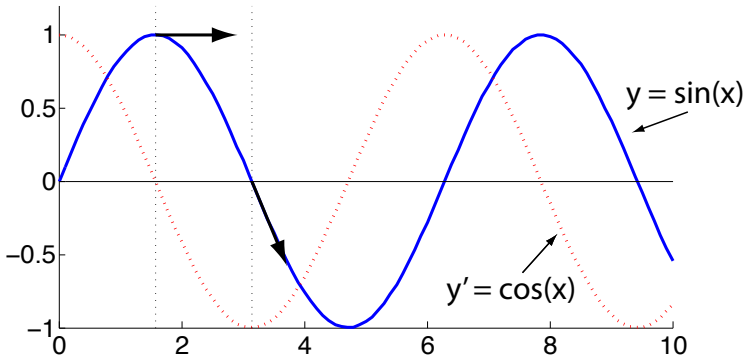
$$\mathrm{d}x \to \Delta x \tag{1.7}$$

$$\int \mathrm{d}x \rightarrow \Delta x \sum \tag{1.8}$$

Note the factor of Δx in front of the summation in the last equation. It is easy to forget this factor when replacing integrals with sums.

1.4.2 Derivatives

The derivative of a quantity y that depends on x is the slope of the function $y(x)$. This derivative can be defined as the limiting process equation 1.6 and is commonly written as $\frac{\mathrm{d}y}{\mathrm{d}x}$ or as y'.



It is useful to know some derivatives of basic functions.

$$y = e^x \rightarrow y' = e^x \tag{1.9}$$

$$y = sin(x) \rightarrow y' = cos(x) \tag{1.10}$$

$$y = x^n \rightarrow y' = nx^{n-1} \tag{1.11}$$

$$y = log(x) \rightarrow \frac{1}{x} \tag{1.12}$$

as well as the chain rule

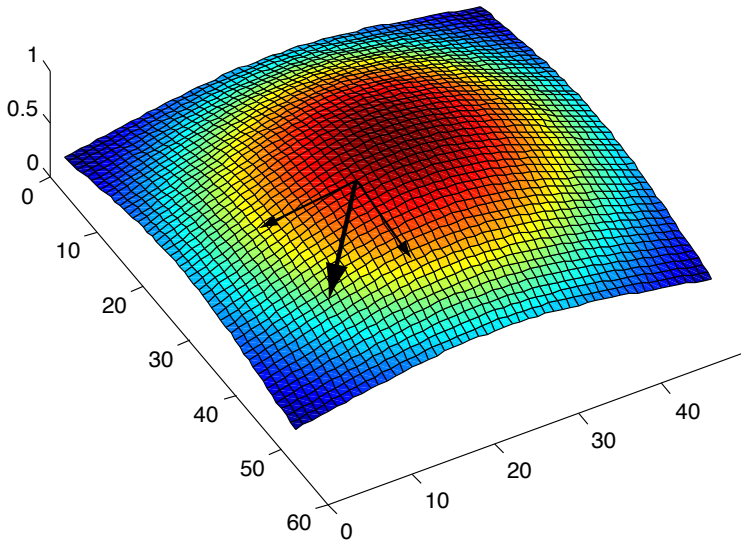
$$y = f(x) \rightarrow y' = \frac{\mathrm{d}y}{\mathrm{d}x} = \frac{\mathrm{d}y}{\mathrm{d}f}\frac{\mathrm{d}f}{\mathrm{d}x}. \tag{1.13}$$

It is also useful to point out that the slope of a function is zero at an extremum (minimum or maximum). A minimum of a function $f(x)$ is hence characterized by $\frac{\mathrm{d}f}{\mathrm{d}x} = 0$, and $\frac{\mathrm{d}^2 f}{\mathrm{d}x^2} > 0$. The second requirement, that the change of the slope is positive, specifies that this is a minimum rather than a maximum at which point this change in slope would be negative.

1.4.3 Partial derivative and gradients

A function that depends on more than one variable is a higher dimensional function. An example is the two-dimensional function $z(x, y)$. The slope of the function in the direction x (keeping y constant) is defined as $\frac{\partial z}{\partial x}$ and in the direction of y (keeping x constant) as $\frac{\partial z}{\partial x}$. The **gradient** is the vector that point in the direction of the maximal slope and has a length proportional t the slope,

$$\nabla z = \begin{pmatrix} \frac{\partial z}{\partial x} \\ \frac{\partial z}{\partial y} \end{pmatrix}. \tag{1.14}$$



Exercise

Calculate the gradient of a two dimensional Gaussian and find the minimum.

1.5 Outline of this course

While the main scientific content of this course is machine learning, we will explore this fascinating area with the help of robotics. Robots provide us with data and uncertain environments that we are trying to model. In order to get ready to use these tools we will start in the next chapter, Chapter 2, with a basic introduction to some more classical robotics, specifically some nomenclature in this area and some basic control theory. This chapter also shows how to use the Lego NXT robot kits within the python programming environment.

In Chapter 3 we will review some probability theory as this is a great language for describing uncertain sensors and environment. We will also introduce motion and sensor models that will be central to enable navigation in later sections.

Chapters 4-6 introduces the basic concepts of modern learning theories, separately for supervised, unsupervised and reinforcement learning. The aim of these sections is to outline the essence of all these areas in machine learning and specifically to see their relation and differences. The concepts will be deepened in later robotics applications.

In Chapter 7 we take a little detour to introduce some computer vision as such sensors provide valuable information in robotics applications and this is also an area where machine learning had major impact.

Chapter 8 tackles an important area of mobile robotics, that of localization and mapping.

Chapter 9 talks about planing, including classical planing through graph search as well as planing using dynamic programming and reinforcement learning.

Finally, in Chapter 10, we discuss some non-traditional robotics that is inspired by neuroscience and general theories of the mind.

2 Robotics

2.1 Sensing, acting and control

In this course, we will demonstrate and explore algorithms and associated problems in machine learning with the help of computer simulations and robots. Computer simulations are a great way to experiment with many of the ideas, but robotics implementations in the physical world have the advantage to show more clearly the challenges in real applications. Our emphasis in this book is to use general machine learning techniques to solve Robotics tasks even though more direct engineering solutions might be possible. While this is not always the way robots are controlled today, machine learning methods are becoming increasingly important in robotics.

The dream of having machines that can act more autonomously for human benefits is quite old. The word 'robot' is sometimes credited to the Czech writer Karel Čapek (1921) or to Isaac Asimov (1941), and the Unimate (1961) is often referred to as the first industrial robot. Robots are now invaluable in manufacturing. There is also much research to make robots more autonomous and to make those machines more robust and able to work in hostile environments. Robotics has many subdisciplines, including mechanical and electrical engineering, computer or machine vision, and even behavioral studies have become prominent in this field. A good definition of a robot is:

> "Robotics is the science of perceiving and manipulating the physical world through computer-controlled devices"[1].

That is, we use the word **robot** or **agent** to describe a system which interacts actively with the environment through **sensors** and **actuators**. An agent can be implemented in software or hardware, and the 'brain' of an agent or robot is commonly called the **controller**. The essential part in the above definition of a robot is that it must act in the physical world. For this it needs sensors to sense the state of the world, actuators that can be used to change the state of the environment. **Mobile robots** are a good example of this and we will be using two prototypes robots for most of our examples in this course, a simple robot arm with a web cam and a tribot with an ultrasonic sensor, that we will build shortly. In contrast, a vision system that uses a digital camera to recognize objects is not a robot in our definition. Software agents such as crawlers surfing the net are on the borderline. They need sensors, such as the ability to read content of web pages, and act in the physical world as they have to visit web pages physically located at different servers. If they gather information and use this to actively influence the physical world than this could also be considered a robot in the strict sense used here.

Robots are intended to make useful actions to achieve some goals, so robotics is all about finding and safely executing those appropriate actions. This area is generally the

[1] Probabilistic Robotics, Sebastian Thrun, Wolfram Burgard, and Dieter Fox, MIT press 2006

subject of **control theory**. A functioning robotics system typically needs controllers on many different levels, controlling the low level functions such as the proper rotation of motors up to ensuring that high level tasks are accomplished. A main question in designing a robotics system is indeed how to combine the different levels of control. A very successful approach has been the **subsumption architecture**, which is illustrated in Figure 2.1. Such systems are typically build bottom-up in that more complex functions use lower-level functions to achieve more complex tasks. The higher level modules can chose to use the lower level function or can inhibit them if necessary.

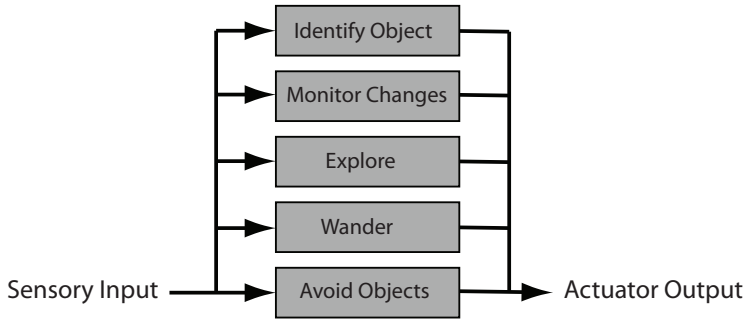


Fig. 2.1 An example of an subsumption architecture. From Maja J. Matari c, *The Robotocs Primer, MIT Press 2007*

More generally, it is useful to think of two opposing approaches to robotics control, the **deliberative approach** and the **reactive approach**, though a combination is commonly useful in practice. In the deliberative approach we gather all available information and plan action carefully based on all available actions. Such a **planning process** usually takes time and is based on searching through all available alternatives. The advantage of such systems is that they usually provide superior actions, but the search for such actions can be time consuming and might thus not be applicable to all applications. Also, deliberative systems require a large amount of knowledge about the environment that might not be available in certain applications. Reactive systems have a more direct approach of translating sensory information in actions. For example, a typical rule in such systems might be to turn the robot when a proximity sensor senses some objects in its path. To generate more complex behavior, such reactive systems typically build response systems by combining lower level control systems with higher level functions.

Robots act in the physical world through actuators based on sensory information. Actuators are mainly motors to move the robot around (locomotion) or to move limbs to crasp objects (manipulation). Motors for continuous rotations are typically DC (direct current) motors, but in robotics we often need to move a limb to a specific location. Motors that can turn to a specific position are called **servo motors**. Such motors are based on gears and position sensors together with some electronics to control the desired rotation angle. The motor of the LEGO NXT robotics kit is shown in Figure 2.2. This actuators is a stepping motors that can be told to run for a specific duration, a specific number or angle of rotations, and with various powers that influence the

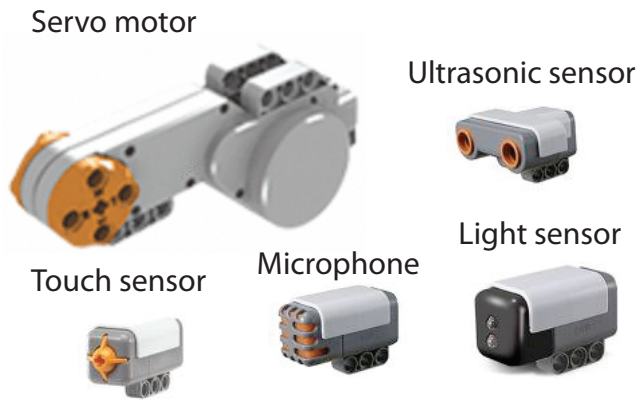


Fig. 2.2 The actuator and sensors of the basic Lego Mindstorm NXT robotics set.

rotation speed.

Sensors come in many varieties. Table 2.1 gives some examples of what kind of sensing technology is often used to sense (measure) certain physical properties. The basic Lego actuator and sensors used in this course are shown in Figure 2.2. The motor itself can be used as a rotation sensor as it provides feedback when it is externally rotated. The ultrasonic sensor sends out an high-frequency tone who's reflection is then sensed by its microphone to estimate distances to surfaces. The light sensor can detect light with different wavelength and can hence be used to detect colour and to some extend also short distance. The basic toolkit also includes a touch sensor and a microphone. We will also use a web camera for basic vision as outlined in the chapter on computer vision. Finally, for special projects we can use additional sensors such as the Kinect sensors from Microsoft, or special Lego sensors such as a GPS, an accelerometer, a giro, or a compass sensor (see Figure 2.3.

2.2 Building a driving a basic LEGO NXT robot

2.2.1 Arm and Tribot

We will actively use the LEGO Mindstorm robotics system in this course. This system is based on common LEGO building blocks that we use for two principle designed that we build below. The NXT LEGO robotics system includes a microprocessor in a unit called the **brick** which can be programmed and used to control the sensors and actuators. The brick is programmable with a visual programming language provided by Lego, and there exists a multitude of systems to program the brick with other common programming languages. We will be using the brick mainly to communicate with the motors and sensors while implementing the machine learning controllers on an external computer connected by either USB cable or wireless bluetooth.

We will be using two basic robot designs for the examples in this course. One is a **simple robot arm** that is made out of two motors with legs to mount it to a surface and a pointer as shown in Fig.2.4 A. Our basic robot arm is constructed by attaching the

Table 2.1 Some sensors and the information they measure. From From Maja J. Matari c, *The Robotocs Primer, MIT Press 2007*

Physical Property	Sensing Technology
Contact	bump, switch
Distance	ultrasound, radar, infra red
Light level	photocells, cameras
Sound levels	microphones
Strain	strain gauges
Rotation	encoders, potentiometers
Acceleration	accelerometers, gyroscopes
Magnetism	compass
Smell	chemical sensors
Temperature	thermal, infra red
Inclination	inclinometers, gyroscopes
Pressure	pressure gauge
Altitude	altimeter

Fig. 2.3 Some additional sensors that we could use such as the Microsft Kinect and a GPS for the Lego NXT.

base of one motor, that we call Motor1, to the rotating part of a second motor, called Motor2, as shown in Fig.2.4A. We also attach a long pointer extension to Motor1 that will become useful in some later exercises. Finally, we add some legs that we can be taped to a table surface in order to stabilize Motor2 to a fixed position. The precise design is not crucial for most of the exercises as long as it can rotate freely both motors.

We will also use a basic terrestrial robot called the **tribot** shown in Fig.2.4B. The tribot used here is a slight modification of the standard tribot as decried in the LEGO NXT robotics kit. A detailed instruction for building the basic tribot is included in the LEGO kits, either in the instruction booklet or the included software package. It is not crucial that all the parts are the same. The principle idea behind this robot is to

have a base with two motors to propel the tribot. and several sensors attached to it. There is commonly a third passive wheel that is only used to stabilize the robot, and we included a way to lock this to a straight position to facilitate cleaner movements along a straight line. Some versions of LEGO kits have tracks that can be used in most of the exercises. The exact design is not critical and can be altered as seen fit.

A. Robotarm with attached dawing pen

B. Tribot with ultrasonic, touch and light sensor

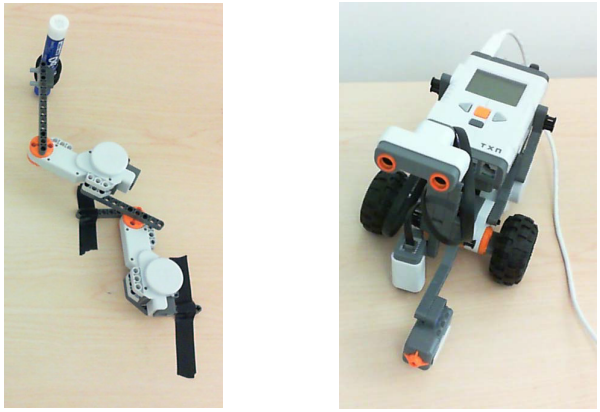


Fig. 2.4 (A) A robot arm made out of two motors, a pointer arm, and some support to tape it to a table surface. This version has also a pen attached to it. (B) Basic Lego Robot called tribot with the microprocessor, two motors, and three sensors, including a ultrasonic and touch sensor pointing forward and a light sensor pointing downwards.

2.2.2 NXT Python Software Environment

The 'brain' of our robots will be implemented on PCs and we will use a Python environment to implement our high-level controllers. Most examples are minimalistic to concentrate on the algorithmic ideas behind machine learning methods explored in this book. While there are more advanced Robotics environments with more elaborate frameworks such as ROS (Robot Operating System), we want to keep the overhead small by using only direct methods to communications with actuators and sensors. This section describes the Python environment an packages that we use in the following.

To program the NXT we will use the NXT-Python package that you need to download at `http://code.google.com/p/nxt-python`. To communicate with the Brick we need to use either the USB cable or use the build in bluetooth. For USB we need PyUSB and libusb-win32. Open the Lib-usb32 and add the filter for your brick connected through usb. The brick can then be linked in python as in the following example program:

```
from nxt.locator import *
b = find_one_brick()
b.play_tone(200,200)
```

For Bluetooth, copy the file `www.cs.dal.ca/~t/CSCI415512/serialsock.py` to

the python library (e.g. `"C:Python27/Lib/site-packages/nxt"`. To synchronize your brick with the bluetooth, go to the Control Panel in the option Hardware and Sound and Add Device. In the list of all recognized bluetooth devises find your NXT and right click to go to Properties. In the tab Services find which port it's synchronized, such as COM5. Sometimes it is connected to two ports, such as COM5 and COM6 for example. The brick can then be accessed in python as in the following example program:

```
from nxt.serialsock import *
s = SerialSock('COM5')    #or you can use  s = SerialSock('COM5','COM6')
b = s.connect()
b.play_tone(200,200)
```

Below is a first example that pulls information from sensors and runs the motors until the ultrasonic sensor detects an object within 15cm.

```
from nxt.locator import *
from nxt.sensor import *
from nxt.motor import *

b = find_one_brick()

t = Touch(b, PORT_1)
print "Touch: ",t.get_sample()

s = Sound(b, PORT_2)
print 'Sound:', s.get_sample()

l = Light(b, PORT_3)
print 'Light:', l.get_sample()

u = Ultrasonic(b, PORT_4)
print 'Ultrasonic:', u.get_sample()

##testing the motors

m_left = Motor(b, PORT_C)
m_right = Motor(b, PORT_B)
m_left.run(60)
m_right.run(60)

while u.get_sample()>15:
    i=0
m_left.brake()
m_right.brake()


print "Final position Left motor: ",m_left.get_tacho()
print "Final position Right motor: ", m_right.get_tacho()
m_left.reset_position(False)
```

Exercises

1. **Wall avoidance:** Write a program that let the tribot explore the environment without running into objets for the tribot. Use the ultrasonic sensor to detect an obstacle and turn randomly if necessary.
2. **Line following:**
   Writing a controller that uses readings from its light sensor to drive the tribot along a line. You can use electrical tape for the line to follow.

2.3 Pose and state space

To enable more advanced control we need to describe the possible states of the system. The physical state of an agent can be quite complicated. For example, the Lego components of the tribot can have different positions and can shift in operation; the battery will have different levels of charge, light conditions change, etc. Thus, description of the actual physical state would need a lot of variables, and such a description space would be very high dimensional. Working in high dimensional spaces is often a major computational challenge. In order to manage even a simple robot such as the robot arm or the tribot we need to consider abstractions. **Abstraction** is an important concept in science. To abstract means to simplify the system in a way that it can be described in as simple terms as possible to answer the specific questions under consideration. This philosophy is sometimes called the **principle of parsimony**, also known as **Occam's razor**. Basically, we want a model as simple as possible, while still capturing the main aspects of the system that the model should capture.

To illustrate this for navigation let us consider a robot that should navigate around some obstacles from point A to point B as shown in Fig.2.5A. The simplifications we make in the following is that we consider the movement in only a two-dimensional (2D) plane. The robot will have a position described by an x and y coordinate, and a heading direction described by an angle α. Note that we ignore for now the physical extension of the robot, that is, we consider it here as a point object. If this is a problem, we can add an extension parameter later. We also ignore many of the other physical parameters mentioned above. The current state of robot is hence described by three real numbers, $[x, y, \alpha]$. The specific realization of the variables describing the state of a robot is called its **pose** and the space of all possible values is called the **state space**. An obstacle, such as the one represented by the grey area in Fig.2.5A, represents state values that are not allowed by the system.

We have reduced the description of an robot navigation system from a very high physical space to a three-dimensional (3D) pose space by the process of abstraction. However, there are still an infinite number of possible states in the state space if the state variables are real numbers. We will now discuss some basic navigation algorithms where we will use search algorithms in the state space to find solutions, and an infinite space is then problematic. A common solution to this problem is to make further abstractions and to **discretize** the state space to allow the state variables to only have a finite number of states. An example is shown in Fig.2.5B, where we used a grid to represent the possible values of the robot positions. Such a discretization is very useful in practice. It is possible to make the discretization error increasingly small

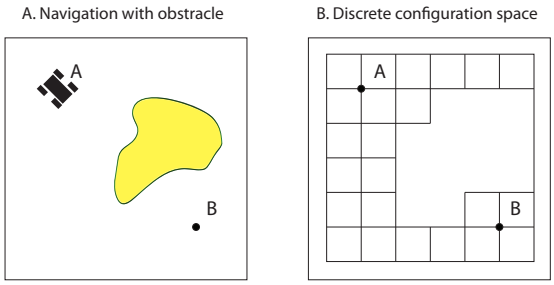


Fig. 2.5 (A) A physical space with obstacles in which an agent should navigate from point A to point B. (B) Descretized configuration space.

by decreasing the grid parameter, Δx, which describes the minimal distance between two states at the expense of increasing the state space. However, there might be a good tradeoff in specific applications. examples. We will use this grid discretization for planing a path through the state space later in this chapter.

Exercise: Robotarm State Space Visualizer

The goal of this exercise is to graph the state space in which a double jointed robot arm is able to move. Use the robot arm build from two NXT motors and secure it on the table so that you can move its pointing finger around by externally turning the motors. We will use the ability of measuring this movement in the motors to record the corresponding angles of the motors. Use a box, a coffee mug or some other items to create an obstacle for the arm to move around.

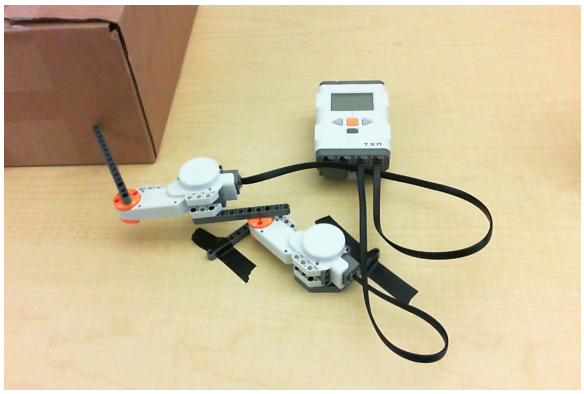


Fig. 2.6 (A) A simple robot arm with two joints that can point at an obstacle.

At the beginning of the execution of this program the tip of the pointer of the robot arm should touch an obstacle. You should then move the pointer along the obstacle to generate and display an **occupancy map**. An occupancy grid is an array with entries

of zeros for cells that are possible poses of a robot and entires of one for state spaces that are occupied by obstacles. Make sure that the movement and discretization is appropriate so that no gaps in state space exists that seem to open a path into the interior of the object. You can use the function `get_tacho()` in python to read the angles of each motor. You can record the trajectory given by the two angles and generate an occupancy grid from this by checking if the trajectory crossed a grid point in a discretized state space.

2.4 Basic controllers

2.4.1 Inverse plant dynamics

Control theory is a discipline in engineering that has been crucial to operate complex machines. Much of classical robotics is based on control theory. Controllers are the brains of robots which use sensory and motivational information to produce goal directed behavior. Our aim is to build smart controllers so that our robots are able to achieve goals in a robust way, particularly in an uncertain environment.

A control system is characterized by a **plant**, which is the object to be controlled such as a robot arm, and the **contoller**, the part of the system that must generate the appropriate motor commands to achieve the goals. A plant is the dynamical object that can be characterized by a **state vector**

$$\mathbf{z}^T(t) = (1, \mathbf{x}(t), \mathbf{x}'(t), \mathbf{x}''(t), ...). \tag{2.1}$$

$\mathbf{x}(t)$ are the basic coordinates (pose) such as the position and heading direction of the tribot or the angles of the robot arm. The state vector includes derivatives of first and higher order to describe dynamic properties such the momentum or acceleration of the plant. The state vector also includes a constant component.

The state of the plant is influenced by a **control command** $\mathbf{u}(t)$. A control command can be, for example, sending a specific current to motors. The effect of a control command $\mathbf{u}(t)$ when the plant is in state $\mathbf{z}(t)$ is described given by the **plant equation**

$$\mathbf{z}(t+1) = \mathbf{f}(\mathbf{z}(t), \mathbf{u}(t)), \tag{2.2}$$

In general we do not know the plant equation a priori, and an important part for machine learning is learning the plant function $\mathbf{f}$.

The most basic **control problem** is finding the appropriate commands to reach a **desired states** $\mathbf{z}^*(t)$. We assume for now that this desired state is a specific point in the state space, also called a **setpoint**. Such a control problem with a single desired state is called **reaching**. The control problem is called **tracking** if the desired state is changing. In an ideal situation we might be able to calculate the appropriate control commands. For example, if the plant is linear in the control commands, that is, if $\mathbf{f}$ has the form

$$\mathbf{z}(\mathbf{t}+\mathbf{1}) = \mathbf{f}(\mathbf{z}(\mathbf{t}), \mathbf{u}(t)), \tag{2.3}$$

and the plant function $\mathbf{f}$ has an inverse, then it is possible to calculate the command to reach the desired state as

$$\mathbf{u}^* = \mathbf{f}^{-1}(\mathbf{z})\mathbf{z}^*. \tag{2.4}$$

A block diagram of this strategy with a basic controller and the plant is shown in Figure 2.7.

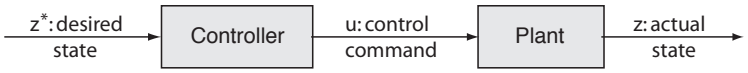


Fig. 2.7 The elements of a basic control system.

As an example, let us consider the tribot moving a specified distance from a starting point and that the system can be described by its position $x(t)$. Let $u = t$ be the motor command for the tribot to move forward for t seconds with a certain motor power. If the tribot moves a distance of d_1 in one second, then we expect the tribot to move a distance of $d_1 * t$ in t seconds. The plant equation for this tribot movement is hence

$$x(t) = x(0) + d_1 * t, \tag{2.5}$$

To find out the parameter d_1, or more precisely an estimate of the parameter that we can mark with a hat, $\hat{d}_1$, we can mark the initial location of the tribot, say x_0, and let the tribot run with a specific motor speed for 1 second. The parameter can then be determined from the end location x_1 by

$$\hat{d}_1 = (x_1 - x_0). \tag{2.6}$$

Note that we **learned** a parameter from measurement in the environment. This is at the heart of machine learning, and the formula above is our first learning algorithm.

Exercise

In this experiment we want the robot to move from a start position to a desired position which is exacty 60cm away from the start position. Ideally this could be achieved by letting the motor run by $t = 60/d_1$ seconds. Try it out. You need to estimate d_1 and to need to run the tribot for t seconds. This can be done in while loop with the `time()` function.

```
from nxt.locator import *
from nxt.sensor import *
from nxt.motor import *
from time import *

b = find_one_brick()
m_left = Motor(b, PORT_C)
m_right = Motor(b, PORT_B)
```

```
tend=10;
m_left.run(60)
m_right.run(60)
start = time()
while time()-start<tend:
    continue
m_left.brake()
m_right.brake()
```

2.4.2 Basic feedback control

In the last experiment we used the knowledge (or measurement) of a desired distance and the knowledge of the inverse kinematic to reach a target pose. While the tribot might come close to the desired state, a perfect match is not likely. There are several reasons for such failures. One is that our measurement of the dynamics might be not accurate enough. We also made the assumption that the rotations of the wheels are linear in time, but it might be that the wheels slow down after a while due to power loss or heating of the gears which alter physical properties, etc. Also, there is likely an initial time the motor needs to get to the constant speed. But most of all, disturbances in the environment can throw the robot off-track (such as a mean instructor). All these influences on the controlled object are indicated in the Fig.2.8 by a disturbance signal to the plant.

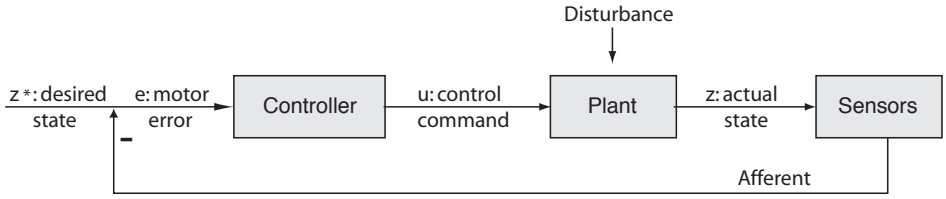


Fig. 2.8 The elements of a basic control system with negative feedback.

How can we compensate for those uncertainties? If we do not reach the desired state we can initiate a new movement to compensate for the discrepancy between the desired and actual state of the controlled object. For this we need a measurement of the new position (which can itself contain some error) and use this measurement to calculate the new desired distance to travel. We call the distance between the desired location x^* and the actual location x the **displacement error**

$$e(t) = x^*(t) - x(t) \tag{2.7}$$

This procedure can be iterated until the distance to the desired state is sufficiently small. Such a controller is called a **feedback controller** and is shown in Fig. 2.8. The desired state is the input to the system, and the controller uses the desired state to determine the appropriate control command. The controller can be viewed as an **inverse plant model** as it takes a state signal and produce the right command so that

the controlled object ends up in the desired state. The motor command causes a new state of the object that we have labelled 'actual state' in Fig. 2.8. The actual state is then measured by sensors and subtracted from the desired state. If the difference is zero, the system has reached the desired state. If it is different than zero than this difference is the new input to the control system to generate the correction move.

The negative feedback controller is amazingly successful in this example to drive the tribot to a certain distance as specified in the program. Also, the controller makes the task somewhat robust to a variety of disturbances. For example, take the robot with your hand and move it to another distance; the tribot will soon reach the desired state again. It is also interesting to change the proportionality constant d_1 in the plant equation to a larger values. The tribot will then overshoot the target distance and might there take some more iterations around the set point to reach the target, but the target will be reached as long as the proportionality constant is not too far off. Finally, take your hand and hold it between the tribot and the wall after the tribot reached the target point. The tribot will then move backwards and again forward if you remove the hand.

2.4.3 PID controler

The negative feedback controller does often work in minimizing the position error of a robotic systems. However, the movement is sometimes jerky and oscillate around the setpoint. This is in particular the case when the plant has considerable inertia or momentum which demands to not only take the current position but also the velocity into account. It might also be useful to keep some history of control commands to optimize the time it takes to get to the setpoint. Such additions are easily added to the proportional controller discussed above. Here we discuss briefly a very common feedback controller call **PID coontroller** that is a common ingredient in robotics systems for basic control.

As already stated, the basic idea of a PID controller is to not only use the current error between the desired state and estimated actual state, but to take some history and momentum into account. For example, when the correction in each state takes a long time, that is, if the sum of the errors is large, then we should make larger changes so that we reach the setpoint faster. In a continuos system, the sum over the last errors becomes and integral. Such a component in the controller should help to accelerate the reaching of the setpoint. However, such a component can also increase overshooting and leads to cycles which can even make the system unstable. To remedy this it is common to take the rate of change in the error into account. Such a term corresponds to the derivative in a continuous system. The name for a PID controller is actually the acronym for **P**roportional, **I**ntegral and **D**erivative, and a block diagram of this controller is illustrated in figure 2.9

The motor command generated by a PID controller is given by

$$u(t) = k_P e(t) + k_I \int e(t)dt + k_D \frac{\mathrm{d}e(t)}{\mathrm{d}t}, \tag{2.8}$$

where we have weighted each of the components with different constants. Numerically we can replace the derivative with the difference of consecutive error terms and the intergral as the sum of error terms. The major difficulty in applying this controller is finding appropriate choices of the constants. These are often determined by trial

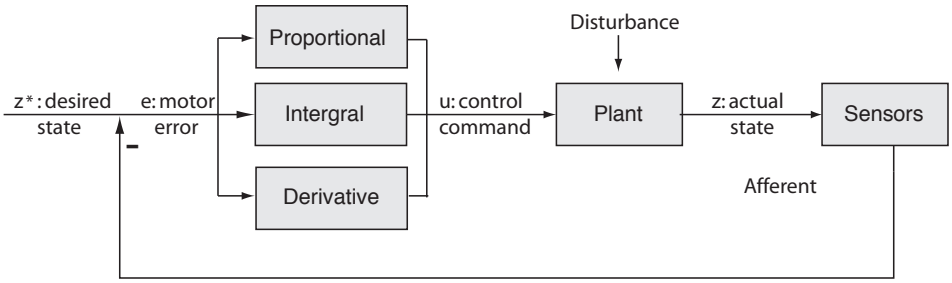


Fig. 2.9 The elements of a PID control system.

and error, and some recipes for choosing them can also be found. For engineering applications it is also important to prove that such controllers lead to stable behavior, which the PID controller does as long as the system is linear or not far from linear.

We have only scratched the surface of classical control theory at this point. In order to make controllers robust and applicable for practical applications, many other considerations have to be made. For example, we are often not only interested in minimizing the motor error but actually minimizing a cost function such as minimizing the time to reach a setpoint or to minimize the energy used to reach the setpoint. Corresponding methods are the subject of **optimal control** theory. While we will not follow classical optimal control theory here, we will come back to this topic in the context of reinforcement learning in the fourth part of the book. Another major problem for many applications is that the plant dynamic can change over time and has to be estimated from data. This is the subject of **adaptive control** where machine learning will become essential. An adaptive controller also uses feedback from the environment but will use this feedback to modify the behaviour of the controller. This is illustrated in Fig.2.10. Adaptive control is a main focus of this book.

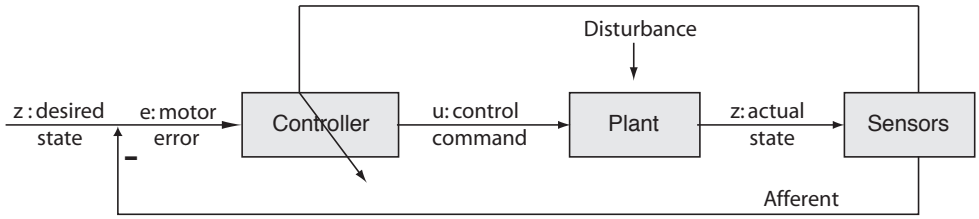


Fig. 2.10 Adaptive control systems incorporate models for corrective adjustments in the system indicated by the arrows going through these components. These models are trained with sensory feedback or other forms of external supervision.

The control theory outlined so far has several other drawbacks in robotics environments. In particular, it treats the feedback signal as the supreme knowledge not taking into consideration that it can be wrong. For example, in the exercise above we measured the distance to a wall to drive the tribot to a particular distance. If we put our

hand in between the wall and the tribot, the controller would treat our hand as the wall and would try to adjust the position accordingly. A smarter controller might ask if this is consistent with previous measurements and its movements it made lately. A smarter controller could therefore benefit from internal models and an acknowledgement, and corresponding probabilistic treatment, that the sensor information is not always reliable. Even more, a smart controller should be able to learn from the environment how to judge certain information. This will be our main strategy to follow in this book.

Exercise: Wall following

Mount the ultrasonic sensor to the tribot so that it points perpendicular to the driving direction. In this exercise you should write a PID controller so that the tribot can follow a wall in a predefined distance.

3 Probability theory and sensor/motion models

As already mentioned, a major milestone for the modern approach to machine learning is to acknowledge our limited knowledge about the world and the unreliability of sensors and actuators. It is then only natural to consider quantities in our approaches as **random variables**. While a regular variable, once set, has only one specific value, a random variable will have different values every time we 'look' at it (draw an example from the distributions). Just think about a light sensor. We might think that an ideal light sensor will give us only one reading while holding it to a specific surface, but since the peripheral light conditions change, the characteristics of the internal electronic might change due to changing temperatures, or since we move the sensor unintentionally away from the surface, it is more than likely that we get different readings over time. Consequently, variables that have to be estimated from sensors, such as the pose of a robot, are fundamentally random variables.

A common misconception about randomness is that one can not predict anything for random variables. But even random variables have usually values that are more likely than others, and, while we might not be able to predict a specific value when drawing a random number, it is possible to say something about how often a certain number will appear when drawing many examples. We might even be able to state how confident we are that a specific number occurs, or, in other words, how uncertain a specific value might be or how it might vary when drawing several examples. The complete knowledge of a random variable, that is, how likely each value is for a random variable x, is captured by the **probability density function** $pdf(x)$. We discuss some specific examples of pdfs below. In these examples we assume that we know the pdf, but in may practical applications we must estimate this function. Indeed, estimation of pdfs is at the heart if not the central tasks of machine learning. If we would know the 'world pdf', the probability function of all possible events in the world, then we could predict as much as possible in this world.

Most of the systems discussed in this course are **stochastic models** to capture the uncertainties in the world. Stochastic models are models with random variables, and it is therefore useful to remind ourselves about the properties of such variables. This chapter is a refresher on concepts in probability theory. Note that we are mainly interested in the language of probability theory rather than statistics, which is more concerned with hypothesis testing and related procedures.

3.1 Random numbers and their probability (density) function

Probability theory is the theory of **random numbers**. We denoted such numbers by capital letters to distinguish them from regular numbers written in lower case. A random variable, X, is a quantity that can have different values each time the variable is inspected, such as in measurements in experiments. This is fundamentally different to a regular variable, x, which does not change its value once it is assigned. A random number is thus a new mathematical concept, not included in the regular mathematics of numbers. A specific value of a random number is still meaningful as it might influence specific processes in a deterministic way. However, since a random number can change every time it is inspected, it is also useful to describe more general properties when drawing examples many times. The frequency with which numbers can occur is then the most useful quantity to take into account. This frequency is captured in the 'frequentist' interpretation of random numbers by the mathematical construct of a **probability**. A slightly different interpretation of a random numbers is that it describes the uncertainty that comes with each drawing of a random variable. This 'Bayesian' interpretation is useful as we would then be comfortable applying such constructs even to events that we can not repeat easily. Most of the time these interpretations are complementary and there seems little advantage to delve to much into a philosophical debate.

We can formalize the idea of expressing probabilities of drawing specific specific values for random variable with some compact notations. We speak of a **discrete random variable** in the case of discrete numbers for the possible values of a random number. A **continuous random variable** is a random variable that has possible values in a continuous set of numbers. There is, in principle, not much difference between these two kinds of random variables, except that the mathematical formulation has to be slightly different to be mathematically correct. For example, the **probability (mass) function**,

$$P_X(x) = P(X = x) \tag{3.1}$$

describes the frequency with which each possible value x of a discrete variable X occurs. Note that x is a regular variable, not a random variable. The value of $P_X(x)$ gives the fraction of the times we get a value x for the random variable X if we draw many examples of the random variable.[2] From this definition it follows that the frequency of having any of the possible values is equal to one, which is the normalization condition

$$\sum_x P_X(x) = 1. \tag{3.2}$$

In the case of continuous random numbers we have an infinite number of possible values x so that the fraction for each number becomes infinitesimally small. It is then appropriate to write the probability distribution function as $P_X(x) = p_X(x)\mathrm{d}x$, where $p_X(x)$ is the **probability density function** (pdf). The sum in eqn 3.2 then becomes an integral, and normalization condition for a continuous random variable is

$$\int_x p_X(x)\mathrm{d}x = 1. \tag{3.3}$$

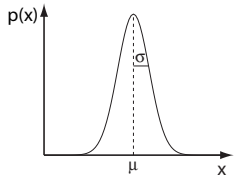
[2] Probabilities are sometimes written as a percentage, but we will stick to the fractional notation.

We will formulate the rest of this section in terms of continuous random variables. The corresponding formulas for discrete random variables can easily be deduced by replacing the integrals over the pdf with sums over the probability function. It is also possible to use a continuous formulation of discrete random variables with the mathematical construct of a δ-function. Thus, the differences between continuous or discrete random variables are mainly technical and will hopefully not distract from the general ideas.

Two important basic examples of a discrete and a continuos random variable is a Bernoulli (binary) random variable and a Gaussian or Normal distributed random variable, respectively. A Bernoulli random variable is a variable from an experiment that has two possible outcomes: success with probability p; or failure, with probability $(1-p)$.

Probability function:
$P(\text{success}) = p; P(\text{failure}) = 1 - p$
mean: p
variance: $p(1-p)$

The Normal or Gaussian distribution decries a continuous random variable with a single bell shaped peak in the distribution as shown below. The pdf depends on two parameters, the mean μ and the standard deviation σ. The importance of the normal distribution stems from the central limit theorem outlined below. This theorem captures an interesting fact about sums of random variables, namely that the sum of many random variables is Gaussian distributed. Formally, this is only correctly true if the random variables are independent and drawn from the same (but arbitrary) distribution, and also that an infinite number of such variables is considered. But the importance in practice is that even small sums of random variables with different underlying pdfs have often a distribution that is well approximated by a Gaussian.



Probability density function:
$p(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-(x-\mu)^2}{2\sigma^2}}$
mean: μ
variance: σ^2

3.2 Density functions of multiple random variables

So far, we have discussed mainly probability (density) functions of single random variables. As mentioned before, we use random variables to describe data such as sensor readings in robots. Of course, we often have then more than one sensor and also other quantities that we describe by random variables at the same time. Thus, in many applications we consider multiple random variables. The quantities described by the random variables might be independent, but in many cases they are also related. Indeed, we will later talk about how to describe various types of relations. Thus, in order to talk about situations with multiple random variables, or multivariate statistics, it is useful to know basic rules. We start by illustrating these basic multivariate rules

with two random variables since the generalization from there is usually quite obvious. But we will also talk about the generalization to more than two variables at the end of this section.

An example of a multivariate density function over several random variables, $x_1, ..., x_n$ is the multivariate Gaussian (or Normal) distribution,

$$p(x_1, ..., x_n) = p(\mathbf{x}) = \frac{1}{(\sqrt{(2\pi)})^n \sqrt{(\det(\mathbf{\Sigma})}} \exp(-\frac{1}{2}(\mathbf{x}-\mu)^T \mathbf{\Sigma}^{-1}(\mathbf{x}-\mu)). \quad (3.4)$$

This is a straight forward generalization of the one-dimensional Gaussian distribution mentioned before where the mean is now a vector, μ and the variance generalizes to a covariance matrix, $\mathbf{\Sigma} = [\mathrm{Cov}[X_i, X_j]]_{i=1,2,...,k;j=1,2,...,k}$ which must be symmetric and positive semi-definit. An example with mean $\mu = (1\ 2)^T$ and covariance $\Sigma = (1\ 0.5; 0.5\ 1)$ is shown in Fig,3.1.

3.3 Basic definitions

We have seen that probability theory is quite handy to model data, and probability theory also considers multiple random variables. The total knowledge about the co-occurrence of specific values for two random variables X and Y is captured by the

$$\textbf{joined distribution:} \quad p(x, y) = p(X = x, Y = y). \quad (3.5)$$

This is a two dimensional functions. The two dimensions refers here to the number of variables, although a plot of this function would be a three dimensional plot. An example is shown in Fig.3.2. All the information we can have about a stochastic system is encapsulated in the joined pdf. The slice of this function, given the value of one variable, say y, is the

$$\textbf{conditional distribution}: \quad p(x|y) = p(X = x|Y = y). \quad (3.6)$$

A conditional pdf is also illustrated in Fig.3.2 If we sum over all realizations of y we get the

$$\textbf{marginal distribution}: \quad p(x) = \int p(x, y)dy, \quad (3.7)$$

which is sometimes called the **sum rule** or **marginalization**.

If we know some functional form of the density function or have a parameterized hypothesis of this function, than we can use common statistical methods, such as maximum likelihood estimation, to estimate the parameters as in the one dimensional cases. If we do not have a parameterized hypothesis we need to use other methods, such as treating the problem as discrete and building histograms, to describe the density function of the system. Note that parameter-free estimation is more challenging with increasing dimensions. Considering a simple histogram method to estimate the joined density function where we discretize the space along every dimension into n bins. This leads to n^2 bins for a two-dimensional histogram, and n^d for a d-dimensional problem. This exponential scaling is a major challenge in practice since we need also considerable data in each bin to sufficiently estimate the probability of each bin.

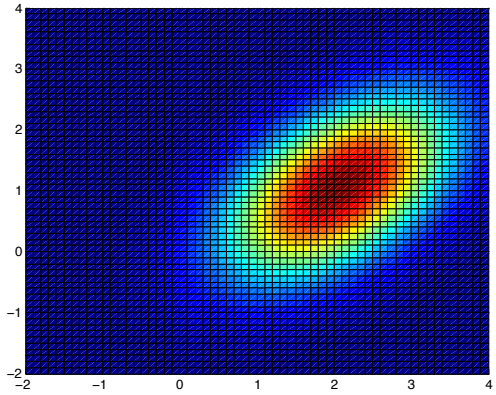


Fig. 3.1 Multivariate Gaussian with mean $\mu = (1\ 2)^T$ and covariance $\Sigma = (1\ 0.5; 0.5\ 1)$.

As mentioned before, if we know the joined distribution of some random variables we can make the most predictions of these variables. However, in practice we have often to estimate these functions, and we can often only estimate conditional density functions. A very useful rule to know is therefore how a joined distribution can be decompose into the product of a conditional and a marginal distribution,

$$p(x, y) = p(x|y)p(y) = p(y|x)p(x), \tag{3.8}$$

which is sometimes called the **product rule**. Note the two different ways in which we can decompose the joined distribution. This is easily generalized to n random variables by the **chain rule**

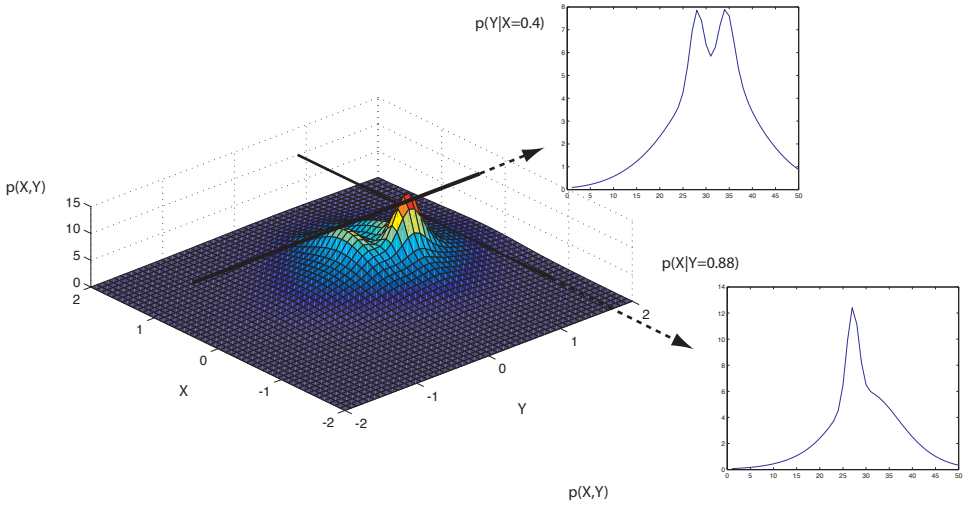


Fig. 3.2 Example of a two-dimensional probability density function (pdf) and some examples of conditional pdfs.

$$p(x_1, x_2, ..., x_n) = p(x_n|x_1, ...x_{n-1})p(x_1, ..., x_{n-1}) \tag{3.9}$$
$$= p(x_n|x_1, ..., x_{n-1}) * ... * p(x_2|x_1) * p(x_1) \tag{3.10}$$
$$= \Pi_{i=1}^{n} p(x_i|x_{i-1}, ...x_1) \tag{3.11}$$

but note that there are also different decompositions possible. We will learn more about this and useful graphical representations in Chapter **??**.

Estimations of processes are greatly simplified when random variables are independent. A random variable X is independent of Y if

$$p(x|y) = p(x). \tag{3.12}$$

Using the chain rule eq.3.8, we can write this also as

$$p(x, y) = p(x)p(y), \tag{3.13}$$

that is, the joined distribution of two independent random variables is the product of their marginal distributions. Similar, we can also define conditional independence. For example, two random variables X and Y are conditionally independent of random variable Z if

$$p(x, y|z) = p(x|z)p(y|z). \tag{3.14}$$

Note that total independence does not imply conditionally independence and visa versa, although this might hold true for some specific examples.

3.4 How to combine prior knowledge with new evidence: Bayes rule

One of the most common tasks we will encounter in the following is the integration of prior knowledge with new evidence. For example, we could have an estimate of

the location of an agent and get new (noisy) sensory data that adds some suggestions for different locations. A similar task is the fusion of data from different sensors. The general question we have to solve is how to weight the different evidence in light of the reliability of this information. Solving this problem is easy in a probabilistic framework and is one of the main reasons that so much progress has been made in probabilistic robotics.

How prior knowledge should be combined with prior knowledge is an important question. Luckily, basically already know how to do it best in a probabilistic sense. Namely, if we divide this chain rule eq. 3.8 by $p(x)$, which is possible as long as $p(x) > 0$, we get the identity

$$p(x|y) = \frac{p(y|x)p(x)}{p(y)}, \tag{3.15}$$

which is called **Bayes theorem**. This theorem is important because it tells us how to combine a **prior** knowledge, such as the expected distribution over a random variable such as the state of a system, $p(x)$, with some evidence called the likelihood function $p(y|x)$, for example by measuring some sensors reading y when controlling the state, to get the **posterior** distribution, $p(y|x)$ from which the new estimation of state can be derived. The marginal distribution $p(y)$, which does not depend on the state X, is the proper normalization so that the left-hand side is again a probability. Estimating the denominator $p(y)$ is often the most labor intensive part of applying Bayes theorem. Indeed, this is often achieve by substituting the marginalization rule

$$p(x|y) = \frac{p(y|x)p(x)}{\int p(y|x)p(x)dx}. \tag{3.16}$$

The integral becomes a sum for discrete random variables that can sometimes be evaluated explicitly. However, for more elaborate real world application this also points to an explicit limitation of an exact Bayesian evaluation and this would require knowledge of all possible states which is usually a huge if not infinite number. Learning sensible approximations of this term is an interesting research area.

3.5 Python support for statistics

Random numbers and statistics are a common tool in science so that support in this area is not surprising. We are mainly concerned here with using random numbers and probability theory rather than statistical method for hypothesis testing. Here we only discuss producing the some basic random numbers.

We will be using the support through the `numpy.random` package of Numpy. This package includes a large number of random number generators for different random numbers. For example, the function

```
rand()
```

generates a random number between 0 and 1. A normal distributed random number can be generated with

```
randn()
```

An 10×2 array of normal distributed numbers can be generated with

`randn(10,2)`

Three integer random numbers between 1 and 2 (inclusive) can be generated with

`print random_integers(1,2,3)`

These are likely the most used random numbers we need in this class, but there is also a large number of random numbers from other distributions available. In the exercise we will generate random numbers from arbitrary pdfs.

Exercises

1. Write a plotting program that plots a Gaussian, a uniform, and the Chi-square distribution (probability density function). Don't forget to provide units on the axis.
2. Plot a histogram of random numbers from the Trappenberg distribution

$$p(x) = \begin{cases} a_n || \sin(x) || & \text{for } 0 < x < n\pi/2 \\ 0 & \text{otherwise} \end{cases} \tag{3.17}$$

   for $n = 5$. What is the mean, variance, and skewness of this distribution?
3. Explain if the random variables X and Y are independent if their marginal distribution is $p(x) = x + 3log(x)$ and $p(y) = 3ylog(y)$ and the joined distributions is $p(x, y) = xylog(x) + 3ylog(xy)$.
4. (From Thrun, Burgard and Fox, Probabilistic Robotics) A robot uses a sensor that can measure ranges from $0m$ to $3m$. For simplicity, assume that the actual ranges are distributed uniformly in this interval. Unfortunately, the sensors can be faulty. When the sensor is faulty it constantly outputs a range below $1m$, regardless of the actual range in the sensor's measurement cone. We know that the prior probability for a sensor to be faulty is $p = 0.01$.
   Suppose the robot queries its sensors N times, and every single time the measurement value is below $1m$. What is the posterior probability of a sensor fault, for $N = 1, 2, ..., 10$. Formulate the corresponding probabilistic model.

3.6 Noisy sensors and motion: Probabilistic sensor and motion models

Having acknowledged that sensors are noisy, we now turn to their corresponding probabilistic description that will be used later in this course. A **sensor model** describes the likelihood of a state value at time t, which we denote here with x_t, given a certain reading of the sensor, denoted by Z. That is, a probabilistic sensor model describes

$$\textbf{Sensor model: } p(x_t|Z). \tag{3.18}$$

We make here the implicit assumption that this measurement model does not depend on the history of previous states. Let us make a specific sensor model for the ultrasonic

sensor. For this take a tape measure and me sure a specific distance from a surface and measure the corresponding reading from the sensor. Repeat this several times for different surfaces and plot the resulting distribution. What shape does this distribution resemble? Let us assume that this distribution is Gaussian with a bias b_s around the actual distance and variance σ_s^2. The noise model can then be written as

$$p(x_t|Z;\theta) = \frac{1}{\sqrt{2\pi}\sigma} e^{\frac{(x+b_s)^2}{2\sigma_s^2}}. \tag{3.19}$$

We have denoted all the parameters on the left hand side with a generic vector θ. The measurement models encapsulates all the information we can get from a sensor.

Sensors are not the only noise part of a robot. Indeed the motion of a robot is commonly even more unreliable. We thus need a motion model that takes uncertainties into account and that returns the probability of a new pose x_t after applying a motor command m. The new state might depend on the previous history $\{x_0, ..., x_{t-1}\}$, but we make here the common **Markov assumption** that the new state only depends on the previous state,

$$\textbf{Motion model: } p(x_t|x_{t-1}, m). \tag{3.20}$$

For example, let us consider the tribot moving on a one-dimensional trajectory were we specify the position with x. We will give motor commands that will turn on both motors for a specific time t_m. The displacement of the robot is then to a first approximation linear in this time,

$$\Delta(x)(x_t - x_{t-1} = a_0 + at_m. \tag{3.21}$$

We included here a constant a_0 that could, for example, describe the effect of a latency when applying the motor command. If we only take this internal model into account to calculate the new position without sensor input, then this is often called **dead reckoning** in navigation. However, we also know that movements will introduce errors such as from slippage of the weals or external factors that alters the position of the robot. You should try this with the tribot. Run the tribot repeatedly for a specific time and measure the distance traveled. You should plot a histogram of these positions to estimate the noise distribution. Let us assume again that this noise is Gaussian. The motion model can then be written as

$$p(x_t|x_{t-1}, t_m; \theta) = x_{t-1} + a_0 + at_m + \frac{1}{\sqrt{2\pi}\sigma} e^{\frac{(b_m)^2}{2\sigma_m^2}}, \tag{3.22}$$

with parameters b_m and σ_m.

Exercise

1. Use the light sensor to measure distances to a surface and derive a sensor model for this sensor. Provide a parametric form of your model and include estimations of the parameters.
2. Derive a motion model for the tribot when driving the motors with different power parameters. Provide a parametric form of your model and include estimations of the parameters.