2 Sensing, acting and control

This is a busy chapter where we review some fundamental techniques for robotics. We will learn how to acquire images from a webcam and to filter the image in order to look for specific items. We will then explain how to use a Lego NXT with Matlab, that is, how to send motor commands and how to receive sensor information that we can then use in our math lab program. We will practice this programming with some first robotics tasks. Furthermore, we will discuss some basic kinematics models for some example robots and finally review some basic control theory as it is essential in robotics applications and it also provides us with a framework to explore machine learning methods.

2.1 Basic computer Vision

Cameras and other sensors that can sense physical objects in the environment, such as infrared cameras to sense heat distributions or scanning sonars for underwater applications, are an important source of sensory information. The main challenge with such data is how to interpret them as we usually want to extract more meaningful information from them such as recognizing objects or distances to obstacles. Vision is a major sensory source for humans and our brain is specialized in interpreting signals from our eyes. Machine learning has contributed considerably to recent progress in computer vision including object tracking and object recognition. We will not enter into this discuss here but rather show how to use a webcam with Matlab and how to do basic operations on such acquired pictures of videos that typically form the first stage of more sophisticated vision systems. These techniques will also become handy in later experiment.

2.1.1 Acquiring data from webcams with Matlab

In order to process video streams, we will use Mathwork's Image Acquisition Toolbox in Matlab[3]. First we make sure the toolbox is installed in your Matlab version and configured and describe commands to retrieve information of your specific system. To do so, use the command

```
imaqhwinfo

ans =
```

[3]There are also some alternatives, For example see http://www.mathworks.com/matlabcentral/fileexchange/35554-simple-video-camera-frame-grabber-toolkit

```
    InstalledAdaptors: {'dcam'  'gige'  'macvideo'}
        MATLABVersion: '8.1 (R2013a)'
          ToolboxName: 'Image Acquisition Toolbox'
       ToolboxVersion: '4.5 (R2013a)'
```

More specific information can be obtained with

```
>> HD=imaqhwinfo('macvideo')

HD =

       AdaptorDllName: [1x85 char]
    AdaptorDllVersion: '4.5 (R2013a)'
          AdaptorName: 'macvideo'
            DeviceIDs: {[1]}
           DeviceInfo: [1x1 struct]
```

More specific information of the supported video format and size can be obtained by inspecting the previously ceated HD object,

```
HD.DeviceInfo(1)

ans =

             DefaultFormat: 'YCbCr422_1280x720'
       DeviceFileSupported: 0
                DeviceName: 'FaceTime HD Camera (Built-in)'
                  DeviceID: 1
     VideoInputConstructor: 'videoinput('macvideo', 1)'
    VideoDeviceConstructor: 'imaq.VideoDevice('macvideo', 1)'
          SupportedFormats: {'YCbCr422_1280x720'}
```

Now we are ready to show how to create a video stream and to display it in a Matlab figure window. On a windows system, likely the most common way to achieve this is

```
stream = videoinput('winvideo', 1);
preview(stream);
```

Under normal circumstances, a new window opens with a preview of the video stream. Mac and Unix user should replace the string `winvideo` with `macvideo` or `unixvideo` respectively. The number ofter this string represents the ID of the camera. The build-in camera has usually ID=1, but you might need to specify another number when you use an external camera. The string you should use is also specified in the VideoInputConstructor line from the DeviceInfo command.

In order to process a frame in this video image we need to retrieve a single frame from the video stream that we created previously. First, we specify the colorspace we want to obtain, such as RGB, then get a snapshot from the video steam, and finally display it with the `imgshow` command,

```
set(stream,'ReturnedColorSpace','rgb');
frame = getsnapshot(stream);
imshow(frame);
```

The picture is stored as object *frame* in the Matlab Workspace. Its size depends on the resolution of the webcam and the chosen colorspace. With a 720x1280 resolution and in RGB for example, the obtained *frame* will be a 720x1280x3 *uint8* object. As an example to process this image directly with Matlab, let us extract the red component and display this alone with the *imshow* command

```
frameGrey=frame(:,:,1);
imshow(framGrey)
\end{verbatim}
The reason that this image appears in grey is that the values  ...
    in the two dimensional matrix are now interpreted as grey  ...
    values.

Finally, in order to read continuously from a camera and  ...
    display the obtained frames in a loop one can use the  ...
    following program.  Press the \textit{q} key to terminate  ...
    the loop.

\begin{lstlisting}
close all; clear all;

stream=videoinput('winvideo',1);
triggerconfig(stream,'manual');

VideoLoop=figure;
while true
    frame=getsnapshot(stream);
    imshow(frame);
    %retrieves a keyboard interruption
    key=get(gcf,'currentkey');
    %if the pressed key is 'q', the loop is interrupted and  ...
        the figure closes
    if strcmp(key,'q')
        close(VideoLoop);
        break;
    end
end
```

2.1.2 Image filtering with convolutions

Let us now start manipulating a singe grey image further. As a first example let us created a new smoothened image I^{mean} by averaging the pixels over a certain region, say over a region of size 11 by 11 pixels. The value of a pixel at (x, y) of the new image is then defined by us to be the average pixel values of an 11×11 image patch, and we shift around the centre pixel at (x, y),

$$I^{\text{mean}}(x, y) = \frac{1}{121} \sum_{u=-5}^{5} \sum_{v=-5}^{5} I(x-u, y-v). \tag{2.1}$$

The new image is a bit smaller than the original as the pixels at the edges don't have pixels on one side. We could adjust for this in various ways such as buffering a surrounding are with with constants pixels or using periodic boundary conditions where we add pixels from the other side of the matrix. In order to accommodate filters with even sizes of pixels, we could also assigned the average within a patch of the image to the upper left corner of this patch, or too any other location within the patch. The important part is that we are moving a square systematically around the image and in this way generate a new processed version of this image.

In order to generalize this averaging procedure later to averages with different weights, we define a matrix $k(u, v)$ with indices u and v. to continue the example above, both indices could run between the values of -5 and 5. All the elements of this matrix are set to one, $k(u, v) = 1$, so that the above equation is equivalent to

$$I^{\text{mean}}(x, y) = \frac{1}{N^2} \sum_u \sum_v I(x - u, y - v)k(u, v), \tag{2.2}$$

where $N = 11$ is the number of pixels in the filter.

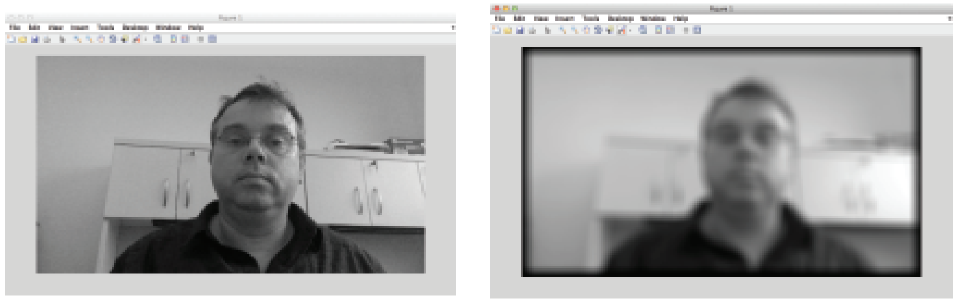


Fig. 2.1 Original picture on the left and the filtered version with a uniform filter of size 40×40.

An example of such a procedure is shown in Fig. 2.1. On the left side is the original image acquired with a webcam with 720×1280 pixels. On the right is a smoothened version of it using the procedure just defined. The image is a bit more blurry, but we will see that this will be useful for some of the applications below such as when downsampling images or to reduce noise in the image.

The matrix k is called a kernel, and the operation described in eq.2.2 is called a **convolution**. For a large number of pixels it is sometimes more convenient to describe the image as a continuum, so that a convolution can be written as

$$I^{\text{mean}}(x, y) = \int_u \int_v I(x - u, y - v)k(u, v)dudv. \tag{2.3}$$

Of course, we can define convolutions in different dimensions, not just in the two dimensional picture plane described here. By defining different kernel function we can achieve different effects. For example, it might seem more natural to average an image more smoothly, given nearby nodes more weight than distant nodes. This can be achieved with a **Gaussian kernel**

$$k(u,v) = \frac{1}{\sqrt{2\pi}\sigma} e^{\frac{-(u,v)^2}{\sigma^2}}. \tag{2.4}$$

Smoothing with Gaussian kernels is a common technique in computer vision, and the resulting picture for our test image is shown in Fig. 2.1b. The kernel function also defines a filter, and a convolution can be seen as a linear filtering operation.

Exercise

An example program that was used to produce the filtered image shown on the right in Fig. 2.1 is given below. This program uses the build in Matlab function *conv2()* to calculate the 2-dimensional convolution. Write a Matlab function that replaces this function and implements the convolution from scratch. Explain the black border in the filtered image.

```
original=frame(:,:,1);
imshow(original)

filter=ones(40);
filtered=conv2(filter,double(original));
filtered=filtered./max(max(filtered))*255;
imshow(uint8(filtered))
```

2.1.3 Linear filtering: Finding a color blob

An easy way to localize some environmental object is by tagging it with some unique colour and trying to detect this in the image. This will be used later for some exercises in localization and planing. For the following exercise take some coloured electrical tape of some other coloured material and attach it to the robot arm. We can first test it statically, but we will later use it to detect the location of the arm when the arm is moving.

To detect a certain colour in an image we need to process the colour channels. We can write a little application that takes an image and in which we could point to a location in the image to return the values. This program is shown in Table **??**. (explain program)

Once we have RGB value for the target colour we can use them to locate the colour in a video stream. For this it is useful to take some of the absolute differences between a video screen colour values and the target values. Small values indicate pixels close to the target colour. Since the target area corresponds to a cluster of such pixels, we could use an averaging method such as Gaussian smoothing followed by finding the minimum to locate the centre of the target area.

An alternative to the colour method for finding the position of the robot arm it motion segmentation. Segmentation of an image is an important step in building scene representations, and the following sections talks about some methods that commonly build the basis of segmentation for still images. The beauty of video streams is that there is more information in it that we can use for segmentation. In the example with the robot arm, we assume that only the robot arm is moving. We can therefore use differences of video captures in consecutive frames to determine the moving object.

Finally we want to translate the tracking of the robot arm to a number representing the degrees of rotation of the upper motor of the robot arm. For this we will use machine learning techniques. The first is to use linear regression on the motion segmented robot arm. The other is to use the support vector regression to map the (x, y) coordinates to rotation angles. Note that both cases correspond to supervised learning that require measurements that we will use as teacher signals.

Exercise

- Write a program to locate a colour blob in a video stream and indicate this target location with a circle. Similarly, use as an alternative motion segmentation and compare the location estimation in form of a pixel coordinate between the two methods.
- Write a program that translates a pixel coordinate to the estimation of the rotation angle of the motor and compare the location estimation of the two segmentation methods with the coordinates returned by the motors.

2.1.4 Gradient filters: Edge detection

While Gaussian smoothing is useful for noise reduction, it does not help us much with the identification of objects. To work towards such a goal we should recognize that objects are somewhat defined by their extensions, and the borders of objects are typically characterized by edges in a two-dimensional image. It is hence useful to think about how to build filters that highlight edges. For example, let us consider an image with a sharp vertical edge like the one give by the matrix

$$I^{\mathrm{v}} = \begin{pmatrix} 100\ 100\ 100\ 10\ 10\ 10 \\ 100\ 100\ 100\ 10\ 10\ 10 \\ 100\ 100\ 100\ 10\ 10\ 10 \\ 100\ 100\ 100\ 10\ 10\ 10 \end{pmatrix}$$

and lets convolve this with the filter $k = (1, -1)$ the resulting image is

$$I^{\mathrm{vedge}} = \begin{pmatrix} 0\ 0\ 90\ 0\ 0 \\ 0\ 0\ 90\ 0\ 0 \\ 0\ 0\ 90\ 0\ 0 \\ 0\ 0\ 90\ 0\ 0 \end{pmatrix}$$

Similar, let us consider an image with a horizontal edge

$$I^{\mathrm{h}} = \begin{pmatrix} 100 & 100 & 100 & 100 & 100 & 100 \\ 100 & 100 & 100 & 100 & 100 & 100 \\ 10 & 10 & 10 & 10 & 10 & 10 \\ 10 & 10 & 10 & 10 & 10 & 10 \end{pmatrix}$$

and the filter $k = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$. The resulting image highlights a horizontal edge

$$I^{\text{vedge}} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 90 & 90 & 90 & 90 & 90 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Of course, edges in our webcam pictures are never this sharp, and it is hence useful to smoothen them. A continuous version of edge filters is for example described by Gabor functions such as the ones shown in Fig. 2.2a and b. A Gabor function is described by a sinosodally-moduated Gaussian,

$$k(u,v) = e^{-\frac{u^2+\gamma v^2}{2*\sigma^2}} \cos(\frac{2\pi}{\lambda}u + \varphi). \tag{2.5}$$

The example of a 64^2 pixel filter with parameters $\gamma = 0.5$, $\sigma = 10$, $\lambda = 32$, and $\varphi = \pi/2$ is shown in Fig. 2.2a. This filter can also be rotated with a rotation matrix

$$\begin{pmatrix} x \\ y \end{pmatrix} \leftarrow \begin{pmatrix} \cos(\varphi) & \sin(\varphi) \\ -\sin(\varphi) & \cos(\varphi) \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \tag{2.6}$$

as shown in Fig. 2.2b for $\varphi = \pi$.

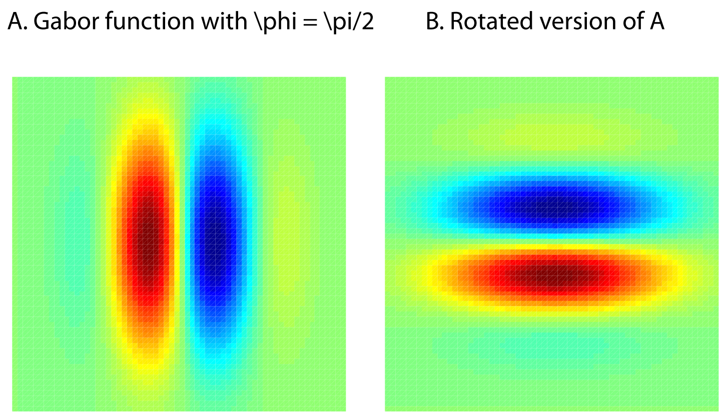


Fig. 2.2 Example of Gabor functions for (a) vertical and (b) horizontal edge detection.

Exercise

Take an image of your choosing and use Gabor filters to filter the image. Show the resulting image with two different angular parameter.

2.2 Building and 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 Lego NXT 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.2.3 A. Our basic robot arm is constructed by attaching the base of one motor, that we call elbow, to the rotating part of a second motor, that we call the shoulder, as shown in Fig.2.3A. We also attach a long pointer extension to elbow 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.2.3B. The tribot used here is a slight modification of the standard tribot as described 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 it 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.

2.2.2 NXT Matlab Software Environment

The 'brain' of our robots will be implemented on PCs and we will use a Matlab environment to implement our high-level controllers. Most examples are minimalistic in order 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 Matlab environment and packages that we use in the following.

2.2.3 Mindstorm NXT toolbox installation

We will use some software to control the Lego actuator and gather information from their sensors within the Matlab programming environment. To enable this we need to install software developed at the German university called 'RWTH Aachen', which in turn uses some other drivers that we need to install. Most of the software should be installed in our Lab, but we will outline briefly some of the installation issues in case you want to install them under your own system or if some problems exists with the current installation. The following software installation instructions are adapted from

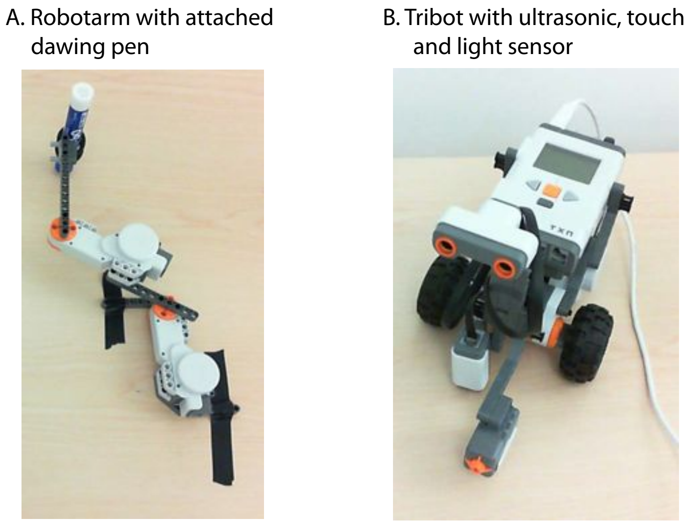

A. Robotarm with attached dawing pen
B. Tribot with ultrasonic, touch and light sensor

Fig. 2.3 (A) A robot arm made out of two motors, shoulder and elbow, 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.

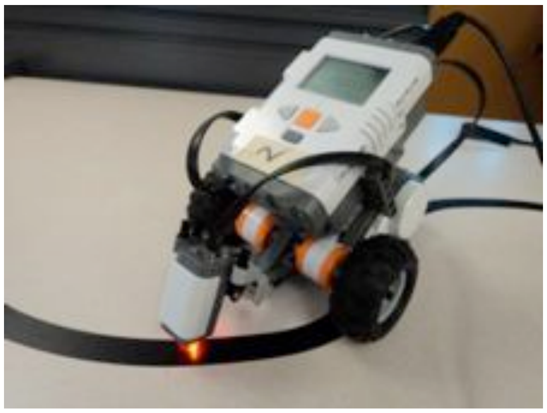


Fig. 2.4 Basic Lego Robot with microprocessor, two motors, and a light sensor.

RWTH Aachen University's NXT Toolbox website:
http://www.mindstorms.rwth-aachen.de/trac/wiki/Download4.03

1. **Check NXT Firmware version**
   Check what version of NXT Firmware is running on the NXT brick by going to "Settings" > "NXT Version". Firmware version ("FW") should be 1.28. If it does not, it needs to be updated (Note: The NXT toolbox website claims version 1.26 will work, however it will not)
   To update the firmware:

The Lego Mindstorms Education NXT Programming software is required to update the firmware. In the NXT Programming software, look under "tools" > "Update NXT Firmware" > "browse", select the firmware's directory, click "download".

2. **Install USB (Fantom) Driver (Windows only)**
If the Lego Mindstorms Education NXT Programming software is already on your computer, this should already be installed. Otherwise, download it from: http://mindstorms.lego.com/support/updates/
If you run into problems with the Fantom Library on windows go to this site for help. http://bricxcc.sourceforge.net/NXTFantomDriverHelp.pdf
If you have Windows 7 Starter edition the standard setup file will not run properly. To install the Fantom Driver go into Products and then Lego_NXT_Driver_32 and run LegoMindstormsNXTdriver32.
The fantom USB driver seem not to work on the Mac, but we will anyhow use the bluetooth connections.
3. **Download the Mindstorms NXT Toolbox 4.03:**
Download: http://www.mindstorms.rwth-aachen.de/trac/wiki/Download
   - Save and extract the files anywhere, but do not change the directory structure.
   - The folder will appear as "RWTHMindstormsNXT"
4. **Install NXT Toolbox into Matlab**
In Matlab: "File" > "SetPath" > "Add Folder", and browse and select "RWTHMindstormsNXT" - the file you saved in the previous step.
   - Also add the "tools" folder, which is a subdirectory of the RWTHMindstormsNXT folder.
   - Click "save" when finished.
5. **Download MotorControl to NXT brick**
Go to http://bricxcc.sourceforge.net/utilities.html for the download. Use the USB cable for this step

Windows: Download NeXTTool.exe to RWTHMindstormsNXT/tools/MotorControl. Under RWTHMindstormsNXT/tools/MotorControl, double click TransferMotorControlBinaryToNXT, click "Run", and follow the onscreen instructions. If this fails, try using the NBC compiler (download from http://bricxcc.sourceforge.net/nbc/) instead of the NeXTTool; again save it under the MotorControl folder.

Mac: Download the NeXTTools for Mac OS X. Run the toolbar and open the XNT Explorer (the globe in the toolbar). With the arrow key at the top, transfer the file MotorControl21.rxe to the brick.

6. **Setting up a Bluetooth connection**

To connect to the NXT via bluetooth you must first turn on the bluetooth in the NXT and make sure that the visibility is set to on. Then use the bluetooth device on your computer to search for your specific NXT. By default the name is NXT, but as a first step we will rename each brick.

Create a connection between the computer and the NXT. When you create the connection between the NXT and the bluetooth device the NXT will ask for a passkey (usually either 0000 or 1234 on the NXT screen and press the orange button. The computer will then ask for the same passkey. To test the connection, type the command `COM_OpenNXT('bluetooth.ini');` in the Matlab command window. The command should run without any red error messages.

If there is an error check to see if the COMPort the Matlab code is looking for is the same as the one used in the connection made between the bluetooth device and the NXT. Also turning the NXT off and back on again can help. After every failed `COM_OpenNXT('bluetooth.ini');` command type `COM_CloseNXT('all');` to close the failed connection for a clean new attempt. To switch on a debug mode enter the command `DebugMode on` before entering the command `COM_OpenNXT('bluetooth.ini');` . Also, make sure that the bluetooth.ini file is present. There are sample files for Windows ad Linux (Mac) in the main RWTH toolbox folder. Also, check if the port name is correct by typing `ls -ltr /dev` in a terminal window.

7. **Does it work?**
In Matlab, enter the commands below into the command window. The command should execute without error and the NXT should play a sound.
```
h=COM_OpenNXT('bluetooth.ini');
COM_SetDefaultNXT(h);
NXT_PlayTone(400,300);
```

2.2.4 Basic Matlab NXT commands

The following is a summary of most Matlab commands from the NXT toolbox. The documentation from the RWTH web site contains always updated information.

2.2.4.1 Startup NXT

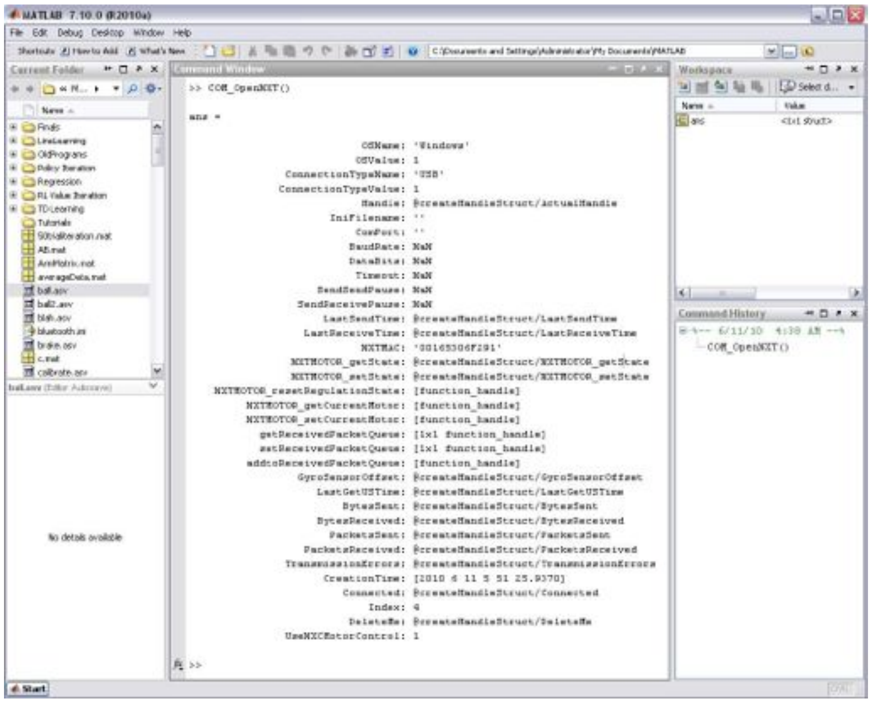
The first thing to do is make sure the workspace is clear. Enter:
```
COM_CloseNXT('all');
close all;
clear all;
```

To start, enter:
```
hNXT=COM_OpenNXT;          %hNXT is an arbitrary name
COM_SetDefaultNXT(hNXT);   %sets opened NXT as the
                           %default handle
```

2.2.4.2 NXT Motors

Motors are treated as objects. To create one, enter:
```
motorA = NXTMotor('a');   %motorA is an arbitrary name, 'a' is
                          %the port the motor connected to
```

This will give:



```
>> COM_OpenNXT()

ans =

                          OSName: 'Windows'
                         OSValue: 1
              ConnectionTypeName: 'USB'
             ConnectionTypeValue: 1
                          Handle: @createHandleStruct/actualHandle
                     IniFilename: ''
                         ComPort: ''
                        BaudRate: NaN
                        DataBits: NaN
                         Timeout: NaN
                   SendSendPause: NaN
                SendReceivePause: NaN
                    LastSendTime: @createHandleStruct/LastSendTime
                 LastReceiveTime: @createHandleStruct/LastReceiveTime
                          NXTMAC: '00165306F291'
               NXTMOTOR_getState: @createHandleStruct/NXTMOTOR_getState
               NXTMOTOR_setState: @createHandleStruct/NXTMOTOR_setState
    NXTMOTOR_resetRegulationState: [function_handle]
        NXTMOTOR_getCurrentMotor: [function_handle]
        NXTMOTOR_setCurrentMotor: [function_handle]
          getReceivedPacketQueue: [1x1 function_handle]
          setReceivedPacketQueue: [1x1 function_handle]
        addtoReceivedPacketQueue: [function_handle]
                GyroSensorOffset: @createHandleStruct/GyroSensorOffset
                   LastGetUSTime: @createHandleStruct/LastGetUSTime
                       BytesSent: @createHandleStruct/BytesSent
                   BytesReceived: @createHandleStruct/BytesReceived
                     PacketsSent: @createHandleStruct/PacketsSent
                 PacketsReceived: @createHandleStruct/PacketsReceived
              TransmissionErrors: @createHandleStruct/TransmissionErrors
                    CreationTime: [2010 6 11 5 51 25.9370]
                       Connected: @createHandleStruct/Connected
                           Index: 4
                        DeleteMe: @createHandleStruct/DeleteMe
               UseNXCMotorControl: 1

>>
```

Fig. 2.5 Example of calling the `COM_OpenNXT()` command without arguments. The command returns some information of about the system.

```
NXTMotor object properties:
                     Port(s): 0    (A)
                       Power: 0
             SpeedRegulation: 1    (on)
                 SmoothStart: 0    (off)
                  TachoLimit: 0    (no limit)
      ActionAtTachoLimit: 'Brake'  (brake, turn off when stopped)
```

Below is a list of these properties and how to change them:

Power
Determines speed of the motor

```
motorA.Power=50;   % value must be between -100 and 100 (negative
                   % will cause the motor to rotate in reverse)
```

SpeedRegulation
If the motor encounters some sort of load, the motor will (if possible) increase it power to keep a constant speed

```
motorA.SpeedRegulation=true;   % either true or false, or
                               alternatively, 1 for true, 0 for
                               false
```

SmoothStart

Causes the motor to slowly accelerate and build up to full speed.
Works only if ActionAtTachoLimit is not set to 'coast' and if TachoLimit>0

```
motorA.ActionAtTachoLimit= true;   % either true or false, or
                                   % 1 for true, 0 for false
```

ActionAtTachoLimit
Determines how the motor will come to rest after the TachoLimit has been reached.
There are three options:
1. 'brake': the motor brakes
2. 'Holdbrake': the motor brakes, and then holds the brakes
3. 'coast' the motor stops moving, but there is no braking

```
motorA.ActionAtTachoLimit='coast';
```

TachoLimit
Determines how far the motor will turn

```
motorA.TachoLimit= 360;   % input is in terms of degrees
```

Alternative Motor Initiation
Motors can also be created this way:

```
motorA=NXTMotor('a', 'Power', 50, 'TachoLimit', 360);
```

SendToNXT
This is required to send the settings of the motor to the robot so the motors will actually run.

```
motorA.SendToNXT();
```

Stop
Stops the motor. There are two ways to do this:
1. 'off' will turn off the motor, letting it come to rest by coasting.
2. 'brake' will turn cause the motor to be stopped by braking, however the motors will need to be turned off after the braking.

```
motorA.Stop('off');
```

ReadFromNXT();
Returns a list of information pertaining to a motor

```
motorA.ReadFromNXT();
```

Entering `motorA.ReadFromNXT.Position();` will return the position of the motor in degrees.

ResetPosition
Resets the position of the motor back to 0

```
motorA.ResetPosition();
```

WaitFor
Program will wait for motor to finish current command. For example:

```
motorA=('a', 'Power', 30, 'TachoLimit', 360);
motorA.SendToNXT();
motorA.SendToNXT();
```

The immediate repetition of the motor command will cause problems as the motor can only process one command at a time. Instead, the following should be entered:

```
motorA=('a', 'Power', 30, 'TachoLimit', 360)
motorA.SendToNXT();
motorA.WaitFor();
motorA.SendToNXT();
```

The exception to this is if TachoLimit of the motor is set to 0.

2.2.4.3 Using Two Motors At Once

Some operations, for example driving forward and backwards, require the simultaneous use of two motors. Entering:

```
B=NXTMotor('b', 'Power', 50, 'TachoLimit', 360);
C=NXTMotor('c', 'Power', 50, 'TachoLimit', 360);
B.SendToNXT();
C.SendToNXT();
```

will start the bot moving, but the signals for both motors to start at will not be sent at exactly the same time, so the robot will curve a little and fail to drive in a straight line. Instead, you should enter:

```
BC=NXTMotor('bc', 'Power', 50, 'TachoLimit', 360);
```

OR

```
BC = NXTMotor('bc);
BC.Power=50;
BC.TachoLimit=360;
```

Turning left or right can be achieved by only running one motor at a time, or by moving both motors, but one slower than the other.

2.2.4.4 Sensors

The following commands are used to open a sensor, plugged into port 1:

```
OpenSwitch(SENSOR_1);              % initiates touch sensor
OpenSound(SENSOR_1, 'DB');         % initiates sound sensor, using
                                   % either 'DB' or 'DBA'
OpenLight(SENSOR_1, 'ACTIVE');     % initiates light sensor as
                                   % either'ACTIVE' or 'INACTIVE',
                                   % plugged into Port 1
OpenUltrasonic(SENSOR_1);          % initiates ultrasonic sensor
                                   % plugged into Port 1
```

The following commands are used to get values from the sensor plugged into port 2:

```
GetSwitch(SENSOR_2);        % returns 1 if pressed, 0 if depressed
GetSound(SENSOR_2);         % returns a value ranging from 0-1023
GetLight(SENSOR_2);         % returns a value ranging from 0 to
                            % a few thousand
GetUltrasonic(SENSOR_2);    % returns a value in cm
```

To close a sensor, ex. Sensor 1:

```
CloseSensor(SENSOR_1);   %properly closes the sensor
```

2.2.4.5 Direct NXT Commands

PlayTone
Plays a tone at a specified frequency for a specified amount of time

```
NXT_PlayTone(400,300);   % Plays a tone at 400Hz for 300 ms
```

KeepAlive
Send this command every once in a while to prevent the robot from going into sleep mode:

```
    NXT_SendKeepAlive('dontreply');
```

Send this command to see how long the robot will stay awake, in milliseconds:

```
   [status SleepTimeLimit] = NXT_SendKeepAlive('reply');
```

GetBatteryLevel
Returns the voltage left in the battery in millivolts

```
   NXT_GetBatteryLevel;
```

StartProgram/StopProgram
To run programs written on LEGO Mindstorms NXT software, enter:

```
NXT_StartProgram('MyDemo.rxe')   % the file extension '.rxe' can be
                                 % omitted, it will then be automatically
                                 % added
```

Entering `NXT_StopProgram` stops the program mid-run

2.2.5 First examples:

Wall avoidance

The following is a simple example of how to drive a robot and use the ultrasonic sensor. The robot will drive forward until it is around 20 cm away from a barrier (i.e. a wall), stop, beep, turn right, and continue moving forward. The robot will repeat this 5 times. Attach the Ultrasonic sensor and connect it to port 1. The study and run the following program.

```
COM\_CloseNXT('all'); &\%cleans up workspace\\
close all;\\
clear all;\mbox{}\\

hNXT=COM\_OpenNXT('bluetooth.ini'); &\% initiates NXT, hNXT is  ...
    an arbitrary name\\
COM\_SetDefaultNXT(hNXT);&\%sets default handle\\\mbox{}\\

OpenUltrasonic(SENSOR\_1);\\\mbox{}\\

forward=NXTMotor('BC';); \&\% setting motors B&C to drive forward\\
forward.Power=50;\\
forward.TachoLimit=0;\\
turnRight=NXTMotor('B'); \&\% setting motor B to turn right\\
turnRight.Power=50;\\
turnRight.TachoLimit=360;\mbox{}\\

for i= 1:5\\
\indent while GetUltrasonic(SENSOR\_1)>20\\
\indent\indent forward.SendToNXT(); &\%sends command for robot  ...
    to move forward\\
\indent\indent &\%TachoLimit=0; no need for a WaitFor() statement\\
\indent end \%while\\
\indent forward.Stop('brake'); &\%robot brakes from going forward\\
\indent NXT\_PlayTone(400,300); &\%plays a note\\
\indent turnRight.SendToNXT; &\%sends the command to turn right\\
\indent turnRight.WaitFor; &\%TachoLimit is not 0; WaitFor()  ...
    statement required\\
end \%for \mbox{}\\

turnRight.Stop('off'); &\%properly closes motors\\
forward.Stop('off');\\
CloseSensor(SENSOR\_1); &\%properly closes the sensor\\
COM\_CloseNXT(hNXT); &\% properly closes the NXT\\
close all;\\
clear all;
```

Exercises: 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

In order to control a robot we must know in which state it is in. This is usually called a **pose** in robotics, though other areas in physics and engineering also call this generally a **state**. The collection of all possible states is called the **state space**. Thus, while the state or pose describes the current configuration of a robot, the state space contains in addition some information of the environment such as pose constrains. Sometimes we could also talk about the state of a system which includes a robot and the environment, and the specific use is usually clear from the context.

The physical state of an agent can be quite complicated. In order to describe the robot in all its details we neeTd to describe the physical shape of the robot, the state of motors, the colour of components, the charging level of the battery etc. Thus, a detailed description of the physical state of a robot 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. However, not all the dimensions are relevant in solving specific tasks, so that we can often simplify the description in order to better manage specific control problems. The simplification of the decryption is better described as **abstraction**, which is an important concept in science in general. 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 need capture to function properly.

For a mobile robot, most of the time we are mainly concerned with the position of the robot in space and its orientation. The pose, l, of a mobile robot that lived in a two-dimensional plane can hence be described by three variable, the planar positions x and y, and the rotation angle Θ as shown in Fig. 2.6A,

$$l = \begin{pmatrix} x \\ y \\ \Theta \end{pmatrix} \tag{2.7}$$

An airplane or marine underwater vehicle usually needs 6 variables to describe a basic pose, three spacial position coordinates and there angles called pitch, roll and yaw as shown in Fig. 2.6B. Our simple robot arm can be described by two angles, although industrial robots have often as many as 6 degrees, and the human arm is considered to have 7 degrees of freedom.

To illustrate a state space, let us consider a planar robot that should navigate around some obstacles from point A to point B as shown in Fig.2.7A. An obstacle, such as the one represented by the grey area in Fig.2.7A, 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 later discuss some navigation algorithms where we will use search algorithms in the state space to find solutions, and having 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.2.7B, where we used a

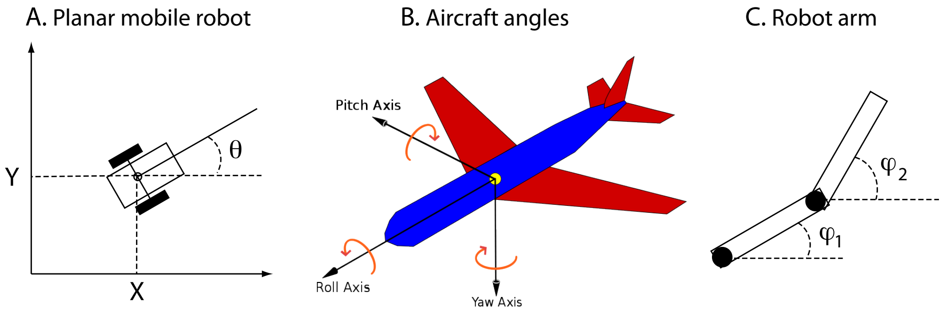


Fig. 2.6 (A) Mobile robot in a plane. (B) Angles of an Aircraft. (C) Robot arm.

grid to represent the possible values of the robot positions. Such a discretization is very common in practice although it introduces some problems of its own. It is possible to make the discretization error increasingly small by decreasing the grid parameter, Δx, which describes the minimal distance between two states at the expense of increasing the state space, though we will later also discuss other solutions.

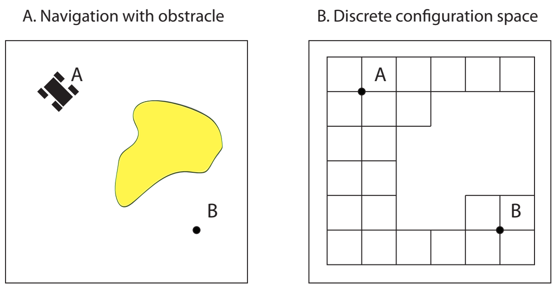


Fig. 2.7 (A) A physical space with obstacles in which an agent should navigate from point A to point B. (B) Descretized configuration space.

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.

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

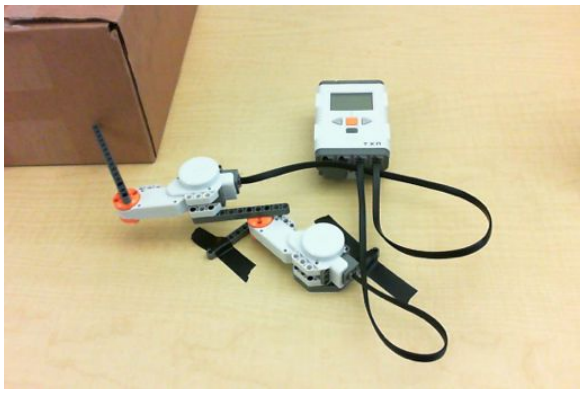


Fig. 2.8 (A) A simple robot arm with two joints that can point at an obstacle.

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 `ReadFromNXT.Position()` in Matlab 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 Kinematics and motion models

2.4.1 Kinematic model of the two-joint robot arm

To control an object, we need to understand the effect of a motor command such as turning on a motor for a certain time. The equations describing such an influence are called a **kinematics model** or **motion model** in robotics. Let us illustrate this with the two-joint robot arm. We consider first a single motor, and assume that we can drive the motor with a constant power p for a certain time t. This would imply that we can start to move the motor instantaneously with a constant rotational velocity v_φ at the initial rotation angle of $\varphi(0)$. The rotation angle at time t is then give by

$$\varphi(t) = \varphi(0) + v_\varphi t \tag{2.8}$$

This is certainly an approximation as the motor need some time to get to a certain speed and will also not stop immediately when turned off. There are many sources of noise in Robotics that will alter the precise end point. A major aim of the approaches advocated in later chapters is to address such noisy operations in a more general way. But for now we only want to illustrate the basic concepts of robotic movements.

You should try to follow our discussions here in practice by trying it out with a simple robot arm, one motor with a pointer. Apply to the motor a small power, e.g. p=10, and find out the time it needs to come back to the initial position after one full term. For example,

```
COM_CloseNXT('all')
```

```
h=COM_OpenNXT(); COM_SetDefaultNXT(h);
motorA=NXTMotor('a', 'Power',10);
tic;
motorA.SendToNXT();
while toc<3.7
end
motorA.Stop('brake');
COM_CloseNXT('all')
```

gives results roughly in a full turn $\Delta_\varphi = 2\pi$, so that the velocity constant is $v_\varphi = 2\pi/3.7$. Of course, we can also provide a desired angle as motor command for the Lego motors.

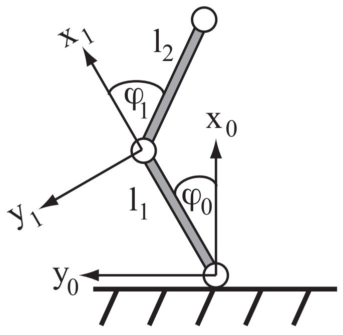


Fig. 2.9 The geometry of a simple robot arm with two joints.

We now know how a motor command is effecting the angle of the joint. Next we can calculate how the angles will effect the position of the arm based on the geometry of the arm. For example, we might want to know how the endpoint of the arm in the coordinate system of the base is related to the angles of the motors

$$(x_0, y_0) = \mathbf{f}(\varphi_1, \varphi_2) \tag{2.9}$$

We denoted here the coordinates of the endpoint x_0 and y_0 with index 0 to indicate that these are the coordinates with respect to the coordinate system given by the base. (x_0, y_0) is hence a specific point in the coordinate system given by the axis X_0 and Y_0.

In order to find this relation, let us consider another coordinate systems, the one that is given by the pointer of the first motor. This is illustrated in Fig. 2.9. A point (x_1, y_1) in the co-ordinate system of axes X_1 and Y_1 can be expressed relative to the base by adding the length of the arm l_1 to the x_1 value and then rotating the corresponding vector with angle φ_1. This can be written elegantly with the matrix operation

$$\begin{pmatrix} x_0 \\ y_0 \\ 1 \end{pmatrix} = A_1 \begin{pmatrix} x_1 \\ y_1 \\ 1 \end{pmatrix} \tag{2.10}$$

where the matrix operator is defined as

$$\mathbf{A}_i = \begin{pmatrix} \cos(\varphi_i) & sin(\varphi_i) & 0 \\ -\sin(\varphi_i) & cos(\varphi_i) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & l_i \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{2.11}$$

Similarly, a point (x_2, y_2) relative to the coordinate system X_2 and Y_2 can be expressed in X_1 and Y_1 coordinates with a similar equation

$$\begin{pmatrix} x_1 \\ y_1 \\ 1 \end{pmatrix} = A_2 \begin{pmatrix} x_2 \\ y_2 \\ 1 \end{pmatrix} \tag{2.12}$$

Now, the coordinates of the endpoint of the two-segment arm in the X_2 and Y_2 coordinate system is simply $x_2 = y_2 = 0$. The coordinates of the endpoint relative to the base are therefore

$$\begin{pmatrix} x_0 \\ y_0 \\ 1 \end{pmatrix} = A_1 A_2 \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \tag{2.13}$$

This equation, together with the description of how the angles change with each motor command, represent the basic kinematic model of our basic robot arm.

While this motion model describes our basic expectation of where end point of the arm is located after executing a motion command (angle update), we sometimes also need the **inverse model**. The inversion of the model is not always straight forward as this is usually an ill posed problem, meaning that there are often more than just one solution. In the specific case of our robot arm there are either no solution (when the target it outside the reach of the arm), exactly one solution (for targets that could be reached when both segments are aligned), or two solutions in all other cases (one with the elbow pointing right and one with the elbow pointing left). In practice is is even common to be able to find the inverse with other constraints such as areas with obstacles of specific paths along a motion. There excises a variety of mathematical tools for such problems to find analytical solutions. We will not discuss this further here, but we will later take another approach, that of finding of inverse kinematics by reinforcement learning.

2.4.2 Motion model for a differential drive robot

We now turn to the kinematics model of a differential drive robot like our tribot system. The geometry of such a robot is outlined in Fig. 2.10. It is again useful to distinguish two coordinate systems, the global reference frame of the floor, (x, y, θ), and the local reference frame centred on the robot (robocentric), $(x_\mathrm{R}, y_\mathrm{R}, 0)$. We can again use the matrix operator 2.11 to translate between these two reference frames, where l is now the distance between the origin of the floor coordinate system and the origin of the robot coordinate system that we have chosen to be a point P between the two driving wheels.

We no want to know how the velocity of the robot, $\mathbf{v}$ is related to the individual velocities of the two motors. Let's call the velocities of the motors $\dot{\varphi}_1$ and $\dot{\varphi}_2$. Let us first calculate the contribution of each wheel to the motion in the robots reference frame. In this reference frame the contribution of movements in the the Y_R direction is zero, $\dot{y}_\mathrm{R} = 0$, since the wheels don't produce a movement in this direction. The

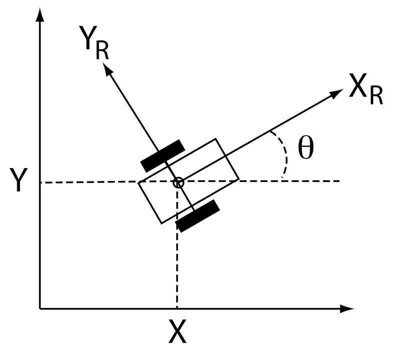


Fig. 2.10 The geometry of a differential drive robot.

contribution of each wheel to movements in the X_R direction is independent of each other, so we can simple add each individual movement. So let's consider that only wheel one moves with velocity $\dot{\varphi}_1$. The it generates a velocity of 1/2 of what it would do if it was the movement of an unattached motor. The same is true for the second motor, so that the instantaneous velocity along the X_R axis is given by

$$\dot{x}_\mathrm{R} = \frac{1}{2} r\dot{\varphi}_1 + \frac{1}{2} r\dot{\varphi}_2, \tag{2.14}$$

where r is the radius of the wheels. The contribution to the rotational speed is also independent for each motor and depends on the distance d between the wheel. SInce the velocity of wheel 1 generates a clockwise movement around wheel 2, and the velocity of wheel 2 produces an anticlockwise rotation around wheel 1, the rotation velocity is given by

$$\dot{\theta} = \frac{1}{2d} r\dot{\varphi}_1 - \frac{1}{2d} r\dot{\varphi}_2. \tag{2.15}$$

To transform this velocity vector into the floor reference frame it is good to realize that the velocity does not depend on the specific distance between the reference frames. Hence, the velocity of the robot in the floor reference frame is given by

$$\mathbf{v} = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \dot{x}_\mathrm{R} \\ 0 \\ \dot{\theta} \end{pmatrix} \tag{2.16}$$

While such a basic differential drive robot is hence quite easy to describe, many robots have more complicated geometries with many more degrees of freedom. They typically have also additional features and constrains that must be considered such as steerable wheels and slippage. Book dedicated to robotics does typically include more discussions on these kind of extensions. Instead of getting into this, we will concentrate on learning such motion models with machine learning methods.

2.4.3 Motion model for a line following robot

In this example we will develop a line racing robot that needs to guess the distance it has traveled. The race is along a straight line of electric tape. The robot starts in the middle of the tape and is expected to go forward until crossing an imaginary line at 1 meter. It then needs to go backwards and cross another imaginary line 1 meter in the other direction of the midline. This back-and-forth movement is repeated a set number of times, and the race ends when the robot crosses the midline the 10th time.

We can deploy the line following program from section **??** to make the tribot follow the line. It is possible to optimize the parameters a bit since we know that the movement is along a straight line. At this point we assume that the robot can reliably follow a line. There movement of the robot we need to describe with the following movement model is then essentially a one dimensional movement for position along the line. The distance traveled, d_f, is then basically linear in the time t_f in which the line following program is activated, namely

$$d_\mathrm{f}(t_f) = v_\mathrm{f} t_{rmf} \tag{2.17}$$

The distance depends on the velocity parameter v_f that has to be determined from experiments. A similar equation applies for the backdoor movement with a possible different velocity parameter v_b.

The better the motion model, the better the chances of winning in this competition. The general challenge is to drive forward or backward only as much as possible to cross the imaginary lines. Valuable time is waisted when overshooting the lines, but not reaching the line is even worst as the competitor would be disqualified. A light barrier could be used to aid in the judgement of the trial, but no feedback from the light barrier to the tribot is allowed.

The individual distances of forward and backward movement have to be added up to always have an estimation of the position,

$$x = \int_{\{t_f\}} v_\mathrm{f} t_\mathrm{f} \mathrm{d}t_\mathrm{f} + \int_{\{t_b\}} v_\mathrm{b} t_\mathrm{b} \mathrm{d}t_\mathrm{b} \tag{2.18}$$

This is called **path integration**. A major problem of path integration is that errors in each of the estimates will add up and can accumulate to a point where the position estimate is not any more sufficient for an application. We will later study this example when explicitly taking noise into account and also how to treat sensory input that can correct our estimates somewhat. Such movements and position estimation without sensory input is often called **dead reckoning** in robotics.

Exercise

Implement the line race and study how best to adjust the parameters to achieve win this race.

2.5 Basic controllers

The basic 'brain' of a simple decision-making robot, usual called the controller in robotics, should decide what actions to take depending on the sensory information. A common example is where the system is given a goal state and the controller determines what motor commands need to be initiated to reach this goal. This typically requires that we know the effect of motor commands on the system. But even this is not sufficient in the case of noisy or disturbed movements so that other strategies have to be included to react to situations where the goal has not yet been met. Control theory is a discipline in engineering that has been crucial to operate complex machines, and much of classical robotics is based on control theory. 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. In short, controllers are the brains of robots which use sensory and motivational information to produce goal directed behaviour.

2.5.1 Inverse plant dynamics

A system to be controlled is often called the **plant** in control theory since methods from this engineering area are typically applied to automation factories. A plant is the dynamical object that can be characterized by a **state vector**

$$\mathbf{z}^T(t) = (1, \mathbf{x}(t), \mathbf{x}'(t), \mathbf{x}''(t), ...). \tag{2.19}$$

$\mathbf{x}(t)$ 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**

$$\mathbf{z}(t+1) = \mathbf{f}(\mathbf{z}(t), \mathbf{u}(t)), \tag{2.20}$$

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 **contoller** is the part of the system that must generate the appropriate motor commands to achieve a goal. 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

$$\mathbf{z}(\mathbf{t+1}) = \mathbf{f}(\mathbf{z}(\mathbf{t}), \mathbf{u}(t)), \tag{2.21}$$

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

$$\mathbf{u}^* = \mathbf{f}^{-1}(\mathbf{z})\mathbf{z}^*. \tag{2.22}$$

A block diagram of this strategy with a basic controller and the plant is shown in Figure 2.11.

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

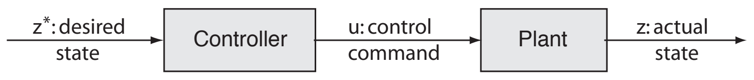


Fig. 2.11 The elements of a basic control system.

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

$$x(t) = x(0) + d_1 * t, \tag{2.23}$$

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

$$\hat{d}_1 = (x_1 - x_0). \tag{2.24}$$

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 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.

2.5.2 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.2.12 by a disturbance signal to the plant.

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

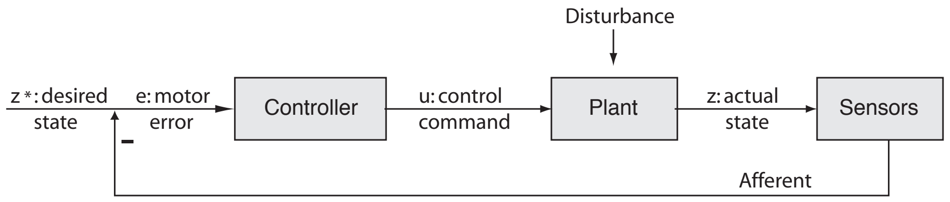


Fig. 2.12 The elements of a basic control system with negative feedback.

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**

$$e(t) = x^*(t) - x(t) \tag{2.25}$$

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. 2.12. 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. 2.12. 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 then 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.

2.5.3 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 an 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 2.13

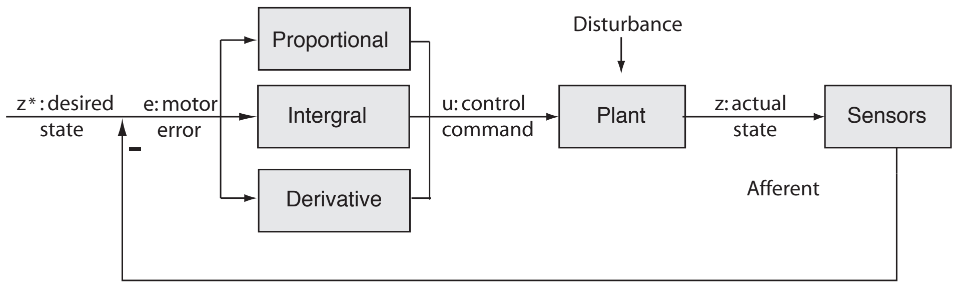


Fig. 2.13 The elements of a PID control system.

The motor command generated by a PID controller is given by

$$u(t) = k_P e(t) + k_I \int e(t) dt + k_D \frac{\mathrm{d}e(t)}{\mathrm{d}t}, \tag{2.26}$$

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 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.2.14. Adaptive control is a main focus of this book.

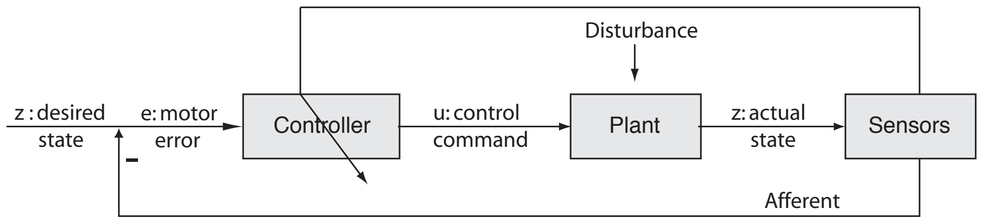


Fig. 2.14 Adaptive control systems incorporate models for corrective adjustments in the system indicated by the arrows going through these components. These models are trained with sensory feedback or other forms of external supervision.

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.