2 Sensing, acting and control

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 =

    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)'
```

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

```
          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

```
1 stream = videoinput('winvideo', 1);
2 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,

```
1 set(stream,'ReturnedColorSpace','rgb');
2 frame = getsnapshot(stream);
3 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

```
1 frameGrey=frame(:,:,1);
2 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 around the centre pixel at (x, y),

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

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 running between -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

$$I^{\text{mean}}(x, y) = \sum_{u} \sum_{v} I(x - u, y - v)k(u, v). \tag{2.2}$$

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. We use the later in the following.

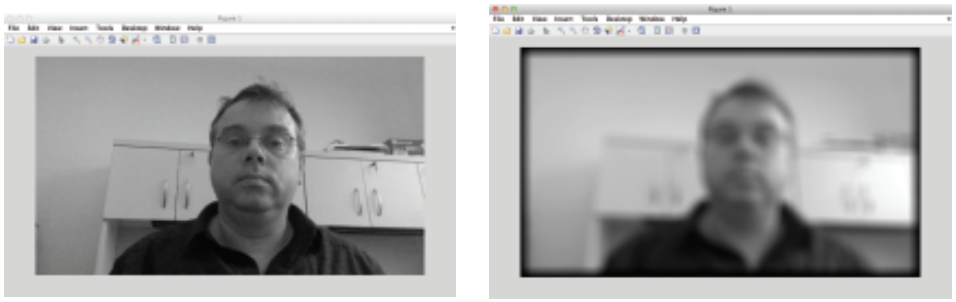


Fig. 2.1 Original picture on the left and the filtered version with a uniform filter of size 40×40.

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 filter 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.

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

```
3
4 filter=ones(40);
5 filtered=conv2(filter,double(original));
6 filtered=filtered./max(max(filtered))*255;
7 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

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

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,

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

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

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

as shown in Fig. 2.2b for $\alpha = \pi$. The figure also includes an example of applying these filters to an image from a webcam.

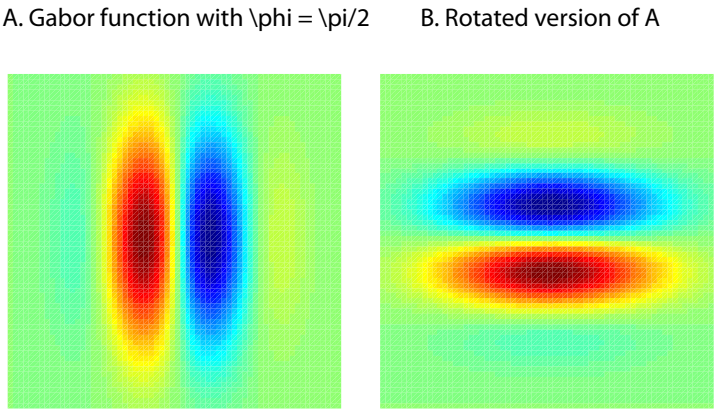


Fig. 2.2 Example of Gabor functions for (a) vertical and (b) horizontal edge detection. (c) Original image. (d) Filter Image using filters in (a) and (b).

Exercise

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

2.2 Building and driving a basic Lego NXT robot

2.2.1 Arm and Tribot

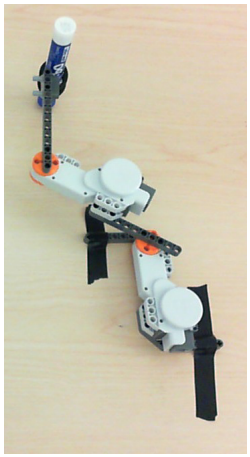
We will actively use the Lego Mindstorm robotics system in this course. This system is based on common Lego building blocks that we use for two principle designed that we build below. The Lego NXT robotics system includes a microprocessor in a unit called the **brick** which can be programmed and used 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.

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.

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