Introduction to openCV

(as a way to train openMP)

Version 1.0

António Abreu
Instituto Politécnico de Setúbal

March 4, 2013
Abstract

Hands on approach to openCV with the intent of supplying a substrate for (my) students to practice openMP.
Contents

1 Computer images ................................. 2
   1.1 Introduction .................................. 2

2 Accessing pixels in an image ............... 4
   2.1 Introduction .................................. 4
   2.2 Installation .................................. 4
   2.3 First way .................................... 5
   2.4 Second way ................................... 7
   2.5 Histogram ................................... 8

3 Pointwise image transformations .......... 10
   3.1 Gray scale ................................... 10
   3.2 The openMP way .............................. 11

4 Image transformations based on pixel neighborhood ............... 14
   4.1 Linear kernel ................................ 15
   4.2 Gaussian kernel .............................. 16
   4.3 Morphological operations - dilation and erosion .............. 18
   4.4 Morphological operations - image boundary ................... 20
   4.5 Morphological operations - image boundary from a gray image, in openMP ................................. 23
   4.6 Morphological operations - opening, closing and gradient .... 27
   4.7 Edge detection – Sobel ........................ 27
   4.8 Pattern recognition ........................... 30
Chapter 1

Computer images

1.1 Introduction

We start by presenting some basic properties of images.

Images are represented by two-dimensional matrices in computers. Our world is three-dimensional, but our ordinal cameras have a bi-dimensional array of light sensors, thus creating a corresponding bi-dimensional array of light intensity values. Each element of the image is called a pixel (from picture element). Computer memory is organized around sets of bytes. A byte is capable of representing $2^8 = 256$ different levels of some quantity. In the case of an image, light is a variable that is non negative, so if we use a single byte to represent levels of light, there can be 256 different light levels; this is enough for most cases.

When an image is represented by a single matrix, this means that a single or mono color image is being considered. Most of the time this means a gray-level image (i.e., from black to white), but the concept applies to any color. Since the pixel values represent levels of light, a zero value means lack of light, i.e., dark, while 255 represents the maximum possible value of light, which is white. When we talk about colors, e.g., red, the meaning is the same: 0 means no red, and 255 refers to the maximum perceivable red intensity, or the level of red that saturates the sensor. When we consider colors we talk about channels, e.g., the red channel, the blue channel, etc.

One of the ways to represent colors is to use the intensity level of each of the basic colors, which are Red, Green and Blue. Such a format is naturally denoted RGB. Any color can be seen as a mixture of any of these three (basic) colors, so this format is universal. See the illustration in fig. 1.1. Storing RGB images in a computer requires three matrices, one for each color channel.

A screen resolution of $800 \times 480$ colored pixels has 384,000 pixels, requiring $384,000 \times 3 = 1.152,000$ bytes of memory for storing the image (either on disk or display memory). If we consider HD resolution, we have $1280 \times 720 = 921,600$ pixels, amounting to 2,764,800 bytes. Full HD resolution is $1920 \times 1080 =$
2.073.600 pixels, corresponding to 6.220.800 bytes. 4K UHD resolution is 4096 \times 2160 = 8.847.360 pixels, amounting to 26.542.080 bytes. Finally, 8K UHD resolution is 7680 \times 4320 = 33.177.600, amounting to 99.532.800 bytes. And so on. If we think about photos, then we need these amounts of memory for each photo we want to store in computers. Ok, the number is huge, so we expect that each image is stored in a compact, or compressed, format. Most of the image formats store the pixel values in a compressed form. This means that we cannot interpret the data in an image file as a collection of explicit pixel values; we need software that understands those image formats in order to uncompress the images when we want to open (visualizing) them, and to compress them when we want to store them. JPEG and PNG are two (compressed) formats that are used in the experiments reported in this document.

Since we pursue a hands on type of approach, we will depart immediately to some practical examples. Our first program will be simple to the point of reading a color image from file, print a few characteristics of the image, accessing a single pixel, printing the corresponding RGB intensity values, and drawing the image on the screen. We present two ways of accessing the pixels in an image. Since this is our first code, we do not use openMP.
Chapter 2

Accessing pixels in an image

2.1 Introduction

We now go into some minimalist details about openCV. In fact, we will limit the use of openCV to functions that allow images in files to be loaded into memory, to write images back to files, and to show images on the screen.

The text will not present details about openMP; they were given elsewhere. The intent is to present an application environment where openMP programming can be trained, and at the same time introducing image processing by way of openCV.

2.2 Installation

First things first.

Get the latest version by downloading the zip file in [http://opencv.org/releases.html](http://opencv.org/releases.html) or through [https://github.com/opencv/opencv](https://github.com/opencv/opencv) to your home directory.

You may need to update

```bash
sudo apt-get update
```

and upgrade

```bash
sudo apt-get upgrade
```

your linux system. You may also have to install the developer tools

```bash
sudo apt-get install build-essential git cmake pkg-config
```

and a lot of libraries (google it!).

Unzip the opencv file and go to the created directory. Then execute the following commands:

```bash
mkdir release
```
cd release
cmake ../
make
sudo make install

After installation, in case you need to know the version currently installed, execute the following command in a terminal,

\texttt{pkg-config --modversion opencv}

If you want to know the location of the include files, execute the following command,

\texttt{pkg-config opencv --cflags}

For knowing the libraries location, execute,

\texttt{pkg-config opencv --libs}

To check the installed version, execute,

\texttt{pkg-config --modversion opencv}

To create the executable \texttt{exe} from the source \texttt{file.cpp}, execute,

\texttt{g++ -o exe file.cpp $(pkg-config opencv --cflags --libs)}

If you need a program to visualize images (png or jpeg) in linux, you can try \texttt{eog} or \texttt{edisplay}, which allows to visualize the values of individual pixels.

In case you need a program to open the video stream from an USB camera for visualization, you may use \texttt{cheese} (\texttt{sudo apt-get install cheese}).

If everything goes well, it is time to test opencv.

2.3 First way

Here’s our first program.

\begin{verbatim}
# include <iostream>
# include <opencv2/opencv.hpp>
using namespace std;
using namespace cv;
int main( int argc, char** argv )
{
    Mat image;
    Vec3b intensity;
    int nRows, nCols;
    int r, c;
    uchar red, green, blue;
    image = imread( argv[1], 1 );
    if( argc != 2 || !image.data ) {
        cout << "No image data" << endl;
        return -1;
    }
    // some facts about the image
    cout << "Dimension of the image array = " << image.dims << endl;
    cout << "Channels = " << image.channels() << endl;
}
\end{verbatim}
cout << "Rows = " << image.rows << endl;
cout << "Columns = " << image.cols << endl;
r = 10; // row 0
c = 100; // column 0
intensity = image.at<Vec3b>(r,c);
blue = intensity.val[0];
green = intensity.val[1];
red = intensity.val[2];
cout << "RBG(" << r << "," << c << ")=" << (int)red << "," << (int)green << "," << (int)blue << endl;
namedWindow( "Display Image", CV_WINDOW_AUTOSIZE );
imshow( "Display Image", image );
waitKey(0);
return 0;
}

The output of the program is,

Dimension of the image array = 2
Channels = 3
Rows = 600
Columns = 800
RBG(0,0)=98,65,24

Notice that the image filename is passed in the command line; it is available as argv[1]. The variable image.dims stores the dimension of the array of each channel. The dimension of the file is 73.078 bytes (compare with 800 × 600 × 3 = 1.440.000, which is about 19.7 times more; this gives us the compressing factor).

The construct image.at<Vec3b>(r,c) gives us the means to access each pixel, as long as 0 ≤ r ≤ 600 and 0 ≤ c ≤ 800. Note that the order of the colors inside the file is BGR (the opposite of the format name); see the illustration in fig. 2.1.

By the way, the color sensors inside our ordinal cameras have a sensor organization that do not follow the organization in fig. 2.1 instead, it follows the organization in fig. 2.2. Only high end cameras have a sensor organization similar to the one in fig. 2.1.

For the sake of completeness, we present our image in fig. 2.3.
2.4 Second way

The new program is,

```cpp
#include <iostream>
#include <opencv2/opencv.hpp>
using namespace std;
using namespace cv;
int main( int argc, char** argv )
{
    Mat image;
    const uchar* pel;
    int r,c;
```
uchar red, green, blue;
int nChannels;
image = imread( argv[1], 1 );
if( argc != 2 || !image.data ){
    cout << "No image data" << endl;
    return -1;
}
nChannels = image.channels();
r = 0; c = 0;
pel = image.ptr<uchar>(r);
blue = *(pel+c*nChannels+0);
green = *(pel+c*nChannels+1);
red = *(pel+c*nChannels+2);
cout<<"RBG("<<r<<","<<c<<") = "<<(int)red<<","<<(int)green<<","<<(int)blue<<endl;
namedWindow( "Display Image", CV_WINDOW_AUTOSIZE );
imshow( "Display Image", image );
waitKey(0);
return 0;
}

The variable pel points to row r. We must take into account that there are nChannels per pixel.
The program’s output is,
RBG(0,0)=98,65,24

2.5 Histogram

The histogram shows how the different pixel values of a RGB image are distributed among the possible values of intensity ([0, 255]).
The code to evaluate the histogram of a colored image is very simple. We start by defining the arrays which represent the bins (this is histogram lexicon).

int rh[256],gh[256],bh[256];
for (int i=0;i<256;i++)
rh[i]=gh[i]=bh[i]=0;

Counting the amount in each bin is done by passing through each pixel in the image.

for(r=0;r<im_rgb.rows;r++)
for(c=0;c<im_rgb.cols;c++) {
    intensity = im_rgb.at<Vec3b>(r,c);
    bh[intensity.val[0]]++;
    gh[intensity.val[1]]++;
    rh[intensity.val[2]]++;
}
Figure 2.4: Histogram of each color of the puzzle image in fig. 2.3. Note that
the scale of the vertical axis is logarithmic.

The histogram of our puzzle image (fig. 2.3) is presented in fig. 2.4. From
the histogram we conclude that there are no dark blues, dark greens, nor dark
red areas in the image.
Chapter 3

Pointwise image transformations

3.1 Gray scale

In this chapter we present some Pointwise image transformations, i.e., transformations that are applicable to each pixel independently of the pixels in its neighborhood. The first transformation is to produce a gray image from a colored one.

```cpp
#include <iostream>
#include <opencv2/opencv.hpp>
using namespace std;
using namespace cv;
int main( int argc, char** argv )
{
  const uchar* pel;
  Vec3b intensity;
  int nRows,nCols,nChannels;
  int r,c;
  uchar red, green, blue;
  Mat im_rgb = imread("puzzle.jpg");
  cout<<"Original image is "<<im_rgb.rows<<" x "<<im_rgb.cols<<" pixels
";
  Mat im_gray(Size(im_rgb.cols,im_rgb.rows),CV_8UC1);
  for(r=0;r<im_rgb.rows;r++)
    for(c=0;c<im_rgb.cols;c++) {
      intensity = im_rgb.at<Vec3b>(r,c);
      blue = intensity.val[0];
      green = intensity.val[1];
      red = intensity.val[2];
      im_gray.at<uchar>(r,c) = (int)((blue+green+red)/3);
    }
```

The declaration `Mat im_gray` is a constructor for variable `im_gray` of type `Mat`, which, among other actions, reserves space for an image with dimensions `im_rgb.cols \times im_rgb.rows \times CV_8UC1`, where the name `CV_8UC1` represents 8 bits/pixel and one channel.

We transformed a colored pixel in a gray one by averaging the RGB channels. There are other (more correct) ways to do this.

Note also that the program produces a file containing the gray image, which is presented in fig. 3.1.

### 3.2 The openMP way

We now change the last program by introducing some openMP directives and functions.

```cpp
#include <iostream>
#include <opencv2/opencv.hpp>
#include <omp.h>
using namespace std;
```
using namespace cv;
int main( int argc, char** argv )
{
    const uchar* pel;
    Vec3b intensity;
    int nRows,nCols,nChannels;
    int r,c;
    uchar red, green, blue;
    int ncores = omp_get_max_threads();
    cout << "This machine has " << ncores << " cores\n";
    omp_set_num_threads(ncores);
    Mat im_rgb = imread("puzzle.jpg");
    cout"Original image is " << im_rgb.rows << " x " << im_rgb.cols << " pels\n";
    Mat im_gray(Size(im_rgb.cols,im_rgb.rows),CV_8UC1);
    #pragma omp parallel for private(c,intensity,blue,red,green) shared(im_rgb)
    for(r=0;r<im_rgb.rows;r++)
    for(c=0;c<im_rgb.cols;c++) {
        intensity = im_rgb.at<Vec3b>(r,c);
        blue = intensity.val[0];
        green = intensity.val[1];
        red = intensity.val[2];
        im_gray.at<uchar>(r,c) = (int)((blue+green+red)/3);
    }
    imshow("color image", im_rgb);
    imshow("gray image", im_gray);
    imwrite("image_gray.jpg", im_gray);
    waitKey(0);
    return 0;
}

We start by determining the amount of cores in the computer, and determine
that any team of threads will have this quantity. This problem is very simple,
and so it does not require a great analysis effort regarding what parts can
be parallelized. We opted to execute the for cycle in parallel. Remember that,
by definition, r is private to each thread, hence it doesn’t appear in the private list.

The program produces an image like the one in fig. 3.1 and presents the
following text in the screen,

This machine has 2 cores
Original image is 600 x 800 pels

We now modify the program by substituting the line

#pragma omp parallel for private(c,intensity,blue,red,green) shared(im_rgb)

by
#pragma omp parallel for

The first time we run the program we get the same result. But at the second time we get the image in fig. 3.2. This is an example of what a race condition can provoke in an image. Remember that the memory model in openMP is shared memory, so variables $c$, etc. will be shared (or visible) by all threads in the team assembled in the parallel for. When some thread modifies the value of a shared variable, that modification is visible by the other threads in the team, and data corruption happens for sure. By using the `private` clause, private copies of the variables in the list are created, thus preventing interference to occur in the private copies.
Chapter 4

Image transformations based on pixel neighborhood

Many interesting transformations are made over a pixel based on the values of the pixels surrounding it. In other words, the new value of a given pixel is some function of its value and the values of the pixels in its neighborhood.

There are a few ways to define which pixels are neighbors. Figure 4.1 presents the two most used types of neighborhood.

When considering image transformations based on some notion of neighborhood, we can think of a small image centered around a pixel, \( P \), of the input image, \( I \), having the exact dimension such as all the pixels in the neighborhood of \( P \) are included; we call such an image a mask. The value of the pixels in the mask depend on the function we want to perform over \( P \). Let the coordinates of \( P \) in \( I \) be represented by \( I(i,j) \). The mask image is represented by \( T \); and \( T(i,j) \) thus represents the value of the pixel in the \((i,j)\) coordinates.

We consider that the mask \( T \) has dimension \( s \times s \). For reasons that will be clear shortly, we consider \( s \) to be and odd number. We may also use \( T \) to name the desired transformation we want to execute over \( I \); thus \( J = T(I) \), i.e., \( J \) results from applying transformation \( T \) over \( I \). It is also typical to call \( T \) a kernel or filter. The transformation of \( T \) over \( I \) is defined by eq. (4.1) which is also the definition of convolution.

\[
J(i,j) = \frac{\sum_{m=-\frac{s-1}{2}}^{\frac{s-1}{2}} \sum_{n=-\frac{s-1}{2}}^{\frac{s-1}{2}} T(m,n) \times I(i+m,j+n)}{\sum_{m=-\frac{s-1}{2}}^{\frac{s-1}{2}} \sum_{n=-\frac{s-1}{2}}^{\frac{s-1}{2}} T(m,n)}
\] (4.1)

Examples of transformations or kernels we will present are: linear/average kernel/filter, and gaussian kernel/filter. We will not be complete in what respects kernels.
Figure 4.1: Two types of neighborhood of a pixel, with two different sizes. The pixel under consideration has a yellow background, while the pixels in the neighborhood have a blue background. The first type can be defined as considering the pixels in the vertical and horizontal directions – figs. b) and d), while in the second the pixels in the diagonals also make part of the neighborhood – figs. a) and c). In figs. a) and b) the size of the neighborhood is 1, while in c) and d) the size is 2. The number in each cell represent the pixel intensity.

4.1 Linear kernel

A linear kernel of size 3 is defined by the following matrix,

\[
T = \begin{bmatrix}
1 & 1 & 1 \\
1 & 1 & 1 \\
1 & 1 & 1 \\
\end{bmatrix}
\]

Substituting this definition in eq. 4.1 we get eq. 4.2 which is the average of all the pixels of \(I\) inside the mask centered in \(P\).

\[
J(i, j) = \frac{\sum_{m=-1}^{1} \sum_{n=-1}^{1} I(i + m, j + n)}{9} \tag{4.2}
\]

The code that implements a Linear kernel with dimension 3 × 3 is presented next.

```cpp
Mat im_linear(Size(im_rgb.cols, im_rgb.rows), CV_8UC3);
for(r=1;r<im_rgb.rows-1;r++)
    for(c=1;c<im_rgb.cols-1;c++) {
        int rd,g,b;
        rd=g=b=0;
        for(int k=-1;k<2;k++)
            for(int l=-1;l<2;l++) {
                intensity = im_noisy.at<Vec3b>(r+k,c+l);
                b += intensity.val[0];
                g += intensity.val[1];
                rd += intensity.val[2];
            }
        im_linear.at<Vec3b>(r,c)[0] = (unsigned char)(b/9);
        im_linear.at<Vec3b>(r,c)[1] = (unsigned char)(g/9);
    }
```
Figure 4.2: Smoothing the puzzle image by a $3 \times 3$ Linear kernel.

```cpp
im_linear.at<Vec3b>(r,c)[2] = (unsigned char)(rd/9);
```

Figure 4.2 presents the result of smoothing the puzzle image by way of a $3 \times 3$ Linear kernel.

### 4.2 Gaussian kernel

A one-dimensional Gaussian function centered around 0 and having standard deviation $\sigma$ is defined by eq. (4.3).

$$f(x) = \frac{1}{\sigma \sqrt{2\pi}} \exp\left(\frac{-x^2}{2\sigma^2}\right)$$  \hspace{1cm} (4.3)

In the case of two dimensions, the Gaussian function centered around (0,0) and having standard deviation $(\sigma_x, \sigma_y)$ is defined by eq. (4.4) for an anisotropic filter (i.e., $\sigma_x \neq \sigma_y$), and by eq. (4.5) for the case of an isotropic filter (i.e., $\sigma_x = \sigma_y = \sigma$).

$$f(x, y) = f(x) \times f(y)$$

$$= \frac{1}{\sigma_x \sqrt{2\pi}} e^{\frac{-x^2}{2\sigma_x^2}} \times \frac{1}{\sigma_y \sqrt{2\pi}} e^{\frac{-y^2}{2\sigma_y^2}}$$

$$= \frac{1}{\sigma_x \sigma_y 2\pi} e^{\frac{-x^2}{2\sigma_x^2} - \frac{y^2}{2\sigma_y^2}}$$  \hspace{1cm} (4.4)

$$= \frac{1}{2\pi \sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$  \hspace{1cm} (4.5)
In the case of image filtering by Gaussian kernels we use the two dimensional symmetric Gaussian function. For example, if we consider \( \sigma = 1 \), then a kernel with dimension 5 has the following definition,

\[
K = \begin{bmatrix}
0.0029150 & 0.0130642 & 0.0215393 & 0.0130642 & 0.0029150 \\
0.0130642 & 0.0585498 & 0.0965324 & 0.0585498 & 0.0130642 \\
0.0215393 & 0.0965324 & 0.1591549 & 0.0965324 & 0.0215393 \\
0.0130642 & 0.0585498 & 0.0965324 & 0.0585498 & 0.0130642 \\
0.0029150 & 0.0130642 & 0.0215393 & 0.0130642 & 0.0029150 \\
\end{bmatrix}
\]

The sum of all elements is 0.98181. Since our kernel must sum to one, in order not to scale the image, we divide the matrix by this value, giving,

\[
K = \begin{bmatrix}
0.0029690 & 0.0133062 & 0.0219382 & 0.0133062 & 0.0029690 \\
0.0133062 & 0.0596343 & 0.0983203 & 0.0596343 & 0.0133062 \\
0.0219382 & 0.0983203 & 0.1621028 & 0.0983203 & 0.0219382 \\
0.0133062 & 0.0596343 & 0.0983203 & 0.0596343 & 0.0133062 \\
0.0029690 & 0.0133062 & 0.0219382 & 0.0133062 & 0.0029690 \\
\end{bmatrix}
\]

If we want to apply such a mask to images (whose pixel values are integers), we might want to have a matrix of integers also. We multiply each element of the matrix by such a constant in order to have the smaller element (i.e., (0,0)) equal to one. The matrix that we arrived at is,

\[
K = \begin{bmatrix}
1 & 6 & 10 & 6 & 1 \\
6 & 27 & 44 & 27 & 6 \\
10 & 44 & 73 & 44 & 10 \\
6 & 27 & 44 & 27 & 6 \\
1 & 6 & 10 & 6 & 1 \\
\end{bmatrix}
\]

whose normalizing constant is 449.

Figure 4.3 presents the result of smoothing the puzzle image using this Gaussian kernel. The C code that implements this transformation is presented next.

```c
Mat im_gaussian(Size(im_rgb.cols,im_rgb.rows),CV_8UC3);
unsigned char G[5][5]={{1,6,10,6,1},{6,27,44,27,6},{10,44,73,44,10},
{6,27,44,27,6},{1,6,10,6,1}};

// evaluate the normalizer
long normalizer = 0;
for(int k=0;k<5;k++)
    for(int l=0;l<5;l++) {
        normalizer += G[k][l];
    }
for(r=2;r<im_rgb.rows-2;r++)
    for(c=2;c<im_rgb.cols-2;c++) {
        //
    }
```


```cpp
long rd, g, b;
rd = g = b = 0;
for(int k=-2;k<3;k++)
    for(int l=-2;l<3;l++) {
        intensity = im_noisy.at<Vec3b>(r+k,c+l);
        b += intensity.val[0] * G[k+2][l+2];
        g += intensity.val[1] * G[k+2][l+2];
        rd += intensity.val[2] * G[k+2][l+2];
    }
    im_linear.at<Vec3b>(r,c)[0] = (unsigned char)(b/normalizer);
    im_linear.at<Vec3b>(r,c)[1] = (unsigned char)(g/normalizer);
    im_linear.at<Vec3b>(r,c)[2] = (unsigned char)(rd/normalizer);
}
```

### 4.3 Morphological operations - dilation and erosion

Morphological processing affects the shapes in an input image based on the size and shape of a certain notion of neighborhood (called the structuring element).

A structuring element has a shape, a size and a center pixel. For calculating the new value of a pixel, \( P \), in the input image, the center of the structuring element is made coincident with \( P \); then some (morphological) calculation takes place. The pixel values in the structuring element are 0 or 1 (i.e., a structuring element is a binary image), where 1 defines the neighborhood of the input image subject to the calculation. Clever choices of size and shape of the structuring
The most basic morphological operations are dilation and erosion. As the name implies, dilation makes pixels more prominent and erosion turns them less prominent, the amount being determined by the size of the structuring element.

Dilation is made by applying the max operator to all pixels in the neighborhood, while in erosion the min operator is applied. Let’s apply these two operators to the image in fig. 4.4.

The C code allowing to dilate an image is,

```c
int dim = 1;
Mat im_dilation(Size(im_rgb.cols,im_rgb.rows),CV_8UC3);
for(r=0+dim;r<im_rgb.rows-dim;r++)
    for(c=0+dim;c<im_rgb.cols-dim;c++) {
        int rm=0; int gm=0; int bm=0;
        for(int k=-dim;k<dim+1;k++)
            for(int l=-dim;l<dim+1;l++) {
                intensity = im_rgb.at<Vec3b>(r+k,c+l);
                bm = max(bm,intensity.val[0]);
                gm = max(gm,intensity.val[1]);
                rm = max(rm,intensity.val[2]);
            }
        im_dilation.at<Vec3b>(r,c)[0] = bm;
        im_dilation.at<Vec3b>(r,c)[1] = gm;
        im_dilation.at<Vec3b>(r,c)[2] = rm;
    }
```

The resulting image is presented in fig. 4.5.

The C code allowing to erode an image is,

```c
// dilation of an image. The filter is 3x3
int dim = 1;
Mat im_dilation(Size(im_rgb.cols,im_rgb.rows),CV_8UC3);
for(r=0+dim;r<im_rgb.rows-dim;r++)
    for(c=0+dim;c<im_rgb.cols-dim;c++) {
        int rm=0; int gm=0; int bm=0;
        for(int k=-dim;k<dim+1;k++)
            for(int l=-dim;l<dim+1;l++) {
                intensity = im_rgb.at<Vec3b>(r+k,c+l);
                bm = max(bm,intensity.val[0]);
                gm = max(gm,intensity.val[1]);
                rm = max(rm,intensity.val[2]);
            }
        im_dilation.at<Vec3b>(r,c)[0] = bm;
        im_dilation.at<Vec3b>(r,c)[1] = gm;
        im_dilation.at<Vec3b>(r,c)[2] = rm;
    }
```

The resulting image is presented in fig. 4.5.

The C code allowing to erode an image is,
dim = 1;
Mat im_erosion(Size(im_rgb.cols,im_rgb.rows),CV_8UC3);
for(r=0+dim;r<im_rgb.rows-dim;r++)
    for(c=dim;c<im_rgb.cols-dim;c++) {
        int rm=255; int gm=255; int bm=255;
        for(int k=-dim;k<dim+1;k++)
            for(int l=-dim;l<dim+1;l++) {
                intensity = im_rgb.at<Vec3b>(r+k,c+l);
                bm = min(bm,intensity.val[0]);
                gm = min(gm,intensity.val[1]);
                rm = min(rm,intensity.val[2]);
            }
        im_erosion.at<Vec3b>(r,c)[0] = bm;
        im_erosion.at<Vec3b>(r,c)[1] = gm;
        im_erosion.at<Vec3b>(r,c)[2] = rm;
    }

The resulting image is presented in fig. 4.6.

4.4 Morphological operations - image boundary

By subtracting the eroded version of an image of the image itself, we can extract
the image boundary. The C code is,

Mat im_boundary(Size(im_rgb.cols,im_rgb.rows),CV_8UC3);
for(r=2;r<im_rgb.rows-2;r++)
    for(c=2;c<im_rgb.cols-2;c++) {
        im_boundary.at<Vec3b>(r,c)[0] = im_rgb.at<Vec3b>(r,c)[0]-im_erosion.at<Vec3b>(r,c)[0];
        im_boundary.at<Vec3b>(r,c)[1] = im_rgb.at<Vec3b>(r,c)[1]-im_erosion.at<Vec3b>(r,c)[1];
        im_boundary.at<Vec3b>(r,c)[2] = im_rgb.at<Vec3b>(r,c)[2]-im_erosion.at<Vec3b>(r,c)[2];
Figure 4.6: Result of eroding the image in fig. 4.4.

Figure 4.7: Result of calculating the boundary of the image in fig. 4.7 using morphological operations.

Let’s apply this operation to the image in fig. 4.8, whose result is presented in fig. 4.9.
Figure 4.8: Image of a house.

Figure 4.9: Boundaries of the image in fig. 4.8.
4.5 Morphological operations - image boundary from a gray image, in openMP

Extracting the boundary of a gray image is the same as doing it over a colored one. However, doing it over a gray image is simpler, because the boundary is not dispersed over three images (red, green and blue), and it is faster too, because a gray image requires two times less data. We will extract the boundary of a gray version of the house and compare it with the colored one.

Here’s the C code.

```c
#include <stdio.h>
#include <opencv2/opencv.hpp>
#include <omp.h>
#define min(a,b) ((a)<(b))?(a):(b)
#define max(a,b) ((a)>(b))?(a):(b)
using namespace cv;

int main( int argc, char** argv )
{
    const uchar* pel;
    Vec3b intensity;
    int r,c,dim,aux;
    if(argc!=2) {
        printf("Usage: smoothing image-name.jpg\n");
        return 0;
    }
    aux = omp_get_max_threads();
    omp_set_num_threads(aux);
    printf("There are %d cores\n",aux);
    Mat im_rgb = imread(argv[1]);
    printf("%s has %d x %d pels\n",argv[1],im_rgb.rows,im_rgb.cols);
    // creation of gray image
    Mat im_gray(Size(im_rgb.cols,im_rgb.rows),CV_8UC1);
    #pragma omp parallel for private(r,c,aux,intensity) shared(im_gray)
    for(r=0;r<im_rgb.rows;r++)
        for(c=0;c<im_rgb.cols;c++) {
            intensity = im_rgb.at<Vec3b>(r,c);
            aux = (intensity.val[0]+intensity.val[1]+intensity.val[2])/3;
            im_gray.at<uchar>(r,c) = (unsigned char)aux;
        }
    // dilation of a gray image.
    dim = 1;
    Mat im_dilation(Size(im_rgb.cols,im_rgb.rows),CV_SUC1);
    #pragma omp parallel for private(r,c,aux) shared(im_dilation)
    for(r=0+dim;r<im_gray.rows-dim;r++)
        for(c=0+dim;c<im_gray.cols-dim;c++) {
            aux = 0;
        }
    return 0;
}
```
for(int k=-dim;k<dim+1;k++)
    for(int l=-dim;l<dim+1;l++)
        aux = max(aux,im_gray.at<uchar>(r+k,c+l));
    im_dilation.at<uchar>(r,c) = aux;
}

// erosion of a gray image.
dim = 1;
Mat im_erosion(Size(im_rgb.cols,im_rgb.rows),CV_8UC1);
#pragma omp parallel for private(r,c,aux) shared(im_erosion)
for(r=0+dim;r<im_gray.rows-dim;r++)
    for(c=0+dim;c<im_gray.cols-dim;c++) {
        aux = 255;
        for(int k=-dim;k<dim+1;k++)
            for(int l=-dim;l<dim+1;l++)
                aux = min(aux,im_gray.at<uchar>(r+k,c+l));
        im_erosion.at<uchar>(r,c) = aux;
    }

// extracting the boundary of an image.
Mat im_boundary(Size(im_rgb.cols,im_rgb.rows),CV_8UC1);
#pragma omp parallel for private(r,c,aux) shared(im_erosion)
for(r=2;r<im_rgb.rows-2;r++)
    for(c=2;c<im_rgb.cols-2;c++) {
        im_boundary.at<uchar>(r,c) = im_gray.at<uchar>(r,c)-im_erosion.at<uchar>(r,c);
    }

// parallelizing this code through parallel sections gives a runtime error,
// so we keep it serial, besides parallelizing I/O is potentially dangerous
imshow("original", im_rgb);
imshow("gray", im_gray);
imshow("dilation", im_dilation);
imshow("erosion", im_erosion);
imshow("boundary_gray", im_boundary);
imwrite("gray.jpg", im_gray);
imwrite("dilation.jpg", im_dilation);
imwrite("erosion.jpg", im_erosion);
imwrite("boundary.jpg", im_boundary);
printf("All's done\n");
waitKey(0);
return 0;
}

The resulting images are presented in figs. 4.10, 4.11 and 4.12.
Figure 4.10: Result of dilating a gray version of image in fig. 4.8.

Figure 4.11: Result of eroding a gray version of image in fig. 4.8.
Figure 4.12: Boundary of a gray version of image in fig. 4.8
4.6 Morphological operations - opening, closing and gradient

Opening an image is achieved by first eroding it and finally dilating it. It’s useful for removing small, bright, regions when in dark foregrounds.

Closing an image is achieved by reversing the operations, i.e., dilation before erosion. It’s useful for removing small, dark, objects when in bright foregrounds.

Finally, after learning how the boundary of an image is extracted, one might be curious about the result of subtracting an eroded version from a dilated version of an image. This operation is termed the morphological gradient of an image.

All these operators provide a good chance to make a program where openCV can be trained using openMP (the inverse is also true).

4.7 Edge detection – Sobel

Edge detection at pixel $P$ of image $I$ measures the amount of change in the intensity of pixels in some neighborhood of $P$, in some directions of interest. The result of edge detection is itself represented by an image, where the higher the pixel intensity at coordinates $(x, y)$ is, the higher is the change in $I(x, y)$ in the directions of interest. The directions of interest are typically horizontal and vertical, and also because most of the man-made artifacts have straight horizontal and/or vertical edges, and any diagonal can be described in terms of horizontal and vertical components.

The calculations allowing edges to be highlighted are based on pointwise matrix multiplication and convolution, just like we saw for filtering.

Examples of kernels for this purpose are,

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}$$

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & 2 & +1 \end{bmatrix}$$

Kernels having this structure are known as Sobel kernels (because each matrix evaluates the gradient in some direction and all its elements sum to one). Edge detection by way of Sobel kernels are calculated according to eq. 4.9, where $I_S(x, y)$ represents the subimage of $I$ centered in location $(x, y)$.

$$I_s(x, y) = \sqrt{(G_x I_S(x, y))^2 + (G_y I_S(x, y))^2} \quad (4.9)$$

The application of this kernel to some of our familiar images are presented in figs. 4.13, 4.14 and 4.15 where the calculations were applied to gray versions. As a suggestion to homework, in case you find these images too noisy, you might want to filter or erode them.
Figure 4.13: Edges of the small cartoon image as calculated by Sobel.

Figure 4.14: Edges of the puzzle image as calculated by Sobel.
Figure 4.15: Edges of the House image as calculated by Sobel.
Let's simplify our kernels, and find the edges in each direction separately. Let's start by the kernel

\[ G_y = \begin{bmatrix} -1 & +1 \end{bmatrix} \]

which finds vertical edges. The result is presented in fig. 4.16. The kernel that finds horizontal edges is,

\[ G_x = \begin{bmatrix} -1 \\ +1 \end{bmatrix} \]

The result is presented in fig. 4.17. The combination of both kernels through eq. 4.9 is presented in fig. 4.18.

### 4.8 Pattern recognition

In pattern recognition the goal is to identify the location of a template image in a given image. There are many ways of doing this, but the most obvious is to convolute the template with the input image, which is a brute force method.

As a suggestion for some training (over openMP and openCV), we can use the puzzle image (fig. 2.3) to extract a template; for instance a yellow cube, just like in fig. 4.19.

It would improve the performance of the program if both images are converted to gray scale, and the respective edges extracted; finally the convolution operator can be applied. Meantime, one can try to denoise the images (of the edges) and also try to improve their contrast in order to increase the chances of recognizing all the cubes on the puzzle (the red, blue and green ones, too).
Figure 4.17: The horizontal edges of the House.

Figure 4.18: Both horizontal and vertical edges of the house image calculated by kernel combination.

Figure 4.19: The template image to be searched in the puzzle image.
Bibliography


