Chapter 2

The tools of the trade: a brief tutorial on OpenCV using C++

2.1 OpenCV

The library to be used for the exercises and assignments in this paper is called *OpenCV*. It was developed by researchers working for Intel [1, 2]. The library is freely available (with the source code) and it works in different platforms (`http://opencv.org/downloads.html`).

In this chapter, we present a very brief OpenCV introduction and show simple examples. Using these examples students can easily complete the assignments proposed during the semester. Students are encouraged to explore the OpenCV library on their own and to experiment using different approaches for their assignments.

The basic documentation for OpenCV is available online (`http://docs.opencv.org/`). OpenCV allows programmers to use either C or C++. The C API is slowly being discontinued, so it is better to use the new C++ API. In this laboratory the latest OpenCV version 3 is installed.

2.2 Getting started in the CV Laboratory

In the CV laboratory there are 18 Linux machines with all the pre-requisites installed. OpenCV contains libraries that can be linked directly by the GCC compiler. This section explains how to create, compile and run a simple OpenCV program for those who never used the Linux environment before.

The first example program should open an image and show it on an OpenCV window. For our purposes we need to:

- edit a file *.c or *.cpp. You can use any editor available , e.g. CodeBlocks.
- include the appropriate files and compile the program.
- run the executable, using OpenCV to open and show images.

Every program using OpenCV needs to have certain *.h (or *.hpp) files included, so the programs find the definitions for functions, prototypes etc. Usually the generic `#include <opencv2/opencv.hpp>` can be used. For specific parts of the OpenCV library, refer to the documentation. In the examples below separate `*.hpp` files were used (`core` for the basic structures, `highgui` for the GUI, and `improc` for image processing functions.)

Listing 2.1: Your first OpenCV program using the C++ API

```
#include <iostream>
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>

using namespace std;
using namespace cv;

int main(int argc, char** argv)
{
  //if there is no argument, exit
  if (argc != 2) { cout << "needs 1 argument, e.g. image.jpg" <<
     endl; exit(0);}
  //create a window to show images
  namedWindow("Windowname1",0);
  //create a Mat instance, it can load an image
  Mat image1;
  //load the file name in the argument onto 'image'
  image1=imread(argv[1]);
  //show the window with the image1
  imshow("Windowname1", image1);
  //hold the image until the user presses any key
  waitKey(0);
  return 0;
}
```

Running the program in listing 2.1 should show a window similar to figure 2.1.

Figure 2.1: imshow(): note the name on top of the window is the name given on namedWindow() function. The second parameter works as follows: 0 creates a resizeable window; 1 creates a fixed size window.

2.2.1 Compiling and Running OpenCV programs in C++

One can compile a program in OpenCV straight from the command line. If the source file is, for example, *program_name.cpp* and you want an executable called *executable_name*, we should issue the command:

```
g++ -o executable_name program_name.cpp
```

However, this will trigger error messages because you did not tell the compiler which libraries you want to link to. We need to link with the OpenCV shared libraries. The problem is that the libraries might be in different locations (directories) in different operating systems. Normally, to indicate where the header files (include) are you use `-I{path}` and `-L{path}` options for the libraries' location. In order to find the path in certain Linux distributions, you can use `pkg-config --libs --cflags opencv`. After informing gcc about the location of the libraries, one needs to link the specific libraries. For the example above, you only need `opencv_core` and `opencv_highgui`:

```
g++ -o executable_name program_name.cpp -lopencv_highgui -lopencv_core -lopencv_imgcodecs
```

or, if the *path* for the include directory and library directory are needed:

```
g++ -o executable_name program_name.c -I /usr/local/include/opencv/
-L /usr/local/lib/ -lopencv_highgui -lopencv_core -lopencv_imgcodecs
```

or yet another form:

```
g++ -o executable_name program_name.c `pkg-config --libs --cflags opencv`
```

In order to run the program from command line:

```
./executable_name
```

2.2.2 Using codeblocks

Another option is to use codeblocks, open a project and compile and run using push-buttons.

Firstly, create a project for console (figure 2.2.a). Modify the *include* search path for the relevant directories for OpenCV (figure 2.2.b). (from the menu `Project`, then `Build options...`)

Modify the *library* search path for the relevant directories for OpenCV (figure 2.3.a). Explicitly tell which libraries to link to (figure 2.3.b). (from the menu `Project`, then `Build options...`, then `Linker settings`)

Press the button to build then the button to run. Note that if you run the first example (listing 2.1), a message comes up (`needs 1 argument, e.g. image.jpg`). The program is expecting arguments from command line. In CodeBlocks you can set arguments from the menu `Project` then `Set program's arguments...`. The arguments can be set to Debug or Release modes, and should be entered with spaces between the arguments, just as one would do in command line mode. If the argument is an image, remember to put the correct path (the default path is to have the image in the same directory of the Debug executable under the CodeBlocks project directory).

2.2.3 Image data structure

In its first versions OpenCV used a data structure called `IplImage`. It is worth knowing this because there are code examples using the old structures and API. An `IplImage` should be always declared as a pointer. For example to declare an image called *myimage* use:

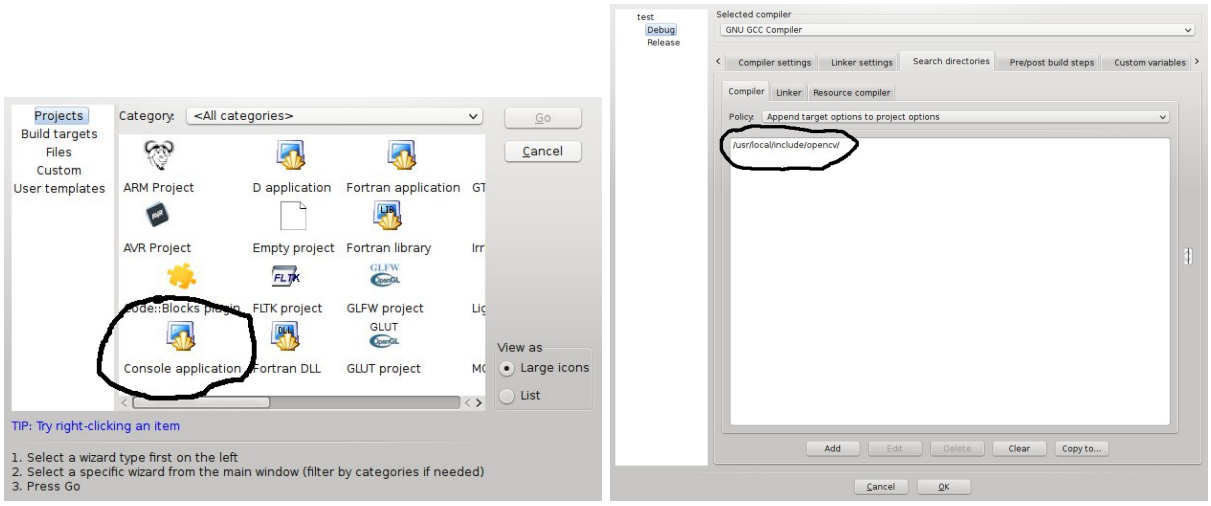

Figure 2.2: CodeBlocks: create a project.

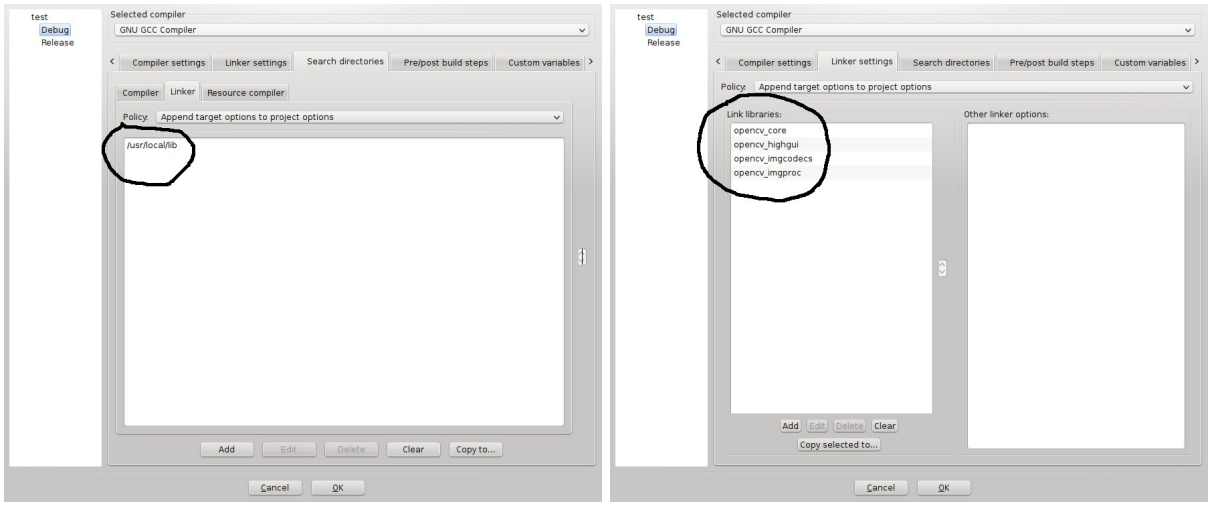

Figure 2.3: CodeBlocks: modify the shared libraries path.

```
IplImage *myimage;
```

The Image structure definition is:

```
class CV_EXPORTS Mat
{
public:
    // ... a lot of methods ...
    ...

    /*! includes several bit-fields:
         - the magic signature
         - continuity flag
         - depth
         - number of channels
```

```
     */
    int flags;
    //! the array dimensionality, >= 2
    int dims;
    //! the number of rows and columns or (-1, -1) when the array has more than 2 dimensions
    int rows, cols;
    //! pointer to the data
    uchar* data;

    //! pointer to the reference counter;
    // when array points to user-allocated data, the pointer is NULL
    int* refcount;

    // other members
    ...
};
```

The new image class (from OpenCV version 2 onwards) has lots of methods to initialise and allocate space to an image. It may be confusing to convert from `IplImage` to the new `Mat` format. In `Mat` the `rows` and `cols` are "inverted", i.e., `rows` represent the height of the image, and `cols` represent the width of the image, so be careful when using `Mat.rows` and `Mat.cols`. You can find more details about accessing parameters from *Mat* in the OpenCV reference guide.

For example, to create an image 100x100 greyscale:

```
Mat myimage;
myimage.create(100,100, CV_8UC1);
```

`IPL_DEPTH_8U` is used for grey-scale images. Each pixel is represented by one `uchar` (8 bits), an integer between 0 and 255.

Note that Mat can also be used as a generic matrix, with many Depth options:

```
CV_8UC1 //unsigned 8-bit integers
CV_8UC3 //unsigned 8-bit integers, three channels (RGB)
CV_8SC1 //signed 8-bit integers
CV_16SC2 //signed 16-bit integers with two channels, e.g. used as a complex number matrix
CV_32SC1 //signed 32-bit integers
CV_32FC1 //single precision floating-point numbers
CV_64FC1 //double precision floating-point numbers
```

Constructors can also be used:

```
Mat myimage(100,100,CV_8C1, 0);//single channel image filled with zeros
```

Image files in OpenCV are automatically identified by their *magical numbers* (each file type has its own header). Now lets see an example where a 3 channel RGB image is loaded onto one Mat, and converted to greyscale onto a different Mat.

Listing 2.2: Converting a colour image into greyscale.

```
#include <iostream>
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>

using namespace std;
using namespace cv;

int main(int argc, char** argv)
{
  //if there is no argument, exit
  if (argc != 2) { cout << "needs 1 argument, e.g. image.jpg" <<
     endl; exit(0);}
  //create windows to show images
  namedWindow("Color",0);
  namedWindow("Grey",0);
  //create Mat instances
  Mat imagecolor, imagegrey;
  //load the file name in the argument onto 'image'
  imagecolor=imread(argv[1]);
  //convert to greyscale
  cvtColor(imagecolor, imagegrey, CV_BGR2GRAY);
  //show the windows
  imshow("Color", imagecolor);
  imshow("Grey", imagegrey);
  //hold the image until the user presses any key
  waitKey(0);
  //write the image into a file
  imwrite("grey.jpg", imagegrey);
  cout << "color image channels " << imagecolor.channels() <<endl;
  cout << "grey image channels " << imagegrey.channels() << endl;
  return 0;
}
```

You can verify what happened with the saved image ("grey.jpg") opening it with the previous sample application (see listing 2.1). You can also check the number of channels of each image using `image.channels()` method. Note that program 2.2 needs one additional include file (`imgproc.hpp`) and library to be linked (`-lopencv_imgproc`)

2.3 Accessing pixels

So far, you learnt how to open, show and save images in OpenCV. One needs to access individual pixels of the Mat for different purposes. The OpenCV documentation includes different ways of accessing these pixels. One has to be extremely careful when accessing pixels from a Mat class because of the subtleties of the internal pixel structures. In all cases, one needs to know what type of image or matrix is being accessed: the pixel type has to be identified properly, otherwise the values might be completely wrong.

2.3.1 the `at` method

This is the method presented in the OpenCV official documentation. It is simple to use, but the syntax is not so friendly.

If image (Mat) `A` is created with a single channel, then the method for accessing a pixel at `(x,y)` is:

`A.at<uchar>(y,x)`

Listing 2.3: Using the `at` method for greyscale images.

```
#include <iostream>
#include "opencv2/opencv.hpp"
using namespace cv;

int main( int argc, char** argv )
{
  Mat image, image2;
    if (argc != 2) exit(0);
    image = imread( argv[1], 0);//0 to transform to greyscale
    image2.create(image.size(),CV_8UC1);
    for (int x=0;x<image.cols;x++){
          for (int y=0;y<image.rows;y++){
            if(x == y){
              image2.at<uchar>(y,x)=255;
            }
            else{
              image2.at<uchar>(y,x)=image.at<uchar>(y,x);
            }
          }
    }
    namedWindow( "SourceImage", CV_WINDOW_AUTOSIZE );
    imshow( "SourceImage", image );
    namedWindow( "DestImage", CV_WINDOW_AUTOSIZE );
    imshow( "DestImage", image2 );
    waitKey(0);
}
```

If the image is a 3 channel images, then the method syntax is:
`A.at<Vec3b>(y,x)`
and the pixel maps into a 3d vector (predefined in OpenCV as Vec3b).

Listing 2.4: Using the `at` method for colour images.

```
#include "opencv2/opencv.hpp"
using namespace cv;

int main( int argc, char** argv )
{
  Mat image, image2;
    if (argc != 2) exit(0);
    image = imread( argv[1], 1);
```

```
9       image2.create(image.size(),CV_8UC3);
10      for (int x=0;x<image.cols;x++){
11              for (int y=0;y<image.rows;y++){
12                Vec3b pixel_image = image.at<Vec3b>(y,x);
13                Vec3b pixel_image2 = image2.at<Vec3b>(y,x);
14                if(x == y){
15                  pixel_image2[0]=255;
16                  pixel_image2[1]=255;
17                  pixel_image2[2]=255;
18                }
19                else{
20                  pixel_image2[0]=pixel_image[0];
21                  pixel_image2[1]=pixel_image[1];
22                  pixel_image2[2]=pixel_image[2];
23                }
24                image2.at<Vec3b>(y,x)=pixel_image2;//copy back!
25              }
26      }
27      namedWindow( "SourceImage", CV_WINDOW_AUTOSIZE );
28      imshow( "SourceImage", image );
29      namedWindow( "DestImage", CV_WINDOW_AUTOSIZE );
30      imshow( "DestImage", image2 );
31      waitKey(0);
32  }
```

2.3.2 Accessing Pixels via a Macro

Now you will learn how to read and modify individual pixels by accessing them through a macro. Macros are quite a good solution because they are pre-compiled, so the execution is as fast as passing by reference. You can combine speed with syntax elegance. The macros can be included in your own *.h file. See the next examples for greyscale and colour images respectively (listings 2.5 and 2.6):

Listing 2.5: Using simple macros to access MAT greyscale pixels

```
1  #include <iostream>
2  #include "opencv2/opencv.hpp"
3  using namespace cv;
4
5  #define Mpixel(image,x,y) ( (uchar *) ( ((image).data) + (y)*((
       image).step) ) ) [(x)]
6
7  int main( int argc, char** argv )
8  {
9    Mat image, image2;
10     if (argc != 2) exit(0);
11     image = imread( argv[1], 0);//0 to transform into greyscale
12     image2.create(image.size(),CV_8UC1);
13     for (int x=0;x<image.cols;x++){
```

```
            for (int y=0;y<image.rows;y++){
              if(x == y){
                Mpixel(image2,x,y)=255;
              }
              else{
                Mpixel(image2,x,y)=Mpixel(image,x,y);
              }
            }
    }
    namedWindow( "SourceImage", CV_WINDOW_AUTOSIZE );
    imshow( "SourceImage", image );
    namedWindow( "DestImage", CV_WINDOW_AUTOSIZE );
    imshow( "DestImage", image2 );
    waitKey(0);
}
```

Listing 2.6: Using simple macros to access MAT colour pixels

```
#include "opencv2/opencv.hpp"
using namespace cv;

#define MpixelB(image,x,y) ( (uchar *) ( ((image).data) + (y)*((image).step) ) ) [(x)*((image).channels())]
#define MpixelG(image,x,y) ( (uchar *) ( ((image).data) + (y)*((image).step) ) ) [(x)*((image).channels())+1]
#define MpixelR(image,x,y) ( (uchar *) ( ((image).data) + (y)*((image).step) ) ) [(x)*((image).channels())+2]

int main( int argc, char** argv )
{
  Mat image, image2;
    if (argc != 2) exit(0);
    image = imread( argv[1], 1);
    image2.create(image.size(),CV_8UC3);
    for (int x=0;x<image.cols;x++){
            for (int y=0;y<image.rows;y++){
              if(x == y){
                MpixelB(image2,x,y)=255;
                MpixelG(image2,x,y)=255;
                MpixelR(image2,x,y)=255;
              }
              else{
                MpixelB(image2,x,y)=MpixelB(image,x,y);
                MpixelG(image2,x,y)=MpixelG(image,x,y);
                MpixelR(image2,x,y)=MpixelR(image,x,y);
              }
            }
    }
    namedWindow( "SourceImage", CV_WINDOW_AUTOSIZE );
```

```
29      imshow( "SourceImage", image );
30      namedWindow( "DestImage", CV_WINDOW_AUTOSIZE );
31      imshow( "DestImage", image2 );
32      waitKey(0);
33  }
```

All of the four listings above will open the image and draw a diagonal line (for pixels where x=y) on the copy of the image. Figure 2.4 shows an example using peppers.jpg (a standard images used in the literature).

Figure 2.4: Accessing pixels to draw a diagonal.

2.4 Simple GUI

OpenCV offers simple trackbars and simple mouse callback functions. For more sophisticate GUI, you should use specialised libraries (e.g. QT).

2.4.1 Trackbars

A trackbar is always created after (and it belongs to) a window (listing 2.7). Note that there is a reference to a callback function (`callback_trackbar_click`) and a markerclick variable (`markerclick`). The callback function is used to catch the changes via the variable markerclick and show the image again with the modifications. For example, suppose that you want to change the brightness of the image. One could write the callback function like this:

Listing 2.7: Using trackbars

```
1  #include "opencv2/opencv.hpp"
2  using namespace cv;
3  using namespace std;
4  //macro for Mat structures
5  #define Mpixel(image,x,y) ( (uchar *) ( ((image).data) + (y)*((
       image).step) ) ) [(x)]
6  Mat image1;
7
8  void callback_trackbar(int value, void * object){
```

```
  Mat image2;
  image2.create(image1.size(),CV_8UC1);
  cvtColor(image1, image2, CV_BGR2GRAY);
  for(int a=0;a<image2.rows;a++){
    for(int b=0;b<image2.cols;b++){
      int newpixel= Mpixel(image2,b,a);
      newpixel=newpixel+value-127;
      if(newpixel>255) newpixel=255;
      if(newpixel<0) newpixel=0;
      Mpixel(image2,b,a)=newpixel;
    }
  }
  imshow("Show results",image2);
}

int main( int argc, char** argv )
{
  image1 = imread( argv[1]);
  int markerclick=0;
  namedWindow("Show results",0);
  createTrackbar( "switch", "Show results", &markerclick, 255,
      callback_trackbar);
  setTrackbarPos("switch","Show results",128);
  imshow("Show results",image1);
  waitKey(0);
}
```

An example is shown in figure 2.5.

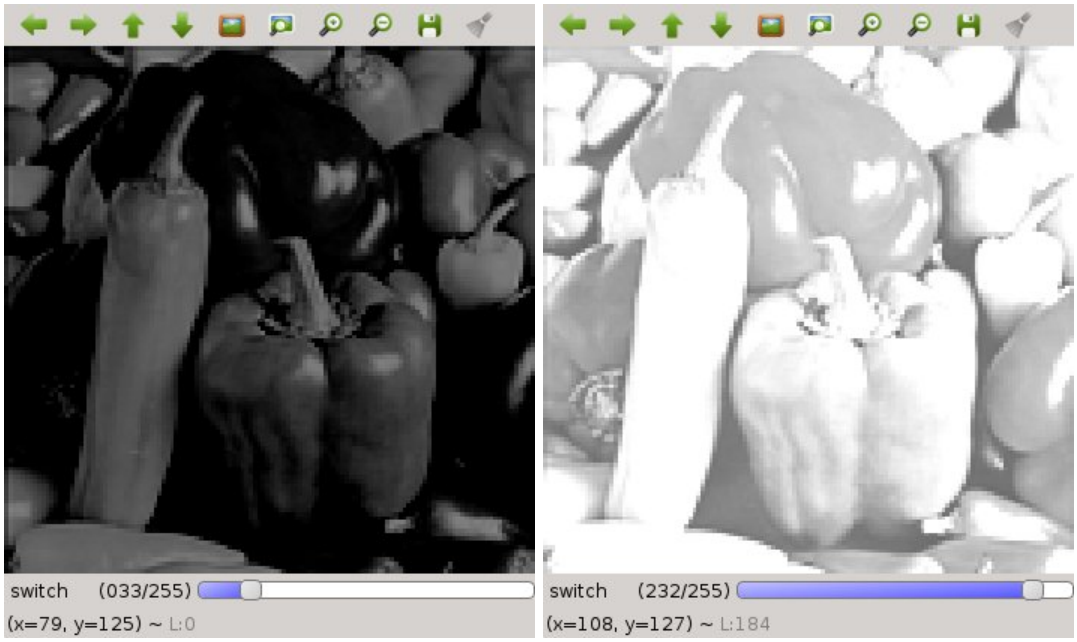

Figure 2.5: Using a trackbar to change the brightness of a greyscale image.

2.4.2 Mouse callbacks

Mouse callbacks are also relatively simple to use. The listing 2.8 is an application that shows a region from the image in a 10x10 grid, and if the left click is pressed it overwrites the image with a white pixel where the cursor is. Note that in order to refresh the image after the mouse actions one needs to call `imshow()` again, with the appropriate window name and image name.

Listing 2.8: Using simple mouse actions

```
#include "opencv2/opencv.hpp"
using namespace cv;
using namespace std;
Mat image1;
#define MpixelB(image,x,y) ( (uchar *) ( ((image).data) + (y)*((
    image).step) ) ) [(x)*((image).channels())]
#define MpixelG(image,x,y) ( (uchar *) ( ((image).data) + (y)*((
    image).step) ) ) [(x)*((image).channels())+1]
#define MpixelR(image,x,y) ( (uchar *) ( ((image).data) + (y)*((
    image).step) ) ) [(x)*((image).channels())+2]
void on_mouse_example(int event,int x,int y,int flag, void* obj)
{
  int calcx, calcy;
  if(x > 10 && x < image1.cols-10) calcx=x;
  else if(x < 10) calcx=10; else calcx=image1.cols-10;
  if(y > 10 && y < image1.rows-10) calcy=y;
  else if(y < 10) calcy=10; else calcy=image1.rows-10;
  Mat imgroi = image1(Rect(calcx,calcy,10,10));
  cout << "position " << x << " " << y << endl;
  //imgroi.repeat(50,50);
  namedWindow("Pixel view",0);
  imshow("Pixel view",imgroi);
  if(event==EVENT_LBUTTONDOWN)//left click
  {
    cout << "clicked left button at " << x << " " << y << endl;
    pixelR(image1,x,y)=255;
    pixelG(image1,x,y)=255;
    pixelB(image1,x,y)=255;
    imshow("Show results",image1);
  }
}

int main( int argc, char** argv )
{
  image1 = imread( argv[1]);
  namedWindow("Show results",0);
  setMouseCallback("Show results", on_mouse_example, NULL);
  imshow("Show results",image1);
  waitKey(0);
  return 0;
}
```

An example is shown in figure 2.6.

Figure 2.6: Using the mouse callback to draw a single pixel in the image.

2.5 Linear Algebra and OpenCV

Since version 3 there were improvements in the way OpenCV treats matrices and vectors. It is now very similar in syntax to MatLab and Octave (and other similar tools). Many operators are now overloaded, and one can even print a matrix using `cout`. Lets see some specific examples on how to create matrices and vectors, and how to operate with them.

2.5.1 Creating Matrices and Vectors

For small matrices, one can use the template `cv::Matx`, with very efficient implementations for small fixed matrices. This study guide only covers larger matrices in the form of the class `cv::Mat`.

We have already seen ways of declaring and instantiating a Mat. For example, create a matrix containing chars, and initialise every element to zero:

Listing 2.9: Basic matrix constructor usage.

```
1 Mat A(100,100,CV_8C1,0);
```

One can also create matrices preloaded with other values, and even with every element pre-specified. See listing 2.10 for examples.

Listing 2.10: Initialising matrices with constants.

```
1 //a 10x10 matrix initialised to 5.0 (floating point elements)
2 Mat A(10,10,CV_32F, Scalar(5));
3
4 Mat B(10,10,CV_32F);//same as above, with the Scalar
5 B = Scalar(5); //after the instantiation
6
7 //a 10x10 matrix initialised to 1 (uchar)
8 Mat C = Mat::ones(10,10,CV_8U);
9
```

```
10 //a 10x10 matrix initialised to 0 (uchar)
11 Mat D = Mat::zeros(10,10,CV_8U);
```

If the matrices need to be initialised with pre-set values, then there are specific forms of the constructor to do that as well. It is also easy to print `Mat` using `cout` (see examples in the listings below).

Listing 2.11: Initialising matrices with pre-set values.

```
1 float a[2][3] = {{1,2,3}, {4,5,6}};
2 Mat A = Mat(2, 3, CV_32FC1, a);
3 cout << "A = " << A << endl;
```

```
A = [1, 2, 3;
 4, 5, 6]
```

Listing 2.12: Initialising a 2x2 identity matrix.

```
1 Mat IDENTITY = (Mat_<float>(2,2) << 1,0,0,1);
2 cout << "IDENTITY = " << IDENTITY << endl;
```

```
IDENTITY = [1, 0;
 0, 1]
```

2.5.2 Matrix Operations

In this version of OpenCV several operators are overloaded. There are also methods such as `add()`, `subtract()`, `multiply()` etc.

Listing 2.13: Operations with Matrices using overloading.

```
1 float a[2][3] = {{1,2,3}, {4,5,6}};
2 float b[2][3] = {{7,8,9}, {0,1,2}};
3 float c[3][2] = {{7,8},{9,0},{1,2}};
4 Mat A = Mat(2, 3, CV_32FC1, a);
5 Mat B = Mat(2, 3, CV_32FC1, b);
6 Mat C = Mat(3, 2, CV_32FC1, c);
7 Mat RES;
8 RES = A + B;
9 cout << "A + B = " << RES << endl << endl;
10 RES = A * C;
11 cout << "A * C = " << RES << endl << endl;
```

```
A + B = [8, 10, 12;
 4, 6, 8]

A * C = [28, 14;
 79, 44]
```

There are also statistical operations: `sum()`, `mean()`, `countNonZero()` etc. Other typical operations are `transpose()`, `invert()` and `determinant()`. Some examples are shown in listing 2.14.

Just to refresh some operations, the transpose of a matrix is:

$$\begin{vmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{vmatrix}^T = \begin{vmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{vmatrix} \tag{2.1}$$

The determinant of a 2x2 matrix is $det = ad - bc$ for a matrix $\begin{vmatrix} a & b \\ c & d \end{vmatrix}$

The inverse of a matrix is another matrix that multiplied by the original one results in the identity matrix, i.e.: $A \times A^{-1} = A^{-1} \times A = I$.

For example, the matrix $\begin{vmatrix} 1 & 2 \\ 2 & 5 \end{vmatrix}$ inverse is $\begin{vmatrix} 5 & -2 \\ -2 & 1 \end{vmatrix}$, if you multiply them you get $\begin{vmatrix} 1 & 0 \\ 0 & 1 \end{vmatrix}$.

Listing 2.14: Other matrix operations.

```
transpose(A,RES);
cout << "Transpose = " << RES << endl << endl;

float d[2][2] = {{1,2}, {2,5}};
Mat D = Mat(2, 2, CV_32FC1, d);
float det=determinant(D);
cout << "Determinant = " << det <<endl<<endl;

Mat INV;
invert(D,INV);
cout << "INV = " << INV << endl << endl;
//Check inversion:
RES = D * INV;
cout << "RES = " << RES << endl << endl;
```

```
Transpose = [1, 4;
 2, 5;
 3, 6]

Determinant = 1

INV = [5, -2;
 -2, 1]

RES = [1, 0;
 0, 1]
```

One operation programmers have to be careful about is the computation of *eigenvalues* and *eigenvectors*. The matrix must be symmetric (and consequently, square), otherwise the results are just wrong with no warning (OpenCV's numerical solution has limitations). Also, one must be aware of rounding (and types used in the matrices), as errors may accumulate. See the example below using matrix D defined above.

Recall the definition of eigenvalues and eigenvectors. Simplifying a bit: given a square matrix A, an eigenvector is a column vector such that multiplied by the matrix results in the scaled vector.

$$A\mathbf{v} = \lambda\mathbf{v} \Leftrightarrow (A - \lambda I)\mathbf{v} = \mathbf{0} \tag{2.2}$$

Actually, any nonzero solution is a valid one, so usually we have a whole family (multiples) that can fit the solution. For small matrices there are also algebraic solutions. That is usually the reason why different sw packages will give a different answers for the eigenvectors.

Let's compute the eigenvalues of the matrix $\begin{vmatrix} 1 & 2 \\ 2 & 5 \end{vmatrix}$. We start by assuming that the matrix is nonsingular (the determinant is nonzero). Therefore, if the determinant of $(A - \lambda I)\mathbf{v}$ is zero, we can find values for the eigenvalues λ.

$$\begin{vmatrix} 1 & 2 \\ 2 & 5 \end{vmatrix} - \lambda I = \begin{vmatrix} 1 & 2 \\ 2 & 5 \end{vmatrix} - \begin{vmatrix} \lambda & 0 \\ 0 & \lambda \end{vmatrix} = \begin{vmatrix} 1-\lambda & 2 \\ 2 & 5-\lambda \end{vmatrix}$$

So, $(1-\lambda)(5-\lambda) - 4 = 0$, and we get the quadratic equation $\lambda^2 - 6\lambda + 1 = 0$. The roots of this equation are: 5.828 and 0.172, which are the eigenvalues for the matrix.

Once the eigenvalues are known, it is just a matter of solving the left side of equation 2.2. This is going to be equivalent to a system of linear equations. Let's get the first eigenvalue (5.828):

$$\begin{vmatrix} 1 & 2 \\ 2 & 5 \end{vmatrix} \begin{vmatrix} e_1 \\ e_2 \end{vmatrix} = 5.828. \begin{vmatrix} e_1 \\ e_2 \end{vmatrix}$$

The two linear equations are: $\begin{cases} e_1 + 2e_2 = 5.828e_1 \\ 2e_1 + 5e_2 = 5.828e_2 \end{cases}$

Therefore, $e_2 = 2.414e_1$, so any eigenvector that has this ratio between e_1 and e_2 is a possible solution.

Listing 2.15: Eigenvalues and eigenvectors.

```
1  Mat EIGENVA;
2  Mat EIGENVE;
3  eigen(D,EIGENVA, EIGENVE);
4  cout << "D = " << D << endl << endl;
5  cout<<"det of D = "<< determinant(D)<<endl<<endl;
6  cout << "EIGENVA = " << EIGENVA << endl << endl;
7  cout << "EIGENVE = " << EIGENVE << endl << endl;
8
9  //Verify eigenvectors
10 Mat RES2;
11 Mat IDENTITY = (Mat_<float>(2,2) << 1,0,0,1);
12 Mat X = (Mat_<float>(2,1) << EIGENVE.at<float
       >(0,0),EIGENVE.at<float>(1,0));
13 cout << "X = " << X << endl << endl;
14 RES2 = (D - EIGENVA.at<float>(0)*IDENTITY)*X;
15 cout << "RES2 = " << RES2 << endl << endl;
```

```
D = [1, 2;
 2, 5]

det of D = 1

EIGENVA = [5.8284273;
 0.17157292]

EIGENVE = [0.38268343,
     0.9238795;
 0.9238795,
     -0.38268343]

X = [0.38268343;
 0.9238795]

RES2 = [-1.1920929e
    -07;
 -1.7881393e-07]
```

We can check that the eigenvalues and eigenvector were computed correctly using the right side of equation 2.2. In listing 2.15 both elements of vector RES2 should be zero. Due to rounding up, it is a small nonzero number instead. The type of the elements may improve these results (e.g., using `double`), but they will always be limited by the floating point precision.

2.6 A practical example: object labelling

In this section a simple object labelling (aka "blob" finder) is discussed. The approach uses the adjacency concept discussed in section 1.2. This approach is very useful to count object in an image after carrying out some form of segmentation, which we will discuss later. The algorithm discussed below assumes that the input image is binary, i.e., that the pixels can only be two values, either zero or not zero. The pixels that are different than zero belong to a certain object, and the ones with value zero belong to the background. In later sections we will discuss how to get binary images.

Humans are very good at the task of counting objects. When using computers, however, one needs to consider that they can only process one pixel at a time. Therefore, one needs to use specific data structures to deal with the problem, such as simple sets that holds information about the pixels that are of a certain value in the binary image.

The most obvious way to join pixels that belong to the same object is to check the adjacency. Every time a pixel is checked for value (say, 255 if it belongs to a "blob", 0 if not), we should also check if any of the connected pixels belong to an existing set, i.e., if any neighbouring pixels with value 255 belong to a set. If they do, we simply copy the current pixel to that set. Otherwise we create a new set with a single pixel and continue to scan the image.

Areas of the image that are connected areas may still end up in different sets. A second pass is needed, in order to coalesce any sets that have connected pixels. See the example in figure 2.7.

It is possible to combine the two passes by coalescing existing sets as pixels are being checked,

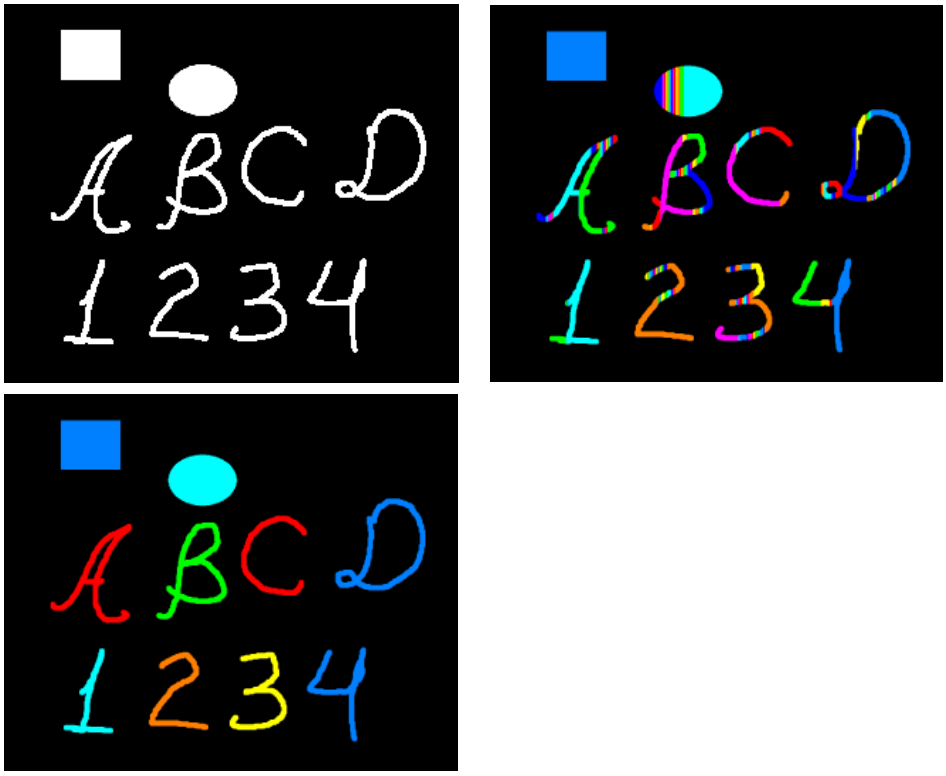

Figure 2.7: Object labelling. a) binary image b) phase one c) coalesced sets

but this needs an appropriate data structure and a careful analysis of the adjacency. Because the scanning will follow a certain order (left to right, top down), we only need to check for 4 pixels (if using 8-adjacency), one on the left and three on top of the pixel in question. The other 4 pixels would still not belong to any set because they would not have been examined yet at this point.

For the 4-adjacency only two pixels need to be checked, the top pixel and the left pixel. Algorithm 1 shows a simplified version of a 4-adjacency labelling. Sets are created and coalesced as the algorithm progresses through the scanning. This also allows to read all the points that belong to a certain blob very efficiently, so one can colour the blobs or compute their centre.

A complete example of the workings of algorithm 1 is shown in figure 2.8. The figure shows that when the sets are coalesced (via a Union operation), the index of the set that is emptied can no longer be used. This is a drawback but it is possible to fix that with a clever reshuffle of the sets. The matrix `A(x,y)` keeps track of the index of each set. At the end of the algorithm, one can tell how many blobs there are by counting the number of sets that are not empty.

2.6.1 Finding the centre of the labelled objects

Once a blob is found and each pixel can be identified, it is easy to compute the centre of mass. The mean of the coordinates of all the points (pixels) that belong to a set (object) is the coordinate of the centre:

$$x_{centre} = \frac{\sum_{i=0}^{n} x_i}{n} \qquad y_{centre} = \frac{\sum_{i=0}^{n} y_i}{n} \tag{2.3}$$

These equations need to be used with care. They work reasonably well for a convex figure, but for a concave shape the centre may be located outside the boundaries of the object itself.

Algorithm 1 ***Object Labelling using 4-adjacency***

Require: a **binary** image $i(x, y)$

declare a vector of sets $SET[\,]$
declare integers $counter = -1$, $s1$, $s2$
declare $A[i.width][i.height]$ {a 2D vector or mat initialised to -1}
for $y = 1$ to $i.height$ **do**
  for $x = 1$ to $i.width$ **do**
    if ($i(x, y) \neq 0$) **then**
      if ($i(x-1, y) \neq 0$ OR $i(x, y-1) \neq 0$) **then**
        $s1 = A[x-1][y]$
        $s2 = A[x][y-1]$
        if ($s1 \neq -1$) **then**
          $i(x, y) \rightarrow SET[s1]$ {insert point $i(x, y)$ into $SET[s1]$}
          $A[x][y] = s1$
        end if
        if ($s2 \neq -1$) **then**
          $i(x, y) \rightarrow SET[s2]$
          $A[x][y] = s2$
        end if
        if (($s1 \neq s2$) AND ($s1 \neq -1$) AND ($s2 \neq -1$)) **then**
          $SET[s1] \cup SET[s2]$ {Union, keep set $SET[s1]$ and empty the other}
          Reset all points of $A(x, y)$ belonging to $SET[s2]$ to $s1$
          empty $SET[s2]$
        end if
      else
        $counter = counter + 1$
        Create new set $SET[counter]$
        $i(x, y) \rightarrow SET[counter]$
        $A[x][y] = counter$
      end if
    end if
  end for
end for

2.7 Exercises

2.7.1 Exercise 1

1. Write a program using OpenCV that opens two images and shows them in the screen in sequence (one at a time).

2. Write a program to copy an image and mirror it. Show the original image and the mirrored image simultaneously (open two windows).

3. Modify the program to copy the image to a grey-scale image. Save the grey-scale image with a different file type than the original one.

4. Write a program that loads a colour image. Using a trackbar, the user should be able to change the image to grey-scale and back to colour (the trackbar should allow for values 0 and 1).

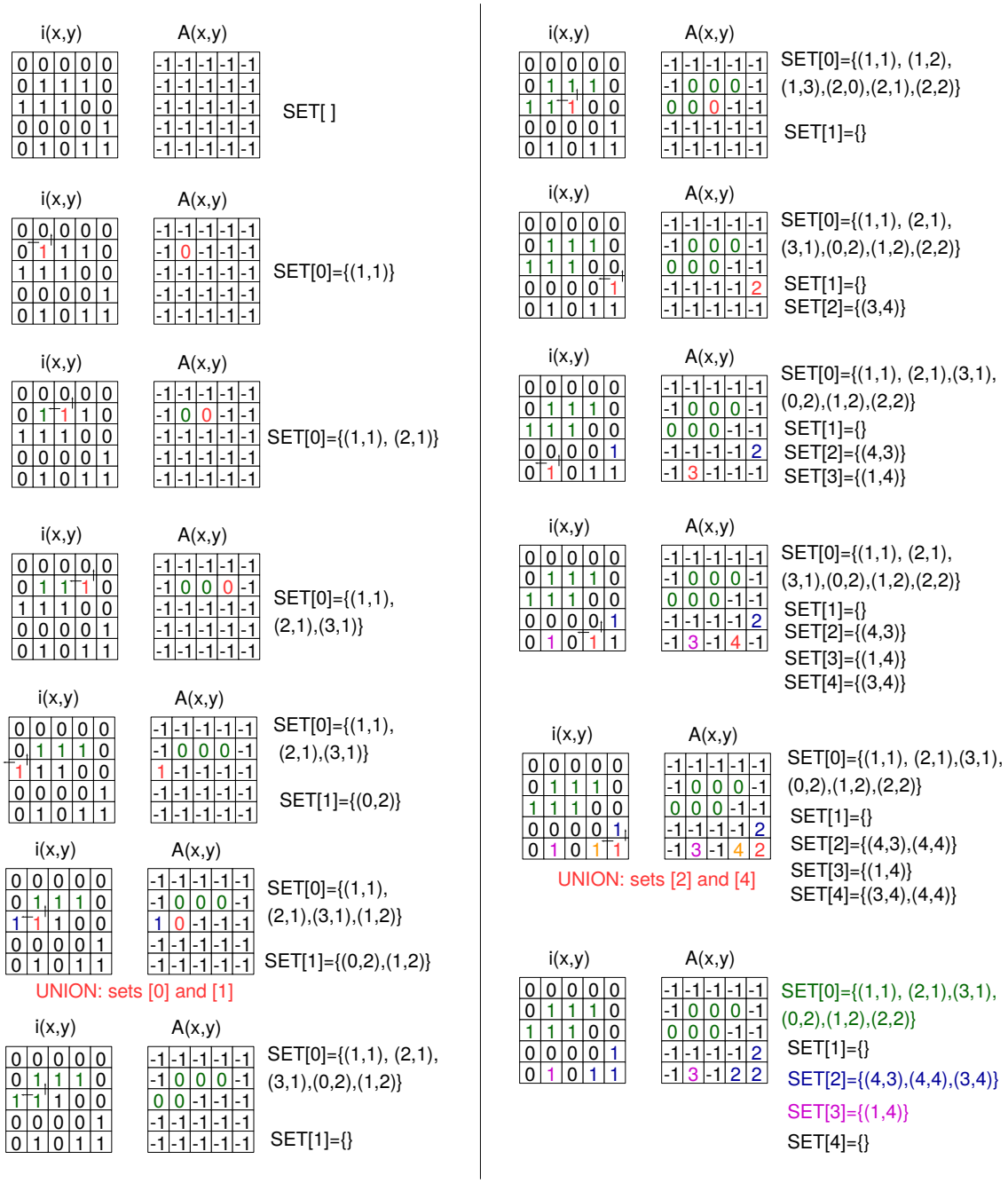

Figure 2.8: Blob finder example.

Bibliography

[1] G. Bradski and A. Kaehler, *Learning OpenCV*. o'Reilly, 2008.

[2] A. Kaehler and G. Bradski, *Learning OpenCV 3*. o'Reilly, 2017.