

Vicknesh Selvam

An Introduction to the CImg Library

A project conducted with the aid of Professor Todd Wittman

Chapter 1: Introduction to the CImg Library

The CImg library is an open source toolkit designed by David Tschumperlé for image processing. It can be used across many platforms, including Windows, OS X, Unix and BSD. For the purposes of this project, we used Visual Studio .NET 2003 in Windows XP.

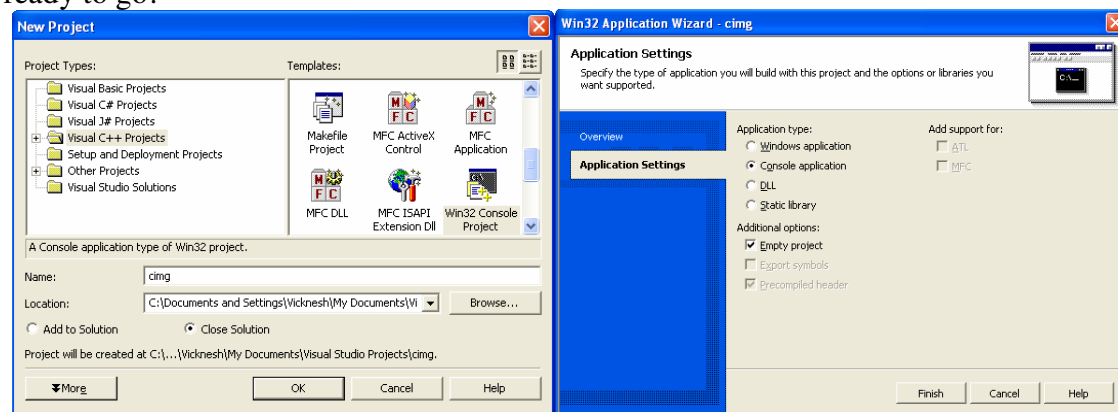
The library is highly portable and is stored in a single .h header file, which is about 1MB in size. The library itself contains several useful algorithms, such as loading, saving, and displaying images with templated pixel types. Typically, we will use double as the pixel type.

To download the CImg Library, go to <http://cimg.sourceforge.net/>, click download, and click "download the latest stable package." The cimg.h file should be in the same project folder as your other source file(s).

A regular color image, the type we shall see in these examples, consists of essentially 3 overlapping grayscale images: Red, Blue, and Green. Each grayscale image, of length x and width y , is a matrix of x rows and y columns. Typically, the value of each element is an number (int or double) between 0 and 255 (8 bits). Essentially, the lower the number, the darker each color would be. For example, $\{0,255,0\}$ is pure blue, $\{100,0,100\}$ is dark yellow, and $\{200,200,200\}$ is light grey. Thus a regular image consists of 3 dimensions, x, y and n , where x is the length, y is the width and n is the number of colors (3). The CImg library allows for 3 dimensional volumes (dimension z). For this tutorial and because it is difficult to display a 3D image on a 2D screen accurately, we shall set z to be 1 for all 2D images.

Opening a project:

The CImg library is built on a console project. To create a new project, from the Visual Studio Start Page, click New Project, Choose Console Project, type in a suitable name, and click OK. Then click Application Settings and check 'Empty project' and click 'Finish'. Add your cimg.h to the header files, create a driver cpp file and you're ready to go!



Chapter 2: A Simple Example -- Making An Image Darker

Now that you know how to install the library, let me show you how it works. We shall use a simple example of making an image darker by cutting its intensity in half. The code for such a program is as follows:

Darkening an image	
<code>#include "CImg.h"</code>	1
<code>using namespace cimg_library;</code>	2
<code>int main(){</code>	3
<code> CImg<double> image("starwars.bmp");</code>	4
<code> CImgDisplay main_disp(image, "Image", 0);</code>	5
<code> CImg<double> darkimage(image.dimx(), image.dimy(), 1, 3, 0);</code>	6
<code> for (int i=0; i<image.dimx(); i++)</code>	7
<code> for (int j=0; j<image.dimy(); j++)</code>	8
<code> for (int k=0; k<3; k++)</code>	9
<code> darkimage(i, j, 0, k)=image(i, j, 0, k)/2;</code>	10
<code> CImgDisplay dark_disp (darkimage, "Dark Image", 0);</code>	11
<code> while (!main_disp.is_closed)</code>	12
<code> main_disp.wait();</code>	13
<code> return 0;</code>	14
<code>}</code>	

Here is an explanation of the program:

Line 1: Includes the CImg library.

Line 2: The namespace used in this library is the `cimg_library` namespace. If you wish to perform regular console applications, like `cout`(to output pixel values, for example), you should use the `std::` function.

Line 4: Constructor which opens the image “starwars.bmp”. The image is templated to double. Unsigned int and chars are also allowed, most other data types are infeasible though.

Lines 5,11: Display function, takes in (in this order) the object name, the output at the top of the screen, and the normalization key. The normalization key 0 is defined as no normalization, so the display is 8-bit with 0 as black and 255 as white. There is a default normalization 1, so the minimum pixel value is black and the maximum is pure white. To use this min-max range, display the image with:

```
CImgDisplay main_disp(image, "Image");
```

For this example we wish to see the picture with its 8-bit intensity, therefore we choose normalization 0.

Line 6: Constructs a new image, ‘darkimage’ with dimensions of image, with all values 0. The x, y, z and v(number of colors) axes can be accessed by `dimx()`, `dimy()`, `dimz()` or `dimv()`, member functions, all returning integers. In general, though, `dimz` is 1 and `dimv` is 3. The last entry is the default value of all the pixels in the matrix. Without it, the line `CImg<double> darkimage(image.dimx(), image.dimy(), 1, 3);` would create a matrix of random values.

Lines 7-9: Creates 3 for loops, to access each value of the darkimage matrix, and set it to half the value of the corresponding value in the image matrix.

Line 10: `image(x,y,z,v)` is an accessor, and a mutator, for the pixel at position `(x,y,z,v)`. Note that the count for each starts at 0, as they are for most data structures.

Lines 12-13: `'is.closed'` is a boolean which checks if the window has been closed. The `wait()` function waits for a keystroke or a mouseclick. This function prevents the program from ending immediately after processing, allowing you to view the fruits of your labor.



Now that you know how to use the library, we will move on to some actual image processing...

Chapter 3: A Harder Example

Now that we have figured out how to load an image, change its values, and display it, we will attempt to perform a 2D convolution. A convolution is basically a pointwise multiplication of a matrix onto another. For example, suppose our convolution kernel K is the 3x3 identity matrix. Suppose the 3x3 neighborhood of pixel x in our image A is the values 1-9, in raster order. Then the result of the convolution $K*A$ at x would give $K*A(x) = 15$, as shown below.

$$K = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

$$K * A(x) = 1*1 + 0*2 + 0*3 + 0*4 + 1*5 + 0*6 + 0*7 + 1*9 = 15$$

Convolutions are very useful, especially for denoising. We will create a function that performs 2D convolution, but first we will describe how to create a kernel matrix.

We will create an array of defined elements with size $N \times N$, where N is odd. In this example, we will use $N=3$. Then we will input the matrix using our 'inputkernel' function in the source file 'cimgextra.cpp'.

```
double entries[9]={0,1,0,1,-4,1,0,1,0};  
CImg<double> kern= inputkernel(entries,3);
```

The inputkernel function takes in an array and the dimension N . Note that all 3 colors will contain the same filter matrix. I have also created another function, 'normalize' which takes in a matrix, and ensures that the total of the values for each filter (across each color) has a sum of 1.

```
normalize(kern);
```

To perform the convolution we created the function convolve, which temporarily creates a matrix with expanded borders to account for filtering the boundary. The function returns a filtered image, the same size as the original image specified in the first parameter.

```
CImgDisplay main_disp(convolve(image,kern),"Convolution",0);
```

As the process is a series of nested for loops, we will not show you the entire code here. However, the basic process is as follows:

1) The `expandmatrix` function creates new matrix, size slightly larger than the image, with the border pixels extrapolated. An extra border of $(N-1)/2$ pixels is sufficient for a kernel of degree N .

```
CImg<double> expand=expandmatrix(image,(degree-1)/2);
```

2) Create a new empty image and, for each value, fill it with a convolution of the matrix and the kernel about that value.

```
For all values i,j,k (sizes of image's x,y and v dimensions)
    for all values l and m (dimensions of kern)
        filtered(i,j,0,k)+=(expand(i+l,j+m,0,k))*kernel(l,m,0,k);
```

The code below and its output gives another use for the convolution.

Convolution main function for a mean filter

```
int main() {
    CImg<double> image("starwars1.bmp");
    int dim=5;
    CImg<double> kern(dim,dim,1,3,2); //The last value can be any
    //positive number , since we're normalizing
    normalize(kern);
    CImgDisplay main_disp(image, "Image", 0);
    CImgDisplay main_conv(convolute(image,kern), "Meanfilter 5", 0);
    while (!main_disp.is_closed) {
        main_disp.wait();
    }
    return 0;
}
```

This creates a mean filter of dimension 5. The dimension can be changed in this example by simply changing the value of dim in line 3.

Performing the convolution without normalizing would result in a very very bright image.



Chapter 4 – Denoising

Now that we are comfortable with convolutions, we shall move on to denoising. Denoising involves the ‘smoothing out’ of an image, removing noise and thus clearing up an image. In this chapter, we will use the following matrix:

$$L = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

This is also known as the Laplacian matrix, or the sum of the 2nd derivatives of the image in the x and y directions. Denoising by isotropic diffusion evolves the image u according to the heat equation

$$\frac{\partial u}{\partial t} = \Delta u$$

Discretizing and solving for u gives










$$u^{n+1} = u^n + \Delta t [D_x^+(D_x^-u) + D_y^+(D_y^-u)]$$

is the Δt is the time step and D^+ and D^- refer to the forward and backward finite differences, respectively. Iterating this equation would result in a smoother, but blurred image.

For this example, we shall set $\Delta t = 0.25$ and look at the changes in the image, up to $T=40$ iterations. The code is as follows:

```
for (double t=0;t<10;t+=dt) {
    newu = newu + dt*(DoubleDx(newu) + DoubleDy(newu) );
}
```

Where newu is the image matrix, and DoubleDx and DoubleDy are defined functions to approximate the 2nd derivatives.

T	0	1.25	2.5	3.75	5
Image					
T	6.25	7.5	8.75	10	
Image					

Notice that the image becomes more blurred as T increases. There is an alternate method, called anisotropic diffusion, that slows the diffusion at points where there is a large gradient, such as at an edge. The result resembles a cartoon image.

In this case, we modify the basic heat equation by introducing an “edge-stopping function” $g(u)$

$$\frac{\partial u}{\partial t} = \nabla \cdot (g(u) \nabla u)$$









The function $g(u)$ should take on values close to 0 near edges and 1 in smooth regions. We chose the classical Perona-Malik function

$$g(u) = \frac{1}{\sqrt{1 + |\nabla u|^2}}$$

Discretizing and solving for u again gives

$$u^{n+1} = u^n + \Delta t \left[D_x^+ (g(u) D_x^- u) + D_y^+ (g(u) D_y^- u) \right]$$

The results of anisotropic diffusion, using the same variables as in the isotropic case, are as follows:

T	0	1.25	2.5	3.75	5
Image					
T	6.25	7.5	8.75	10	
Image					

The effect on this picture appears to remove the stubble on the man's face, which resembles image noise.