Table of Contents

The OpenCV Library

OpenCV provides utilities for reading and writing a large variety of image types, video sequences, and cameras. These are part of `HighGUI` which is included in the OpenCV package.

First Programs

The first program we need to write is that which reads an image from file and displays it on the screen:

```
#include <opencv2/opencv.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv/cvaux.h>

int main( int argc, char** argv )
{
  IplImage* img = cvLoadImage( argv[1] );
  cvNamedWindow("Example1", CV_WINDOW_AUTOSIZE );
  cvShowImage("Example1", img );
  cvWaitKey(0);
  cvReleaseImage( &img );
  cvDestroyWindow("Example1");
}
```

As a second program, we can use OpenCV to read in an AVI video and play it on the screen:

```
#include <opencv2/opencv.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv/cvaux.h>
```

```
int main( int argc, char** argv ) {
    cvNamedWindow( "Example2", CV_WINDOW_AUTOSIZE );
    //CvCapture* capture = cvCaptureFromAVI( argv[1] ); // either one will work
    CvCapture* capture = cvCreateFileCapture( argv[1] );
    IplImage* frame;
    while(1) {
        frame = cvQueryFrame( capture );
        if( !frame ) break;
        cvShowImage( "Example2", frame );
        char c = cvWaitKey(33);
        if( c == 27 ) break;
    }
    cvReleaseCapture( &capture );
    cvDestroyWindow( "Example2" );
}
```

Let's examine some of the code:

- CvCapture* capture = cvCreateFileCapture(argv[1]); This function takes the filename of the video to be loaded and returns a pointer to a `CvCapture` structure which contains all of the information about the AVI file being read.
- frame = cvQueryFrame(capture); Takes the next video frame into memory and returns a pointer to that frame.
- cvReleaseCapture(&capture); Releases the memory allocated for the frames of the video.

Gaussian Convolution

We now write a program to perform Gaussian smoothing of images. This type of smoothing is useful as a preprocessing step in order to reduce the noise content of images, as it attenuates high frequencies within the image.

```
#include <opencv2/opencv.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv/cvaux.h>


void example2_4( IplImage* image )
{
    // Create some windows to show the input
    // and output images in.
```

```
    //
    cvNamedWindow( "Example2_4-in", CV_WINDOW_AUTOSIZE );
    cvNamedWindow( "Example2_4-out", CV_WINDOW_AUTOSIZE );

    // Create a window to show our input image
    //
    cvShowImage( "Example2_4-in", image );

    // Create an image to hold the smoothed output
    //
    IplImage* out = cvCreateImage(
        cvGetSize(image),
        IPL_DEPTH_8U,
        3
    );

    // Do the smoothing
    //
    cvSmooth( image, out, CV_GAUSSIAN, 5, 5 );

    // Show the smoothed image in the output window
    //
    cvShowImage( "Example2_4-out", out );

    // Be tidy
    //
    cvReleaseImage( &out );

    // Wait for the user to hit a key, then clean up the windows
    //
    cvWaitKey( 0 );
    cvDestroyWindow("Example2_4-in" );
    cvDestroyWindow("Example2_4-out" );

}

int main( int argc, char** argv )
{
  IplImage* img = cvLoadImage( argv[1] );
  cvNamedWindow("Example1", CV_WINDOW_AUTOSIZE );
  cvShowImage("Example1", img );
  example2_4( img );
//  cvWaitKey(0);
  cvReleaseImage( &img );
```

```
  cvDestroyWindow("Example1");
}
```

This program performs Gaussian smoothing by convolving a Gaussian kernel with the image. A Gaussian function with unit area in 2D may be written as

$$G(\vec{x};\sigma)=\frac{1}{2\pi\sigma^2}\exp\left\{\frac{-x^2+y^2}{2\sigma^2}\right\}$$

The convolution operation is denoted as $I(\vec{x})*G(\vec{x};\sigma)$ and, in the continuous domain, is a double integral of the form:

$$\int\int I(x-u,y-v)G(u,v;\sigma)\,du\,dv$$

where u and v vary over the pre-determined extent of the kernel function G. Its discrete version is a double summation over the extent of the discrete kernel (or mask) G:

$$\sum\sum I(x-u,y-v)G(u,v;\sigma)$$

Convolution can be an expensive operation to apply to an image if the kernel is not a separable function. A function is separable if $G(x,y)=G(x)G(y)$. The 2D Gaussian is a separable function:

$$G(\vec{x};\sigma) = \frac{1}{2\pi\sigma^2}\exp\left\{\frac{-x^2+y^2}{2\sigma^2}\right\} = g(x;\sigma)g(y;\sigma) = \frac{1}{\sqrt{2\pi}\sigma}\exp\left\{\frac{-x^2}{2\sigma^2}\right\}\times\frac{1}{\sqrt{2\pi}\sigma}\exp\left\{\frac{y^2}{2\sigma^2}\right\}$$

and the 2D convolution can be performed as series of 1D convolutions with 1D kernels. Because of Gaussian separability, convolution in the continuous domain can be rewritten as

$$\int G(u)\left[\int G(v)I(x-u,y-v)G(u)\,du\right]dv$$

And similarly for the discrete case:

$$\sum_u G(u)\left[\sum_v G(v)I(x-u,y-v)\right]$$

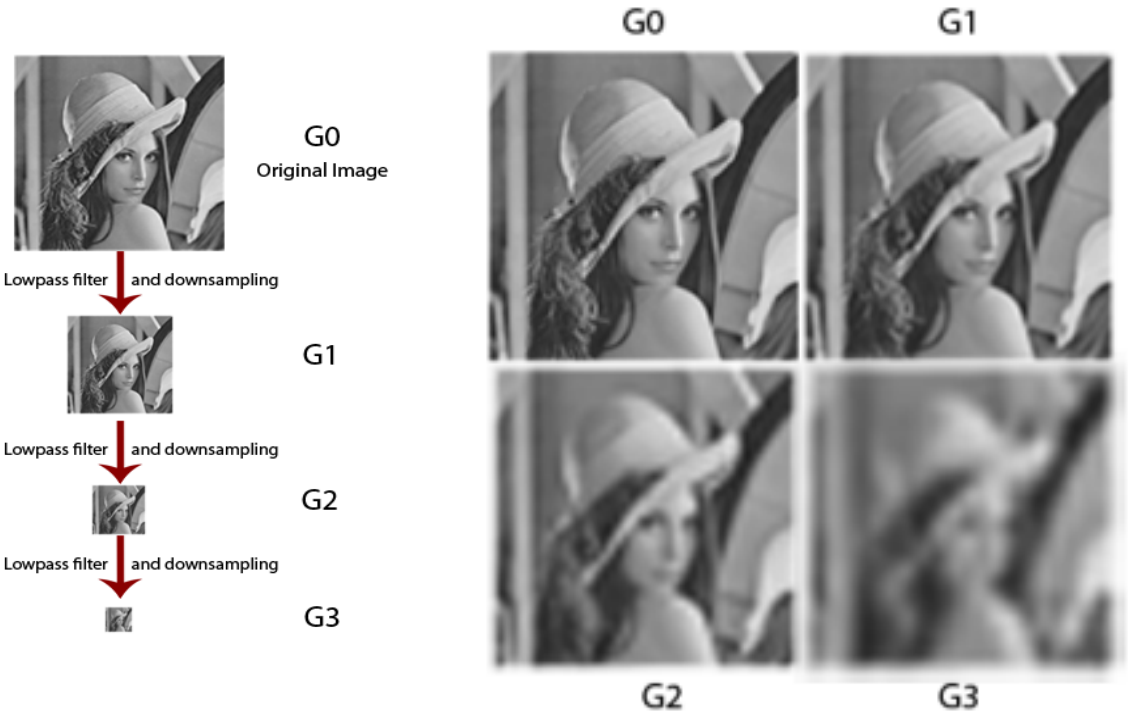
One order of magnitude in computational time is saved by this technique.

Gaussian Pyramids

Consider the reduction in size of an image by a factor of 2. This is a common and important in a wide variety of computer vision algorithm. The function `cvPyrDown` performs a Gaussian convolution of an image and creates a sized down image by removing every other row/column of the convolved image. Here is a program that performs this:

```
#include <opencv2/opencv.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv/cvaux.h>

IplImage* doPyrDown(
  IplImage* in,
  int       filter = IPL_GAUSSIAN_5x5)
{

    // Best to make sure input image is divisible by two.
    //
    assert( in->width%2 == 0 && in->height%2 == 0 );

    IplImage* out = cvCreateImage(
        cvSize( in->width/2, in->height/2 ),
        in->depth,
        in->nChannels
    );
    cvPyrDown( in, out );
    return( out );
};

int main( int argc, char** argv )
{
  IplImage* img = cvLoadImage( argv[1] );
  IplImage* img2 = cvCreateImage( cvSize( img->width/2,img->height/2 ), img->depth, img-
>nChannels);
  cvNamedWindow("Example1", CV_WINDOW_AUTOSIZE );
  cvNamedWindow("Example2", CV_WINDOW_AUTOSIZE );
  cvShowImage("Example1", img );
  img2 = doPyrDown( img );
  cvShowImage("Example2", img2 );
  cvWaitKey(0);
  cvReleaseImage( &img );
  cvReleaseImage( &img2 );
  cvDestroyWindow("Example1");
  cvDestroyWindow("Example2");
}
```

If repeated several times, one obtains a series of scaled down images that contain less and less high-frequency structures.



Acquisition Noise

Two images taken by the same camera and of the same scene under the same conditions are never exactly identical. We have come accustomed to regard these noisy variations as probabilistic noise, which we may model with random variables. Estimating the amount of noise in a sequence of n images of the same scene is usually accomplished with computing pixel-wise averages through the frames of the sequence:

$$\bar{I}(x,y)=\frac{1}{n}\sum_{k=0}^{n-1} I_k(x,y)$$

followed by determining the standard deviation at each pixel:

$$\sigma(x,y)=\left(\frac{1}{n}\sum_{k=0}^{n-1}\left(\bar{I}(x,y)-I_k(x,y)\right)^2\right)^{\frac{1}{2}}$$

This way, we can obtain an estimate of the acquisition noise that a particular visual sensor generates. However, in the general case, pixel values are correlated (CCD temperature, light-wave diffusion, etc). To obtain estimates of this correlation, an auto covariance measure may be used.

Image Noise

Image noise is defined as spurious and somewhat random fluctuations of pixel values. A common assumption is that noise is additive and random:

$$\hat{I}(x,y)=I(x,y)+n(x,y)$$

Different types of noise are countered by different techniques. However it is important to note that what is considered noise for one computer vision task might constitute useful information for another.

The amount of noise in imagery is often quantified by a signal-to-noise ratio, such as:

$$SNR=\frac{\sigma_s}{\sigma_n}$$

Alternatively, the signal-to-noise ratio may be expressed in decibels:

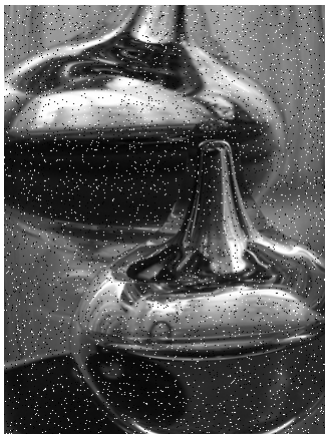
$$10\log_{10}\left(\frac{\sigma_s}{\sigma_n}\right)$$

In general, an additive model for noise is adequate for most computer vision systems.

Gaussian Noise

The Gaussian model for noise is the most widely used in the absence of noise information. In this case, we assume that $n(x,y)$ is a white, zero-mean Gaussian stochastic process. That is to say: for each pixel $I(x,y)$ there is a random variable $n(x,y)$ whose behavior is Gaussian.

Impulsive Noise



Impulsive noise, as opposed to Gaussian noise, alters pixels randomly, in such a way as to make their values very different from what they should be. A form of impulsive noise is salt-and-pepper noise:

$$\hat{I}(h,k)=\begin{cases} I(h,k) & \text{if} \quad x<l \\ i_{min}+y(i_{max}-i_{min}) & \text{if} \quad x\geq l \end{cases}$$

where x and y are random variables in the interval $[0,1]$, while l is a parameter controlling the noise density over the image, and i_{max} and i_{min} control the

severity of the noise.

Noise Filtering

Given a noisy image, noise filtering attempts to attenuate the noise as much as possible without significantly altering the signal itself. A common technique to noise reduction is linear filtering. Suppose an image $I(x,y)$, and a kernel containing a linear filter $A(h,k)$. The filtered version of the image is obtained by the following discrete convolution:

$$I_A(x,y)=I*A=\sum_{h=\frac{-m}{2}}^{\frac{m}{2}}\sum_{k=\frac{-m}{2}}^{\frac{m}{2}}A(h,k)I(x-h,y-k)$$

where the kernel $A(h,k)$ is of (odd) size $m\times m$. A linear filter replaces the pixel values $I(x,y)$ with a weighted sum of the pixel values from a neighborhood of size $m\times m$ centered at (x,y) . The weights are the entries in the kernel $A(h,k)$.

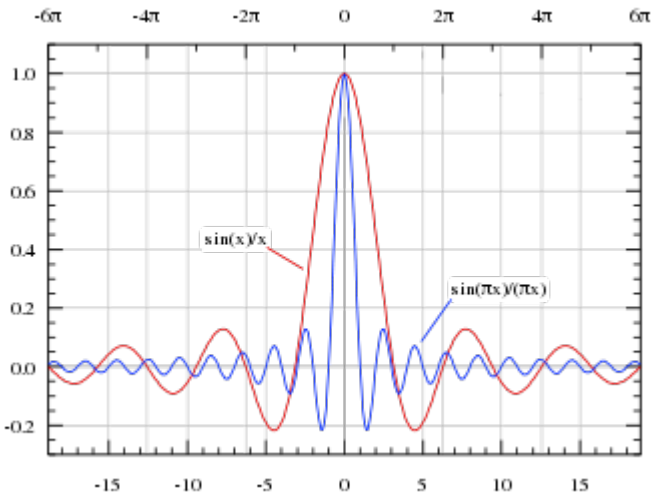
The convolution of two signals (for instance, an image and a kernel) is equivalent to the multiplication of their respective Fourier transforms: $I*A=F^{-1}[F[I]\times F[A]]$, where F and F^{-1} are the Fourier transform and its inverse, respectively.
There are various kernels that can be applied to images to perform different types of image filtering. For instance, an averaging 3 by 3 kernel can be used to attenuate noise. This kernel is:

$$A=\frac{1}{9}\begin{bmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{bmatrix}$$

This kernel attenuates noise because, in general, noise is a high-frequency component and hence, averaging will attenuate sharp local variations. To understand this, consider the Fourier transform of a 1D box function (which is a 1D profile of the averaging kernel):

$$F[\text{box}(x)]=\frac{2\sin(\omega)}{\omega}$$

The result of this Fourier transform is the sinc function, as depicted on the right. Some problems are associated with attenuating the high frequency contents of images. For example, high frequencies are not always caused by noise. In the particular case of an averaging filter, the secondary lobes of its Fourier transform lets in some high frequency content, which may be undesirable.



The most commonly used approach to noise reduction is Gaussian filtering, which is implemented with a sampled Gaussian function as kernel entries. One of the main reasons this is a common approach is that the Fourier transform of a Gaussian is also a Gaussian, and hence there are no secondary lobes in the spectrum, as opposed to the averaging kernel. In addition, 2D Gaussian functions are separable, which allows performing 2D image convolutions very efficiently. The Gaussian filtered version of an image is obtained by the following discrete convolution:

$$I_G(x,y) = I * G = \sum_{h=\frac{-m}{2}}^{\frac{m}{2}} \sum_{k=\frac{-m}{2}}^{\frac{m}{2}} G(h,k)\, I(x-h, y-k)$$

and can be separated along the two dimensions as follows:

$$\sum_{h=\frac{-m}{2}}^{\frac{m}{2}} \exp\left(\frac{-h^2}{2\sigma^2}\right) \sum_{k=\frac{-m}{2}}^{\frac{m}{2}} \exp\left(\frac{-k^2}{2\sigma^2}\right) I(x-h, y-k)$$

The programming technique to perform this 2D convolution efficiently is to proceed with the following steps:

1. Convolve in 1D all rows of the original image, and store the result in a temporary image

2. Convolve in 1D all columns in temporary image, and store result as a Gaussian filtered image.

Note that you may start this process with columns and then rows, as this is interchangeable.
To create a 1D Gaussian kernel, we must sample a Gaussian function which has the appropriate variance for the filtering task at hand. The relationship between the variance of the continuous Gaussian and the kernel size is given by $m=5\sigma$, which subtends 98.76% of the area under the Gaussian curve. Note that m must be odd. Add 1 if necessary.

To speed up computations further, we may construct Gaussian kernels that contain integer values instead of floating points. Since the pixels are also represented by integer values, this amounts to performing discrete 1D convolutions in integer arithmetic. The following steps are taken to construct an integer Gaussian kernel:

1. Create a floating point Gaussian kernel

2. Find its smallest value

3. Find the coefficient which scales this value to 1

4. Apply this coefficient to all the values in the kernel

5. Store the resulting kernel entries as integers

6. Compute the sum of these integer kernel values

7. Use this sum as a divisor in front of the kernel

Alternatively, Gaussian filtering can be achieved with repeated average filtering. For instance, convolving a 3×3 averaging kernel n times approximates a Gaussian convolution with a Gaussian kernel built with:

$$\sigma = \sqrt{\frac{n}{3}}$$

and size $2n+3$.

Non-linear filtering can also be performed on images. This type of filtering cannot be applied to images through the means of convolutions. As an example of such filtering, consider a median filter which replaces each pixel value by a median value obtained from a local neighborhood. Clearly, such techniques cannot be implemented using convolutions.