Implementation of a Fast Convolution Technique to LabView (comparative speed test)

Zoltan Sári
Department of Information Technology, Faculty of Engineering, University of Pécs
Hungary, Tel.: +36 72 503-650 (3727), Fax.: +36 72 501-534, e-mail: ski@morpheus.pte.hu

Abstract **- One of the most fundamental techniques in Digital Signal Processing is the convolution. It has many widely used applications. In this paper a special implementation of convolution which is significantly faster than traditional ones is presented. The language of implementation is LabView.**

Index terms **– Image processing, Convolution, LabView**

I. Introduction

Convolution is a very powerful and versatile technique, but it has a disadvantage: The required computation is very time consuming. Every bit of improvement of its efficiency is definitely well worth the effort. The larger the amount of data have to be worked with, the larger the problem is with speed. The field which has been chosen to make our examinations is image processing, because:

1. Pictures usually have information content, large enough to motivate building up fast algorithms.
2. There are typical applications in image processing (for example "pattern recognition"), where the convolution is applied several times, and a slight improvement in one step can significantly decrease the time needed to complete the entire operation.

Discrete convolution in two dimensions can be defined by the next equation:

$$g(m,n) = \sum_{i=0}^{M-1}\sum_{j=0}^{N-1} h(i,j)\cdot f(i-m, j-n) \qquad (1)$$

where $f(m,n)$ denotes the original image, $g(m,n)$ the edited one, and $h(,)$ is the convolution kernel. It can be seen clearly that convolution is made of additions and multiplications. The number of these operations depends on the size of the image and the convolution kernel. It is known that convolving with a function which contains only impulses is very easy, but in an image, every nonzero pixel is an impulse which has to be processed [3]. If the number of these impulses can be reduced, the amount of time required to convolution can also be reduced. It is important that reducing the number of impulses will distort the image some way, however a good method will keep the level of this distortion as low as possible. The method presented in this paper is a special case of a more general approach [1], applied to the field of image correlation. Furthermore the efficiency of the implementation has been tested considering execution time.

II. The Method

Throughout in this paper 8-bit grayscale images are discussed.

A picture can be represented as a matrix of impulse functions where every matrix element (pixel) is a scaled and shifted version of the unity impulse, and the goal is to eliminate as many of them from the picture as possible, maintaining an acceptable amount of distortion of the picture. The solution can be evaluated in two steps. In the first step the picture is divided into several small parts ("boxlets"), where pixel values are "almost constant" and each pixel value in the partition is substituted with the average brightness. (The exact definition of this partitioning will be discussed later.) The second step is to calculate the first order partial derivatives of the picture in both directions (horizontal and vertical). As a result of calculating the derivatives the image becomes a field of impulses and zeros. To get the final result of convolution, an integration has to be taken along the two axes. With the method described above the number of impulses can be significantly reduced in the image, thus the convolution (with any convolution kernel) can be performed much easier and faster.

A. Partitioning the image

Consider an image as a function *f(x, y)* over a domain *M*, and let $B = \{b_i\}$ a partitioning of the image where: $b_i \cap b_j = 0 \mid i \neq j \;\; and \;\; \bigcup_i b_i = M$. And let choose a constant brightness(C) for each partition to minimize the next summation.

$$s_i = \sum_{(x,y)\in b_i} \left(f(x,y) - C\right)^2 \qquad (2)$$

The average brightness is used on each partition as C, thus the above equation has the form:

$$s_i = \sum_{(x,y)\in b_i} \left(f(x,y) - \bar{f}_{b_i}\right)^2 \qquad (3)$$

Considering a more general approach, a polynomial can be used instead the constant [1]. In this case the constant can be regarded as a zero order polynomial. Equation (3) defines an error function for each partition, and the purpose is to minimize this error. On the other hand the number of partitions should be kept minimal too. So the problem is to find a partitioning where both the number of partitions, and $\sum_i s_i$ should be minimal. It can be seen clearly that these goals can only be achieved by making a compromise, because the less error is allowed, the more partitions has to be made. The most straightforward solution is to choose a constant K what can be called **threshold**, and make the largest possible partitions, where $K \geq s_i$. Thus both the error and the number of partitions can be minimized with respect to K. Selecting a proper K is not so difficult, and for a particular application it has to be selected only once.

Interpolating regions of the image with zero order polynomials will result increasing the overall noise level in the image. Nevertheless in most cases images have an inherent noise, and this increase in noise level does not have a great impact on the result of certain applications as pattern recognition.

In Fig. 1. below can be seen the effect of breaking an image into partitions. Upper left is the original image, upper right and lower right images are partitioned with K=1000, the only difference is that the boundaries of partitions can be seen on the upper image, the lower left image is partitioned with K=100. It can be seen that if K is small enough, the resulting image is very similar to the original (the error is small). You can also notice that the partitions are much larger in the homogenous regions of the picture, and this way, large groups of pixels can be represented by a few impulses.

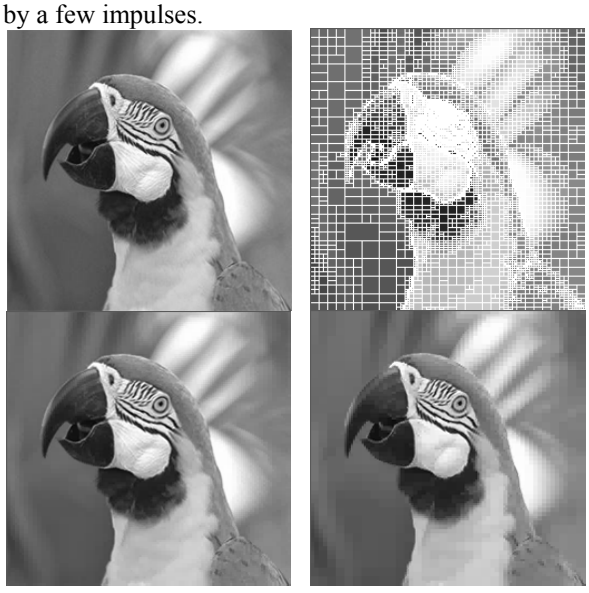

Fig. 1. The effect of partitioning

B. Calculating the derivatives

Now the partitioning is successfully realized with the requirements above. The next step in the process is calculating the partial derivatives along both the horizontal and vertical directions. The equation below defines the derivatives [4]:

$$\frac{\partial f}{\partial x} = f(x,y) - f(x-1,y), \qquad \frac{\partial f}{\partial y} = f(x,y) - f(x,y-1). \tag{4}$$

Applying these to the partitioned image, an interesting thing will happen. Since the brightness value on each partition is constant, the derivative will be zero here, and only a few nonzero values will remain at the boundaries of the partitions. Furthermore if two (or more) partitions have their corners at the same place, these two impulses becomes one: the sum of the former two.

In Fig. 2. an image and its impulse representation can be seen. You can see that only the boundaries of objects remain. Notice that the rectangle is completely disappeared except its four corners, and the same thing happened to the rectangular part of the arrow head.

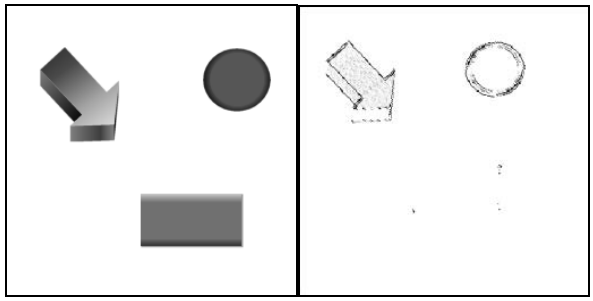

Fig. 2. An image and its impulse representation

III. The Implementation

The method described above can be implemented in many ways, in various programming languages. The programming language of this implementation is LabView. This is a powerful graphical programming environment for building applications. It's easy to use and offers several built-in functions, controls, and various visualization options to speed up the development process. Besides its advantages it has a disadvantage: it doesn't support recursive algorithms. This is not a problem in most cases, and it is known that every recursive algorithm can also be solved by iterations, but in certain cases it would be very useful if the programmer had a support for it. Fortunately there is a solution to this problem. LabView is capable of using code written in other

languages, through a mechanism called **Code Interface**. With a code interface it becomes possible to make the visualization and user interaction in the high level LabView environment, and the underlying recursive code in C++.

Let's take a closer look to the algorithms. The heart of the program is the recursive algorithm which is responsible for the partitioning. The theory behind it is very simple:

1. Take the image
2. Calculate the average brightness
3. Calculate the error
4. If it is too large, cut the image into halves and do the same process on the two halves.

So this is the algorithm to break the image into partitions. It works, but there are a few nuances for the practical application. It is good to maximize the partition size; it speeds up the algorithm due to avoiding a large amount of unnecessary computation (step 2, and step 3 for large images). Furthermore cutting an image into halves can be done in two basic ways: horizontal and vertical. The good way is to change the direction for every cut, because this will make the partitioning more balanced.

After the partitioning, applying the derivatives is quite straightforward. The result of these operations is a field which the convolution is very easy to perform with, because it contains only a few impulses compared to the original number of pixels. To convolve another function with this field, the only thing to do is making copies of the function at every impulse and adding them together.

To obtain the final result of convolution, an integration has to be realized with respect to both the horizontal and vertical directions.

Well, now an overall idea of the implementation of the method is introduced. It is time to take a closer look. In the next few paragraphs the most important parts of the algorithm is described in more detail, and some further detail about the code interface mechanism.

A. The code interface

This is a very useful capability of the LabView environment, because it enables using code written in other programming languages. This is an especially useful aid when solving a problem which requires recursive algorithms (just as in the previous case) that cannot be implemented in the LabView environment, and it can be also used for writing the speed critical sections of your code in fast, low level languages. Using the code interface is a five step process:

1. Defining a high level programming element (Code interface node (CIN)) to handle the underlying code, and declare inputs and outputs for the low level code.
2. Generating a low level code frame according to the structure of the defined inputs and outputs.
3. Filling this code frame with your low level code.
4. Compiling the code, and making a special link using a utility provided by the LabView environment. The result of this step will be a resource file which can be used in the high level program.
5. Loading the resource file into the CIN, and this high level element can be used just as usual.

The first two steps and the last one can be performed in the LabView environment, the remaining two steps has to be done in some other development environment e.g. Microsoft Visual Studio. To complete the 3rd and 4th steps a *DLL Project* has to be made in Visual C++, adding some further files to your project (these files can be found in the CIN directory of LabView), furthermore some special linker options also has to be defined. The next figure *(Fig. 3.)* shows a LabView generated code frame. The instructions and operations that can be used on LabView objects are defined in LabView External Code manual [2].

```
/* CIN source file */

#include "extcode.h"

/* Typedefs */

typedef struct {
        int32 dimSizes[2];
        uInt8 Numeric[1];
        } TD1;
typedef TD1 **TD1Hdl;

typedef struct {
        int32 dimSizes[2];
        uInt16 Numeric[1];
        } TD2;
typedef TD2 **TD2Hdl;

MgErr CINRun(TD1Hdl *In, int32
*Threshold, int32 *MaxBoxSize,
TD2Hdl *Out);

MgErr CINRun(TD1Hdl *In, int32
*Threshold, int32 *MaxBoxSize,
TD2Hdl *Out)
        {

        /* Insert code here */

        return noErr;
        }
```

Fig. 3. A LabView generated code frame

It can be seen from the function declaration that using handlers (pointers to pointers) is the common way to handle composite objects, just like an array

which can dynamically grow and shrink. And the value returned is also a handler. This way of object handling means that the memory management is not automatic, and the problem of allocation and freeing memory for the dynamic objects has to be solved by the programmer. It is important to do this task right, particularly in the case of an algorithm which executes a thousand times or more.

B..The program

As it was mentioned above the method was implemented in LabView, so the program looks like a flowchart of functions and operations. In this particular case the program contains a few blocks which perform the operations. The structure of the LabView program can be seen below in *Fig. 4*. The diagram can be separated to three main parts: the quantization algorithm (*quant*), the conversion from the inner representation of partitions to image (*b>pix*), and the calculation of the derivatives (*the two for loops*).

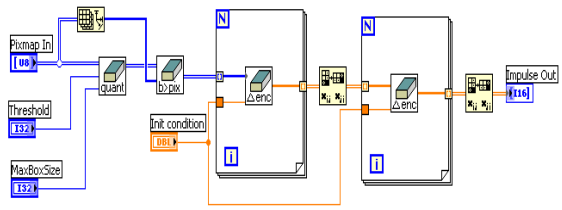

Fig. 4. The program to create the impulse representation

IV. The Test

A. Accuracy and noise

Now the implementation of the fast convolution method is done. The next task is to test its efficiency, and the application for this purpose is pattern recognition, which is a technique to find i.e. a small part of image in the whole. The most basic way of finding a pattern in an image is to correlate the image with the pattern. The correlation in two dimensions is described as follows:

$$g(m,n) = \sum_{i=0}^{M-1}\sum_{j=0}^{N-1} h(i,j)\cdot f(i+m, j+n) \qquad (5)$$

Taking a closer look at this equation, and comparing it to (1), there can be seen that they are very similar, except the signs in function *f(,)*. So convolution could be used to perform correlation if the pattern had been prepared properly. This preparation in this case means that the pattern has to be mirrored to both axes.

The result of the correlation is a function which shows the location of the pattern on the image. In ideal cases, the resulting function will contain a small amount of noise and one well defined maximum indicating the location of the pattern. If the result looks like noise it means that the pattern has not been found. The next three figures show the result of correlating a pattern with an image. Each of them was created by correlating a 256x256x8 image by a 30x30x8 pattern. The first one *(Fig. 5.)* shows the result of traditional correlation, here a well defined spike and an acceptable amount of noise can be seen. The second one *(Fig. 6.)* shows the correlation performed by the fast convolution method implemented. It can be noticed that there is a measurably larger amount of noise in this function, but the maximum is defined well enough yet. On the last figure *(Fig. 7.)* it can be seen what happens when threshold is too high. On each figures dark color indicates the high, and light indicates the low values of the function.

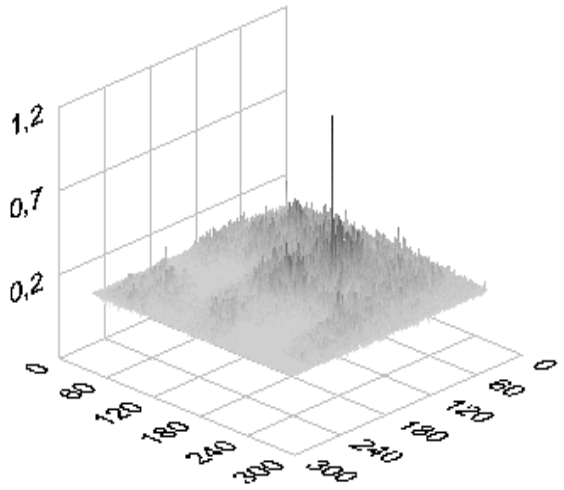

Fig. 5. Correlation by traditional convolution

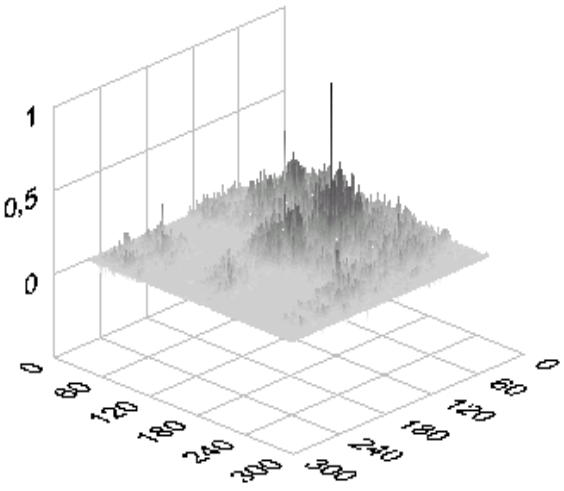

Fig. 6. Correlation by fast convolution K=1000

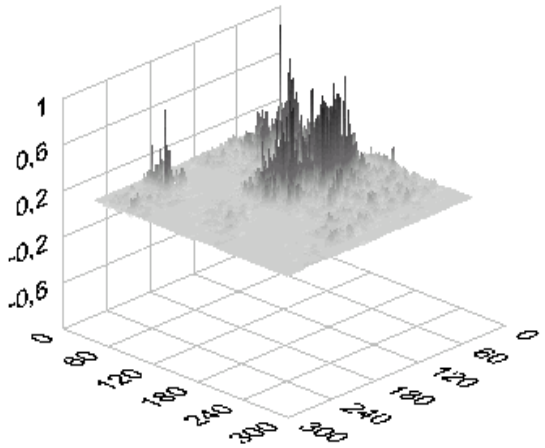

Fig. 7. Correlation by fast convolution K=8000

It can be seen that the threshold constant has a great impact on the result, because too large threshold can increase the amount of noise (error) to a level, where the pattern could not be recognized. The value of acceptable threshold depends on the size of the pattern, and the content of the pattern and the image.

B. Execution time

It has been shown that the method works, and with a proper selection of the threshold the results of the fast method are exactly the same as the traditional one. But there is one more question to answer: How fast is it? How much faster is it than the original method?

The answers to these questions can be obtained through a few speed tests. The tests have been made in the following hardware and software environment.

Hardware configuration:

- CPU: AMD Barthon 2500+
- Memory: 512MB, 266MHz
- Motherboard: ASUS A7N8X

Software configuration:

- National Instruments LabView 7
- Microsoft Visual Studio 6.0

Execution times of the algorithms were provided by the profiler of the LabView environment.

The following figures are "Execution time vs. pattern size" diagrams, which were created by correlating a 256x256x8 image by different sized patterns. It can be seen the newly implemented algorithm is much faster than the original one. On each diagram the solid colored columns shows the time statistics of the fast method and the diagonally striped ones correspond to the traditional one. In the first case *(Fig. 8.)* the threshold constant was set to K=1000, on next figure (*Fig. 9.)* it was adjusted to a high, but acceptable level for the various size of patterns.

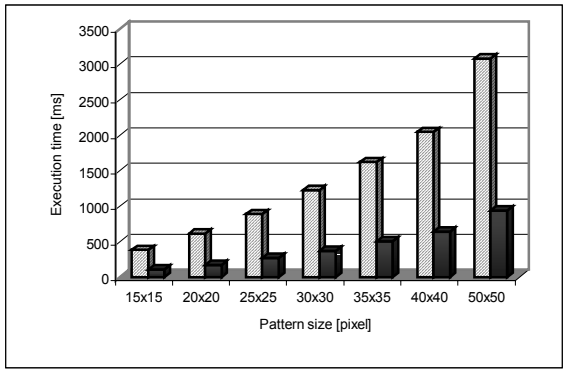

Fig. 8. Execution time vs. pattern size, K=1000

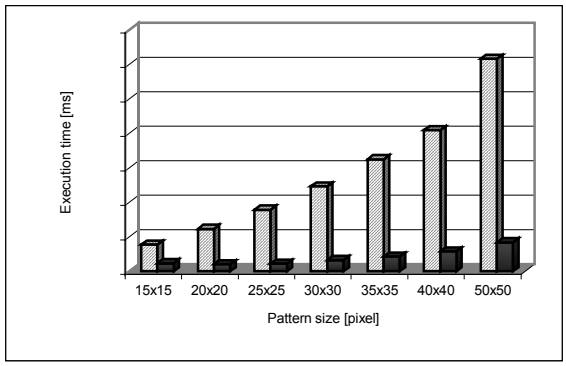

Fig. 9. Execution time vs. pattern size, threshold adjusted

Noticing the difference between *Fig.8.* and *Fig.9.* leaves no doubt about the importance of adjusting the threshold constant for any particular applications. It can be seen that the fast algorithm is fast enough without adjusting the threshold level, but in most cases (it depends on the pattern and the image) a higher level of threshold can be found, which can make the overall process approximately two times faster. Finding the highest possible threshold value to work with is an important part of using this method effectively. At first glance it seems that the threshold value can be determined only by experiments, because the level of the highest acceptable threshold highly depends on the pattern and the image itself. It can be a challenge to get the highest possible value of threshold automatically, but as it can be seen from the following comparison the threshold value adjustment is not so important in the case of small patterns.

The next figure shows a speed comparison between the fast and the traditional algorithm. The graph indicates a "Speed ratio vs. pattern size". The upper line signed with boxes corresponds to the condition when the threshold was adjusted; the lower line corresponds to the constant threshold (K=1000). The threshold values of the second case (when threshold was adjusted according to the size of the pattern) can be found in *Table 1*.

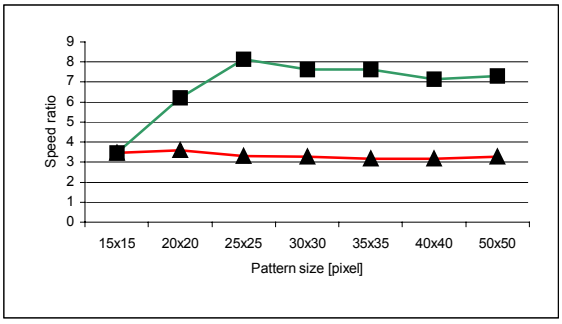

Fig. 10. Speed ratio vs. pattern size

TABLE.I:
THRESHOLD VALUES TO PATTERN SIZES

Pattern size	15x15	20x20	25x25	30x30
Threshold	1000	4500	6000	6500
Pattern size	35x35	40x40	50x50	-
Threshold	6500	5500	7000	-

V. CONCLUSION

It has been shown that the fast "Boxlet" method is significantly faster than traditional convolution in pattern recognition applications. The overall performance of the fast method is approximately 3 times better, but by adjusting the threshold properly we can achieve nearly 8 times faster execution. In *Fig.10.* it can be seen that the advantage from adjusting the threshold level diminishes at small pattern sizes. The main reason behind this phenomenon is that partitioning the image will distort the features of it, and the smaller the pattern size, the greater the impact this distortion has on the result.

ACKNOWLEDGMENT

The author would like to thank Amalia Ivanyi for her useful hints and the Department of Information Technology for the support and resources.

REFERENCES

[1] Patrice Y. Simard, Léon Bottou, Patrick Haffner, Yann LeCun, *„Boxlets: a Fast Convolution Algorithm for Signal Processing and Neural Networks"*, [Online], http://research.microsoft.com/~patrice/PDF/boxlet.pdf

[2] National Instruments, *Using External Code in LabView*, 2003

[3] Stephen W. Smith, *The Scientist and Engineers Guide to Digital Signal Processing,* California Technical Publishing, 1997

[4] Rafael C. Gonzalez, Richard E. Woods, *Digital Image Processing*, Prentice Hall, 2002