1. Convolution

1.1. Introduction

One of the basic operations performed in image and signal processing is an operation called **convolution.** In image processing, many noise reduction filters utilize the convolution operation in order to perform their tasks. These filters will be discussed in later sections.

From a purely mathematical standpoint, convolution is an integral. Imagine two functions *f* and *g*. The purpose of the operation is to shift *g* over *f*. The resulting amount overlap that occurs when *g* is shifted over *f* is the **convolution** of *f* and *g*. Because of the nature of convolution, the resulting integral is a "blending" of *f* and *g*. This can be expressed in the following finite integral:

$$1.1 \qquad f * g \equiv \int_0^t f(\tau)g(t-\tau)d\tau$$

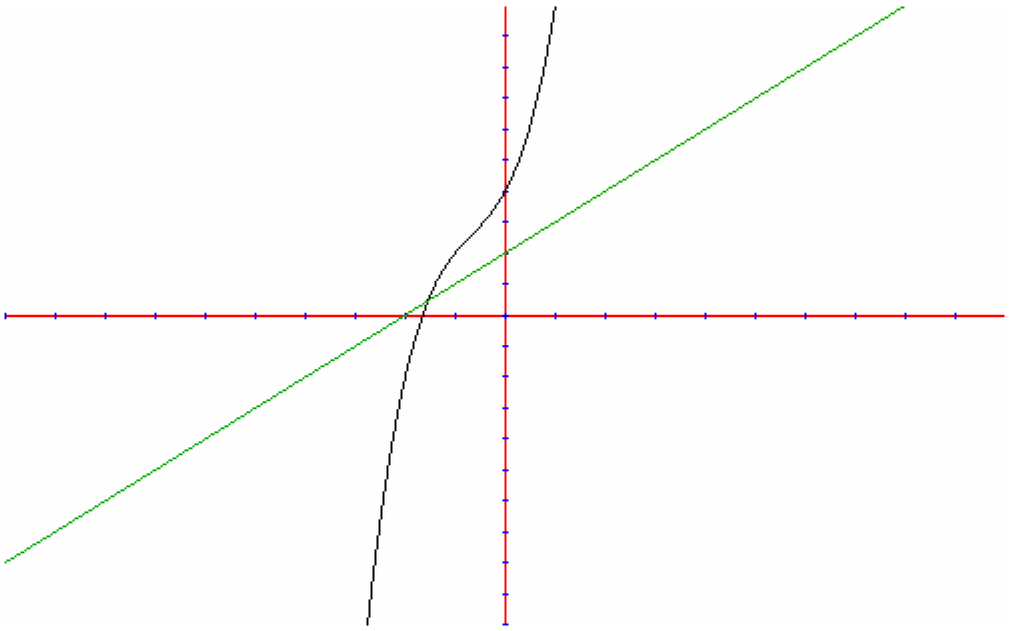
From this point on, the symbol "*" will refer to convolution. Convolution can also be expressed over an infinite range by the following integrals:

$$1.2 \qquad f * g \equiv \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau \qquad = \qquad \int_{-\infty}^{\infty} g(\tau)f(t-\tau)d\tau$$

Since image processing involves computing convolution on a computer, only the finite integral will be used; however, keep the infinite integral in mind, as it will reinforce the overall idea of convolution later as convolution is used for calculations.

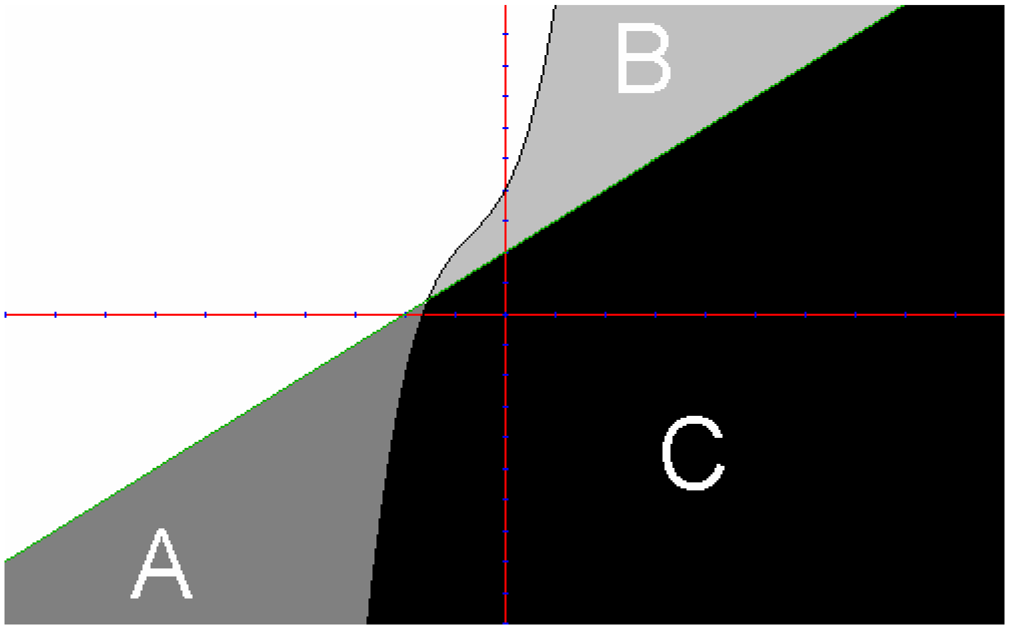
1.2. Working Example

A great way to understand the workings of convolution is by example. Assume that we have two functions *f* and *g*, where $f = x^3 + 2x^2 + 3x + 4$, and $g = x + 2$:



Recall that the convolution is the overlap of *f* and *g* as *g* is shifted over *f*. The integral in the convolution definition represents the summation of each portion of the overlap of the two functions. This is the same idea behind the integral representing the area under a curve bounded by some interval. Therefore, the integral of the convolution between two functions is the area represented by the overlap of the two functions. In terms of Boolean logic, the convolution of the functions *f* and *g* is the area under *f* AND the area under *g* (expressed as $f \otimes g$). Hence: $f * g$ is the convolution of *f* and *g*.

The following graph illustrates these ideas for our example situation:



The darker gray region (region A) is the area under *g*, the lighter gray region (region B) is the area under *f*, and the black region (region C) is the area under both *f* and *g* (i.e., the overlap of *f* and *g*). In terms of our *f* and *g*, which are both functions of *x*, the darker gray region (A) is the area under $g(x)$, the lighter gray region (B) is the area under $f(x)$, and the black region (C) indicates the product of $f(x)\ g(t-x)$. The black region, a.k.a. the overlap region, is precisely the convolution of *f* and *g*.

Now that a graphical analysis has been laid out, the underlying convolution formula becomes more understandable. Applied to this example, the convolution of *f* and *g* is simply the product of *f* and *g*:

$$\begin{aligned} f * g &= (x^3 + 2x^2 + 3x + 4)\ (x + 2) \\ &= x^4 + 4x^3 + 7x^2 + 10x + 8 \end{aligned}$$

This result is consistent with the aforementioned observation that the overlap area of *f* and *g* is $f \otimes g$.

1.3. Convolution as a Matrix

1.3.1 Method

All complex mathematics can be performed on a computer using matrices. Matrix math possesses many unique properties that allow fairly complex operations to be performed by the computer with relative ease. Convolution is no different. The integrals expressed in the previous section demonstrate the fundamental, underlying mathematical basis behind convolution. The example provided demonstrates mathematically how convolution functions. But now it is time to translate that idea into practical terms and instruct a computer to perform this operation.

The matrix representations of our functions *f* and *g* functions are as follows:

$$f = [1 \quad 2 \quad 3 \quad 4] \qquad , \qquad g = [1 \quad 2]$$

To better understand convolution from both a mathematical and a computer science standpoint, it is best to compute the convolution of *f* and *g* manually before writing an algorithm to do so. Calculating the convolution by hand further demonstrates the fundamental mathematic operations at hand, and what implications they will have on the final result.

Convolution, as stated earlier, involves a shifting of one function over another. The result is the overlap that occurs as one function is shifted over the other. Calculating the convolution of *f* and *g* manually involves that exact operation. In order to better grasp the idea of convolution, only 1-dimensional matrices (vectors) will be analyzed. Multi-dimensional convolution can become quite cumbersome to visualize unless the basics are illustrated first. Once single-dimensional convolution is understood, that knowledge can then be used to better understand more complex convolution operations in multiple dimensions. Therefore, only single-dimensional convolution will be analyzed here.

To perform the convolution of *f* and *g*, arrange the two matrices as follows:

```
        1    2    3    4
   2    1
```

For this example, *g* is shifting over *f*. With each shift, the elements of *f* that are vertically aligned with the elements of *g* are multiplied together. Multiplication starts with the first element of *g*, then the second, and so on, until all elements of *g* have been multiplied. Then, each vertical product created by the alignment of *f* and *g* is summed together, to produce a final result. That final result becomes the new coefficient of the convolution of *f* and *g*.

In other words, each coefficient of the convolved function $f * g = f_g$ will be:

$$1.4 \qquad f_g(\mathrm{i}) = f * g = \sum_{h=-m/2}^{m/2} g(h) f(i-h),$$

where *m* is the size of the smaller of the two functions (either *f* or *g*), and *m/2* indicates integer division (e.g., 3/2 = 1).

The first iteration of convolution will result in:

```
        1    2    3    4
   2    1
```

$f_g(1) = 1 * 1 + 2 * 0 = 1$

Observe that the second element of *g* does not align vertically with any elements in *f*. Therefore, the second element of *g* is multiplied by 0. Graphically, this occurs during a partial overlap of *f* and *g*, when only part of *g* is overlapping *f*, rather than the whole function. To visualize this better and help keep the shifting in order, zeros can be inserted at the beginning and end of *f* as follows:

0	1	2	3	4	0
2	1				

Adding zeros to either side of *f* will not change its value. The number of zeros to add on either side of *f* corresponds to the number of elements in *g*, minus one. For this example, since *g* has two terms, a total of two zeros can be added—one either side of the elements of *f*—to aid in calculating the convolution.

The first element in the convolution demonstrates that at this particular area in the graph, only the rightmost portion of *g* overlaps the leftmost portion of *f*. The shifting of *g* to the right towards *f* has resulted in a partial overlap of *f* and *g*. This is accounted for the first element in *g* lining up with and multiplying with the first term of *f*. While this occurs, the leftmost portion of *g* contains area that does not overlap *f* at all, since only part of *g* has been shifted over far enough to overlap with *f*. This is accounted for the second element of *g* multiplying with zero (i.e., the second element of g is not multiplying any term of *f*).

This completes the first iteration of convolution. In order to prepare for the second iteration of convolution, it is necessary to shift *g* over by one space:

1	2	3	4
2	1		

Now, the first element of *g* aligns vertically with the second element of *f*, and the second element of *g* aligns vertically with the first element of *f*. Performing the operations outlined in Formula 1.4 will yield our second element of f_g:

1	2	3	4
2	1		

$f_g(2) = 2 * 1 + 1 * 2 = 4$

Once *g* is shifted over far enough, such that the last term of *g* aligns vertically with the first term of *f*, the area under *g* fits neatly inside the area under *f*. It is here that full overlapping of *f* and *g* occurs. This is accounted for in this calculation by the fact that each element in *g* multiplies some element in *f*, and not some extra zero not included in *f*.

Subsequent iterations of this process look exactly the same as this previous step, as long as each element in *g* still multiples some element in *f*:

1	2	3	4
	2	1	

$f_g(3) = 3 * 1 + 2 * 2 = 7$

1	2	3	4
		2	1

$f_g(4) = 4 * 1 + 3 * 2 = 10$

Once *g* is shifted over far enough, such that the rightmost portion of *g* no longer overlaps the area under *f*, it is necessary to revert back to the method described in the first iteration of this procedure. Here, the first element of *g* is multiplied by 0, then (if necessary) the second, and so on. The last iteration occurs when the last element of *g* lines up vertically with the last element of *f*. For this example, the last iteration occurs on the 5th step:

$$\begin{array}{cccccc} 1 & 2 & 3 & 4 & & \\ & & & 2 & 1 & \end{array}$$

$f_g(5) = 0 * 1 + 4 * 2 = 8$

As stated earlier, *f* can be padded on either side by the necessary number of zeros:

$$\begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 0 \\ & & & & 2 & 1 \end{array}$$

The convolution operation is now complete. The resulting convolution of *f* and *g* is the following:

$$f * g \quad = \quad f_g \quad = \quad [1 \quad 4 \quad 7 \quad 10 \quad 8]$$

Since each element of f_g represents the coefficients of the convolved function, the final function is:

$$f_g(x) \quad = \quad x^4 + 4x^3 + 7x^2 + 10x + 8$$

Recall the result obtained in Section 1.2 using our initial statements. The result obtained using the integration formulae and the graphs is *exactly* the same as the result obtained using this matrix method. This should come as no surprise, since the underlying operations performed in both of these methods is exactly the same. Though they may look different, the internal workings of each produce the same result.

1.3.2 Analysis

One observation to take note of is the degree of the convolved function compared to the degrees of the original functions. Likewise, take note of the size of the convolved matrix in comparison to the size of the original matrices. The convolved result will *always* be larger than the original matrices used as input. For the example used here, f is of degree 3, and g is of degree 1. The resulting convolved function $f * g$ is of degree 4. Likewise, the matrix f contains four elements, and the matrix g contains two elements. The resulting convolved matrix f_g contained five elements. This observation can be summed up by the following formulae:

1.5 $$degree(f * g) = degree(f) + degree(g) - 1$$

1.6 $$size(f_g) = size(f) + size(g) - 1$$

Here, Formula 1.5 applies to polynomials, and Formula 1.6 refers to matrices. Another observation to be aware of: since we are dealing with vectors (1-dimensional matrices) in Formula 1.6, the size of each vector used is equivalent to its norm. Therefore, Formula 1.6 can be rewritten as:

1.7 $$norm(f_g) = norm(f) + norm(g) - 1$$

Now the convolution operation, presented in the Section 1.1 as an indefinite integral unable to be properly represented on a computer, can be executed using matrices, which *can* be represented properly on a computer. This cute method of calculating the convolution $f * g$ of two functions f and g using matrices has fulfilled the original goal outlined at the start of this section. Now this knowledge can be put to practical use.

1.4. The Inverse of a Convolution

1.4.1 Mathematical Basis

An important issue that is brought up concerning the nature of convolution: every time convolution is performed on two functions or matrices, the result grows in size. How, then, is it possible to "undo" a convolution? In other words, what is the inverse of convolution? Inverting a convolution is not as simple as what it may seem. Indeed, the mathematical inverses of multiplication and addition are division and subtraction, respectively. However, due to the nature of operation of convolution, in addition to the aforementioned size increase in the result, the inverse cannot be simply calculated by simple means.

Intuitively, it might be a first guess to shift *g* over *f*, but divide and subtract the elements instead of multiply and add as *g* shifts. Yes, division and subtraction are the inverse operations to multiplication and addition respectively, but this does not yield the correct result. This method simply does not match the fundamental definition of convolution.

Next, it might be a second guess to convolve the inverse of *g* with *f*, in order to obtain the inverse of $f * g$. This would be the same as performing: (9 * 3) / 3. However, this does not work either. By definition, the convolution is the area of the overlap of two functions. In order to find the inverse of a convolution, it is necessary to (essentially) find the negative area of the overlap. Convolving *g*` with *f* will only create another unique overlap that will not equal the inverse of the area of overlap by $f * g$.

What is necessary is to create an overlap that will, when added with the overlap of *f* * *g*, will produce the original area of either *f* or *g*, whichever is desired. The method that will achieve this result utilizes the same idea behind integrating an odd function for a range that is bisected by the y-axis. When integrated, there is an equal amount of area under the curve to the right and to the left of the origin (and therefore the y-axis). In other words, the amount of positive area equals the amount of negative area. Therefore, the addition of the areas equals zero. For example:

$$\int_{-1}^{1} x\,dx = \tfrac{1}{2} x^2 \Big|_{-1}^{1} = \tfrac{1}{2}(1)^2 - \tfrac{1}{2}(1)^2 = 0.$$

The result of this integration is zero. Each term in the final step of this integration does have area, but the fact that the result takes into account an equal amount of positive and negative area yields the result of zero. This is precisely the effect that is necessary to invert a convolution.

1.4.2 Implementation

To illustrate this point, imagine two matrices *i* and *a*:

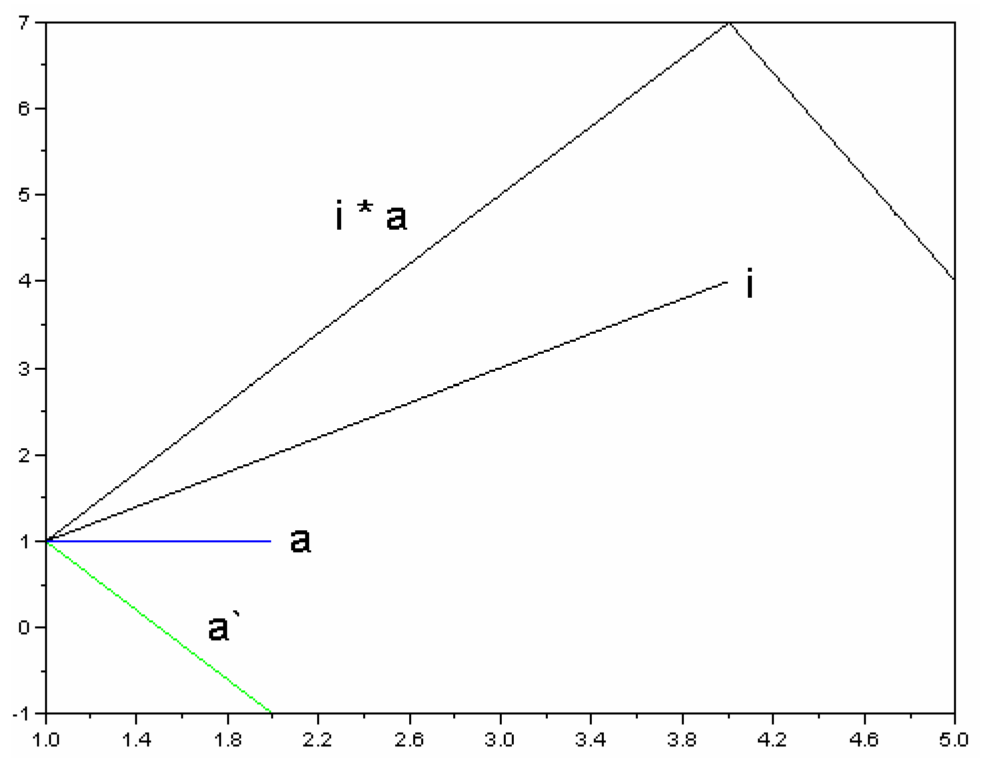
$$i = [1\ \ 2\ \ 3\ \ 4],\ a = [1\ \ 1]$$

When convolved, $i * a = [1\ \ 3\ \ 5\ \ 7\ \ 4]$. The goal now is to perform the inverse of convolution to obtain the original matrix *i* again. Another way of looking at this is to simply "undo" the convolution just performed on *i* and *a* to get *i* back. For this particular example, intuition might dictate modifying *a* in a way that is consistent with the idea presented in the example of integrating an odd function. Carrying out this idea yields the following "inverse" matrix of *a*, *a*`:

$$a` = [1 \;\; -1]$$

To get the original *i* back, the procedure follows the same steps as a basic multiplicative inverse: *(x * y) * y` = x*. Therefore, in terms of this example: *(i * a) * a` = i*. Before actually calculating the convolution of *(i * a) * a`*, it is necessary to understand the implications of this operation, and why it works.

The following is a graph of the matrices *i, a, a`,* and *i * a*:



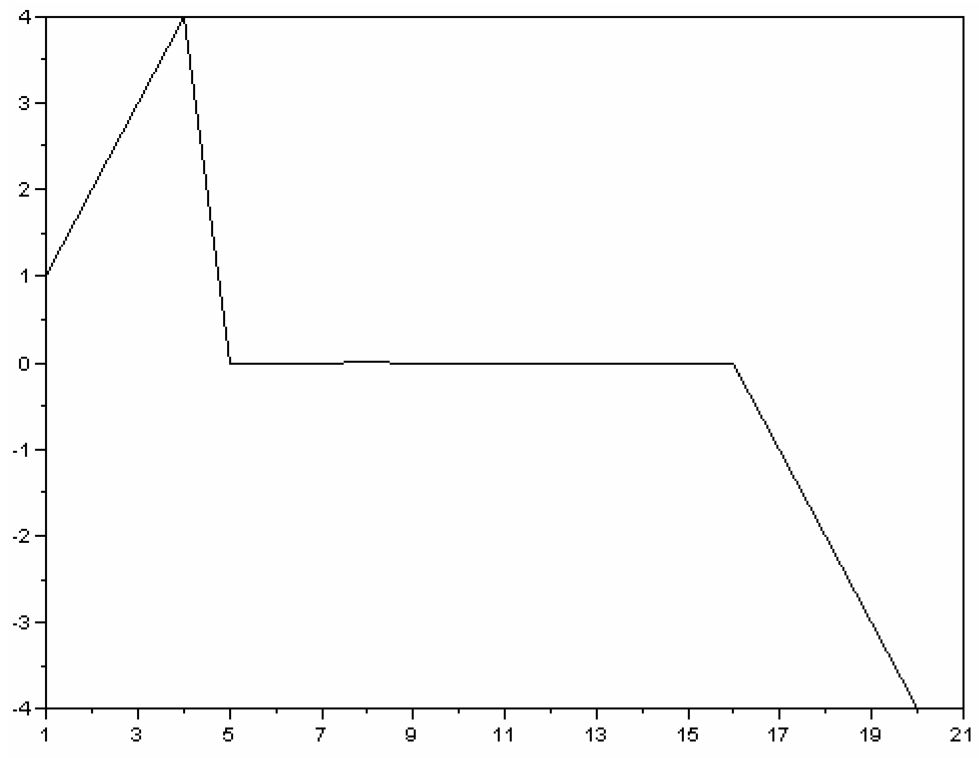
Notice the orientation of *a* and *a`*. When convolving *i * a*, the result is moved upwards on the vertical axis. During convolution, the magnitude of the values in each matrix involved will determine the overall magnitude of the result. Positive values in the matrix will contribute to a positive or zero slope for the graph of that matrix, while negative values will contribute to a negative slope for the graph. Since the values in *a* are all positive, the graph of *a* has a positive slope. On the other hand, one value of *a`* is negative, and contributes to the negative slope that *a`* exhibits. In order to compute the inverse of the convolution *i * a*, it is necessary to convolve the result *i * a* with a matrix *a`* that has exactly the opposite slope of the shifted matrix *a* used in the first convolution. The opposing slopes of the *a* and the *a`* matrices will balance each other out during the second convolution, and the original value of *i* can be obtained again.

However, there is one caveat to this method. In a mathematical sense, this method will reproduce *i* after completing *(i * a) * a`*. But keep in mind that the fundamental definition of convolution (Formula 1.2) states that the integral is infinite. Therefore, the *a`* matrix must be of infinite length in order to fully preserve the original *i* matrix after convolution. This spells bad news from a computer science standpoint. The method that is discussed here is in fact only a pseudo-inverse. It will be impossible to calculate the full, real, actual inverse of a convolution on a computer. Computers are only capable of storing a finite number of digits in calculations. Because of this serious limitation, it will not be possible to obtain the original *i* matrix *exactly*. What ends up occurring is the formation of a bloated matrix *i*— which will be referred to as *i`*—that includes the original values of *i* in their original order, followed by a specific number of zeros[1], and terminated with the *negated* original values of *i* in their original order. The

[1] In actuality, the values are not exactly zero. The values are close to the ε of the machine being used for the calculations. These values are actually on the order of 10^{-13} to 10^{-16}, depending on the accuracy of the machine used. Refer to Section 1.5.2 for a further discussion.

number of zeros inserted is determined by the size of *a'*, in which the values of *a`* are repeated in succession. This property will be discussed further; but first, an examination of the SciLab output and graph is necessary to illustrate this peculiarity:

```
-->i1

 i1  =

!   1.    2.    3.    4. !

-->a1

 a1  =

!   1.    1. !

-->ia1

 ia1  =

!   1.  - 1.    1.  - 1.    1.  - 1.    1.  - 1.    1.  - 1.    1.  - 1.    1.  - 1.    1.  - 1. !

-->ia=convol(i1,a1)

 ia  =

!   1.    3.    5.    7.    4. !

-->ip=convol(ia,ia1)

 ans  =

         column  1 to 12

!   1.    2.    3.    4.  - 1.021D-15  - 6.683D-16  - 6.433D-16  - 1.899D-16  -
9.992D-16  - 1.027D-15  - 1.110D-15  - 5

.551D-16 !

         column 13 to 20

! - 1.335D-15  - 8.147D-16  - 1.348D-15    9.290D-16  - 1.  - 2.  - 3.  - 4. !

-->
```



The value of *i*` (ip), as calculated by SciLab, is consistent with the value of *i* (i1) for the first *n* terms, *n* being the number of terms in *i*. Past that index, the zero values are inserted, and then the negated values of *i*. The above graph actually contains both i1 and ip. This may not be clear at a first glance, since i1 and ip are colinear between x = 1 and x = 4. From x = 5 to x = 16, the data represented by the graph are the zero values inserted. Finally, the negated values of i1 (found between x = 1 and x = 4) appear on the range from x = 17 to x = 20.

Interestingly, the size of *a*` determines the number of zeros that are padded in the middle of *a*`. If the values in *a*` are repeated over and over, more zeros are inserted in the middle of *i*`, and the negated values of *i* are pushed even farther away from the original values of *i*. This observation is consistent with the inherent properties of convolution discussed earlier. The size of the convolved result is determined by the sum of the size of the inputs, minus one. Therefore, the longer $a`$ is (i.e., the more times the values of $a`$ are repeated within $a`$), the more zeros will be inserted into $i`$, and longer $i`$ will become. However, there is a lower limit to this observation. To obtain the correct value for $i`$, the size of $a`$ has to be at least the size of the original *i* matrix. This means that the appropriate number of repetitions of values in $a`$ must be inserted to create an $a`$ that is long enough to give the correct result. In regards to our example, this means that the size of $a`$ must be at least the size of *i*, which is 4. Therefore, $a`$ must include at least one repetition of its values: $a` = [1\ \ -1\ \ 1\ \ -1]$. If $a`$ is not of the appropriate length, then the resulting $i`$ matrix will not be correct. The reason behind this problem refers back to the properties of convolution. The size of the convolved result is the sum of the sizes of the inputs, minus one. In order for the inverse matrix $i`$ to contain the correct values, $i`$ must be at least 2x the size of *i*, since $i`$ holds the values of *i* as well as the negated values of *i*. If $a`$ is exactly 2x the size of *i*, then $i`$ will not include any zeros, and will only possess the original and negated values of *i* itself. An improper choice for the size of $a`$ will result in an $i`$ matrix that still includes influences from the values of *a* from the original convolution. From a mathematical perspective, the convolution of $a`$ with $i * a$ did not fully invert $i * a$ to return *i*.

Even though this method of determining the inverse only gives us a pseudo-inverse, it can still be useful. Despite the fact that this method results in a bloated matrix $i`$ based on our *i* input, all is not lost. Recall the observations noted above about the structure of the $i`$ matrix: the original values of *i* are present, followed by a certain number of zeros (depending on the size of $a`$), and finally the negated values of *i*. It is clear that the zeros and negated values of *i* are useless to any calculation that requires the inverse of a convolution. So why are they needed? They are in fact not needed. So what prohibits the truncation of *i`* to retain just the original values of *i*? Answer: nothing. The size of *i* is already known. So in order to extract the necessary data from *i`*, all that is necessary to do is to merely extract the first *n* elements of *i`*, and throw away all the rest of the values. This truncation will preserve the original values of *i*, and ensure that *i`* does indeed equal *i*.

This brief SciLab statement illustrates how to perform this operation:

```
-->i1

 i1  =

!   1.    2.    3.    4. !

-->ia

ia  =

         column  1 to 12

!   1.    2.    3.    4.  - 1.222D-15  - 9.966D-16  - 6.466D-17    3.827D-16  - 8.327D-
16  - 9.714D-16  - 1.055D-15  - 1

.055D-15 !

         column 13 to 20

! - 1.379D-15  - 1.746D-15  - 2.346D-16  - 4.760D-17  - 1.  - 2.  - 3.  - 4. !

-->new_i = ia(1:max(size(i1)))

 new_i  =

!   1.    2.    3.    4. !
```

Keep in mind that this is a *pseudo-inverse*. Due to this fact, the results that this method will give are not 100% accurate. A further discussion involving error analysis and the formation of these errors can be found in Section 1.5.2. The accuracy of this method can be demonstrated by computing difference between the original *i* matrix and the computed *i`* matrix:

```
-->new_i - i1
 ans  =
   1.0D-14 *
!   0.0666134    0.0888178  - 0.0888178  - 0.1776357 !
-->
```

The difference between *i* and *i`* are close to that of the ε of the machine. Even though these errors are small, they may affect the validity of any calculations that are based off these results. In most cases, basic calculations should not be affected to a large degree. But more complex calculations, and simply more numerous calculations, can start exhibiting growing errors. Unfortunately, this is the nature of computer systems today, being what they are. It is up to the user to determine what is an acceptable level of error. From there, the user can adjust and accommodate for those errors to help eliminate them, and provide more accurate results.

1.5. Convolution as a Computer Algorithm

1.5.1 Algorithm Implementation

Now that it is possible to compute the convolution $f * g$ of two functions f and g using matrices, it is quite trivial to develop a computer algorithm to perform the operation. Formula 1.4 provides the foundation for the computer algorithm, and gives a clear direction of how to program it. The algorithm can take advantage of this method directly, and compute the convolution of two matrices in the same manner as would a human being, yet it is optimized to work efficiently on a computer.

The algorithm can be written in any computer language. SciLab was specifically chosen to handle this algorithm because of its simple syntax and ease of manipulating matrices. Since the method outlined previously makes use of matrices, it makes most sense to utilize the matrix functions already provided with SciLab. This avoids having to write extra code to handle matrices specifically, and then having to write the algorithm on top of that code.

If the programmer decides to use a different programming environment other than SciLab, pseudocode can be composed using the method described in Section 1.3. From there, the programmer can port the algorithm to any machine platform as deemed necessary. The following convolution algorithm is the SciLab implementation, following the method outlined in Section 1.3:

```
function [ia] = myconvol(i,a)

//Date:           February 8, 2005
//Author:         Tom S. Lee
//Purpose:        My Implementation of 1-D Matrix Convolution
```

```
//ensure that i holds the longer matrix (between i and a)

if max(size(i)) < max(size(a)) then,
        temp = i;
        i = a;
        a = temp;
end;


//ia holds my answer for convolution

s=size(i,'c');
s2=size(i,'r');
if s2>s then
        i=i';
        s=s2;
end

if min(size(i)) != 1 then
end


t=size(a,'c');
t2=size(a,'r');
if t2>t then
        a=a';
        t=t2;
end

v=min(s,t);
h=s+t-1;        //h = length of convolved matrix ia

umax=h/2;       //umax is ½ total size of resulting matrix

//initialize matrix to hold convolved result
ia=zeros(1,h);

for u = 1:umax,
  starta = t-u;
  for k=max(1,u-v+1):u,

         //first half (left side)
         ia(u) = ia(u) + (i(k)*a(u-k+1));

        //second half (right side)
```

```
            ia(h-u+1) = ia(h-u+1) + (i(s-k+1) * a(starta + k));
            end;
end;

////////////////////////////////////////////////////////////////////////////////
//          For odd-length convolved matrices, calculate the middle element

if modulo(max(size(ia)),2) > 0 then

            mid = umax + 0.5;
            n = max(size(i));
            m = max(size(a));

            for k = m:-1:1,
                        index = (n/2) + (m/2) + 1 - k;
                        ia(mid) = ia(mid) + (i(index) * a(k));
            end;
end;
////////////////////////////////////////////////////////////////////////////////
endfunction
```

Here is the proper syntax for calling myconvol(), and the resulting output:

```
-->getf('C:\Imaging\myconvol.sci')

-->i=[1 2 3 4]

 i  =

!   1.    2.    3.    4. !

-->a=[1 2]

 a  =

!   1.    2. !

-->myconvol(i,a)

 ans  =

!   1.    4.    7.    10.    8. !

-->
```

Of course, the algorithm yields the same result that was obtained in Sections 1.2 and 1.3, as shown by the following output:

```
-->getf('C:\Imaging\myconvol.sci')

-->i=[1 2 3 4]

 i  =

!   1.    2.    3.    4. !

-->a=[1 2]

 a  =

!   1.    2. !

-->convolme(i,a)

 ans  =

!   1.    4.    7.    10.    8. !

-->
```

As stated earlier, this algorithm uses the same method used in Section 1.3 to compute the convolution of the input matrices. Instead of using *f* and *g* in the convolution, *i* and *a* are used. The naming convention used here will become more clear as the applications of convolution are discussed later.

There are three additional steps that are performed in this algorithm that were not performed in the method used in Section 1.3. Recall back to Section 1.3 that *g* was the shorter of the two functions/matrices, and was shifted over *f*, the longer of the two functions/matrices. To ensure that this condition is met with every execution of the algorithm, an initial test is performed:

```
//ensure that i holds the longer matrix (between i and a)

if max(size(i)) < max(size(a)) then,
        temp = i;
        i = a;
        a = temp;
end;
```

Here, *a* is assumed to be the shorter matrix, which is shifting over *i*, the longer matrix. This test here will check the sizes of *i* and *a*, and will swap the matrices if the size of *a* is greater than that of *i*.

The second additional step that is performed in this algorithm is found in the heart of the algorithm:

```
for u = 1:umax,
  starta = t-u;
  for k=max(1,u-v+1):u,

         //first half (left side)
         ia(u) = ia(u) + (i(k)*a(u-k+1));

        //second half (right side)
        ia(h-u+1) = ia(h-u+1) + (i(s-k+1) * a(starta + k));
        end;
end;
```

Here, the values of the convolved matrix *ia* are calculated. In Section 1.3, *g* was shifted over *f* from left to right. A closer analysis of this method will reveal its overall inefficiency. The number of steps that would be performed would be *size(i) + size(a) – 1*. What happens if *i* and *a* are both very long matrices? The algorithm already has a time complexity of $O(n^2)$, since there are two for loops iterating over the matrices. Therefore, it is quite advantageous to speed up this algorithm in any way possible. The most obvious method of speedup would be to shift *a* over *i* from both sides simultaneously. If this was to be applied to the method in Section 1.3, one iteration would look like:

```
        1    2    3    4
   2    1              2    1
```

For a human being, this method would quickly become cumbersome for larger matrices. However, this method is quite trivial for a computer to perform, so it is best to leave this method for the computer to utilize. Because of this optimization, only ½ of the shifts are required to compute the convolution, and our algorithm will therefore run twice as fast as before. The number of required calculations to determine the convolved matrix *ia* would be equal to: (*size(i)* + *size(a) – 1) / 2*. The *a* matrix is therefore shifted only ½ the distance on each side of *i*.

The third and final additional step performed in this algorithm is a result of the second step. Since the *a* matrix is only shifted ½ the distance on each side of *i,* there is a special situation that crops up as a result. If the size of *i* is odd, the middle element of *i* will never be considered by any element of *a*. Therefore, the middle element of *ia* will be zero. Graphically, this would appear as a hole in the middle of the overlap of *i* and *a*. Clearly this is not correct. This is illustrated by the following SciLab output:

```
-->convolme(i,a)
 ans  =
!   1.    4.    0.    10.    8. !
-->
```

The correct result should be [1 4 7 10 8]. But since the size of the convolved matrix is odd (size = 5), the middle element (at index 3) never gets calculated. This will yield the incorrect result for ½ of all the possible convolved matrices possible. In order to handle this special case, an extra step is necessary:

```
if modulo(max(size(ia)),2) > 0 then

        mid = umax + 0.5;
        n = max(size(i));
        m = max(size(a));

        for k = m:-1:1,
                index = (n/2) + (m/2) + 1 - k;
                ia(mid) = ia(mid) + (i(index) * a(k));
        end;
end;
```

Here, the index of the middle element of *ia* is calculated, and the proper terms of *i* and *a* are lined up to produce that middle element of *ia*. The step performed here is the same as the step performed in the two for loops in the heart of the algorithm. The only time that the algorithm will enter this piece of code is if the size of *ia* is odd, i.e., the modulus of the size of *ia* and 2 is not zero.

1.5.2 Algorithm Analysis

An interesting note concerning this algorithm is its accuracy. By no means is this the most accurate algorithm in the world. When compared to the built-in convolution function provided with SciLab, there are very small discrepancies. However, these discrepancies are very miniscule, on the order of 10^{-13} to 10^{-16}, depending on the machine used. These numbers represent the machine's accuracy, known as *epsilon* (ε). The ε of a machine determines the smallest number that the machine can represent. Floating point operations can only achieve a certain degree of accuracy, since there is a finite amount of memory storage space to store values. Divisions and multiplications of values can introduce fractional numbers that may continue forever (repeating digits). In order to fit the value into memory, the value must be rounded. This rounding error will affect the final answer. Since multiplications and divisions are sensitive operations, errors occur most when these operations are used. Therefore, it is logical to reduce the number of multiplications and divisions.

Horner's method uses nesting to aid humans in finding the zeros of a higher degree polynomial. This method breaks down polynomials in the following fashion:

Old Polynomial: $ax3 + bx2 + cx + d$

New Polynomial: $d + x (c + x (b + ax))$

Fortunately, Horner's method also greatly reduces the number of multiplications necessary to evaluate a given polynomial. The original polynomial requires a total of 8 multiplications to compute the final answer. The new nested polynomial only requires 3 multiplications to compute the final answer. Even though both methods will compute the same final answer in theory, the nested polynomial is more advantageous to use. Not only are fewer operations required, but the number of multiplications has been reduced to less than half of the original number. On a computer, this reduction in the number of multiplications greatly increases the accuracy of the nested polynomial. There will still be errors present in both methods; however, the nested polynomial will feature a lower degree of error than the original polynomial.

If we apply this knowledge to the computer algorithm for convolution, we can see the implications of using this ideal in the heart of the algorithm:

```
for u = 1:umax,
  starta = t-u;
  for k=max(1,u-v+1):u,

         //first half (left side)
         ia(u) = ia(u) + (i(k)*a(u-k+1));

        //second half (right side)
        ia(h-u+1) = ia(h-u+1) + (i(s-k+1) * a(starta + k));
        end;
end;
```

Here, two multiplications are performed for each iteration of the inner for loop. This means that, in terms of the overall time complexity of this algorithm (which was stated to be *$O(n^2)$* earlier), there are a total of *$2n^2$* multiplications performed for each execution of this algorithm. Due to the sheer number of multiplications necessary for this implementation, there will be some degree of error in our results. These errors are not enough to cause any large problems; however, the errors are still present. Quite surprisingly, when compared to the built-in convolution function convol() included with SciLab, this implementation is actually *more* accurate, despite the large number of multiplications performed to obtain the answer. This can be illustrated if the results of convol() and myconvol() are compared directly:

```
-->myconvol(i,a)

 ans  =

!   1.    4.    7.    10.    8. !

-->convol(i,a)

 ans  =

!   1.    4.    7.   10.    8. !

-->convol(i,a) - myconvol(i,a)
```

```
 ans  =

   1.0E-14 *

!    .0444089  -  .0888178    0.  -  .1776357    0. !

-->convol(i,a) - [1 4 7 10 8]

 ans  =

   1.0E-14 *

!    .0444089  -  .0888178    0.  -  .1776357    0. !

-->myconvol(i,a) - [1 4 7 10 8]

 ans  =

!   0.    0.    0.    0.    0. !

-->
```

These results were also repeated when using larger matrices

Theoretically, the result should be [0 0 0 0 0] for both implementations. An analysis of the SciLab convol() function's code was not possible at this time, so a direct comparison of the algorithms was not possible. However, it is clear that the method used by the SciLab convol() function allows errors to grow, and therefore the answer is not as good as that obtained by myconvol().

One possible reason for this could be the nature of the SciLab algorithm. The number of multiplications necessary to compute the convolved matrix in convol() may be greater than the number required by myconvol(). A conclusive answer cannot be obtained unless the code for convol() is examined, and the exact method used by convol() is analyzed. The execution times of convol() and myconvol() appear to be different. The convol() function seems to execute more quickly than myconvol(), yet yields a less accurate answer. Although myconvol() requires more execution time, the answer that it yields is more accurate. The actual execution times of each algorithm have not been quantified, so a quantitative analysis of each is not possible at this time. From a purely qualitative standpoint, there actually is a difference in execution times present between these two methods. This becomes more apparent as the size of the input matrices increases, and the number of calculations performed increases as a result.

This creates a nice situation for programmers. If the goal is to calculate the convolution as quickly as possible, and small errors are not important in the final answer, then programmers might want to consider using convol(). Choosing convol() in situations that involve large amounts of data will prove advantageous. myconvol() will still work, but will operate more slowly than convol(). If, on the other hand, the goal of the programmer is to achieve the most accurate results possible, and the amount of time to obtain those results is not important, then the programmer will want to choose myconvol(). The amount of error that is generated in the results is clearly less than the amount produced by convol(). Needless to say, myconvol() will be more useful in critical situations, where the results of convol() will be used in calculating more complex data.

In terms of image and signal processing, the accuracy of a crucial operation, be it noise filtering or data sampling, is directly related to the accuracy of the convolution method used. An inaccurate or crude convolution method will throw off the results of any image or signal processing operation. In these applications, data accuracy is of the utmost importance. Errors will throw off data results, and may produce a shoddy or erroneous outcome. Unfortunately, all the errors cannot be removed from any complex calculation. It is possible, however, to reduce the number of errors. The reduction of errors in a core operation such as convolution will improve the accuracy of the results of other more complex operations, especially those used in image and signal processing. Again, the amount of error that is deemed “acceptable” must be left up to the user. But it is important to keep in mind that the error is indeed present, so that the user will not wonder as to the reason for calculations not yielding expected values.

***NOTE: myconvol() is not compatible with SciLab version 3.0. An indexing error occurs in the heart of the algorithm, and this appears to be a bug specific to SciLab 3.0. What is even more peculiar is the fact that the code works without error when executed by itself. Once the code is used and accessed as a function, the indexing error occurs. The following is an example execution of myconvol() illustrating this error:

```
-->getf('C:\Imaging\myconvol.sci')

-->i=[1 2 3 4]
 i  =

!   1.    2.    3.    4. !

-->a=[1 2]
 a  =

!   1.    2. !

-->myconvol(i,a)
 !--error    21
```

```
invalid index
at line      51 of function myconvol               called by :
myconvol(i,a)
-->
```

The algorithm was tested in SciLab 2.7.2 and runs flawlessly. Until the issue is resolved in a later release of SciLab, please only run myconvol() in version 2.7.2..