Area Processes

So far we have only dealt with point processes. Area processes are the next class to consider.

In programming terms, point processes are easy to deal with since we use the old pixel value to calculate the new.

Area processes modify values depending on their existing value, AND surrounding values.

Many are based on a process called convolution.

Convolution involves sliding a mask over the image repeatedly performing sum of the products calculations between the mask and the values covered.

These bring their own problems (programming logic & others!)

Discrete convolution

Firstly, convolution generally operates on intensity.

If we work with monochrome images, this presents no problems, however for colour images, we either need to convolve the R G B channels individually (a bad idea really), or convert to HSI, convolve intensity, convert back. The second of these will give better colour results but involves greater computational effort.

YOU ARE ONLY REQUIRED TO CONVOLVE GREYSCALE IMAGES!

Secondly, it is important to place the generated pixels in a NEW image.

If we overwrite pixels in our working image, we create an “infinite impulse response” which will generate very strange results.
This happens because if we have a 3x1 mask and a row of pixels, the first operation will modify pixel 1 based on pixels 1 & 2, the second pixel 2 based on 1,2,3; the third pixel 3 based on pixels 2,3,4. As a result we feed some of the previous change into the next pixel.

We will be discussing convolution quite a lot. Whilst it appears difficult at first, it is important to remember that the only variables between all of the techniques we’ll discuss are:

- mask size
- coefficients
- edge handling
- offset

If you're sensible, you can produce a pair of generic convolvers, and simply recycle these.

If you have one function to perform integer convolution and another for floating point, then we can simply pass the variables given above to the convolver.

Due to the very large number of computations required to implement convolution, integer arithmetic should be preferred, some operations however will require floating point arithmetic, e.g. Frei-Chen edge detection.

Once you’ve got the idea, it’s easy to play around!

The Basic idea:

- A 2 dimensional mask is generated, usually containing an odd number of pixels (we won’t consider those with even numbers).

- If we don’t wish to affect image intensity, the mask should sum to 1, many edge detectors however sum to zero, in which case negative values may be generated.
  - Simplest solution – add a constant (e.g. max intensity/2 ), and if the sum is still less than zero, set to zero.
• How do we handle edges? (simplest pad with zeros). More complex “wrap” the image.

\[
\begin{array}{cccccc}
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 2 & 3 & 4 & 0 \\
0 & 5 & 6 & 7 & 8 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\quad
\begin{array}{cccccc}
8 & 5 & 6 & 7 & 8 & 5 \\
4 & 1 & 2 & 3 & 4 & 1 \\
8 & 5 & 6 & 7 & 8 & 5 \\
4 & 1 & 2 & 3 & 4 & 1 \\
\end{array}
\]

**First Example: Emboss using a 3x3 mask**

Many digital photography / imaging software products often filters. Most allow you to define "custom" filters. These are generally implemented using convolution.

Embossing is one of Adobe photoshop’s “stylise” filters. Unfortunately, photoshop only offers integer convolution kernels in its “custom filter”, however to see the sort of effect we’re looking for:

Custom filter:

\[
\begin{bmatrix}
-2 & 0 & 0 \\
0 & 4 & 0 \\
0 & 0 & -2 \\
\end{bmatrix}
\] with an offset of 128

will perform the transform on the next slide:
This is the effect that we will attempt to recreate first.

The shuttle image is available as a pgm file from: 
http://www.csm.uwe.ac.uk/~irjohnso/coursenotes/uqc146/shuttle.pgm.

(… and also from the images page) If you want to work with a different image, choose one with clear edges and high contrast for best effects.

If you want to work with a colour image, see what the image appears like when it is viewed as a greyscale.
The key element of an emboss convolution is the cancelling coefficients about the centre coefficient.

A standard embossing mask would be:

\[
\begin{bmatrix}
-1 & 0 & 0 \\
0 & 0 & 0 \\
0 & 0 & -1
\end{bmatrix}
\]

However this will lower the intensity – so we need to add a constant value as an offset.

Image fragment:

<table>
<thead>
<tr>
<th>168</th>
<th>168</th>
<th>168</th>
<th>168</th>
<th>168</th>
<th>168</th>
<th>168</th>
<th>168</th>
</tr>
</thead>
<tbody>
<tr>
<td>66</td>
<td>66</td>
<td>66</td>
<td>66c</td>
<td>66</td>
<td>40</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>20</td>
<td>20d</td>
<td>45a</td>
<td>45c</td>
<td>45</td>
<td>45</td>
<td>45</td>
<td>45</td>
</tr>
<tr>
<td>113</td>
<td>113g</td>
<td>113n</td>
<td>113i</td>
<td>113</td>
<td>63</td>
<td>63</td>
<td>63</td>
</tr>
<tr>
<td>155</td>
<td>155</td>
<td>161</td>
<td>161</td>
<td>161</td>
<td>161</td>
<td>161</td>
<td>161</td>
</tr>
<tr>
<td>155</td>
<td>155</td>
<td>155</td>
<td>155</td>
<td>155</td>
<td>123</td>
<td>123</td>
<td>123</td>
</tr>
<tr>
<td>27</td>
<td>27</td>
<td>27</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>113</td>
<td>113</td>
<td>113</td>
<td>113</td>
<td>113</td>
<td>113</td>
<td>113</td>
<td>116</td>
</tr>
<tr>
<td>155</td>
<td>155</td>
<td>155</td>
<td>155</td>
<td>155</td>
<td>155</td>
<td>155</td>
<td>155</td>
</tr>
</tbody>
</table>

In order to calculate the new value for the shaded pixel (e) we need to multiply each of the outlined pixels (a to i) by the corresponding value in the mask:

\[
66 \times -1 + \\
66 \times 0 + \\
66 \times 0 + \\
20 \times 0 + \\
45 \times 0 + \\
45 \times 0 + \\
113 \times 0 + \\
113 \times 0 + \\
113 \times -1 = -179
\]
we then add our offset (some authors call it a **bias**), in this case 128 and as the value is less than 0, clip the value to zero.

- We need to check values do not overflow or underflow.
- To get the best effect you'll need to experiment
- Adjusting the offset can have quite dramatic effects
- …But increasing the diagonal values in the mask will have the greatest effect

If you code the following (or something similar):

```c
char x[] = {-1,0,0,0,0,0,0,-1};
int mask_size = 9;
unsigned char offset=128;
convolve(x, mask_size, offset);
```

Then experimenting with altering mask and offset settings will be much easier.

**Blurring (correctly low pass spatial filtering)**

Essentially creates a “moving average”

Here all pixels in our mask may have equal values.

Removes noise (sudden changes).

The larger the mask, the greater the effect.

Blurring tends to lower contrast.

If we want to maintain levels, we need to sum to 1, however, if we multiply first by an integer, and then divide after we can avoid floating point arithmetic.
e.g.

\[
\text{Mask} = \begin{bmatrix}
1 & 1 & 1 \\
1 & 1 & 1 \\
1 & 1 & 1
\end{bmatrix}
\]

then divide by nine.

Using this approach, the larger the mask, the greater the effect.

**Gaussian smoothing:**

A more complex approach, here the "nearer" to the target pixel, the greater the effect. We are not going to go into detail here.

Gaussian smoothing is defined as:

\[
G[x, y] = e^{-\left(\frac{x^2 + y^2}{2\sigma^2}\right)}
\]

where \(x=0, y=0\) is the centre of your mask, and locations above and or left have negative \(x\) & \(y\) values, for example.

\[-1, -1 \quad 0, -1 \quad 1, -1 \]
\[-1, 0 \quad 0, 0 \quad 1, 0 \]
\[-1, 1 \quad 0, 1 \quad 1, 1 \]

A common 3x3 approximation (which you can implement if you wish) is:

\[
\begin{bmatrix}
1/16 & 1/8 & 1/16 \\
1/8 & 1/4 & 1/8 \\
1/16 & 1/8 & 1/16
\end{bmatrix}
\]

\(\sigma\) controls the size of the mask, which needs to be large enough to cover all non-zero coefficients. With \(\sigma\) equal to 3 a 23x23 mask is needed, \(\sigma = 2\) would require a 15x15 mask.

Small \(\sigma\) values = taller narrower covers (remember to draw)

In summary:
• removes noise
• costs detail
• damages edge detail
• the averaging filter is good at removing random noise, but not extreme values or "impulse noise".

**Median Filtering**

Median filtering is extremely similar, but reduces the effect of extreme values on the new pixel value chosen.

Again, the degree of effect depends on the size of the mask chosen.

Almost always used with an "odd" sized mask e.g. 3x3, 5x5 etc.

Method:
1) collect pixel values covered by mask
2) sort these values
3) new pixel value = middle value
   assuming unsigned char values[9],
   then when sorted values[4] would be chosen.

**Sorting**

In C is easy.

1) Write a compare function.
2) Call qsort, man qsort will explain usage.

See the next slide for an example!
**Example sort program**

```c
#include <stdio.h>
#include <stdlib.h>

int compare(const void *x, const void *y)
{
    unsigned char *a, *b;
    a = (unsigned char *) x;
    b = (unsigned char *) y;
    return (((*a) > (*b)) ? 1 : -1);
}

main()
{
    unsigned char values[9] = {47, 3, 2, 59, 9, 12, 1, 4, 99};
    int i;
    qsort(values, 9, sizeof(unsigned char), compare);
    for (i=0; i<9; i++)
        printf("%d ", values[i]);
}
```
Sharpening (correctly, high pass spatial filtering)

Emphasises contrast

Removes low frequency components. (i.e. “gradual changes”)

Common sharpening masks are:

\[
\begin{bmatrix}
0 & -1 & 0 & -1 & -1 & -1 & 1 & -2 & 1 \\
-1 & 5 & -1 & -1 & 9 & -1 & -2 & 5 & -2 \\
0 & -1 & 0 & -1 & -1 & -1 & 1 & -2 & 1 \\
\end{bmatrix}
\]

Another form of high pass filtering can also be implemented as high boost filtering.

Here:

\[
\text{HighBoost} = (\text{multiplier} \times \text{Original}) - \text{Lowpass}
\]

Where the multiplier is 1, we have a high pass filter, commonly known as unsharp masking. Most commercial unsharp making filters only act on a restricted range of pixel values however.

Where multiplier > 1, we are replacing some of the removed low frequency components, (restoring gradual changes that were originally removed) however we will also make the image lighter (since pixel values are being increased. Here a negative offset would reduce the lightening effect.
Whilst this can be accomplished with a single convolution:

e.g.

\[
\begin{pmatrix}
-1/9 & -1/9 & -1/9 \\
-1/9 & w/9 & -1/9 \\
-1/9 & -1/9 & -1/9
\end{pmatrix}
\]

where \( w = (9 \times \text{original}) - 1 \)

This would require a custom convolution function. Also note that our convolver would have to work with int values, since 9 times current pixel value will often be too large for an unsigned char.

Now is probably the time to talk about frame operations.

**Frame operations**

Frame operations are our third class of algorithm.

Frame operations involve two images. Often however for serious applications one of the images will be a mask.

Frame operations are easy to implement:

**Addition**

Usually a weighted sum to 1.

\[
(z \times \text{image1}[x,y] \& (1-z) \times \text{image2}[x,y])
\]

where \( z < 1 \) to maintain intensity.

**Subtraction**

Remove known noise, detect changes (common in industrial inspection).
Masking techniques

The following operations generally only make sense when one of the image is a mask, or has a region which is, since they alter bit values.

For example, we could:
1) "clear" part of an image by ANDing with 0's
2) Convert the remaining image to a mask (if bit != 0, bit = 1)
3) Invert (if bit = =1, bit = 0 else bit = 1)
4) Apply mask to another image of the same size (which would leave image information only in the areas cut out in the first set)
5) Then AND the results of 4) & 1)

AND (can be used to mask out parts of an image)

OR (can be used to add parts of another image)

Mean (or average) (useful for multiple noisy frames of the same image, for example when working with video)

Returning to Convolution:

We need to consider edge-detection.

Edge Detection

Edge detection is not strictly speaking a convolution operation however, it is extremely similar, when considered from a coding perspective.

Problem:

Different sorts of edges:

Vertical, Horizontal, Roof, Line, Step, Ramp, Circular etc.

Edges exist in different directions.