Final Project Report – Image Processing Techniques
UCLA Electrical Engineering
EE 113D Digital Signal Processing Design
Spring 2006
Professor Rajeev Jain
TA Rick Lan

Authors:
Cheng, Herman
Huang, Zhicong
Kumimoto, Mark
INTRODUCTION:
Image processing is the field of signal processing where both the input and output signals are images. Images can be thought of as two-dimensional signals via a matrix representation, and image processing can be understood as applying standard one-dimensional signal processing techniques to two-dimensional signals. Image processing is a very important subject, and finds applications in such fields as photography, satellite imaging, medical imaging, and image compression, just to name a few.

In the past, image processing was largely done using analog devices. However, as computers have become more powerful, processing shifted toward the digital domain. Like one-dimensional digital signal processing, digital image processing overcomes traditional analog "problems" such as noise, distortion during processing, inflexibility of system to change, and difficulty of implementation. The image processing technique we will be implementing will be image blurring. As the board we have does not support a direct connection for the input image, we will use MATLAB to output the image as a matrix and store it in the data memory of the DSP. To do this, we will use the parallel port connection to get our input data into the board. The DSP will then do the processing and write the output data in the program memory. We extract the output data and go back to MATLAB to analyze the results.

Our project methodology includes the following:
1. Use MATLAB to simulate the processing technique.
2. Implement the technique with assembly language on a TMS320C542-based DSKplus board to perform the same operation.
3. Carefully locating the memory blocks where we will store our original and output image.
4. Comparing our results from MATLAB and DSP outputs.

THEORY:
There are various ways of implementing the image blurring technique:
   i. **Linear blur** – horizontal or vertical averaging of a fixed number of pixels.
   ii. **Block blur** – averaging a small block of pixels by propagating a fixed sized window through the entire image.
   iii. **Gaussian blur** – convolution of the image with a two-dimensional Gaussian function.

*Linear blur*:
This is the simplest image blurring technique. It is done by taking the $N$-point average of a linear block of pixels (either horizontally or vertically). In our implementation, $N$ will be 8, and we will be using the horizontal blur. An $1 \times N$-pixel window is placed at the top left of the image, and the average of the window is stored in the $N/2^{th}$ pixel of the window (in a new image to prevent overwriting). The window is then shifted across the row and the process is repeated. Once the window reaches the end of the row, it is moved to the next row and the process repeats itself. See *linear_blur.m* for the MATLAB implementation of this algorithm.
The advantage of this method is that it is the simplest of the three. However, it also gives the poorest blurring quality. This is because by taking the horizontal average of each row, there will be averaging “lines” in the output image. Also, parts of the picture where the detail does not span enough horizontal pixels will be lost after blurring. Finally, by the way this algorithm is designed, there will be an outer frame of the output image identical to the input image (i.e. the outer part of the image remains not blurred).

**Block blur:**
This method is analogous to the linear blur, except that our window is now an $N \times N$-pixel window. The procedure is the same as the linear blur, with the averaged pixel stored in the $(N/2, N/2)$ position of the window. See `block_blur.m` for the MATLAB implementation of this algorithm.

This method improves upon the quality of the linear blur in that averaging “lines” are no longer visible in the output image. It also helps to retain details that span small horizontal distances in the original image better. However, it still does not overcome the problem of an outer frame in the output image that remains not blurred.

**Gaussian blur:**
This is the best implementation of the image blurring technique, and is used in such commercial software as Adobe Photoshop. Unfortunately, it is also the most complex. It works by performing a two-dimensional convolution on the input image with a normalized two-dimensional $M \times M$-pixel Gaussian function given by:

\[
f(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{(x^2 + y^2)}{2\sigma^2}}
\]

Intuitively, each pixel of the output image is actually a Gaussian function centered at each point of the input image. Hence, the convolution will increase the size of the output image to $N+M-1$, so that after convolution we must crop the image to reduce it to its proper size.

This method is the best of the three. It has no averaging “lines” present, and it also blurs the entire image. Its only weakness is that it is by far the most difficult to implement. See `gaussian_blur.m` for the MATLAB implementation of this algorithm.
**PROJECT DEVELOPMENT:**

**Code Size Issue**

From the DSKplus Memory Map, we determined that the total memory we have:

\[
(0800h-0180h) + (27FFh-1800h) = 680h + 1000h = 1680h
\]

or 5760 words. Since each pixel of an 8-bit grayscale image consumes 8 bits (half a word), we could ideally fit two pixels in each word, giving us twice the amount of memory compared to storing pixels to words in a 1:1 ratio. However, due to the complication of implementing a bit-mask on the DSP, (and more importantly the time constraint), we decided to abandon the idea of the 2:1 ratio storing possible with the bit-mask and just focus on programming the DSP without the bit-mask for now. As a result of mapping each word to each pixel of the image, the theoretical largest image we can store on the DSP is a \[\sqrt{5760} = 75 \times 75\]. Note that this does not account for the actual program code and any necessary temporary memory storage. Thus, this value is the absolute maximum and any practical application will necessitate a smaller image. To simplify things, we made the decision that all the image data should be stored in the second block of memory (1800h – 27FFh), thus each of the two images (see below for our argument on using two images instead of just one to prevent overwriting) could be

<table>
<thead>
<tr>
<th>Original image</th>
<th>Linear blur</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block blur</td>
<td>Gaussian blur</td>
</tr>
</tbody>
</table>

Figure 2. Comparison of Various Blurring Techniques
stored contiguously. With these restrictions, the maximum image size would be 45×45 pixels. In the end, we chose to implement our code for an image size of 32×32 pixels. This decision was based on the fact that using powers of 2 greatly simplifies the code, as a division or multiplication can be realized by a simple bit shifting to the right or left respectively.

*Image offset*
We originally wanted to implement the DSP to average an array of 8 pixels and store the result on the last pixel of the array. However, this resulted in a shifted version of the original image, as shown below.

![Figure 3. Shifted image](image)

We implemented a loop to connect the end pixel of each row to the starting pixel of the next row in order to efficiently blur the entire image (please refer to *Average Across Row* section below). Therefore, the only pixels that weren’t modified were the first 3 and the last 4 pixels of each row of the image.

*Overwriting of Output Image*
We initially aimed for storing a single 64×64 pixel image into DSP, but that implies whatever modifications we made would overwrite the original image, which would affect the averaging of pixels thereafter. Therefore, we decided to have 2 image stored in DSP so that we can retain the original image while averaging the pixels along the rows. This meant that we are no longer able to use a 64×64-pixel image, but rather two 32×32-pixel images.

To be exact, the original image is stored from 1800h-1BFFh, and the output image is stored from 1C00h-0x1FFFh (both size 400h). The following will illustrate the differences:

![Figure 4. Comparison of Output Image With and Without Overwriting](image)
| Original image | Single image stored in DSP, progressing with original image being overwritten | Twos image stored in DSP, averaging and copying the result to the other image instead of the original one |

Although we can hardly see the difference between the two image outputs, if we look close enough, we can see that the single-imaged process produced a dimmer image than the two-imaged process. This is because the single-imaged process averaged and overwrote the pixels from the original image, which would be further averaged by the next iteration. It was rather difficult to work with 32×32 images because the picture itself had poor detail due to its size. However, the emphasis is the actual result which must indicate that our method is valid and correctly tested.

**Border Issue**

Another important issue was the borders of the output image. Since our operation puts a window around the first 8 pixels of each row, averages them, and outputs the result to the 4th pixel, the first 3 pixels of the original image will be zero in the blurred image. The same case happens during the last iteration on a row, when the window propagates to the 8th to the last pixel, storing the result to the 5th to the last pixel, and leaving the last 4 pixels zero. Consequently, the output image will have 2 black stripes on both ends, for the first 3 pixels and the last 4 pixels of each row. In order to solve the problem, we simply load the first 3 and the last 4 pixels of each row from the original image to the output image.

![Figure 5. Image with Black Border](image)

**Averaging across Different Rows**

Another way of solving the problem was to convert the image matrix to a huge vector with 1024 values. Then we averaged across the vector as if it was a row in the matrix. By doing so, we could successfully propagate through all the data points instead of leaving the borders unfiltered. Although the resulting image looked fine, we soon realized that this operation theoretically changed the content of the original image by fusing the last 4 and the first 3 pixels of the rows into the same averaging window, which is inapplicable in our case of image processing.
Pseudocode of DSP Algorithm
for each row
    initialize buffer to hold first 7 values
    at the same time copy buffer into new picture

    for each pixel in row(after initial buffer)
        input next pixel into FIR
        divide sum by 8
        store average back into middle of window

    copy last 4 pixels of row into new pic

Some notes regarding the DSP code:
The row loop (labeled rowlp) loops the program through each row. Each row offset is calculated by multiplying the desired row by the number of pixels in each row. After calculating the offset, this value is added to the address of the beginning of the picture. This allows us to address the beginning of each row within the single large array, and process it as if it were a two dimensional array.

To address the output picture, the placement of the pointer is calculated based on the original picture, then the size of the whole picture is added. This allows us to place the second picture directly following the input picture.

When initializing the buffer, the 8 values are also copied into the location of the new picture. This prevents having to reread the first few pixels and copy them over to the new picture. At the end of each row, the last few pixels must be read a second time (the first being their input into the buffer for averaging).

In the case of an averaging FIR, it is common to make each coefficient equal to 1/ntaps. However, since the DSP will only allow integral numbers we set each coefficient to one, essentially giving us a sum. Then this sum is divided by the number of taps in the filter (in our case 8). By choosing a power of 2 for the number of taps, this is easily done by right shifting the sum log base 2 times.

Reading and writing to registers in rapid succession can lead to unreliable execution as physical limitations of the registers cannot keep up. Thus in the course of debugging our
code, we found it necessary to add a 'NOP' which stands for “No Operation.” This allowed for reliable execution of our code.

More details of the DSP code can be found in the comments of the asm file blur.asm.

DISCUSSION OF RESULTS:
MATLAB model versus DSP algorithm
The following shows the upper-left 16×16-pixel output of the MATLAB model and the upper-left 16×16-pixel output of the DSP algorithm.

Table 1. MATLAB Upper-Left 16×16 Output

<table>
<thead>
<tr>
<th>205</th>
<th>222</th>
<th>217</th>
<th>215</th>
<th>198</th>
<th>217</th>
<th>218</th>
<th>216</th>
<th>216</th>
<th>217</th>
<th>216</th>
<th>216</th>
<th>216</th>
<th>217</th>
<th>216</th>
<th>217</th>
<th>216</th>
</tr>
</thead>
<tbody>
<tr>
<td>218</td>
<td>216</td>
<td>211</td>
<td>214</td>
<td>213</td>
<td>214</td>
<td>215</td>
<td>215</td>
<td>214</td>
<td>213</td>
<td>212</td>
<td>211</td>
<td>213</td>
<td>212</td>
<td>210</td>
<td>210</td>
<td></td>
</tr>
<tr>
<td>225</td>
<td>214</td>
<td>216</td>
<td>218</td>
<td>217</td>
<td>217</td>
<td>215</td>
<td>209</td>
<td>203</td>
<td>198</td>
<td>192</td>
<td>188</td>
<td>184</td>
<td>182</td>
<td>184</td>
<td>190</td>
<td></td>
</tr>
<tr>
<td>217</td>
<td>216</td>
<td>226</td>
<td>218</td>
<td>215</td>
<td>207</td>
<td>191</td>
<td>173</td>
<td>155</td>
<td>136</td>
<td>115</td>
<td>95</td>
<td>82</td>
<td>78</td>
<td>84</td>
<td>99</td>
<td></td>
</tr>
<tr>
<td>214</td>
<td>220</td>
<td>228</td>
<td>210</td>
<td>195</td>
<td>174</td>
<td>150</td>
<td>130</td>
<td>111</td>
<td>98</td>
<td>89</td>
<td>77</td>
<td>68</td>
<td>67</td>
<td>70</td>
<td>76</td>
<td></td>
</tr>
<tr>
<td>223</td>
<td>226</td>
<td>224</td>
<td>197</td>
<td>176</td>
<td>151</td>
<td>128</td>
<td>105</td>
<td>85</td>
<td>78</td>
<td>80</td>
<td>78</td>
<td>76</td>
<td>76</td>
<td>74</td>
<td>73</td>
<td></td>
</tr>
<tr>
<td>220</td>
<td>222</td>
<td>214</td>
<td>176</td>
<td>152</td>
<td>128</td>
<td>105</td>
<td>83</td>
<td>67</td>
<td>69</td>
<td>84</td>
<td>98</td>
<td>101</td>
<td>101</td>
<td>100</td>
<td>99</td>
<td></td>
</tr>
</tbody>
</table>

Table 2. DSP Algorithm Upper-Left 16×16 Output

| 205 | 222 | 217 | 215 | 198 | 217 | 218 | 216 | 216 | 217 | 216 | 216 | 217 | 216 | 217 | 216 |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 218 | 216 | 211 | 214 | 213 | 214 | 215 | 215 | 214 | 213 | 212 | 211 | 213 | 212 | 210 | 210 |
| 225 | 214 | 216 | 218 | 217 | 217 | 215 | 209 | 203 | 198 | 192 | 188 | 184 | 182 | 184 | 190 |
| 217 | 216 | 226 | 218 | 215 | 207 | 191 | 173 | 155 | 136 | 115 | 95 | 82 | 78 | 84 | 99 |
| 214 | 220 | 228 | 210 | 195 | 174 | 150 | 130 | 111 | 98 | 89 | 77 | 68 | 67 | 70 | 76 |
| 223 | 226 | 224 | 197 | 176 | 151 | 128 | 105 | 85 | 78 | 80 | 78 | 76 | 76 | 74 | 73 |
| 220 | 222 | 214 | 176 | 152 | 128 | 105 | 83 | 67 | 69 | 84 | 98 | 101 | 101 | 100 | 99 |

In fact, upon subtraction of the MATLAB model image matrix with the DSP algorithm image matrix, we obtained a matrix of all zeros, which means that the DSP perfectly implements the MATLAB model.

Figure 7. Original and Output Images
Performance
This algorithm has $O(N^2)$ complexity, where $N$ refers to the number of pixels along the side of an $N \times N$-pixel image. The exact number of iterations is $N^2$ plus a constant, which shows the $O(N^2)$ complexity. The reason is that for each row each pixel must be read into the averaging filter as well as copying over the last few pixels, which requires re-reading them and copying them over, this is then repeated across each of the $N$ rows. This complexity is rather large, and, as we saw in our MATLAB model for a $512 \times 512$-pixel image, this took quite a long time to implement. One way to shorten the implementation time while still following this algorithm is to decrease the window size, but that would result in less blur in the output image.

Due to the time constraint and code complexity, we were not able to implement a bit-mask. In essence, this would double the space available on the DSP to allow larger images to be processed. Right now without the bit-mask we can simultaneously manage two images of only $45 \times 45$ pixels each. On the other hand, with a bit-mask we can manage two images of $64 \times 64$ pixels. This is double the number of pixels, but in terms of the actual visual "size" of the image, it is not a dramatic increase. Note that as described earlier, these are the absolute maximum sizes that can be stored without accounting for any program code, and thus are not particularly useful. Using the above restriction that our picture data fit into the second block of memory, we could possibly allow for a $64 \times 64$ image and a second processed image of the same size. One caveat though is that implementing the bit-mask procedure would require additional program code and one need be careful that all the program code and temporary storage fit in the available memory in the first block. Also note that this would add some additional overhead to the actual execution as extracting each pixel would take additional computing cycles as opposed to a direct read from memory.

In this case, the trade-off would seem worthwhile and with more time, adding a bit-mask would definitely be a feasible improvement to the code. However with the time frame we had, we believe that we made the right choice in omitting the bit-mask in favor of more time to debug our main program.
**CONCLUSION:**

Overall our project was very successful. We were able to perfectly implement the MATLAB linear blur model on the DSP, and were able to see that the DSP output corresponded to the MATLAB model output. We also went through many revisions of the DSP code and overcame many programming obstacles that we were not able to predict at the onset of the project. Some of these problems, such as inter-row averaging, is unique to image processing's two-dimensional nature.

Although we were never able to implement the block blur or the more complicated Gaussian blur, we did have working MATLAB code for both, and even C++ code for the block blur. With more time, the block blur algorithm is definitely a possible improvement that could be made. On the other hand, the Gaussian blur is much harder to implement, and would also require more memory as a lookup table must be stored for the values of the Gaussian function. Also, there remains the problem cropping the output image back to its original size which arises naturally from the size increase following convolution. However, as shown in Figure 2, the block blur and Gaussian blur produced images of comparable quality, while the block blur has a drastically simpler implementation. Hence for future projects, the block blur should be the best balance between blur quality and code complexity.

This project gave us some perspective on the tradeoffs between complexity and development time, as well as the importance of writing code that makes good use of the DSP's limited memory. We also learned to manage our project goals and make to cutbacks when necessary in order to meet the deadlines.
REFERENCES:
Electrical Engineering 113D Course Notes

Electrical Engineering 114D Course Notes

Two-Dimensional Gaussian Function:
http://astronomy.swin.edu.au/~pbourke/other/distributions/

Wikipedia Gaussian Blur:
http://en.wikipedia.org/wiki/Gaussian_blur

Wikipedia Gaussian Function:
http://en.wikipedia.org/wiki/Gaussian_function

Wikipedia Image Processing:
http://en.wikipedia.org/wiki/Image_processing
**CODE:**

**MATLAB**
- block_blur.m – model for 2D block averaging filter
- gaussian_blur.m – model for 2D convolution with Gaussian function smoothing filter
- image_extract.m – turns DSP raw output data into image file
- jpg2asm.m – converts JPEG (8-bit grayscale) to a format that can be input to the DSP
- linear_blur.m – model for 1D averaging filter

**C++**
- blockblur.cpp - C++ code for block blur
- 32.h – input image (32×32 star) in C++ input format

**DSP**
- blur.asm – DSP code for blur
- 32.asm – input image (32×32 star) in DSP input format
- 8tap.asm – 8-tap filter with all 1's