Gaussian Blur - ARM

Aran Nolan
nolanar@tcd.ie
implementation at: github.com/nolanar

Contents

1 Introduction

The outline of this assignment was to create an image manipulation effect and apply them to the TCD Crest. This was to be done using a ARM7TDMI microcontroller, and written the in ARM assembly language.

Figure 1: TCD crest without manipulation

Two candidate ideas were that of an unsharp masking, and a general convolution transformation. The latter seemed too general and would not leave much room for optimisation.

1.1 Unsharp Masking

This roughly works as follows:

- Blur the image.
- Subtract this from the original image giving you the unsharp mask. This is essentially a high-pass filter, resulting in only the 'edges' in the image.
- Use this to selectively contrast along the edges in the original image.

A good blur effect is central to this process. For a well implemented unsharpen masking, a clumsy box blur would not cut it.

1.2 Gaussian Blur

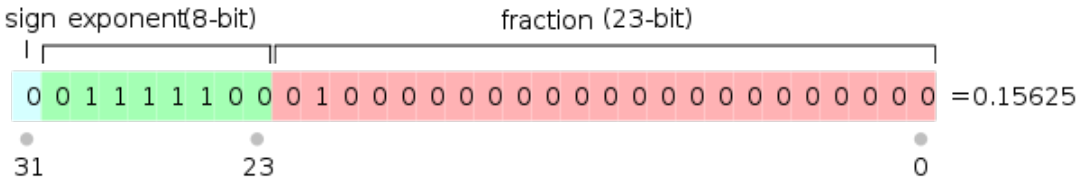
I decided that a 'better' blur was in order. However instead of creating this and using it in the un-sharpening mask, I would just focus on the blur but do something special.

2 Floating Point Arithmetic

The ARM7TDMI microcontrollers we are using for this assignment do not have hardware support for floating point arithmetic. It only seemed reasonable to implement it my self in software. This was an excellent learning experience, however I did not have time enough to refine many of the algorithms as much as I would have liked to. The explanations in this section will be kept brief, unless it is something really cool.

2.1 Structure

I cannot claim that my implementation is IEEE-754 compliant, but is as close as could be with the time invested. The floating point representation of a value is structured as is standard:



Rounding is done towards 0 for all implemented operations.

2.2 Operations

The standard operations of addition, subtraction, multiplication, and division were of course necessary:

Addition Two floating point numbers are added in the following way:

- The mantissa of the smaller of the two operands is shifted to make the exponents of the operands equal.
- The mantissas are then added. The resulting mantissa is associated with the exponent above.
- Any necessary adjustments are made to get this floating point representation in standard form.

Details of implicit 1's, exponent bias, and sign bit are left out here for the sake of brevity.

Subtraction By simply changing the sign of the second operand, this becomes equivalent to floating point addition.

Multiplication Multiplication almost seems more straight forward than addition. The mantissas are multiplied and the exponents are added.

One point of note is that the mantissas are 24 bits in length (with implicit 1 added on) thus the product of two such mantissas may require 48 bits which is beyond the standard 32 available. The `UMULL` instruction (unsigned multiply long) is used to work around this limitation.

Division Division is much the same as multiplication, however the division algorithm is implemented manually. The mantissa of one operand is divided into the other, and the exponents are subtracted. The division algorithm preforms binary long division:

```
quotient = 0
while (leading 1 not on bit index 23) {
  quotient << 1
  if (numerator >= denominator) {
    quotient += 1
    numerator -= denominator
  }
  numerator << 1
}
fDivAlg_eFor
return quotient
```

There is potential for some unnecessary loops to be carried out here (if the denominator is a divisor of the numerator), but the algorithm is neat when expressed this way.

2.3 Square root (and a bit of magic)

Despite floating point addition taking over 50 lines of code, while also calling other subroutines, the square root of a floating point number can be approximated within about an accuracy of 3% with the following two instructions:

```
LDR     r1, =0x1FBD1DF5
ADD     r0, r1, r0, LSR #1
```

where `r0` is the value we wish to take the square root of. This to me is magic! However, the explanation is even more so.

The approximation: The first observation required is the following:

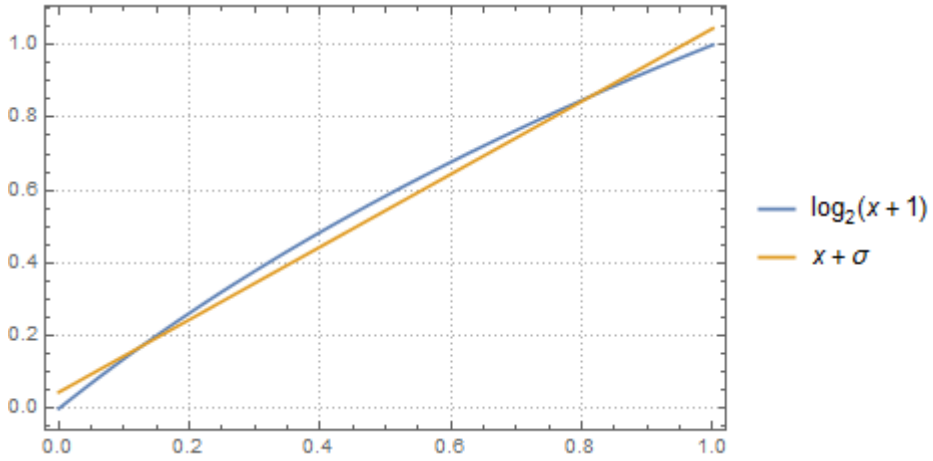


Figure 2: $log_2(x+1)$ vs $(x+\sigma)$, $\sigma = 0.0450465$

As the graph above demonstrates, x is a very close approximation to $log_2(x+1)$ on the interval $[0,1]$. This approximation is improved by adding a small error term σ which we take as 0.0450465.

Interpreting Floating Point: As we are taking the square root of a number, we shall forget about the sign bit.

Let (uppercase) M (mantissa) represent the first 23 bits of a floating point, and E (exponent) represent the next 8 bits. For generality, let $L = 23$. Let us relate this to the following expression for a value n:

$$n = (1+m)2^e \tag{1}$$

where m is a value in the interval $[0,1)$, e an integer.

If n was to be represented in floating point format ($\ll$ denotes logical shift):

$$Fl(n) = E \ll L + M \tag{2}$$

M and m are related in the following way:

$$m = M \gg L$$

as this will move the binary point one place in front of the most significant bit of M, which is precisely what m is (recall that L is the number of bits in

M).
E and e are related in the following way:

$$e = E - B$$

Where B is the exponent bias (equal to 127 as standard).

The Trick: Let us consider $y = \sqrt{x}$. This can be written as $y = x^{\frac{1}{2}}$. Taking log_2 of both sides:

$$\begin{aligned} log_2(y) &= log_2(x^{\frac{1}{2}}) \\ &= \frac{1}{2} log_2(x) \end{aligned}$$

Substituting x and y for expresions of the form eqn (3):

$$\begin{aligned} log_2((1+m_y)2^{e_y}) &= \frac{1}{2} log_2((1+m_x)2^{e_x}) \\ log_2(1+m_y) + e_y &= \frac{1}{2}\left(log_2(1+m_x) + e_x\right) \end{aligned}$$

Using the approximation show above:

$$\begin{aligned} m_y + \sigma + e_y &\approx \frac{1}{2}\left(m_x + \sigma + e_x\right) \\ m_y + e_y &\approx \frac{1}{2}\left(m_x + e_x\right) - \frac{1}{2}\sigma \end{aligned}$$

Substituting lowercase varables for their upper case equivalent:

$$\begin{aligned} M_y \gg L + E_y - B &\approx \frac{1}{2}\left(M_x \gg L + E_x - B\right) - \frac{1}{2}\sigma \\ M_y \gg L + E_y &\approx \frac{1}{2}\left(M_x \gg L + E_x\right) + \frac{1}{2}\left(B - \sigma\right) \end{aligned}$$

Right shift everything by L:

$$M_y + E_y \ll L \approx \frac{1}{2}\left(M_x + E_x \ll L\right) + \frac{1}{2}\left(B - \sigma\right) \ll L$$

At this point something wonderful has happened. Floating point representations, expressions of the form eqn (4), have come about through integer operations:

$$Fl(y) \approx \frac{1}{2} Fl(x) + \frac{1}{2}\left(B - \sigma\right) \ll L$$

Also, the second term on the RHS consists just of constants:

$$\frac{1}{2}(B-\sigma) \ll L = \frac{1}{2}(127 - 0.0450465) \ll 23 = \texttt{0x1FBD1DF5}$$

And we arrive at our two line approximation for the square root of a floating point number:

$$Fl(y) \approx Fl(x) \ll 1 + \texttt{0x1FBD1DF5}$$

2.4 Fast Multiplication and Division

Carrying out similar steps on floating point multiplication and division leads to similar approximations. These are present as the subroutines `fMulFast` and `fDivFast` in the code, however are not used anywhere.

Also as a final point of note, there is nothing special about the exponent of x here. This method could be used to approximate the value of x^p just as well!

2.5 Refining Result

To improve the approximation, Newton-Raphson method can be used. Two iterations of Newton-Rapson is all that is required to improve the accuracy to the precision of a floating point number.

3 Program Structure

`loadToWorkspace` **subroutine:** A region of memory is designated as the "workspace". Before an image can be manipulated it must be loaded into the workspace. This splits the red, green and blue channels of the image into separate blocks of memory. This allows each channel to be independently manipulated. The color values are in floating point representation in the workspace, allowing for far finer control in manipulation. The `loadToWorkspace` subroutine preforms this transfer.

`loadFromWorkspace` **subroutine:** When Finished manipulating an image, to display the image it is moved back into the region of memory from which the source image came. `loadFromWorkspace` subroutine handles this. In the process it collects the individual color channels back into RGB pixels.

4 The Big Idea

The target is a Gaussian blur effect. This however is an expensive operation (if done directly). Each of the entries of the Gaussian convolution matrix would have to be calculated, and then application of the convolution would be quadratic in complexity. Seeing as we have implemented floating point with software, this would be exceedingly slow for a blur of any appreciable size. Clearly better must be done!

4.1 Box Blur to Gaussian Blur

Due to the Central Limit Theorem, a Gaussian blur can be approximated by a number of iterations of Box blurs. Using this knowledge we can avoid having to calculate the entries of a Gaussian convolution matrix.

4.2 Linear blur to Box blur

Another significant optimisation that can be made is in how we calculate the Box blur. One box blur is equivalent to a x-axis motion blur with a y-axis motion blur. However, this is linear in complexity as opposed to quadratic. For large blurs, this should make a significant difference.

Due to the commutativity of convolutions, the necessary x-axis blurs can be calculated together, and then the y-axis blurs preformed. This is how I have implemented this effect.

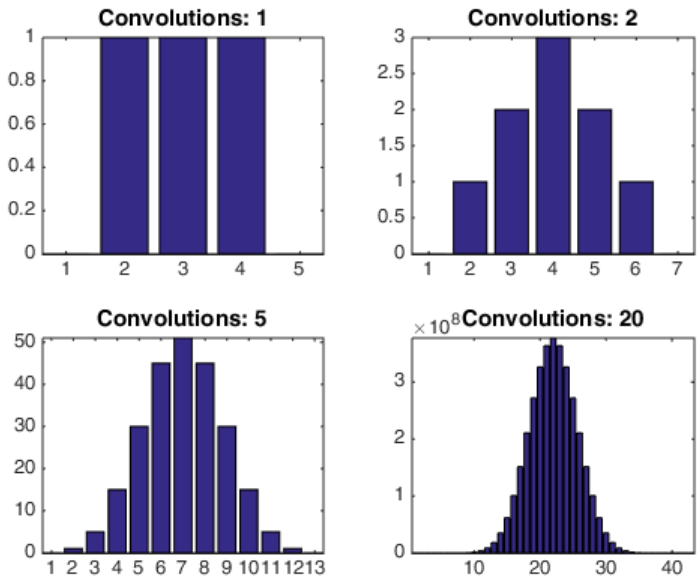


Figure 3: Demonstration of box blurs approx. Gaussian blur

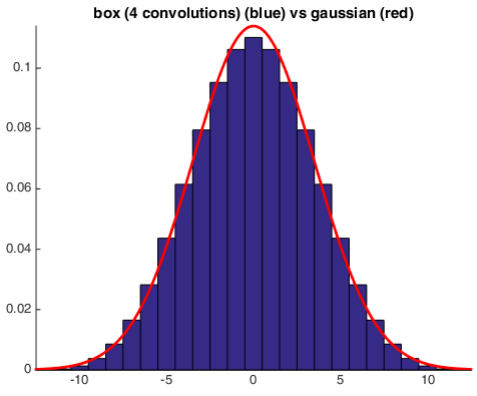


Figure 4: 4 iterations (blue) gives a good approx. of true Gaussian (red)

5 User Input

The radius of a Gaussian blur is typically defined by it's standard deviation. An increase in the number of iterations of Box blurs will increase how approximate the blur is to a true Gaussian blur. Because of this, the standard deviation and the number of iterations should be the values the user inputs, and the program should calculate the radius of the required box blur.

5.1 Edge Terms

As the specified SD (standard deviation) is continuous, there must be a continuity to the radius of the box blur. To achieve this, edge terms have been introduced to the traditional Box blur. These edge terms take a fractional part of the pixel they overlap.

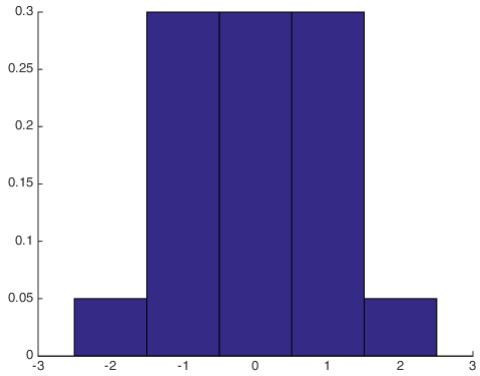


Figure 5: Graphical representation of edge terms with full terms

5.2 Calculating Box Radius

The following will briefly explain how the integer part of the radius (full terms) and fractional part of the radius (edge terms) are calculated. A more rigorous analysis would be provided if not for a lack of time.

5.3 Integer part of Box Radius

Without any edge terms, the variance (σ^2, σ = standard deviation) of a Box blur is given by:

$$\sigma^2 = 2\sum_{k=1}^{l} k^2 = \frac{2l^3 + 3r^2 + l}{3} = \frac{L^2 - 1}{12}$$

where l is box radius. Box diameter given by $L = 2l + 1$. D iterations of box blur has a variance of $D\sigma^2$, let r = radius of Gaussian blur:

$$r^2 = D\sigma^2 = D\frac{L^2 - 1}{12}$$

Solving for L:

$$L = \sqrt{12\frac{r^2}{D} + 1}$$

Substituting in l and taking the integer part:

$$l = \left\lfloor \frac{\sqrt{12\frac{r^2}{D} + 1} - 1}{2} \right\rfloor$$

Which gives the integer part of the radius of the required box blur. This is the calculation preformed by the `boxIntRadius` subroutine.

5.4 Fractional part of Box Radius

With edge terms of height ϵ, the variance of a box blur with D iterations is given by:

$$D\sigma^2 = D\left(2\sum_{k=1}^{l} k^2 + 2\epsilon(l+1)^2\right)$$

Solving for ϵ with the use of the above calculations and definitions we find:

$$\epsilon = \frac{(2l+1)(3r^2 - dl(l+1))}{6(d(l+1)^2 - r^2)}$$

This is the calculation that the `boxFracRadius` subroutine preforms.

Regrettably I have had to leave out much of the justification for these few steps. Hopefully it gives the reader a feel for what is going on at the least.

6 Gaussian blur

With the fractional part and the integer part of the box radius calculated, we are ready to implement the Gaussian blur. This is built up one step at a time:

`linearSum` **subroutine:** This subroutine calculates the linear sum (without scaling down at end to find average) of a single line (in x or y direction depending on parameters) of a single color channel. More on this subroutine below.

`linearColorBlur` **subroutine:** This subroutine iterates `linearSum` over the axis orthogonal to `linearSum` thus carrying out a linear blur on an entire color channel. It is also at this point that the scaling to find the averages is carried out.

`linearBlur` **subroutine:** This subroutine applies `linearColorBlur` to the three color channels of the image in the workspace. This results in a full linear blur on an image.

`gaussianBlur` **subroutine:** This subroutine completes the process. It iterates `linearBlur` the requested number of times, and applies this in both the x and y directions.

6.1 `linearSum` subroutine

The last thing that requires inspection is how `linearSum` works. In Part 2, we had to temporarily store the transformed image to avoid interference as we applied the motion blur. However, in the name of efficiency we harness this interference.

The idea is that we can calculate the sum of the terms $S_1 = \{+ : x_1, x_1, \ldots, x_n\}$ if we know the sum of terms $S_0 = \{+ : x_0, x_1, \ldots, x_{n-1}\}$, x_0, and x_n. In that case

$$S_1 = S_0 - x_0 + x_n.$$

The edge terms also have to be included in our case, but follow a similar idea:

$$\begin{aligned} B[i] = B[i-1] \quad &+ \quad \epsilon * px[i+r] + px[i-1+r] - \epsilon * px[i-1+r] \quad (3) \\ &- \quad \epsilon * px[i-r] - px[i+1-r] + \epsilon * px[i+1-r] \quad (4) \end{aligned}$$

Where $B[k]$ is the transformation of the k^{th} pixel, $px[j]$ is the value of the j^{th} pixel, and ϵ is the height of the edge terms (or fractional part of the box radius).

Queue implementation: For this I required a Queue. A Byte of memory is designated as the space for the queue. The queue cycles through these available memory addresses. This limits the queue to a size of 64, which allows for a blur radius of 32 due to how the Queue is utilised.

The queue has subroutines to add an element, remove (and return) an element, peak at the front element, and reset the queue.

Using the Queue The first pixel in the `lineSum` subroutine must be manually calculated (similar to part 2) and the Queue put in the appropriate state. This acts as the initialisation to the recursion.

For the remaining iterations:

- The terms in eqn (6) are found at the head of the queue in this order. To retrieve these values and leave the queue in the appropriate state for the next iteration, the first two values are removed and the third term is peaked at.
- $px[i + r]$ is added to the queue and then $\epsilon * px[i + r]$ is added, leaving the in the queue appropriate state for the next iteration.

Advantages There are two advantages to doing it this way:

1. We avoid having to build the transformation in another memory location and then copy it back.
2. The time complexity of the algorithm is independent (after initialisation) of the box radius.

This is more efficient in both space and time.

7 Result

The Gaussian blur results in a much smoother and more pleasant blur than the Box blur:

Figure 6: Standard box blur (left) vs Gaussian blur ($\sigma = 3.5$) approx by 4 iterations of box blur (right)