FAST AND SOMEWHAT ACCURATE ALGORITHMS

MARIO BARELA, SU CHEN, EMILY GUNAWAN, MORGAN SCHREFFLER, MINHO SONG, DOYEOB YEO, AND MENTOR: CHAI WAH WU (IBM), TEAM 4

1. Introduction and Problem Statement

In applications such as image processing, computer vision or image compression, often times accuracy and precision are less important than processing speed as the input data is noisy and the decision making process is robust against minor perturbations. For instance, the human visual system (HVS) makes pattern recognition decisions even though the data is blurry, noisy or incomplete and lossy image compression is based on the premise that we cannot distinguish minor differences in images. In this project we study the tradeoff between accuracy and system complexity as measured by processing speed and hardware complexity.

emily: This paragraph above is exactly what Chai wrote in the IMA page, so we probably need to paraphrase it.

We seek to produce algorithms using look-up-tables (LUT) for speeding up filter operations (such as edge-detection, sharpening, blurring, and embossing filters). In order for any LUT-based algorithm to be useful, the size of the LUTs need to be reasonably small. To meet this aim, we use the following strategies.

(1) a byte-truncation technique,
(2) convolution decompositon of a kernel 3×3 matrices (rank one),
(3) approximate convolution decompositon of a kernel 3×3 matrices (rank two)
(4) and taking advantage of the observation that edge-detection algorithm shows higher tolerance to byte-truncation.

2. Background

[WSLQE06] discusses fast algorithms using look-up tables (LUT).

An h-by-w-pixel image file can be thought of as a three-dimensional array IMAGE $= [p_{ijk}]_{h\times w\times 3}$ whose entries are integers between 0 and 255. Given pixel (i, j), the three entries p_{ijk}, $k = 1, 2, 3$, determine the color of the pixel in some colorspace. When the image is grayscale, let IMAGE $= [p_{ij}]_{h\times w}$, and let the entries indicate the pixel's intensity (0 for black, 255 for white). For the sake of simplicity, we will only consider grayscale images in this paper.

2.1. Filtering images using kernel convolutions.

When filtering an image, first choose a matrix $K = [k_{ij}]_{n\times n}$ with double-precision entries, where n is an odd number greater than 1. The matrix K is called the *filter kernel* or *convolution kernel*. In this paper we usually only consider 3×3 and 5×5 kernels.

Date: August 5-14, 2015.

Next, we compute the *matrix convolution* $F = K * \text{IMAGE}$, a $\max\{h,n\} \times \max\{w,n\}$ matrix whose ijth entry is

$$f_{ij} = \begin{cases} \displaystyle\sum_{q=(1-n)/2}^{(n-1)/2} \sum_{r=(1-n)/2}^{(n-1)/2} k_{i+q,j+r} p_{i-q,j-r} & \text{if } 1 \le i+q \le h,\ 1 \le j+r \le w \\ 0 & \text{otherwise.} \end{cases}$$

To eliminate the piecewise nature of determining f_{ij}, one can first "pad" IMAGE with a frame of zeros that is $\frac{n-1}{2}$ entries thick. Observe that this definition of matrix convolution coincides with the usual discrete convolution, but with two-dimensional indexing instead of one-dimensional. In particular, it is associative and distributive.

2.2. **Truncation methods.** Recall that the value of a pixel is an integer between 0 and 255, which is an 8-bit number. Given an $n \times n$ filter kernel K, we say that $T = [t_{ij}]_{n\times n}$ is a *truncation matrix* if its entries are all integers between 0 and 8. When performing a kernel convolution on an $n \times n$ block IMAGE_B of IMAGE, T indicates that we ignore (or truncate) the last t_{ij} bits of the ijth entry of IMAGE_B when applying K.

emily: Chai suggested we run through a suite of images, for example, maybe from the Kodak website, and confirm that the errors are similar.

2.3. **Measures of errors.**

Definition 2.1. Given two images $A = [a_{ij}]_{h\times w}$ and $B = [b_{ij}]_{h\times w}$, we will define ℓ^2-difference(A,B) and ℓ^∞-difference(A,B) by

$$\ell^2\text{-difference}(A,B) := \sqrt{\frac{\sum_{i,j}(a_{ij}-b_{ij})^2}{h\times w}},\ \text{and}$$
$$\ell^\infty\text{-difference}(A,B) := \max_{i,j}|a_{ij}-b_{ij}|.$$

The PSNR (in dB) [wikipedia/PSNR] is defined as

$$\begin{aligned} \text{PSNR}(A,B) &= 10\cdot\log_{10}\left(\frac{\text{MAX}_B^2}{\text{MSE}}\right) \\ &= 20\cdot\log_{10}\left(\frac{\text{MAX}_B}{\sqrt{\text{MSE}}}\right) \end{aligned}$$

where

$$\text{MSE} = (\ell^2\text{-difference}(A,B))^2 \qquad \text{and} \qquad \text{MAX}_B = 255.$$

Remark 2.2. In general, MAX_B is the maximum *possible* pixel value of an image. Since we are assuming each pixel is an 8-bit integer, MAX_B will be 255 for our purposes.

Remark 2.3. PSNR can be easily calculated by using MATLAB command psnr(A,B). PSNR is the abbreviation for Peak Signal-to-Noise Ratio. In brief, it shows us how different two images are. As PSNR is higher, it becomes harder to recognize the difference between original image and filtered image. Typical values for the PSNR in lossy images are between 30 and 50 dB, provided the bit depth is 8 bits.

Definition 2.4. The *IMage Euclidean Distance (IMED)* is given by

$$d^2_{\text{IMED}}(A,B) = \frac{1}{2\pi}\sum_{i,k=1}^{h}\sum_{j,\ell=1}^{w}\exp\left(\frac{-(k^2+\ell^2)}{2}\right)\big(a_{ij}-b_{ij}\big)\big(a_{i+k,j+\ell}-b_{i+k,j+\ell}\big).$$

Remark 2.5. See [FWZ05] for an analysis of why IMED might be a reasonable image distance. Another reasonable image distance may be $\frac{d_{\text{IMED}}}{\sqrt{h \cdot w}}$.

3. Challenges

On Wednesday August 5, we decided that look-up tables are not suitable for a median filter algorithm.

Although high-pass filters seem to respond well to truncation, blurring filter does not respond well to truncation.

4. Results

4.1. **Truncation and Look-up tables (without decomposition).**

4.1.1. *A High-pass kernel.*

4.1.2. *Sobel Magnitude Operation.* $\mathbf{G} = \sqrt{{\mathbf{G}_x}^2 + {\mathbf{G}_y}^2}$

where

$$\mathbf{G}_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * \mathbf{A}.$$

Here $*$ denotes the 2-dimensional convolution operation.

Let $K_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}$. A full look-up table $\text{LUT}(K_x)$ for K_x requires 256^6 bytes to store. In contrast, a look-up table $\text{LUT}(K_x)_T$ with truncation matrix $T = \begin{bmatrix} 5 & 8 & 5 \\ 5 & 8 & 5 \\ 5 & 8 & 5 \end{bmatrix}$ takes up 8^6 bytes.

Let $\mathbf{G}_T$ denote the Sobel magnitude operation using the truncated look-up table $\text{LUT}(K_x)_T$. For images in our test suite, the PSNR values between $\mathbf{G}$ and $\mathbf{G}_T$ range between 25 and 32.

emily: I will add the range of errors from the picture [26.5950] [25.8214] [31.0469] [30.6689]

4.2. **Decomposing rank 1 matrix.**

chensu: high-pass filter

emily: Sobel gradient filter

4.3. **Approximate decomposition of a rank 2 matrix.**

mario: Approximate decomposition of a rank

4.4. **Decomposing a Kernel Matrix by Rows.** Let K be an $n \times n$ filter kernel. For $k = 1, \ldots, n$, let e_k be the $n \times 1$ matrix whose $k1$th entry is 1 and whose other entries are 0, and let r_k be the $1 \times n$ matrix whose entries are the kth row of K. Note that K may always be decomposed as $K = \sum_{k=1}^{n} r_k * e_k$. Taking advantage of the associativity of $*$, it

follows that $K * P = \sum_{k=1}^{n} r_k * (e_k * P)$. This is just another way of writing that summation is distributive, i.e.,

$$\sum_{q=(1-n)/2}^{(n-1)/2} \sum_{r=(1-n)/2}^{(n-1)/2} k_{i+q,j+r} p_{i-q,j-r} = \sum_{q=(1-n)/2}^{(n-1)/2} \left(\sum_{r=(1-n)/2}^{(n-1)/2} k_{i+q,j+r} p_{i-q,j-r} \right).$$

If using lookup tables to compute a matrix convolution, this observation greatly reduces the amount of memory required. Indeed, rather than having a single LUT which takes n^2 inputs and contains 2^{8n^2} entries, we have n LUTs (one for each row of K), each of which takes n inputs and contains only 2^{8n} entries. Even when $n = 3$ this is a tremendous saving of memory (with 8-bit entries, one 4-zebibyte (2^{72}) LUT versus three 16MB (2^{24}) LUTs).

Further, the multiple-LUT approach can be easily implemented, and at a lower cost than direct computation. Before filtering any pixels, rotate K by 180^o about its center. Then, take an $n \times n$ submatrix P_{temp} of P centered about the pixel to which you wish to apply the kernel. Now simply enter the ith row of P_{temp} into the LUT associated with r_i for $i = 1, 2, \ldots, n$, and add the results. Thus, if we make the naive assumption that lookups and additions are computationally equivalent, only $2n - 1$ operations (n lookups and $n - 1$ additions) are required per pixel. Compare this to $n^2 + 2n - 2$ operations (n^2 multiplications and $2(n - 1)$ additions) per pixel using direct computation.

Finally, when combined with bit truncation, each LUT can be made even smaller at the cost of an extra n^2 truncation operations per pixel. Unfortunately, if truncation is assumed equivalent to lookups and additions, this is not much more efficient than direct computation, since it saves only one operation per pixel.

As an example, choose an arbitrary 3×3 kernel K and a truncation matrix $T = \begin{bmatrix} 5 & 5 & 5 \\ 5 & 2 & 5 \\ 5 & 5 & 5 \end{bmatrix}$, using the ith rows of K and T to produce the ith LUT. The associated LUTs will have 512, 4096, and 512 entries. Even if each entry uses double precision, only 20kB of memory is required to store the LUTs, which is certainly small enough to fit into cache memory. However, 14 operations (5 to use the LUTs as described previously, and 9 truncations) are required per pixel, versus 13 for direct computation.

4.5. **Using the truncating technique for a blurring filter: Taking advantage of the fact that truncating works better for edge-detecting than for blurring.**

e a sen-
e here.

Remark 4.1. (1) The truncation method T can be used to reduce running time.
(2) Truncation introduces too much error for blurring, but is decent for edge-detecting.

Let I, L, and H denote the identity, a low-pass filter operation, and a high-pass filter operation on an image. Let $L(T)$, and $H(T)$ denote L and H with a truncation using the truncation matrix T. To informally say that applying a truncation technique to a high-pass filter gives close enough result, we can write $H \approx H(T)$.

Since $I = L + H$, we have $L = I - H \approx I - H(T)$.

Example 4.2. Consider a high-pass filter $H = \frac{1}{8} \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$. Let

$$T_1 = \begin{bmatrix} 8 & 4 & 8 \\ 4 & 2 & 4 \\ 8 & 4 & 8 \end{bmatrix}, T_2 = \begin{bmatrix} 8 & 5 & 8 \\ 5 & 3 & 5 \\ 8 & 5 & 8 \end{bmatrix} \text{ and } T_3 = \begin{bmatrix} 8 & 6 & 8 \\ 6 & 4 & 6 \\ 8 & 6 & 8 \end{bmatrix}$$

be truncation matrices. The table1 shows us how $L(T_i)$ and $I - H(T_i)$ are different from the low-pass filter $L = I - H$ for each i. $L(T_i)$ is Old and $I - H(T_i)$ is New in the table1.

	T_1		T_2		T_3	
	Old	New	Old	New	Old	New
PSNR	38.12	39.58	30.80	32.84	22.59	25.59
l_2 - error	3. 06	2.80	7.40	5.87	18.87	13.52
l_∞ - error	8.42	7.40	19.64	15.81	43.35	35.70

TABLE 1. Comparison of errors

As you see, our new approach gives us the better results no matter which truncation matrix you use. See also Figures 1 made via the truncation matrix T_3.

(A) LPF (B) LPF with Truncation (C) Result of the New Idea

FIGURE 1. Comparison of filtered images

As you see, you can easily check that our new idea reduces staircase effects against the usual truncation method.

Likewise, we can apply our method to any other low-pass filter as follows:

1. Let L be the our target low-pass filter.
2. Set $\tilde{H} := I - L(T)$ for a certain truncation T.
3. Find $\tilde{L} := I - \tilde{H}$ as an approximation of L.

Note that the second and third step stands for avoiding staircase effects.

4.6. **Approximate Decomposition.**

5. FURTHER QUESTIONS

6. INSTRUCTION FOR USING COMMENTS (WHILE WE ARE DRAFTING THE REPORT)

morgan: to write a comment to Morgan

mario: to write a comment to Mario

minho: to write a comment

chensu: to write a comment

emily: to write a comment

doyeob: to write a comment

chaiwah: to write a comment

7. Thursday August 6: Daily Notes

7.1. **Compute an algorithm using a look-up table for a high-pass kernel matrix.** $\frac{1}{8}\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$ using a truncation technique using truncation matrix $\begin{bmatrix} 8 & 4 & 8 \\ 4 & 2 & 4 \\ 8 & 4 & 8 \end{bmatrix}$. This particular look-up table was created in about 15-19 seconds.

7.2. **Compare results of linear filters on Matlab with and without truncating digits.** Truncating with the following truncating matrices look acceptable for gray-scale images: $T_0 = \begin{bmatrix} 0 & 4 & 0 \\ 4 & 0 & 4 \\ 0 & 4 & 0 \end{bmatrix}, T_1 = \begin{bmatrix} 0 & 4 & 0 \\ 4 & 1 & 4 \\ 0 & 4 & 0 \end{bmatrix}$ $T_2 = \begin{bmatrix} 0 & 4 & 0 \\ 4 & 2 & 4 \\ 0 & 4 & 0 \end{bmatrix}$ $T_3 = \begin{bmatrix} 0 & 5 & 0 \\ 5 & 3 & 5 \\ 0 & 5 & 0 \end{bmatrix}$ $T_4 = \begin{bmatrix} 0 & 3 & 0 \\ 3 & 1 & 3 \\ 0 & 3 & 0 \end{bmatrix}$.

A normalized gradient magnitude from Sobel-Feldman operator [Wikipedia/Sobel] $\mathbf{G} = \sqrt{{\mathbf{G}_x}^2 + {\mathbf{G}_y}^2}$

where

$$\mathbf{G}_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * \mathbf{A}.$$

Here * denotes the 2-dimensional convolution operation.

Some high-pass kernel matrices:

- $\frac{1}{8}\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$
- $\frac{1}{8}\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$

A low-pass kernel matrix:

- $\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$

A Scharr Gradient kernel matrix:

- $\frac{1}{32}\begin{bmatrix} 3 & 10 & 3 \\ 0 & 0 & 0 \\ -3 & -10 & -3 \end{bmatrix}$

An unsharp mask kernel filter:

- $\frac{1}{8}\begin{bmatrix} 0 & -\lambda & 0 \\ -\lambda & 1 - 0.3 * \lambda & -\lambda \\ 0 & -\lambda & 0 \end{bmatrix}$ where $\lambda = 0.1$.

7.3. **Matrix-decomposition?** Started looking into doing matrix decomposition to make the LUT smaller.

8. Friday August 7

Group meeting

(1) Testing filters with truncation
(2) Generating LUT to match with above.
(3) Decomposition into multiple LUTs.
(4) $L \approx Id - H(T)$
(5) Color (RGB) filters (see what Matlab does with edge detection for color pictures to make sure we are doing filters correctly.)
(6) Sobel / Scharr magnitude operator (see what Matlab does with color).

8.1. **A rank 1 decomposition.** The usage of look up tables (LUTs) provides a way to reduce the implementation time caused by direct algebraic computations. By storing the values of convolution of all possible square matrices with the kermel matrix K, the look up table serves as a dictionary whose values can be retrieved readily. The potential challenge for LUTs lies in the size which can be so large that storing them in the computer memory become impracticle. As an example, for a 3×3 kernel K with all entries non-zero will require $S = (2^8)^9 = 2^{72}$ bytes storage, which is fall beyond the storing capability of modern computers.

$$\begin{bmatrix} x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 \\ x_7 & x_8 & x_9 \end{bmatrix} * \begin{bmatrix} k_1 & k_2 & k_3 \\ k_4 & k_5 & k_6 \\ k_7 & k_8 & k_9 \end{bmatrix} \longrightarrow \begin{bmatrix} y_1 & y_2 & y_3 \\ y_4 & y_5 & y_6 \\ y_7 & y_8 & y_9 \end{bmatrix}$$

A remedy to this problem is to use truncated versions of the LUTs. Namely, we drop a prescribed number of bits from the entries in the square matrix that will be convolved with the kernel. As an illustration, a binary number 11111011 will become 11111000 after we drop the last two digits, i.e. replacing them by 0's. In this case, the original set $\{0, 1, \ldots, 255\}$ is turned into the set $\{0, 4, 8, 12, \ldots, 252\}$ after the truncation by two bits.

Although the above truncation is able to reduce the size of LUTs substantially, it is still not enough to generate LUTs which is acceptable in practice, especially for LUTs constructed for low pass filter with 5 by 5 or even larger kernel. To deal with this issue, we introduce the decomposition of the kernel so that the LUTs for each decomposition matrix is small enough to be generated and stored in the memory.

We first examine the simple case when $rank(K) = 1$ where $K \in \mathbb{R}^{n \times n}$ and n is an odd number. It is well known that every rank 1 matrix can be decomposed as

$$K = b \cdot c^T$$

for some $b, c \in \mathbb{R}^{n \times 1}$. Denoting the matrix convolution by $*$, the following relationship can be verified directly:

$$K = b \cdot c^T = b * c^T$$

So is the equation

$$Image * K = Image * (b \cdot c^T) = K * (B * C) = (K * B) * C$$

Here $B = (0, \ldots, b, \ldots, 0)$ and $C = (0, \ldots, c, \ldots, 0)$ where 0 represents $n \times 1$ column vectors and b,c are in the center. We need to point out that the above relationship may not be true for general kernel which is not rank one. This decomposition allows us to build two LUTs

for the kernel K, each of which is a table of the column vector b or c since we can ignore all 0's. The size of LUTs are reduced significantly in this way.

As an example, let us consider the following low pass filter kernel

$$K = \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

It is apparent that $rank(K) = 1$ and an obvious decomposition of K is given by

$$K = \frac{1}{4}\begin{bmatrix} 0 & 1 & 0 \\ 0 & 2 & 0 \\ 0 & 1 & 0 \end{bmatrix} * \frac{1}{4}\begin{bmatrix} 0 & 0 & 0 \\ 1 & 2 & 1 \\ 0 & 0 & 0 \end{bmatrix} \triangleq B * C$$

The symmetric decomposition of K, that is, $C = B^T$, simplifies the LUT further: only one LUT is required, since

$$Image * C = Image^T * B$$

The normalizing constant $\frac{1}{4}$ rules out the possibility of overflow and the maximum size (no truncation) of the LUT is $(2^8)^3 = 2^{24}$ bytes, or 16Mb. Truncation will further reduce the size to magnitude of kilobyte. Numercal test is given in subsection 8.3.

8.2. **Singular value decomposition.** Notice that the relationship (???) does not hold for general matrix K even if K can be decomposed as $K = B \cdot C$. However, it is well known that any matrix can be written as the sum of a series of rank 1 matrices, i.e. we have the following singular value decomposition (SVD):

$$K = \sum_{i=1}^{r} \sigma_i u_i \cdot v_i^T$$

Here u_i,$v_i \in \mathbb{R}^{n\times 1}$ are the left and right singular vectors respectively and σ_m are singular values. Moreover, u_i's and v_i's compose of orthornormal matrices and r is the rank of K. Now we are able to use the rank 1 LUTs as in section 8.1. As a result, at most $2r$ LUTs are required, each of which has a maximum size of $(2^8)^n = 2^{8n}$ bytes. For example, a 3 by 3 kernel with rank 2 needs at most 4 LUTs with maximum size of 16Mb for each. The benefit of SVD is that for most kernels in practice, their rank (usually 1 or 2) is far less than their dimension so that very few LUTs are required even if the kernel has large dimensions.

8.3. **Numerical test.** In this section, we implement the ideas in 8.1 and 8.2 and perform two numerical tests. We first examine the kernel

$$K = \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} = \frac{1}{4}\begin{bmatrix} 0 & 1 & 0 \\ 0 & 2 & 0 \\ 0 & 1 & 0 \end{bmatrix} * \frac{1}{4}\begin{bmatrix} 0 & 0 & 0 \\ 1 & 2 & 1 \\ 0 & 0 & 0 \end{bmatrix} \triangleq B * C$$

8.4. **Sobel magnitude filter.** Ran an RGB picture with Mario's color code and the results look similar as the one in Wikipedia (but not as nice), and the results look acceptable even with truncating to 5,6 digits.

9. Saturday

9.1. **A Decomposition Approximation.** Recall that a rank 1 decomposition of our kernal K had the form

$$K = b * c^T$$

where $c, b \in \mathbb{R}^n$. This resulted in two LUTs of size 2^{24} bytes instead of the full LUT of size 2^{72} bytes. This drastically reduces the size of the LUT before any truncation but it requires us to have a rank 1 kernal K. What can we do for a filter that is not rank 1?
Question: Can we decompose the filter K in a better way?
Consider the decomposition of our (3×3) filter K into the convolution:

$$K = A * B$$

where A and B are of the form

$$A = \begin{bmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 0 \end{bmatrix}, B = \begin{bmatrix} g & h & i \\ j & k & l \\ 0 & 0 & 0 \end{bmatrix}$$

The main reason we would like a decomposition of this form is that due to the number of 0 entries in A and B we can automatically reduce the size of our full LUT of 2^{72} bytes to two LUTs of size 2^{48} bytes. In general this decomposition is not possible but we may be able to find an approximate decomposition by posing this as a minimization problem. The problem is

$$\min \|K - A * B\|_F$$

Notice that that are other ways we can arrange the variables $a, b, .., l$ in the matrices A and B. There are in fact 9 choose 6 possibilities for each of the two matrices. This gives us a total of $C(9, 6)^2 = 84^2 = 7056$ possible minimization problems to solve. This seems like quite a lot of computation but this problem will actually be part of pre-processing. It is important to note the the minimizing values $(a, b, c, ..., k, l)$ may be 0 and so we may further reduce LUT size. For example, the decomposition may lead to A having only 2 nozero entries and B 4 nonzero entries. This would lead to two LUTs of size 2^{16} and 2^{32} bytes respectively.

We can actually change the structure of the matrices A and B above to have a variable amount of nonzero entries in them. This will allow us to force more entries in the matrices to be zero resulting in smaller LUTs.
Note: After solving all possible minimization problems for a particular decmoposition we will have a vector *fmin* and a matrix *xmin* containing all the min function values and minimizers. To get the "best" decomposition we look at the smallest function value. Since this will be an approximate decomposition we should look at all function value under a threshold

Ran an algorithm using fmin

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} = \begin{bmatrix} a & d & 0 \\ b & e & 0 \\ c & f & 0 \end{bmatrix} * \begin{bmatrix} g & h & i \\ j & k & \ell \\ 0 & 0 & 0 \end{bmatrix}$$

$$A = \begin{bmatrix} -0.372739294911747 & -0.443690961374357 & -0.457110102971229 \\ -0.509997537573061 & 2.683304035822869 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$B = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 2.681646089142412 & -0.509741093569434 \\ -0.456856367067759 & -0.443428387610633 & -0.372501463359094 \end{bmatrix}$$

conv2(A,B,'same')

$$\begin{bmatrix} -0.999554872469786 & -0.999821595552767 & -0.999640004082445 \\ -0.999646676848929 & 8.000063237563769 & -0.999819205287410 \\ -0.999737147772875 & -0.999878353018540 & -0.999534679981381 \end{bmatrix}$$

norm(F-conv2(A,B,'same'),'fro')

ans =

$9.063656510413503e-04$

10. ideas about what to do (Emily)

- Filter color images with these http://r0k.us/graphics/kodak/

11. Monday

emily: todo: Check errors $||L(A-B)||_F$ where L is a low-pass filter. Our eyes are essentially a low-pass filter. Chai thinks that PSNR-HVS is essentially this.

PSNR-HVS is proposed based on ... "Many studies confirm that the HVS is more sensitive to low frequency distortions rather than to high frequency ones.". That is, PSNR-HVS has similar philosophy with $||L(A-B)||_F$.

References

[WSLQE06] Wu, C. W., Stanich, M., Li, H., Qiao, Y., Ernst, L., "Fast Error Diffusion and Digital Halftoning Algorithms Using Look-up Tables," Proceedings of NIP22: International Conference on Digital Printing Technologies, Denver, Colorado, pp. 240-243, September 2006.

[Wikipedia/Sobel] Sobel operator

[wikipedia/PSNR] PSNR(Peak Signal-to-Noise Ratio)

[FWZ05] Jufu Feng, Liwei Wang, Yan Zhang, "On the Euclidean Distance of Images," IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 27, no. 8, pp. 1334-1339, 2005.