PDE-based Image Compression

Markus Peloquin, Leland Jefferis
{markus@cs,jefferis@math}.wisc.edu

Dec. 18, 2009

1 Introduction

Image compression is a branch of computer science concerned with taking an arbitrary image and compressing it down to its smallest possible size without loosing to much information about the exact starting image. The purpose of image compression is to reduce the file size of an image so that it may be transfered more quickly over a network or simply take up less space in storage.

In general, different compression methods are better suited to specific types of images (say photographs or computer graphics). Galić et al. focus on photographs [2]. Some of the methods being tested in her paper are ones that we reproduced in more detail than the paper supplied, and are described below.

2 PDE-Based image compression

The general approach to these methods is laid out in the following steps:

1. Take the starting image and select a subset of the pixels to be stored according to some algorithm.
2. *Compression* Store these pixels in as condensed a way as possible.
3. *Decompression* Interpolate these stored pixels using a PDE smoothing operator of some kind to restore the image over the whole grid.

2.1 Selecting a subset of pixels

This step of the process we will approach in two different ways. Two simplistic methods are to take a random scattering of pixels in the original picture, or to uniformly select them. However, it will turn out to be more beneficial to choose a denser concentration of points near the natural occurring edges in the picture; in other words, areas where the image's color gradient is changing quickly. There is a preexisting algorithm for selecting these points called the *B-tree triangular coding* method [1].

2.1.1 B-tree triangular coding

The BTTC algorithm is a simple and efficient algorithm to sample points closest to areas of highest curvature. It works by dividing the image's surface into right triangles. Whenever a triangle's linear interpolation is too far from the original image by some threshold ϵ, the triangle is subdivided into two right triangles. The points chosen are then close to the areas of high curvature.

The time complexity is $O(n \lg n)$ for an image with n pixels. The B-tree structure also allows for efficiently doing the linear interpolations of all the triangles ($\Theta(n)$), though this is inconsequential for our goals.

2.2 Storing the pixels

We will not bother with optimizing this step since we are mostly interested in the recovery process in the following step. General-purpose data compression like DEFLATE or LZMA can be used to compress the representation.

2.3 PDE interpolation

Recovering the image is done by treating the final image as the steady state of some diffusion process taken over the data points. Generally speaking we write

$$\partial_t u = L[u] \tag{1}$$

Where $L[u]$ is some kind of smoothing operator. For the sake of our tests, we chose to try two of the three suggested operators in the paper, namely the Laplacian $\nabla^2 u$ and the biharmonic $-\nabla^4 u$. As mentioned before, we solve the above system for the steady state solution by setting the left hand side of (1) to zero. But before we move to setting this up, we first need to talk about how we are going to decompose the image.

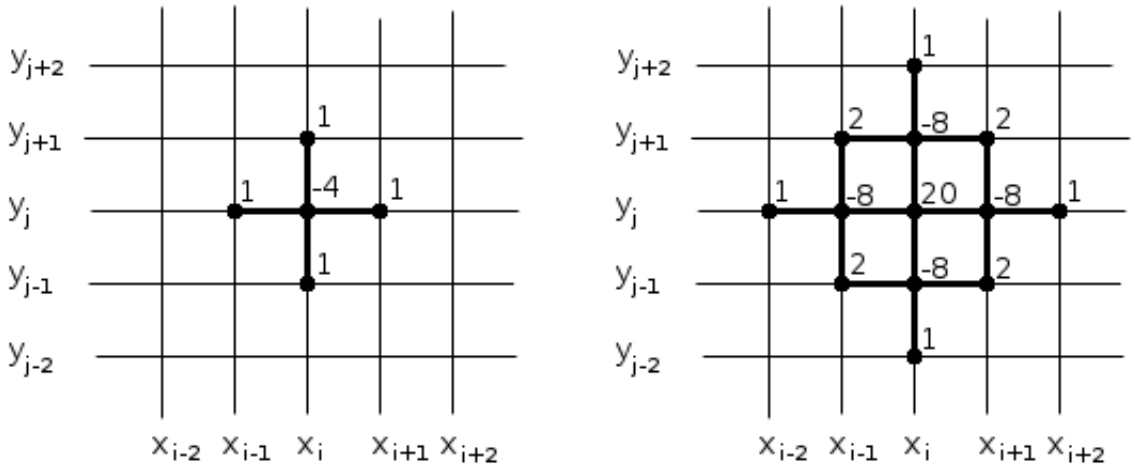
Since we are dealing with color images, suppose that we had a 512×512 pixel image. We break this down into its red, green, and blue layers so that we have three 512×512 grids of data to store. After the subset of pixels has been chosen, we will interpolate the points for each of the layers individually and then put them back together to recreate the original image.

With this in mind, we set out to discretize the smoothing operators in order to recover our image. Note that for now, we assume that we have already selected our subset of pixels by some method, random or otherwise.

3 Discretization of the smoothing operators

To discretize the Laplacian, we used the standard five point stencil. To discretize the biharmonic operator, we had to use a 13 point diamond with the weights as seen in Figure 1.

Figure 1: Stencils



(a) 5 point for ∇^2

(b) 13 point for ∇^4

We claim that this 13 point discretization for the biharmonic operator is second order accurate which we justify below with error analysis and numerical simulation.

First we start with error analysis. Note that the biharmonic operator can be written:

$$\nabla^4 u = u_{xxxx} + 2u_{xxyy} + u_{yyyy}. \tag{2}$$

So we need to find discretization for each of the 3 pieces in (2) and then add them together. For the u_{xxxx} term we use the following discretization:

$$u_{xxxx}(x_i, y_j) \approx \frac{1}{h^4}(U_{i-2,j} - 4U_{i-1,j} + 6U_{i,j} - 4U_{i+1,j} + U_{i+2,j}). \tag{3}$$

Similarly,

$$u_{yyyy}(x_i, y_j) \approx \frac{1}{h^4}(U_{i,j-2} - 4U_{i,j-1} + 6U_{i,j} - 4U_{i,j+1} + U_{i,j+2}). \tag{4}$$

The above is an extension of the second order central difference with the coefficients given by alternating sine binomial coefficients. A quick Taylor expansion will verify the truncation error is indeed order h^2. To get the $2u_{xxyy}$ term we could start with a general 9 point stencil and do Taylor expansions and although we did do this, there is a cleaner way using previously derived results. We use the formula for the 5 point and 9 point stencil given in LeVeque [3].

$$\nabla_5^2 u = \nabla^2 u + \frac{1}{12}h^2(u_{xxxx} + u_{yyyy}) + O(h^4)$$
$$\nabla_9^2 u = \nabla^2 u + \frac{1}{12}h^2(u_{xxxx} + 2u_{xxyy} + u_{yyyy}) + O(h^4)$$

If we take a difference we can get a formula for $2u_{xxyy}$:

$$\nabla_9^2 u - \nabla_5^2 u = \frac{1}{12} h^2 2u_{xxyy} + O(h^4)$$
$$\frac{12}{h^2}(\nabla_9^2 u - \nabla_5^2 u) = 2u_{xxyy} + O(h^2).$$

Doing the above algebra gives

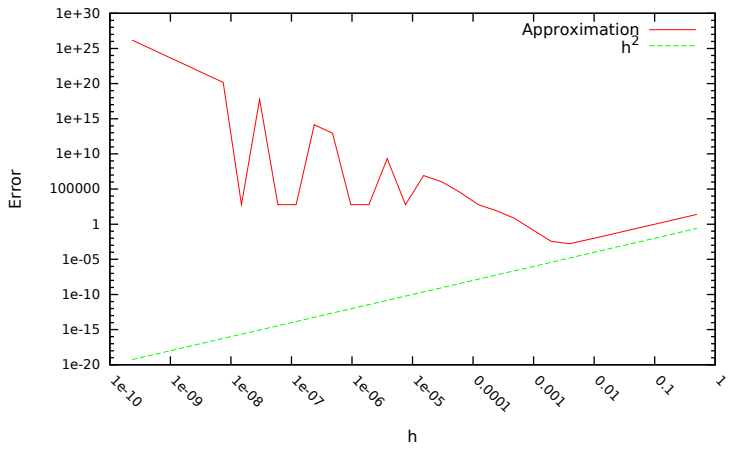
$$2u_{xxyy}(x_i, y_j) \approx \frac{1}{h^4}\Big[8U_{i,j} - 4(U_{i+1,j} + U_{i-1,j} + U_{i,j+1} + U_{i,j-1}) \\ +2(U_{i+1,j+1} + U_{i-1,j+1} + U_{i+1,j-1} + U_{i-1,j-1})\Big].$$

Now combining everything together:

$$u_{xxxx}(x_i, y_j) + 2u_{xxyy}(x_i, y_j) + u_{yyyy}(x_i, y_j) \approx \frac{1}{h^4}\Big[20U_{i,j} \\ -8(U_{i+1,j} + U_{i-1,j} + U_{i,j+1} + U_{i,j-1}) \\ +2(U_{i+1,j+1} + U_{i-1,j+1} + U_{i+1,j-1} + U_{i-1,j-1}) \\ +(U_{i+2,j} + U_{i-2,j} + U_{i,j+2} + U_{i,j-2})\Big]. \tag{5}$$

(5) corresponds exactly to the stencil shown in Figure 2(b). Note that from the above analysis we expect that this method be order h^2. To verify this we ran a numerical test on the function $f(x, y) = e^{x+y}$, comparing the exact solution with the numerical result at the point $(3, 2)$. The error for various grid resolutions is shown in Figure 2. The result verifies

Figure 2: h vs. error on a log-log scale



that the method is order $O(h^2)$. Note that the machine error becomes an issue near $h = .001$ because in the code for the biharmonic stencil, there is division and multiplication by h^4.

3.1 Note on implementation

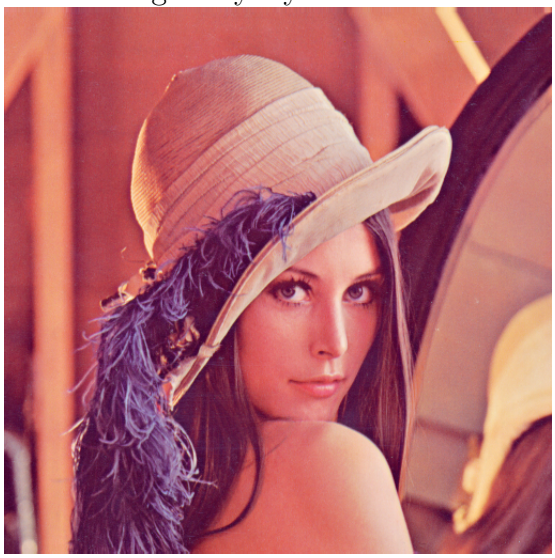
For boundary conditions, the paper's suggestion was to use *mirrored* boundary conditions. This means that at each of the boundaries, the image's pixels across the boundary would appear in reverse. At the right boundary, for example, $U_{N+1,j} = U_{N,j}$ and $U_{N+2,j} = U_{N-1,j}$, where N is the grid size. We instead chose to repeat the boundary pixels outward since it was simpler, and *near*-boundary pixels would have less influence over the boundary pixels.

Now finally, we use the Gauss-Seidel with SOR to solve for the linear system that arises from our discretization. This was the approach used by Galić, and it was also very convenient to implement since the linear system is more implicit. In practice, a more efficient solver may be used, but we were more interested in the quality of the image. The Python implementation of biharmonic Gauss-Seidel is included in Appendix A.

4 Results

As was mentioned earlier, there are multiple ways to choose the subset of pixels that will be stored in the compression step. We experimented with both choosing random pixels and storing pixels according to the BTTC algorithm. Below is a summary of results where we vary the pixel selection method, the smoothing operator used and the percentage of total pixels stored for compression. All test were run on *Lena*, the conventional test image in image processing.

Figure 3: Lena Söderberg: Playboy centerfold from November 1972

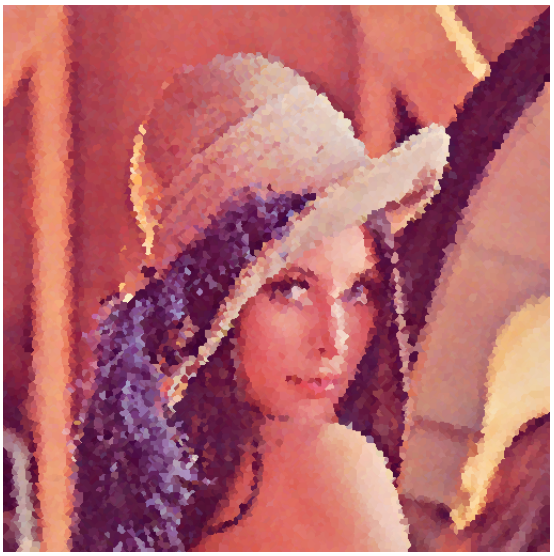


4.1 Pixel selection

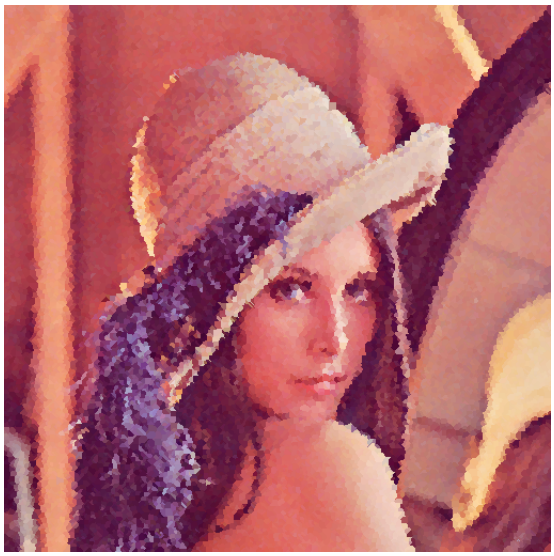
After selecting the set of pixels either randomly or according to BTTC, we create an image based from those pixels by assigning to every pixel in the grid the color of its closest neighbor

in the chosen set. As it happens this is simply the L_1 Voronoi diagram.

Figure 4: Voronoi Diagrams



(a) Random: 5% of pixels stored



(b) Random: 7% of pixels stored (so same count as BTTC_{32})

(c) BTTC, $\epsilon = 32$

(d) BTTC, $\epsilon = 16$

Figure 4 demonstrates that the BTTC algorithm better resolves the naturally occurring edges in the image than a random or uniform selection of pixels.

4.2 Harmonic operator—∇^2

The results of applying the harmonic operator to each of the four initial conditions in the previous section are shown in Figure 5.

Figure 5: Harmonic Smoothing

(a) Random: 5% of pixels stored

(b) Random: 7% of pixels stored

(c) BTTC, $\epsilon = 32$

(d) BTTC, $\epsilon = 16$

The images created using BTTC are superior to the ones created with random points. The most notable difference being near the natural edges of the image. Also note that in Figure 6(c), Lena has lines on her shoulder that are residual from the BTTC initial condition.

4.3 Biharmonic operator—∇^4

The results of applying the biharmonic operator to the four initial conditions are shown in Figure 6.

Figure 6: Biharmonic Smoothing

(a) Random: 5% of pixels stored

(b) Random: 7% of pixels stored

(c) BTTC, $\epsilon = 32$

(d) BTTC, $\epsilon = 16$

Again, the images created using BTTC are superior to the ones created with random points. In Figure 7(c) we notice some strange dark artifacts appearing around the circumference of Lena's hat and that the image is in general blotchy when compared to the harmonic smoothing. But these artifacts are reduced significantly when the error threshold is lower as

in Figure 7(d).

4.4 Error analysis

Table 1 shows the error in the image approximations. It should be noted that the compressed files have the same number of points, and so should have the same compressed size. In terms of the average error (essentially the 1-norm), the best approximation is biharmonic with biharmonic smoothing, but you would not know by looking at it. Visually, the BTTC approximations look the best. This is confirmed by its low RMS error (essentially the 2-norm). In particular, the BTTC images with harmonic smoothing both have the lowest RMS error and the best perceived quality.

	7% Random		BTTC_{16}	
	Average	RMS	Average	RMS
Harmonic	7.50	12.9	8.23	11.4
Biharmonic	6.81	12.2	9.25	14.7

Table 1: Errors of image approximations

5 Conclusion

In doing error analysis for the two smoothing operators, it was found that in the case of choosing random pixels, the RMS error was very similar. In the case of using BTTC, however, the biharmonic seemed to have a significantly higher RMS error. But since we only did tests with a single image, it is hard to draw any solid conclusion.

The biggest criticism of PDE-based compression that we have is in the implementation. We used Python to code the PDE solver and found that decompressing an image took on the order of minutes with results that are not nearly as good as JPEG. The BTTC_{16} image is 83% larger than a perceptibly perfect JPEG compression. The LZMA-compressed size of the BTTC_{16} is 29.5% of the PNG, which is slightly encouraging.

Perhaps with optimization of the code in C/C++, use of a faster PDE solver, and a better smoothing operator, this method for compression could start to approach usefulness. But even with the more sophisticated smoothing operator used in [2], the paper admitted that what they had accomplished was, at that stage, purely for expository purposes and needed refining. Even so, at the end of [2], the authors sounded more optimistic about the potential of PDE-based image compression than we are currently. We believe the use of PDEs for smoothing is probably best left to image processing.

A Implementation

The code for our biharmonic smoother follows. One small variation from Gauss-Seidel is that odd iterations start in the bottom-left corner, even iterations in the upper-right.

Listing 1: The SOR biharmonic smoother.

```
def filter_biharmonic(dimen, iter, u, const):
        (width, height) = dimen

        omega = 1.5

        # precompute index lists
        i_vals = range(width)
        i_rev = range(width)
        i_rev.reverse()

        j_vals = range(height)
        j_rev = range(height)
        j_rev.reverse()

        # map from (position, stencil index) => position
        # positions are in natural
        map = numpy.zeros((width * height, 12))
        # stencil (natural ordering)
        coeff = [ 1, 2, -8, 2, 1, -8, -8, 1, 2, -8, 2, 1 ]

        # construct just the map
        k = 0
        for j in j_vals:
                for i in i_vals:
                        left = i
                        lleft = i > 1
                        right = i < width - 1
                        rright = i < width - 2
                        below = j
                        bbelow = j > 1
                        above = j < height - 1
                        aabove = j < height - 2

                        if left:
                                if below:
                                        map[k,1] = k - width - 1
                                else:
                                        map[k,1] = k - 1
                                if lleft:
                                        map[k,4] = k - 2
                                else:
                                        map[k,4] = k - 1
                                map[k,5] = k - 1
                                if above:
                                        map[k,8] = k + width - 1
                                else:
                                        map[k,8] = k - 1
                        else:
                                if below:
                                        map[k,1] = k - width
                                else:
                                        map[k,1] = k
                                map[k,4] = k
                                map[k,5] = k
                                if above:
                                        map[k,8] = k + width
                                else:
```

```
                        map[k,8] = k
                if right:
                        if below:
                                map[k,3] = k - width + 1
                        else:
                                map[k,3] = k + 1
                        map[k,6] = k + 1
                        if rright:
                                map[k,7] = k + 2
                        else:
                                map[k,7] = k + 1
                        if above:
                                map[k,10] = k + width + 1
                        else:
                                map[k,10] = k + 1
                else:
                        if below:
                                map[k,3] = k - width
                        else:
                                map[k,3] = k
                        map[k,6] = k
                        map[k,7] = k
                        if above:
                                map[k,10] = k + width
                        else:
                                map[k,10] = k
                if below:
                        if bbelow:
                                map[k,0] = k - 2 * width
                        else:
                                map[k,0] = k - width
                        map[k,2] = k - width
                else:
                        map[k,0] = k
                        map[k,2] = k
                if above:
                        map[k,9] = k + width
                        if aabove:
                                map[k,11] = k + 2 * width
                        else:
                                map[k,11] = k + width
                else:
                        map[k,9] = k
                        map[k,11] = k

                k += 1

# do the smoothing
iter_num = 0
while iter_num < iter:
        print '    iter %d/%d' % (iter_num+1,iter)

        # bottom-left => top-right
        k = 0
        for j in j_vals:
                for i in i_vals:
                        if not const[k]:
                                sum = 0
                                for n in range(12):
                                        sum += coeff[n] * u[
                                            map[k,n]]

                                u[k] += omega * (sum * -.05 - u[k])
                        k += 1
```

```
iter_num += 1
if iter_num >= iter:
        break
print '    iter %d/%d' % (iter_num+1,iter)

# top-right => bottom-left
for j in j_rev:
        for i in i_rev:
                k -= 1
                if not const[k]:
                        sum = 0
                        for n in range(12):
                                sum += coeff[n] * u[
                                    map[k,n]]

                        u[k] += omega * (sum * -.05 - u[k])
iter_num += 1
```

References

[1] Distasi, R., Nappi, M., Vitulano, S., *Image compression by B-tree triangular coding.* IEEE Transactions on Communications 45.9, 1997.

[2] Galić, I., Weickert, J., Welk, M., Bruhn, A., Belyaev, A., Seidel, H., *Towards PDE-based image compression.* Lecture Notes in Computer Science, vol. 3752, pp. 37–48, 2005.

[3] LeVeque, Randall J., *Finite difference methods for ordinary and partial differential equations*, SIAM, Philadelphia, 2007.