Color2gray: Implementation Notes

1 Equations

Let $\mathcal{K}$ be a set of unordered pairs of indices, and let $\mathcal{L}$ be a set of ordered pairs of indices, such that each $\{i, j\} \in \mathcal{K}$ has exactly one matching element in $\mathcal{L}$, either (i, j) or (j, i).

We consider the minimization problem:

$$\min_x \ f(x) = \frac{1}{2} \sum_{(i,j)\in\mathcal{L}} (x_i - x_j - \delta_{ij})^2. \tag{1}$$

1.1 Linear Algebra

This is a linear least squares problem, as it has the form

$$\min_x \ \frac{1}{2}(Ax - b)^T(Ax - b).$$

Which can be rearranged to yield

$$\min_x \ (\frac{1}{2}x^T A^T Ax - (A^T b)^T x + \frac{1}{2}b^T b). \tag{2}$$

Equation (2) is a quadratic with a symmetric positive semi-definite Hessian[1], therefore minimizing it is equivalent to satisfying the linear equation:

$$A^T Ax = A^T b.$$

[1] $A^T A$ is always a symmetric positive semi-definite matrix, regardless of the form of A.

For the N=4 case, with $\mathcal{K}$ containing all pairs of indices, the values of these matrices are:

$$A = \begin{bmatrix} 1 & -1 & 0 & 0 \\ 1 & 0 & -1 & 0 \\ 1 & 0 & 0 & -1 \\ 0 & 1 & -1 & 0 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & -1 \end{bmatrix}_{N \times \binom{N}{2}}, \quad b = \begin{bmatrix} \delta_{12} \\ \delta_{13} \\ \delta_{14} \\ \delta_{23} \\ \delta_{24} \\ \delta_{34} \end{bmatrix}_{\binom{N}{2} \times 1},$$

$$A^T = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 1 & 1 & 0 \\ 0 & -1 & 0 & -1 & 0 & 1 \\ 0 & 0 & -1 & 0 & -1 & -1 \end{bmatrix}_{\binom{N}{2} \times N}, \quad A^T b = \begin{bmatrix} \delta_{12} + \delta_{13} + \delta_{14} \\ -\delta_{12} + \delta_{23} + \delta_{24} \\ -\delta_{13} - \delta_{23} + \delta_{34} \\ -\delta_{14} - \delta_{12} - \delta_{34} \end{bmatrix}_{N \times 1}.$$

Given that, we begin to suspect

$$[A^T b]_k = \sum \delta_{kj} - \sum \delta_{ik}.$$

1.2 Calculus

Doing out some math,

$$\begin{aligned} \frac{\partial f}{\partial x_k} &= \sum_{(k,j) \in \mathcal{L}} (x_k - x_j - \delta_{kj}) - \sum_{(i,k) \in \mathcal{L}} (x_i - x_k - \delta_{ik}). \\ \frac{\partial^2 f}{\partial x_k \partial x_k} &= \sum_{\{k,i\} \in \mathcal{K}} 1. \\ \frac{\partial^2 f}{\partial x_k \partial x_l} &= \begin{cases} -1 & \text{if } \{k,l\} \in \mathcal{K} \\ 0 & \text{otherwise} \end{cases}. \end{aligned}$$

Taking the derivatives of (2) reveals that

$$\nabla^2 f = A^T A, \quad \nabla f = A^T A x - A^T b.$$

And thus

$$A^T b = (\nabla^2 f) x - \nabla f.$$

So

$$\begin{aligned} [A^T b]_k &= \Big[(\nabla^2 f) x \Big]_k - \Big[\nabla f \Big]_k \\ &= \Big(\sum_{\{k,l\} \in \mathcal{K}} x_k - \sum_{\{k,l\} \in \mathcal{K}} x_l \Big) - \Big(\sum_{(k,j) \in \mathcal{L}} (x_k - x_j - \delta_{kj}) - \sum_{(i,k) \in \mathcal{L}} (x_i - x_k - \delta_{ik}) \Big) \\ &= \sum_{(k,j) \in \mathcal{L}} \delta_{kj} - \sum_{(i,k) \in \mathcal{L}} \delta_{ik}. \end{aligned} \tag{3}$$

(Just as we suspected.)

1.3 Special Cases

1.3.1 Complete

For the complete case, the Hessian has a very regular form:

$$\nabla^2 f = \begin{bmatrix} (N-1) & \dots & -1 \\ \vdots & \ddots & \vdots \\ -1 & \dots & (N-1) \end{bmatrix}.$$

Defining

$$d_k \quad = \quad [A^T b]_k \quad = \quad \sum \delta_{kj} - \sum \delta_{ik}.$$

We see that $A^T Ax = A^T b$ expands to

$$(N-1)x_k - \sum_{l \neq k} x_l = d_k.$$

For any two indices, this implies

$$\begin{aligned} (N-1)x_k - \big(\sum_{l \neq k,j} x_l\big) - x_j \quad &= \quad d_k. \\ (N-1)x_j - \big(\sum_{l \neq k,j} x_l\big) - x_k \quad &= \quad d_j. \end{aligned}$$

Thus

$$\begin{aligned} d_k - d_j \quad = \quad \Big((N-1)x_k - x_j\Big) - \Big((N-1)x_j - x_k\Big), \\ d_k - d_j \quad = \quad Nx_k - Nx_j, \\ x_k \quad = \quad \frac{d_k - d_j + Nx_j}{N}. \end{aligned} \tag{4}$$

This is a useful equation, because we know that (1) has infinitely many solutions. In fact, for any $x_k = c$, there is exactly one solution to (1), which may be obtained by taking any known minimal vector $\mathbf{x}'$, and shifting all of its elements by $x_k - x'_k$. Thus, for the special case where $\mathcal{K}$ is complete, we can solve the system in $O(N)$ by setting $x_0 = 0$, solving for all other x_k using (4), and then shifting the elements as described in the paper [1] (see also Appendix A).

1.3.2 Fast Approximate Solve for the Complete Case

For the complete case, we can borrow an idea from [2], and use a quantized version of the image to accelerate the calculation of d_k, dramatically speeding up the solve at the cost of a little accuracy, and turning the algorithmic bound of the entire process into whatever the cost of our quantization algorithm is.

More exactly, if c is one of the quantized colors, $\mathcal{C}$ is the set of all indices of the source image that are assigned to that color, and s_i is the color of the i^{th} pixel of the image, then:

$$\sum_{i \in \mathcal{C}} \delta(s_k, s_i) \approx |\mathcal{C}| \delta(s_k, c).$$

I don't know enough about either statistics or quantization procedures to be able to say anything specific about the quality of that approximation, but it seems like it could be quite good.

If Q is the set of indices of the quantized colors, then we can use the above to approximate d.

$$d_k = \sum_i \delta(s_k, s_i),$$

and so

$$d_k \approx \sum_{p \in \mathcal{Q}} |\mathcal{C}_p| \delta(s_k, c_p).$$

Because finding d was the dominant cost of solving the complete case, and it now appears that a good approximation to d can be found in $O(N|\mathcal{Q}|)$, the bound on the cost of finding this approximate solution will reduce to the cost of the quantization algorithm, which is presumably $O(N)$ or greater.

A similar acceleration could be arranged for the incomplete case, but, because that solve is actually at least as expensive as the d_k calculation, it probably wouldn't gain you anything. More attractive for the incomplete case is a multiresolution solver, which could be implemented independently of any quantization.

2 Code

Calculating the δ_{ij}'s tends to be one of the largest computational costs. Luckily, because of (3), there is no need to keep all the δ_{ij}'s in memory, we can just build up the **d** vector incrementally. (Which is a very good thing, because when N is the number of pixels in an image, $\binom{N}{2}$ is uncomfortably large.)

2.1 Complete $\mathcal{K}$

```
for( i=0; i<N; i++) d[i] = 0;

for( i=0; i<N; i++) for(j=i+1;j<N; j++) {

    float delta = calc_delta(i,j);
    d[i]+=delta;
    d[j]-=delta;
}
```

Calculate the **d** vector for a given image, using a complete $\mathcal{K}$

```
//solve by substitution.
//assume something sensible is in data[0]
// (0 or the source luminance both work fine).
for(i=1;i<N;i++) {
    data[i] = d[i]-d[i-1]+N*data[i-1];
    data[i] /= (float)N;
}
```

Solve the complete case, given **d**.

A faster approximate solve using quantized data looks about the same as the above; the only difference is that the **d** vector calculation iterates through quantized bins, instead of through all pixels.

2.2 Sparse $\mathcal{K}$

In [1], we used conjugate gradient iterations to perform a general solve, but the cost of solving the sparse case tends to be small, and so we can get away with an unsophisticated solver. Restating some choice equations from Section 1:

$$\begin{aligned} A^T A x &= A^T b. \\ \sum_{\{k,l\}\in\mathcal{K}} x_k - \sum_{\{k,l\}\in\mathcal{K}} x_l &= d_k. \end{aligned}$$

Thus we can define an iteration:

$$x_k^{n+1} = \frac{\sum_{\{k,l\}\in\mathcal{K}} x_l^n + d_k}{\sum_{\{k,l\}\in\mathcal{K}} 1}.$$

(Which can be trivially rearranged to yield your favorite variant of a Gauss-Seidel or Jacobi iteration; alternatively, you can work a little harder, and do something similar to end up with a preconditioned conjugate gradient iteration.)

```
for( i=0; i<N;i++) d[i] = 0;

for(x=0;x<w;x++) for(y=0;y<h;y++) {
    float sum=0;
    int count=0;

    i=x+y*w;

    for(xx=x-r;xx<=x+r;xx++) {
        if(xx<0 || xx>=w) continue;
        for(yy=y-r;yy<=y+r;yy++)          {
            if (yy>=h || yy<0) continue;
```

```
            int j=xx+yy*w;
            float delta = calc_delta(i,j);
            d[i]+=delta;
            d[j]-=delta;
        }
    }

}
```

Calculate **d** for the sparse case.

```
for(k=0;k<iters;k++) {

    //perform a Gauss-Seidel relaxation.
    for(x=0;x<w;x++) for(y=0;y<h;y++) {
        float sum=0;
        int count=0;

        for(xx=x-r;xx<=x+r;xx++) {
            if(xx<0 || xx>=w) continue;
            for(yy=y-r;yy<=y+r;yy++)          {
                if (yy>=h || yy<0) continue;
                sum+=data[xx+yy*w];
                count++;
            }
        }

        data[x+y*w]=(d[x+w*y] +  sum) / (float)count;
    }
}
```

Solve the sparse case, given **d**.

References

[1] Amy A. Gooch, Sven C. Olsen, Jack Tumblin, and Bruce Gooch. Color2gray: Salience-preserving color removal. *To Appear in SIGGRAPH 2005.*

[2] K. Rasche, R. Geist, and J. Westall. Re-coloring images for gamuts of lower dimension. *To appear in Eurographics 2005.*

A Post Solve

We shift the image found by the solve to be as close to the source luminances as possible. Formally, if **x** are the gray values returned by the solver, and **y** are

the source luminances, we want

$$\min_s f(s) = ||\mathbf{x} - \mathbf{y} - \mathbf{s}||$$

where s is some scalar, and $\mathbf{s} = \begin{bmatrix} s \\ s \\ \vdots \end{bmatrix}$. This minimization is equivalent to

$$\min_s f(s) = \sum (x_i - y_i - s)^2$$

which is, once again, a linear least squares problem. This time,

$$A = \begin{bmatrix} -1 \\ -1 \\ \vdots \end{bmatrix}, \quad b = \begin{bmatrix} y_1 - x_1 \\ y_2 - x_2 \\ \vdots \end{bmatrix}, \quad A^T A = N, \quad A^T b = \sum (x_i - y_i).$$

So the solution is simply to shift by the average difference,

$$s = \frac{\sum (x_i - y_i)}{N}.$$

or, in code:

```
float error=0;
for(i=0;i<N;i++)
    error+=data[i]-(source.data)[i].l;
error/=N;
for( i=0;i<N;i++) data[i]=data[i]-error;
```