A Brief Introduction to ICM

Jerod Weinman

Background

Images get corrupted by many things, from noisy transmissions (e.g., from Mars rover to NASA) to dirty scanner glass and camera lenses. In such situations, we want to restore the image to its original or proper state as best we can. One method for doing this is called "Iterated Conditional Modes." We won't go in to all of the rich theory behind it, but the name does tell us two things. It's iterated, so it will probably involve a loop. And by "mode" we mean finding a maximum (or actually a minimum, in our case).[1] In short, we'll be iteratively optimizing something within a loop, and that something is our best guess of what a corrupted image pixel should be.

In this lab, we are going to be concerned with binary images. That is, images whose pixels are either on or off, black or white. Thus, the only noise that is possible is whether a black pixel (or bit) was flipped to a white one, and vice versa.

When these images are *huge* (very high resolution telescope scans of the sky, or Google-scale satellite images of the earth), distributed parallel processing can make the job much faster.

Image Processing Background

(If you took 161 within the last two years, you are probably already familiar with this section.)

An image is thought of as a two dimensional array, where each pixel is indexed by its row and column. In C code, this is often something like

```
image[i][j]
```

which refers to the element of the image at row `i` and column `j`, which we'd call pixel (i, j). To look at the left neighbor of the pixel, we'd index the column, offsetting by one:

```
image[i][j-1]
```

Now, image folks like to view the world from the top-down. Therefore, rows of the image start at the top, and work their way down. This is in contrast to the usual cartesian coordinate system you're using to graphing where "0" on the y axis is at the bottom-left. Making this concrete with some code,

```
image[0][5]     // the TOP row, and the 5th column
image[M][3]     // if there are M rows, this pixel is in the bottom row
image[i-1][j]   // this is the upper neighbor of (i,j)
image[i+1][j+1] // this is the lower-right neighbor of (i,j)
```

Hopefully that's enough to get you started with the data structures we'll be using.

Often times, it is more convenient to abstract from the two dimensional indexing and think of a pixel as a single index $p = (i, j)$. If $\mathbf{x}$ represents the entire image, then we can go back and forth between the mathematical construct/notation, and the program code notation:

$$\texttt{image[i][j]} \leftrightarrow x_p.$$

As it turns out, this will make it much easier to describe the ICM algorithm using mathematical notation. Part of your job is to translate this to program code.

[1] The use of "mode" is a bit of a confusing appropriation of a term from statistics (meaning the most frequent observation) that can be interpreted as finding a maximum. It would be better if the M in ICM stood for maximization (or minimization).

ICM Motivation

This image restoration algorithm makes two basic assumptions, one based on the contents of images in general, and another based on the noise process, having nothing to do with image properties. The first assumption about images says:

Assumption 1 Neighboring pixels tend to have the same values.

Think about it. Images are made up of regions corresponding to objects that tend to have roughly the same color and brightness. In a binary image, there will be a lot of black regions and a lot of white regions. Only on the borders between these regions is the assumption violated, but we'll just have to hope that isn't too big a deal. The power behind this assumption is that if we see a white pixel in a whole sea of black pixels, we might reasonably conclude that it was corrupted by noise, accidentally flipped, and that it too should probably be a black pixel. (The reverse situation would also hold.)

The second assumption about the noise process says:

Assumption 2 Each pixel is corrupted (flipped) independently, and with some probability.

This assumption says (rather straightforwardly) that the corruption, or lack thereof, of any two pixels is not dependent in any way. Thus, if one pixel is flipped by a noisy transmission line, it has no bearing on whether any of its neighbors are flipped. This covers the first part of the assumption. The second part simply says that there is a likelihood of any pixel being flipped. A probability of 100% flipping is not exactly noise, since every pixel would be flipped and it would simply result in an inverse image. A probability of 10% that a pixel would be flipped is a moderately noisy image, and a probability of 50% would be an utter disaster, because every pixel would be essentially random. So long as there is a moderate amount of noise, we can create a process using these assumptions that does a pretty good job of fixing up the image.

ICM Implementation

Once again, there is some very fascinating theory behind the image models that are derived from the above assumptions. Since this is an OS class, we're going to focus only on the basic algorithm, in order to reiterate that topics like parallelization are important in a wide variety of computational domains.

One important aspect of the ICM algorithm is the "pixel neighborhood." Recall that assumption 1 states that neighboring pixels have the same value. Well, a pixel must then ask, "who is my neighbor?" The typical answer to this question is the 8 pixels surrounding $p = (i, j)$. The two dimensional indices of these eight neighbors are shown below.

$(i-1, j-1)$	$(i-1, j)$	$(i-1, j+1)$
$(i, j-1)$	(i, j)	$(i, j+1)$
$(i+1, j-1)$	$(i+1, j)$	$(i+1, j+1)$

Note that pixels on the edge of the image will not have all of these neighbors. For instance, the top row of the image will not have the upper row of 3 neighbors. Corner pixels only have 3 neighbors!

We will refer the set of neighbors for a given pixel $p = (i, j)$ as $N(p)$. This will make it easy to refer to properties of the neighborhood of p. For instance, if $x_p \in \{0, 1\}$ where 0 means a pixel is black, and 1 means a pixel is white the total number of white pixels surrounding p is given by the sum

$$\sum_{q \in N(p)} x_q.$$

Now we're finally ready to get to the algorithm. Our goal is to estimate the original, uncorrupted image $\mathbf{y}$ from the observed noisy image $\mathbf{x}$. To do this, the algorithm iteratively minimizes a *cost* of the reconstruction of each pixel. We'll do this iteratively, so we represent the restored image at iteration k as $\mathbf{y}^{(k)}$. Updates $\mathbf{y}^{(k+1)}$ are repeatedly calculated from the current restored image $\mathbf{y}^{(k)}$.

The two assumptions above are reflected in two components of the cost. First, due to assumption 2 there is a penalty if the reconstruction flips a bit from the observation . This value corresponds to the probability of a bit being flipped

from noise. A *large* penalty for each flipped bit in the restorationcorresponds to a *small* probability of corruption. A second penalty term, due to assumption 1, is paid for every pair of neighboring pixels that are different. Once again, this promotes smoothness of the image. The larger the penalty, the smoother we are asking the reconstruction to be. These two penalty terms, which we will call α and β, interact to determine how the image is reconstructed.

Now, on to the heart of it. Given the current reconstruction $\mathbf{y}^{(k)}$, we can define the cost of giving a particular pixel p either value $y_p \in \{0, 1\}$

$$C\left(y_p\right) = \alpha\left(1 - \delta\left(y_p, x_p\right)\right) + \beta \sum_{q \in N(p)} \left(1 - \delta\left(y_p, y_q^{(k)}\right)\right).$$

The function $\delta\left(a, b\right)$ is known as the "Kronecker delta function" has the simple definition

$$\delta\left(a, b\right) = \begin{cases} 1 & \text{if } a = b \\ 0 & \text{if } a \neq b. \end{cases}$$

The meaning of the cost $C\left(y_p\right)$ is given intuitively above. The first term corresponds to assumption 2 and says "add α if the reconstructed pixel and the observed (original) pixel don't match." The second term corresponds to assumption 1 and says "add β for every neighbor in the current reconstruction that doesn't match y_p." We then assign an updated reconstruction by choosing each pixel value as the one that minimizes the cost

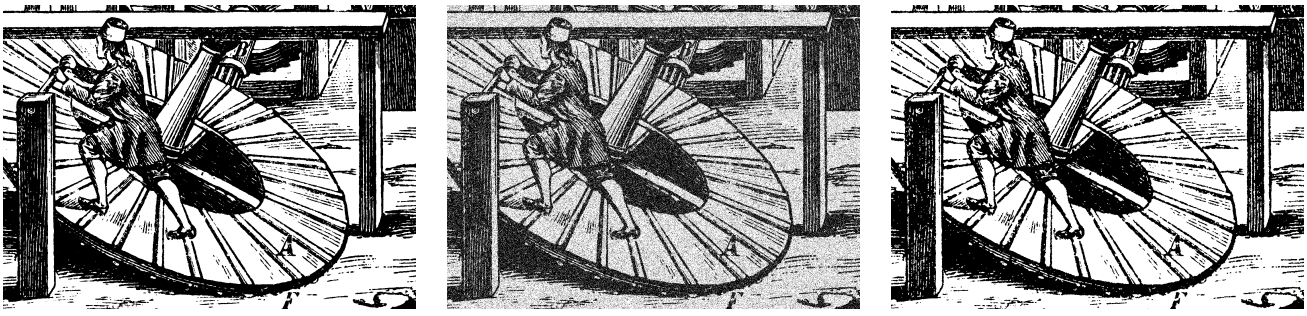
$$y_p^{(k+1)} \leftarrow \arg \min_{y_p \in \{0,1\}} C\left(y_p\right).$$

Note that all of the reconstructed pixels are updated synchronously. In other words, we must use values from the previous reconstruction $\mathbf{y}^{(k)}$ for calculating every updated reconstructed pixel in $\mathbf{y}^{(k+1)}$. The algorithm converges when the updates cease to change the reconstruction,

$$y_p^{(k+1)} = y_p^{(k)}, \forall p.$$

Sometimes, though, the algorithm tends to oscillate between a few states, so a iteration limit on k is needed. This can be easily detected, but you won't be asked to do that.

Below is an example original image, a corrupted version of the image, and a few steps of the iterations for restoring it with $\alpha = 2$ and $\beta = 1$.



Original Image | Noisy Image (15%) | Restored Image

Original (Cropped) | Noisy (Cropped) | 1st Iteration | 2nd Iteration | 3rd Iteration