Computer Vision Lab 1

Eric Purdy *

April 8, 2008

1 File Formats

Image files come in two flavors: compressed and uncompressed. Uncompressed images are bitmap formats. They are generally much larger than compressed files, but they say exactly what the image looks like at every pixel. Compressed images save space by throwing out some information, usually in a way that is not easily visible to the naked eye. This introduces artifacts which could potentially mess up our algorithms. In this class, we only want to deal with uncompressed images.

We will deal with files in three formats, PGM, PBM, and PPM. These formats are all related, and they are collectively referred to as PNM files.

- PGM: these generally have the suffix .pgm. These are greyscale images. Every pixel has an integer value between 0 and 255, where 0 represents black and 255 represents white.
- PBM: these generally have the suffix .pbm. These are binary images. Every pixel is either 0 or 1. Generally, when displayed, 1 gets rendered as black and 0 as white, which is kind of backwards. (But we are thinking of the 1's as "events" or "hits", so we want black lines on a mostly white background.)
- PPM: these generally have the suffix .ppm. These are color images. Every pixel has three integer values, each between 0 and 255. These three values represent the intensity in the red, green, and blue channels.

2 Image Processing Libraries

Images are specified in a sort of upside-down Cartesian plane, where `(0,0)` is the upper left corner of the image, the x coordinate grows to the right, and the y coordinate grows down. This is also the convention used for computer graphics. All of the formats we are using will obey this rule.

*Email: epurdy@cs.uchicago.edu

2.1 C++: VLIB

This library is written by Pedro. You can download it here:

`http://www.classes.cs.uchicago.edu/archive/2006/fall/35040-1/vision-lib.zip`

You can add it into your code like so:

```
#include "VLIB/pnmfile.h"
#include "VLIB/image.h"
```

There are quite a few other files, but these are the ones you need to get up and running.

2.1.1 Data Representation

In VLIB, images are represented as objects of the `image` class. (Note that the class name is lower-case!) The `image` class is mostly just an array of pixel data, which is stored in row-major order, like we discussed in class. (It also has the width and the height of the image and some other stuff.) This class is templated, so we can have images with the intensities represented by different C++ types. The most important types are:

- `uchar`: short for `unsigned char`, this is the standard type for images that are represented in PGM format. You will use it whenever you load or save a greyscale image. `uchar`'s can take values from 0 to 255, so 0 represents pure black, and 255 represents pure white. This is also generally the type you use for PBM's.
- `float` or `double`: sometimes you want to have a little more precision in your data. Always rounding your data to the nearest integer can introduce a lot of rounding error, and it also adds high-frequency noise, which messes up smoothing operators. Whenever you smooth an image, the output should be an `image<float>` or `image<double>`.
- `rgb`: For representing color images. A color image is basically just three greyscale images, one for each of the red, green, and blue channels. The `rgb` struct is defined in `misc.h`, and it just consists of three `uchar` values, which you access as `p.r, p.g, p.b`.

To make an image or delete it:

```
image<uchar> *im = new image<uchar>(width,height,init);
delete im;
```

Make `init` 1 if you want this image to be initialized to 0, and 0 if you don't want it initialized.

To load or save an image:

```
image<uchar> *im = loadPGM(filename);
savePGM(im,filename);
```

where filename is a C-style string (i.e., a null-terminated array of `char`'s).

To touch a pixel:

```
imRef(im,x,y)
```

`imRef` is a macro, so it has the nice property that we can use it on either side of an assignment. For example:

```
imRef(im,x,y)=imRef(im2,x,y);
```

copies the value at `(x,y)` in `im2` into the same pixel of `im`.

Images contain their width and height, which can be accessed as follows:

```
int w=im->width();
int h=im->height();
```

To copy an image:

```
image<uchar> *out = in->copy();
```

One important thing to note: you should always be passing around pointers to images, never images. This is especially important for function calls. The image class doesn't have a copy constructor, which it would need to pass an image by value. So, if you call a function on an image instead of an image pointer, the result will be kind of weird. (Probably you will wind up with two image objects which share their data pointer. This is almost certainly not what you want.)

2.1.2 Useful Functions

There are a bunch of image conversion functions in `imconv.h`. Most of them are self-explanatory. One of the more useful ones is:

```
image<uchar> *imageFLOATtoUCHAR(image<float> *im, float low, float high);
```

This function is in `imconv.h`, which you will need to include separately. This function converts a `float` image to a `uchar` image. You can use it to save intermediate `image<float>`'s, which makes it very useful for debugging. If you leave out low and high, this function makes black the lowest value in your image and white the highest, so that you represent all values with the maximum amount of contrast. Sometimes this is not what you want, so you can pass it `low` and `high` for the range you want to map to $[0, 255]$.

2.2 MATLAB: Image Processing Toolbox

This is already installed in the CS Department machines, at least the Linux ones.

2.2.1 Data Representation

Like everything else in MATLAB, images are represented as matrices. They are stored in row-major order, like we discussed in class.

To load or save an image:

```
I=imread('image.pgm');
imwrite(I,'image.pgm');
```

To touch a pixel:

```
I(y,x)
```

Note that it is backwards! (It's correct for matrix subscripting.) Also, MATLAB stores matrices in column-major order, so the coordinates have to be backwards so that we can think of it as an image stored in row-major order.

```
J(y,x)=I(y,x);
```

copies the value at `(x,y)` in `I` into the same pixel of `J`.

```
J(:,:)=I(:,:);
```

copies an entire image.

To get the width and height of an image:

```
[height,width]=size(I);
```

You can view an image from inside of MATLAB by typing:

```
imshow(I)
```

This function uses a dynamic range, so it displays the lowest number in your image as black, the highest as white, and everything in between as some linearly determined shade of gray. Sometimes this can be annoying. Use

```
imshow(I,[low,high])
```

to specify a range explicitly.

2.3 OCaml

You can download a copy of Trevor Smith's OCaml vision library here:

```
http://people.cs.uchicago.edu/~epurdy/vision-lib.tar.gz
```

If you're using OCaml, I'm going to assume that you know what you're doing, so I won't go through all the details. However, the documentation is very well done. You can generate if for yourself by running:

```
make all && make doc
```

There's also a copy of the documentation online here:

```
http://people.cs.uchicago.edu/~epurdy/vision-lib/doc/
```

3 Making a Negative Image

We want to write the following function, which makes a negative image, where the value at each pixel is 255 minus the old value.

```
image<uchar> *invert(image<uchar> *in);
```

This is basically the easiest thing ever, so it's a good place to start to make sure that we know how to load images, play with them, and then save them.

3.1 How Important IS Cache Coherence?

This section might not make sense if you're using MATLAB.

In most vision algorithms, we're going to loop over the image in a double for loop, like this:

```
for(int y=0;y<height;y++){
  for(int x=0;x<width;x++){
    ...
  }
}
```

Just this once, we're going to do it the wrong way, looping over `x` first:

```
for(int x=0;x<width;x++){
  for(int y=0;y<height;y++){
    ...
  }
}
```

Write the function

```
image<uchar> *BADBADBADinvert(image<uchar> *in);
```

which does it this way. (You could have written the invert function by iterating over the array without using `x` and `y`. Let's assume/pretend that you didn't.) How much longer does it take to make the negative image? (You can use the unix `time` command to answer this question if you're running from the command line.) You should probably have the program do it 100 times so that it's easier to tell how much difference it makes.

4 Resizing Images

Now we want to take an image and change its size.

When we resize an image, we iterate over the destination image, not the source image. This makes sure that we cover every pixel in the destination image. For example, if we double the image in both directions, we will have four times as many pixels. If we mapped each pixel in the source image to a pixel in the destination image, we would have lots of holes. Instead, for every pixel in the destination image, we look back at the source image and try to figure out what the color at that pixel should be.

Sometimes the answer is obvious: if we are doubling the size of an image, then the value at pixel (100,200) of the destination image should be the value at pixel (50,100) of the source image. However, most of the time, when we go backwards, we will end up wanting to know what the value of the source image was at some point between pixels. There are several different ways to calculate a reasonable value for this. The two ways we will examine are to pick the nearest pixel, and bilinear interpolation.

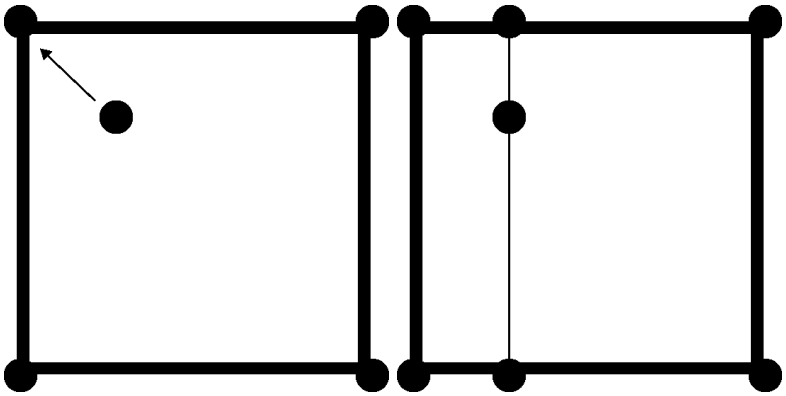


Figure 1: Left: nearest neighbor. Right: bilinear interpolation

4.1 Nearest Neighbor

Write the function

```
image <uchar> *resizeNN(image<uchar> *im, double factor);
```

by iterating over the pixels of the return value image, mapping each pixel back to an point in the source image, and picking the pixel nearest that point.

4.2 Bilinear Interpolation

If instead of an image, we had a one-dimensional array A, one natural interpretation of A[3.6] would be a weighted average of the values at A[3] and A[4]:

```
B[3.6] := 0.4 * A[3] + 0.6 * A[4]
```

More generally, we would have

```
B[x] := (ceil(x)-x) * A[floor(x)] + (x-floor(x)) * A[ceil(x)];
```

This gives us a value in between the two, and it reflects the fact that we are closer to A[4] than we are to A[3]. If we were to look at the array as a function, we would have a bunch of line segments connecting the values of A at the integers.

For an image, we would like to do something similar. And, in fact, we can - we can resize in one coordinate, just treating every row as an independent array, using the above strategy. Then we resize in the other coordinate, treating every column of the image as an independent array.

```
B[x,floor(y)] := (ceil(x)-x) * A[floor(x),floor(y)]
               + (x-floor(x)) * A[ceil(x),floor(y)];
B[x,ceil(y)] := (ceil(x)-x) * A[floor(x),ceil(y)]
              + (x-floor(x)) * A[ceil(x),ceil(y)];
C[x,y] := (ceil(y)-y) * B[x,floor(y)] + (y-floor(y)) * B[x,ceil(y)];
```

You might worry that the result depends on whether we do this in the horizontally or vertically first, but it turns out that it doesn't matter - we get the same result. (Modulo floating-point nonsense.)

Use this to write the function

```
image <uchar> *resizeBI(image<uchar> *im, double factor);
```

4.3 Writing a `diff` Function

Write the function

```
image<uchar> *diff(image<uchar> *im1, image<uchar> *im2);
```

which computes the difference between two images at every pixel and stores the results in a new image. How much do your two resize functions differ? What does their difference look like?

In MATLAB this function is trivial - it's just the difference of the two matrices:

```
D=Im1-Im2;
```

The diff function is kind of cool - if you apply it to two succesive frames of a video segment, you will very easily be able to see objects which are moving.

4.4 Other Approaches?

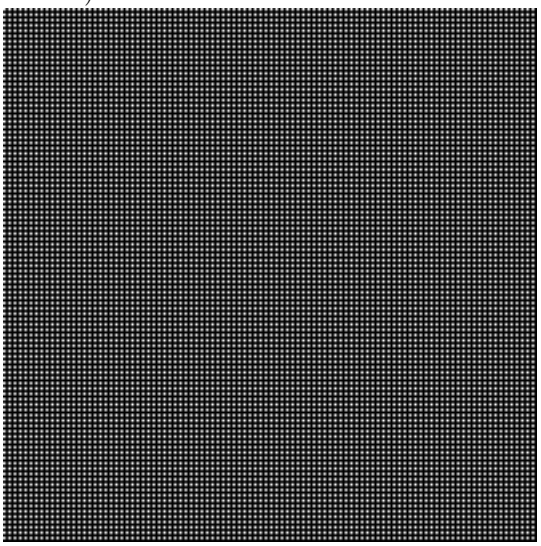
There are some other ways to calculate a value between pixels. For instance, any three points in an image define a unique linear interpolation. So, we could find a triangle of three pixels that our point lies in, calculate the unique linear function that goes through all three of these pixels with the correct values, and then use the value of this function at our point.

4.5 The Problem with Subsampling

Download the following image from

`http://people.cs.uchicago.edu/~epurdy/problem.pgm`
`http://people.cs.uchicago.edu/~epurdy/problem-matlab.pgm`

(Matlab users may need to get the second image, depending on how you wrote your code.)



and resize it by half, using either of your resize functions. What happens? Why? (You might need to use a good image-viewer to see what's going on in the image.)

How would you fix that?

5 Rotating Images

Recall that we can rotate a point (x,y) by an angle θ as follows:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Use this to write a function to rotate an image. Again, iterate over the destination image, and figure out which point each pixel came from. Then use nearest neighbor or bilinear interpolation to get a value for that point. This is basically just using the above matrix, except that we need to shift the whole image over and up by some amount so that none of the coordinates are negative. You can calculate this amount by just figuring out what happens to the four corners of the source image; you want one of these to touch each side of the destination image. (When you rotate an image, you're going to wind up with blank space. You can make these areas black or white, according to preference.) Also, θ will probably be the opposite of what you expect, since image coordinates have the opposite orientation to Cartesian coordinates.