CS 465 Program 2: Resample

(revised September 23, 2004)

out: 17 September 2004
due: 30 September 2004

1 Introduction

In this assignment you will implement two different kinds of filtering operations: discrete filtering on an image (to do blurring and sharpening) and image resampling for enlarging and reducing images. We provide a framework that handles loading images, displaying them on the screen, and choosing an operation and filter type. You provide implementations of the filtering algorithms, and the filters themselves.

2 Assignment Overview

The user interface provided by the framework allows the user to load a single image, and it has two modes: one for filtering and resampling the image to particular dimensions (and displaying the result) and one for auto-resampling the image (enlarging it or reducing it to fit the window as it is resized). In each case, the framework calls the filtering code every time the image needs to be recomputed, giving it the single stored image and displaying the result. There is an "apply" button that copies the result image to the stored image, thereby making the current operation permanent.

Your work breaks down into two main parts: image filtering and image resampling. They are completely independent (since the code from one never calls the code from the other). Each of those parts has two subparts: implement the filtering operation for a general filter, and write code to evaluate the individual filters.

2.1 Filtering

All filters have a radius, and can be defined in terms of the canonical form of the given filter. The filters you're required to implement for discrete image filtering are:

1. A constant (box) filter — This should perform an averaging operation over some area.
2. A separable tent filter — This should perform a linear weighting of values. Its definition is given in many of the course resources.

3. A gaussian filter — The gaussian has infinite support, so the radius parameter should scale the filter in the same way as always, but rather than defining the distance over which the filter drops to zero it should define the standard deviation of the gaussian. The gaussian in the notes:

$$f_g(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}$$

has a standard deviation, or "radius" in terms of this project, of 1.0.

Since you can't compute a convolution with an infinite filter, you have to cut it off somewhere to use it in practice. Therefore the gaussian filter has a second parameter, the "cutoff," which determines the size of the actual discrete filter that gets used in the computation. The cutoff is measured in standard deviations, so if the cutoff is 3.0 that means we want to cut off the filter beyond 3 standard deviations, or 3 times the "radius." For instance, if the radius is set to 2.5 and the cutoff is set to 4, then the resulting discrete filter is 21 by 21 pixels (points beyond 2.5 $\times$ 4 = 10 pixels away from the center are excluded). We'll stick with a square support region because then the filter is separable.

Cutting off the support of the filter, of course, causes the filter not to integrate to 1, but this can be ignored for two reasons. The first is that the filter integrates to something pretty close to one as the cutoff increases. The second reason is that you will be doing a discrete form of renormalization akin to Homework Problem 5, part 3 to handle edge cases, and that will also cause the filter to be renormalized everywhere. This will be explained later in this document.

4. Sharpening using an impulse-minus-gaussian filter — If you have all of the other filters implemented, this can be implemented fairly quickly. The "unsharp mask", as it is called, actually sharpens the image and is the result of subtracting a blurred version of the image from the original. In symbols:

$$S' = S \star ((1+\alpha)I - \alpha G)$$

Where, S' is the new set of samples, S is the original sample set, I is the identity filter, G is the gaussian filter, and $\star$ is the convolution operator. The value α is a free parameter for the filter, just as the radius of the gaussian is.

All of these filters can be expressed as separable filters built from 1D filters. You should implement these filters both ways: as 2D filters (ignoring the fact that they are separable) and using two 1D filtering operations, taking advantage of separability. For large filter radii you should see a dramatic difference in performance. If you engineer it right, this should just require two versions of a general filtering routine; the code for the individual filters should be able to be shared.

2.2 Resampling

The resampling filters (resamplers for short) have to deal with inputs and outputs of different sizes. That is, they resample the image at different frequencies. In the upsampling (enlarging) case, the radius of each filter, measured in terms of the source image's pixels, should be the radius in its canonical form. In the downsampling (shrinking) case, the radius should be the radius of the canonical form, scaled by the new sample spacing. This is the same as saying that the size of the filter is determined by either the source image's sample spacing or the output image's sample spacing, whichever is larger.

The filters for resampling are:

1. The box filter — In the upsampling case, this filter reduces to nearest-neighbor sampling, taking each output sample directly from the nearest source sample. On an upsampling case, this becomes point sampling as it will simply grab the nearest neighbor. In the case where an output sample falls equally between two source samples, you should choose the point that has the higher x or y value. Think of this as rounding up when your fraction is 0.5. In the downsampling case, this does the area averaging just like its filter counterpart.

2. The tent filter — Upsampling with this should give you a linear interpolation between the nearest values. Down sampling will perform much the same as the tent filter for blurring images.

3. The B-spline filter — This is a smooth, positive filter which will result in a nice blur to handle some of the more difficult aliasing problems. Because of its larger size (its canonical radius is 2, compared with 1 for the tent filter) it is also fairly expensive to compute. Its definition can be found in the posted notes on sampling and reconstruction.

4. The Mitchell-Netravali filter — This filter produces a result similar to the B-spline but with a bit less blurring. It is actually a weighted sum of two other filters: the Catmull-Rom filter and the B-spline filter. You can ignore this fact, though, and just use the resulting cubic equations; the code doesn't have to know anything about B-spline or Catmull-Rom filters.

Even though these filters are all separable, you aren't required to take advantage of that fact in your implementation.

When you implement resampling, the most complicated part is keeping the new and old sample grids straight so that you can compute which pixels of the source image need to be looked at to compute a particular pixel of the output. Usually the easiest aproach is to adopt the pixel grid of the source image as your coordinate system. Doing things this way, you compute an output pixel value by first computing the coordinates of that pixel in terms of the source image, then figuring out based on the filter radius which source pixels you need to loop over. Then the code for weighting and adding those pixels follows straight from the equation for discrete–continuous convolution.

2.3 Edge Cases

In practice, there are many ways to handle the edge cases when part of the filter extends beyond the edge of the image. Some of these were talked about in class, such as wrapping around to the other side, reflecting across the border, and using the closest value. For this project we want you to renormalize the convolution operation. That is, as you add up weighted pixel values you should also add up the filter weights, so that at the end you can divide by the total weight. This will be 1.0 for a normalized filter in the middle of the image, but near the edge (or for downsampling at odd ratios, see homework problem 5) the weights won't add up exactly, and the normalization will prevent the image brighness from wavering when it shouldn't. The resulting equation for filtering is:

$$I'[j] = \frac{\sum_i I[i]f[j-i]}{\sum_i f[j-i]}$$

and for resampling it is:

$$I'(x) = \frac{\sum_i I[i]f(x-i)}{\sum_i f(x-i)}$$

I' is the new image, I is the original image, f is the filter, and r is the radius of the filter. The sums logically run over all the pixels in the source image, though of course in practice you'd only compute the part where the filter is nonzero. It's fine if you always renormalize, even for pixels in the middle of the image where you know the filter will already be normalized, just because it lets the code remain simpler. For image filtering, this will also prevent having to worry about normalization for the truncated Gaussian filter.

3 Implementation

The framework contains two abstract classes, `Resampler` and `Filter`, each with several subclasses corresponding to the different filters you need to implement. We've provided empty classes for all the required filters (both discrete image filters and resampling filters). You'll need to implement the `apply` method for each filter.

For the filters, you should take advantage of separability. All of the filters we are having you write can be written in a separable manner. The abstract subclass of `Filter`, `SeparableFilter` should override the `apply` method and write it in such a way that it takes advantage of separability.

Efficiency is definitely a concern in this project. You do not need to use very complicated optimization techniques, but you will not get a perfect score for code that wastes effort. You'll need to do things like caching values to avoid computing them thousands of times from scratch; computing your loop bounds carefully to avoid wasting time on pixels that will always have weight zero; and avoiding memory allocation in your inner loops.

3.1 Hints

- Implement discrete filtering first; it is simpler to think about than resampling.
- You can save yourself a lot of work by implementing `apply` in the base classes `Resampler` and `Filter` and arranging it so that the filter evaluations themselves can be handled by the subclasses. The box filter for resampling is a little tricky (because of the issue of choosing the correct sample when you land exactly halfway between two samples) and it may be easier to implement a separate `apply` for that case rather than trying to make it fit into the general framework.
- The framework provides a `Point` resampling filter, just so you can see an image on the screen (with plenty of aliasing). The code is highly specialized for the point sampling case, though, so you may do better starting from scratch rather than adapting this loop.

4 Submission and FAQ

Submission will be through CMS. A FAQ page will be kept on the course web site detailing any new questions and their answers brought to the attention of the course staff.