Image Processing & Antialiasing

Part III
Outline

• Overview
• Example Applications
• Jaggies & Aliasing
• Sampling & Duals
• Convolution
• Filtering
• **Scaling**
  • Reconstruction
  • Scaling, continued
  • Implementation
Image Scaling

- Back to image scaling
- In particular, problem of resampling
Mapping

Samples, Reconstruction, and Filters

- build new image out of old one
- new image = transformed version of old one
- $N$ new pixels will be derived from $M$ old pixels. How to get new pixels?
- Sample old image. Exactly where we sample is determined by size of old and new image
- Example: scaling up. Old image: 10 x 10 pixels; New image: 15 x 15 pixels
  - need to generate 15 x 15 sample points; have 10 x 10 pixels
  - given 15 x 15 samples across 10 x 10 pixels, must sample “in between” some pixels in old image
Sampling

- Each pixel in new image will map to a sample point in old image
- If sample point lies exactly on pixel in old image, then we can use that pixel value
- But what if our sample point lies somewhere between two old pixels?
- Answer #1: if we had “original” continuous picture represented by our 10 x 10 image, then we could resample it 15 x 15 times instead. However, we only have a discrete version
- Answer #2: take our best guess at what original picture would have looked like. Using old data, try to reconstruct value at any arbitrary sample point on, or in between, our 10 x 10 pixels
Sampling Reconstructed Image (1/3)

- Resampling original image (say of three gray vertical bars) between integer pixel locations: use best guess

Each integer pixel location in new transformed image corresponds to a real-valued sample location in old image (not necessarily on pixel boundaries! In this case, we must sample 15 times in a 10-pixel area, i.e., every 10/15 or 2/3 pixel intervals)

SWAG: “simple wild-ass guess”

No guess needed

Good guesses
Sampling Reconstructed Image (2/3)

- Can think of image sampling as two distinct steps
- Reconstruct “original” continuous picture information from our discrete samples; note: most likely reconstructed image does not actually match original

Then, resample continuous data at mapped locations. Note: new sample points don’t always match with old ones
Sampling Reconstructed Image (3/3)

- Display sampled values at their proper pixel locations in transformed image (we have scaled image)
Discrete Reconstruction

- In implementation, reconstruction/resampling stages are one step
  - We can only do it analytically for special cases
- “Original” function is reconstructed at sample points needed for our new image. Why reconstruct continuous “original” image if we only need to sample at a few locations?

To get this…

Why build this...

When you only need this?
Outline

• Images & Hardware
• Example Applications
• Jaggies & Aliasing
• Sampling & Duals
• Convolution
• Filtering
• Scaling
• **Reconstruction**
• Scaling, continued
• Implementation
• How to perform reconstruction from sampled signal?
• Use same filters as for pre-filtering, i.e., removing high frequencies from continuous signals
• Take as input an arbitrary sample point in old image (sample points can lie between pixels)
  – best guess at what new value should be, using old image’s nearby pixel values.
• Taking weighted average of discrete pixel values, much easier than integrating over continuous differential areas
Filters (2/2)

- In general, the better a filter will estimate the original signal, the more expensive it is.
  - We want to find a good middle ground between cost and quality

- Review of general traits of filters:
  - take weighted sum of pixel values near sampled point
  - pixels further away are less influential
  - area covered by filter—called *support*—is centered around sample point and usually finite in extent
  - normalize area under filter to sum to 1
    - ensures transformed image will have same overall brightness as original image
  - play with different filter shapes using scaling applet
Digression on Scanning (1/2)

- Just looked at resampling signal reconstructed from samples to scale image; leads to notion of reconstruction filter
- Look at scanning a photo using CCD scanner to understand sampling/reconstruction in spatial and frequency domains
- Scanning “continuous” photograph results in unweighted area sampling: each rectangular element responds to total amount of light falling on it, without regard for how the light is distributed across the sensor.

Scanner CCD

Camera CCD

Thanks to Albert Einstein for photo-electric effect
Digression on Scanning (2/2)

Look conceptually at process of scanning continuous image using theoretical model of pre-filtering each scan line’s \( f(x) \), followed by sampling

1) Pre-filtering [bad]:
   - scanner convolves with box function (unweighted area sampling) in spatial domain (CCD’s fault; not continuous, not ideal filter)
   - dual is \( sinc \) multiplication in frequency domain
   - produces intermediary continuous signal \( \hat{f}(x) \) : mostly band-limited to main frequencies near zero
   - scanning loses high-frequency information but corrupts signal by adding aliases

2) Sampling:
   - sample \( \hat{f}(x) \) ; yield discrete pixels \( P_i \) (information we store about image)
   - intensity of \( P_i \) determined by signal from \( i^{th} \) CCD element

   • Sample \( P_i \) is corrupted version of original intensity of that area; cannot represent original exactly, so do best we can
Reconstruction Filters (1/4)

Sampling introduces infinite replicas of a spectrum

- Sampling continuous signal \( \hat{f}(x_i) \) at discrete pixel locations = multiplying continuous function \( \hat{f}(x_i) \) in spatial domain by comb function, \( g(x) \). Comb function (spatial domain) is zero-valued, except for series of impulses with amplitude 1 at sampling location

- Dual of comb \( g(x) \) in spatial domain is comb \( G(u) \) in frequency domain (Fig 14.26b), set of Dirac Delta functions; Delta is zero everywhere except near center, its area is 1. Multiplying \( \hat{f}(x_i) \) with comb in spatial domain is same as convolving \( \hat{f}(x_i) \)'s dual \( \hat{F}(u_i) \) with comb’s dual \( G(u_i) \) in frequency domain

- Convoluting frequency spectrum with comb \( G(u) \) yields \( \hat{F}_s(u) \) (See Figure 14.26d). \( \hat{F}_s(u) \) contains infinite number of replications of band-limited spectrum of \( \hat{F}(u_i) \) at multiples of sampling frequency
Reconstruction Filters (2/4)

Sampling introduces infinitely replicated spectrum of any band-limited signal

- (a) Comb function $G(x)$ (b) its frequency spectrum $G(u)$, also comb
- (c) frequency spectrum of band-limited signal
- (d) Convolution of (b) and (c) in frequency domain –*infinite frequencies* with high amplitude! (A function convolved with Delta is function at Delta’s “location”). Unless spectrum is low-pass pre-filtered (box in frequency domain), tails of overlapping spectra add, causing further corruption
- Samples form step function (vertical edges between adjacent pixels) in spatial domain; need infinite frequencies to describe signal
Motivation of why sampling introduces infinitely replicated spectrum

- Think of comb of Deltas in frequency domain as skinny boxes with narrow base, height to make area = 1
- Take just the "tooth" at the origin, single skinny box
- Then convolution with signal in frequency domain is just box filtering
  - introduces aliases, blurs signal
  - will yield an attenuated, truncated, corrupted version of signal's spectrum at origin
- As box gets skinnier and skinnier, it will weigh fewer frequencies away from the center of the spectrum
- In the limit, it will simply reproduce the spectrum, without aliasing/blurring
- Same process at each Delta's location
  - depending on sampling frequency, the spectra will have overlapping tails, further corrupting
How do we kill all replicas?

- We remove all but original band-limited spectrum by multiplying with low-pass box filter in the frequency domain, which once again corresponds to convolving with \( \text{sinc} \) in the spatial domain.
Reconstruction Filters (4/4)

Use $sinc$ (or cheaper triangle) to remove replicas in the frequency domain

- Next images (figure 14.27) show discrete sampling and removal of unwanted replicas
  - Images on left represent signal in spatial domain
  - Images on right represent frequency domain
- Best: optimal reconstruction filtering with box in frequency domain ($sinc$ in spatial domain)
- Adequate: approximate reconstruction filtering with triangle in spatial domain:
  - leaves higher frequencies
  - will cause some aliasing
- Triangle reconstruction convolution is decent and inexpensive for removing replicas. Note: reconstructed signal and its frequency spectrum less accurate using triangle
- Later: pre-filtering and reconstruction filtering can be condensed to single filtering step
- Note: we do discrete convolution of samples with filter by placing filter at consecutive sample points (at pixels or in between them, if image is scaled)
Reconstruction Filters: \( \text{sinc} \) vs. Triangle

Fig. 14.27 Sampling and reconstruction: Adequate sampling rate. (a) Original signal. (b) Sampled signal. (c) Sampled signal ready to be reconstructed with \( \text{sinc} \). (d) Signal reconstructed with \( \text{sinc} \). (e) Sampled signal ready to be reconstructed with triangle. (f) Signal reconstructed with triangle. (Courtesy of George Wolberg, Columbia University.)
Reconstruction Filters: Inadequate Sampling Rate

Fig. 14.28 Sampling and reconstruction: Inadequate sampling rate. (a) Original signal. (b) Sampled signal. (c) Sampled signal ready to be reconstructed with sinc. (d) Signal reconstructed with sinc. (e) Sampled signal ready to be reconstructed with triangle. (f) Signal reconstructed with triangle. (Courtesy of George Wolberg, Columbia University.)
Box Filter with 2 unit support

\[ B(x) = \begin{cases} 
  1/2 & \text{if } -1 < x < 1 \\
  0 & \text{elsewhere} 
\end{cases} \]

- Reconstructing with \( B \) as filter “draws bars”:

- Box filter has discontinuities at boundaries, discrete convolution must be special cased:
  - at old pixel locations: just use pixel values
  - everywhere else: use box filter (covers two pixels); value is just \textit{unweighted} average of two neighboring pixels
- Not bad for signals with large constant areas
Box Filter (2/2)

- Lousy for steadily varying signals, for instance, \( \sin(x) \)
**Triangle Filter (1/3)**

- This normalized filter, with 2 unit support, approximates $sinc$ filter of right width to remove replicas.

$$T(x) = \begin{cases} 
1 - |x| & -1 < x < 1 \\
0 & \text{elsewhere}
\end{cases}$$

$$A = \frac{1}{2}bh = 1$$

- Acts as linear interpolation filter. Takes average of neighboring pixel values *weighted* by distance from sample point, i.e., “connect the dots.”
Triangle Filter (2/3)

- If we center triangle filter directly over pixel in source image, it returns that pixel’s value:

  \[ \text{Filter value at } x_0 = 1.0 \]
  \[ \text{Pixel value at } x_0 = 0.8 \]
  \[ \text{Filter}_{x_0} \cdot \text{Pixel}_{x_0} = 0.8 \]

- What happens when we sample halfway between pixels in source image?
- Intuition: would expect it to return average of two pixel values. Intuition is correct:

  \[ \text{Filter value at } x_1 = 0.5 \]
  \[ \text{Pixel value at } x_1 = 0.5 \]
  \[ \text{Filter}_{x_0} \cdot \text{Pixel}_{x_0} = 0.4 \]
  \[ \text{Filter}_{x_1} \cdot \text{Pixel}_{x_1} = 0.25 \]
  \[ F_{x_0} \cdot P_{x_0} + F_{x_1} \cdot P_{x_1} = 0.65 \]
Similarly, if filter is 70% towards one pixel and 30% towards another, will return average of two pixels weighted by 70% and 30%; it linearly interpolates

$$Weight = 0.3 \cdot P_0 + 0.7 \cdot P_1$$

See the discrete convolution applet:

Avoiding Continuous Convolution

• Whatever reconstruction filter we use, save work by combining reconstruction filtering with sampling the reconstructed signal at only a discrete number of sample points determined by the image transformation, e.g. scaling.

• Do this by placing the reconstruction filter at sample points only and evaluating discrete convolution there.

• But what are those sample points for scaling?

• Scaling down adds additional complexity, as we’ll soon see.
In forward direction, filtering is many-to-many, i.e. multiple pixels in source contribute to each pixel in destination, and each pixel in source contributes to multiple pixels in destination.

Many-to-many mapping:
- pixel "1" in source contributes to pixels "1," "2," and "3" in destination
- pixel "3" in destination is contributed to by both pixels "1" and "2" in source
Scaling: Backwards Mapping

Where to place filter in source image?
Compute sample points by back-mapping from destination back to source image

- Then mapping is one-to-many instead of many-to-many
  - For each pixel in destination, which pixels in source contribute to it?
  - Determined by placing filter in source image as function of scale factor
- Backwards mapping is inverse of whichever image transformation we choose to perform, e.g., if we scale source up by 2.0, then map each pixel x in destination from pixel x/2 in source image

For each pixel, x, in destination image, sample x/2 from source image

Scaling up by factor r:

We don’t forward map this way: Instead, we back-map this way:

\[ \text{source}(x) \rightarrow \text{dest}(r \cdot x) \]  \[ \text{dest}(x) \rightarrow \text{source}(x/r) \]
Outline

• Images & Hardware
• Example Applications
• Jaggies & Aliasing
• Sampling & Duals
• Convolution
• Filtering
• Scaling
• Reconstruction
• **Scaling, continued**
• Implementation
Roadmap

- Apply mapping to old image to produce new image
  - calculate sample point locations on old image used to calculate pixel values for new image
  - calculate pixel value of each sample point: evaluate convolution of reconstruction filter with old image at those points
  - some subsequent mapping may include manipulating computed intensity value; we will use it as-is for scaling

- To build new image, convolve reconstruction filter with old image once per pixel location in new image

- We represent convolution with an asterisk:
  \[ P: \text{pixels in image} \quad g: \text{filter} \]
  \[ H = P \ast g: \text{convolution of filter and image} \]

- Convolution is integrating two continuous functions. Only apply filter at discrete points in old image (once per pixel in new image), but still call it convolution. Later we’ll show “discrete” convolution computation. Next: make each step more concrete
1D Image Filtering/Scaling
Again

• Once again, consider following scan-line

• As we saw, if we want to scale by any rational number $r$, must sample every $1/r$ pixel intervals in source image

• Having shown qualitatively how various filter functions help us resample, let’s get more quantitative: show how one does convolution in practice, using 1D image scaling as driving example
Resampling (1/2)

- Call continuous reconstructed image intensity function $h(x)$. For triangle filter, it looks as before:

- To get intensity of integer pixel $k$ in 15/10 scaled destination image, $\tilde{h}(\tilde{k})$, sample reconstructed image $h(x)$ at point $x = \frac{k}{1.5}$

- Therefore, intensity function transformed for scaling is:

$$\tilde{h}(\tilde{k}) = h\left(\frac{k}{1.5}\right)$$
Resampling (2/2)

- As before, to build transformed function \( \tilde{h}(k) \) take samples of \( h(x) \) at non-integer locations.

\[
\tilde{h}(k) = \left( \sum_{i} h(\frac{k}{1.5}) \right)
\]

Resampling (2/2)
Resampling: An Alternate Approach (1/3)

• Previous slide shows scaling up by following conceptual process:
  - reconstruct original continuous intensity function from discrete number of samples, e.g. 10 samples
  - resample reconstructed function at higher sampling rate, e.g. 15 samples across original 10
  - stretch our inter-pixel samples back into integer-pixel-spaced range, i.e. map 15 samples onto 15 pixel range in scaled-up output image

• Thus conceptually first we resample reconstructed continuous intensity function at inter-pixel locations, then stretch out our samples to produce integer-spaced pixels in scaled output image

• Alternate conceptual approach: can change when we scale and still get same result: first stretch out reconstructed intensity function, then sample at integer pixel intervals
Resampling: An Alternate Approach (2/3)

- This new method performs scaling in second step rather than third: stretches out \textit{reconstructed function} rather than \textit{sample locations}
  - as before, reconstruct original continuous intensity function from discrete number of samples, e.g. 10 samples
  - scale up reconstructed function by desired scale factor, e.g. 1.5
  - sample (now 1.5 times broader) reconstructed function at integer pixel locations, e.g. 15 samples
Resampling: An Alternate Approach (3/3)

- Here is what alternate conceptual approach looks like (compare to diagrams on slides 30 and 31)

\[
\begin{align*}
\hat{h}(\hat{x}) & \quad \leftarrow \quad h(\frac{x}{1.5}) \\
\tilde{h}(k) & \quad \rightarrow \\
\end{align*}
\]

These first two are completely theoretical
Scaling Down (1/5)

Why scaling down is more complex than scaling up

- Try same approach as scaling up
  - reconstruct original continuous intensity function from discrete number of samples, e.g. 15 samples (different from 10 sample one we just used)
  - scale down reconstructed function by desired scale factor, e.g. 3
  - sample (now 3 times narrower) reconstructed function at integer pixel locations, e.g. 5 samples

- Unexpected and unwanted side effect: by compressing waveform into 1/3 its original interval, spatial frequencies tripled, which extends (somewhat) band-limited spectrum by factor of 3 in frequency domain. Can’t display these higher frequencies without aliasing!

- Back to low pass filtering again. Multiply by box in frequency domain to limit to original frequency band, e.g. when scaling down by 3, low-pass filter to limit frequency band to 1/3 its new width
Scaling Down (2/5)

Simple sine wave example
• First we start with sine wave:

1/3 Compression of sine wave and expansion of frequency band:

• Get rid of new high frequencies (only one here) with low-pass box filter in frequency domain:

We should be convolving the sine wave with a sinc filter, but, as usual, we’ll approximate the sinc with a triangle filter.

• Only low frequencies will remain
Scaling Down (3/5)
Same problem for a complex signal

Note signal addition (frequency aliasing)!

Band-limited signal

Sampled band-limited signal

Scaled-down band-limited signal

Sampled scaled-down signal
Scaling Down (4/5)

- Revised (conceptual) pipeline for scaling down image:
  - reconstruction filter: Low-pass **filter** to reconstruct continuous intensity function from old scanned (box-filtered and sampled) image, also get rid of replicated spectra due to sampling
  - **scale down** reconstructed function
  - scale-down filter: low-pass **filter** to get rid of newly introduced high frequencies due to scaling down
  - **sample** scaled reconstructed function at pixel intervals

- Now we’re filtering explicitly twice (after box filtering done implicitly by scanner):
  - first to reconstruct signal (filter $g_1$)
  - then to get rid of high frequencies in scaled-down version (filter $g_2$)

- In actual implementation, can combine reconstruction and frequency band-limiting into one filtering step. Why?

- Associativity of convolution:
  \[ h = (f * g_1) * g_2 = f * (g_1 * g_2) \]

- Convolve our reconstruction and low-pass filters together into one combined filter!

- Result is simple: convolution of two sinc functions is just larger sinc. In our case, approximate larger sinc with larger triangle, support 2, and convolve only once with it. Why does support > 2 make sense for down-scaling?
Scaling Down (5/5)

Why does something as complex-sounding as convolution of two differently-scaled sinc filters have such simple solution?

- Convolution of two sinc filters in spatial domain sounds complicated, but consider equivalent multiplication of two pulses in frequency domain
- Multiplication of two pulses is easy—product is narrower of two pulses:

\[
\begin{array}{c}
\hline
\\
\hline
\times
\hline
\hline
\end{array}
\]

- Narrower pulse in frequency domain is wider sinc in spatial domain
- Therefore, instead of filtering twice (once for reconstruction, once for low-pass), just filter once with wider of two filters
  - True for sinc or triangle approximation—it is the width of the support that matters
Outline

- Images & Hardware
- Example Applications
- Jaggies & Aliasing
- Sampling & Duals
- Convolution
- Filtering
- Scaling
- Reconstruction
- Scaling, continued
- Implementation
Algebraic Reconstruction (1/2)

- So far *textual explanations*; let’s get algebraic!

- Reconstructed, filtered image intensity function $h(x)$ returns image intensity at sample location $x$, where $x$ is *real*; convolution of source image $\hat{f}(x)$ (where $\hat{f}(x)$ is the box-filtered version of $f(x)$ produced by the scanner just prior to sampling) with filter $g(x)$ centered at $x$ is defined as:

$$ h(x) = \hat{f}(x) * g(x) = \int_{-\infty}^{\infty} \hat{f}(\tau) g(x - \tau) \, d\tau $$
Algebraic Reconstruction (2/2)

- Thoreau says “Our life is frittered away by detail... Simplify, simplify, simplify!”
- Only have pixel values $P_i$ as samples of $\hat{f}(x)$, and $g(x)$ has finite support $\rightarrow$ only evaluate at pixel locations
- Replace integral with finite sum over pixel locations covered by filter $g(x)$ centered at $x$.
- Thus convolution reduces to:

\[
h(x) = \sum_{i} P_i g(x - i) = \sum_{i} P_i g(i - x)
\]

For all pixels $i$ falling under filter support centered at $x$

Filter value at pixel location $i$

Pixel value at $i$

- Note: sign of argument of $g$ does not matter since filter is symmetric
- Note 2: Since convolution is commutative, can also think of $P_i$ as weight and $g$ as function
- e.g. if $x = 13.7$, and a triangle filter has support of 2, evaluate $g(13 - 13.7) = 0.3$ and $g(14 - 13.7) = 0.7$ and multiply those weights by pixel 13 and pixel 14’s values respectively
Unified Approach to Scaling Up and Down

• Nomenclature summary:
  - \( \hat{f}(x) \) is original scanned-in mostly band-limited continuous intensity function—never produced in practice!
  - \( P_i \) is sampled (box-filtered, comb-multiplied) \( \hat{f}(x) \) stored as pixel values
  - \( \hat{g}(x) \) is combined filter function, wider of the reconstruction and scaling filters
  - \( h(x) \) is reconstructed, filtered intensity function (either ideal continuous or discrete approximate)
  - \( h(k) \) is scaled version of \( h(x) \) dealing with image scaling
  - \( a \) is scale factor
  - \( k \) is index of a pixel in destination image

• Scaling up just like scaling down, only with a narrower reconstruction filter, support = 2

• Can parameterize image functions with scale: write a generalized formula for scaling up and down
  - \( g(x, a) \) is parameterized filter function
  - \( h(x, a) \) is reconstructed, filtered intensity function (either ideal continuous, or discrete approximation)
  - \( h(k, a) \) is scaled version of \( h(x, a) \) dealing with image scaling, sampled at pixel values of \( x=k \)
Reconstruction for Scaling

- Just as the filter is a function of $x$ and $a$ (the scale factor), so too is the reconstruction-filtered image function $h(x, a)$

$$h(x, a) = \sum_i P_i g(i - x, a)$$

For all pixels $i$ where $i$ is in support of $g$  
Pixel at integer $i$

Filter $g$, centered at $x$, evaluated at $i$

- Recall: scaling mapped destination pixel $k$ to (non-integer) source location $k/a$

$$\tilde{h}(k, a) = h \left( \frac{k}{a}, a \right) = \sum_i P_i g \left( i - \frac{k}{a}, a \right)$$

- Can *almost* write this sum out as code but still need to figure out summation limits and filter function
Two for the Price of One (1/2)

- Triangle filter, modified to be reconstruction for scaling by factor of $a$:

```
double g(double x, double a) {
    double radius;
    if (a < 1)
        radius = 1.0/a;
    else
        radius = 1.0;
    if ( (x < -radius) || (x > radius) )
        return 0;
    else
        return (1 - fabs(x)/radius) / radius;
}
```

- Careful...
  - this function will be called a lot. Can you optimize it?
  - remember: fabs() is just floating point version of abs()
Two for the Price of One (2/2)

- The pseudocode tells us support of $g$
  
  \[ a < 1 : (-1/a) < x < (1/a) \]
  \[ a \geq 1 : \quad -1 \leq x \leq 1 \]

- Can talk about leftmost and rightmost pixels that we need to examine for pixel $k$ in destination image as delimiting a window around $k/a$. Window is size 2 for scaling up, and size $2/a$ for scaling down.

- Note $k/a$ is not, in general, an integer, yet we want integer indexes of leftmost and rightmost pixels to consider. Use floor(..) and ceiling(..)

- If $a > 1$ (scale up)
  
  \[ \text{left} = \text{ceil}(\frac{k}{a} - 1) \]
  \[ \text{right} = \text{floor}(\frac{k}{a} + 1) \]

- If $a < 1$ (scale down)
  
  \[ \text{left} = \text{ceil}(\frac{k}{a} - \frac{1}{a}) \]
  \[ \text{right} = \text{floor}(\frac{k}{a} + \frac{1}{a}) \]
Triangle Filter Pseudocode

double h-tilde(int k, double a) {
    double sum = 0, weights_sum = 0;
    int left, right;
    if (a > 1) {
        left = ceil(k/a - 1.0);
        right = floor(k/a + 1.0);
    } else {
        left = ceil(k/a - 1.0/a);
        right = floor(k/a + 1.0/a);
    }
    for (int i = left; i <= right, i++) {
        sum += g(i - k/a, a) * orig_image.P_i;
        weights_sum += g(i - k/a, a);
    }
    result = sum/weights_sum;
}

- To ponder: When don’t you need to normalize sum? Why? How can you optimize this code?
The Big Picture, Algorithmically Speaking

• For each pixel in destination image:
  – determine which pixels in source image are relevant
  – by applying techniques described above, use values of source image pixels to generate value of current pixel in destination image
Normalizing Sum of Filter Weights (1/5)

- Notice in pseudocode that we sum filter weights, then normalize sum of weighted pixel contributions by dividing by filter weight sum. Why?

- Because non-integer-width filters produce sums of weights which vary as function of sampling position. Why is this a problem?
  - “Venetian blinds”—sums of weights increase and decrease away from 1.0 regularly across image.
  - This “bands” scaled image with regularly spaced lighter and darker regions.

- First we will show example of why filters with integer radii do sum to one and then why filters with real ones may not
Normalizing Sum of Filter Weights (2/5)

- Verify that integer-width filters have weights that always sum to one: notice that as filter shifts, one weight may be lowered, but it has a corresponding weight on opposite side of filter, a radius apart, that increases by same amount.

Consider our familiar triangle filter.

When we place the filter halfway between two pixels, we get two weights, each 0.5. The symmetry of pixel placement ensures that we will get identical values on each side of the filter. The two weights again sum to 1.0.

When we place it directly over a pixel, we have one weight, and it is exactly 1.0. Therefore, the sum of weights (by definition) is 1.0.

If we slide the filter 0.25 units to the right, we have effectively slid the two pixels under it by 0.25 units to the left relative to it. Since the pixels move by the same amount, an increase on one side of the filter will be perfectly compensated for by a decrease on the other. Our weights again sum to 1.0.
But when filter radius is non-integer, sum of weights changes for different filter positions.

In this example, first position filter (radius 2.5) at location A. Intersection of dotted line at pixel location with filter determines weight at that location. Now consider filter placed slightly right of A, at B.

Differences in new/old pixel weights shown as additions or subtractions. Because filter slopes are parallel, these differences are all same size. But there are 3 negative differences and 2 positive, hence two sums will differ.
Normalizing Sum of Filter Weights (4/5)

- When radius is \textit{integer}, contributing pixels can be paired and contribution from each pair is equal. The two pixels of a pair are at a radius distance from each other.

- Proof: see equation for value of filter with radius $r$ centered at non-integer location $d$:

\[
g(x) = \frac{1}{r} \left( 1 - \frac{|x|}{r} \right)
\]

- Suppose pair is $(b,c)$ as in figure above. Contribution sum becomes (note $|d-c| = x$ and $|d-b| = r-x$):

\[
g(b) + g(c) = \frac{1}{r} \left( 1 - \frac{|d-b|}{r} \right) + \frac{1}{r} \left( 1 - \frac{|d-c|}{r} \right)
\]

\[
= \frac{1}{r} \left( 2 - \frac{r-x}{r} - \frac{x}{r} \right) = \frac{1}{r} \left( 2 - \frac{r}{r} \right) = \frac{1}{r}
\]
Normalizing Sum of Filter Weights (5/5)

- Sum of contributions from two pixels in a pair does not depend on \( d \) (location of filter center).

- Sum of contributions from all pixels under filter will not vary, no matter where we’re reconstructing.

- For integer width filters, we **do not** need to normalize.

- When scaling **up**, we always have integer-width filter, so we **don’t** need to normalize!

- When scaling **down**, our filter width is generally non-integer, and we **do** need to normalize.

- Can you rewrite the pseudocode to take advantage of this knowledge?
Scaling in 2D

• Do it in 1D twice – once to rows, once to columns
  – easy to implement
  – for certain filters, works pretty decently
  – requires intermediate storage

• Do it in 2D “all at once”
  – harder to implement
  – more general
  – generally more “correct” – deals with high frequency “diagonal” information

• For your assignment the first approach suffices, and is easier to implement

• Note that ideally we would like to use a cone for our filter, but this is too difficult to deal with, so we are satisfied with a pyramid.
Pyramid Filter

- 2D version of triangle filter (the one actually used on 2D images) is a pyramid filter

- Certain mapping operations (such as image blurring, sharpening, edge detection, etc.) change destination pixel values, but don’t remap pixel locations, i.e., don’t sample between pixel locations. Their filters can be precomputed as a "kernel" (or "pixel mask")

- Other mappings, such as image scaling, require sampling between pixel locations. For these operations, often easier to approximate pyramid filter by applying triangle filters twice, once along x-axis of source, once along y-axis
Precomputed Filter Kernels (1/3)

- Filter kernel is filter value precomputed at predefined sample points.
- Kernels are usually square, odd number by odd number size grids (center of kernel can be at pixel that you are working with [e.g. 3x3 kernel shown here]):

<table>
<thead>
<tr>
<th></th>
<th>1/16</th>
<th>2/16</th>
<th>1/16</th>
</tr>
</thead>
<tbody>
<tr>
<td>2/16</td>
<td>4/16</td>
<td>2/16</td>
<td></td>
</tr>
<tr>
<td>1/16</td>
<td>2/16</td>
<td>1/16</td>
<td></td>
</tr>
</tbody>
</table>

- Why does precomputation only work for mappings which sample only at integer pixel intervals in original image?
- If filter location is moved by fraction of a pixel in source image, pixels fall under different locations within filter, correspond to different filter values.
- Can’t precompute for this.
Evaluating the kernel

- Filter kernel evaluated as normal filters are: multiply pixel values in source image by filter values corresponding to their location within filter.

- Place kernel’s center over integer pixel location to be sampled. Each pixel covered by kernel is multiplied by corresponding kernel value; results are summed.

- Note: have not dealt with boundary conditions. One common tactic is to act as if there is a buffer zone where the edge values are repeated.
Precomputed Filter Kernels

Filter kernel in operation

- Pixel in destination image is weighted sum of multiple pixels in source image
Supersampling for Image Synthesis

Anti-aliasing of primitives in practice

- Bad Old Days: Generate image and post-filter, e.g. with pyramid – blurs image (with its aliases)
- Alternative: super-sample and post-filter, to approximate pre-filtering before sampling
  - pixel’s value computed by taking weighted average of several point samples around pixel’s center. Again, approximating (convolution) integral with weighted sum
  - stochastic point sampling as an approximation converges fast, much faster than equi-spaced grid sampling

Why does supersampling work?

- sampling at a higher frequency pushes the replicas apart, and since spectra fall off approximately as $1/f^p$, for $1<p<2$ (i.e. somewhere between linearly and quadratically), the tails overlap much less, causing much less corruption before the low-pass filtering
- with fewer than 128 distinguishable levels of intensity, being off by one step is hardly noticeable
- stochastic sampling may introduce some random noise