# CS448f: Image Processing For Photography and Vision

Lecture 2

# Today:

- More about ImageStack
- Sampling and Reconstruction
- Assignment 1

# ImageStack

- A collection of image processing routines

- Each routine bundled into an Operation class
  - void help()
  - void parse(vector<string> args)
  - Image apply(Window im, … some parameters …)

# ImageStack Types

- Window:
  - A 4D volume of floats, with each scanline contiguous in memory.

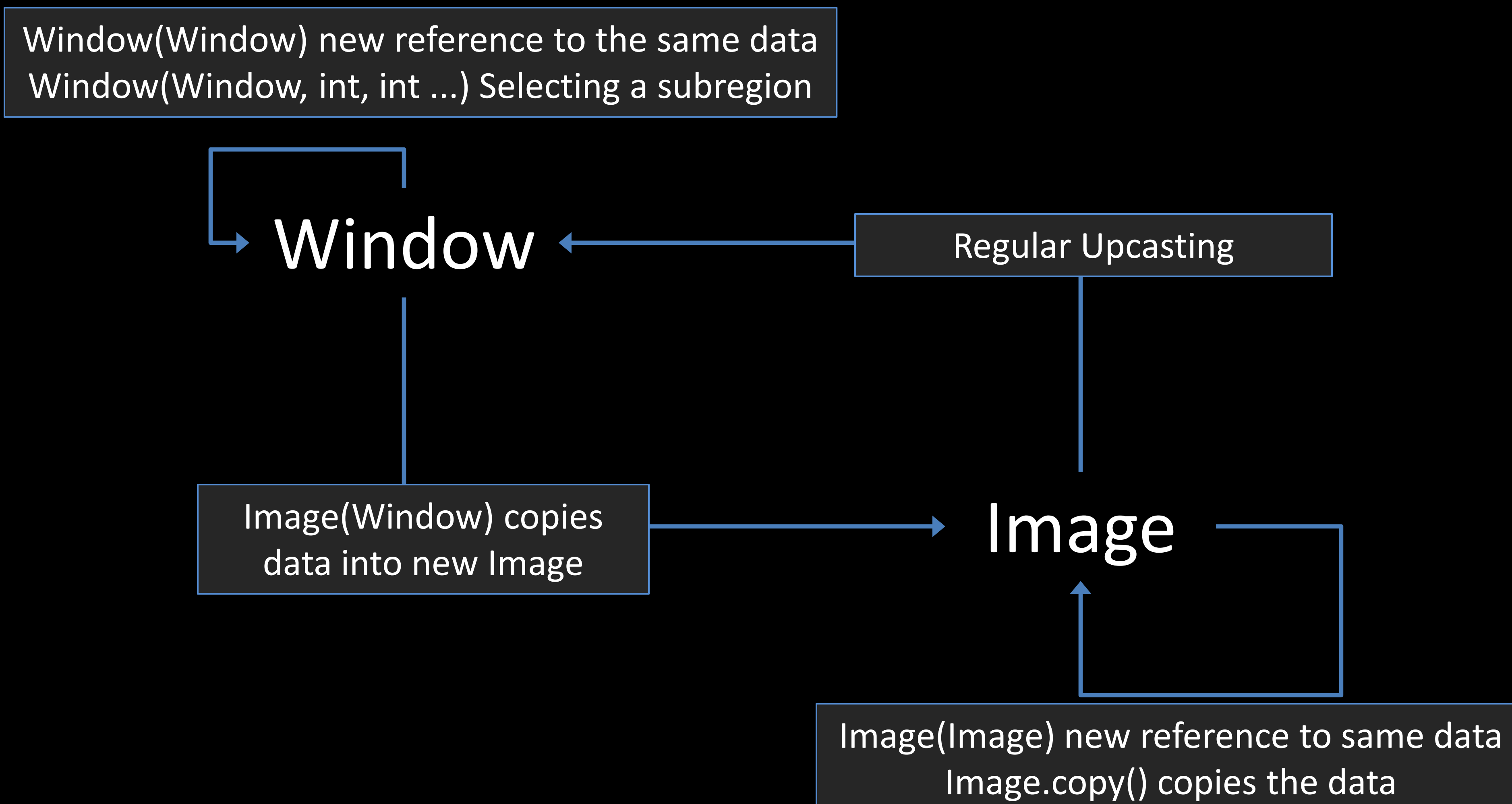
```
class Window {
    int frames, width, height, channels;
    float *operator()(int t, int x, int y);
    void sample(float t, float x, float y, float *result)
};
```

# ImageStack Types

- Image:
  - A subclass of Window that is completely contiguous in memory
  - Manages its own memory via reference counting (so you can make cheap copies)

```
class Image : public Window {
   Image copy();
};
```

# Image and Windows



# 4 Way to Use ImageStack

- Command line
- As a library
- By extending it
- By modifying it

# Fun things you can do with ImageStack

- ImageStack –help
- ImageStack –load input.jpg –save output.png
- ImageStack –load input.jpg –display
- ImageStack –load a.jpg –load b.jpg –add –save c.jpg
- ImageStack –load a.jpg –loop 10 – –scale 1.5 –display
- ImageStack –load a.jpg –eval “(val > 0.5) ? val : 0”
- ImageStack –load a.jpg –resample width/2 height/2
- ... all sorts of other stuff

# Where to get it:

- The course website
- http://cs448f.stanford.edu/imagestack.html

```
float *operator()(int t, int x, int y)
```

↓

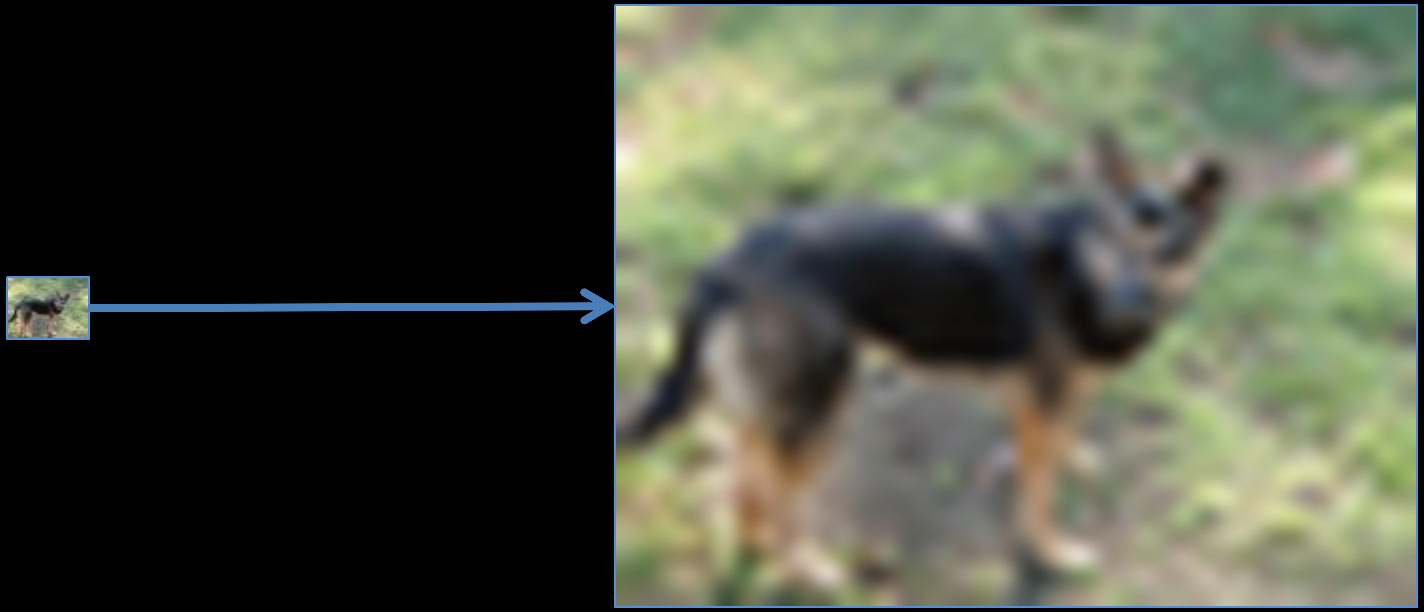
**Sampling and Reconstruction**

↓

```
void sample(float t, float x, float y, float *result);
```

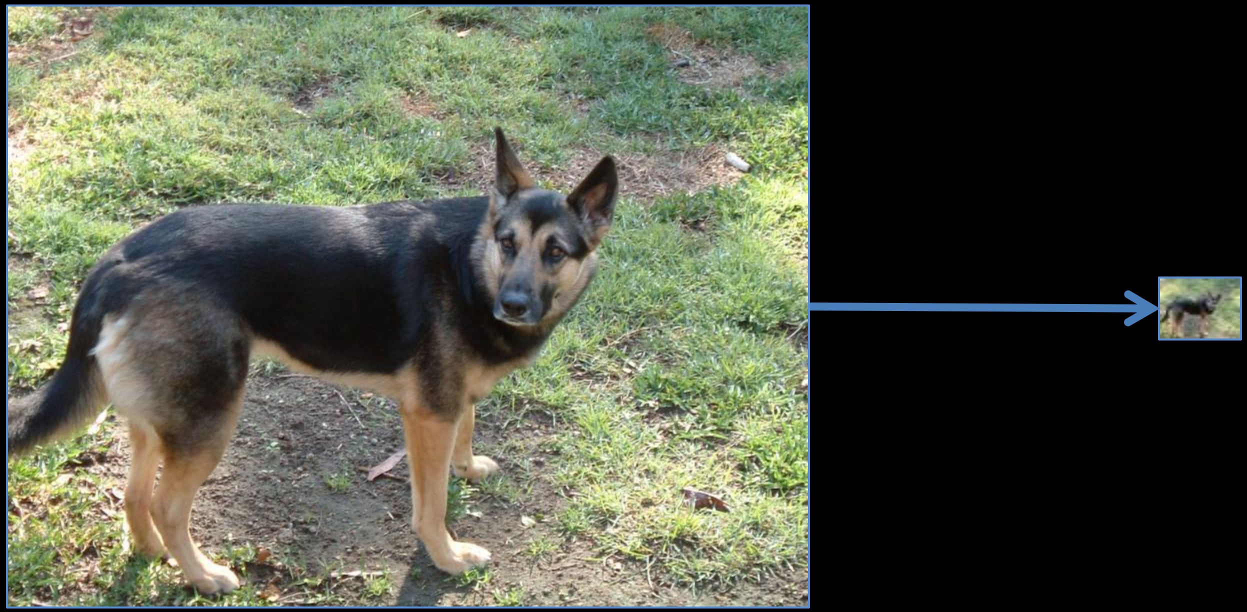
# Why resample?

- Making an image larger:



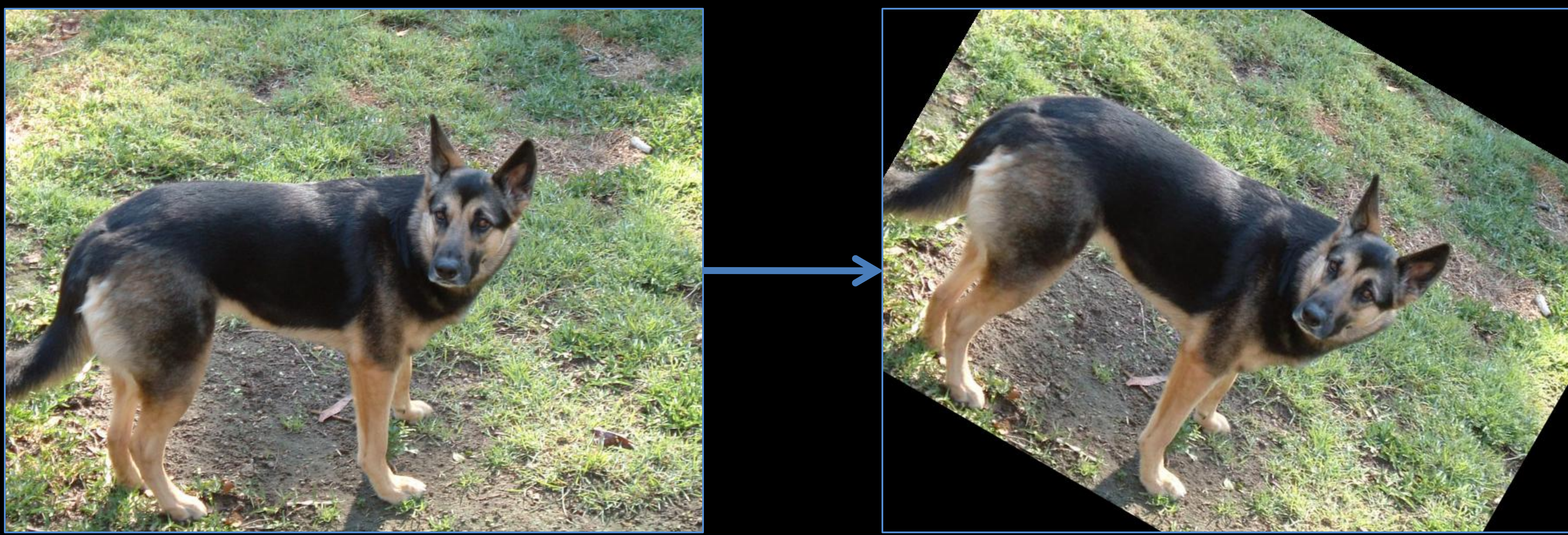
# Why resample?

- Making an image smaller:



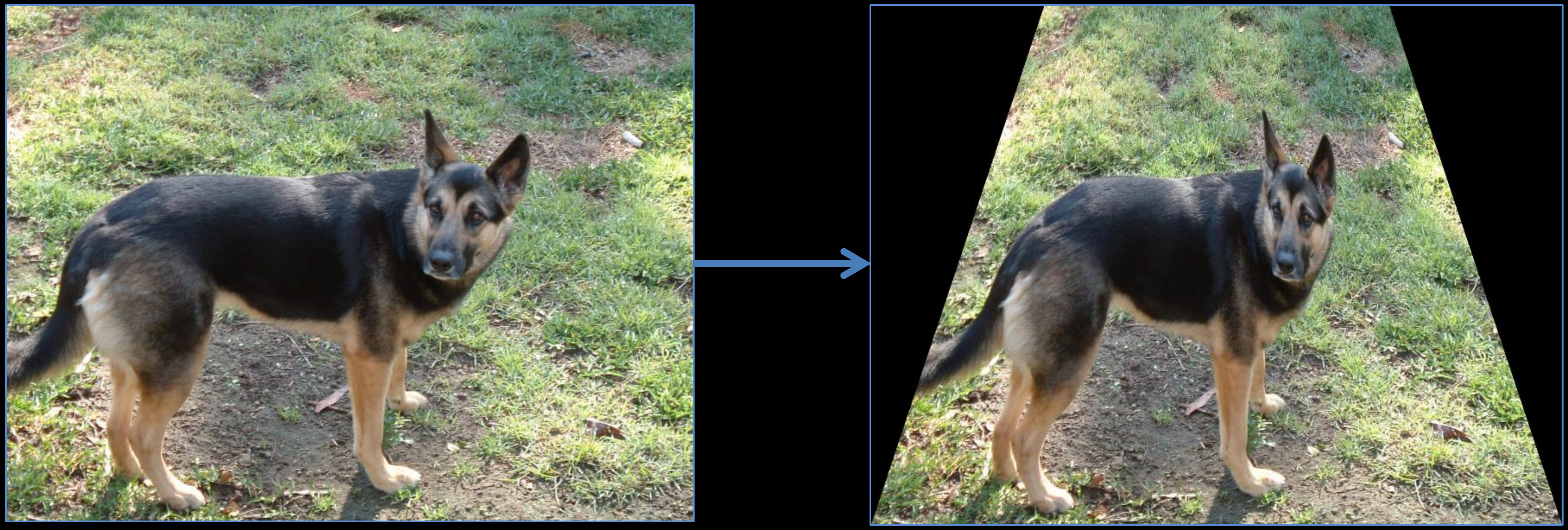
# Why resample?

- Rotating an image:



# Why resample?

- Warping an image (useful for 3D graphics):



# Enlarging images

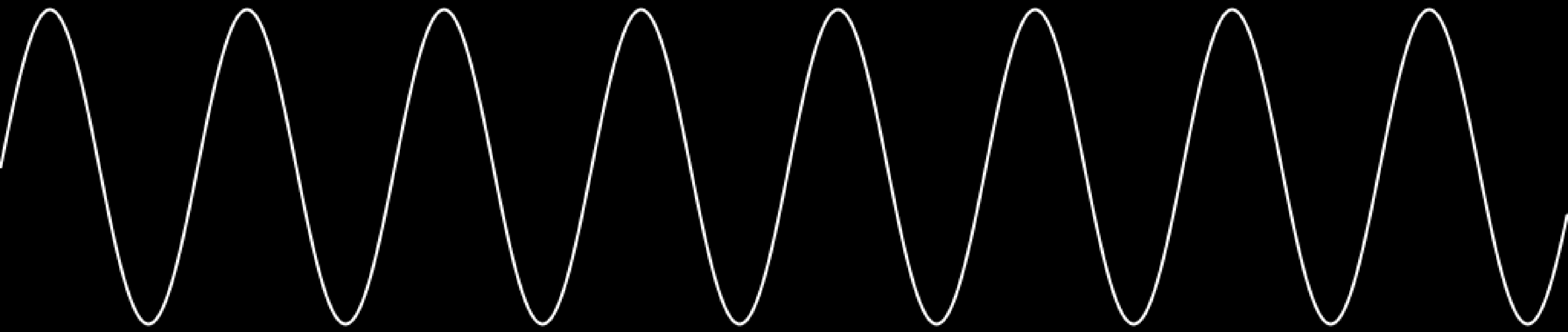
- We need an interpolation scheme to make up the new pixel values
- (applet)
- Interpolation = Convolution

# What makes an interpolation good?

- Well… let’s look at the difference between the one that looks nice and the one that looks bad…

# Fourier Space

- An image is a vector
- The Fourier transform is a change of basis
  - i.e. an orthogonal transform
- Each Fourier basis vector is something like this:

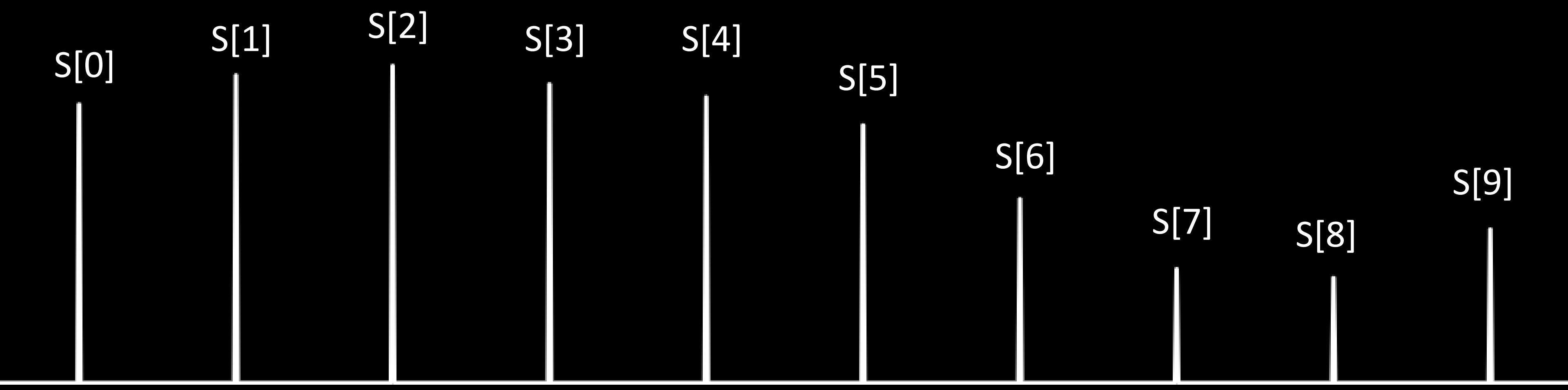


# Fourier Space

- The Fourier transform expresses an image as a sum of waves of different frequencies
- This is useful, because our artifacts are confined to high frequencies
- In fact, we probably don't want ANY frequencies that high in our output – isn't that what it means to be smooth?

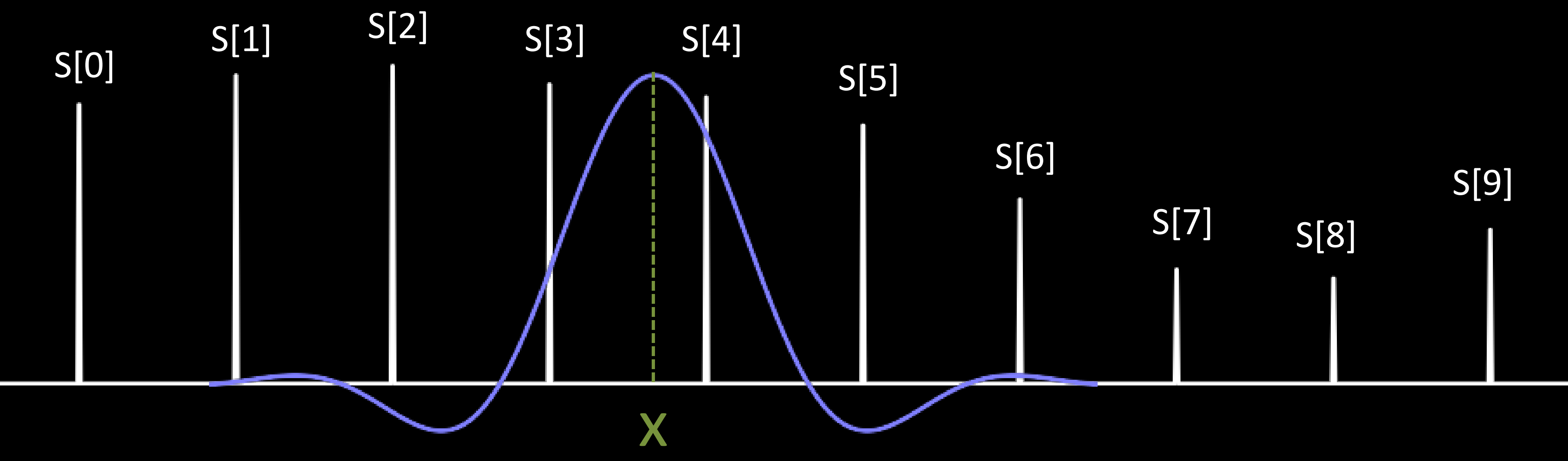
# Deconstructing Sampling

- We get our output by making a grid of spikes that take on the input values s:



# Deconstructing Sampling

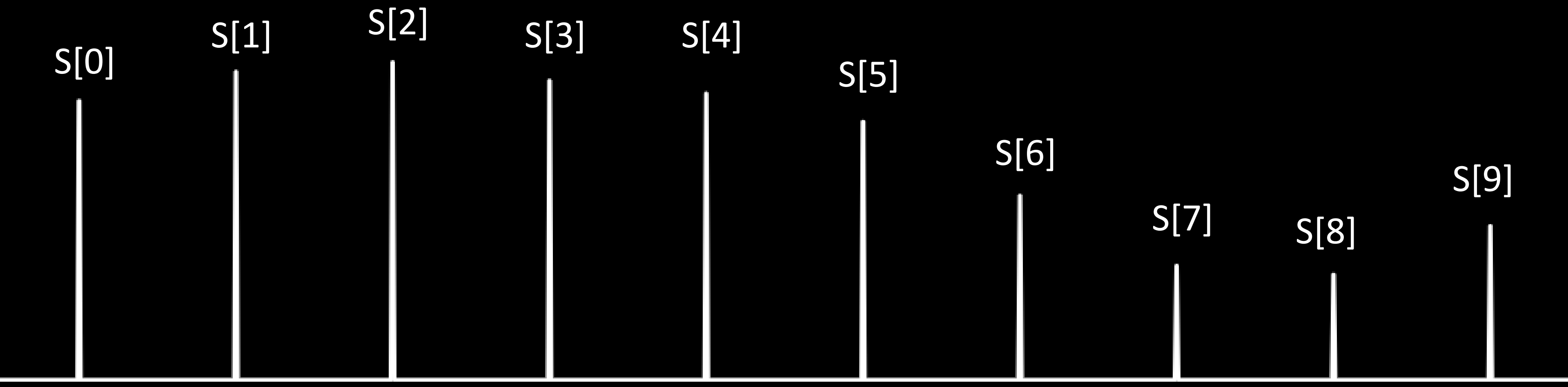
- Then evaluating some filter f at each output location x:




```
for (i = 1; i < 7; i++) output[x] += f(x-i)*s[i];
```

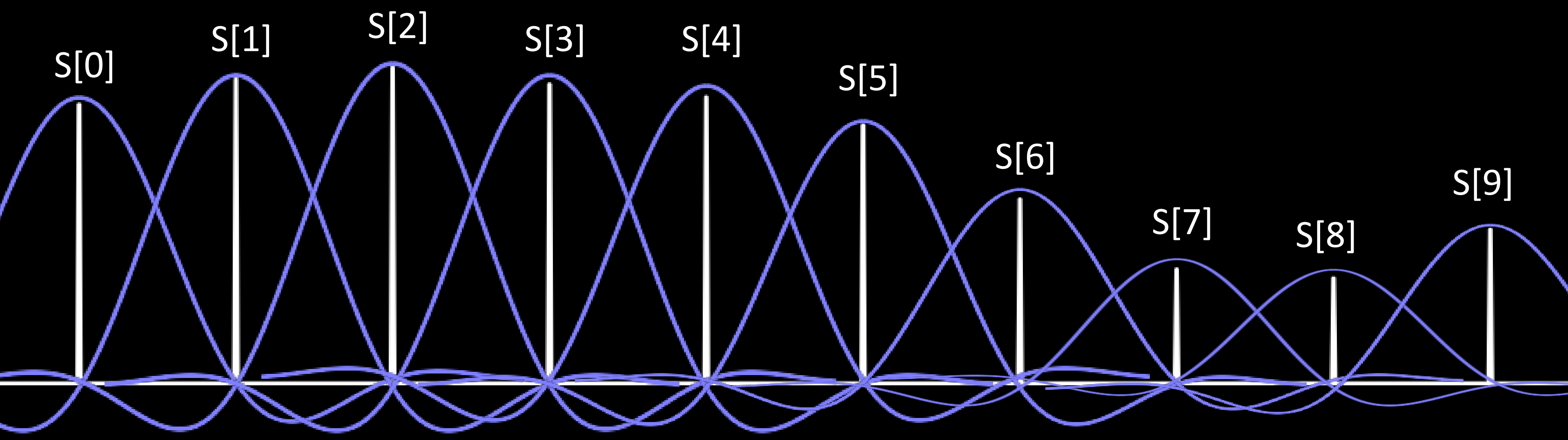
# Alternatively

- Start with the spikes



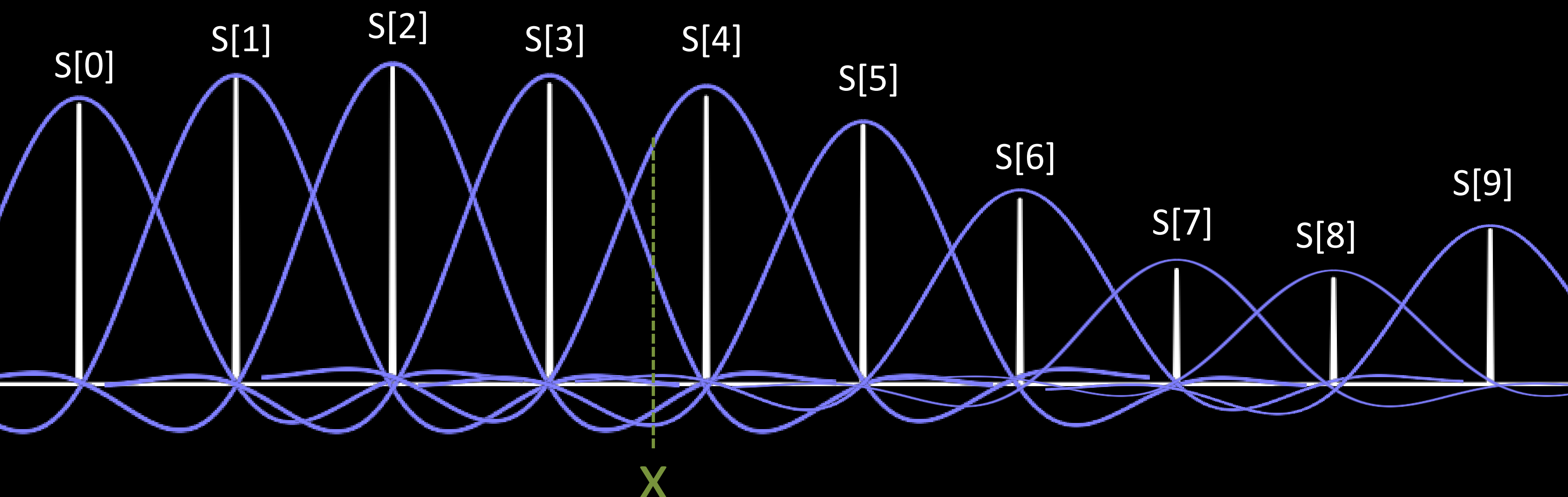
# Alternatively

- Convolve with the filter f



# Alternatively

- And evaluate the result at x




for (i = 1; i < 7; i++) output[x] += s[i]*f(i-x);

# They're the same

- Method 1:

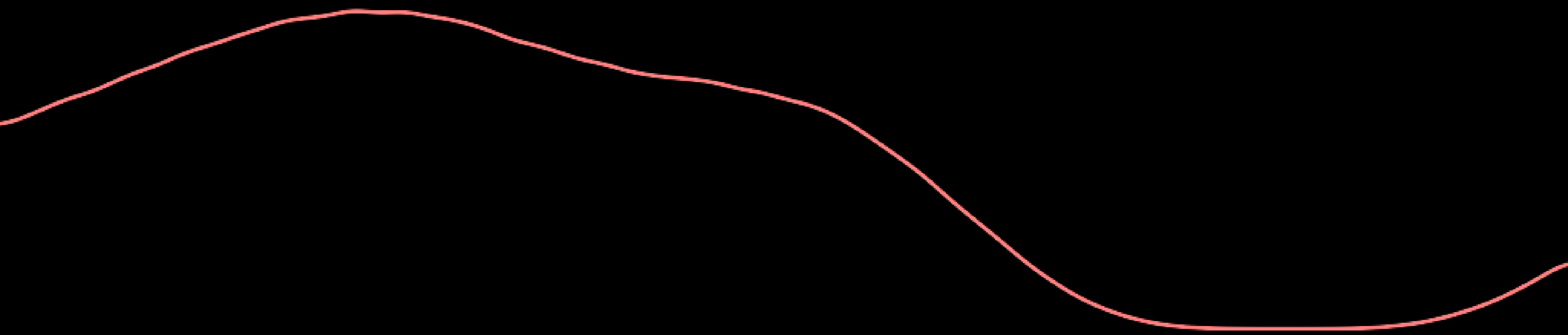
```
for (i = 1; i < 7; i++) output[x] += s[i]*f(i-x);
```

- Method 2:

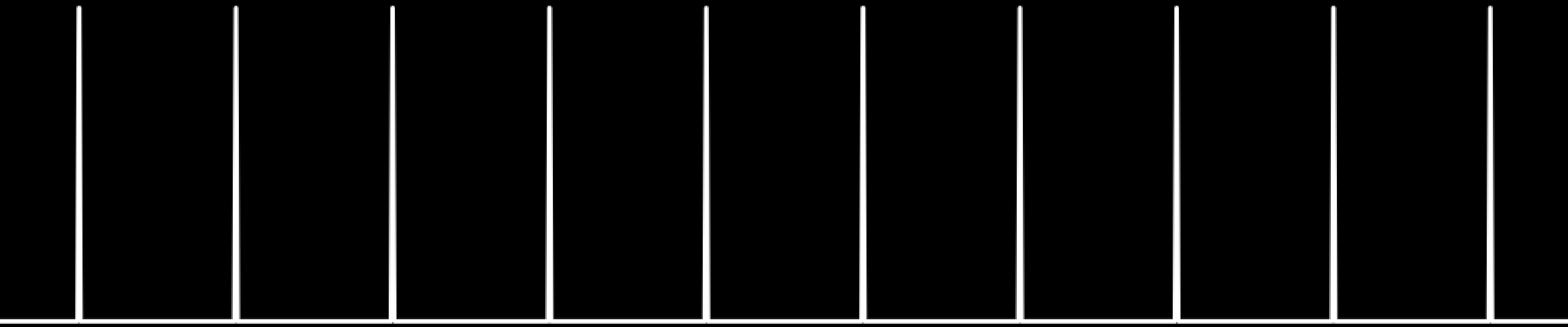
```
for (i = 1; i < 7; i++) output[x] += f(x-i)*s[i];
```

- f is symmetric, so f(x-i) = f(i-x)

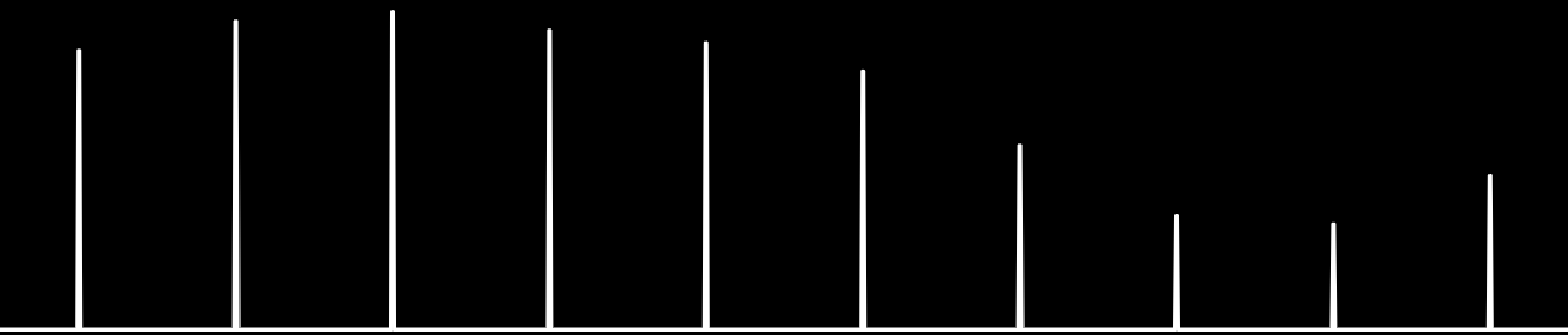
# Start with the (unknown) nice smooth desired result R



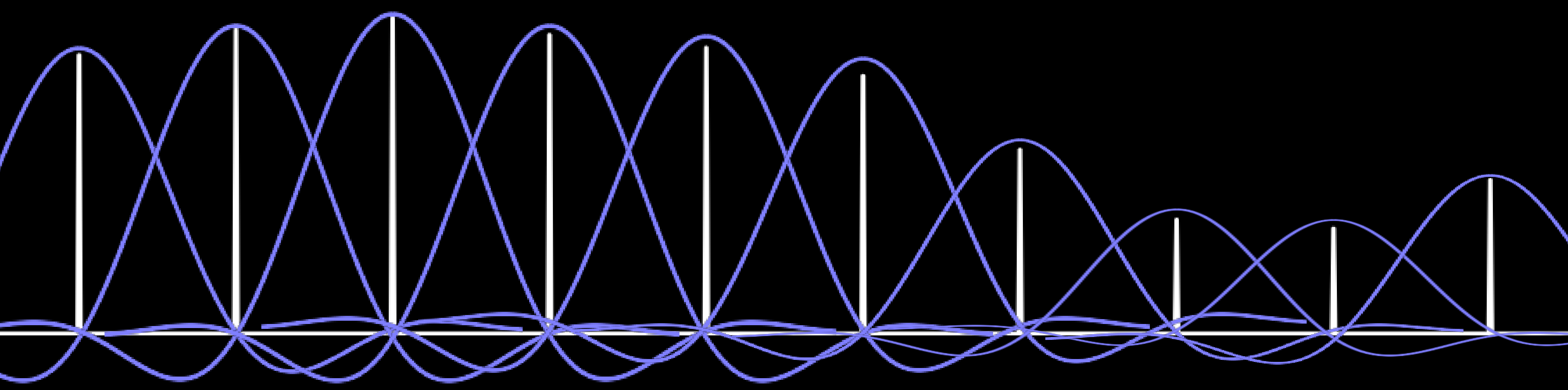
# Multiply by an impulse train T



# Now you have the known sampled signal R.T

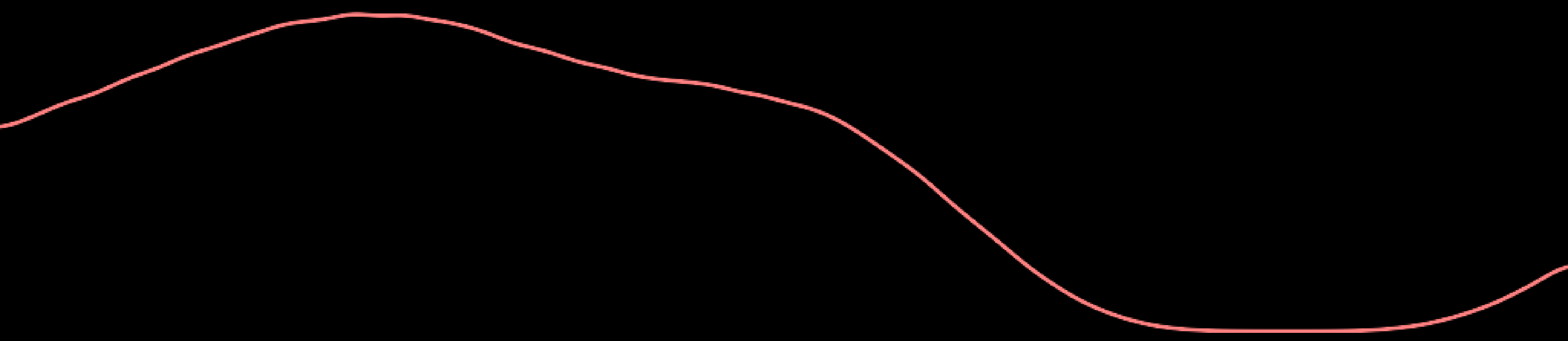


# Convolve with your filter f
# Now you have (R.T)*f



# And get your desired result R

R = (R.T)*f

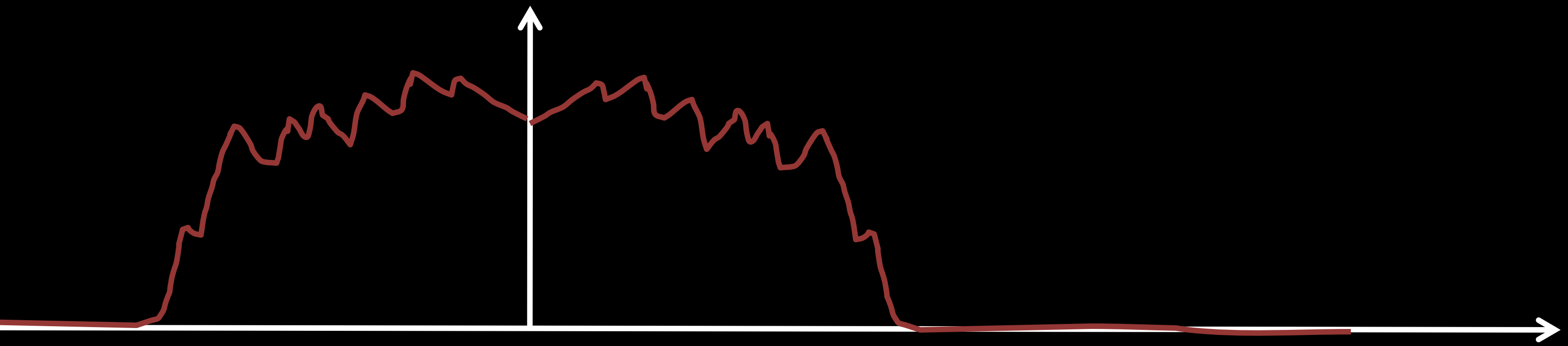


# Therefore

- Let’s pick f to make (R.T)*f = R
- In other words, convolution by f should undo multiplication by T
- Also, we know R is smooth
  - has no high frequencies

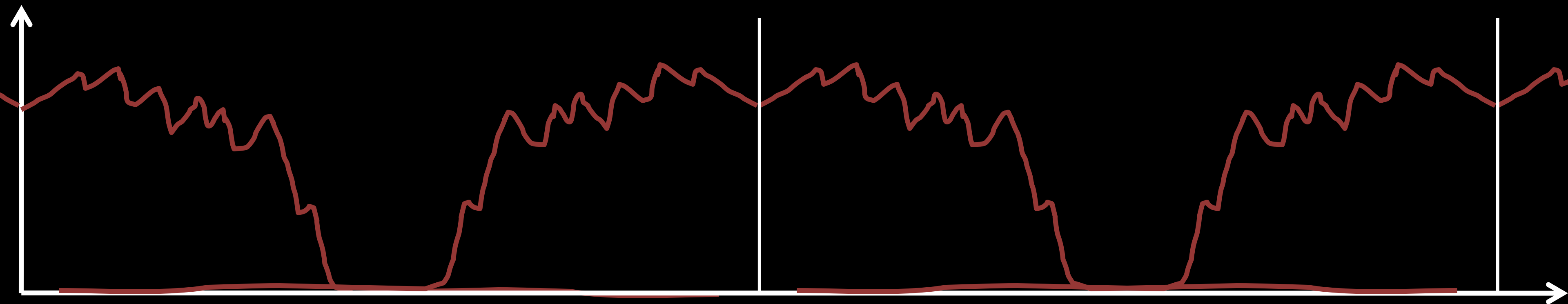
# Meanwhile, in Fourier space…

- Let’s pick f’ to make (R’*T’).f = R’
- In other words, multiplication by f’ should undo convolution by T’
- Also, we know R’ is zero above some point
  - has no high frequencies



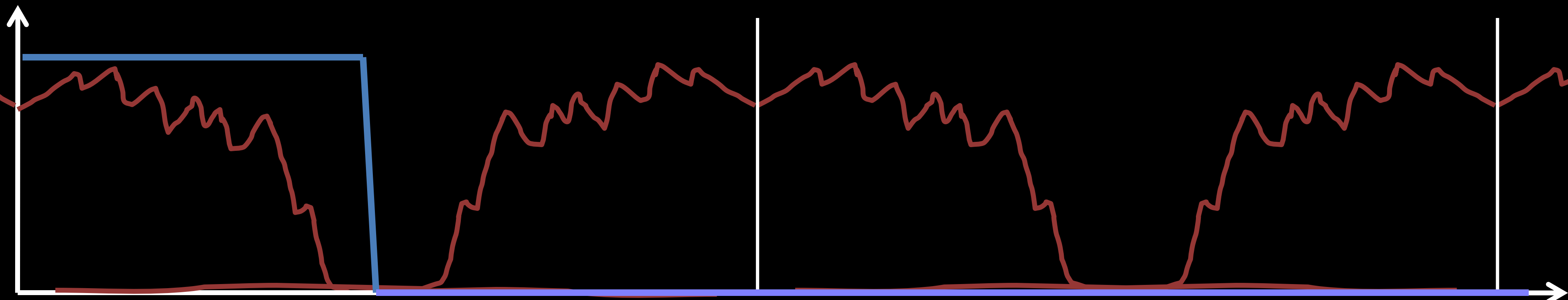
# T vs T’

- Turns out, the Fourier transform of an impulse train is another impulse train (with the inverse spacing)
- R’*T’:



# T vs T’

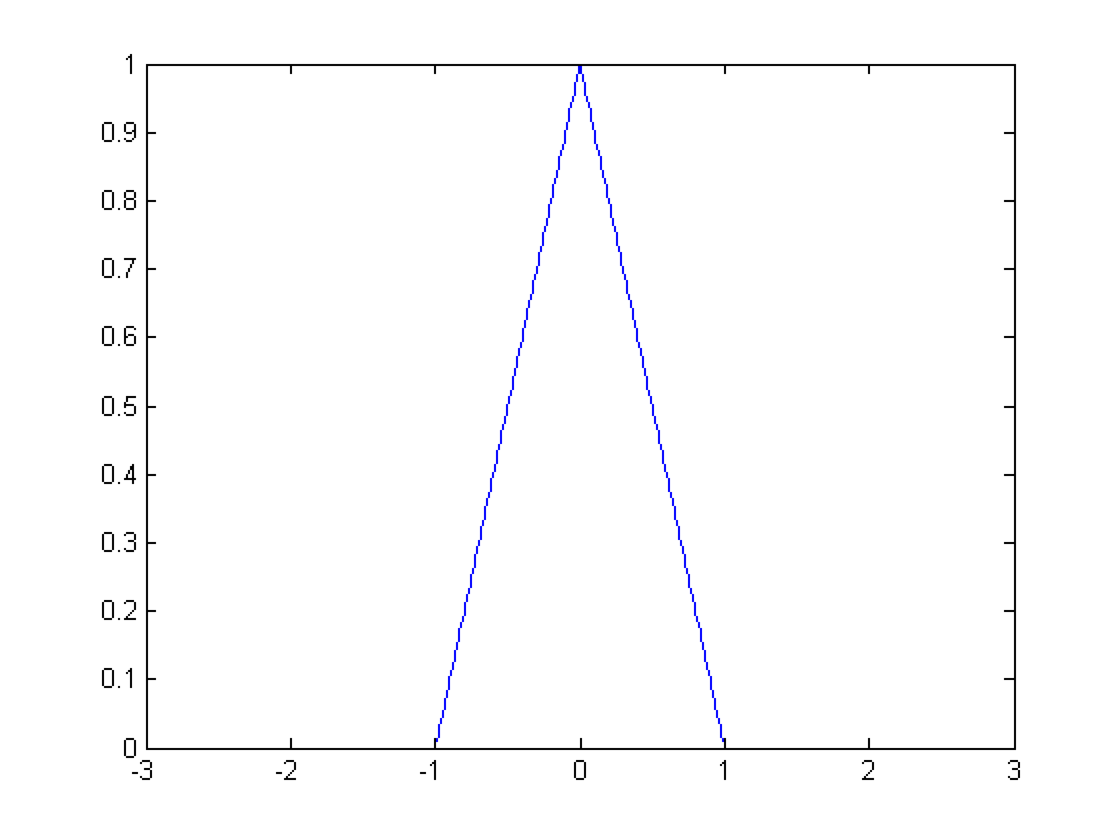
- All we need to do is pick an f’ that gets rid of the extra copies:
- (R’*T’).f’:



# A good f’

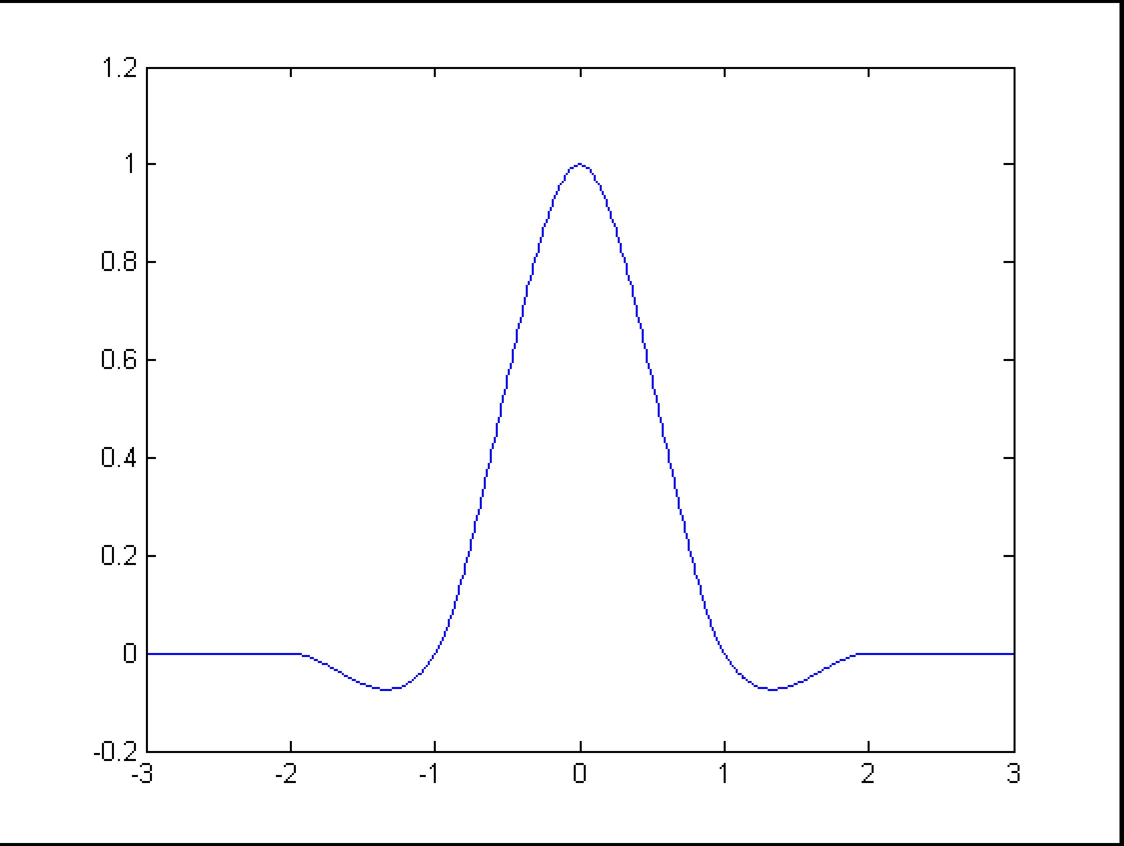
- Preserves all the frequencies we care about
- Discards the rest
- Allows us to resample as many times as we like without losing information
- (((((R’*T’).f’)*T’.f’)*T’.f’)*T’.f’) = R’

# How do our contenders match up?



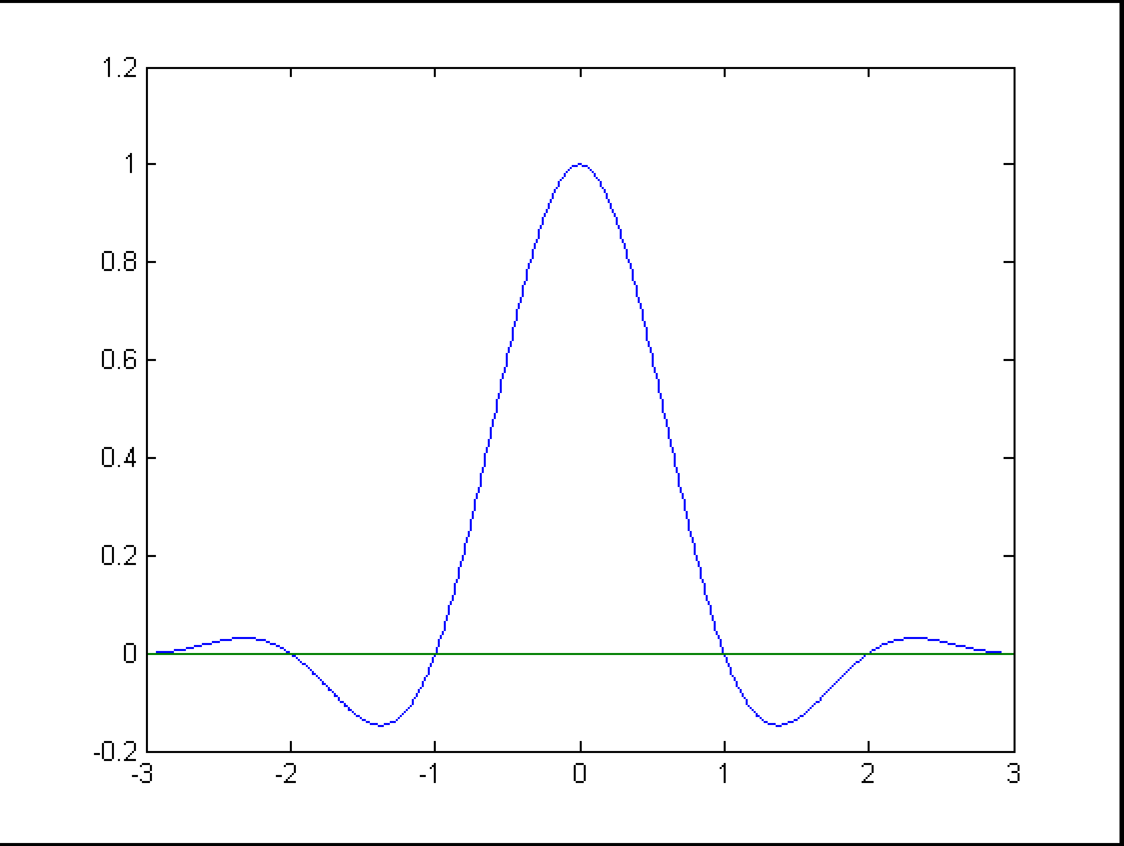

Linear

# How do our contenders match up?



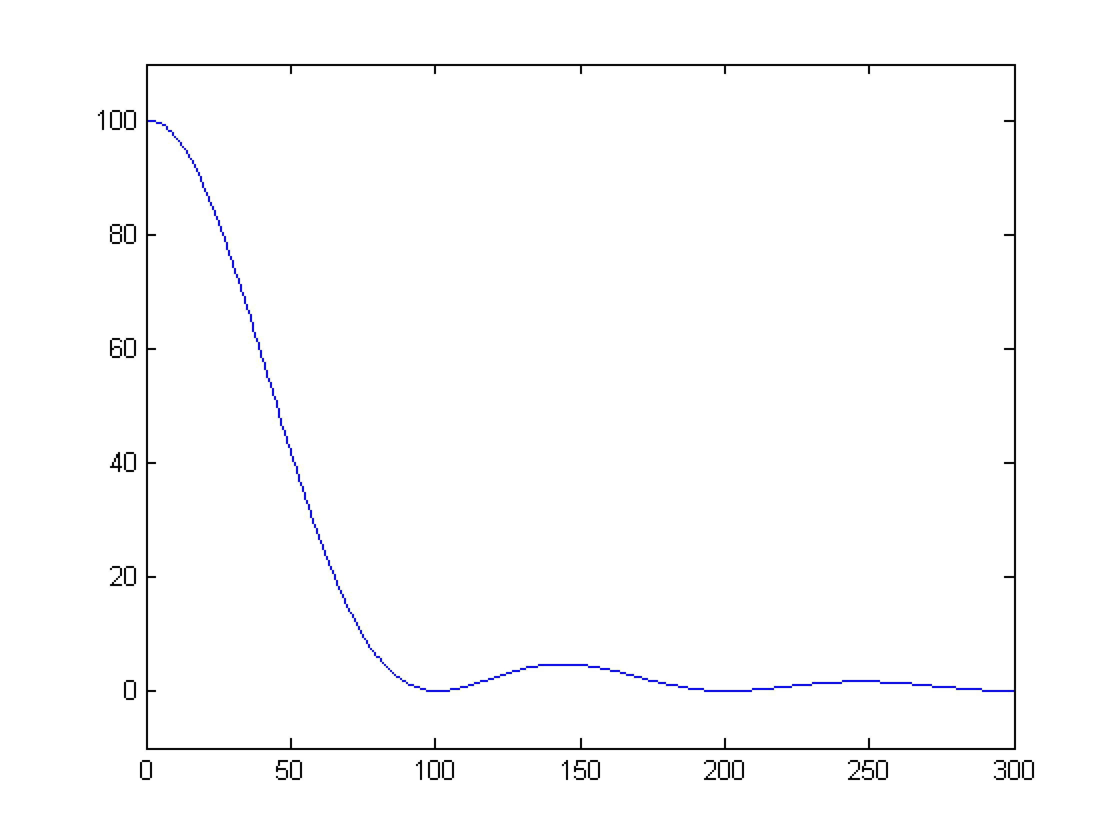

Cubic

# How do our contenders match up?



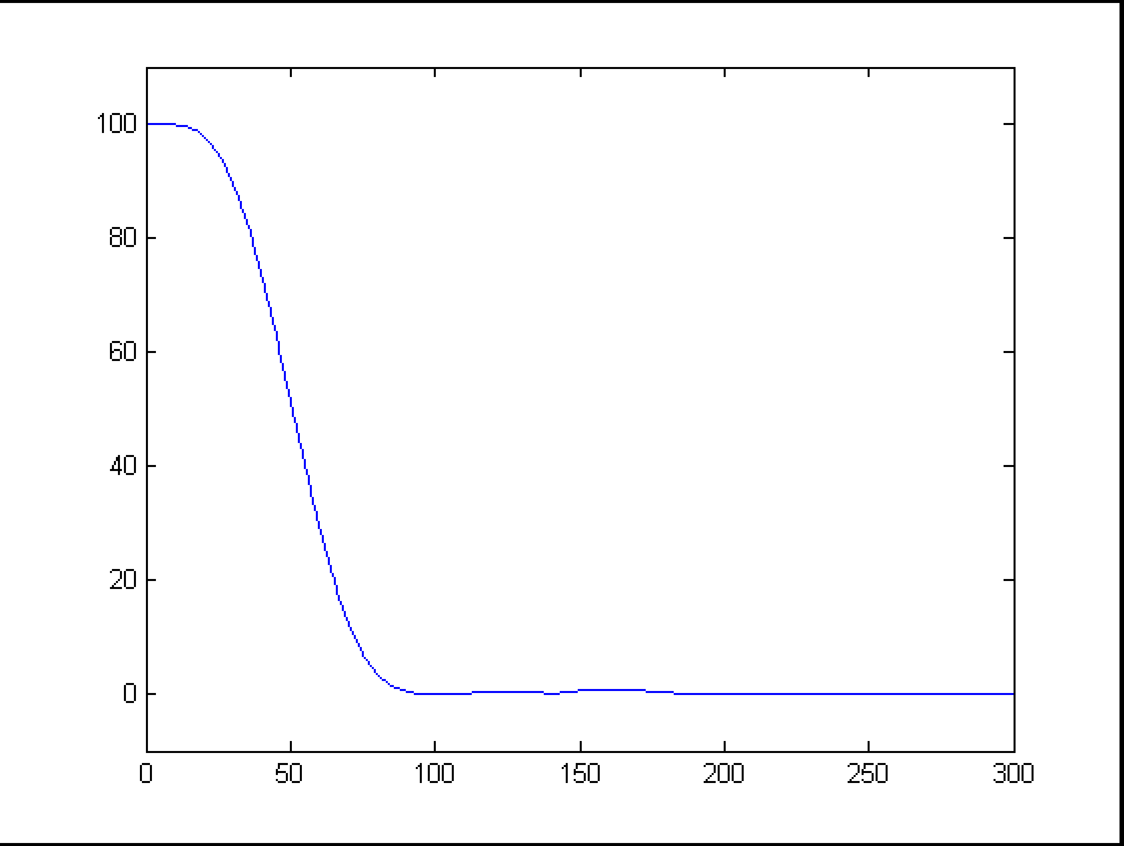

Lanczos 3 = sinc(x)*sinc(x/3)

# How do our contenders match up?



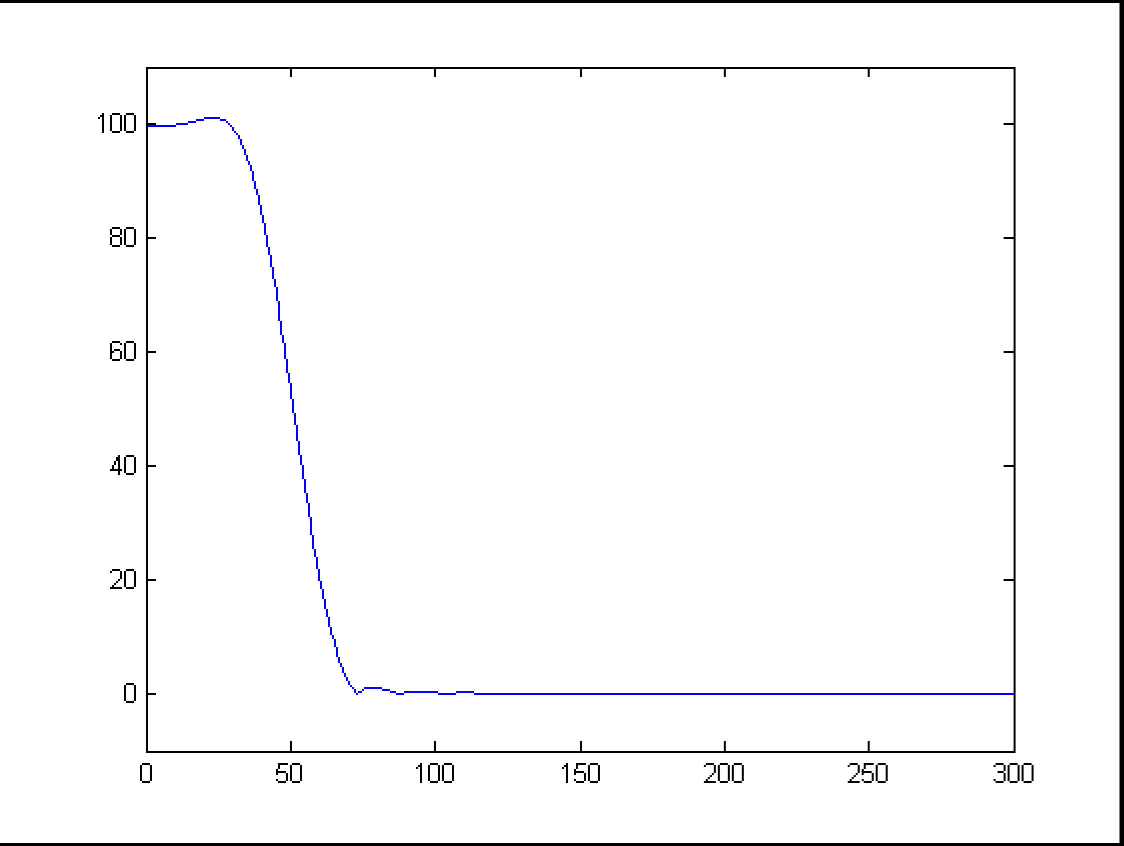

Linear

# How do our contenders match up?



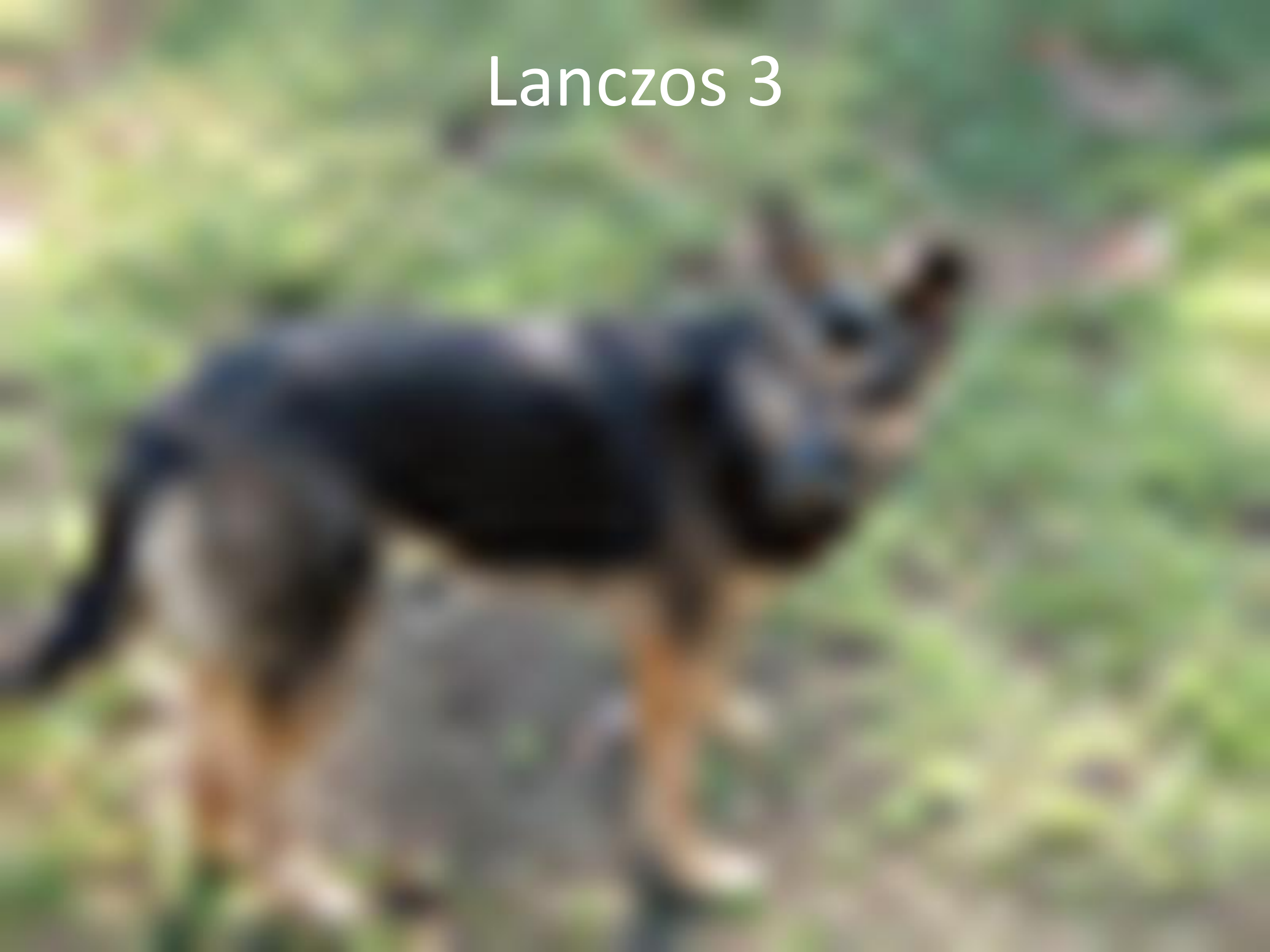

Cubic

# How do our contenders match up?




Lanczos 3

# Lanczos 3



# Sinc - The perfect result?

# A good f’

- Should throw out high-frequency junk
- Should maintain the low frequencies
- Should not introduce ringing
- Should be fast to evaluate
- Lanczos is a pretty good compromise
- Window::sample(...);
- Window::sampleLinear(...);

# Inverse Warping

- If I want to transform an image by some rotation R, then at each output pixel x, place a filter at $R^{-1}(x)$ over the input.
- In general warping is done by
  - Computing the inverse of the desired warp
  - For every pixel in the output
    - Sample the input at the inverse warped location

# Forward Warping (splatting)

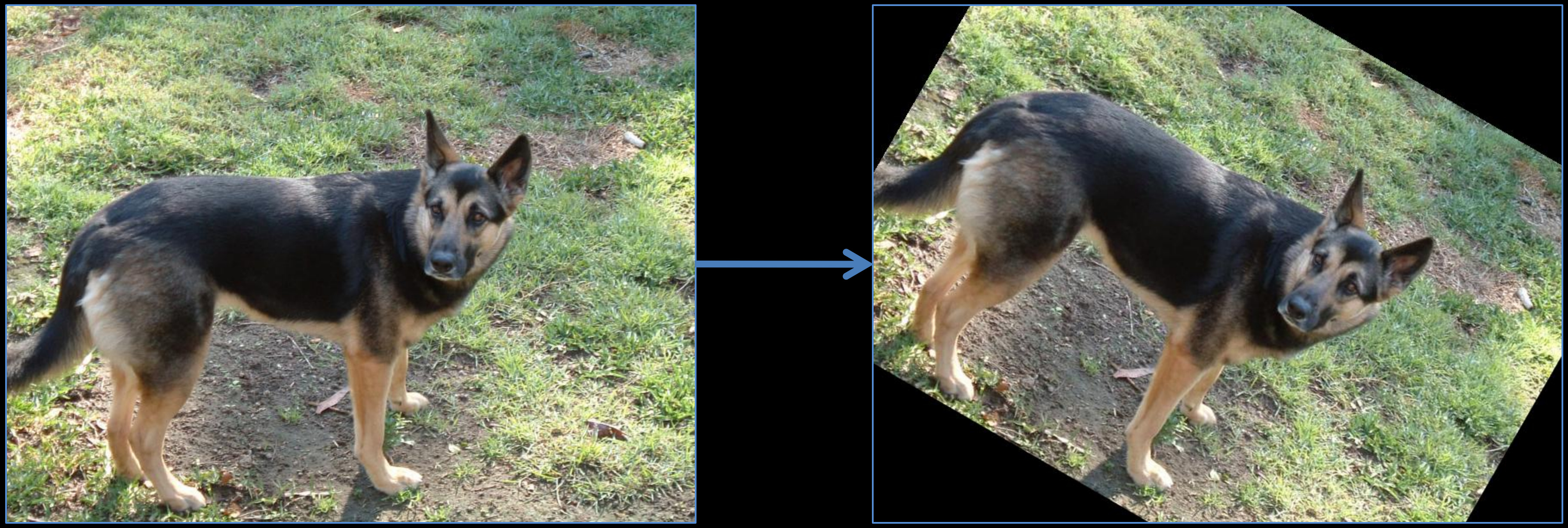
- Some warps are hard to invert, so...
- Add an extra weight channel to the output
- For every pixel x in the input
  - Compute the location y in the output
  - For each pixel under the footprint of the filter
    - Compute the filter value w
    - Add (w.r w.g w.b w) to the value stored at y
- Do a pass through the output, dividing the first n channels by the last channel

# Be careful sizing the filter

- If you want to enlarge an image, the filter should be sized according to the *input* grid
- If you want to shrink an image, the filter should be sized according to the *output* grid of pixels
  - Think of it as enlarging an image in reverse
  - You don't want to keep ALL the frequencies when shrinking an image, in fact, you're trying to throw most of them out

# Rotation

- Ok, let’s use the lanczos filter I love so much to rotate an image:



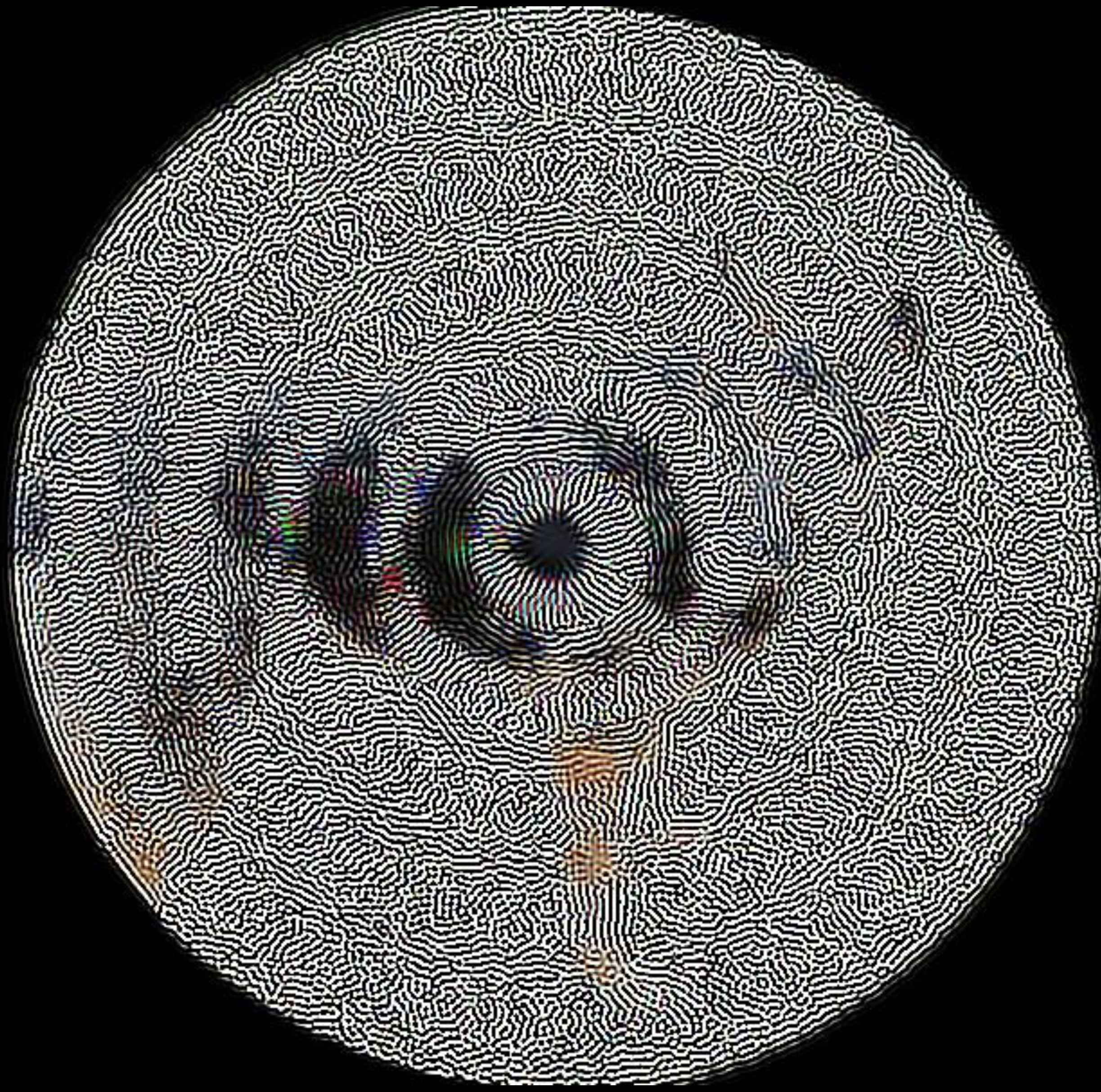
# Original

# Rotated by 30 degrees 12 times

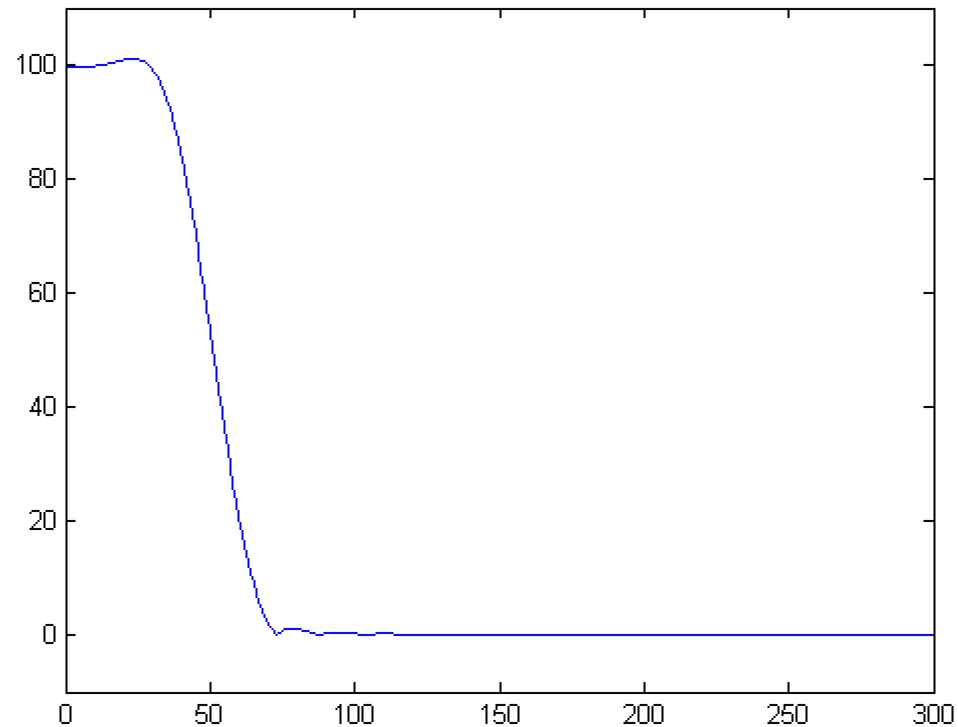
# Rotated by 10 degrees 36 times

# Rotated by 5 degrees 72 times

# Rotated by 1 degree 360 times



# What went wrong?



# Your mission: Make rotate better

- Make it accurate and fast
- First we'll check it's plausible:

```
ImageStack -load a.jpg -rotate <something> -display
```

- Then we'll time it:

```
ImageStack -load a.jpg -time --loop 360 ---rotate 1
```

- Then we'll see how accurate it is:

```
for ((i=0;i<360;i++)); do
   ImageStack -load im.png -rotate 1 -save im.png
done
ImageStack -load orig.png -crop width/4 height/4 width/2
   height/2 -load im.png -crop width/4 height/4 width/2
   height/2 -subtract -rms
```

# Targets:

- RMS must be < 0.07
- Speed must be at least as fast as -rotate

- My solution has RMS ~ 0.05
- Speed ~ 50% faster than -rotate (No SSE)

- Prizes for the fastest algorithm that meets the RMS requirement, most accurate algorithm that meets the speed requirement

# Grade:

- 20% for having a clean readable algorithm
- 20% for correctness
- 20% for being faster than -rotate
- 40% for being more accurate than -rotate

# Due:

- Email your modified Geometry.cpp (and whatever other files you modified) in a zip file to us by midnight on Thu Oct 1
  - cs448f-aut0910-staff@lists.stanford.edu

# Finally, Check out this paper:

- Image Upsampling via Imposed Edge Statistics
- http://www.cs.huji.ac.il/~raananf/projects/upsampling/upsampling.html