CS448f: Image Processing For Photography and Vision

Fast Filtering

Problems in Computer Vision

Computer Vision in One Slide

1) Extract some features from some images

2) Use these to formulate some (hopefully linear) constraints

3) Solve a system of equations your favorite method to produce...

Computer Vision in One Slide

0) Blur the input

1) Extract some features from some images

2) Use these to formulate some (hopefully linear) constraints

3) Solve a system of equations your favorite method to produce...

Why do we blur the input?

- To remove noise before processing
- So we can use simpler filters later
- To decompose the input into different frequency bands
  - tonemapping, blending, etc

- **Fast Filtering**
  - Composing Filters
  - Fast Rect and Gaussian Filters
  - Local Histogram Filters
  - The Bilateral Grid
- **Applications**
  - Joint Bilateral Filter
  - Flash/No Flash
  - Joint Bilateral Upsample
  - ASTA

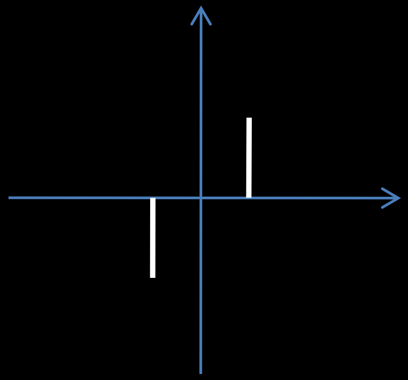
- **Fast Filtering**
  - Composing Filters
  - Fast Rect and Gaussian Filters
  - Local Histogram Filters
  - **The Bilateral Grid**
- **Applications**
  - Joint Bilateral Filter
  - Flash/No Flash
  - Joint Bilateral Upsample
  - ASTA

This thing is awesome.

Composing Filters

- F is a bad gradient filter
- It's cheap to evaluate
  - val = Im(x+5, y) – Im(x-5, y)

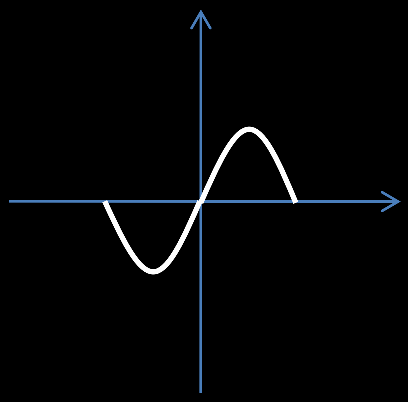
F=



- G is a good gradient filter ->
- It's expensive to evaluate
  - for (dx=-10; dx<10; dx++)

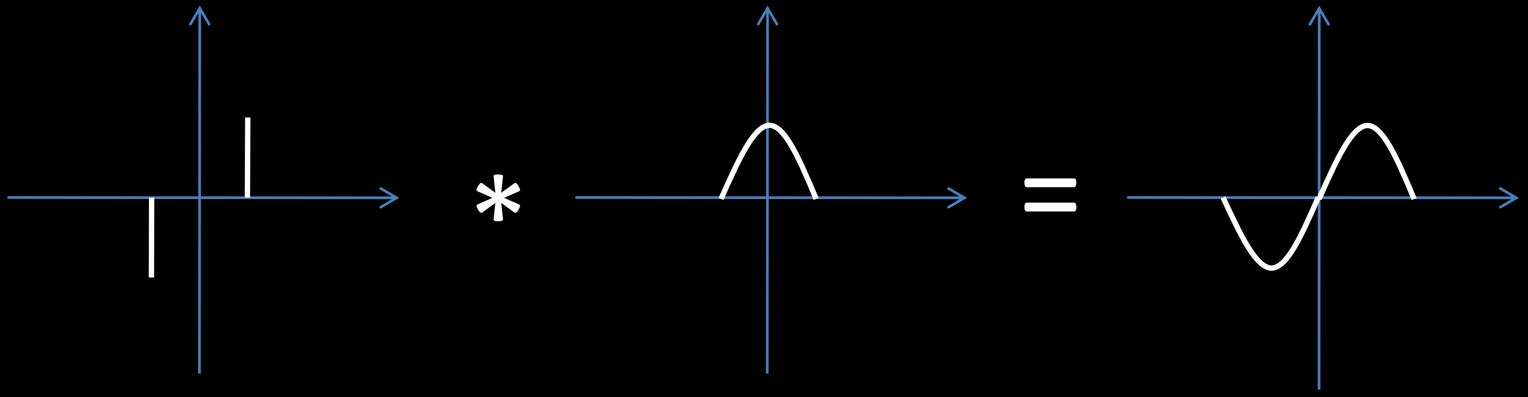
    val += filter(dx)*Im(x+dx, y)

G=



Composing Filters

- But F * B = G



- and convolution is **associative**
- so: G*Im = (F*B)*Im = F*(B*Im)

Composing Filters

- So if you need to take lots of good filters:
- Blur the image nicely once $Im_2 = (B*Im)$
- Use super simple filters for everything else

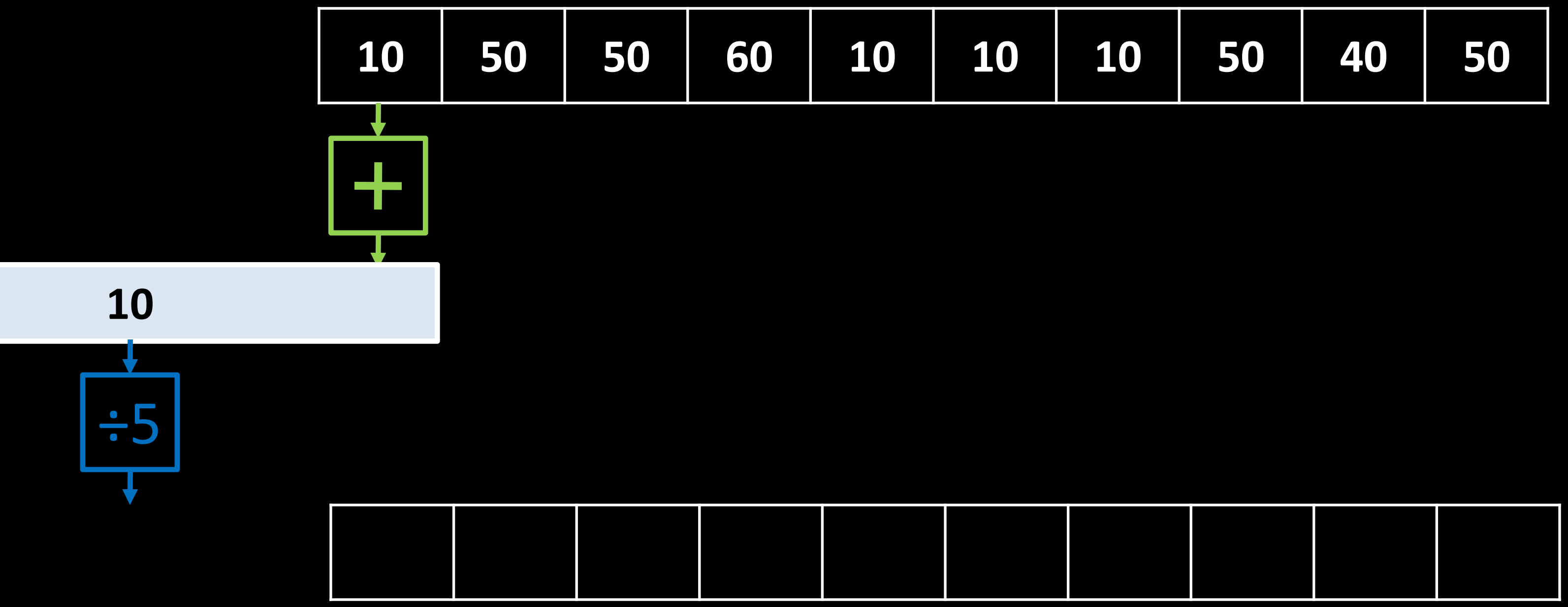
- $F_1 * Im_2 \quad F_2 * Im_2 \quad F_3 * Im_2 \quad \ldots$

- You only performed one expensive filter (B)
- Let's make the expensive part as fast as possible

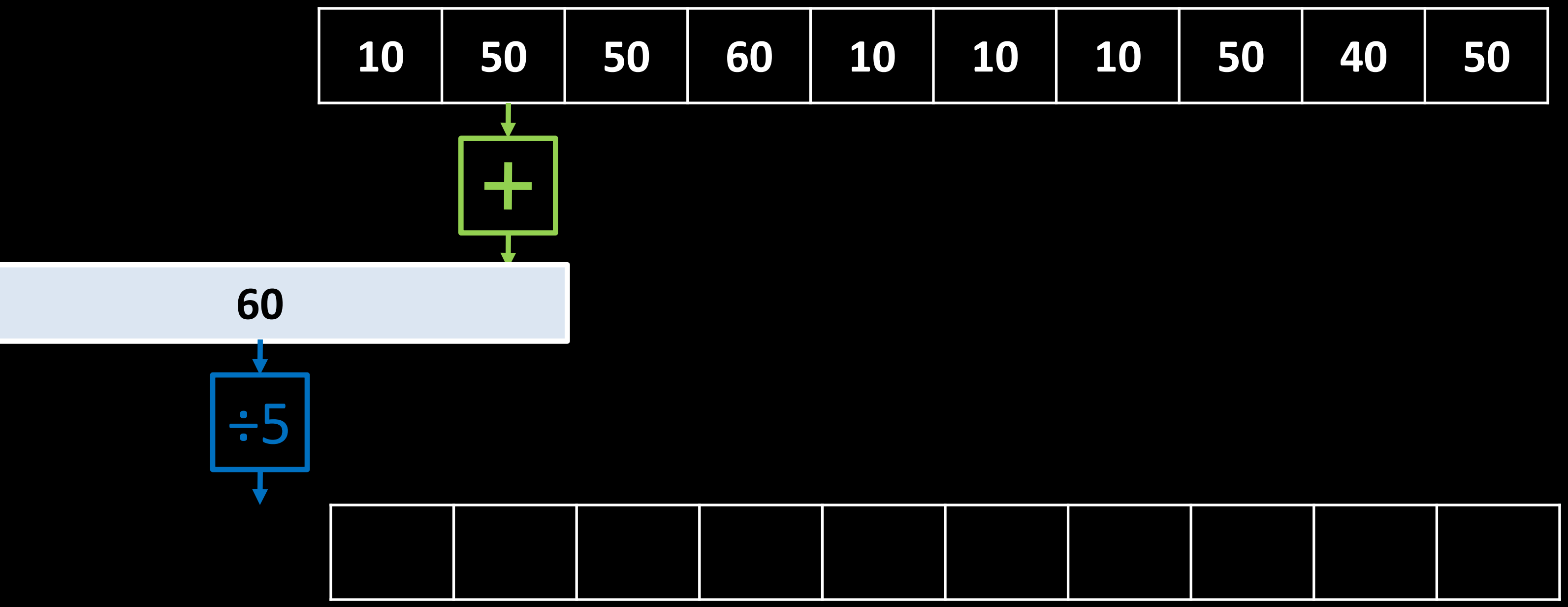
Fast Rect Filters

- Suggestions?

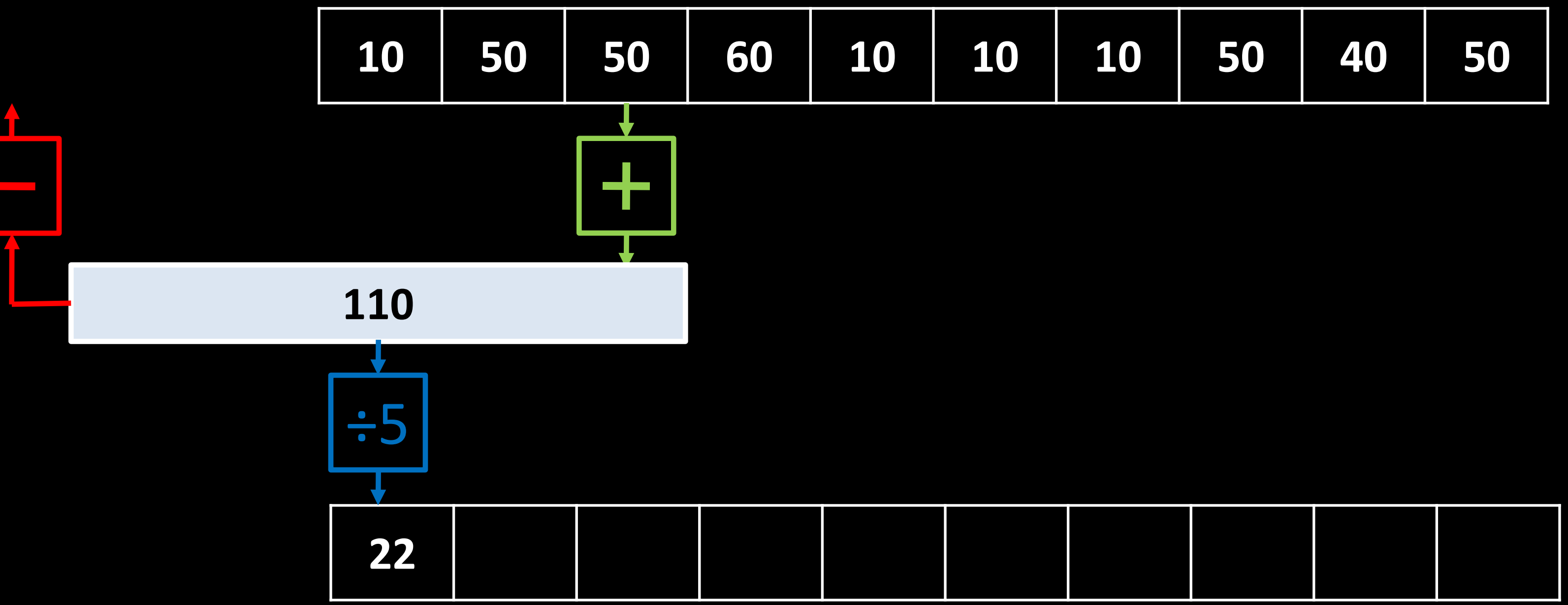
Fast Rect Filters



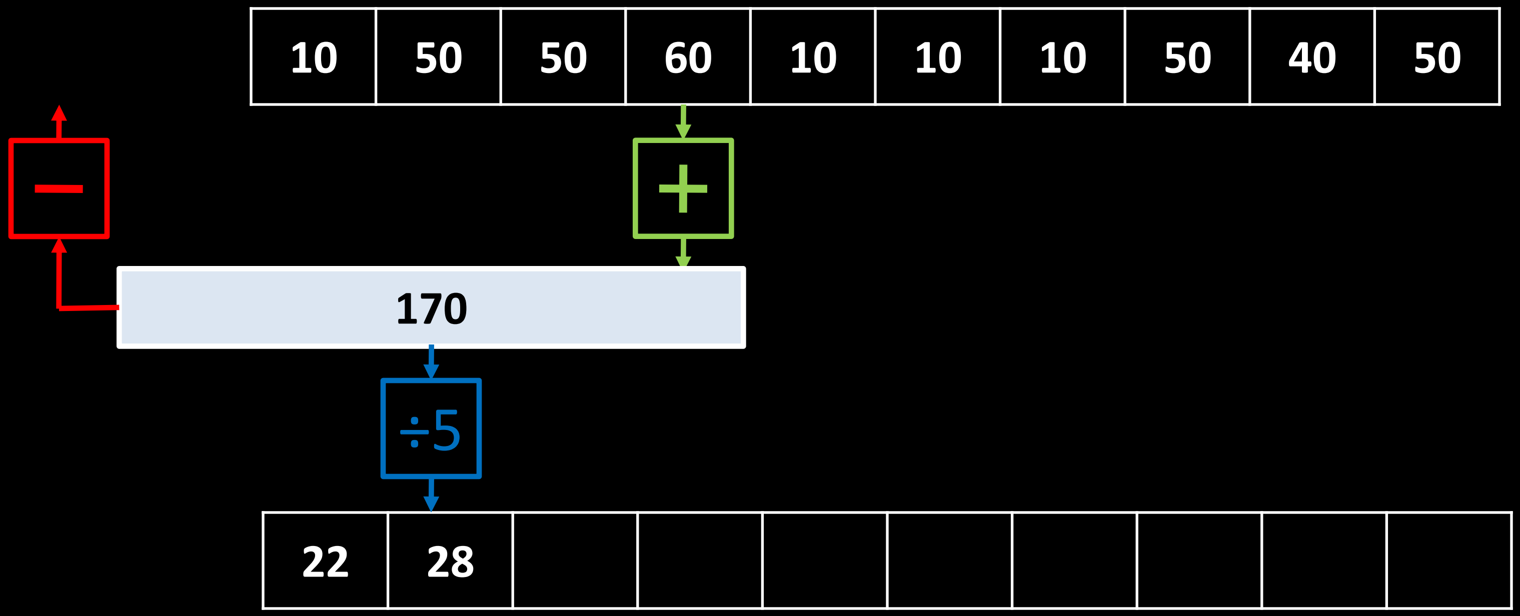
Fast Rect Filters



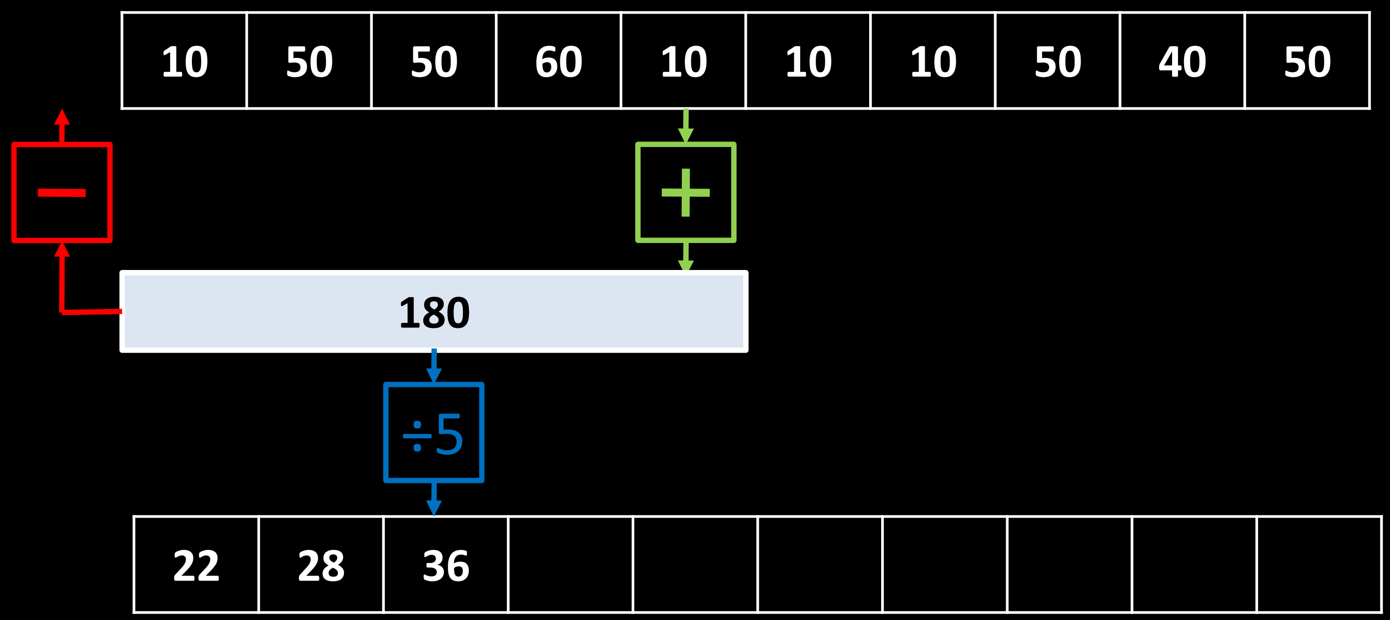
Fast Rect Filters



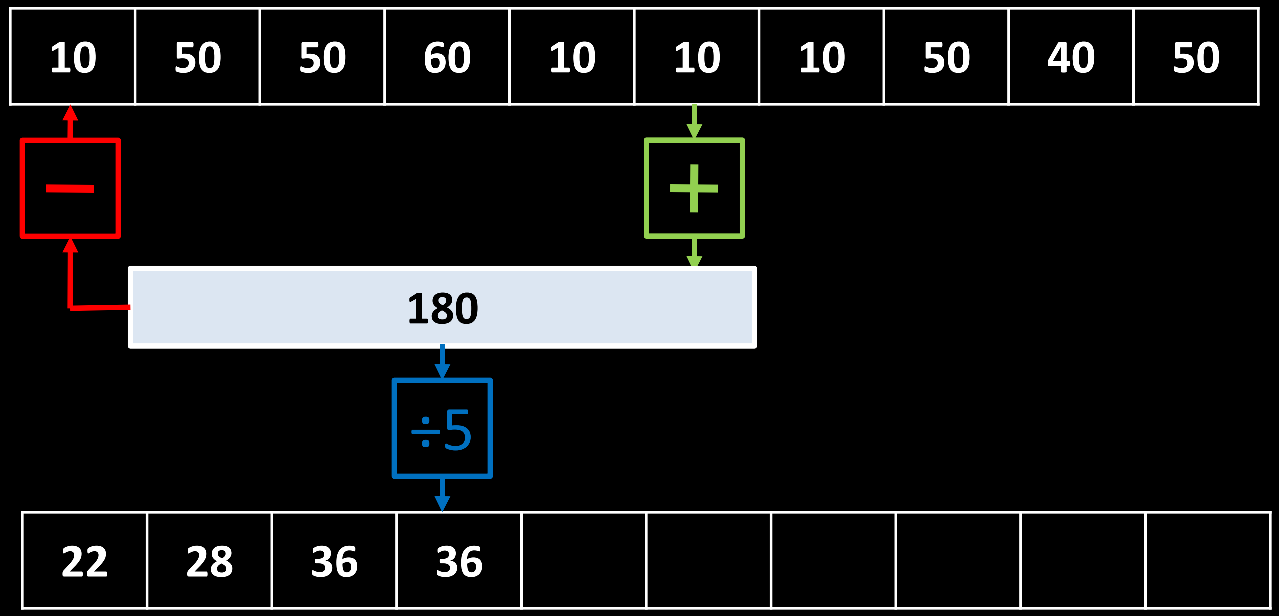
Fast Rect Filters



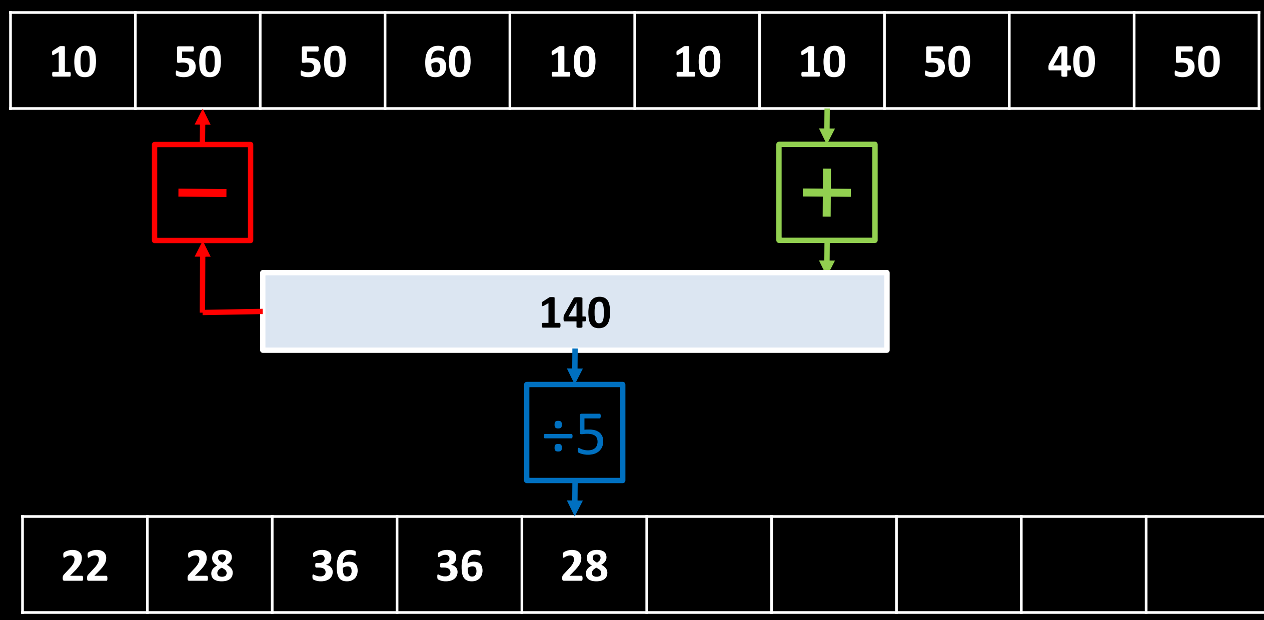
Fast Rect Filters



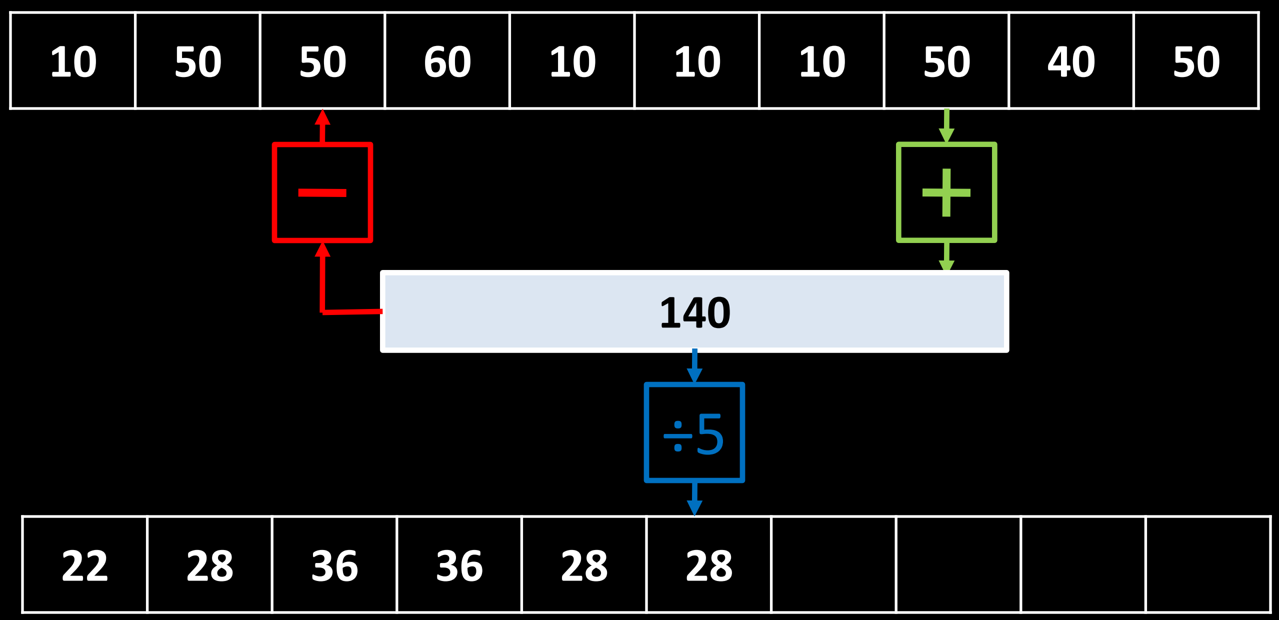
Fast Rect Filters



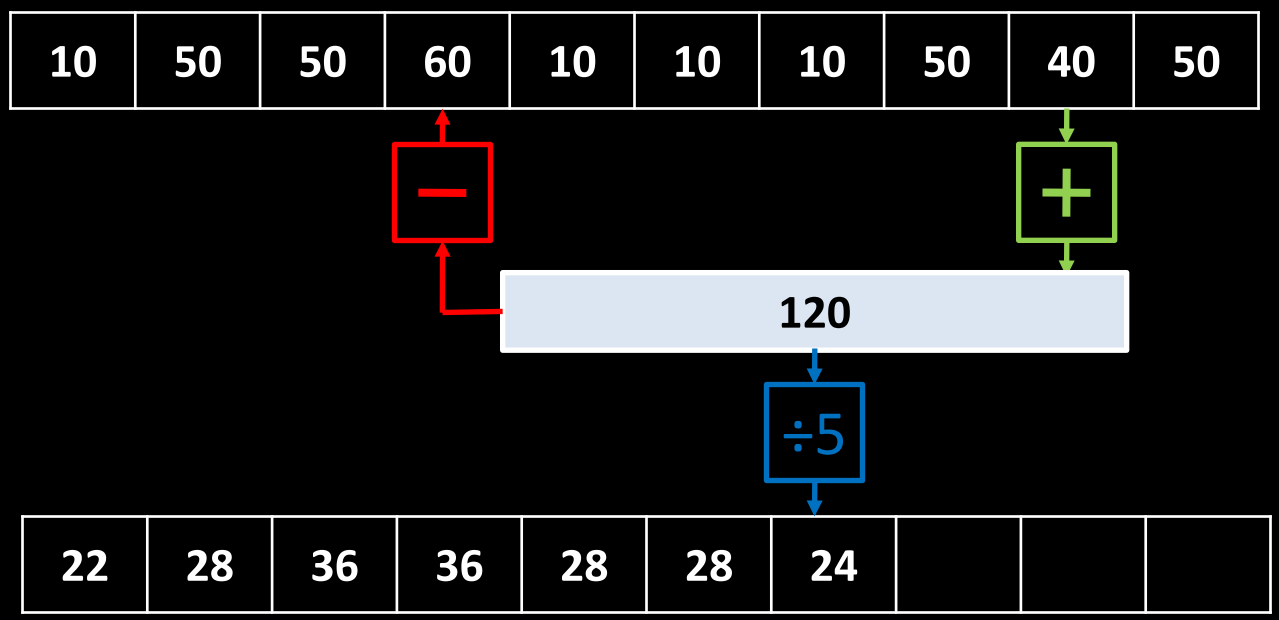
Fast Rect Filters



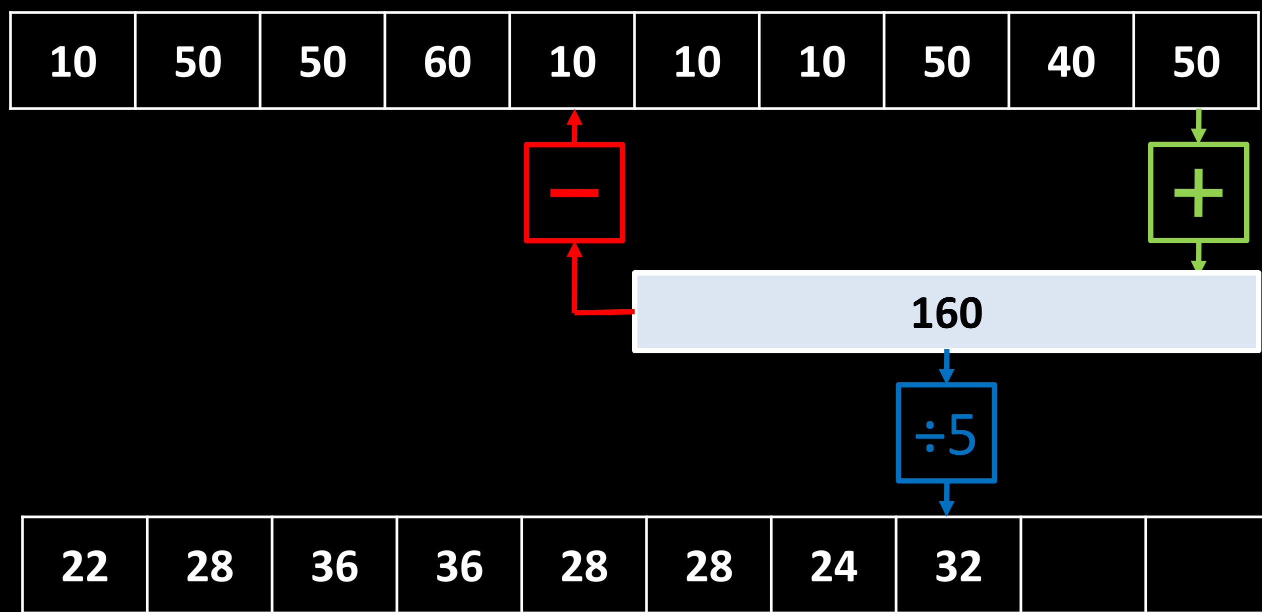
Fast Rect Filters



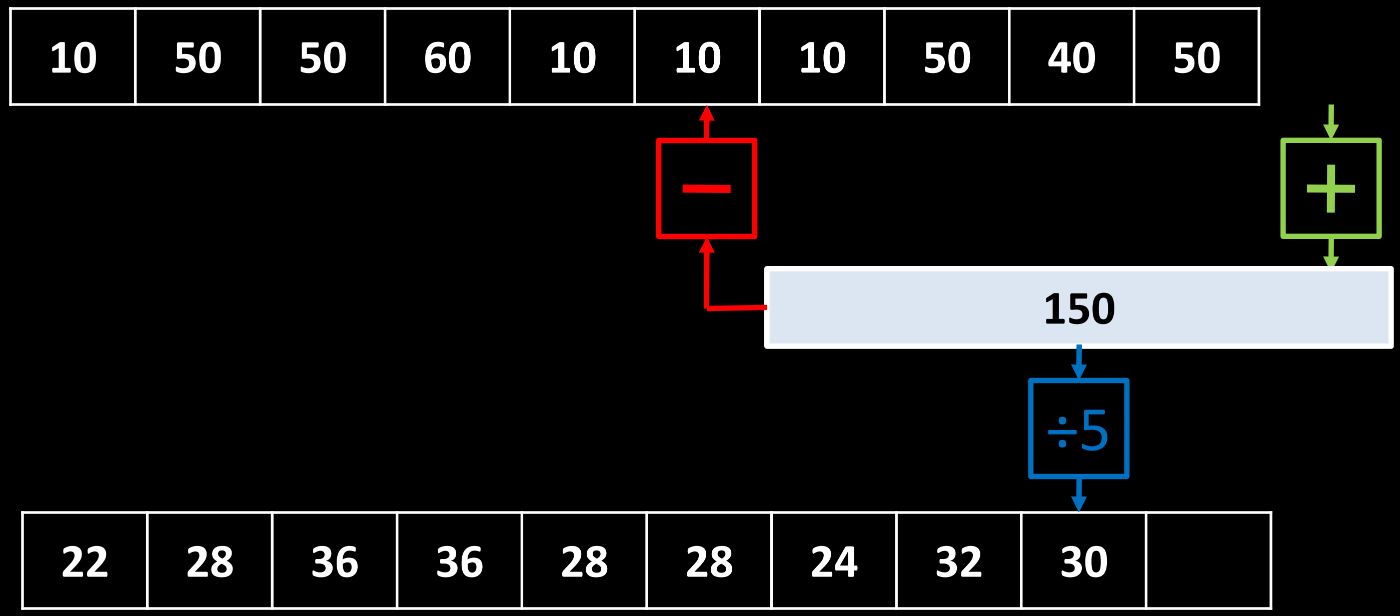
Fast Rect Filters



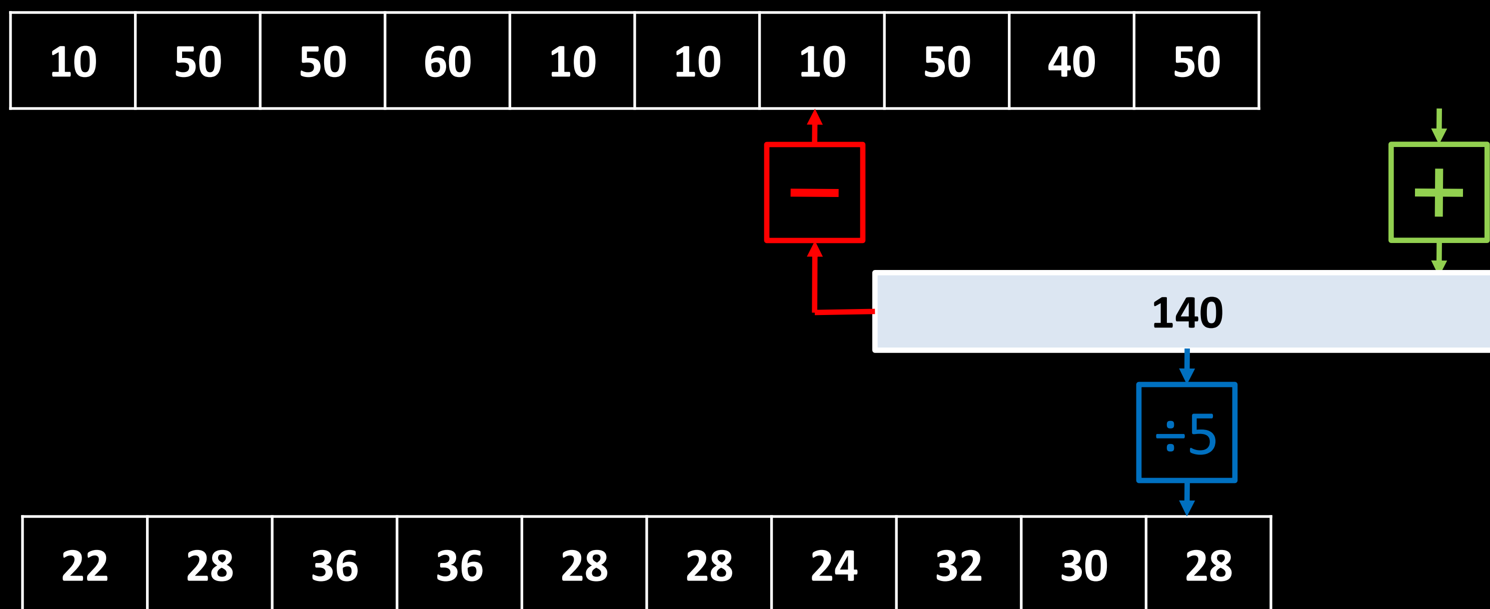
Fast Rect Filters



Fast Rect Filters



Fast Rect Filters



Fast Rect Filters

- Complexity?
  - Horizontal pass: $O((w+f)h) = O(wh)$
  - Vertical pass: $O((h+f)w) = O(wh)$
  - Total: $O(wh)$
- Precision can be an issue

0	0	5^{10}	5	0	0	0	0	0	0

5^9	5^9	5^9	5^9	5^9	0	-1	-1	-1	-1

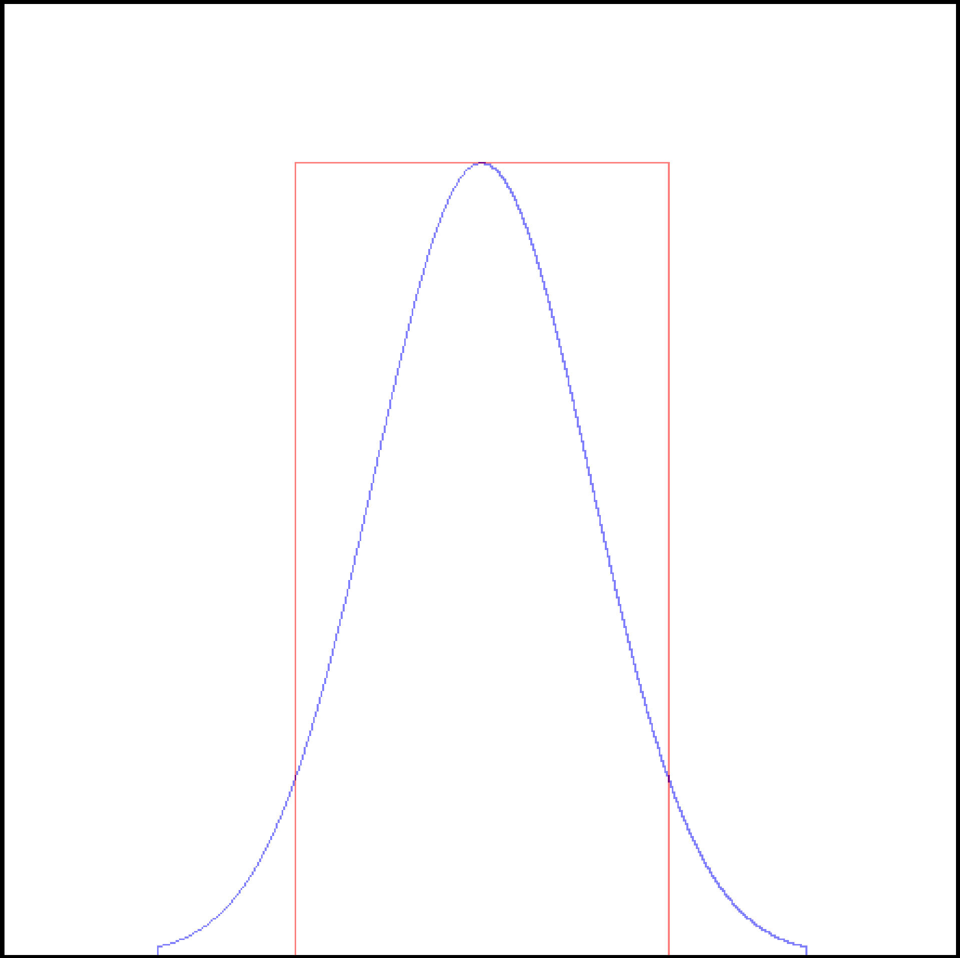
Fast Rect Filters

- How can I do this in-place?

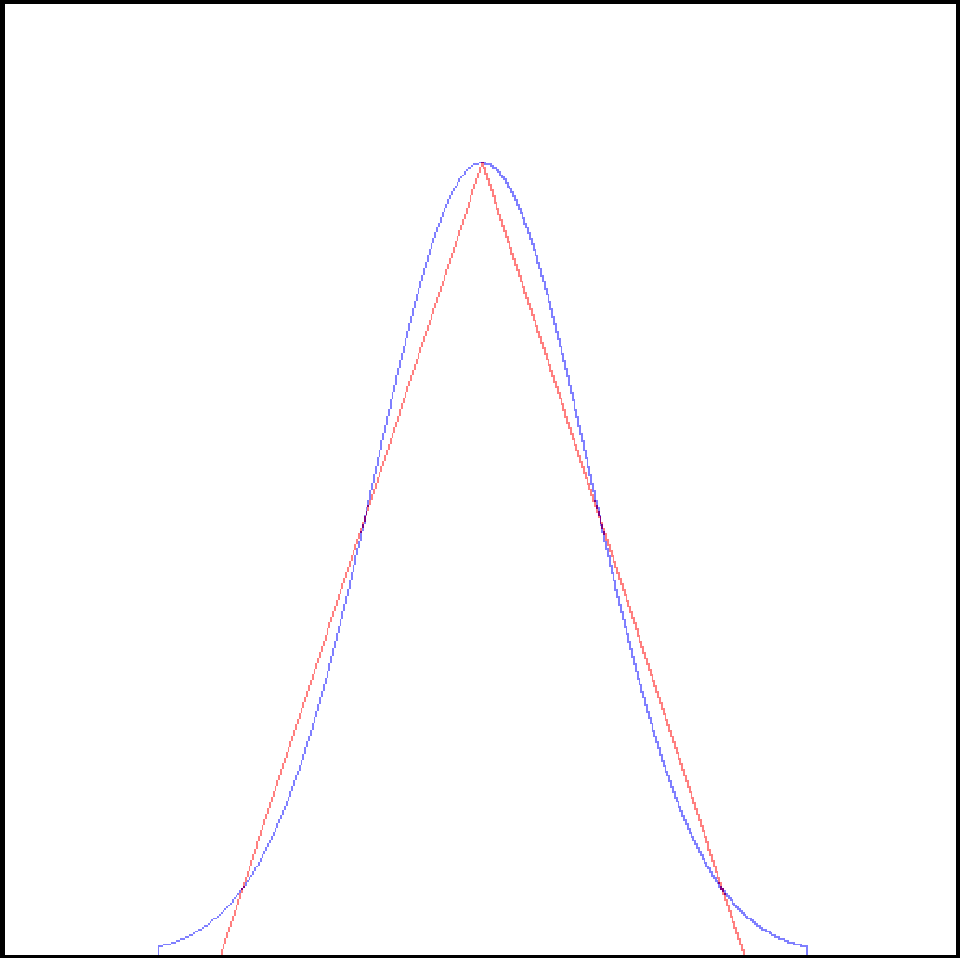
Gaussian Filters

- How can we extend this to Gaussian filters?
- Common approach:
  - Take FFT **O(w h ln(w) ln(h))**
  - Multiply by FFT of Gaussian **O(wh)**
  - Take inverse FFT **O(w h ln(w) ln(h))**
  - Total cost: **O(w ln(w) h ln(h))**
- Cost independent of filter size ☺
- Not particularly cache coherent ☹

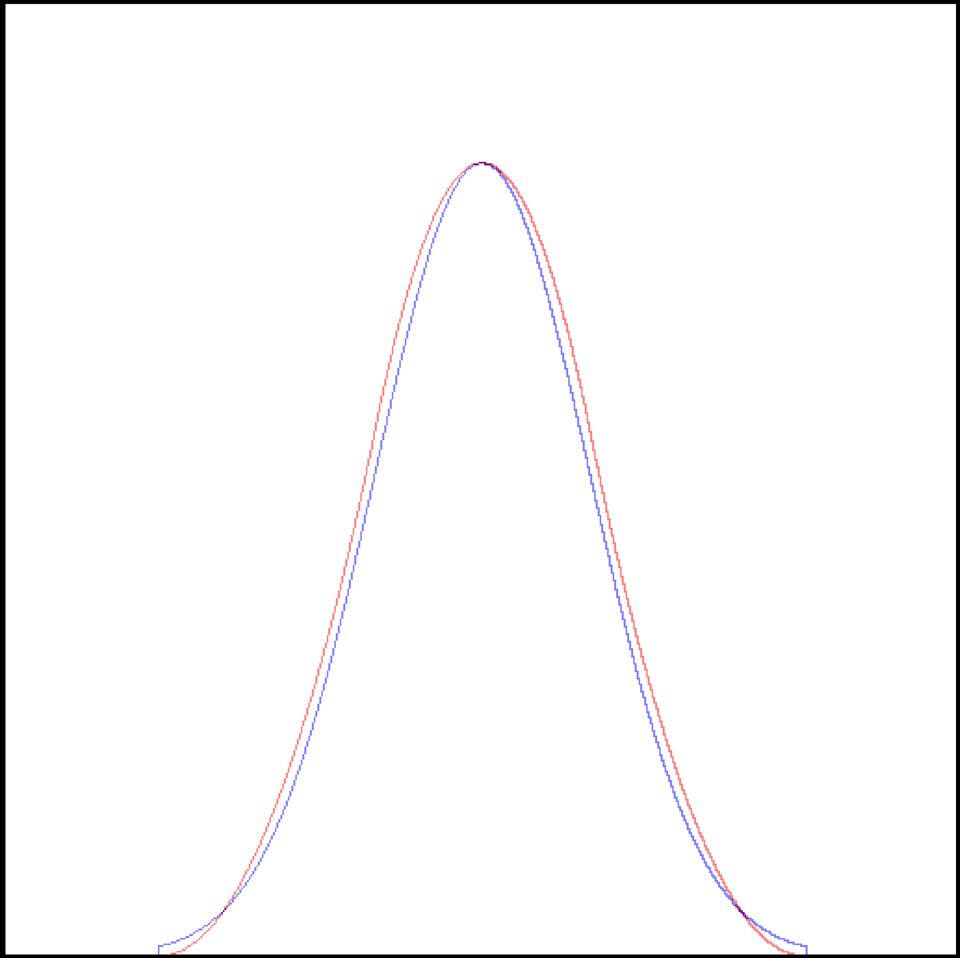
Gaussian v Rect



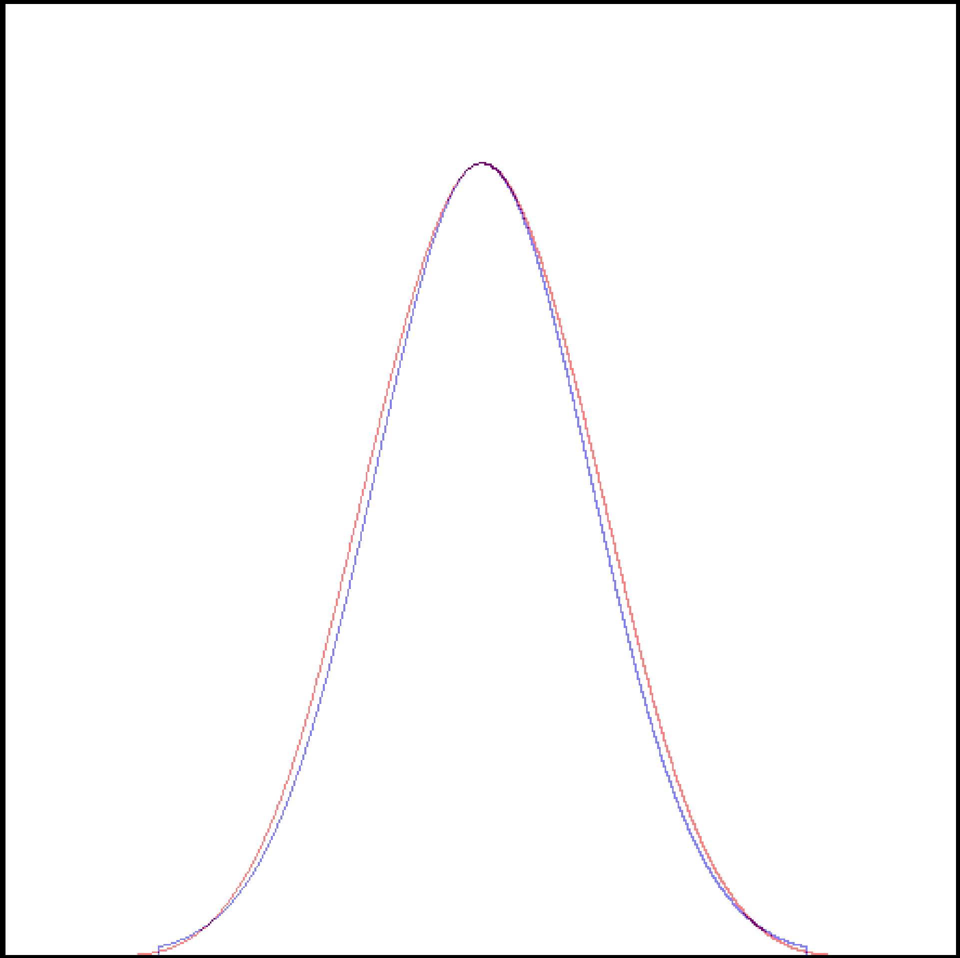
Gaussian v Rect*Rect



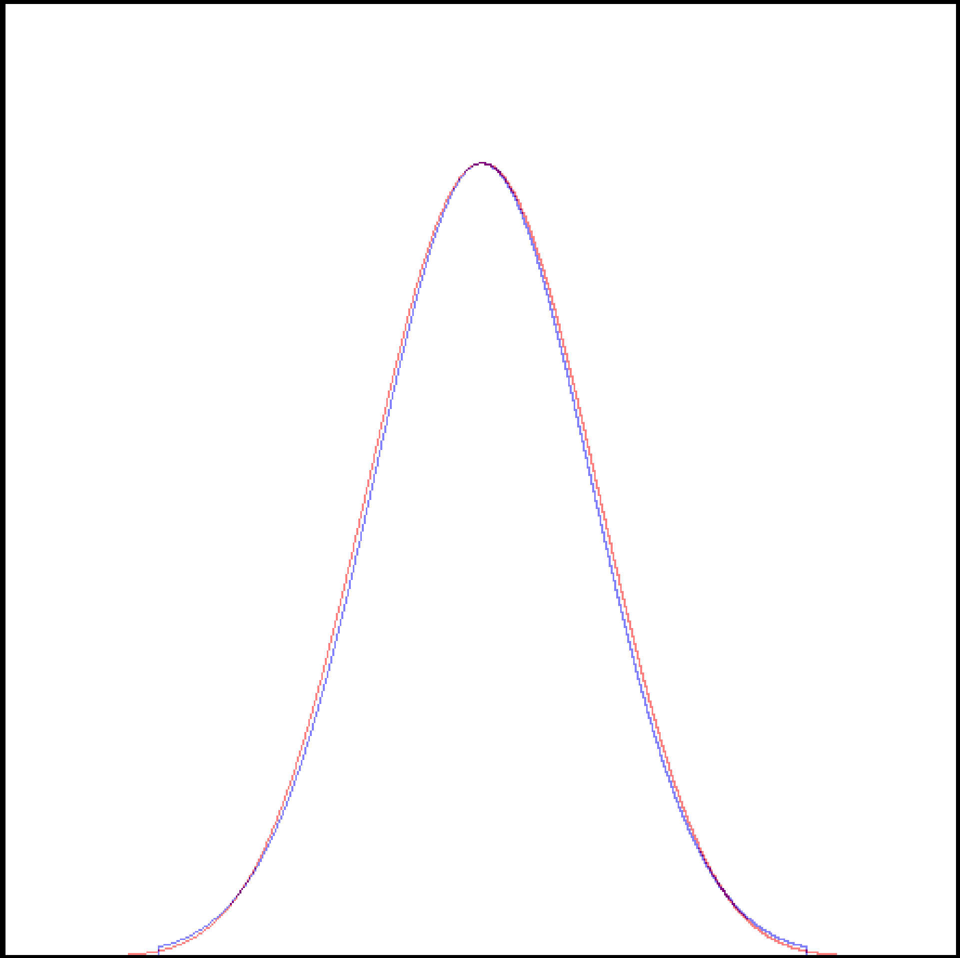
Gaussian v $Rect^3$



Gaussian v Rect^4



Gaussian v $Rect^5$



Gaussian

Rect (RMS = 0.00983)

Gaussian

$Rect^2$ (RMS = 0.00244)

Gaussian

$Rect^3$ (RMS = 0.00173)

Gaussian

Rect^4 (RMS = 0.00176)

Gaussian

Rect^5 (RMS = 0.00140)

Gaussian Filters

- Conclusion: Just do 3 rect filters instead
- Cost: O(wh)
- Cost independent of filter size ☺
- More cache coherent ☺
- Be careful of edge conditions ☹
- Hard to construct the right filter sizes: ☹

Filter sizes

- Think of convolution as randomly scattering your data around nearby
- How far data is scattered is described by the **standard deviation** of the distribution
- standard deviation = sqrt(variance)
- Variance adds
  - Performing a filter with variance v twice produces a filter with variance 2v

Filter sizes

- Think of convolution as randomly scattering your data around nearby
- How far data is scattered is described by the **standard deviation** of the distribution
- standard deviation = sqrt(variance)
- Variance adds
  - Performing a filter with variance v twice produces a filter with variance 2v

Filter Sizes

- Variance adds
  - Performing a filter with variance v twice produces a filter with variance 2v
- Standard deviation scales
  - A filter with standard deviation s, when scaled to be twice as wide, has standard deviation 2s

Constructing a Gaussian out of Rects

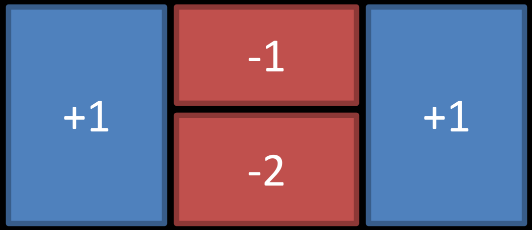
- A rect filter of width 2w+1 has variance: w(w+1)/3
- Attainable standard deviations using a single rect [sqrt(w(w+1)/3)]:
  - 0.82 1.41 2 2.58 3.16 3.74 4.32 ...
- Composing three identical rects of width 2w+1 has variance: w(w+1)
- Attainable std devs [sqrt(w(w+1))]:
  - 1.41 2.45 3.46 4.47 5.48 6.48 7.48 ... 1632.5

Constructing a Gaussian out of Rects

- Attainable standard deviations using three different odd rect filters:
  - 1.41 1.825 2.16 2.31 2.45 2.58 2.83 2.94 3.06 ... 8.25 8.29 8.32 8.37 8.41 8.45 8.48
- BUT: if they're too different, the result won't look Gaussian
- Viable approach: Get as close as possible with 3 identical rects, do the rest with a small Gaussian

Integral Images

- Fast rects are good for filtering an image...
- But what if we need to compute lots of filters of different shapes and sizes quickly?



- Classifiers need to do this

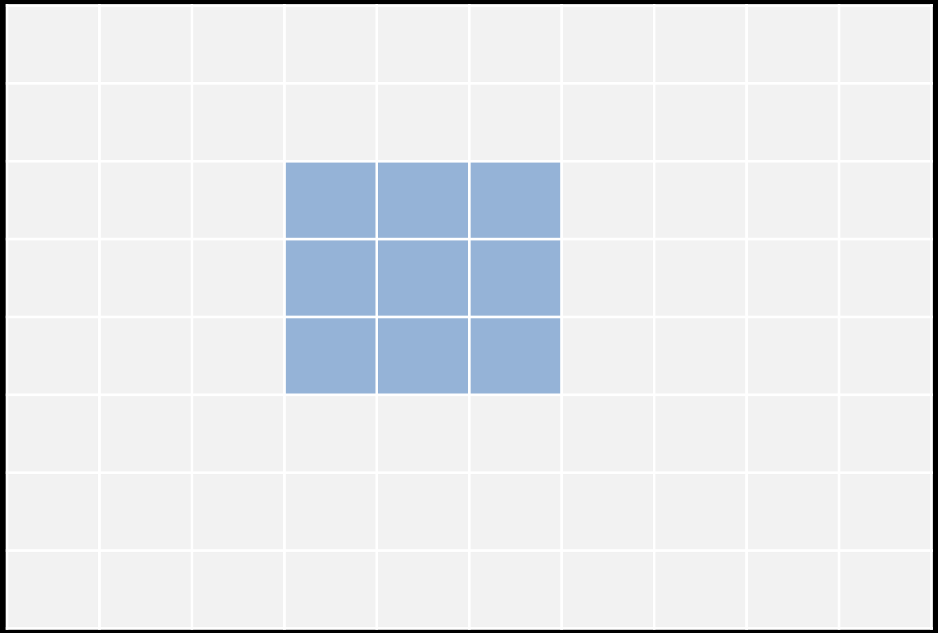
Integrate the Image ahead of time

$$Integral(x,y) = \sum_{u,v=(0,0)}^{(x,y)} Input(u,v)$$

- Each pixel is the sum of everything above and left
- ImageStack -load dog1.jpg -integrate x
  -integrate y

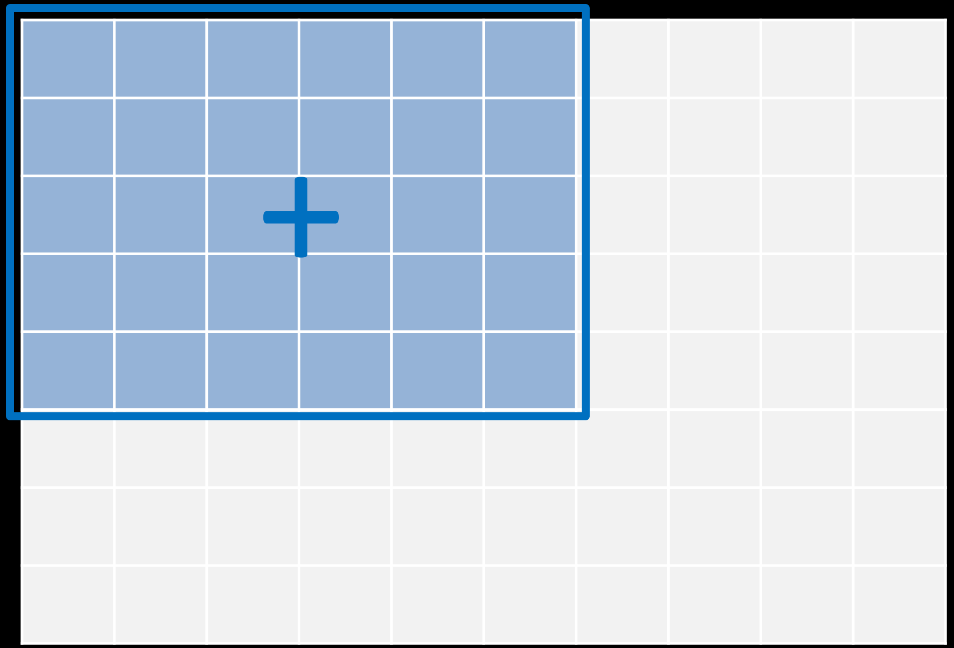
Integral Images

- Fast to compute (just run along each row and column adding up)
- Allows for arbitrary sized rect filters



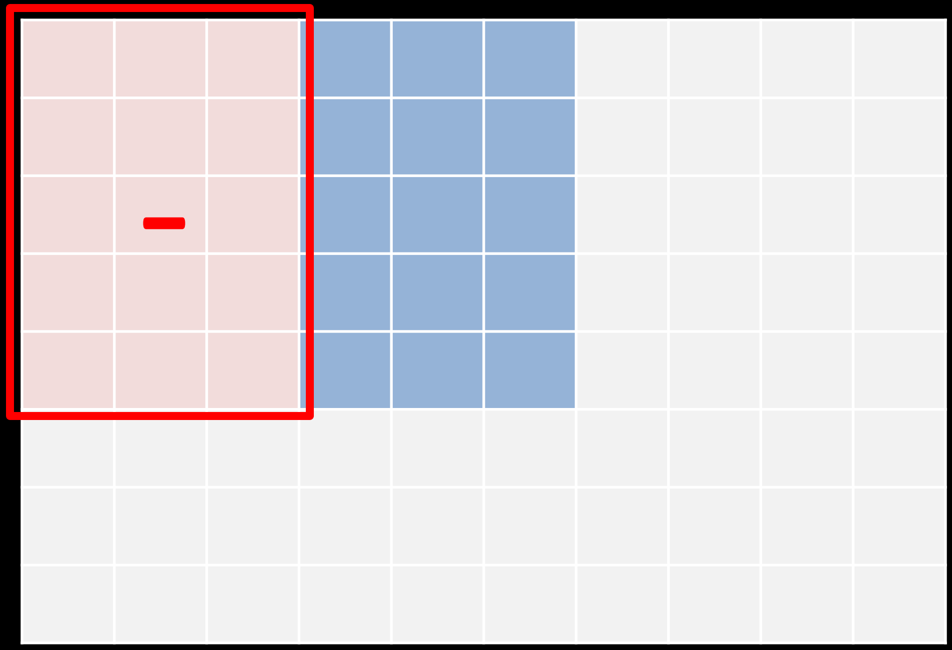
Integral Images

- Fast to compute (just run along each row and column adding up)
- Allows for arbitrary sized rect filters



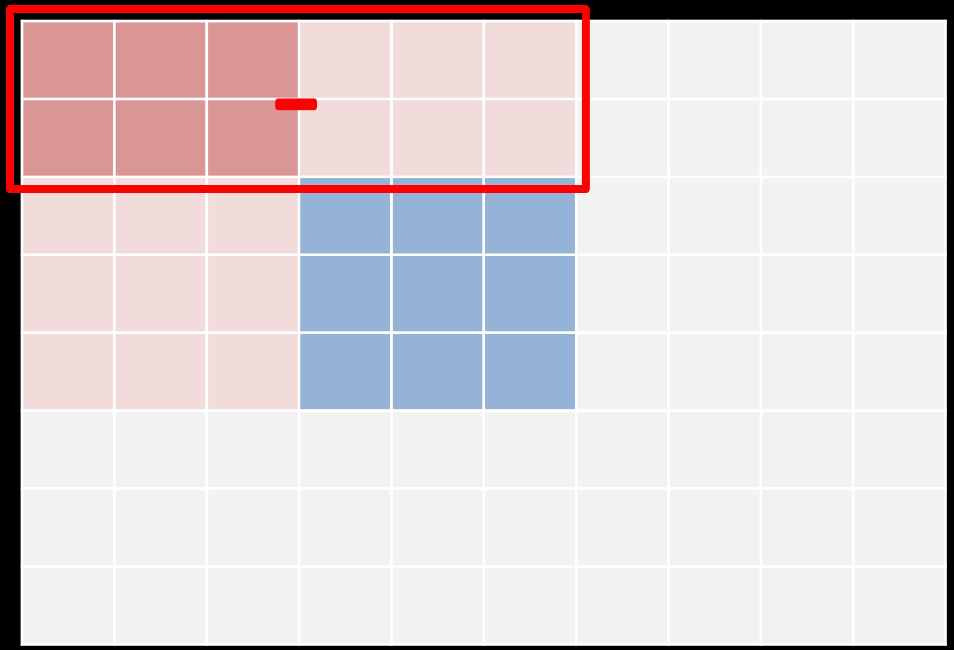
Integral Images

- Fast to compute (just run along each row and column adding up)
- Allows for arbitrary sized rect filters



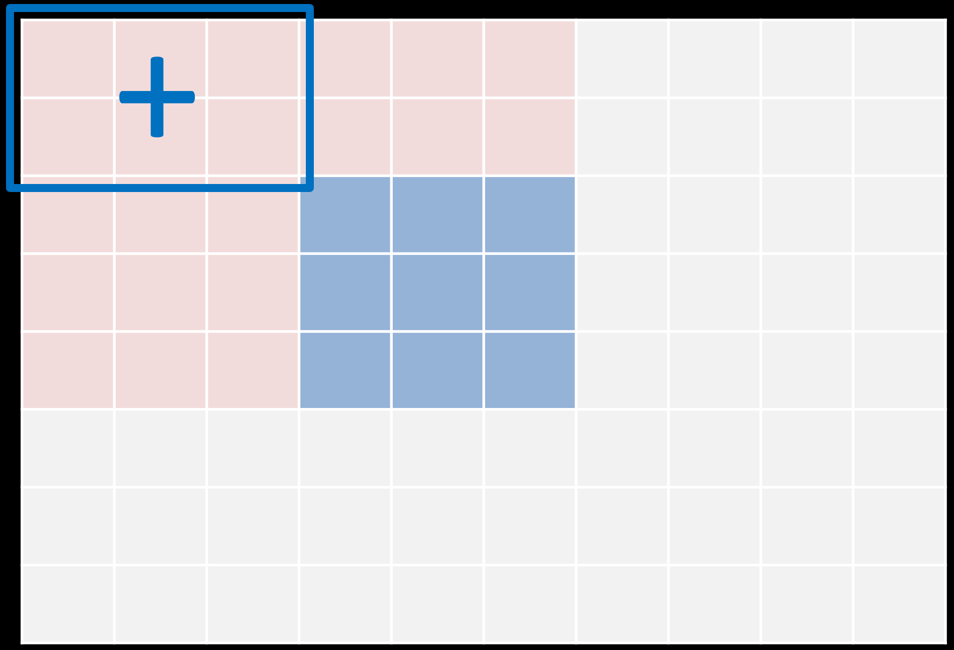
Integral Images

- Fast to compute (just run along each row and column adding up)
- Allows for arbitrary sized rect filters



Integral Images

- Fast to compute (just run along each row and column adding up)
- Allows for arbitrary sized rect filters



Integral Images

- Fast sampling of arbitrary rects
- Precision can be an issue
- Can only be used for rect filters...

Higher Order Integral Images

- Fast sampling of arbitrary polynomials

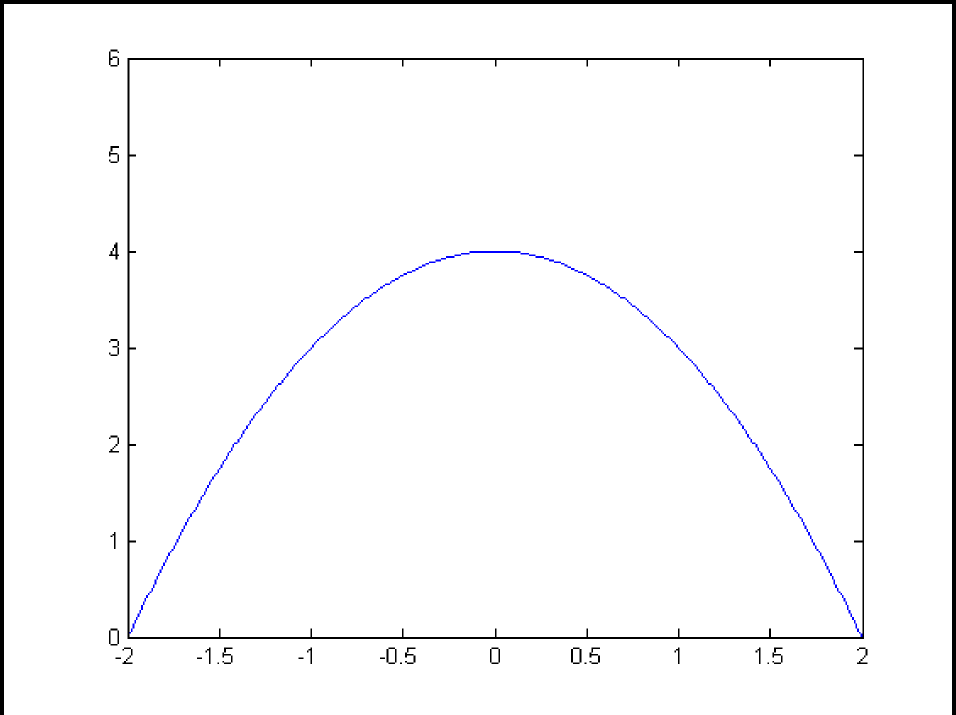
$$Integral_0(x) = \sum_{u=0}^{x} Input(u)$$

$$Integral_1(x) = \sum_{u=0}^{x} Input(u).u$$

$$Integral_2(x) = \sum_{u=0}^{x} Input(u).u^2$$

Higher Order Integral Images

- Let's say we want to evaluate a filter shaped like $(4\text{-}x^2)$ centered around each pixel



Higher Order Integral Images

$$Out(x) = \sum_{u=x-2}^{x+2} I(u)(4-(u-x)^2)$$

$$O(x) = (4-x^2)\sum_{u=x-2}^{x+2} I(u)$$

$$+\,2x\sum_{u=x-2}^{x+2} uI(u)$$

$$+\sum_{u=x-2}^{x+2} u^2 I(u)$$

Higher Order Integral Images

We can compute each term using the integral images of various orders.

No summations over u required.

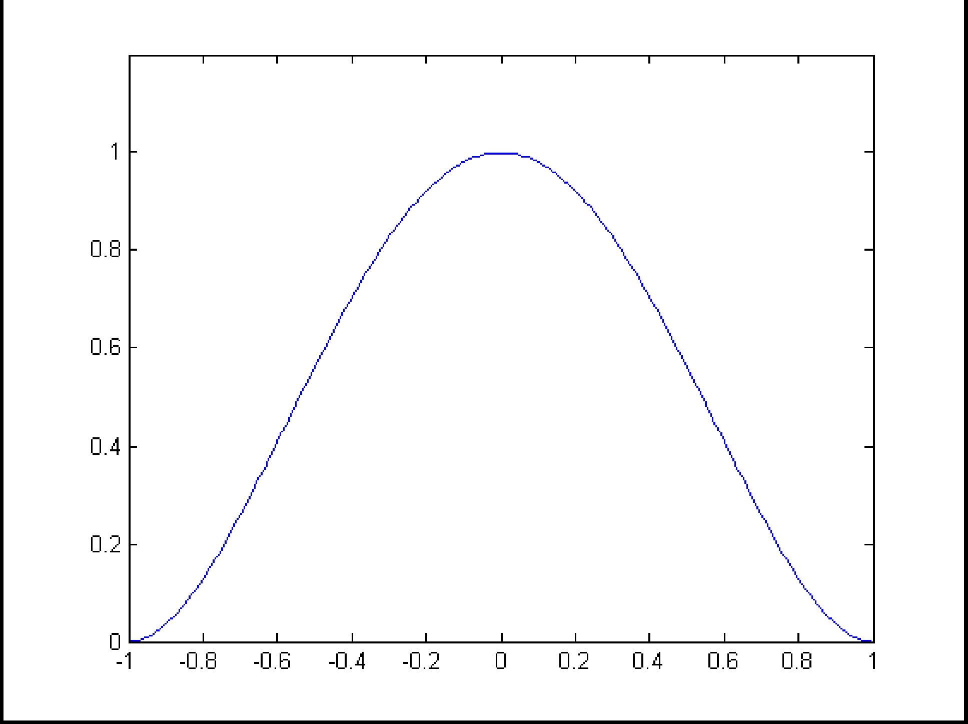
$$O(x) = (4 - x^2) \sum_{u=x-2}^{x+2} I(u)$$

$$+ 2x \sum_{u=x-2}^{x+2} uI(u)$$

$$+ \sum_{u=x-2}^{x+2} u^2 I(u)$$

Gaussians using Higher Order Integral Images

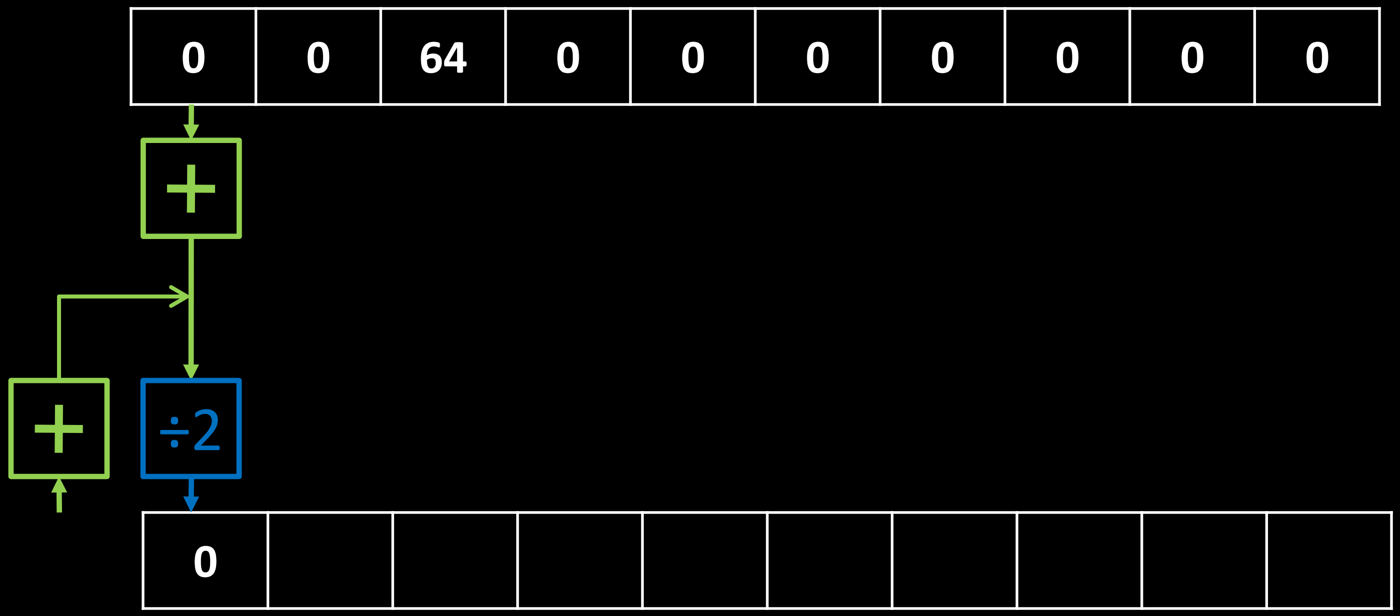
- Construct a polynomial that looks kinda like a Gaussian, e.g. $(x-1)^2(x+1)^2$



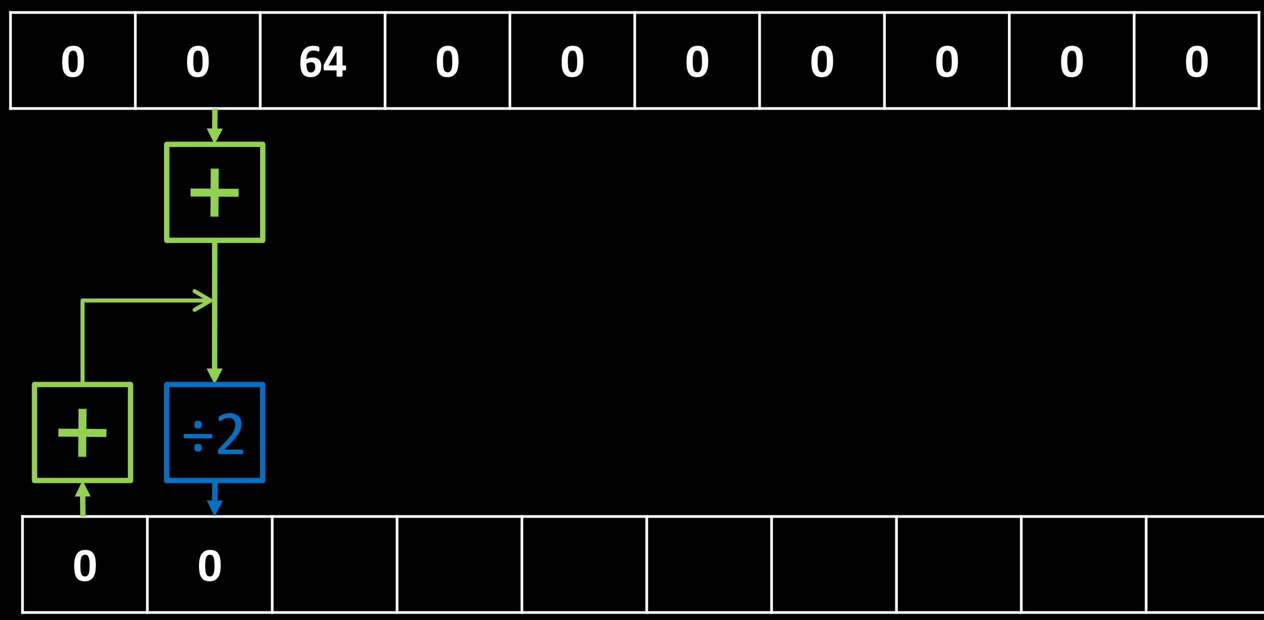
IIR Filters

- We can also use feedback loops to create Gaussians...

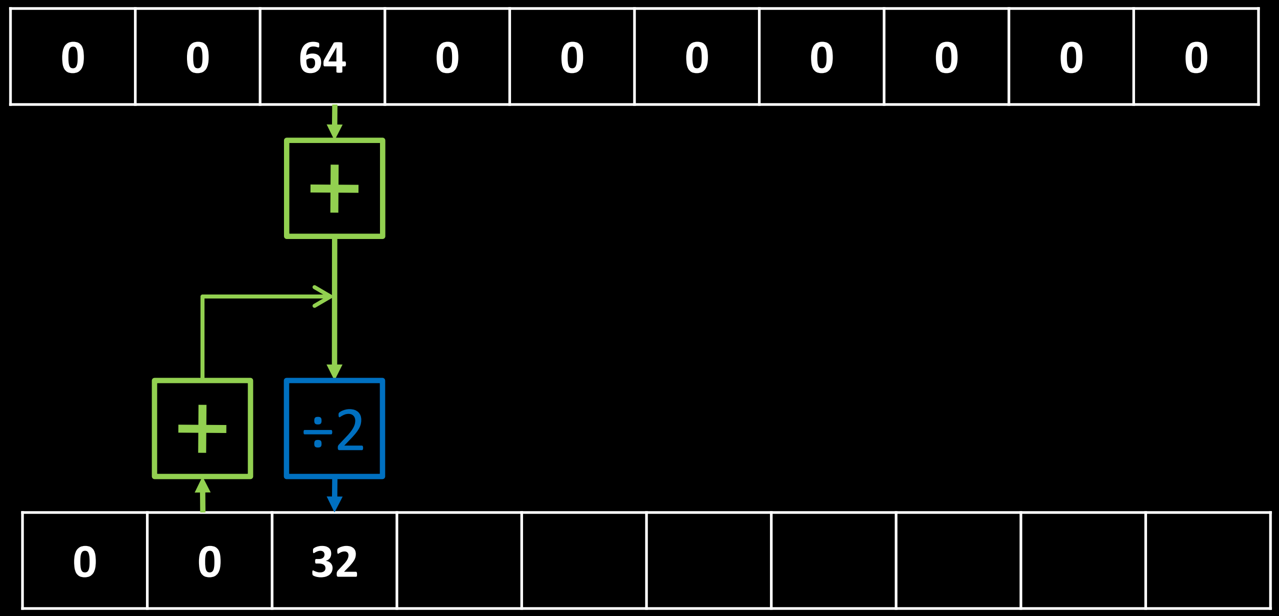
IIR Filters



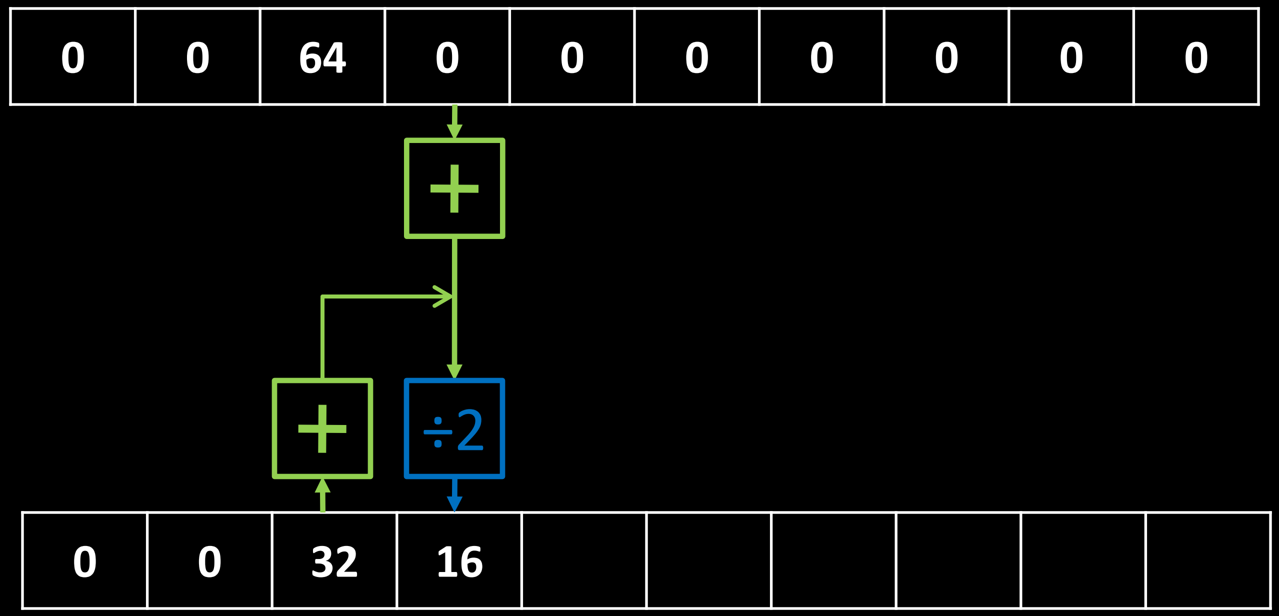
IIR Filters



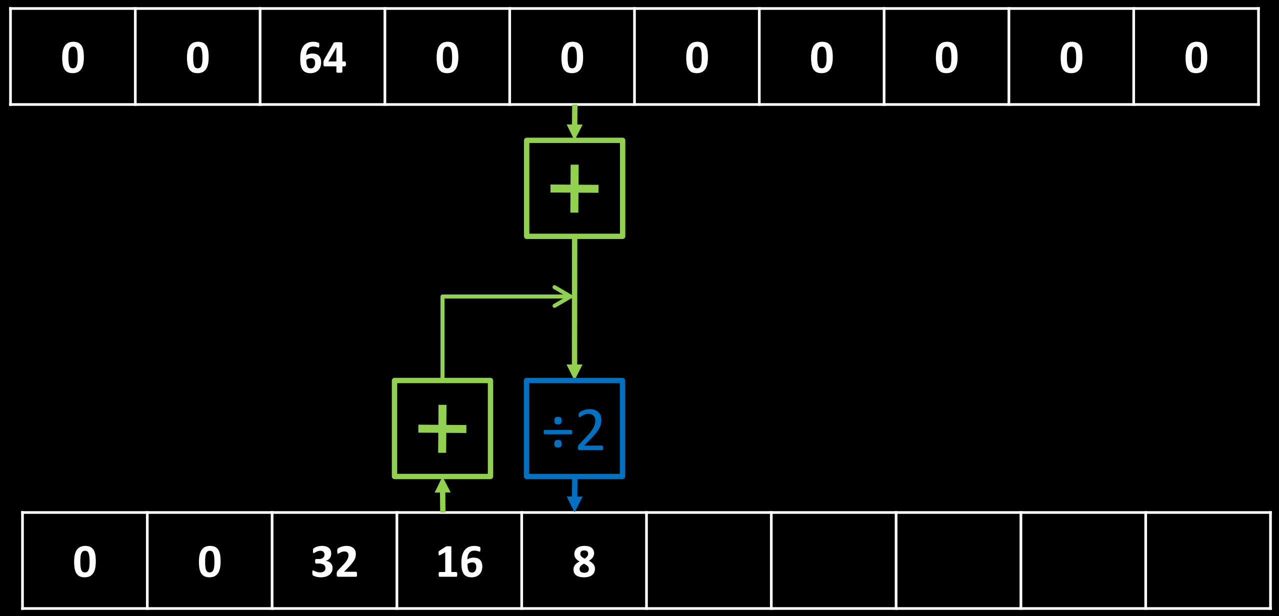
IIR Filters



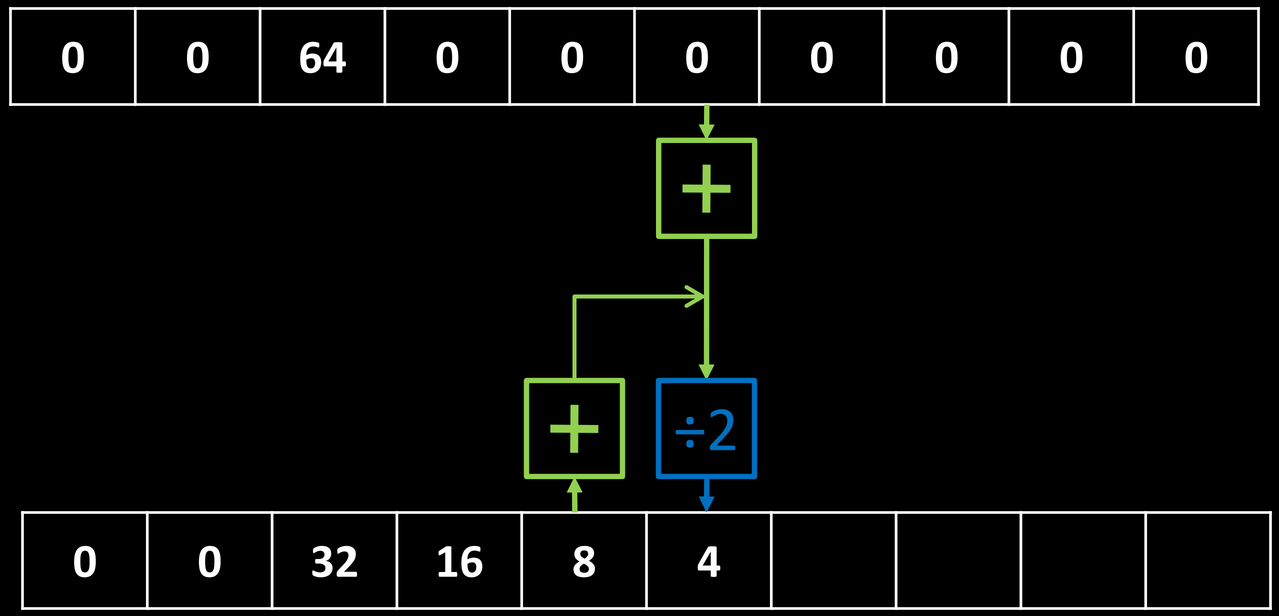
IIR Filters



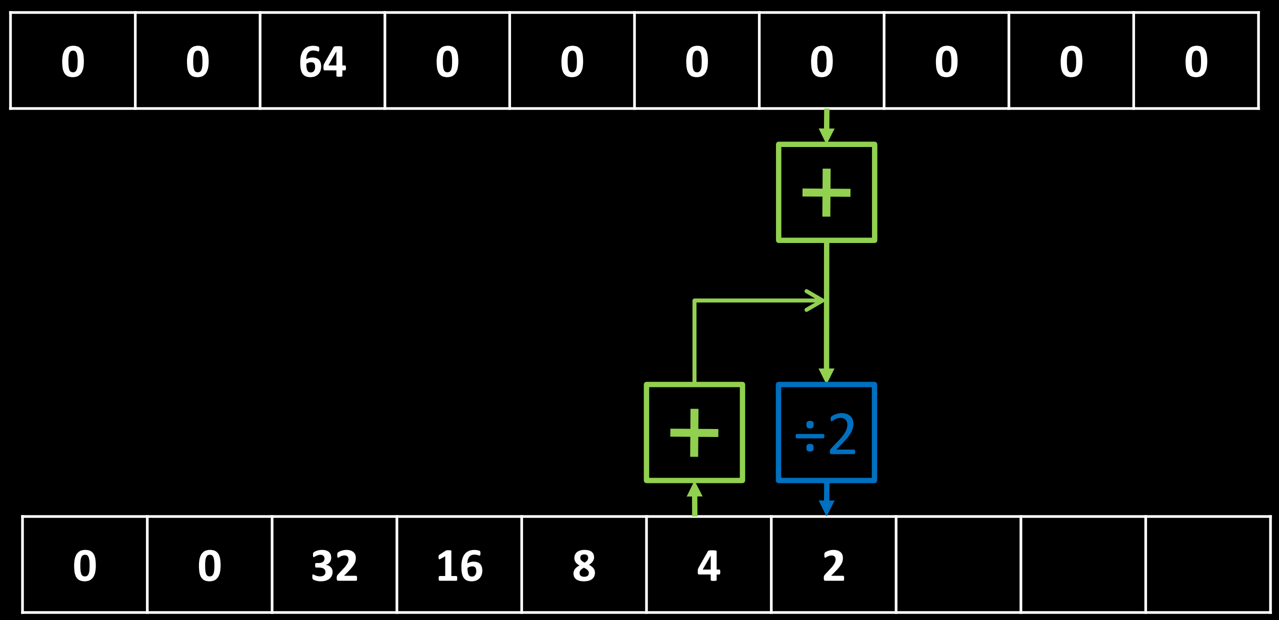
IIR Filters



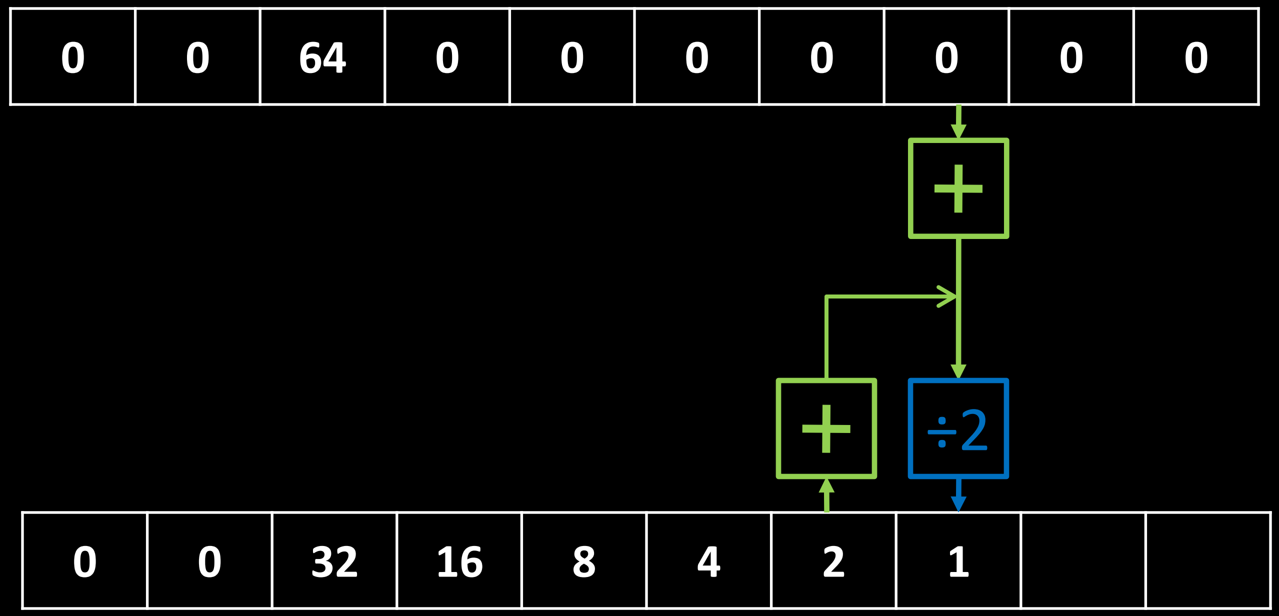
IIR Filters



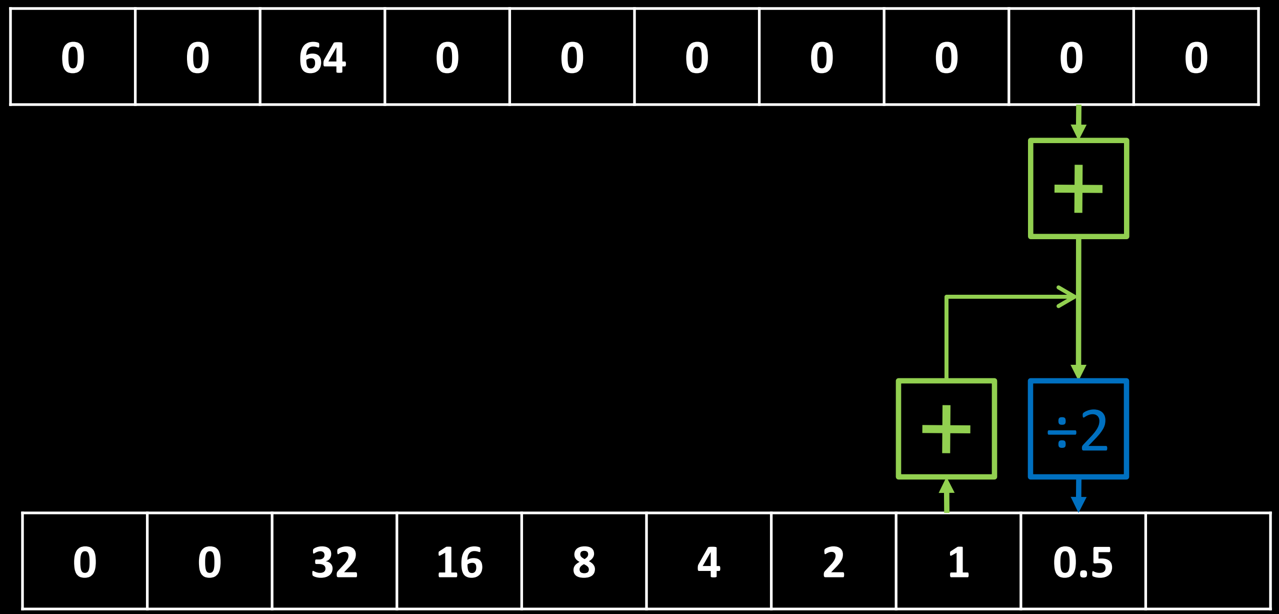
IIR Filters



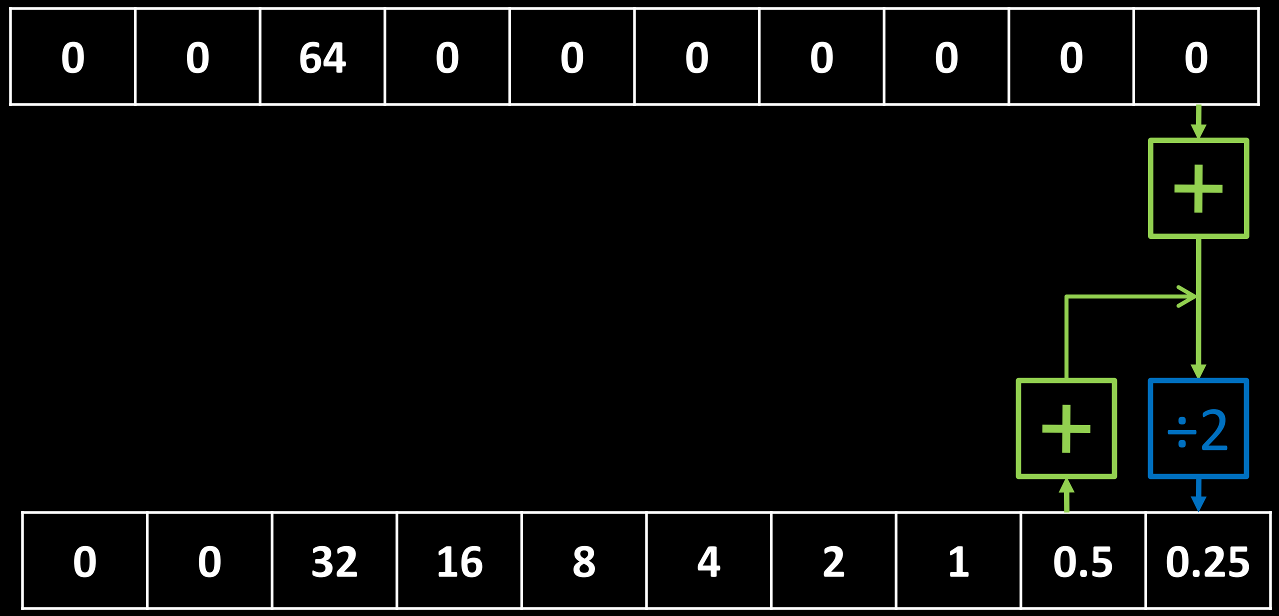
IIR Filters



IIR Filters

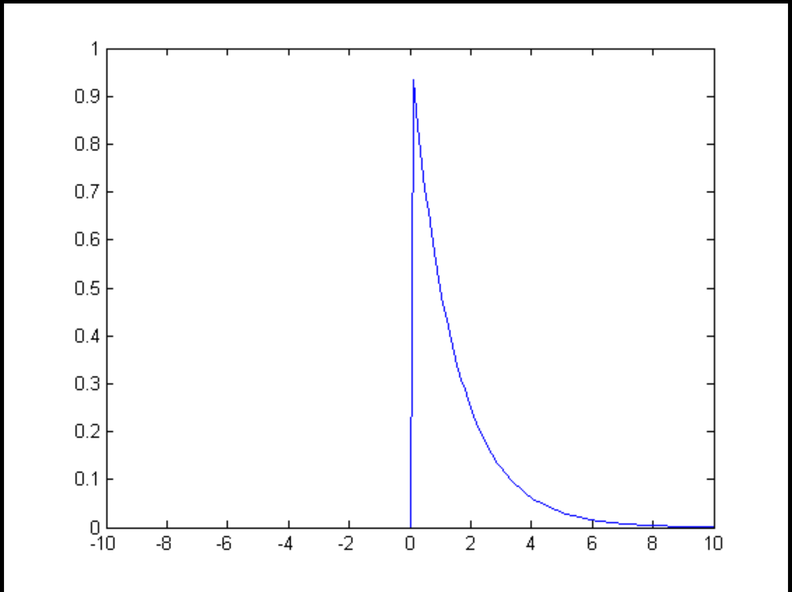


IIR Filters



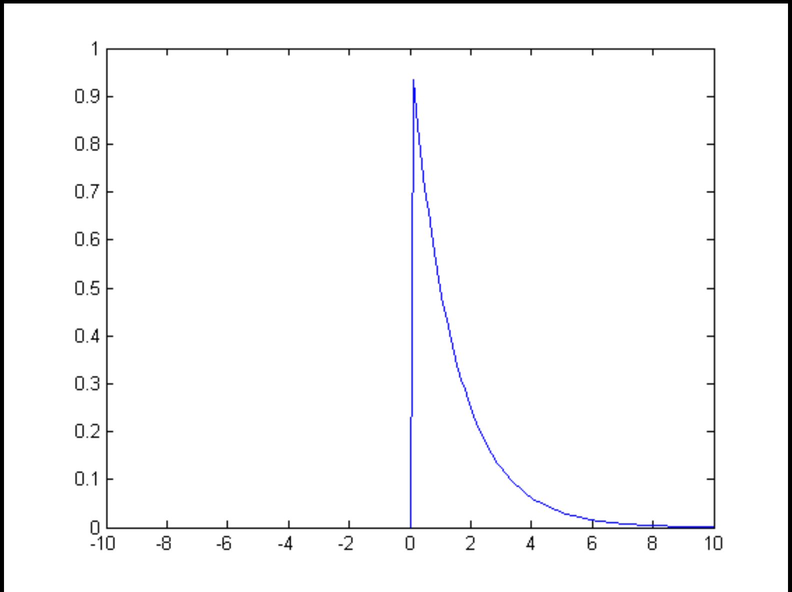
IIR = Infinite Impulse Response

- A single spike has an effect that continues forever
- The example above was an exponential decay
- Equivalent to convolution by:



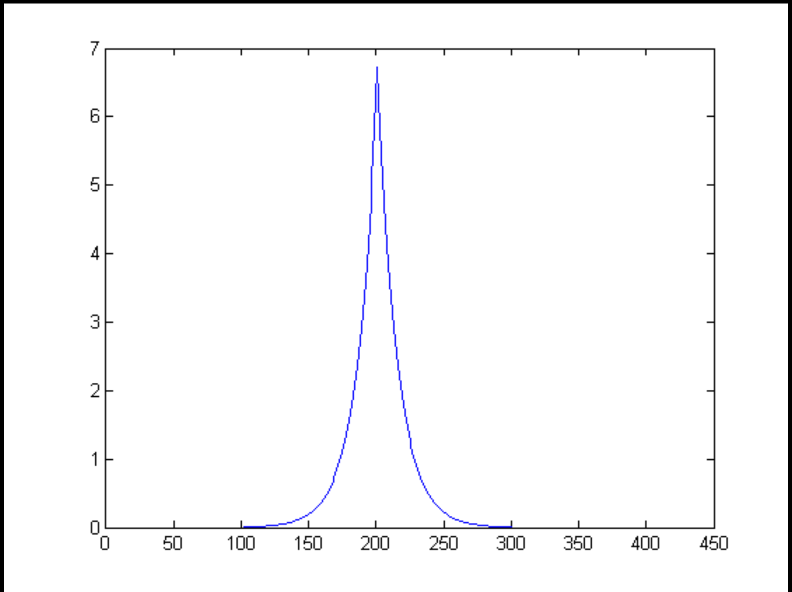
IIR Filters

- Can be done in place ☺
- Makes large, smooth filters, with very little computation ☺
- Somewhat lopsided...

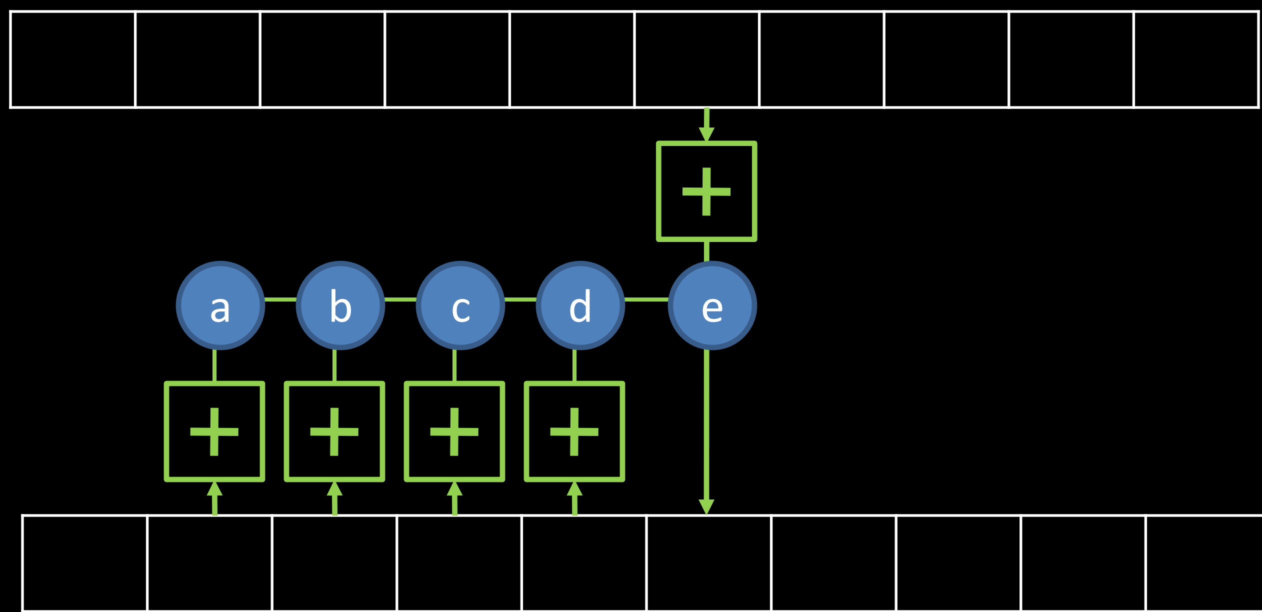


IIR Filters

- One forward pass, one backward pass
- = exponential decay convolved with flipped exponential decay
- Somewhat more Gaussian-ish



More Advanced IIR Filters



Each output is a weighted average of the next input, and the last few outputs

More Advanced IIR Filters

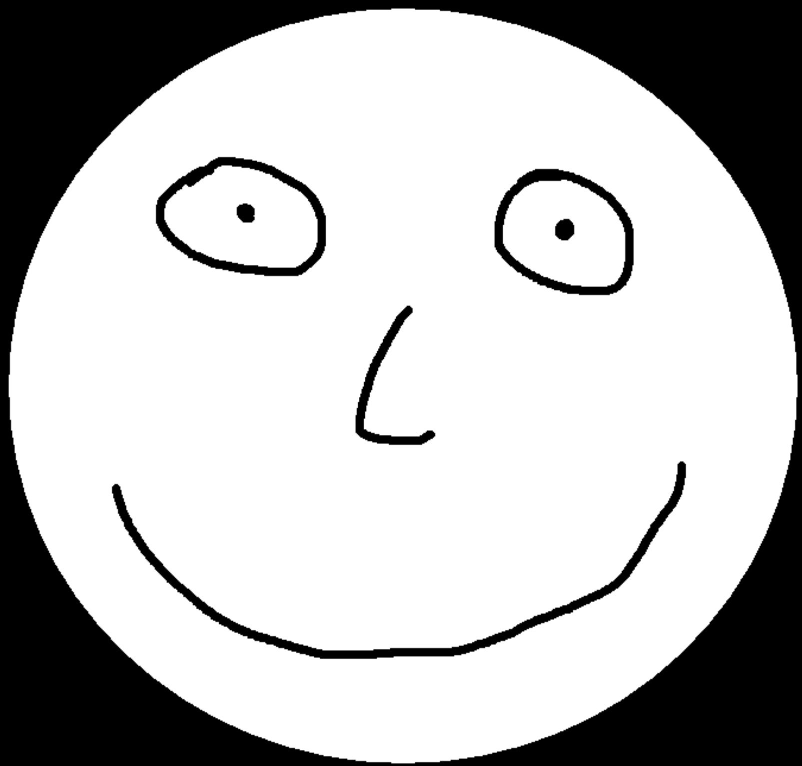
- It’s possible to optimize the parameters to match a Gaussian of a certain std.dev.
- It’s harder to construct a family of them that scales across standard deviations

Filtering by Resampling

- This looks like we just zoomed a small image

- Can we filter by downsampling then upsampling?

Filtering by Resampling



Filtering by Resampling

- Downsampled with rect (averaging down)
- Upsampled with linear interpolation

Use better upsampling?

- Downsampled with rect (averaging down)
- Upsampled with bicubic interpolation

Use better downsampling?

- Downsampled with tent filter
- Upsampled with linear interpolation

Use better downsampling?

- Downsampled with bicubic filter
- Upsampled with linear interpolation

Resampling Simulation

Best Resampling

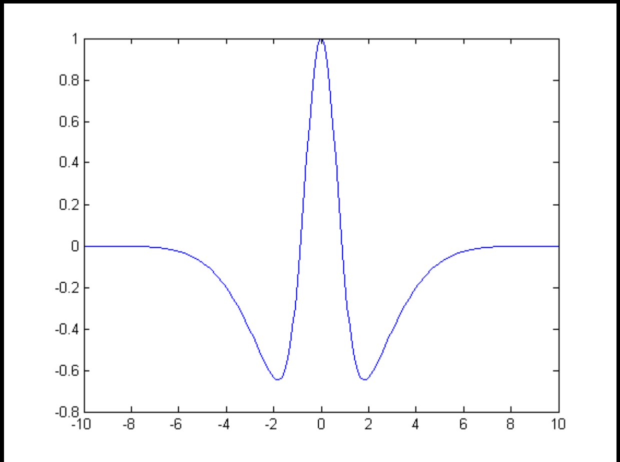
- Downsampled, blurred, then upsampled with bicubic filter

What's the point?

- Q: If we can blur quickly without resampling, why bother resampling?
- A: Memory use
- Store the blurred image at low res, sample it at higher res as needed.

Recap: Fast Linear Filters

1) Separate into a sequence of simpler filters
   - e.g. Gaussian is separable across dimension
   - and can be decomposed into rect filters
2) Separate into a sum of simpler filters



Recap: Fast Linear Filters

3) Separate into a sum of easy-to-precompute components (integral images)

- great if you need to compute lots of different filters

4) Resample

- great if you need to save memory

5) Use feedback loops (IIR filters)

- great if you never need to change the std.dev. of your filter

Your mission:

- Implement one of these fast Gaussian blur methods
- We only care about standard deviations above 2.
- We don't care about boundary conditions (ie we'll ignore everything within 3 standard deviations of the boundary)
- It should be faster than -gaussianblur, and accurate enough to have no visual artifacts
  - precise timing and RMS requirements will be put up soon

Your mission:

- This time we care more about speed and less about accuracy. (40% 20%)
- There will be a competition the Tuesday after the due date.
- Due next Thursday at 11:59pm.
- Email us as before to submit.

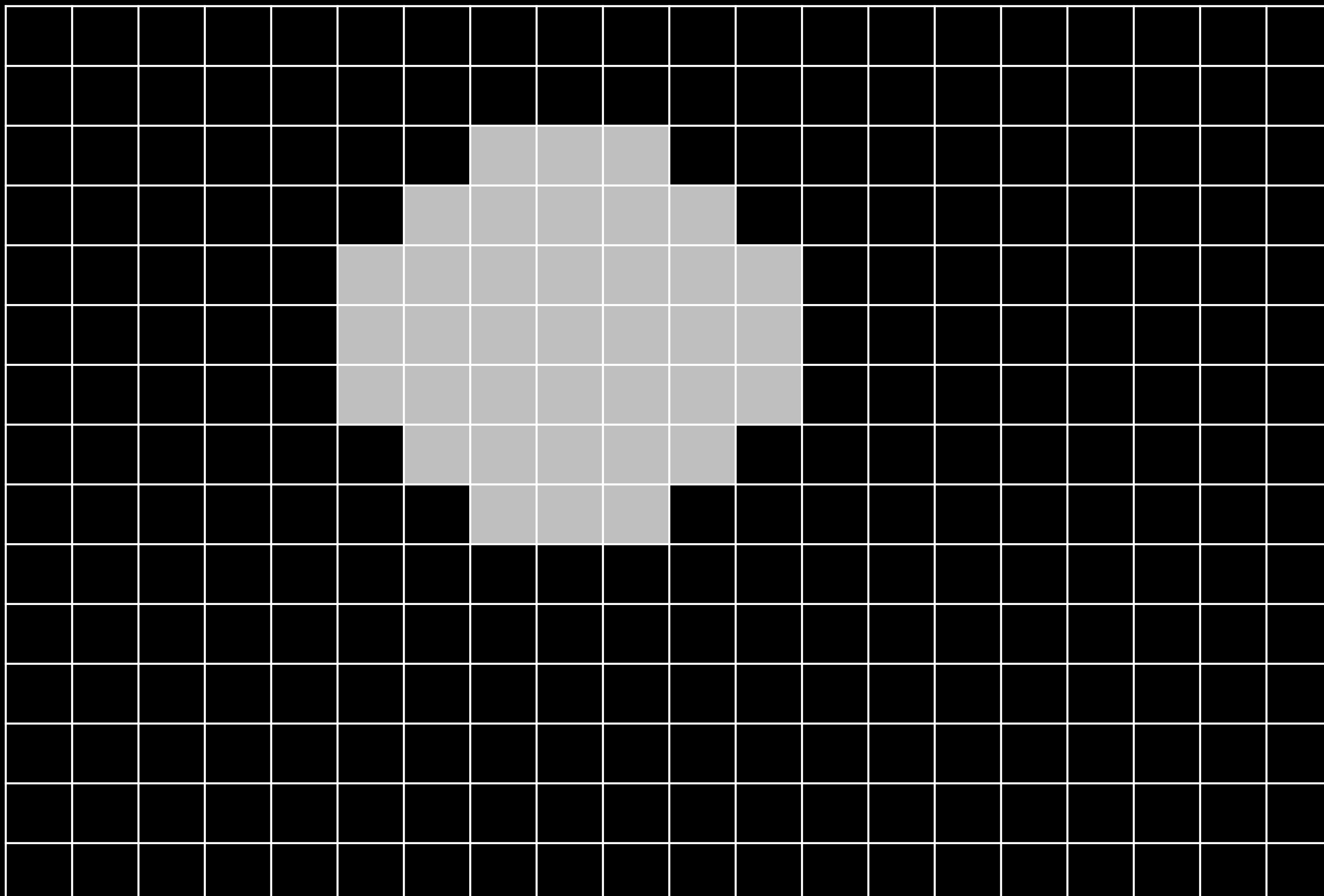
Histogram Filtering

- The fast rect filter
  - maintained a sum
  - updated it for each new pixel
  - didn't recompute from scratch
- What other data structures might we maintain and update for more complex filters?

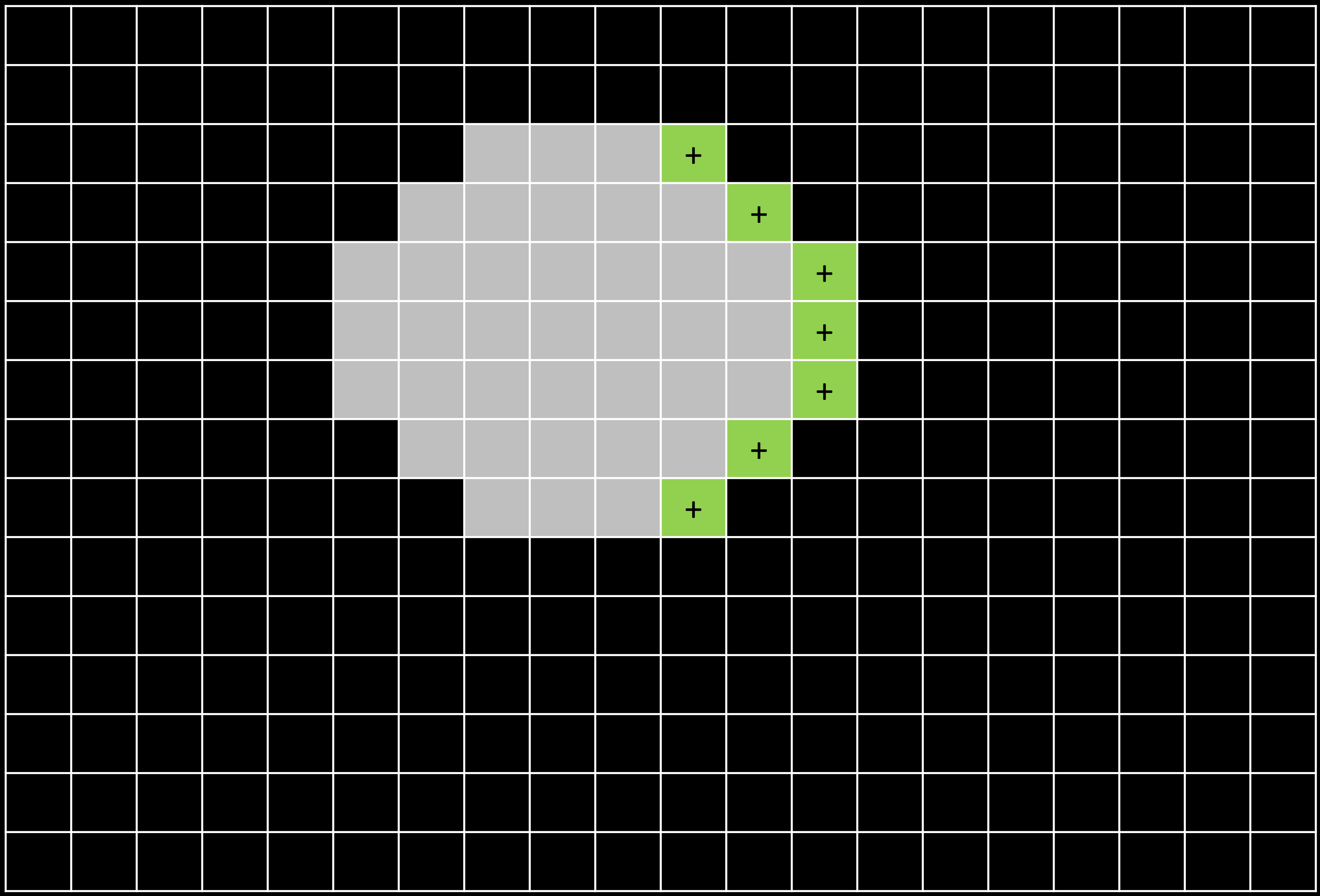
Histogram Filtering

- The min filter, max filter, and median filter
  - Only care about what pixel values fall into neighbourhood, not their location
  - Maintain a histogram of the pixels under the filter window, update it as pixels enter and leave

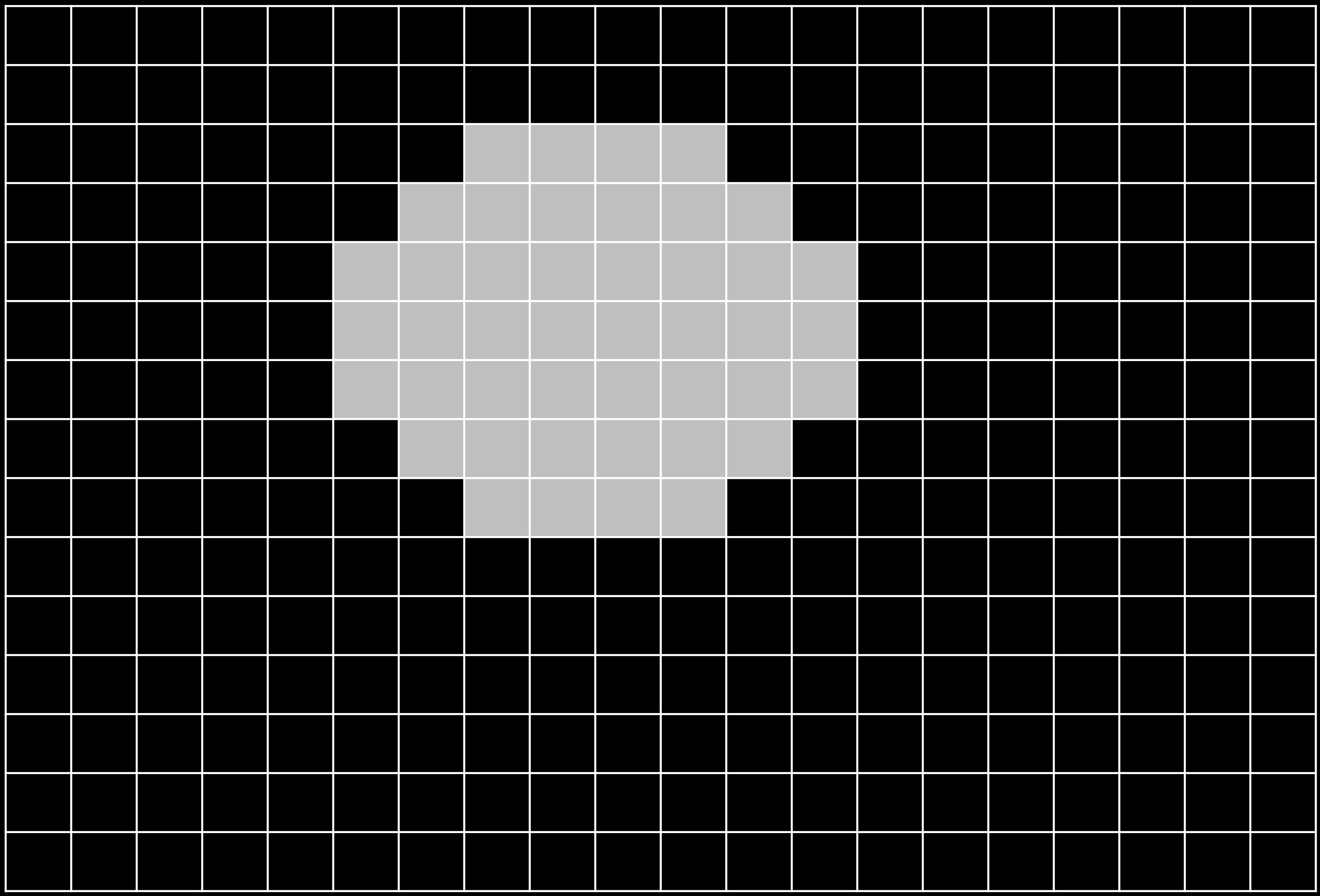
Histogram Updating



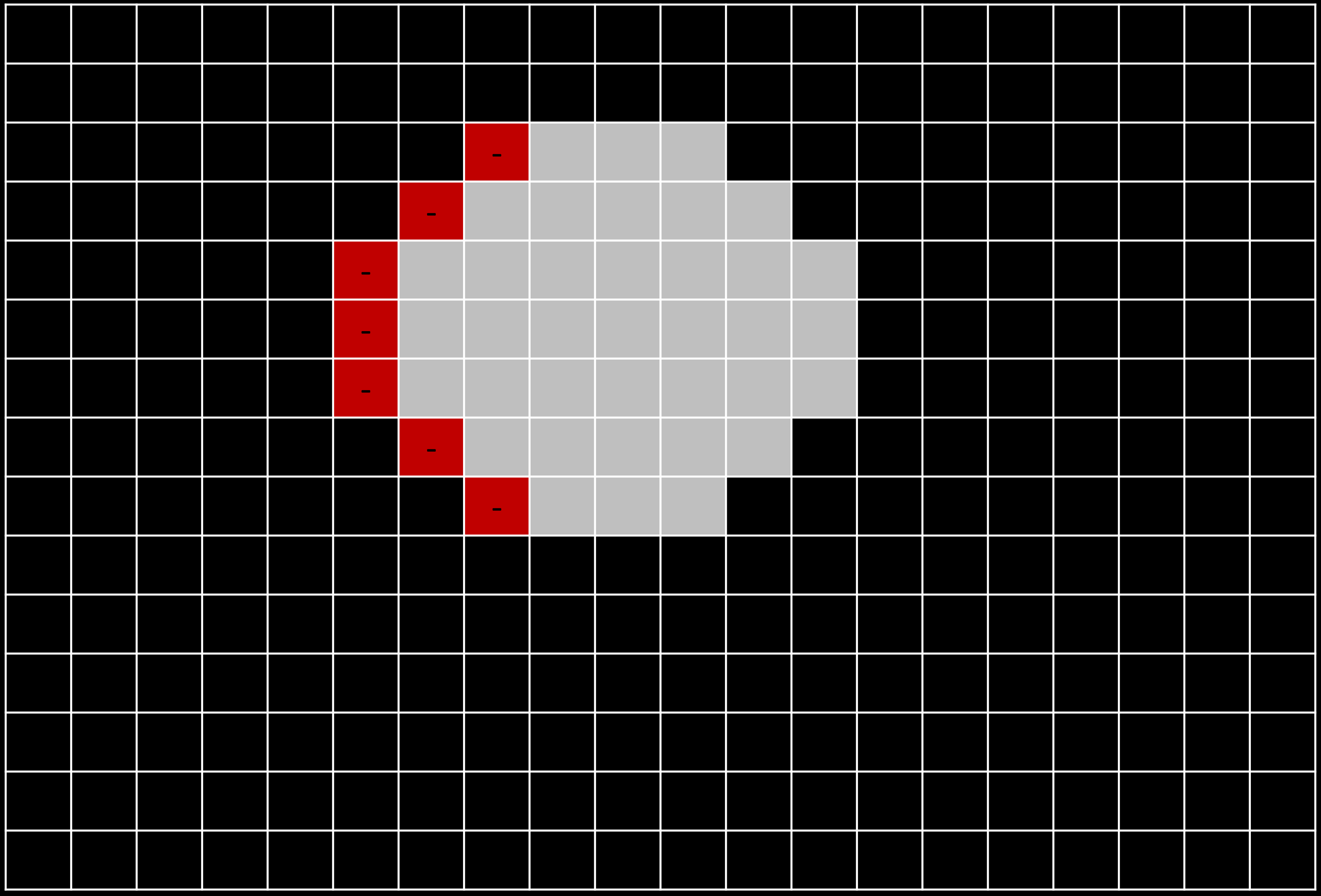
Histogram Updating



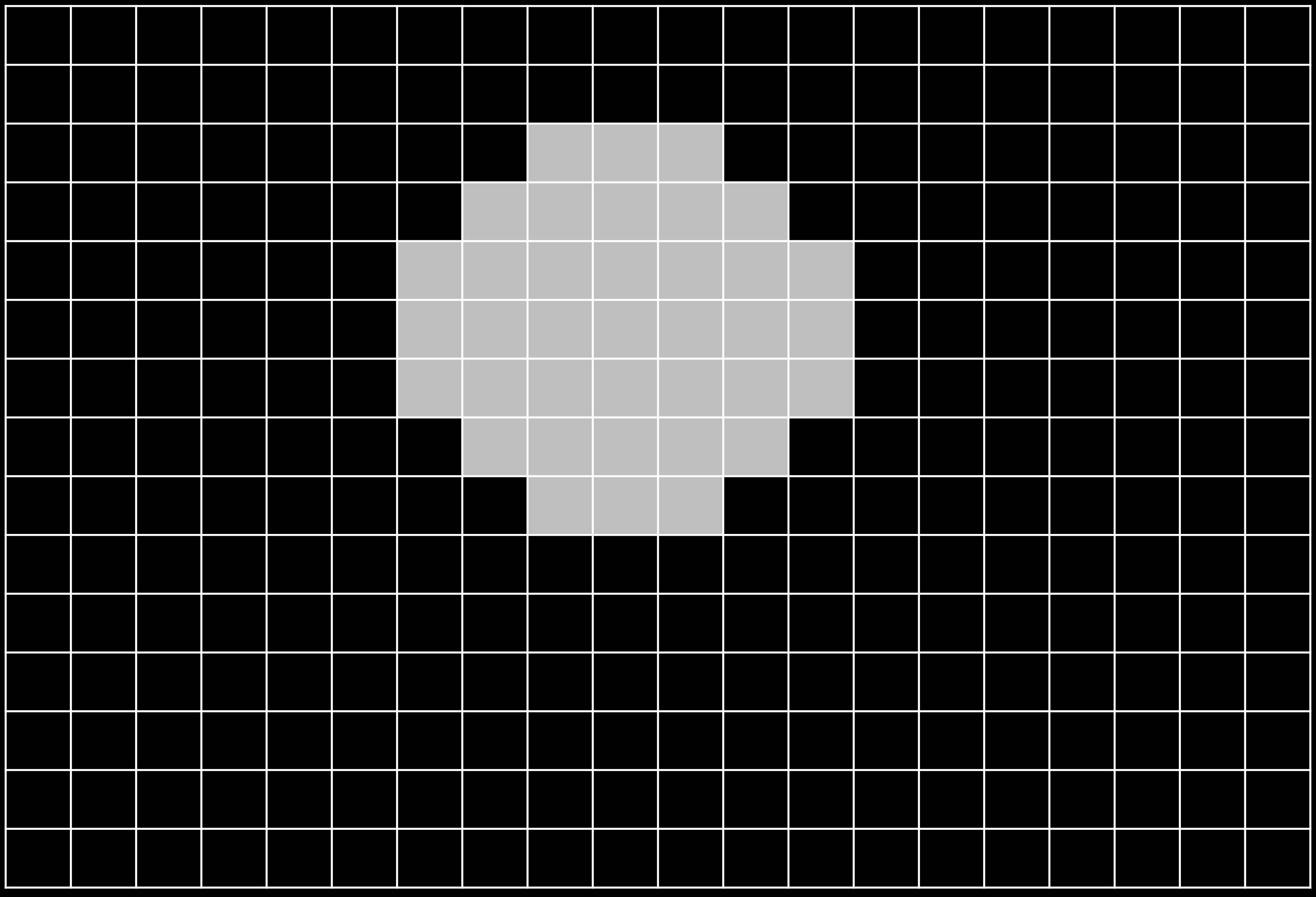
Histogram Updating



Histogram Updating



Histogram Updating



Histogram-Based Fast Median

- Maintain:
  - hist = Local histogram
  - med = Current Median
  - lt = Number of pixels less than current median
  - gt = Number of pixels greater than current median

Histogram-Based Fast Median

- while (lt < gt):
  - med--
  - Update lt and gt using hist
- while (gt < lt):
  - med++
  - Updated lt and gt using hist

Histogram-Based Fast Median

- Complexity?
- Extend this to percentile filters?
- Max filters? Min filters?

The Bilateral Filter

- Pixels are mixed with nearby pixels that have a similar value

$$O(x)=\frac{\sum_{x'=x-f}^{x+f} I(x').e^{-(\sigma_1(I(x)-I(x'))^2)}.e^{-(\sigma_2(x-x')^2)}}{\sum_{x'=x-f}^{x+f} e^{-(\sigma_1(I(x)-I(x'))^2)}.e^{-(\sigma_2(x-x')^2)}}$$

The Bilateral Filter

- We can combine the exponential terms...

$$O(x) = \frac{\sum_{x'=x-f}^{x+f} I(x').e^{-(\sigma_1 (I(x)-I(x'))^2 + \sigma_2 (x-x')^2)}}{\sum_{x'=x-f}^{x+f} e^{-(\sigma_1 (I(x)-I(x'))^2 + \sigma_2 (x-x')^2)}}$$

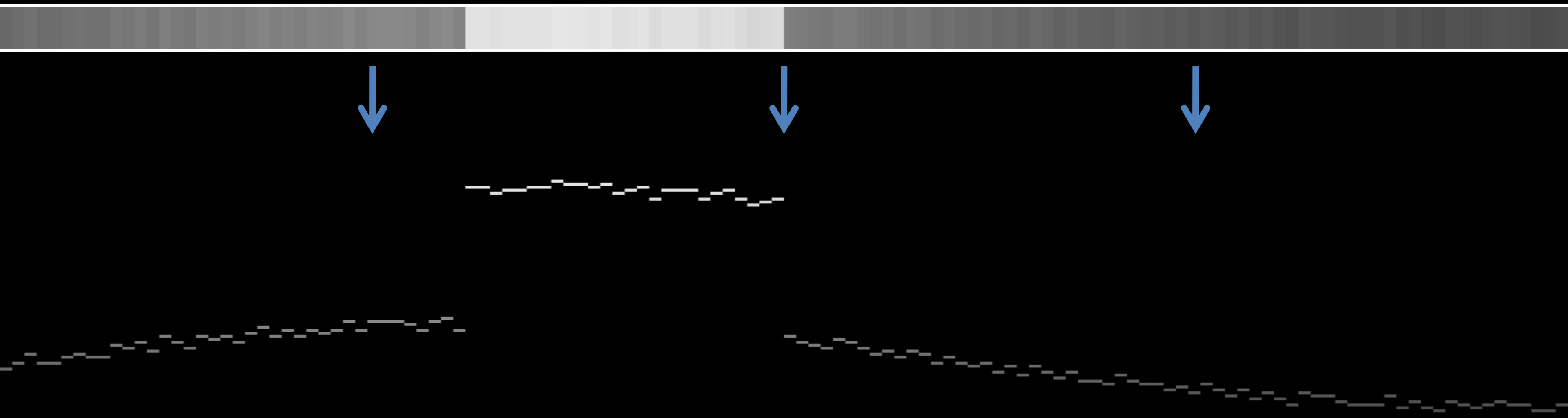
Linearizing the Bilateral Filter

- The product of an 1D gaussian and an 2D gaussian across different dimensions is a single 3D gaussian.
- So we're just blurring in some 3D space
- Axes are:
  - image x coordinate
  - image y coordinate
  - pixel value

The Bilateral Grid – Step 1

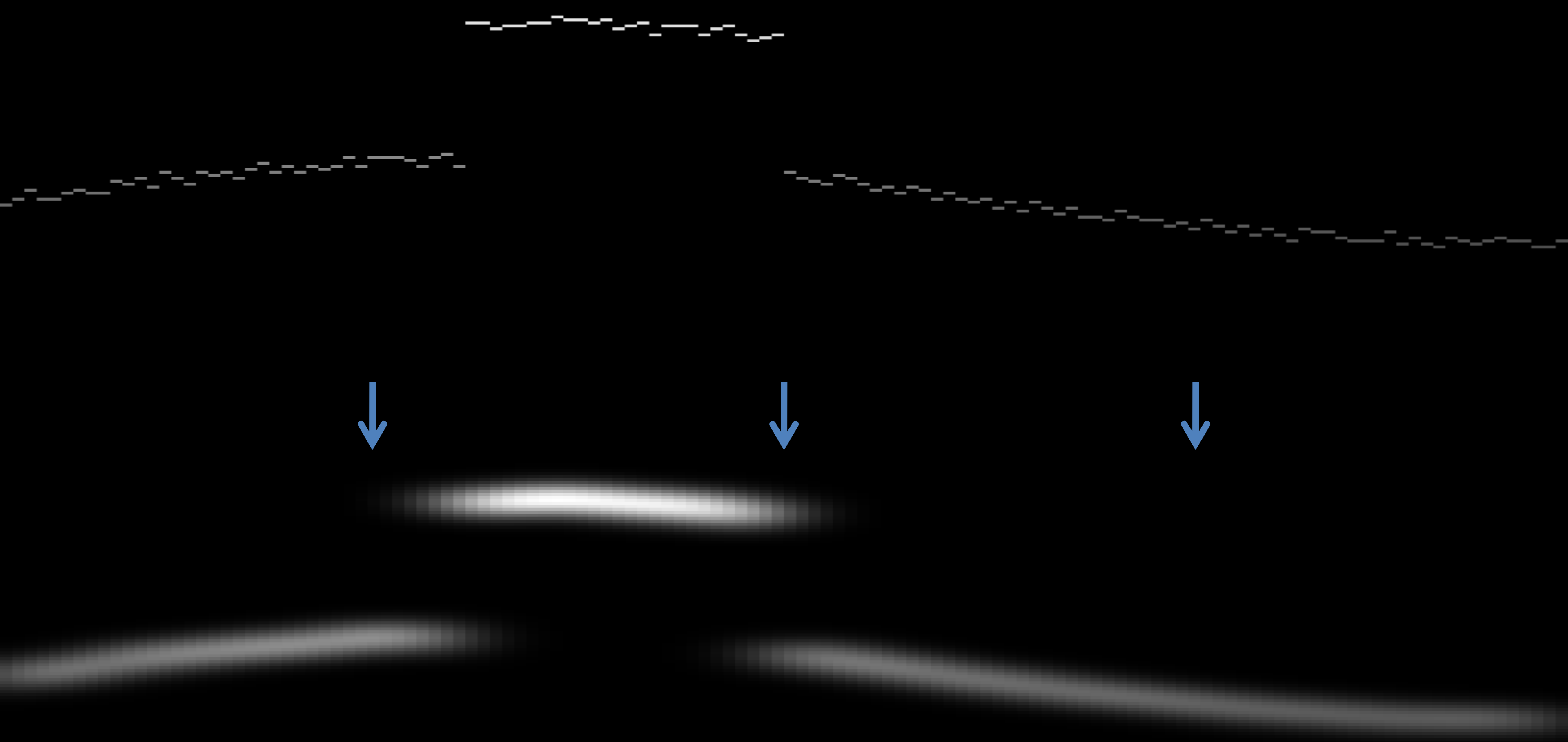
Chen et al SIGGRAPH 07

- Take the 2D image Im(**x**, **y**)
- Create a 3D volume V(**x**, **y**, **z**), such that:
  - Where Im(x, y) = z, V(**x**, **y**, **z**) = **z**
  - Elsewhere, V(**x**, **y**, **z**) = 0



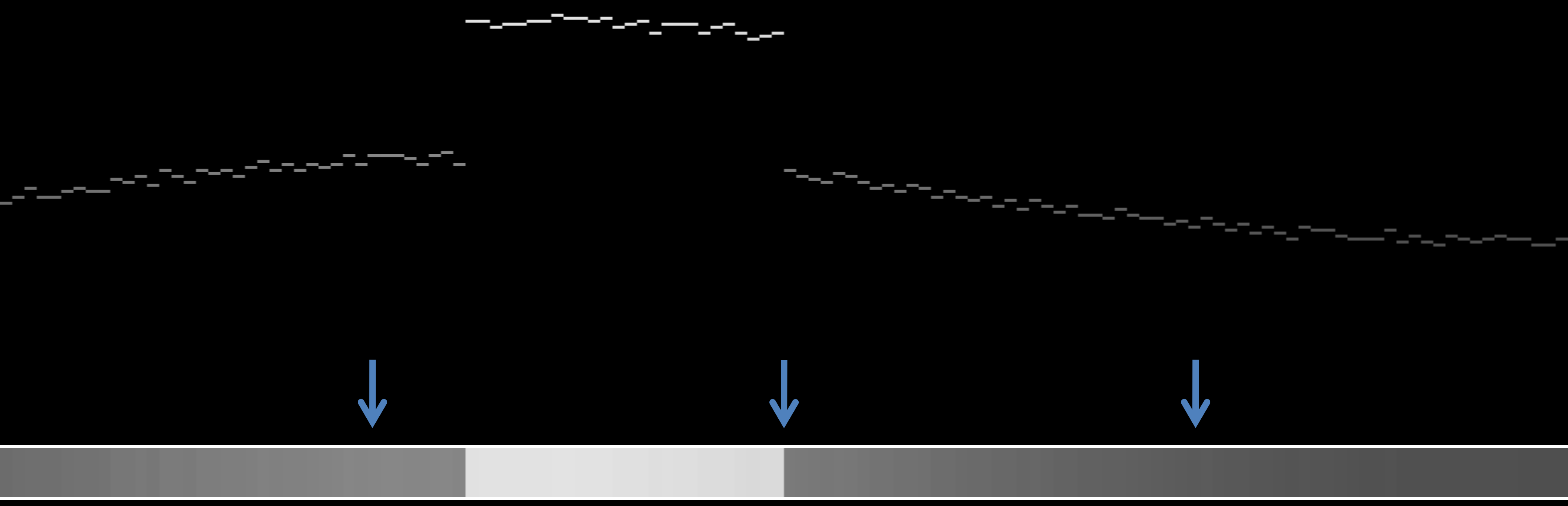
The Bilateral Grid – Step 2

- Blur the 3D volume (using a fast blur)



The Bilateral Grid – Step 3

- Slice the volume at z values corresponding to the original pixel values



Comparison

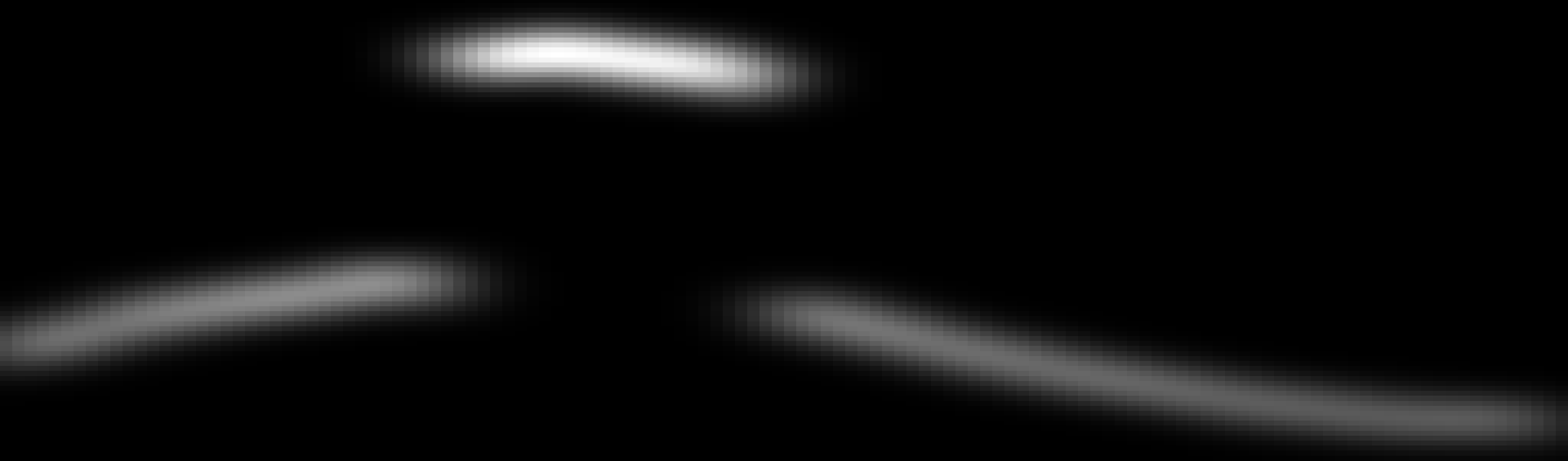
Input

Regular blur

Bilateral Grid Slice

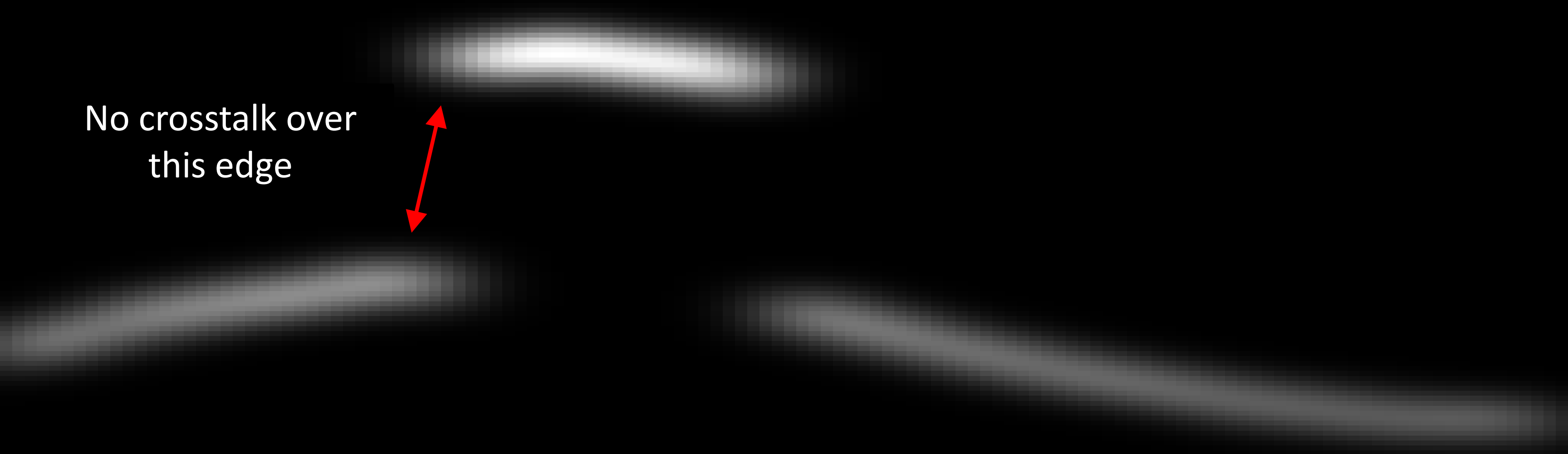
Pixel Influence

- Each pixel blurred together with
  - those nearby in space (x coord on this graph)
  - and value (y coord on this graph)



Pixel Influence

- Each pixel blurred together with
  - those nearby in space (x coord on this graph)
  - and value (y coord on this graph)



The weight channel

- This actually just computes:

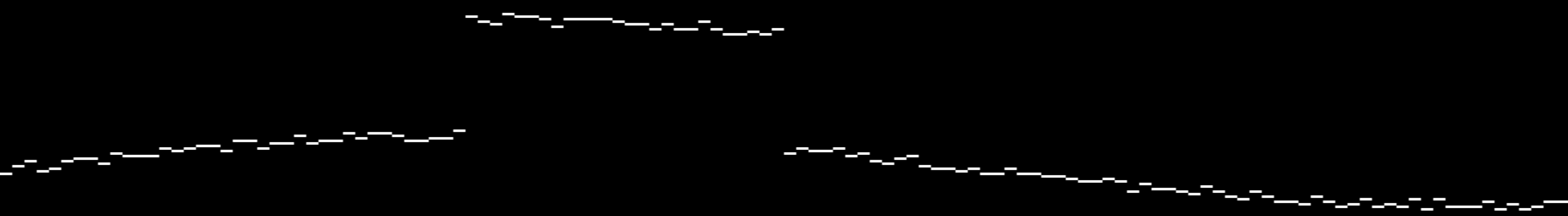
$$O(x,y)=\sum_{v=-f}^{f}\sum_{u=-f}^{f} I(x+u,y+v).e^{-(\sigma_1 b^2+\sigma_2 u^2+\sigma_2 v^2)}$$

- We need:

$$O(x,y)=\frac{\sum_{v=-f}^{f}\sum_{u=-f}^{f} I(x+u,y+v).e^{-(\sigma_1 b^2+\sigma_2 u^2+\sigma_2 v^2)}}{\sum_{v=-f}^{f}\sum_{u=-f}^{f} e^{-(\sigma_1 b^2+\sigma_2 u^2+\sigma_2 v^2)}}$$

The weight channel

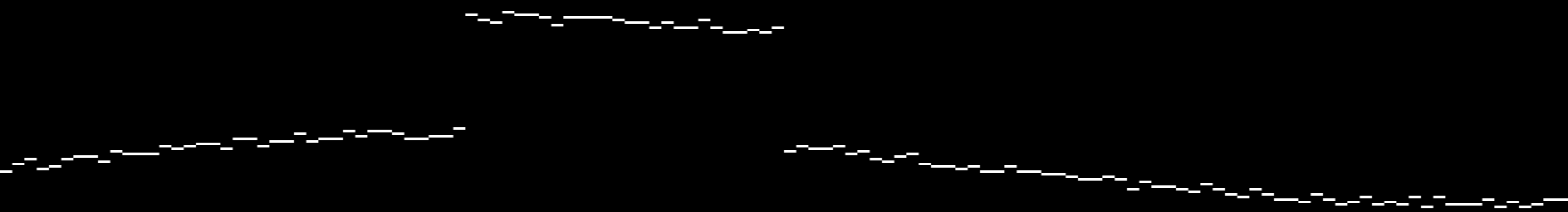
- Solution: add a weight channel
- Create a 3D volume V(**x**, **y**, **z**), such that:
  - Where Im(x, y) = z, V(**x**, **y**, **z**) = **(z,1)**
  - Elsewhere, V(**x**, **y**, **z**) = (0,0)



- At the end, divide by the weight channel

Bilateral Grid = Local Histogram Transform

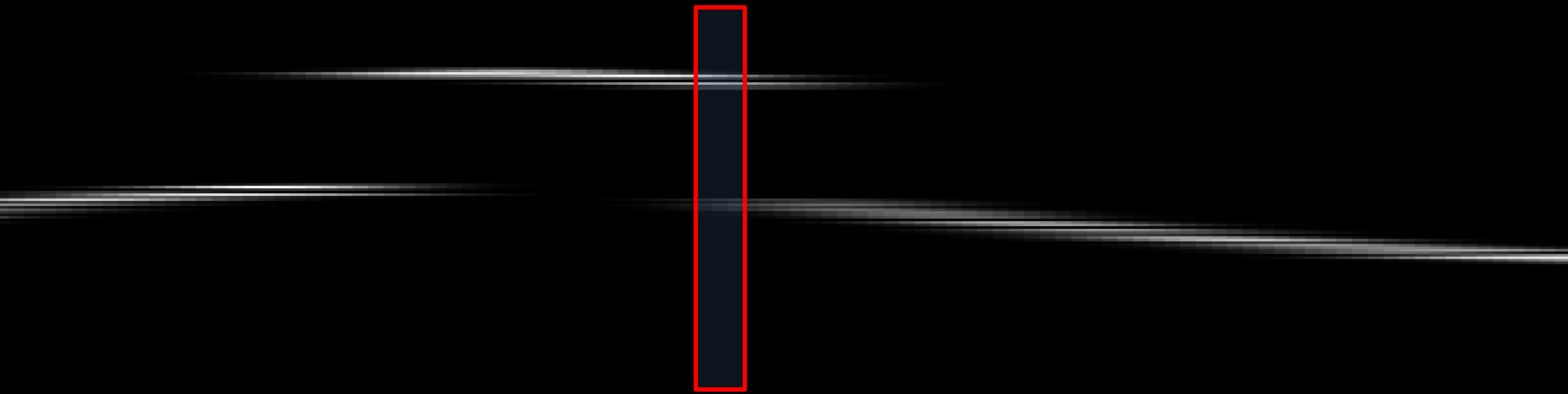
- Take the weight channel:



- Blur in space (but not value)

Bilateral Grid = Local Histogram Transform

- One column is now the histogram of a region around a pixel!



- If we blur in value too, it's just a histogram with fewer buckets
- Useful for median, min, max filters as well.

The Elephant in the Room

- Why hasn't anyone done this before?
- For a 5 megapixel image at 3 bytes per pixel, the bilateral grid with 256 value buckets would take up:
  - 5*1024*1024*(3+1)*256 = 5120 Megabytes
- But wait, we never need the original grid, just the original grid blurred...

Use Filtering by Resampling!

- Construct the bilateral grid at low resolution
  - Use a good downsampling filter to put values in the grid
  - Blur the grid with a small kernel (eg 5x5)
  - Use a good upsampling filter to slice the grid
- Complexity?
  - Regular bilateral filter: O(w*h*f*f)
  - Bilateral grid implementation:
    - time: O(w*h)
    - memory: O(w/f * h/f * 256/g)

Use Filtering by Resampling!

- Construct the bilateral grid at low resolution
  - Use a good downsampling filter to put values in the grid
  - Blur the grid with a small kernel (eg 5x5)
  - Use a good upsampling filter to slice the grid
- Complexity?
  - Regular bilateral filter: O(w*h*f*f)
  - Bilateral grid implementation:
    - time: O(w*h)
    - memory: O(w/f * h/f * 256/g)

Gets smaller as the filter gets larger!

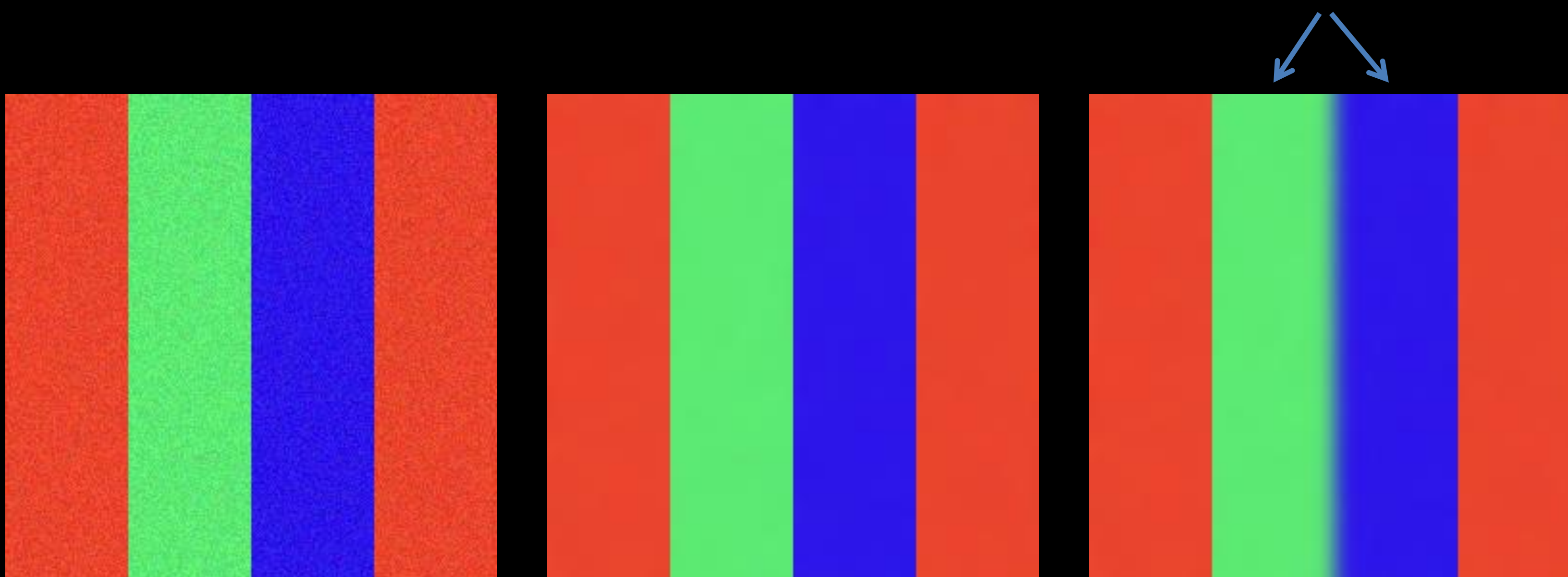
Dealing with Color

- I've treated value as 1D, it's really 3D
- The bilateral grid should hence really be 5D
- Memory usage starts to go up...
- Most people just use distance in luminance instead of full 3D distance
  - ***values*** in grid are 3D colors (4 bytes per entry)
  - ***positions*** of values is just the 1D luminance

    = (R+G+B)/3

Using distance in 3D
vs
Just using distance in luminance

Same luminance



Input

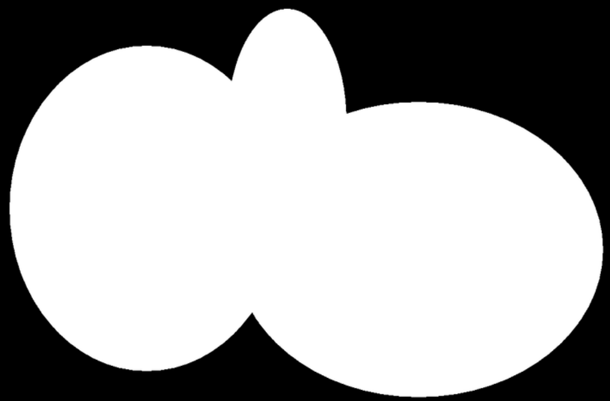
Full Bilateral

Luminance Only Bilateral

Wait, this ‘limitation’ can be useful

- **Values** in the bilateral grid are the things we want to blur
- **Positions** (and hence distances) in the bilateral grid determine which values we mix
- So we could, for example, get the positions from one image, and the values from another

Joint Bilateral Filter



Reference Image

Input Image

Result

Joint Bilateral Application

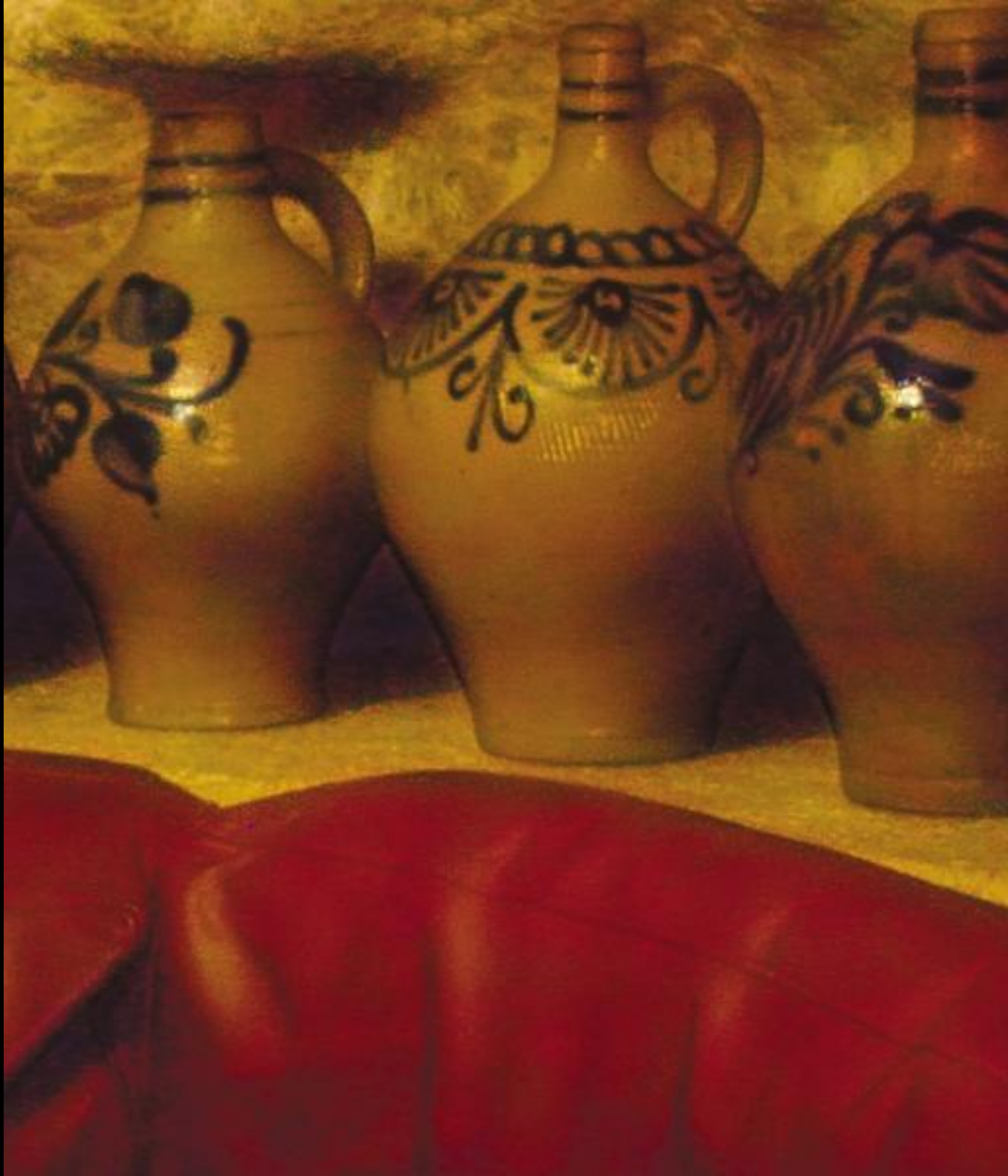
- Flash/No Flash photography
- Take a photo with flash (colors look bad)
- Take a photo without flash (noisy)
- Use the edges from the flash photo to help smooth the blurry photo
- Then add back in the high frequencies from the flash photo
- **Digital Photography with Flash and No-Flash Image Pairs**

  Petschnigg et al, SIGGRAPH 04

Flash:

No Flash:

Result:



Joint Bilateral Upsample

Kopf et al, SIGRAPH 07

- Say we've computed something expensive at low resolution (eg tonemapping, or depth)
- We want to use the result at the original resolution
- Use the original image as the positions
- Use the low res solution as the values
- Since the bilateral grid is low resolution anyway, just:
  - read in the low res values at positions given by the downsampled high res image
  - slice using the high res image

Joint Bilateral Upsample Example

- Low resolution depth, high resolution color
- Depth edges probably occur at color edges

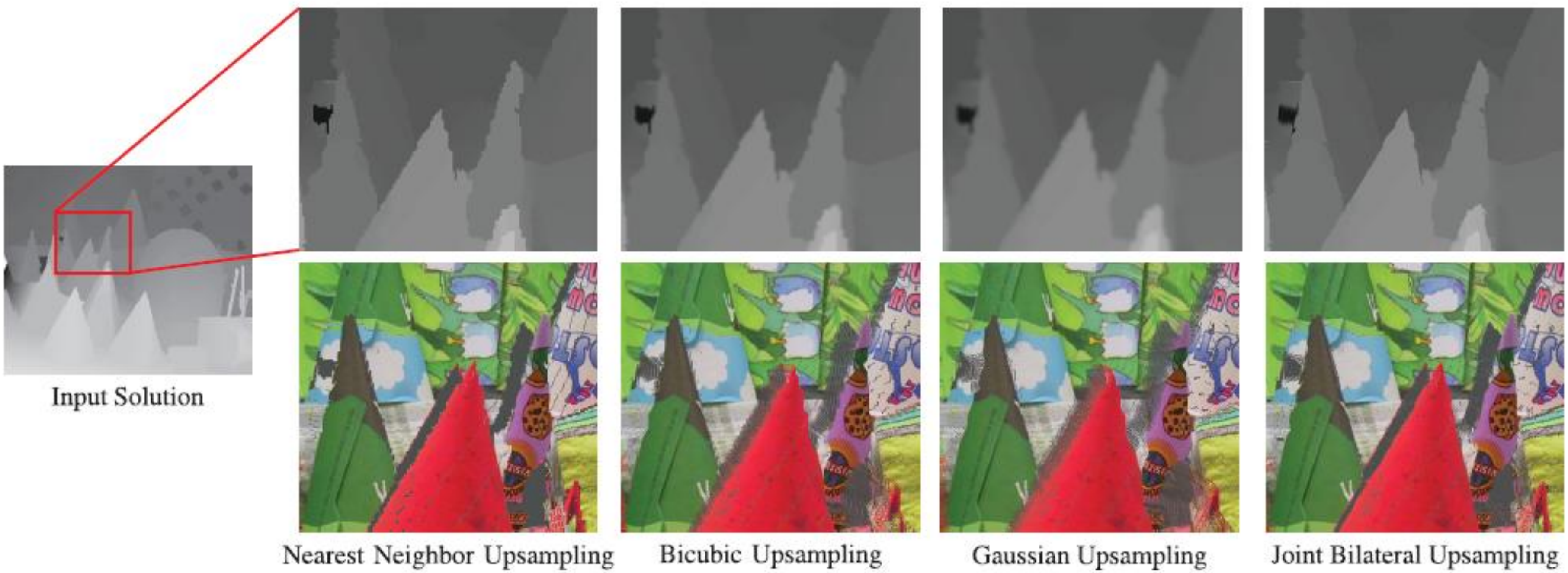


Figure 4: Stereo Depth: The low resolution depth map is shown at left. The top right row shows details from the upsampled maps using different methods. Below each detail image is a corresponding 3d view from an offset camera using the upsampled depth map.

One final use of the bilateral filter

Video Enhancement Using Per-Pixel Virtual Exposures

Bennett & McMillan, SIGGRAPH 05

- ASTA: Adaptive Spatio-Temporal Accumulation
- Do a 3D bilateral filter on a video
- Where something isn’t moving, it will mostly average over time
- Where something is moving, it will just average over space

Key Ideas

- Filtering (even bilateral filtering) is O(w*h)
- You can also filter by downsampling, possibly blurring a little, then upsampling
- The bilateral grid is a local histogram transform that's useful for many things