A Gentle Introduction
to Bilateral Filtering
and its Applications

Efficient Implementations of the Bilateral Filter

Sylvain Paris – Adobe

Outline

- Brute-force Implementation
- Separable Kernel [Pham and Van Vliet 05]
- Box Kernel [Weiss 06]
- 3D Kernel [Paris and Durand 06]

Brute-force Implementation

$$BF[I]_{\mathbf{p}} = \frac{1}{W_{\mathbf{p}}} \sum_{\mathbf{q} \in S} G_{\sigma_s}(\|\mathbf{p}-\mathbf{q}\|)\, G_{\sigma_r}(|I_{\mathbf{p}} - I_{\mathbf{q}}|)\, I_{\mathbf{q}}$$

For each pixel $\mathbf{p}$

For each pixel $\mathbf{q}$

Compute $G_{\sigma_s}(\|\mathbf{p}-\mathbf{q}\|)\, G_{\sigma_r}(|I_{\mathbf{p}} - I_{\mathbf{q}}|)\, I_{\mathbf{q}}$

8 megapixel photo: 64,000,000,000,000 iterations!

V E R Y S L O W !

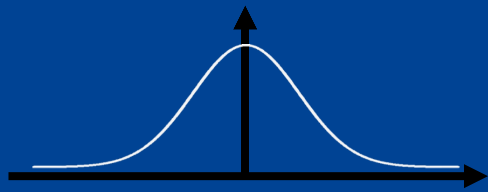
More than 10 minutes per image

Complexity

- Complexity = *"how many operations are needed, how this number varies"*
- S = space domain = set of pixel positions
- $|S|$ = cardinality of S = number of pixels
  - In the order of 1 to 10 millions
- Brute-force implementation: $O(|S|^2)$

Better Brute-force Implementation

Idea: Far away pixels are negligible

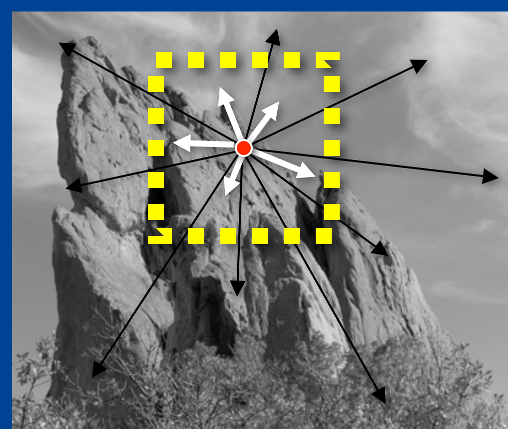


For each pixel $\mathbf{p}$

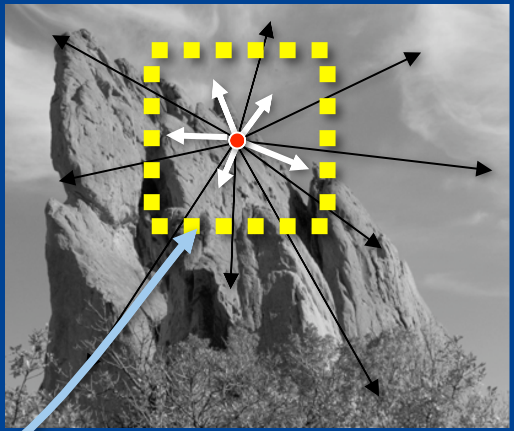
a. For each pixel $\mathbf{q}$ such that $\| \mathbf{p} - \mathbf{q} \| < cte \times \sigma_s$

looking at all pixels

looking at neighbors only



Discussion



- Complexity: $O(|S| \times \sigma_s^2)$

neighborhood area

- Fast for small kernels: σ_s ~ 1 or 2 pixels

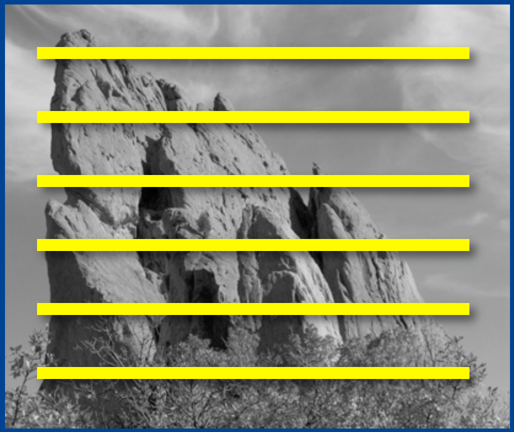
- BUT: slow for larger kernels

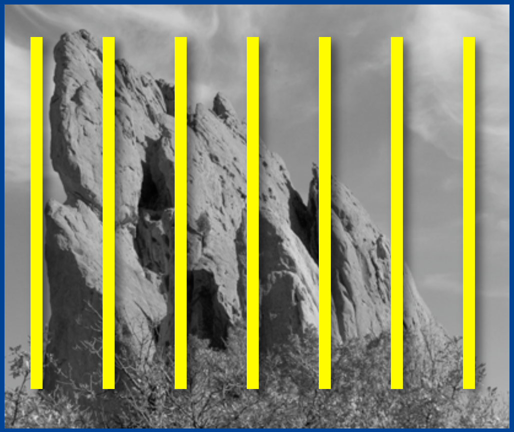
Outline

- Brute-force Implementation
- Separable Kernel [Pham and Van Vliet 05]
- Box Kernel [Weiss 06]
- 3D Kernel [Paris and Durand 06]

Separable Kernel [Pham and Van Vliet 05]

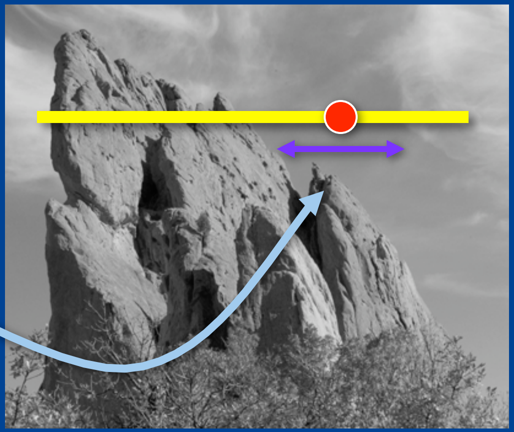
- Strategy: filter the rows then the columns





- Two "cheap" 1D filters
  instead of an "expensive" 2D filter

Discussion



- Complexity: $O(|S| \times \sigma_s)$
  - Fast for small kernels (<10 pixels)
- Approximation: BF kernel not separable
  - Satisfying at strong edges and uniform areas
  - Can introduce visible streaks on textured regions


input


brute-force
implementation


separable kernel
mostly OK,
some visible artifacts
(streaks)

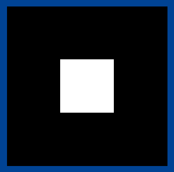
Outline

- Brute-force Implementation
- Separable Kernel [Pham and Van Vliet 05]
- Box Kernel [Weiss 06]
- 3D Kernel [Paris and Durand 06]

Box Kernel [Weiss 06]

- Bilateral filter with a square box window [Yarovlasky 85]

$$Y[I]_{\mathbf{p}} = \frac{1}{W_{\mathbf{p}}} \sum_{\mathbf{q} \in S} B_{\sigma_s}(\| \mathbf{p} - \mathbf{q} \|) G_{\sigma_r}(| I_{\mathbf{p}} - I_{\mathbf{q}} |) I_{\mathbf{q}}$$



box window

restrict the sum

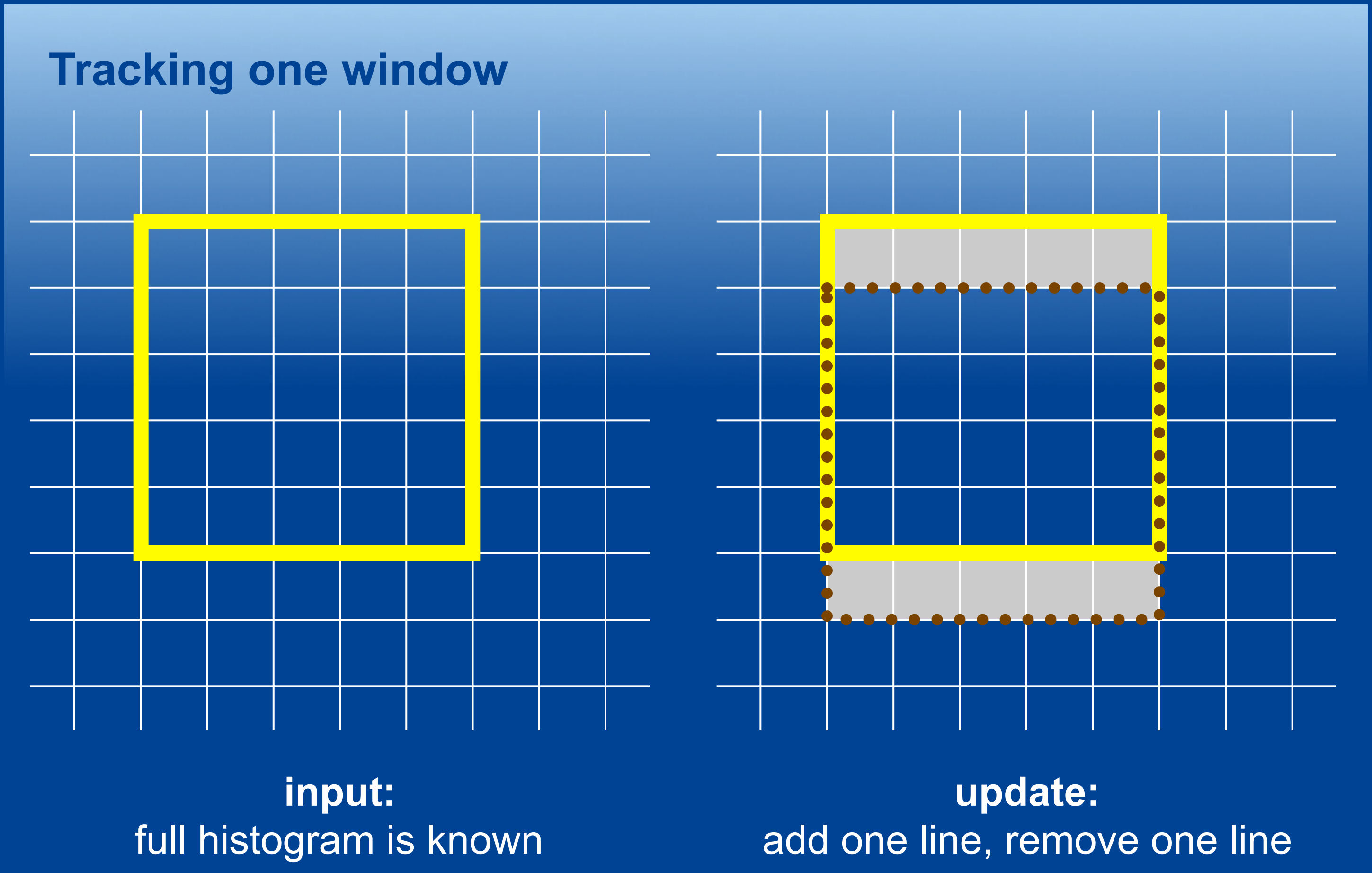
$$Y[I]_{\mathbf{p}} = \frac{1}{W_{\mathbf{p}}} \sum_{\mathbf{q} \in B_{\sigma_s}} G_{\sigma_r}(| I_{\mathbf{p}} - I_{\mathbf{q}} |) I_{\mathbf{q}}$$

independent of position $\mathbf{q}$

- The bilateral filter can be computed only from the list of pixels in a square neighborhood.

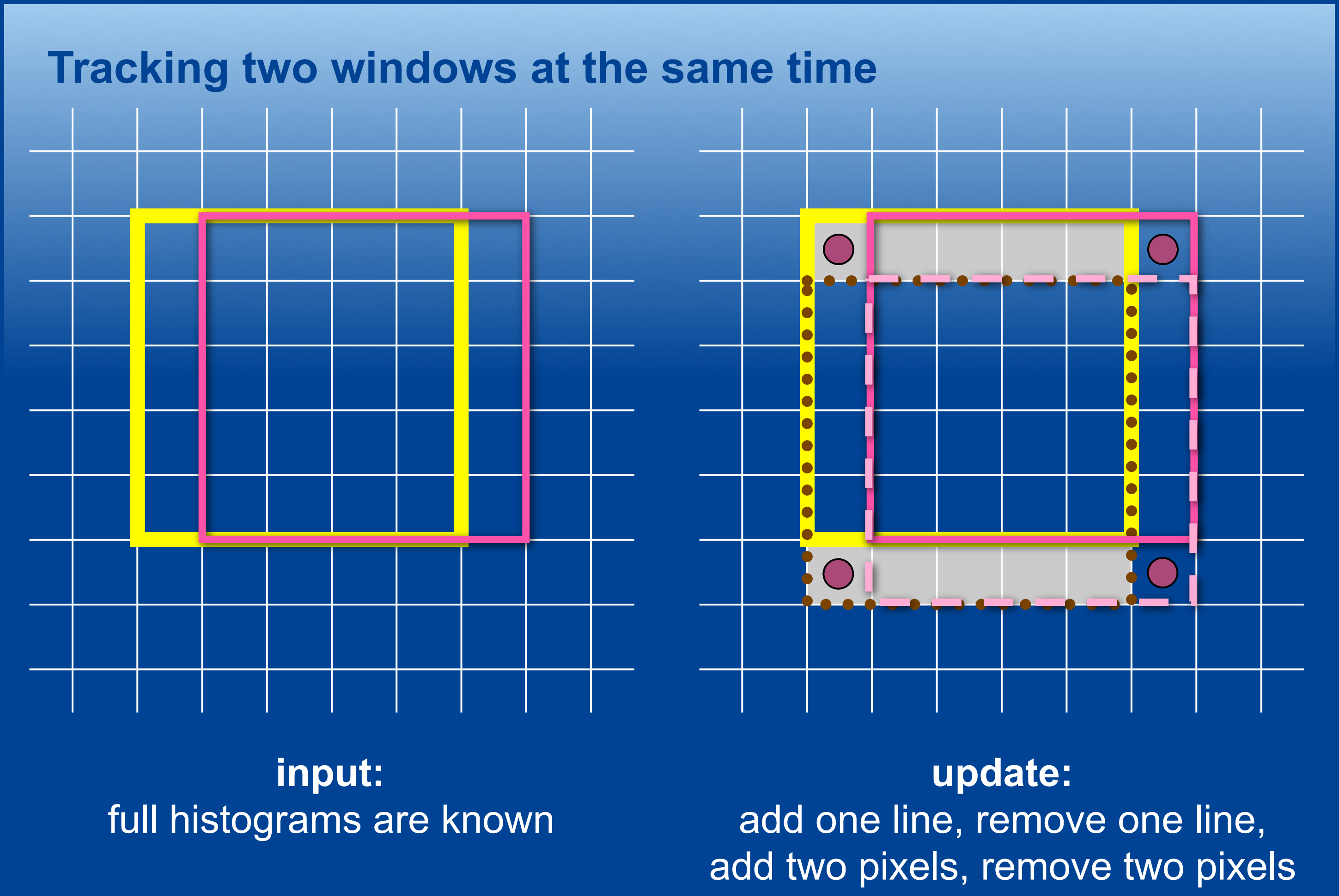
Box Kernel [Weiss 06]

- Idea: fast histograms of square windows



Box Kernel [Weiss 06]

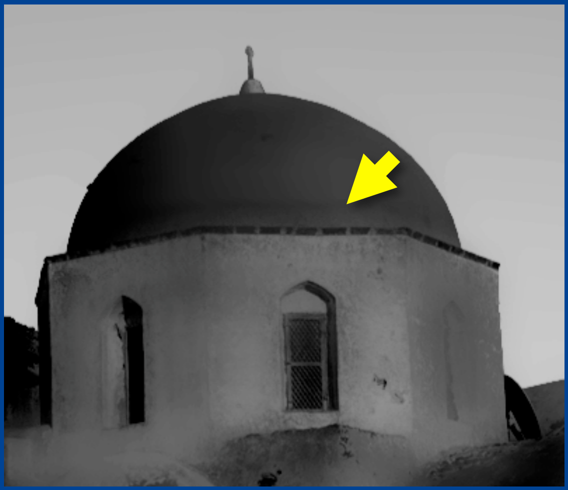
- Idea: fast histograms of square windows



Discussion

- Complexity: $O(|S| \times \log \sigma_s)$
  - always fast
- Only single-channel images
- Exploit vector instructions of CPU
- Visually satisfying results (no artifacts)
  - 3 passes to remove artifacts due to box windows (Mach bands)

1 iteration



3 iterations

Bilateral Filtering in O(1) [Porikli CVPR'08]

- Uses *integral histograms* to remove the log
- Uses Taylor expansion and *power images*
- Memory intensive (1 histogram per pixel)


input


brute-force
implementation


box kernel
visually different,
yet no artifacts

Outline

- Brute-force Implementation
- Separable Kernel [Pham and Van Vliet 05]
- Box Kernel [Weiss 06]
- 3D Kernel [Paris and Durand 06]

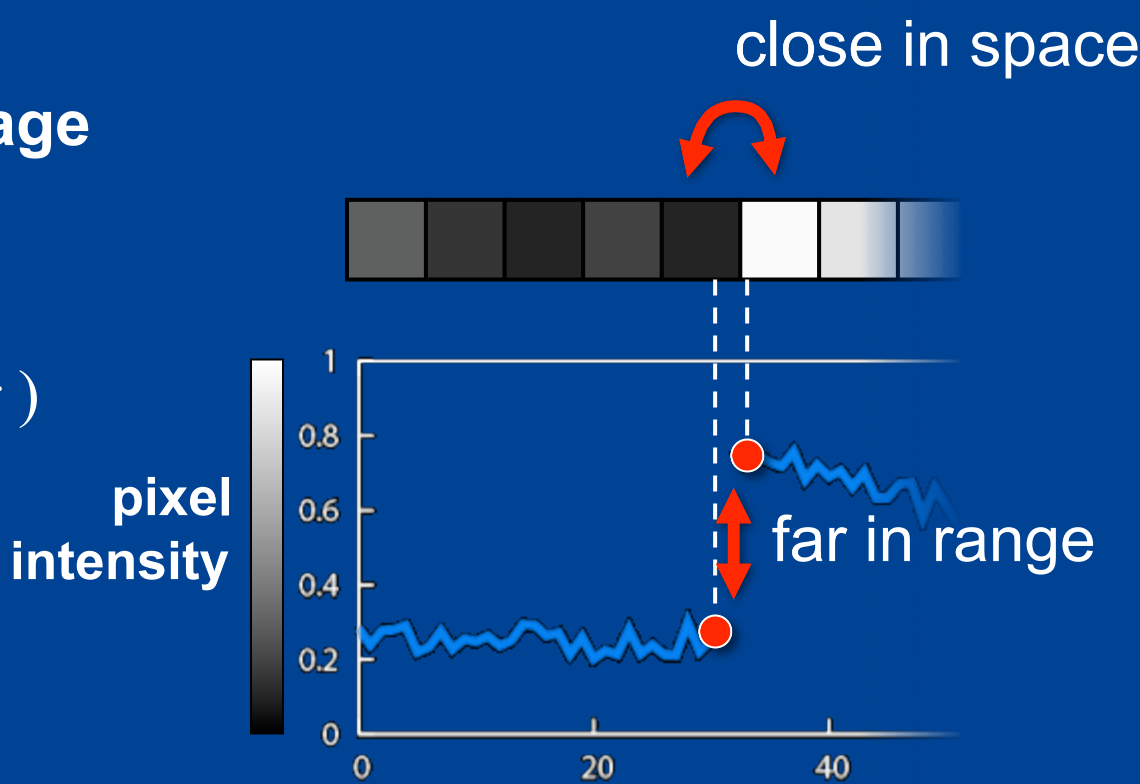
3D Kernel [Paris and Durand 06]

- Idea: represent image data such that the weights depend only on the distance between points

1D image

Plot

$I = f(x)$



1st Step: Re-arranging Symbols

$$BF[I]_{\mathbf{p}} = \frac{1}{W_{\mathbf{p}}} \sum_{\mathbf{q} \in S} G_{\sigma_s}(\| \mathbf{p} - \mathbf{q} \|) \, G_{\sigma_r}(| I_{\mathbf{p}} - I_{\mathbf{q}} |) \, I_{\mathbf{q}}$$

$$W_{\mathbf{p}} = \sum_{\mathbf{q} \in S} G_{\sigma_s}(\| \mathbf{p} - \mathbf{q} \|) \, G_{\sigma_r}(| I_{\mathbf{p}} - I_{\mathbf{q}} |)$$

Multiply first equation by $W_{\mathbf{p}}$

$$W_{\mathbf{p}} \, BF[I]_{\mathbf{p}} = \sum_{\mathbf{q} \in S} G_{\sigma_s}(\| \mathbf{p} - \mathbf{q} \|) \, G_{\sigma_r}(| I_{\mathbf{p}} - I_{\mathbf{q}} |) \, I_{\mathbf{q}}$$

$$W_{\mathbf{p}} = \sum_{\mathbf{q} \in S} G_{\sigma_s}(\| \mathbf{p} - \mathbf{q} \|) \, G_{\sigma_r}(| I_{\mathbf{p}} - I_{\mathbf{q}} |) \, 1$$

1st Step: Summary

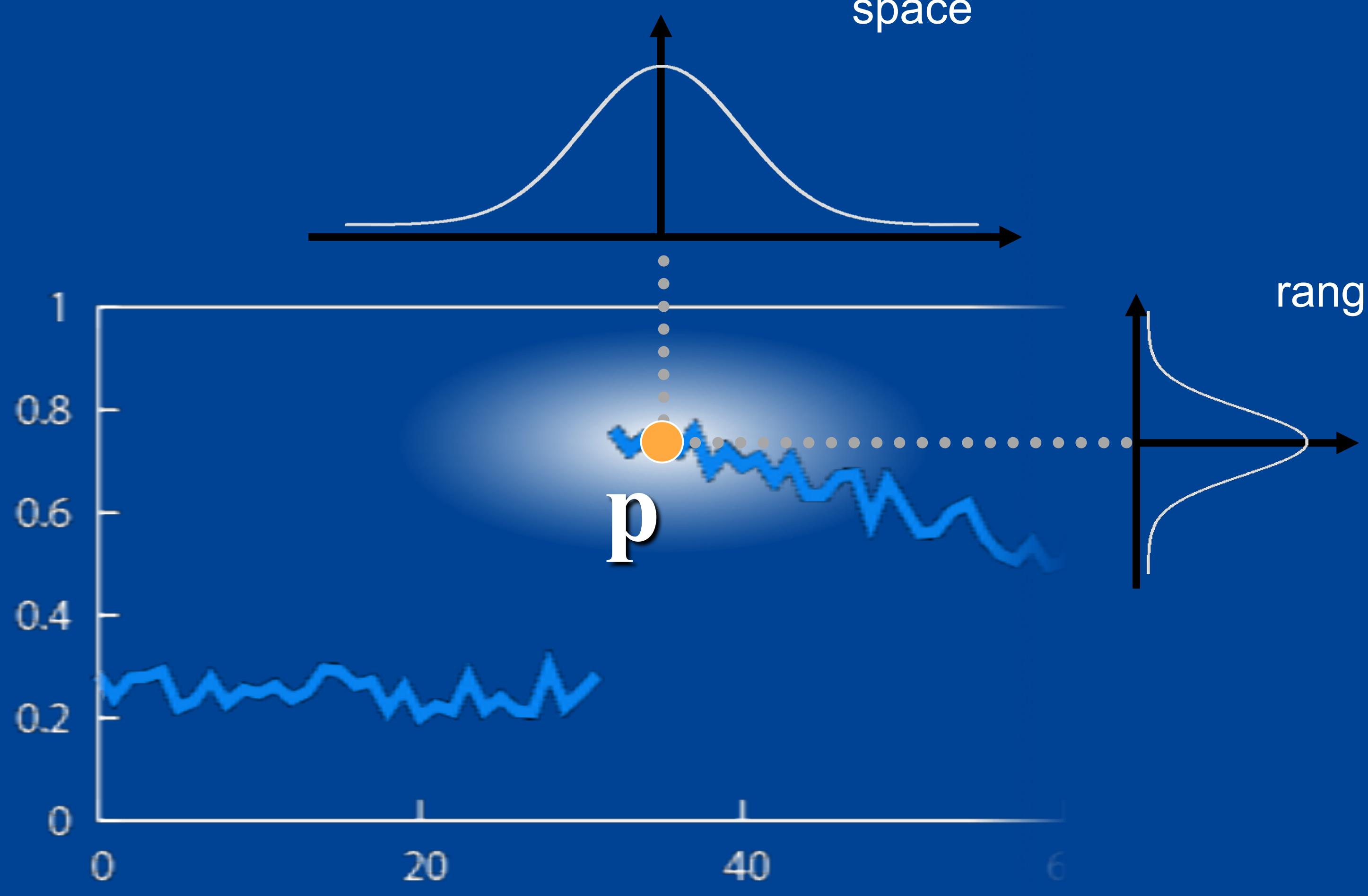
$$W_{\mathbf{p}}\, BF[I]_{\mathbf{p}} = \sum_{\mathbf{q}\in S} G_{\sigma_s}\left(\|\mathbf{p}-\mathbf{q}\|\right) G_{\sigma_r}\left(|I_{\mathbf{p}} - I_{\mathbf{q}}|\right) I_{\mathbf{q}}$$

$$W_{\mathbf{p}} = \sum_{\mathbf{q}\in S} G_{\sigma_s}\left(\|\mathbf{p}-\mathbf{q}\|\right) G_{\sigma_r}\left(|I_{\mathbf{p}} - I_{\mathbf{q}}|\right) 1$$

- Similar equations
- No normalization factor anymore
- Don’t forget to divide at the end

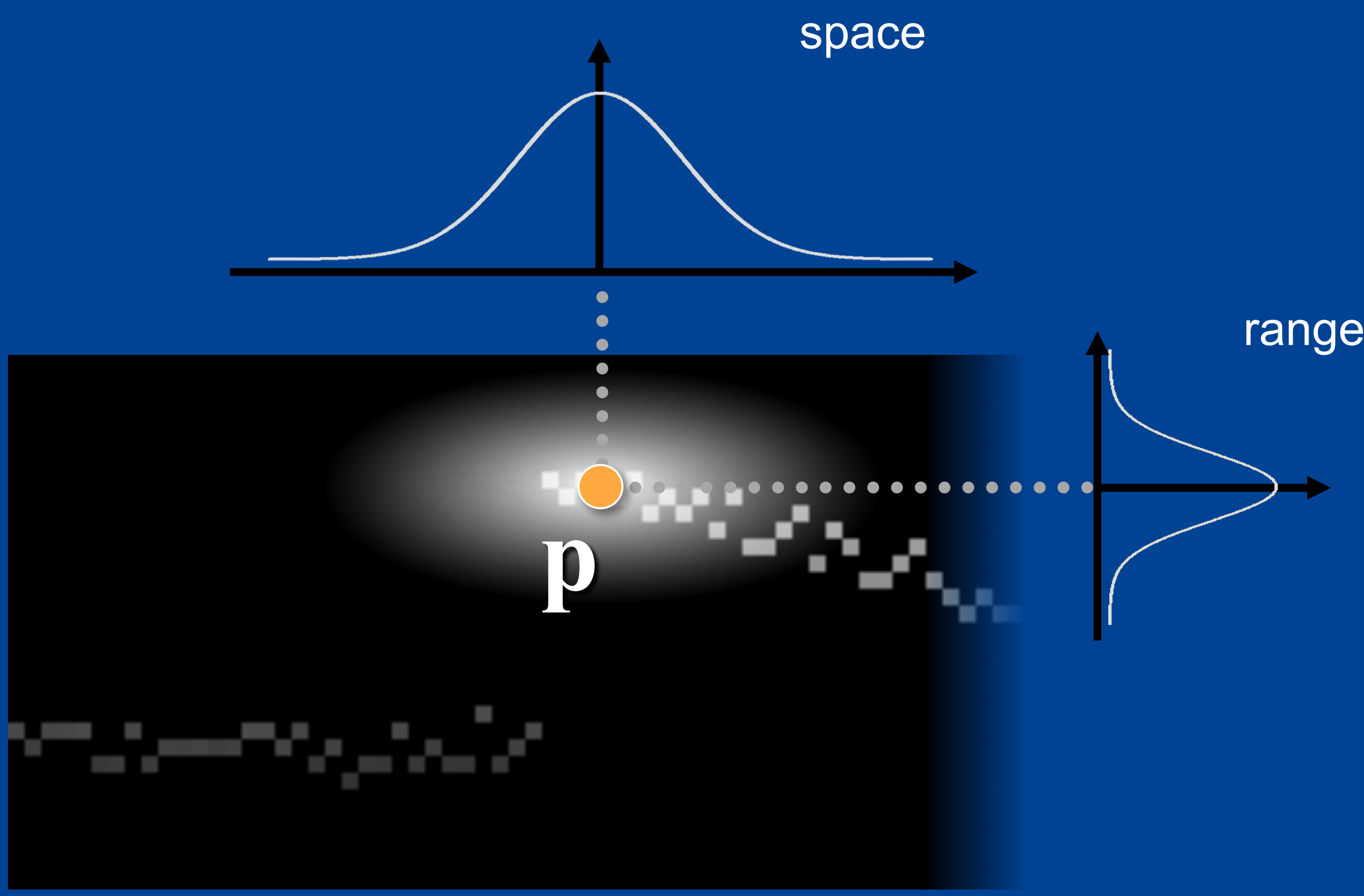
2nd Step: Higher-dimensional Space

- "Product of two Gaussians" = higher dim. Gaussian



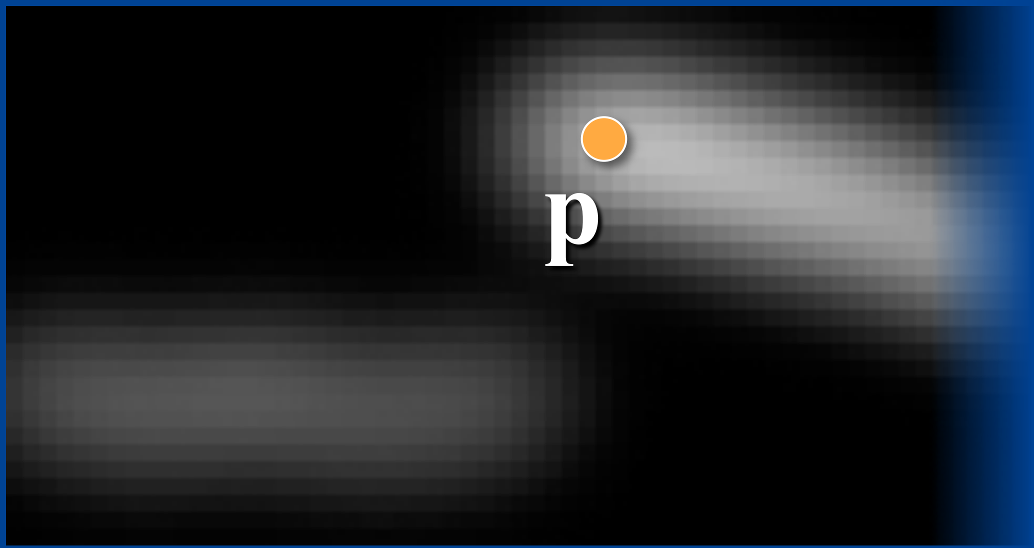
2nd Step: Higher-dimensional Space

- 0 almost everywhere, *I* at "plot location"



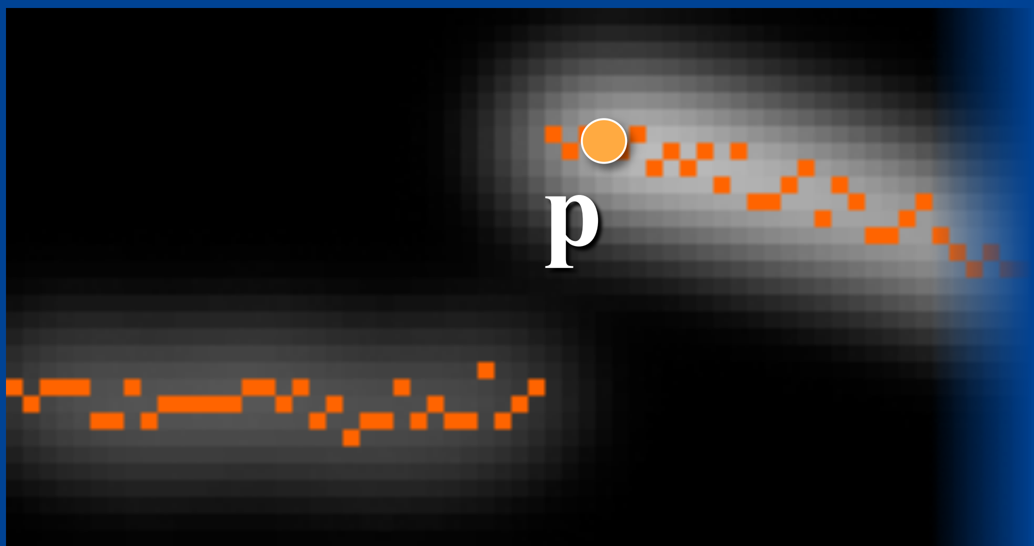
2nd Step: Higher-dimensional Space

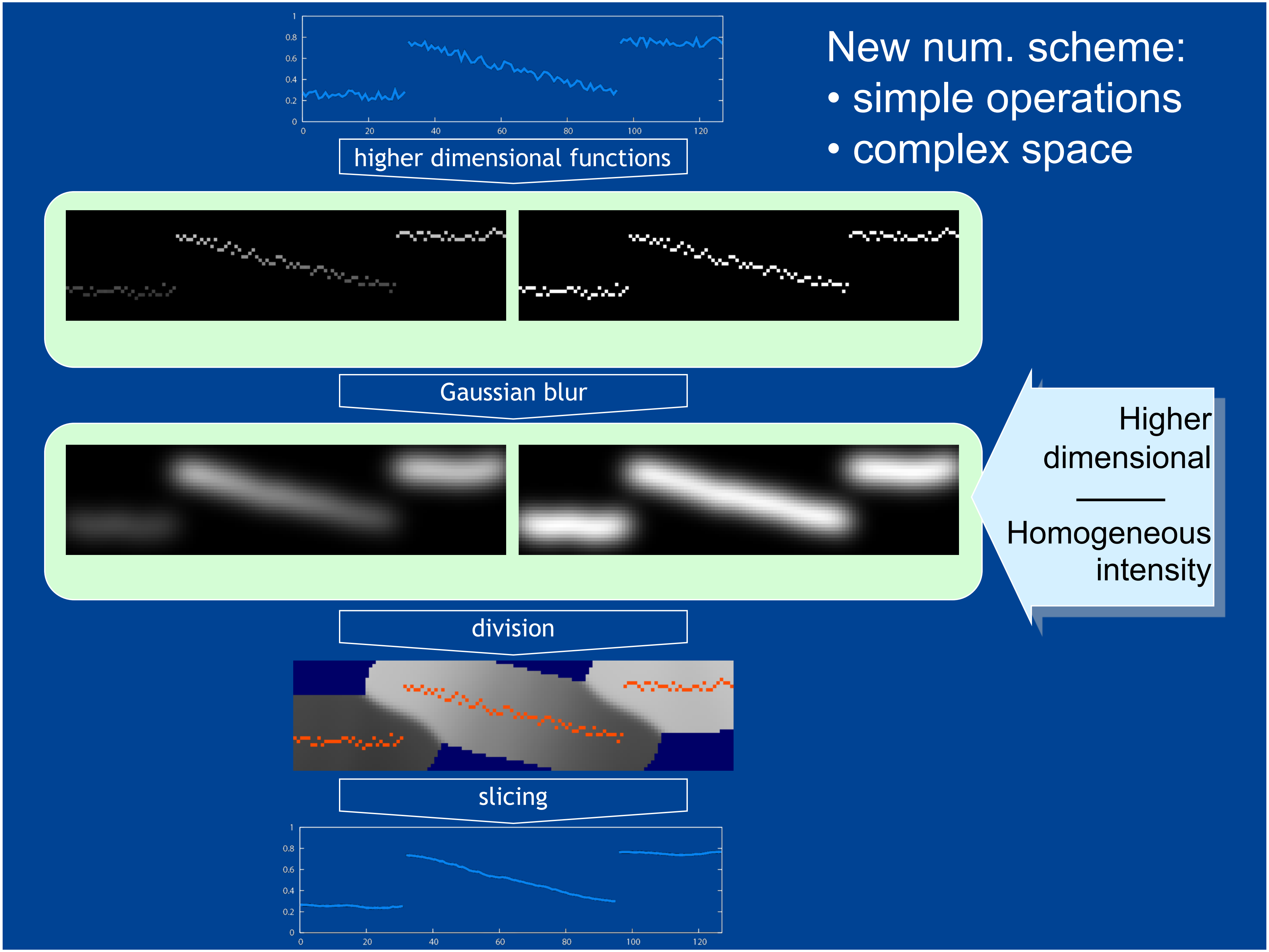
- 0 almost everywhere, I at "plot location"
- Weighted average at each point = Gaussian blur



2nd Step: Higher-dimensional Space

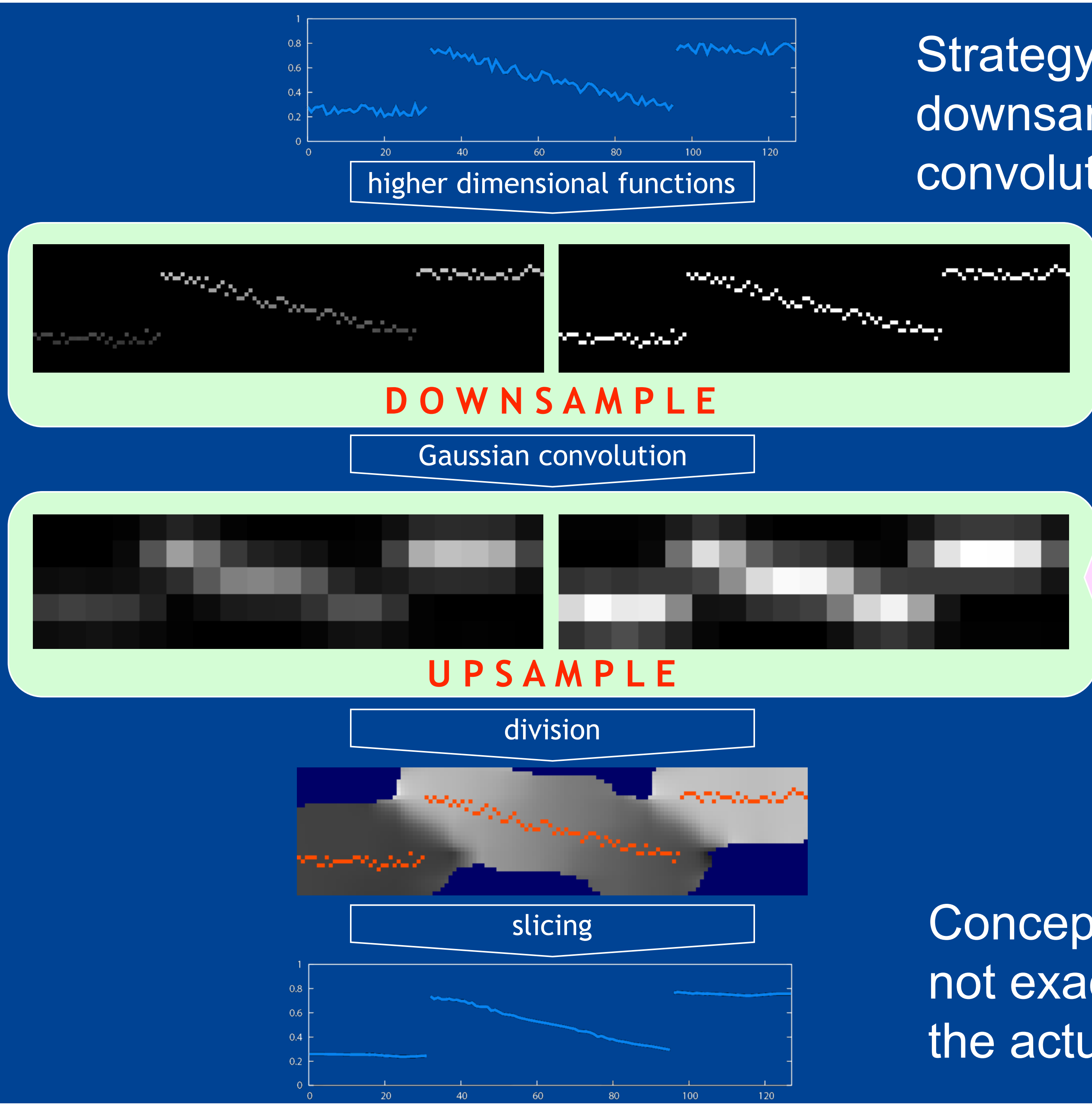
- 0 almost everywhere, I at "plot location"
- Weighted average at each point = Gaussian blur
- Result is at "plot location"

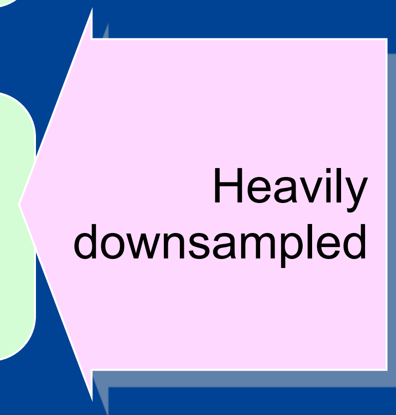




New num. scheme:
• simple operations
• complex space
higher dimensional functions
Gaussian blur
Higher dimensional
Homogeneous intensity
division
slicing

Strategy: downsampled convolution





Conceptual view, not exactly the actual algorithm

Actual Algorithm

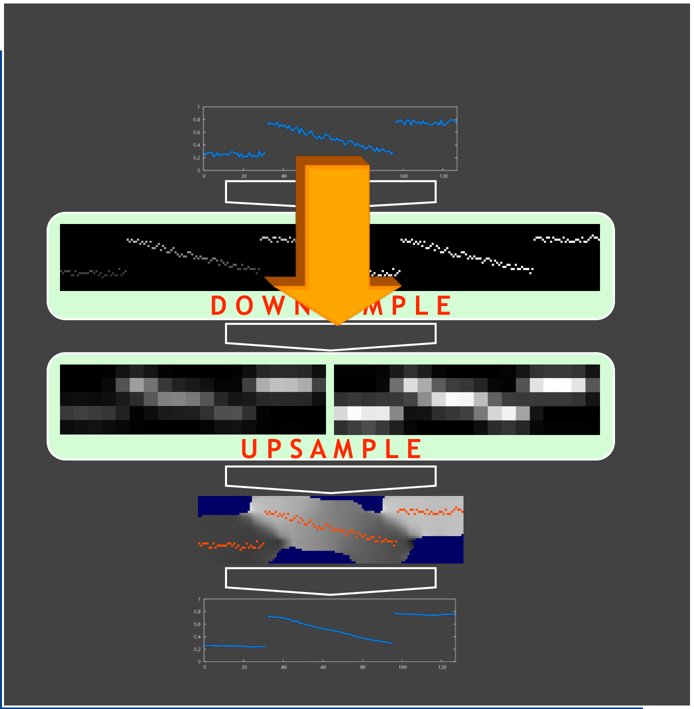
- Never compute full resolution
  - On-the-fly downsampling
  - On-the-fly upsampling

- 3D sampling rate = $(\sigma_s, \sigma_s, \sigma_r)$

Pseudo-code: Start

- Input
  - image I
  - Gaussian parameters σ_s and σ_r
- Output: $BF[\,I\,]$
- Data structure: 3D arrays wi and w (init. to 0)

Pseudo-code: On-the-fly Downsampling



- For each pixel $(X,Y) \in S$
  - Downsample:

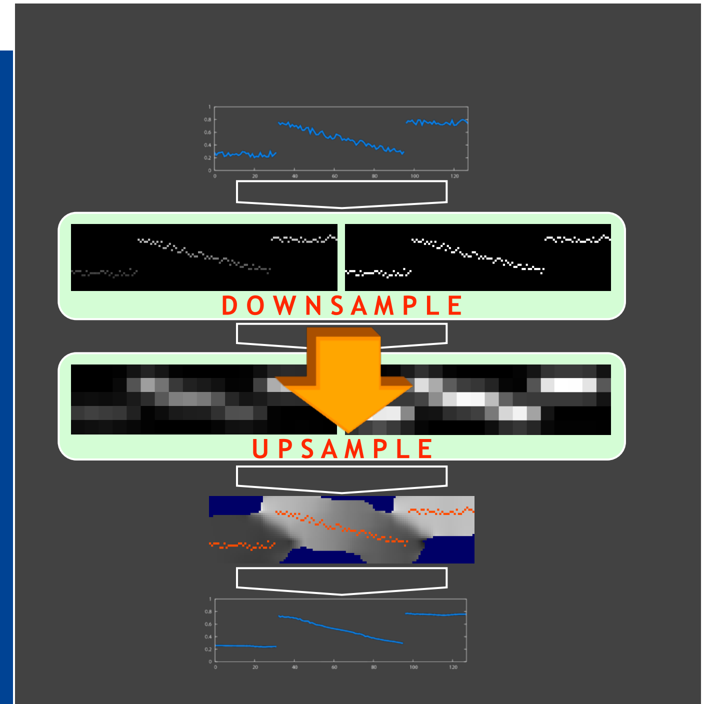
$$(x, y, z) = \left(\left[\frac{X}{\sigma_s} \right], \left[\frac{Y}{\sigma_s} \right], \left[\frac{I(X,Y)}{\sigma_r} \right] \right)$$

[ ] = closest int.

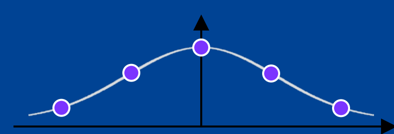
  - Update:

$$wi(x, y, z) \mathrel{+}= I(X,Y)$$
$$w(x, y, z) \mathrel{+}= 1$$

Pseudo-code: Convolving



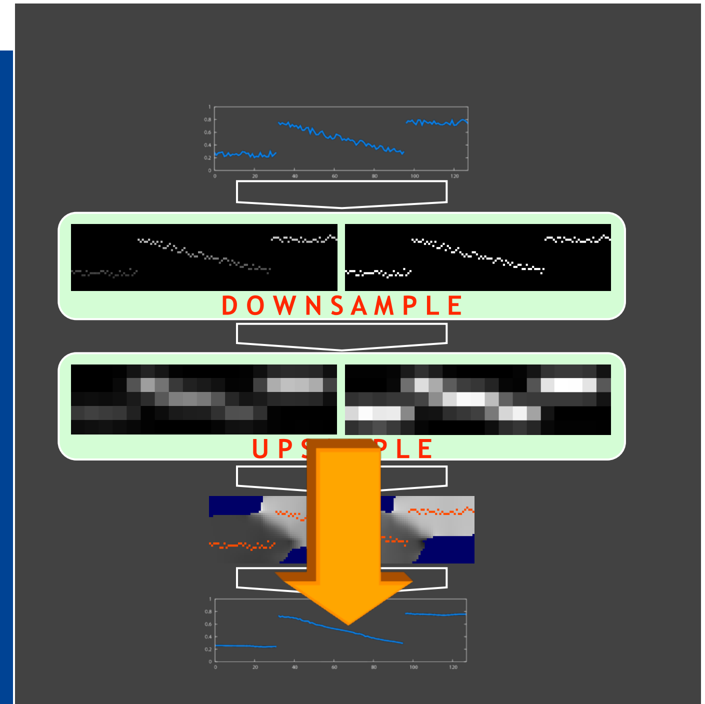
- For each axis $\vec{x}$, $\vec{y}$, and $\vec{z}$
  - For each 3D point (x, y, z)



    - Apply a Gaussian mask $(1, 4, 6, 4, 1)$ to wi and w
      e.g., for the x axis:

$$wi'(x) = wi(x-2) + 4.wi(x-1) + 6.wi(x) + 4.wi(x+1) + wi(x+2)$$

Pseudo-code: On-the-fly Upsampling



- For each pixel (X,Y) in S
  - Linearly interpolate the values in the 3D arrays

$$BF[I](X,Y) = \frac{\text{interpolate}(wi,\, X,\, Y,\, I(X,Y))}{\text{interpolate}(w,\, X,\, Y,\, I(X,Y))}$$

Discussion

- Complexity: $O\left(|S| + \frac{|S|}{\sigma_s^2}\frac{|R|}{\sigma_r}\right)$

  $|S|$: number of pixels

  $\frac{|S|}{\sigma_s^2}\frac{|R|}{\sigma_r}$: number of 3D cells

  $|R|$: number of gray levels

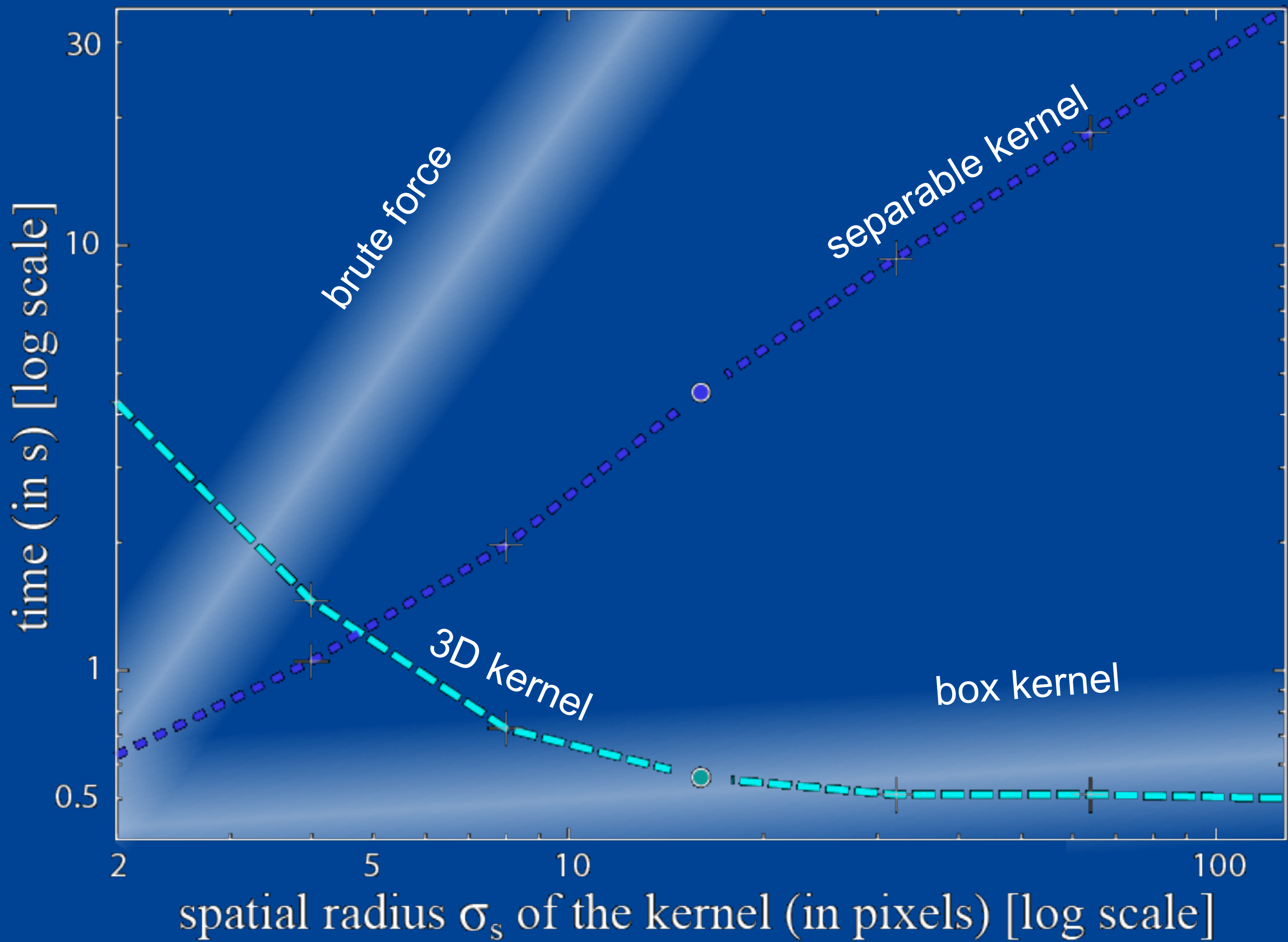
- Fast for medium and large kernels
  - Can be ported on GPU [Chen 07]: always very fast
- Can be extended to color images but slower
- Visually similar to brute-force computation


input


brute-force
implementation


3D kernel
visually similar

Running Times



How to Choose an Implementation?

Depends a lot on the application. A few guidelines:

- Brute-force: tiny kernels or if accuracy is paramount
- Box Kernel: for short running times on CPU with any kernel size, e.g. editing package
- 3D Kernel:
  - if GPU available
  - if only CPU available: large kernels, color images, cross BF (e.g., good for computational photography)
- Bilteral Pyramid [Fattal 07]: for multi-scale

Questions ?