CV 2026 – Week 1 – HC1b

Histograms
and
Point Operators

Dimitris Tzionas

d.tzionas@uva.nl, CC: e.a.veltmeijer@uva.nl
(please CC your TA)

Images

Recap

How can we store a continuous image?

Discretize
- Image range → **Quantization**
- Spatial domain → **Sampling**

Per pixel: 1 byte / 8 bit / uint8

1	2	3	4	5	6	7	8

Range: 0 … 255

We can store a total of $2^8 = 256$ values

3 byte / 24 bit

8 bit Red	8 bit Green	8 bit Blue

8 bit Blue	8 bit Green	8 bit Red

OpenCV → BGR

⚠ Make sure numbers can be represented properly!

- For ***processing*** switch to float *
- For ***storing*** & ***displaying*** switch to uint8

(*) Avoid ***overflow***!!! Mind the ***execution order*** (left→right)

```
g =         (f - f.min()) / (f.max()-f.min()) * 255
```

~~`g = 255 * (f - f.min()) / (f.max()-f.min())`~~

Images

Recap

How can we store a continuous image?

Discretize
- Image range → **Quantization**
- Spatial domain → **Sampling**

Per pixel: 1 byte / 8 bit / uint8

1	2	3	4	5	6	7	8

Range: 0 … 255

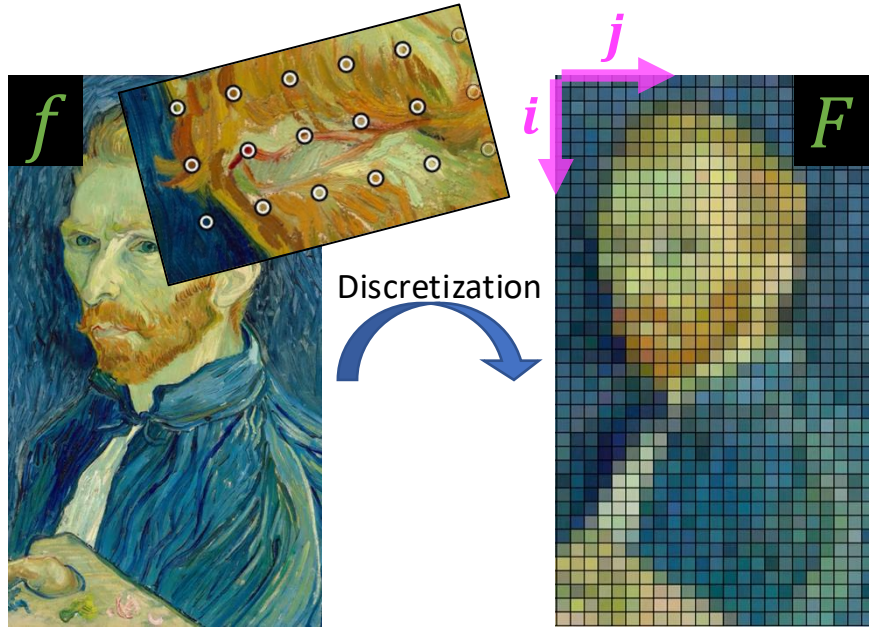
We can store a total of $2^8 = 256$ values

3 byte / 24 bit

8 bit Red	8 bit Green	8 bit Blue

8 bit Blue	8 bit Green	8 bit Red

OpenCV → BGR



Pixels: Sampling 'probes' in grid over continuous f

$$F(\overset{\text{row}}{i}, \overset{\text{column}}{j}) = f(i\Delta y, j\Delta x)$$

Distance between samples ($\Delta x = \Delta y = 1$)

Images

Recap

How can we store a continuous image?

Discretize
- Image range → **Quantization**
- Spatial domain → **Sampling**

Per pixel: 1 byte / 8 bit / uint8

1	2	3	4	5	6	7	8

Range: 0 … 255

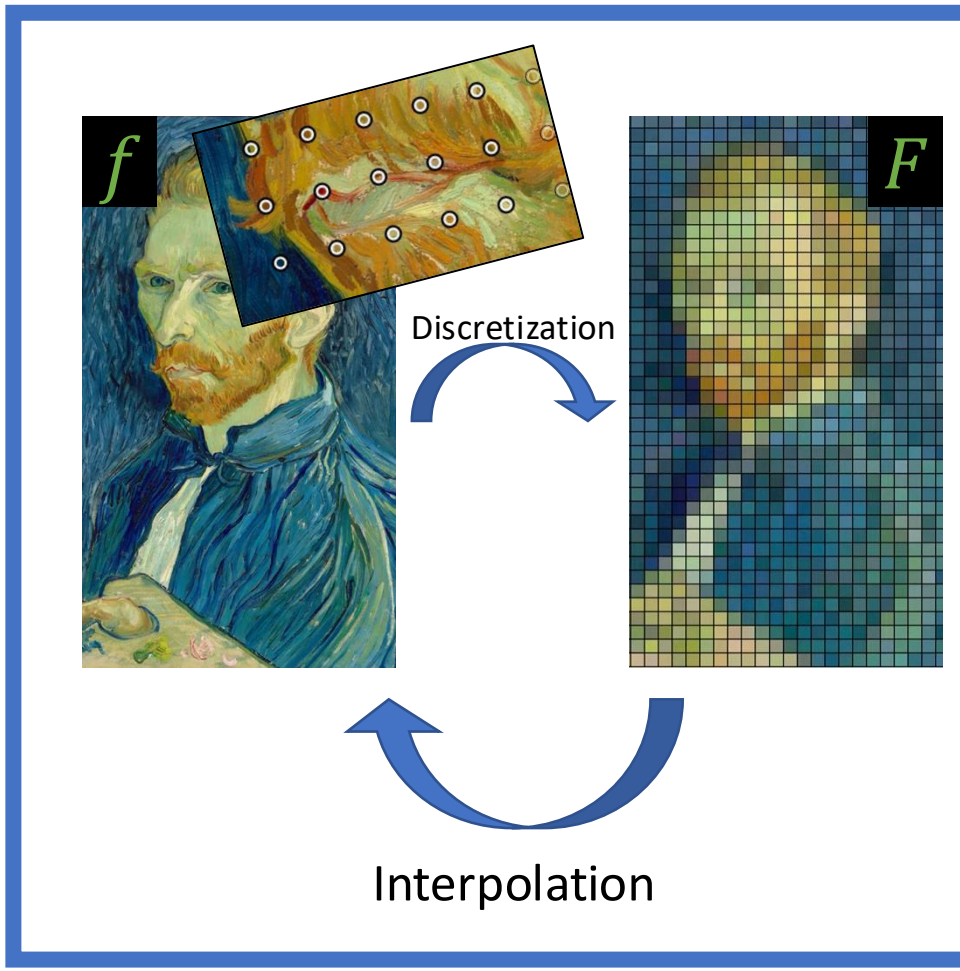
We can store a total of $2^8 = 256$ values

3 byte / 24 bit

8 bit Red	8 bit Green	8 bit Blue

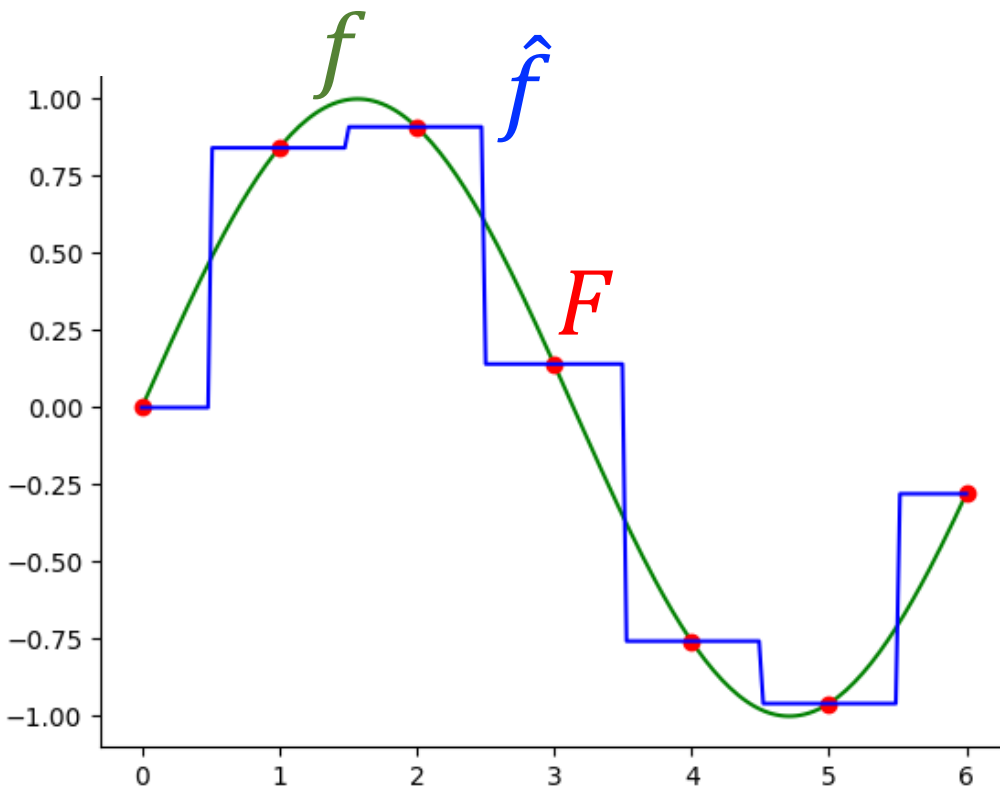
8 bit Blue	8 bit Green	8 bit Red

OpenCV → BGR



1D Interpolation

Goal



Input:

$F(x), x \in \mathbb{Z}$ → Samples of f

Goal – 'reconstruct':

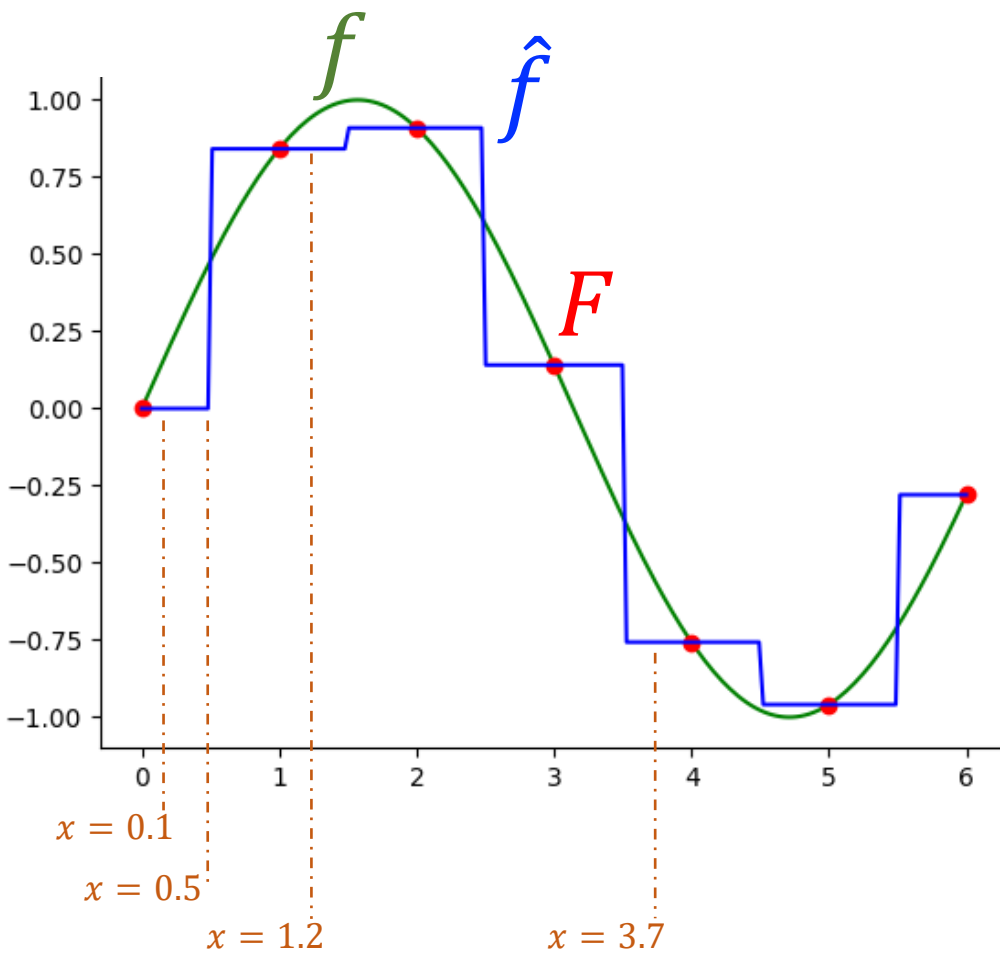
$f(x) = \sin(x),\ x \in \mathbb{R}$ → Continuous '*target*' function

… by approximating it via $\hat{f}$

Infer $\hat{f} \approx f$ given samples F

Images

1D Interpolation - Nearest Neighbor



$$\hat{f}(x) = F(\lfloor x + 0.5 \rfloor)$$ → Copy nearest sample

programmer's view: $round(x)$

'Floor' function

$\lfloor x \rfloor$ → largest int that is $\leq x$

$$\hat{f}(0.1) = F(\lfloor 0.1 + 0.5 \rfloor) = F(\lfloor 0.6 \rfloor) = F(0)$$

$$\hat{f}(0.5) = F(\lfloor 0.5 + 0.5 \rfloor) = F(\lfloor 1.0 \rfloor) = F(1)$$

$$\hat{f}(1.2) = F(\lfloor 1.2 + 0.5 \rfloor) = F(\lfloor 1.7 \rfloor) = F(1)$$

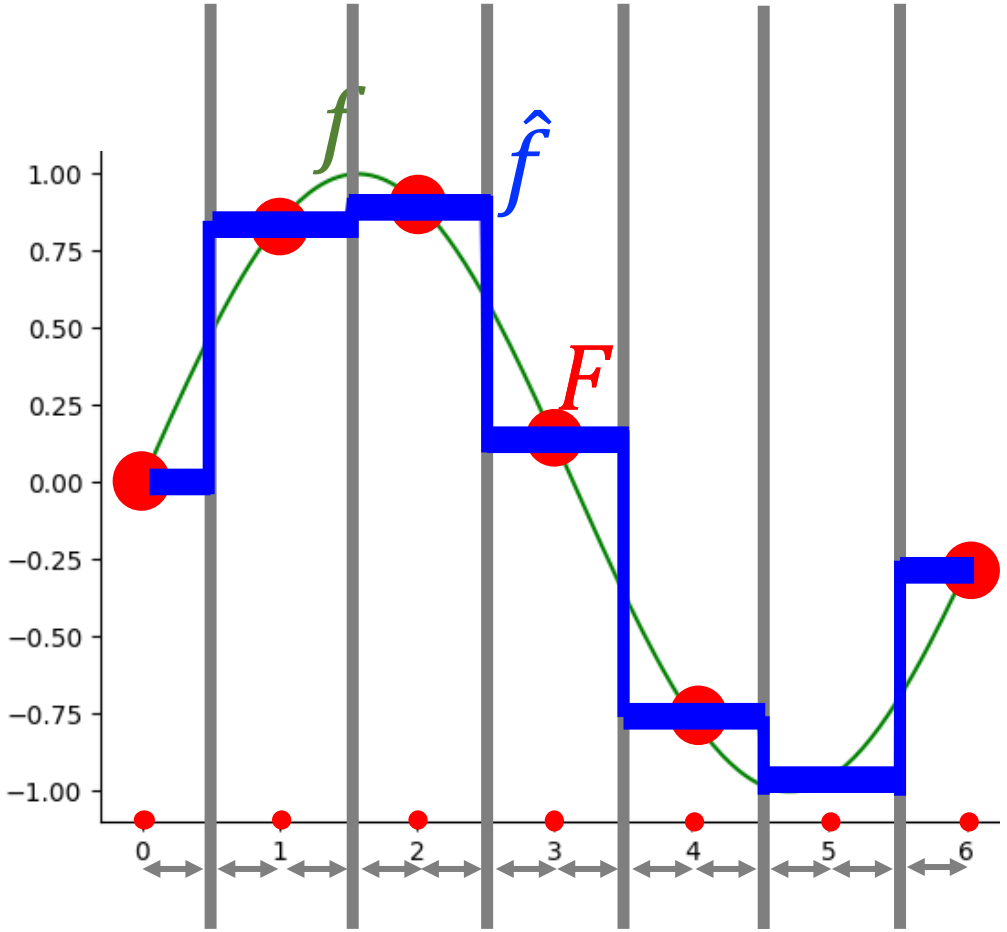
$$\hat{f}(3.7) = F(\lfloor 3.7 + 0.5 \rfloor) = F(\lfloor 4.2 \rfloor) = F(4)$$

?

$\hat{f}$ could be be computed for each point x independently! Can be heavily parallelized!

1D Interpolation

Nearest Neighbor



$$\hat{f}(x) = F(\lfloor x + 0.5 \rfloor)$$

programmer's view: $round(x)$

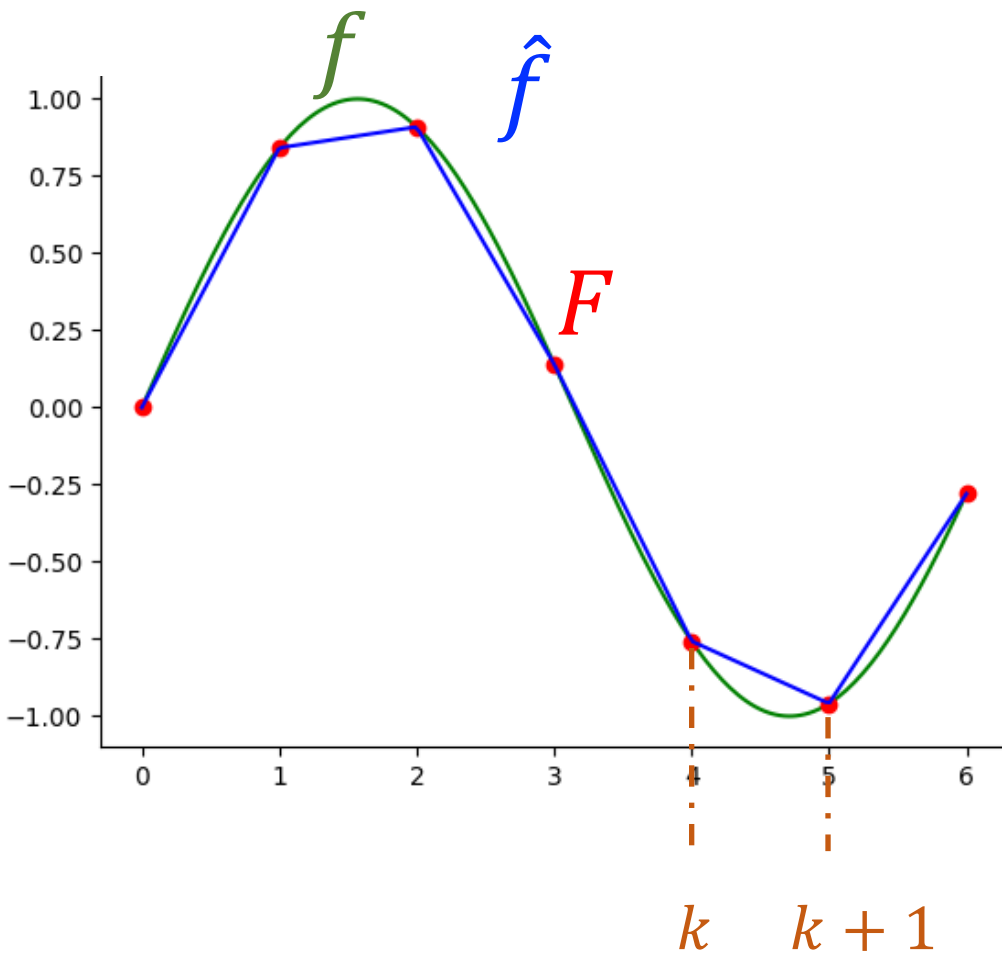
'Floor' function

$\lfloor x \rfloor$ → largest int that is $\leq x$

$\hat{f}$ → **'Staircase'** ~~**Continuous**~~ ~~**Differentiable**~~

1D Interpolation

Linear



Fit a **line** equation (y=ax+b) ***separately*** for each *pair* of *consecutive samples* $F(k)$ & $F(k+1)$

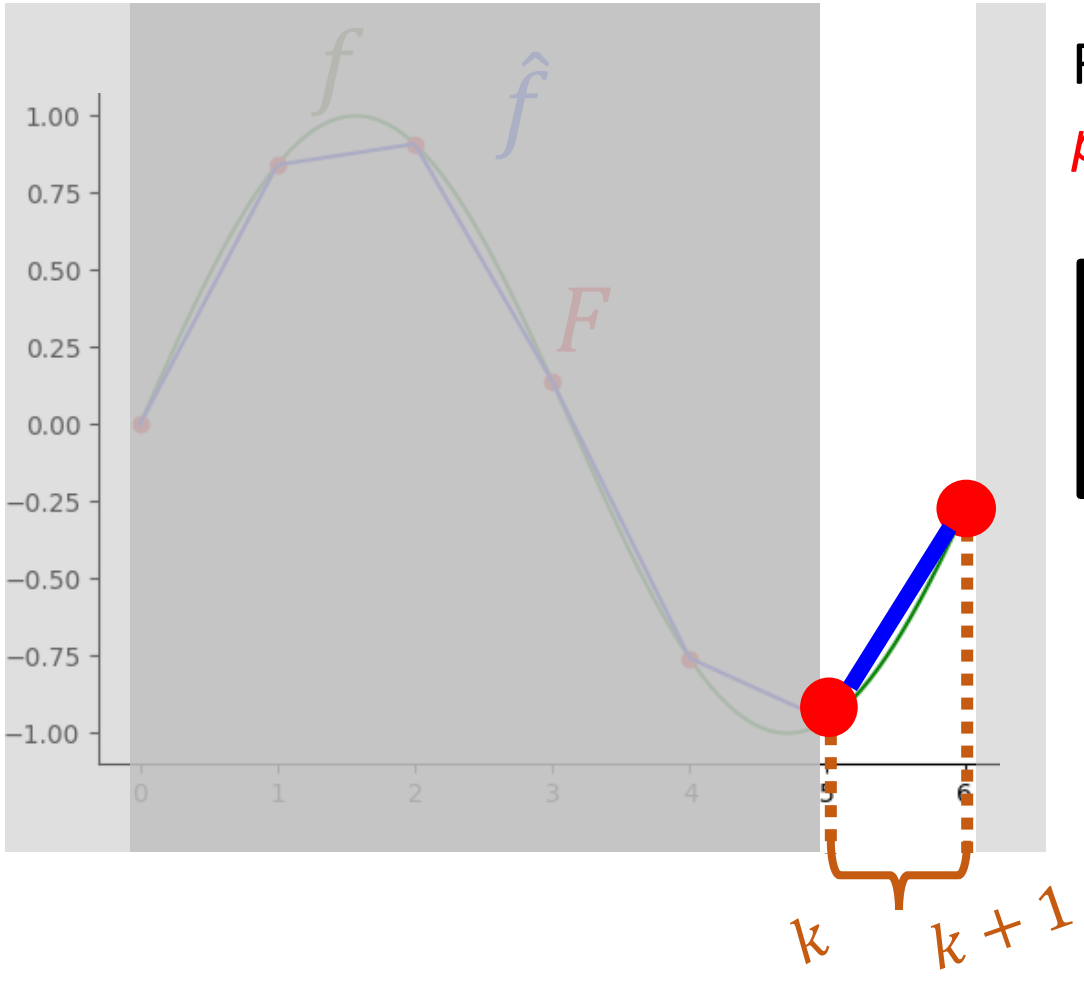
$$\hat{f}(x) = (1-(x-k))F(k) + (x-k)\,F(k+1)$$

steering / mixing weights

samples (input)

1D Interpolation

Linear

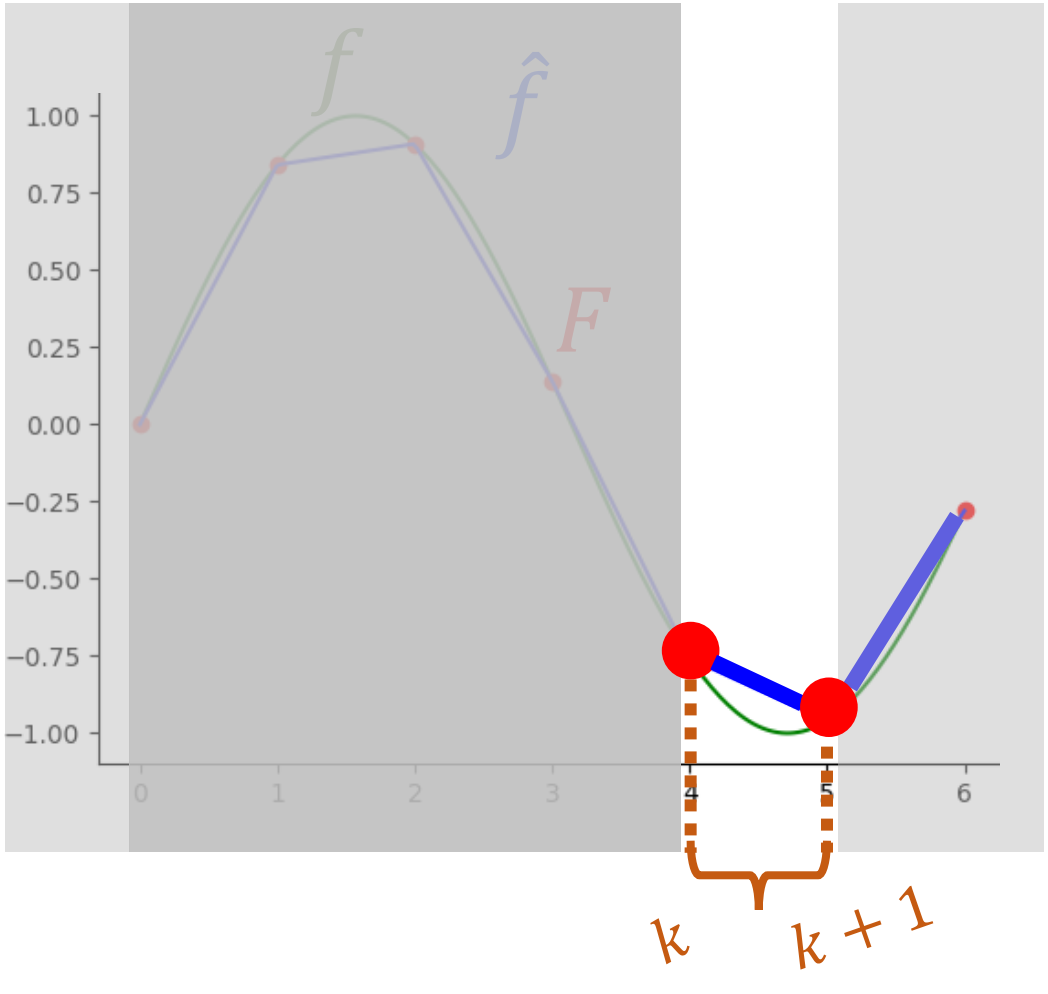


Fit a **line** equation (y=ax+b) ***separately*** for each *pair* of *consecutive samples* $F(k)$ & $F(k+1)$

$$\hat{f}(x) = (1-(x-k))F(k) + (x-k)\,F(k+1)$$

1D Interpolation

Linear

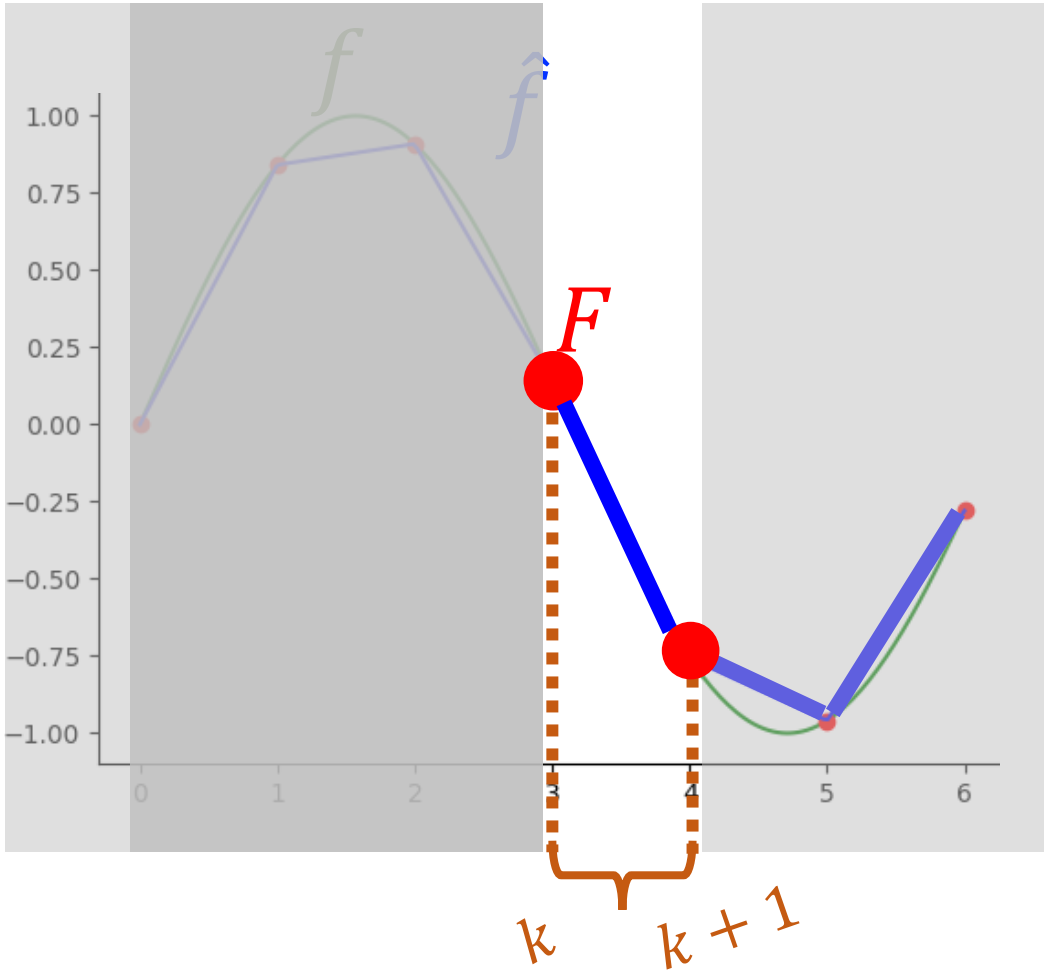


Fit a **line** equation (y=ax+b) ***separately*** for each *pair* of *consecutive samples* $F(k)$ & $F(k+1)$

$$\hat{f}(x) = (1-(x-k))F(k) + (x-k)\,F(k+1)$$

1D Interpolation

Linear

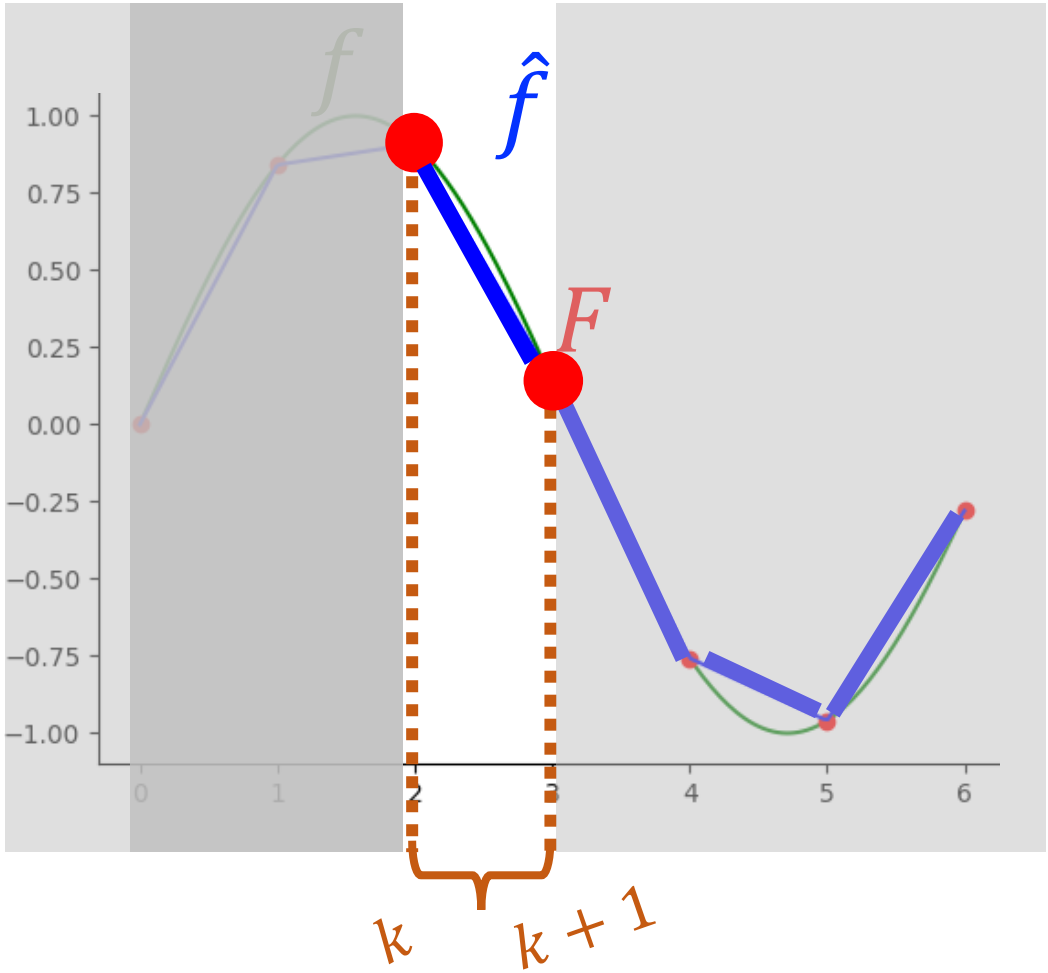


Fit a **line** equation (y=ax+b) ***separately*** for each *pair* of *consecutive samples* $F(k)$ & $F(k+1)$

$$\hat{f}(x) = (1-(x-k))F(k) + (x-k)\,F(k+1)$$

1D Interpolation

Linear

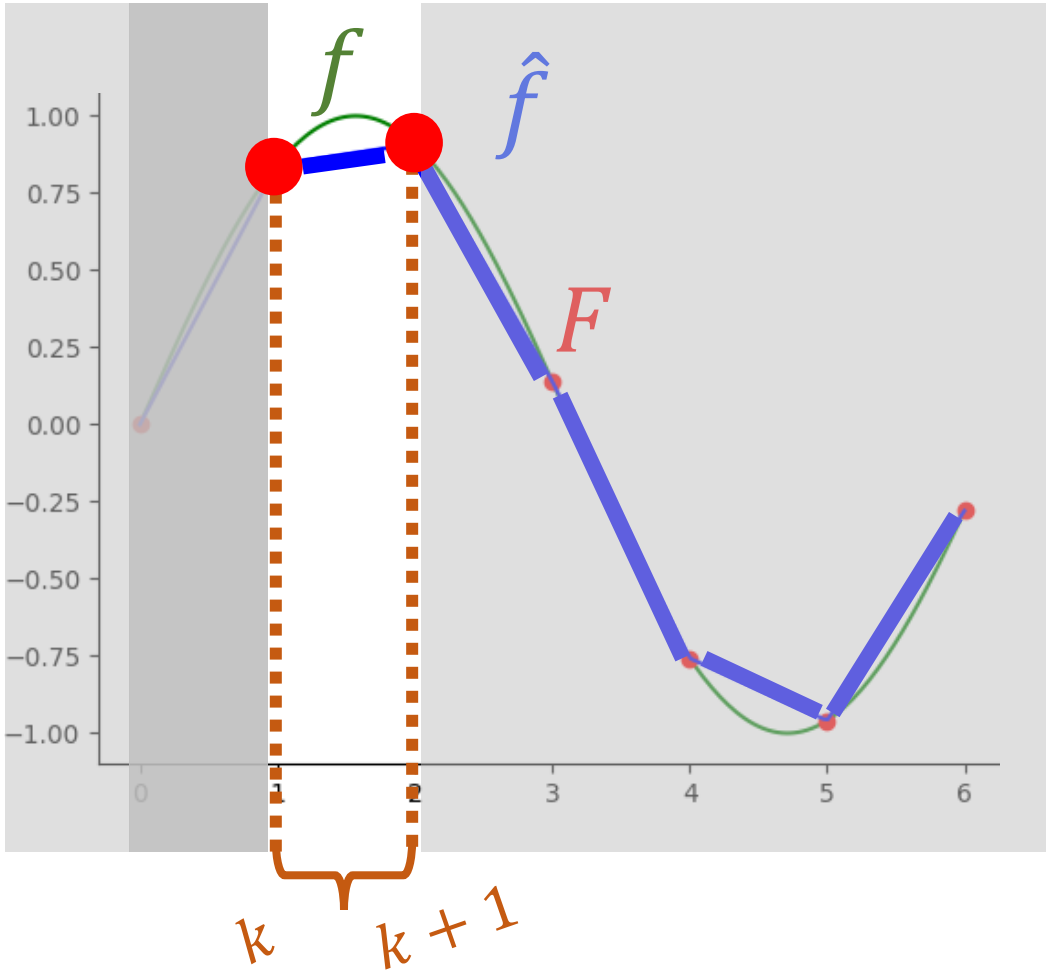


Fit a **line** equation (y=ax+b) ***separately*** for each *pair* of *consecutive samples* $F(k)$ & $F(k+1)$

$$\hat{f}(x) = (1-(x-k))F(k) + (x-k)\,F(k+1)$$

1D Interpolation

Linear

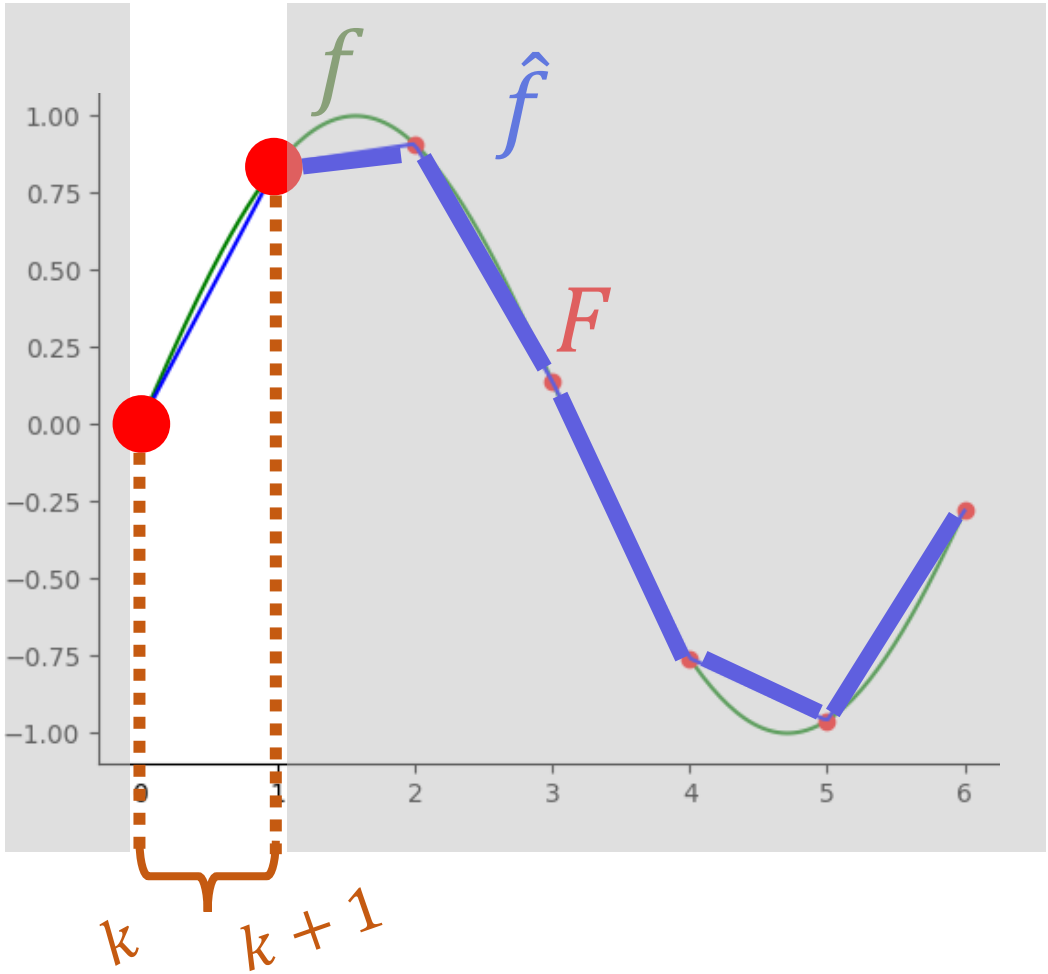


Fit a **line** equation (y=ax+b) ***separately*** for each *pair* of *consecutive samples* $F(k)$ & $F(k+1)$

$$\hat{f}(x) = (1-(x-k))F(k) + (x-k)\,F(k+1)$$

1D Interpolation

Linear

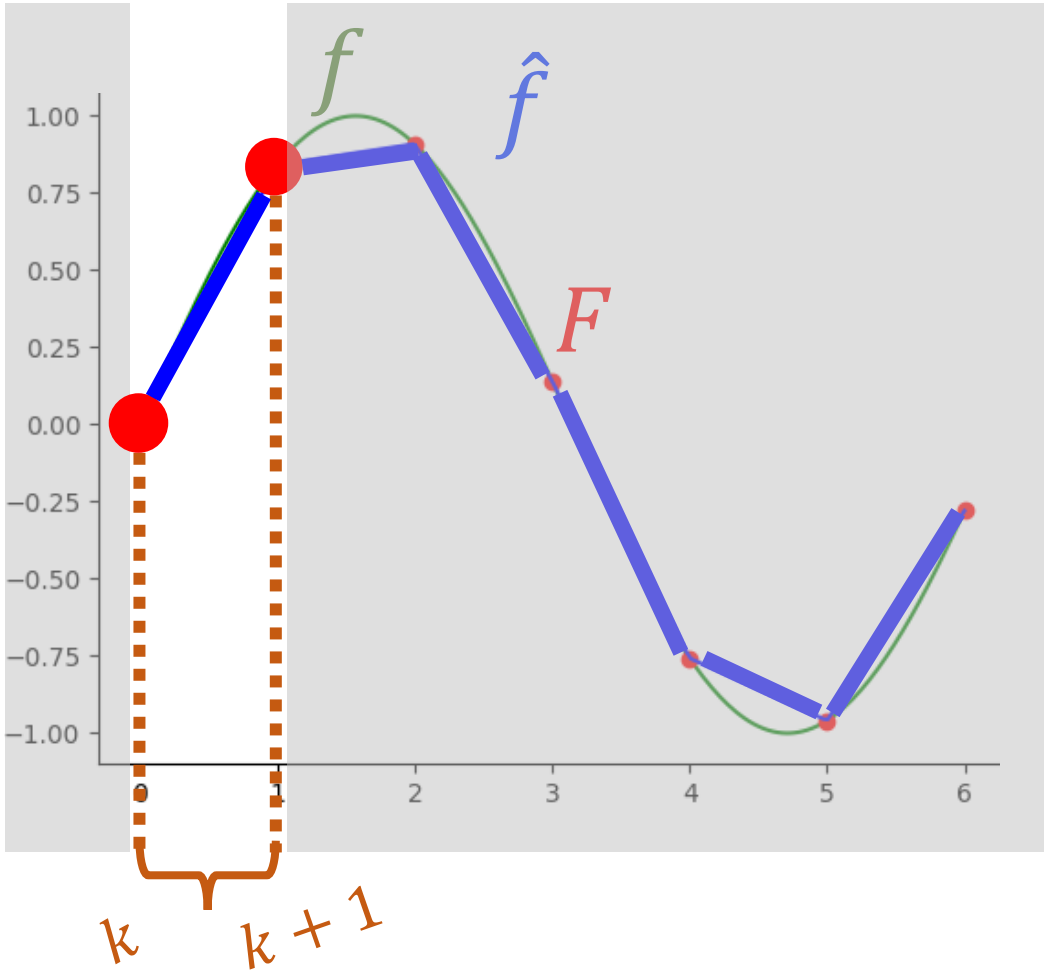


Fit a **line** equation (y=ax+b) ***separately*** for each *pair* of *consecutive samples* $F(k)$ & $F(k+1)$

$$\hat{f}(x) = (1-(x-k))F(k) + (x-k)\,F(k+1)$$

1D Interpolation

Linear



Fit a **line** equation (y=ax+b) ***separately*** for each *pair* of *consecutive samples* $F(k)$ & $F(k+1)$

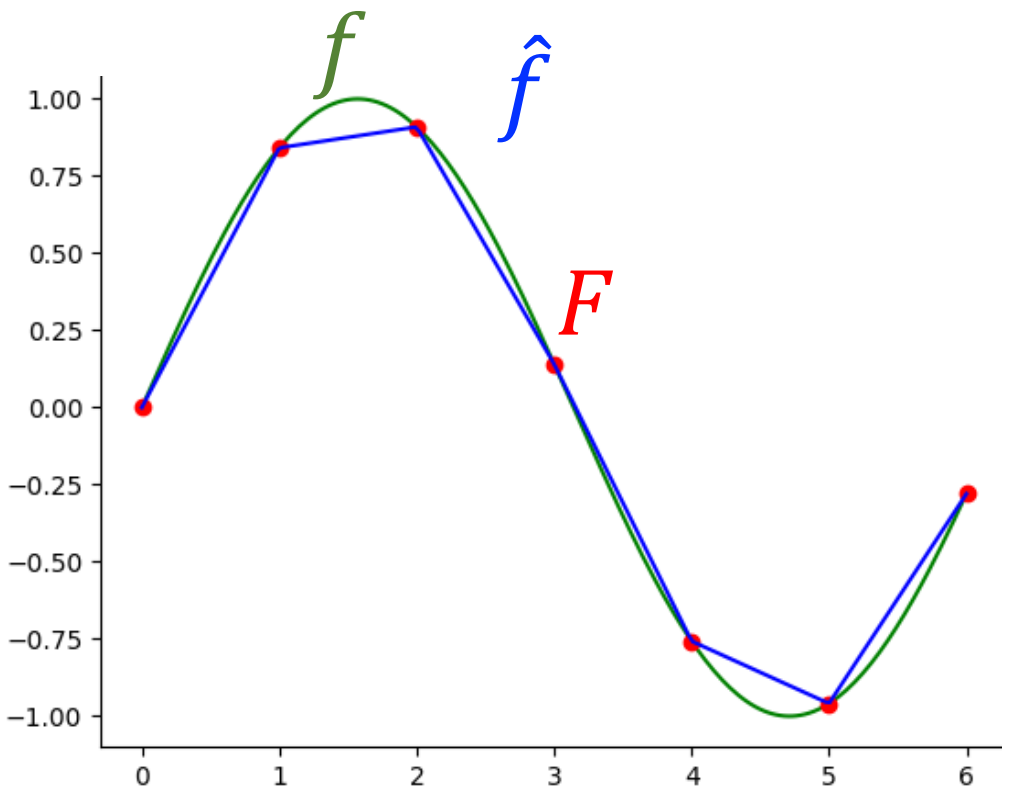
$$\hat{f}(x) = (1-(x-k))F(k) + (x-k)\,F(k+1)$$

steering / mixing weights — *samples*

Each line segment of $\hat{f}$ could be computed independently from other segments – in parallel!

1D Interpolation

Linear



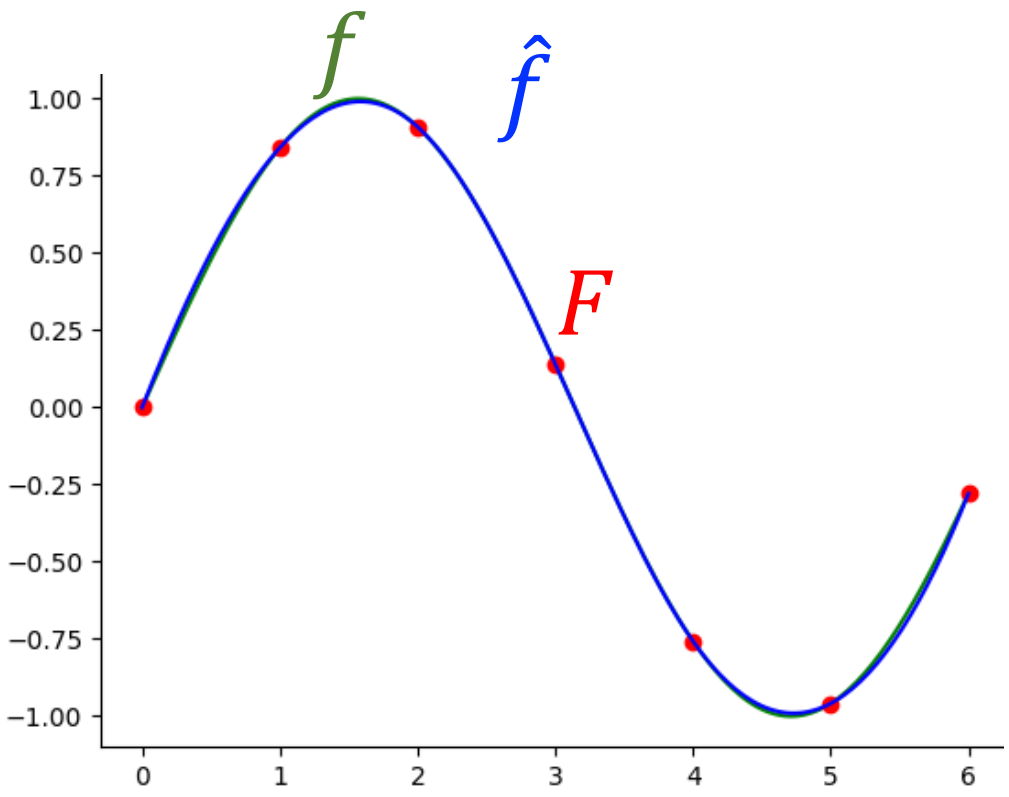
$$\hat{f}(x) = (1 - (x - k))F(k) + (x - k)\,F(k+1)$$

$\hat{f}$ → **Piece-wise linear** **Continuous**

- **not differentiable @ sample point locations**
- **high-order deriv. = 0**

1D Interpolation

Cubic



Segment ***between*** points: $x \in [k, k+1]$

For 4x samples
incl. 2x *outside* range!

$$\begin{cases} x = k & x = k-1 \\ x = k+1 & x = k+2 \end{cases}$$

$$\hat{f}(x) = a(x-k)^3 + b(x-k)^2 + c(x-k)^1 + d$$

$$\begin{aligned} F(k) &= d \\ F(k-1) &= a(-1)^3 + b(-1)^2 + c(-1) + d \\ F(k+1) &= a(+1)^3 + b(+1)^2 + c(+1) + d \\ F(k+2) &= a(+2)^3 + b(+2)^2 + c(+2) + d \end{aligned}$$

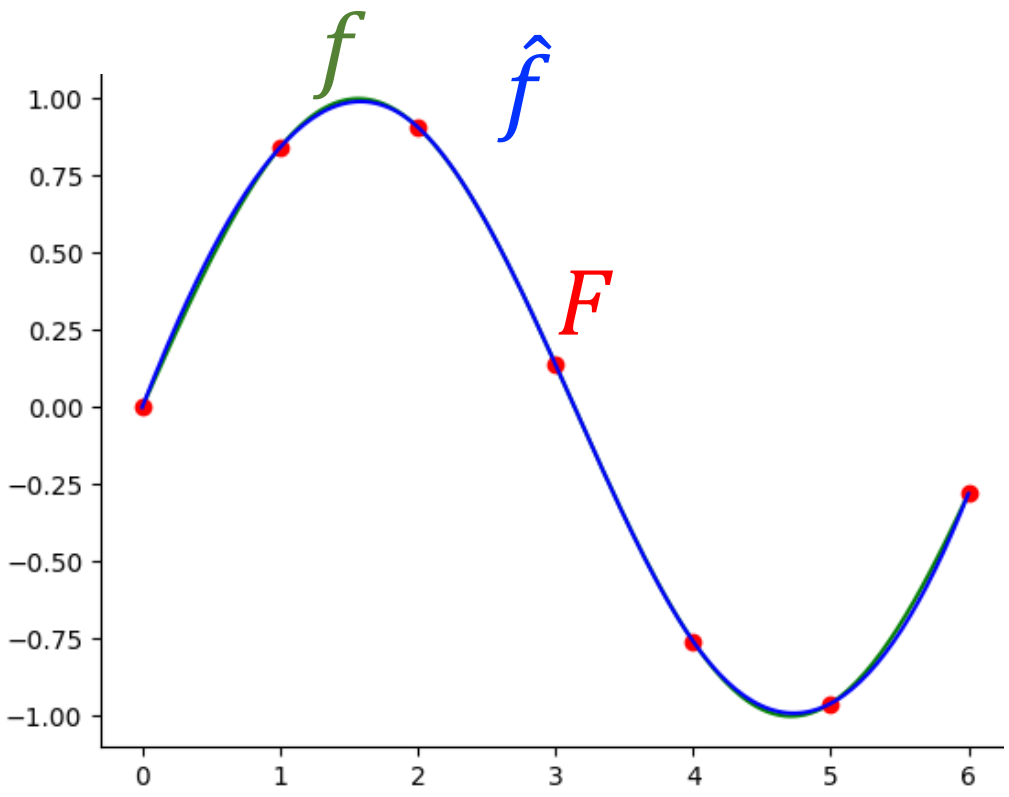
4 equations *linear* for **4 unknowns**

Input samples (known)

Unknown params to solve for

1D Interpolation

Cubic



Segment ***between*** points: $x \in [k, k+1]$

For 4x samples
incl. 2x *outside* range!

$$\begin{cases} x = k & x = k-1 \\ x = k+1 & x = k+2 \end{cases}$$

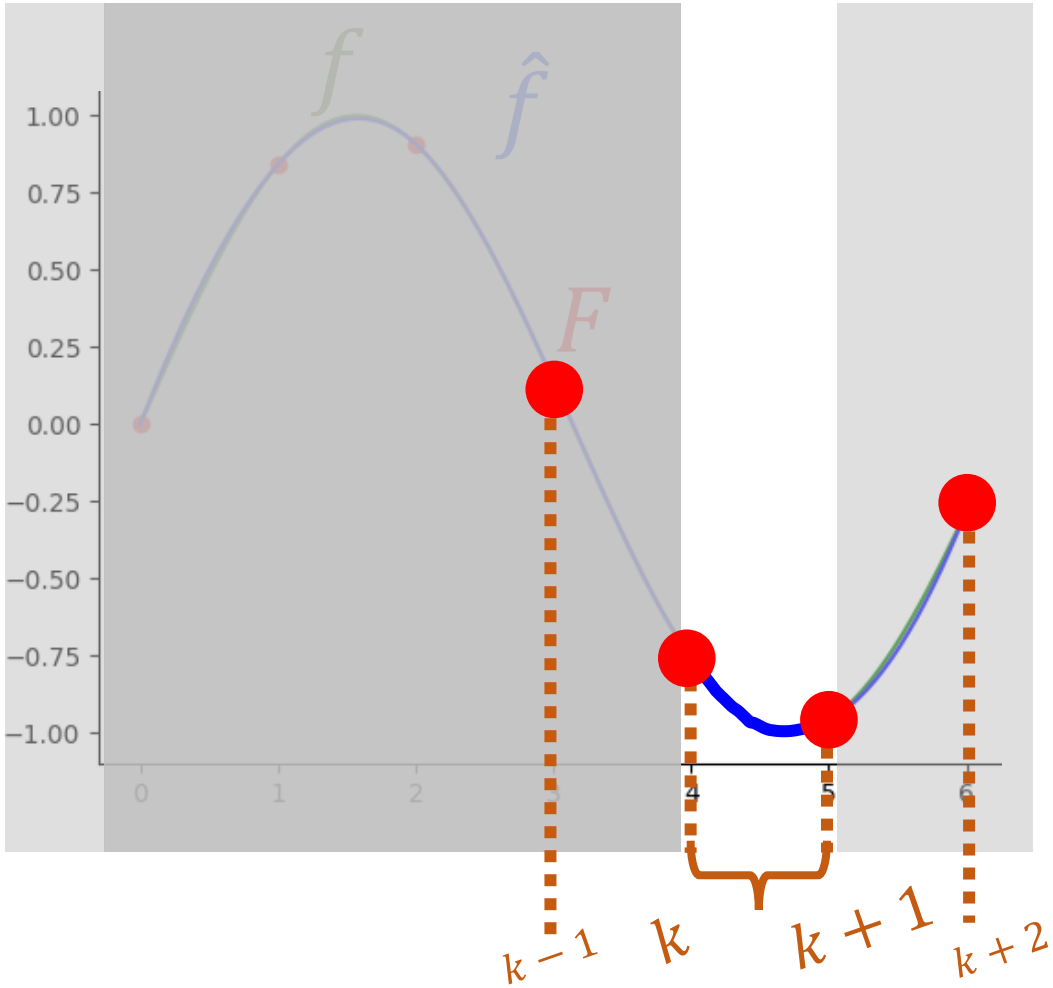
$$\hat{f}(x) = a(x-k)^3 + b(x-k)^2 + c(x-k)^1 + d$$

$$a = {}^1/_6\left(-F(k-1) + 3F(k) - 3F(k+1) + F(k+2)\right)$$
$$b = {}^1/_2\left(F(k-1) - 2F(k) + F(k+1)\right)$$
$$c = {}^1/_6\left(-2F(k-1) - 3F(k) + 6F(k+1) - F(k+2)\right)$$
$$d = F(k)$$

Solution for unknowns

1D Interpolation

Cubic



Segment ***between*** points: $x \in [k, k+1]$

For 4x samples
incl. 2x *outside* range!

$$\begin{cases} x = k & x = k-1 \\ x = k+1 & x = k+2 \end{cases}$$

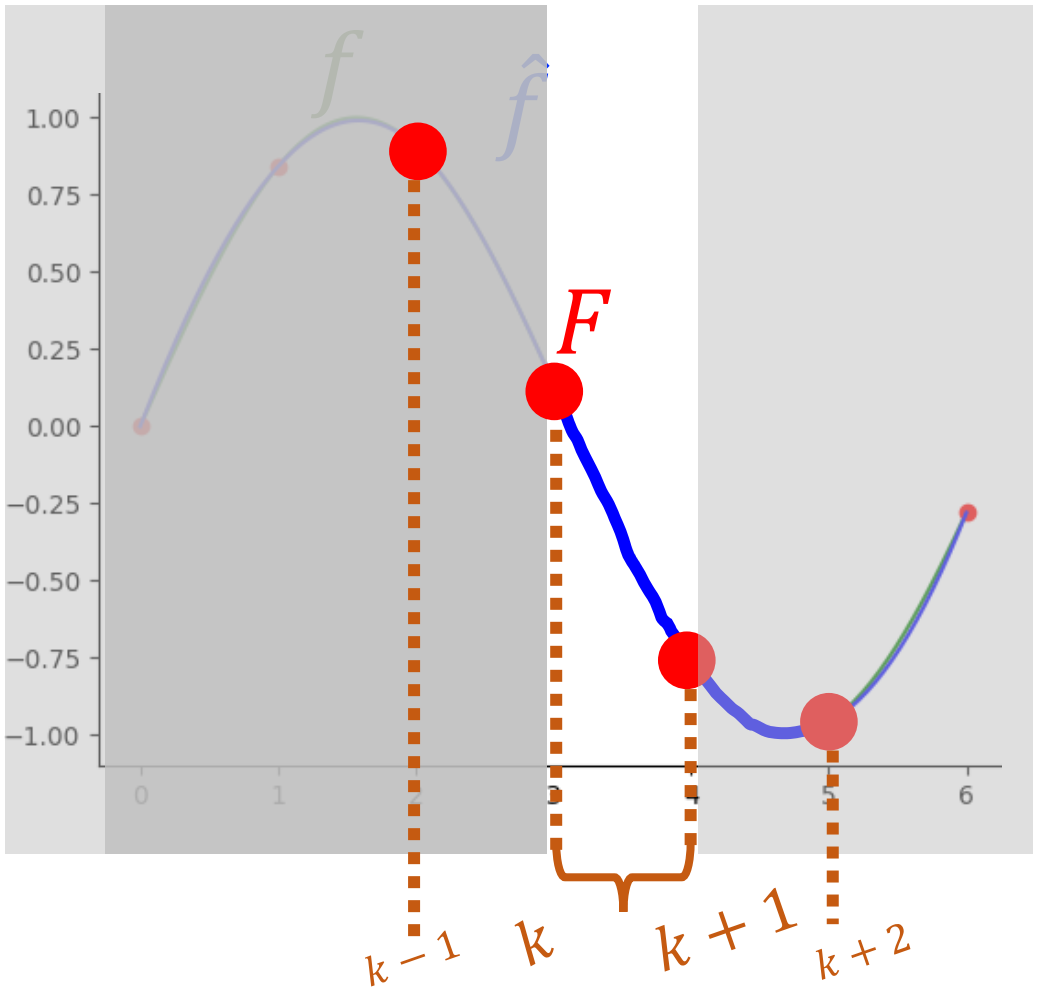
$$\hat{f}(x) = a(x-k)^3 + b(x-k)^2 + c(x-k)^1 + d$$

$$a = {}^1/_6\left(-F(k-1) + 3F(k) - 3F(k+1) + F(k+2)\right)$$
$$b = {}^1/_2\left(F(k-1) - 2F(k) + F(k+1)\right)$$
$$c = {}^1/_6\left(-2F(k-1) - 3F(k) + 6F(k+1) - F(k+2)\right)$$
$$d = F(k)$$

Solution for unknowns

1D Interpolation

Cubic



Segment ***between*** points: $x \in [k, k+1]$

For 4x samples
incl. 2x *outside* range!

$$\begin{cases} x = k & x = k-1 \\ x = k+1 & x = k+2 \end{cases}$$

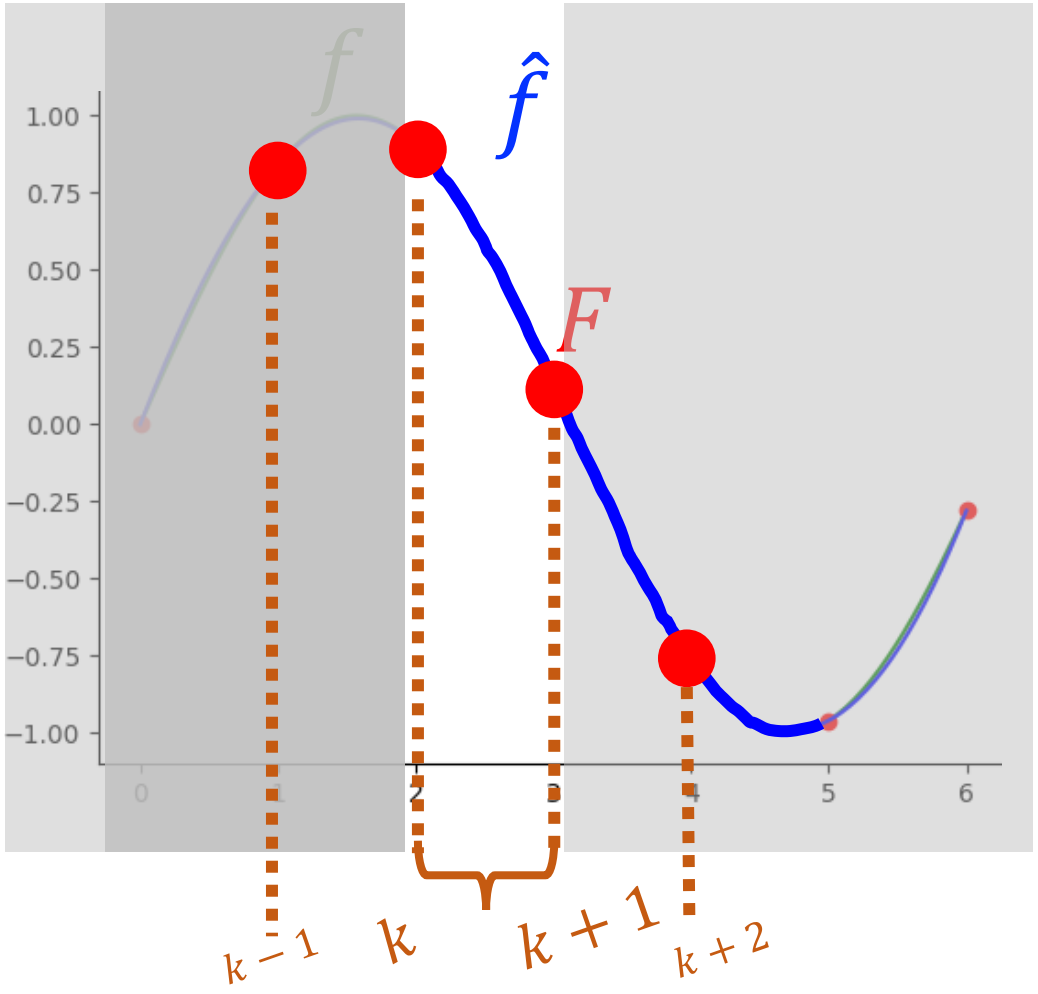
$$\hat{f}(x) = a(x-k)^3 + b(x-k)^2 + c(x-k)^1 + d$$

$$a = {}^1/_6\left(-F(k-1) + 3F(k) - 3F(k+1) + F(k+2)\right)$$
$$b = {}^1/_2\left(F(k-1) - 2F(k) + F(k+1)\right)$$
$$c = {}^1/_6\left(-2F(k-1) - 3F(k) + 6F(k+1) - F(k+2)\right)$$
$$d = F(k)$$

Solution for unknowns

1D Interpolation

Cubic



Segment ***between*** points: $x \in [k, k+1]$

For 4x samples
incl. 2x *outside* range!

$$\begin{cases} x = k & x = k-1 \\ x = k+1 & x = k+2 \end{cases}$$

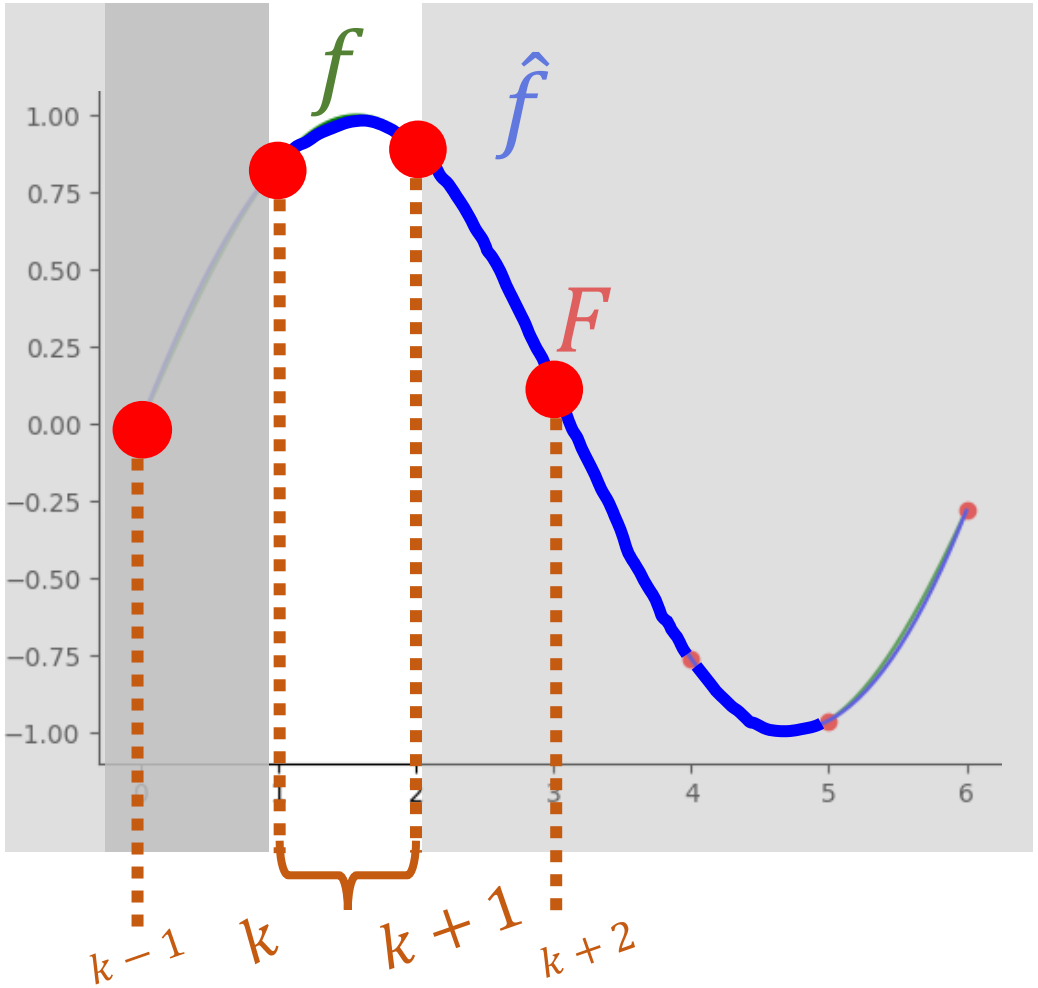
$$\hat{f}(x) = a(x-k)^3 + b(x-k)^2 + c(x-k)^1 + d$$

$$a = {}^1/_6\left(-F(k-1) + 3F(k) - 3F(k+1) + F(k+2)\right)$$
$$b = {}^1/_2\left(F(k-1) - 2F(k) + F(k+1)\right)$$
$$c = {}^1/_6\left(-2F(k-1) - 3F(k) + 6F(k+1) - F(k+2)\right)$$
$$d = F(k)$$

Solution for unknowns

1D Interpolation

Cubic



Segment ***between*** points: $x \in [k, k+1]$

For 4x samples
incl. 2x *outside* range!

$$\begin{cases} x = k & x = k-1 \\ x = k+1 & x = k+2 \end{cases}$$

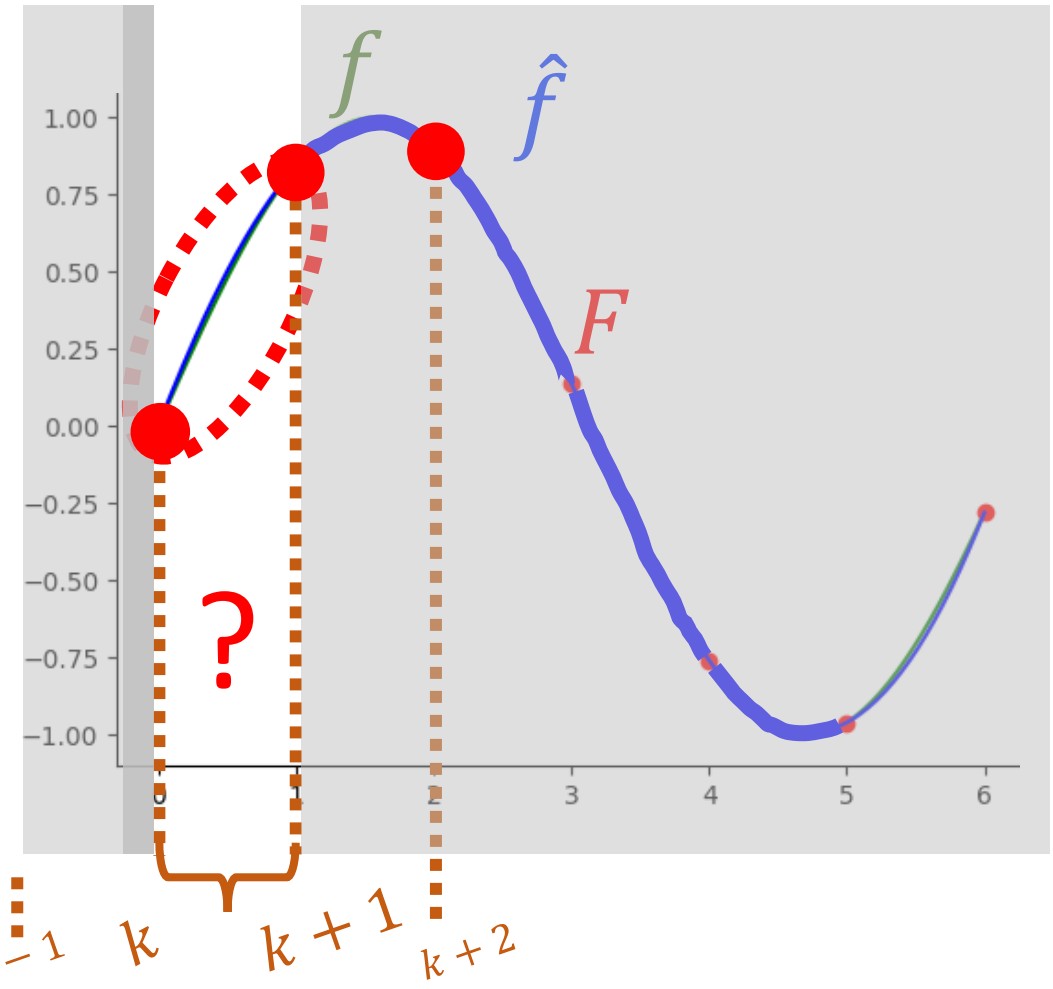
$$\hat{f}(x) = a(x-k)^3 + b(x-k)^2 + c(x-k)^1 + d$$

$$a = {}^1/_6\left(-F(k-1) + 3F(k) - 3F(k+1) + F(k+2)\right)$$
$$b = {}^1/_2\left(F(k-1) - 2F(k) + F(k+1)\right)$$
$$c = {}^1/_6\left(-2F(k-1) - 3F(k) + 6F(k+1) - F(k+2)\right)$$
$$d = F(k)$$

Solution for unknowns

1D Interpolation

Cubic



Segment ***between*** points: $x \in [k, k+1]$

For 4x samples
incl. 2x *outside* range!

$$\begin{cases} x = k & x = k-1 \\ x = k+1 & x = k+2 \end{cases}$$

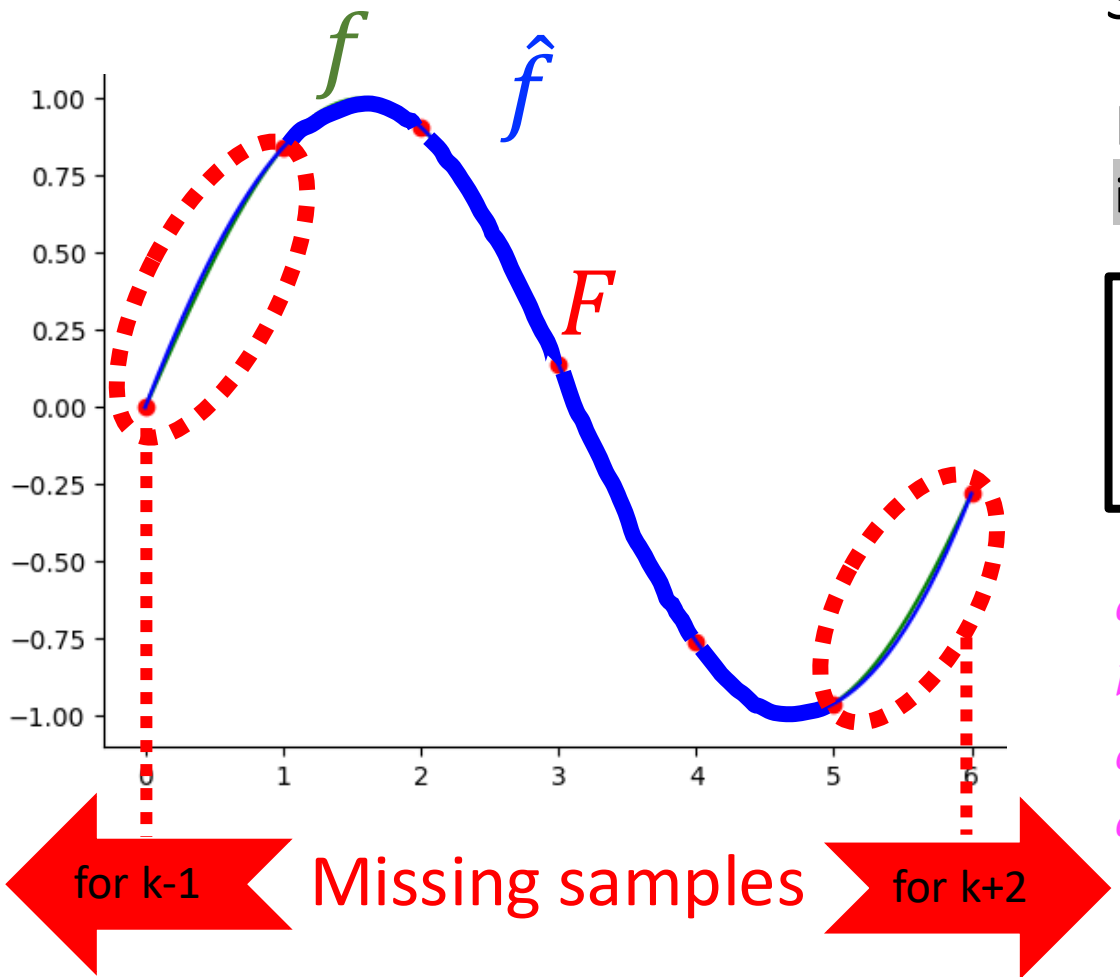
$$\hat{f}(x) = a(x-k)^3 + b(x-k)^2 + c(x-k)^1 + d$$

$$a = {}^1/_6\left(-F(k-1) + 3F(k) - 3F(k+1) + F(k+2)\right)$$
$$b = {}^1/_2\left(F(k-1) - 2F(k) + F(k+1)\right)$$
$$c = {}^1/_6\left(-2F(k-1) - 3F(k) + 6F(k+1) - F(k+2)\right)$$
$$d = F(k)$$

Solution for unknowns

1D Interpolation

Cubic



Segment ***between*** points: $x \in [k, k+1]$

For 4x samples
incl. 2x *outside* range!

$$\begin{cases} x = k & x = k-1 \\ x = k+1 & x = k+2 \end{cases}$$

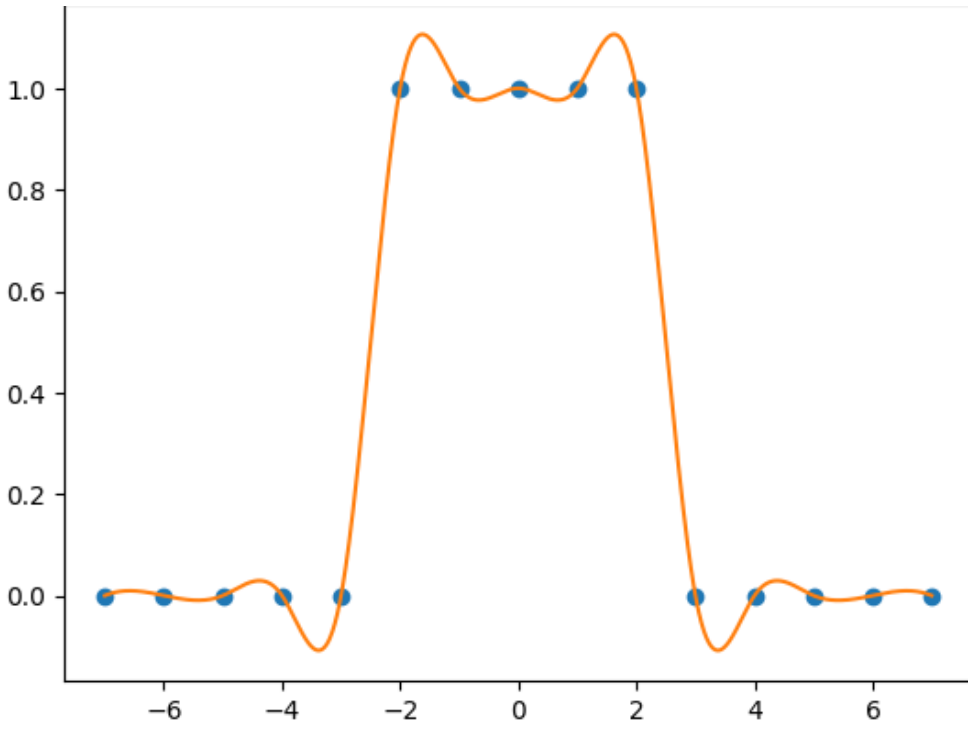
$$\hat{f}(x) = a(x-k)^3 + b(x-k)^2 + c(x-k)^1 + d$$

$$a = {}^1/_6\left(-F(k-1) + 3F(k) - 3F(k+1) + F(k+2)\right)$$
$$b = {}^1/_2\left(F(k-1) - 2F(k) + F(k+1)\right)$$
$$c = {}^1/_6\left(-2F(k-1) - 3F(k) + 6F(k+1) - F(k+2)\right)$$
$$d = F(k)$$

Solution for unknowns

1D Interpolation

Higher-order Polynomial (n>3)



For $2n-1$ samples
- n to the left
- n to the right

Higher-Order Polynomials (HOP) are better for bigger n?

HOPs tend to ***overfit*** to data!

'Fluctuate' wildly between sample points

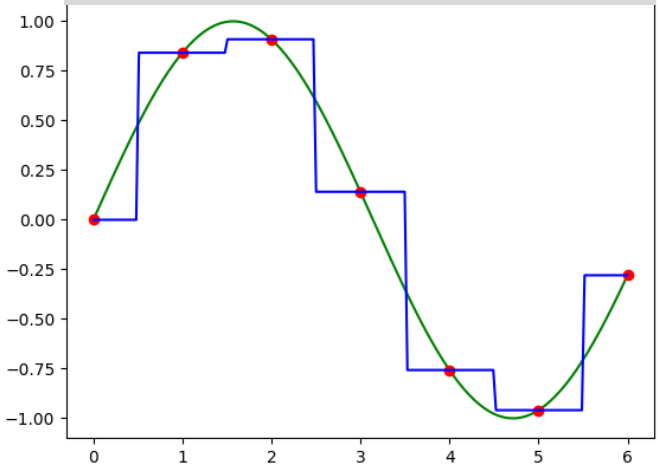
In practice: Stick to the *smallest 'good-enough'* n

f → Continuous function
F → Discrete samples
$\hat{f}$ → Approx. f from F

1D Interpolation

Cheatsheet

Nearest Neighbor



For each $f(x)$: 1 sample (nearest)

Form: Staircase

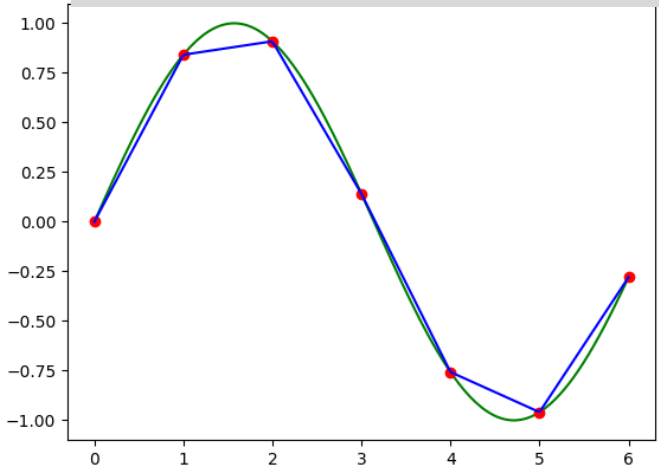
Defined for all x: Yes

Continuous: No (jumps)

Differentiable: No

$$\hat{f}(x) = F(\lfloor x + 0.5 \rfloor)$$

Linear



Uses: 2 samples (1 left & 1 right)

Form: Piece-wise linear

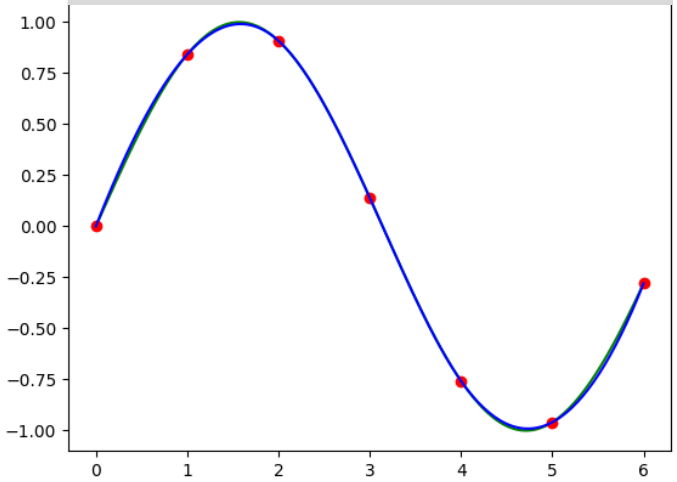
Defined for all x: Yes

Continuous: Yes

Differentiable: No (not at sample points)

$$\hat{f}(x) = (1 - (x - k))F(k) + (x - k)\,F(k + 1)$$

Cubic



Uses: 4 samples (2 left & 2 right)

Form: Smooth

Defined for all x: Yes

Continuous: Yes

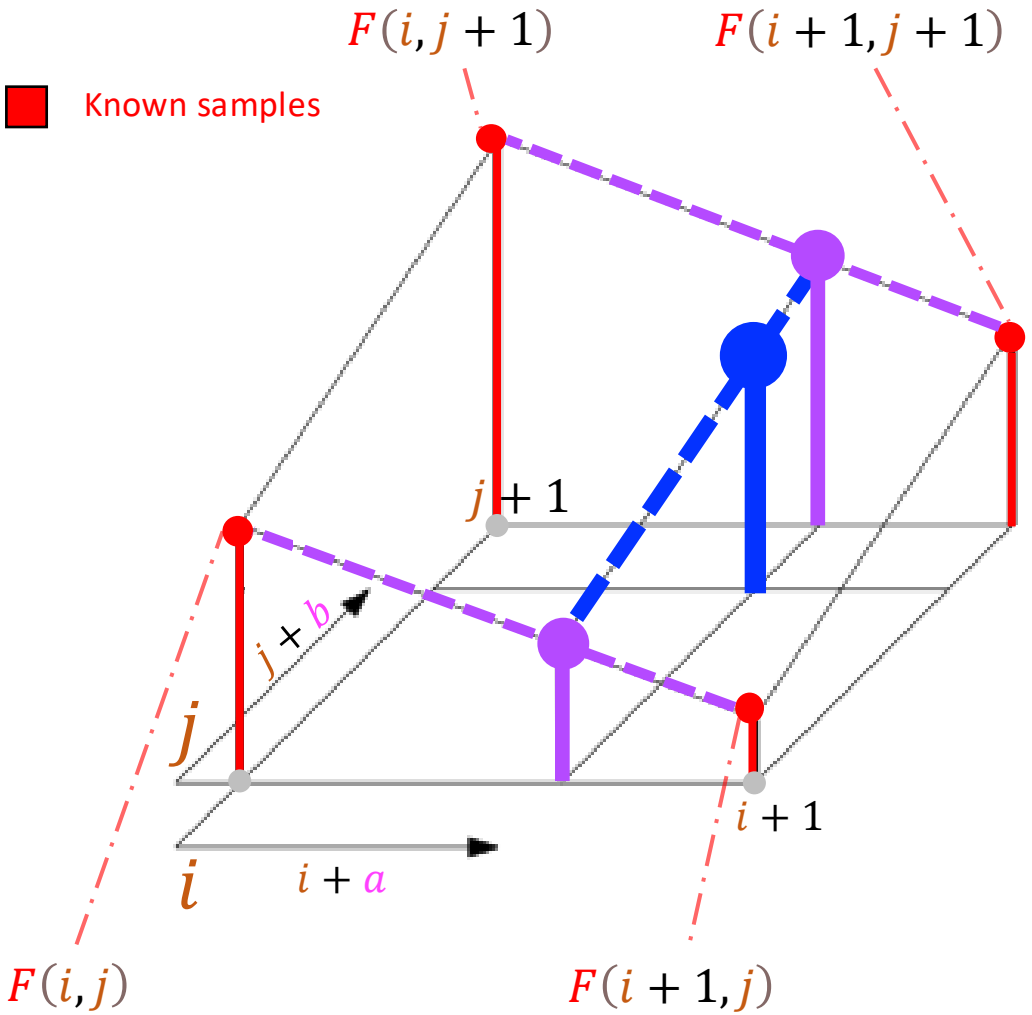
Differentiable: Yes

$$\hat{f}(x) = a(x - k)^3 + b(x - k)^2 + c(x - k)^1 + d$$

a, b, c, d functions of F

2D Interpolation

Bilinear



Estimate $\hat{f}(x, y)$ for $i \le x \le i+1$, $j \le y \le j+1$

$x = i + a$
$y = j + b$
$a, b \in [0,1]$

2-step process:

- First, 1D interpolation ***in x-direction***

$$\hat{f}(x, l)_{\,l=\text{fixed}} = (1-(x-k))F(k, l) + (x-k)\,F(k+1, l)$$

- Then 1D interpolation ***in y-direction***

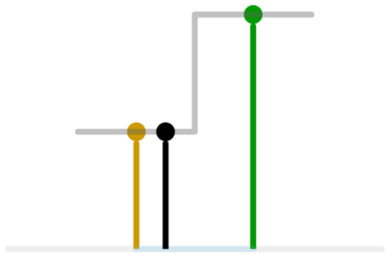
Final 2D interpolation $\hat{f}(x, y)$

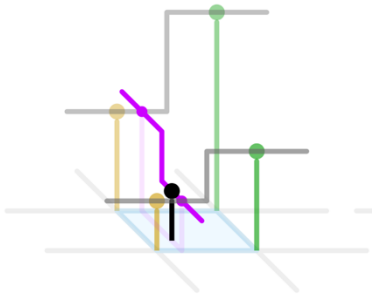
$$f(x, y) = f(i+a, j+b) = (1-a)(1-b)\,F(i, j) + (1-a)\;b\;F(i, j+1) + a\;(1-b)\,F(i+1, j) + a\;b\;F(i+1, j+1)$$

Known samples!

Images

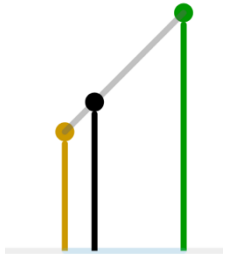
Interpolation: 1D → 2D Generalization

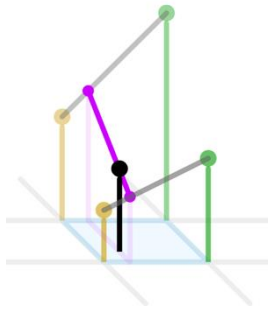




NN (1D) → NN (2D)

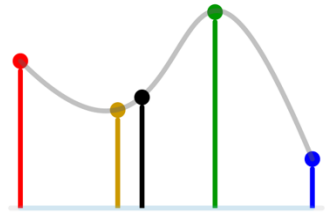
1 sample → 1 sample

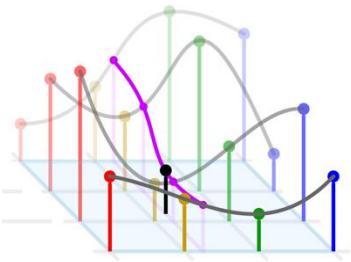




Linear (1D) → Bilinear (2D)

2 samples → 2^2 = 4 samples





Cubic (1D) → Bicubic(2D) ?

4 samples → 4^2 = 16 samples

Black and *red*/*yellow*/*green*/*blue* dots correspond to the *interpolated point* and *neighbouring samples*

Trilinear (3D) ?

2^3 = 8 samples

#samples^#dimensionality

https://en.wikipedia.org/wiki/Bilinear_interpolation

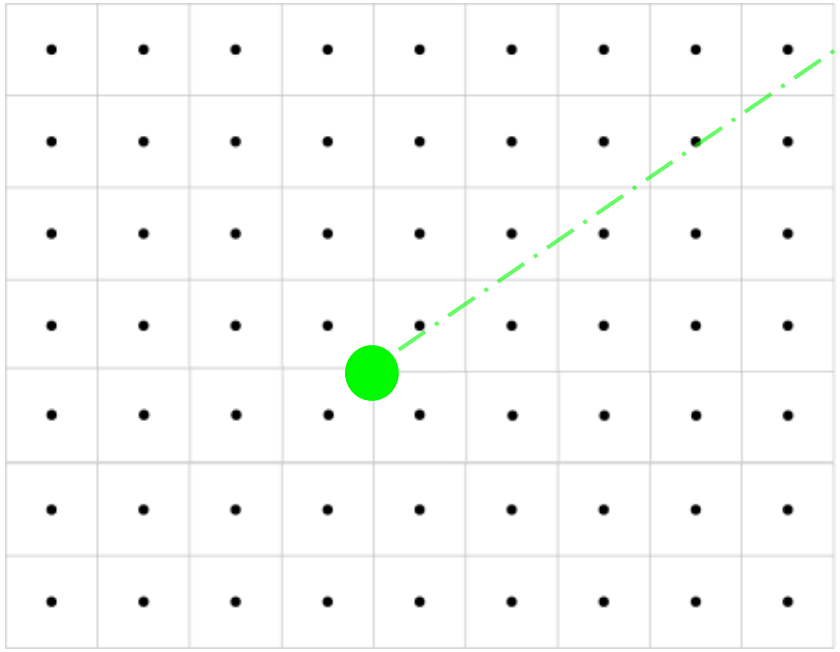
https://en.wikipedia.org/wiki/Bilinear_interpolation#/media/File:Comparison_of_1D_and_2D_interpolation.svg

Interpolation –VS– Extrapolation

Infer values @ points…

Interpolation → ***Inside the*** image grid
(informed ‘guess’ between samples)

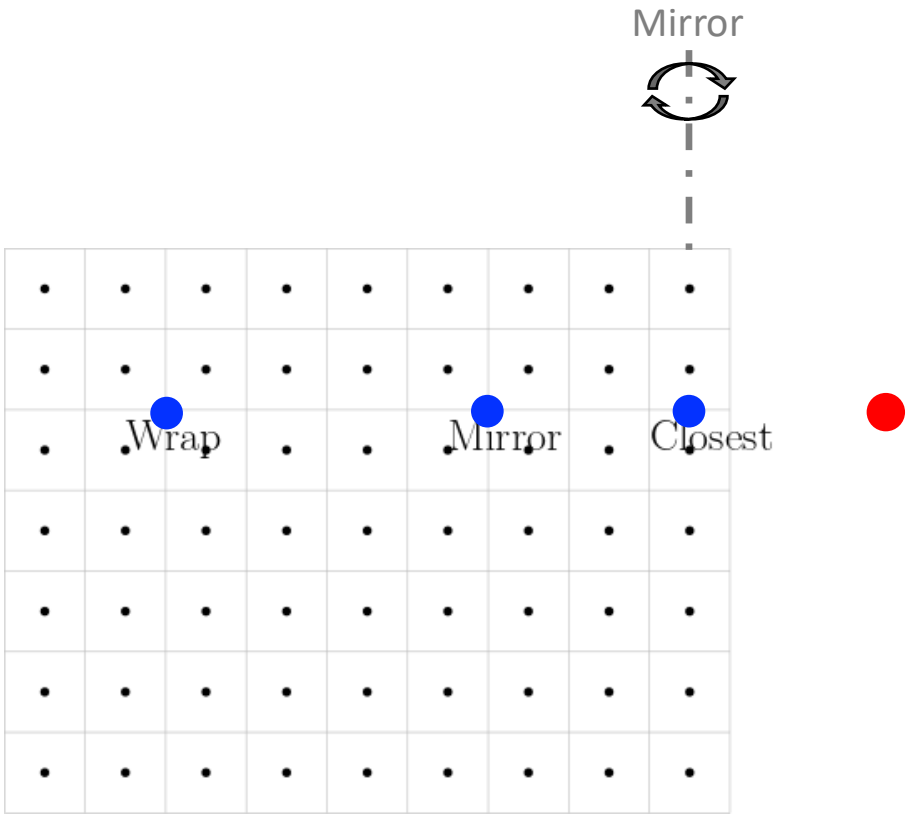
Extrapolation → ***Outside*** the image grid



Mathematician’s view:
Continue trend of image function along an ‘inside→outside’ ray

Dummy example: Color-bleeding TVs →

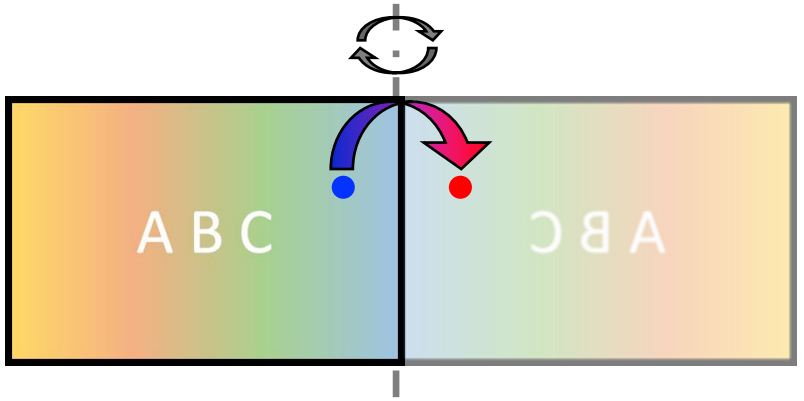
Extrapolation



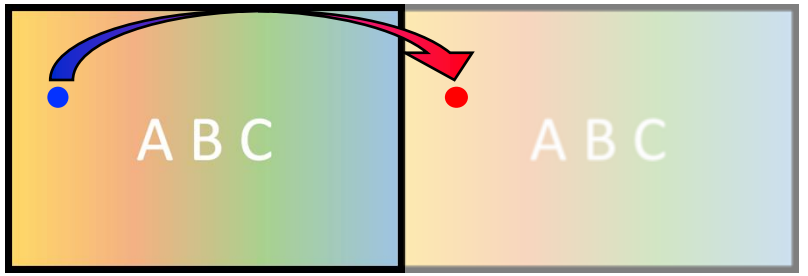
Types

Closest point → Copy value of *closest* point in grid (think: color-bleeding TV)

Mirrored point →



Wrapped point →

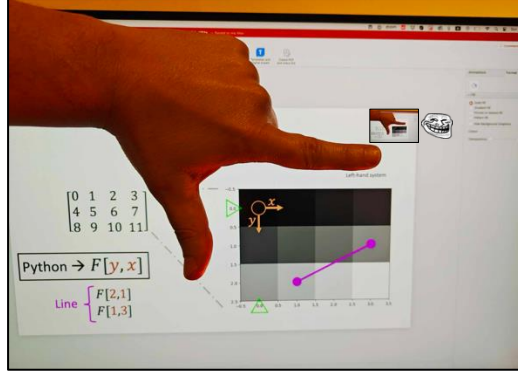


…

'Tiling' imgs

Images

Representation @ NumPy/OpenCV

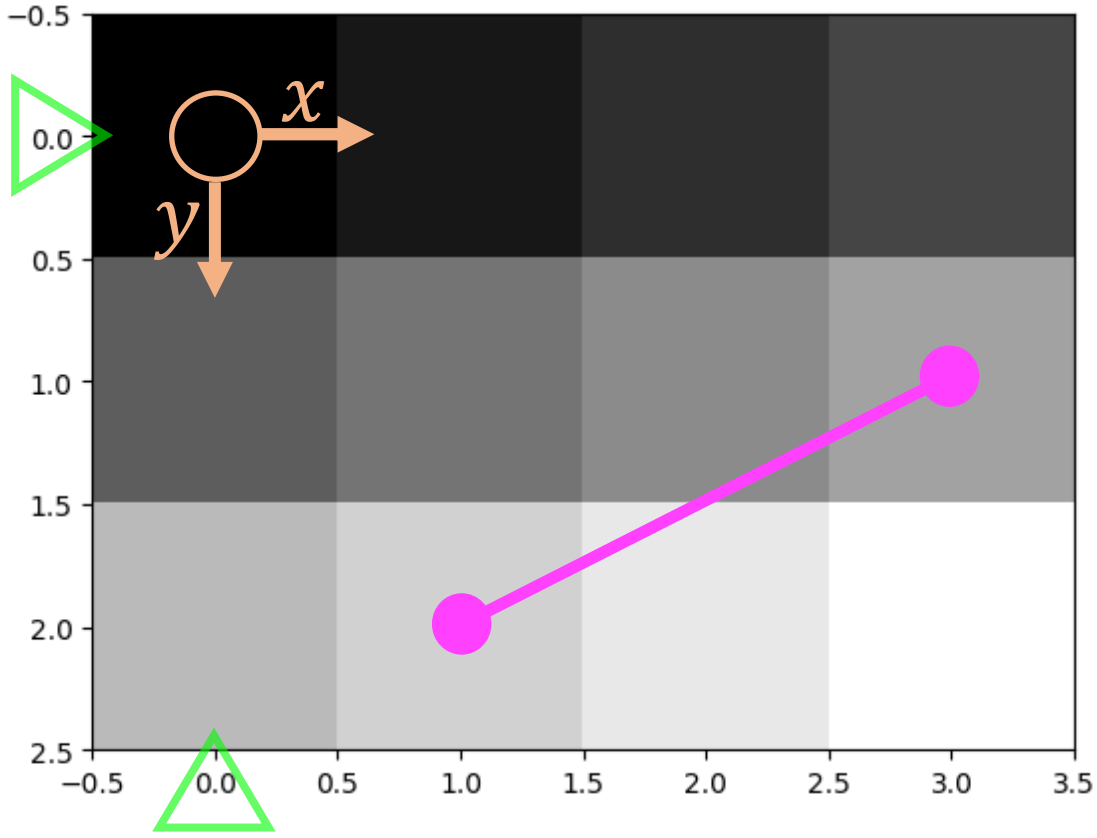


Left-hand system

$$\begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \end{bmatrix}$$

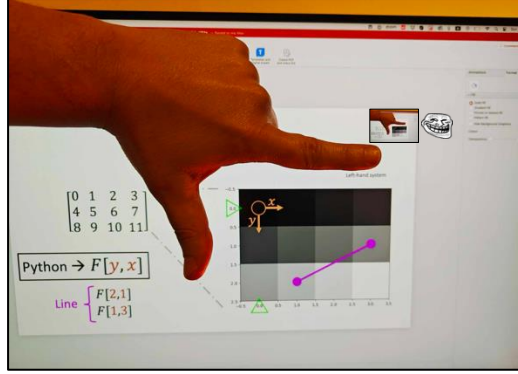
Python → $F[y, x]$

Line $\begin{cases} F[2,1] \\ F[1,3] \end{cases}$



Images

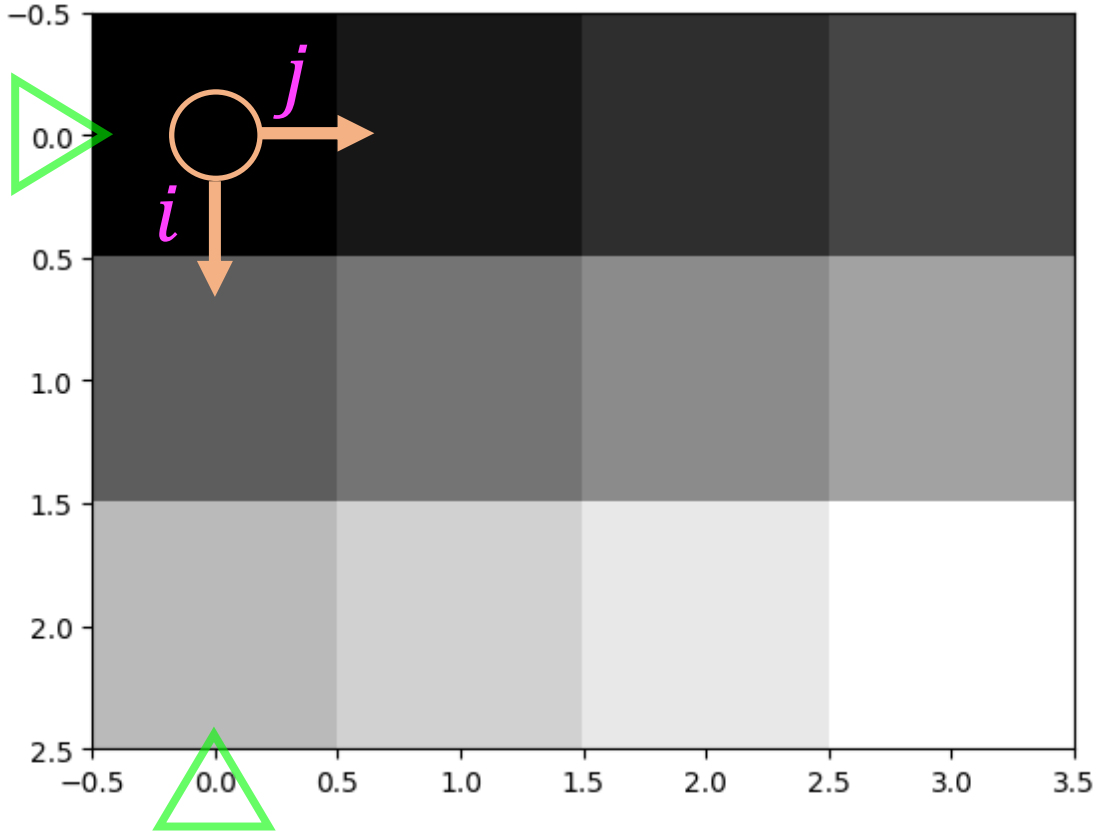
Representation @ NumPy/OpenCV



Left-hand system

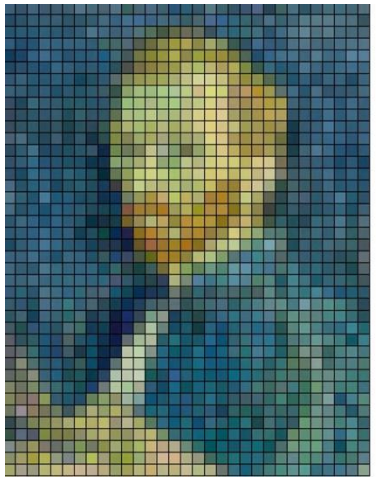
$$\begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \end{bmatrix}$$

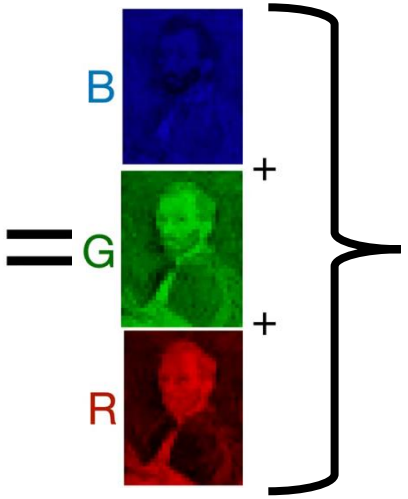
Python → $F[i, j]$

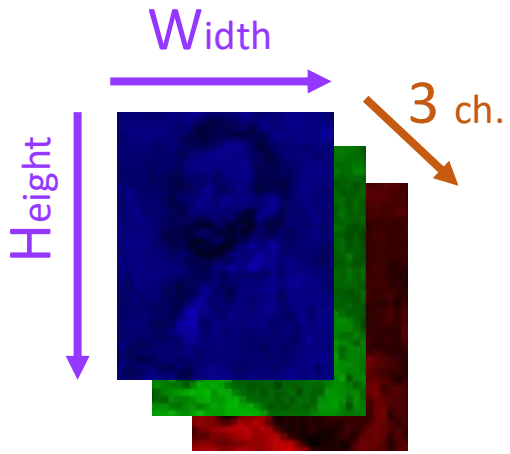


Images

Representation @ NumPy/OpenCV







Tensors
H x W x 3

Optional 4th channel for transparency

OpenCV (NumPy) → $[B, G, R]$

Rows (all) Columns (all) Channel ID

$F[:,:,0]$

$F[:,:,1]$

$F[:,:,2]$

Images

Domain Iterators

Domain Iterators → ***Enumerate*** & ***'visit'*** ***every pixel*** as ***index tuple***

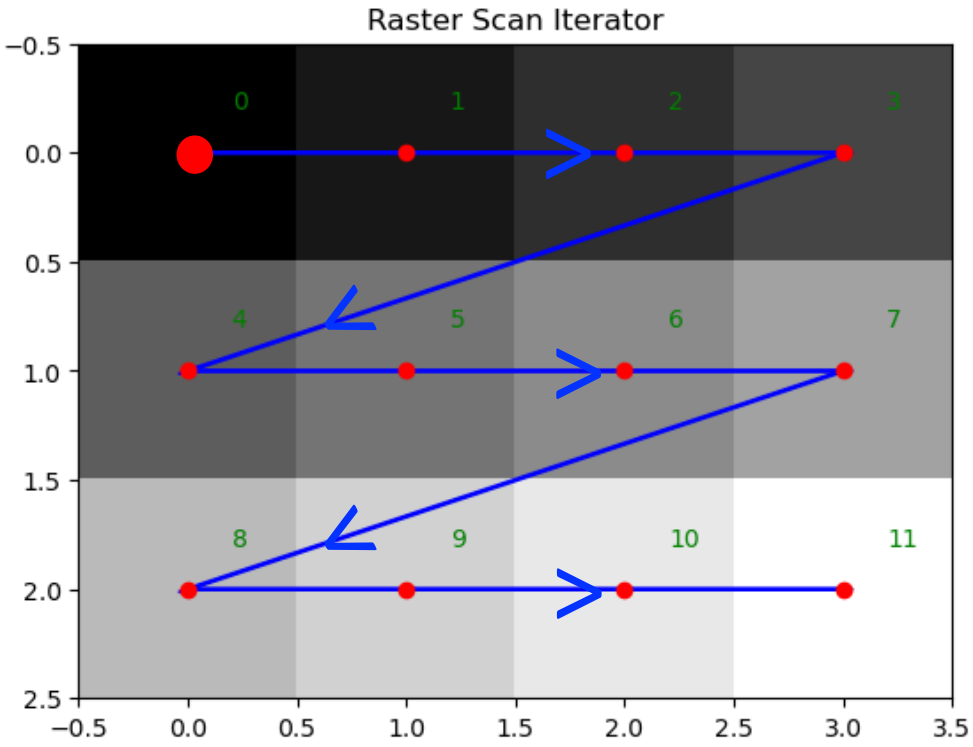
Example → Negation of image:

```
def negateImage(image):
        result = empty(image.shape)
        for p in domainIterator(image.shape):
                result[p] = 1 - image[p]
        return result
```

2-Tuple [p] instead of [i,j]
to access multi-dimensional array

```
def domainIterator(size):
        for i in xrange(size[0]):
                for j in xrange(size[1]):
                        yield (i,j)
```

Mind the order!



Domain Iterators →

Faster than nested 'for' loops

Can also be used for sub-images

Histograms

How do we ‘summarize’ images?

← https://rvdboomgaard.github.io/ComputerVision_LectureNotes/LectureNotes/IP/Images/ImageHistograms.html

https://rvdboomgaard.github.io/ComputerVision_LectureNotes/LectureNotes/IP/PointOperators/index.html →

Image Histograms

1D / Univariate Histogram

Summarize image f via ***Histogram*** of its values (over all possible values)

NumPy: `histogram()`

https://numpy.org/doc/stable/reference/generated/numpy.histogram.html

$$h_f[i] = \sum_{x \in E} [e_i \leq f(x) < e_{i+1}]$$

for $i = 0, \dots, k-1$

Iverson Bracket

$$[c] = \begin{cases} 1 & \text{for } c = \text{true} \\ 0 & \text{for } c = \text{false} \end{cases}$$

Is pixel value inside ***bin***?

$$f(x) \in [e_i, e_{i+1})$$

Bin edges

e_i for $i = 0, \dots, k$

How choose ***bin size/number***?

- No clear answer
- Empirically → More art than science
- Sturges' rule (assumes *Gaussian* data):

$$k = \lceil \log_2 n \rceil + 1$$

bins number

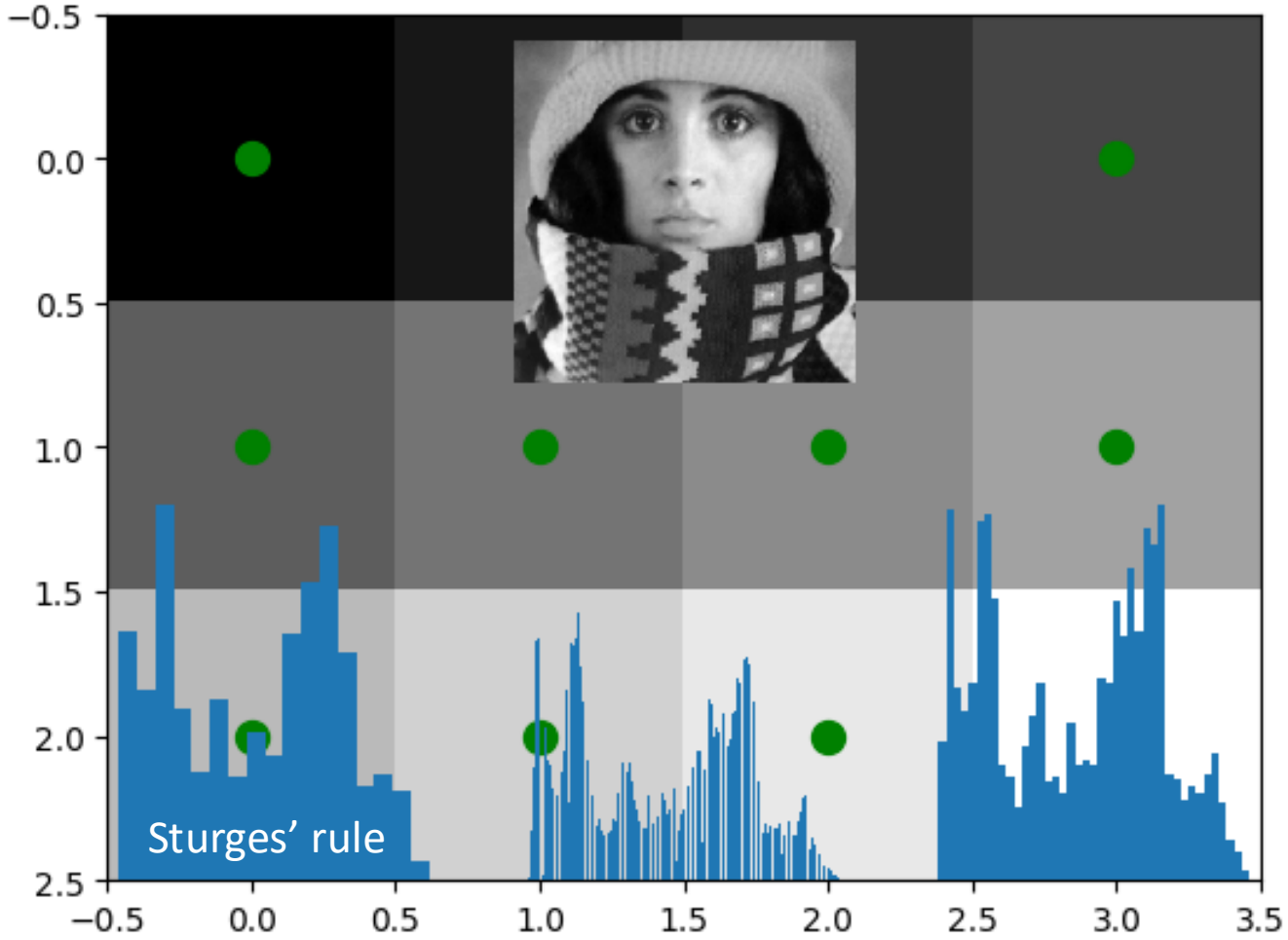
pixels number

`histogram_bin_edges()`

https://numpy.org/doc/stable/reference/generated/numpy.histogram_bin_edges.html

Image Histograms

1D / Univariate Histogram



Too many (and too small) bins?

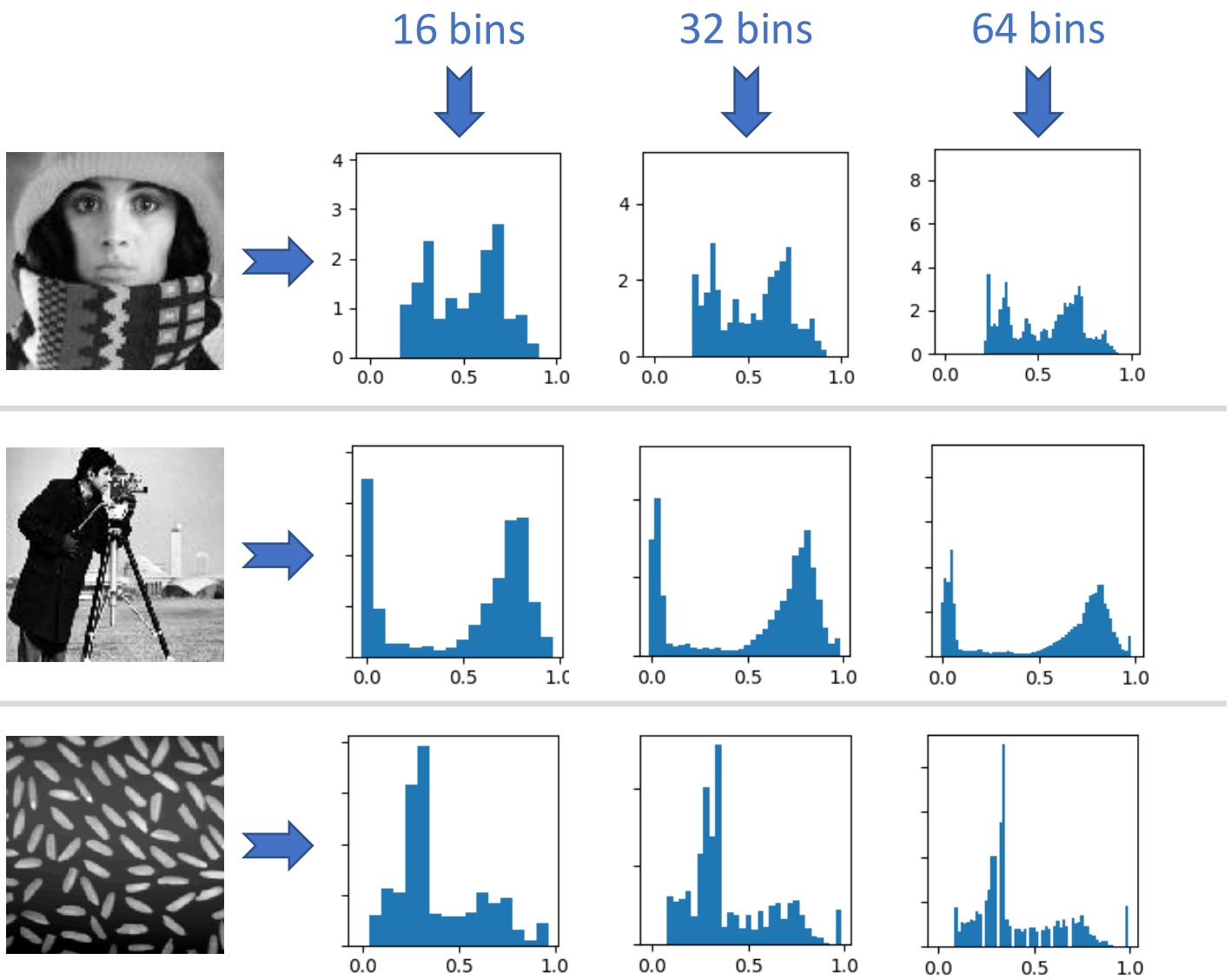
→ Many bins will be empty

→'High-frequency' info

Too few (and too big) bins?

→ Will not encode 'well enough' the distribution of values

Image Histograms



Too many (and too small) bins?

→ Many bins will be empty

→'High-frequency' info

Too few (and too big) bins?

→ Will not encode 'well enough' the distribution of values

Image Histograms

Histogram → $h_f(u)$ counts pixels with value $f(x) = u$

PDF – Probability-density function

normalized such that: $\sum h_f(u) = 1$

Cumulative Histogram → $H_f(u)$ counts pixels with value $f(x) \leq u$

Cumulative distribution function

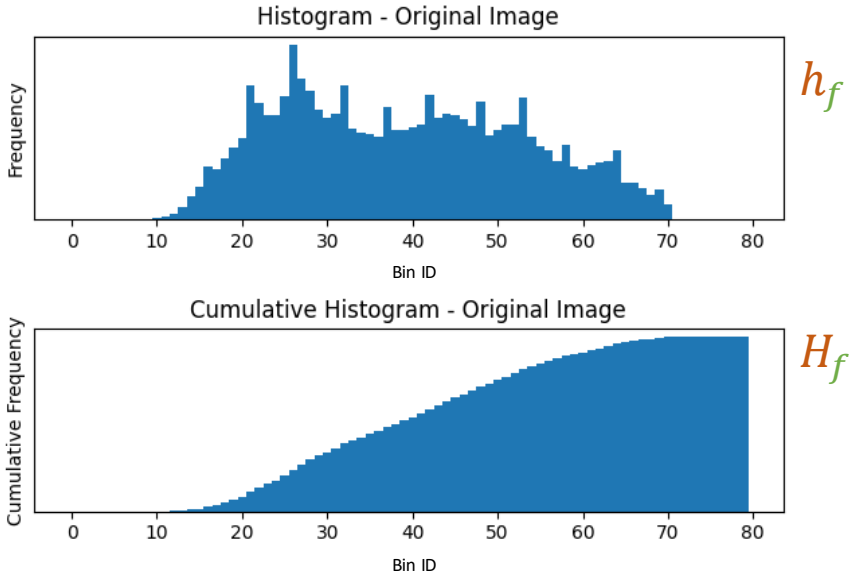


Image Histograms

3D / Multivariate Histogram

Grayscale images → ***1D Histogram***

(each pixel → Scalar value) (Univariate)

What about ***Color*** images?

(each pixel → Triplet of BGR value)

$$\boldsymbol{f}(\boldsymbol{x}) = \begin{bmatrix} f_1(\boldsymbol{x}) \\ f_2(\boldsymbol{x}) \\ f_3(\boldsymbol{x}) \end{bmatrix} = \begin{bmatrix} b(\boldsymbol{x}) \\ g(\boldsymbol{x}) \\ r(\boldsymbol{x}) \end{bmatrix}$$

Ignores color **Correlations**!

Trivial solution: ***3x separate*** 1D histograms per channel

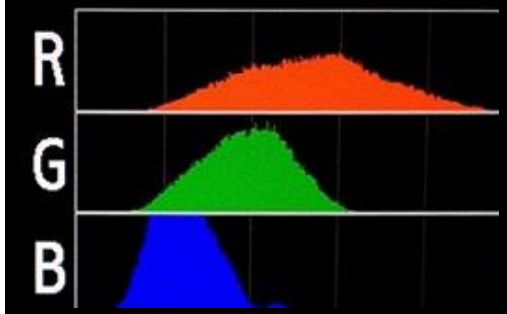


Image Histograms

3D / Multivariate Histogram

Better solution → ***3D Histogram***

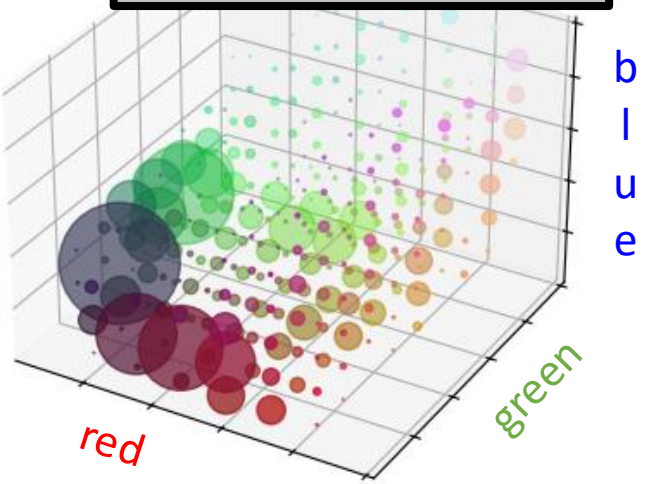
$$h_f[i,j,k] = \sum_{x \in E} \left[e_{1,i} \le f_1(x) < e_{1,i+1}\right] \left[e_{2,j} \le f_2(x) < e_{2,j+1}\right] \left[e_{3,k} \le f_3(x) < e_{3,k+1}\right]$$

Is pixel value inside

3D bin

with indices (i, j, k) ?

3D bins: 8x8x8 cubes



Bin centered @ RGB point

Sphere size == Hist. count

`histogramdd()`

https://numpy.org/doc/stable/reference/generated/numpy.histogramdd.html

Image Histograms

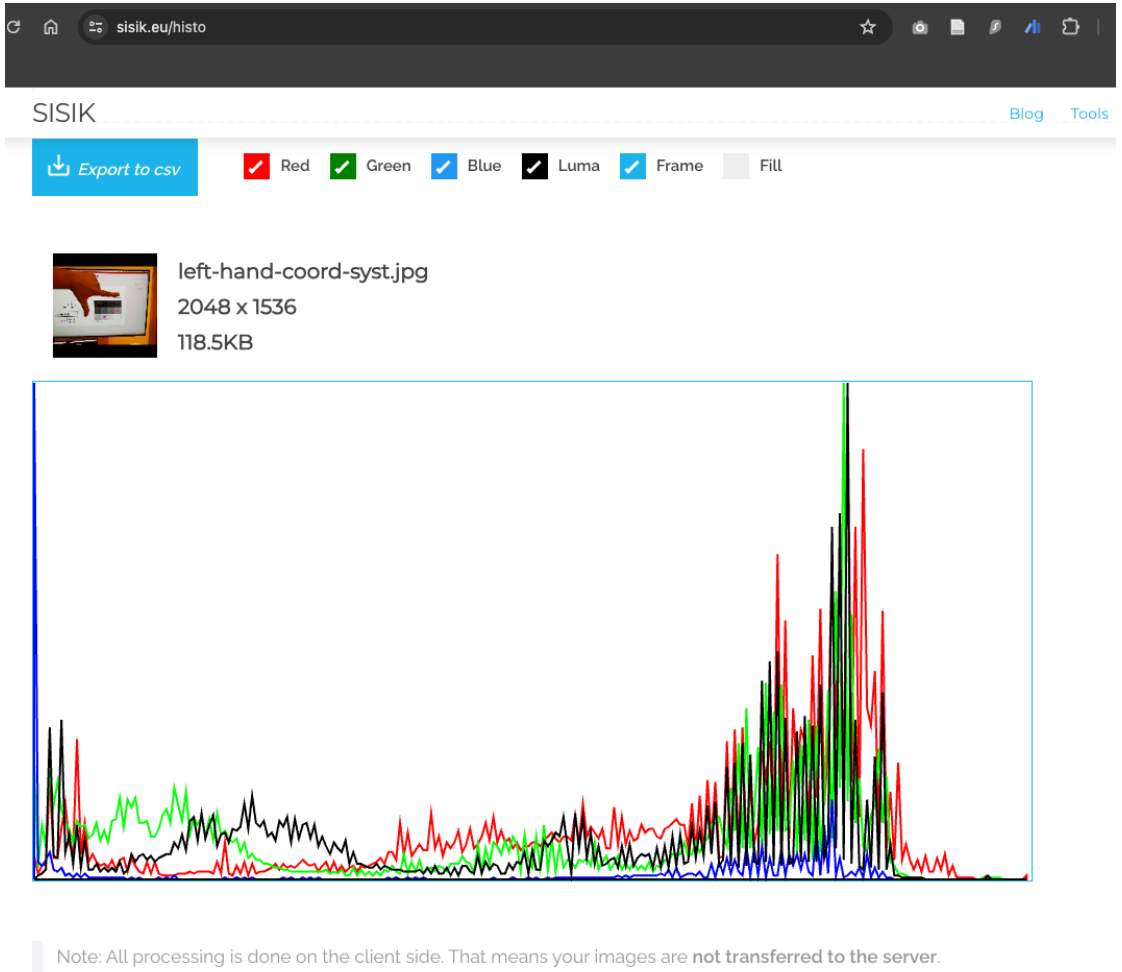
Online Demo - 1D Histograms

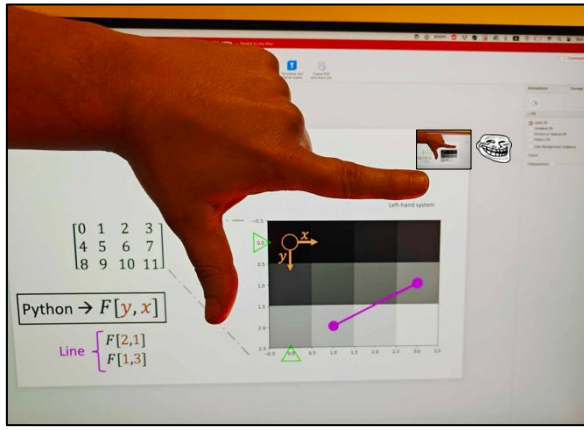
*'This tool basically creates **256 bins** for each color (red, green, blue) and greyscale (luma) intensity. The number of bins is shown on the **horizontal** axis.'*

*'The tool then loops through every image pixel and counts the **occurrence of each intensity**. The counts of occurrences in each bin are then displayed on **vertical** axis.'*

*'Counts for each pixel-intensity are **normalized** to range 0 to 255 before they are displayed on the graph.'*

'Used JavaScript in combination with WebAssmebly to create this tool.'





https://sisik.eu/histo

Point Operators

How do we ‘process’ images?

https://rvdboomgaard.github.io/ComputerVision_LectureNotes/LectureNotes/IP/PointOperators/index.html →

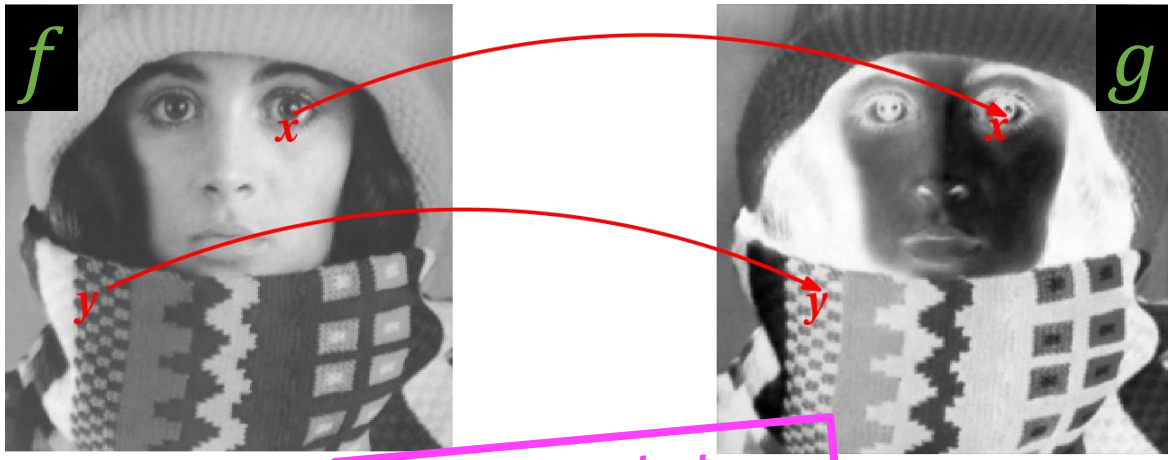
Point Operators

Point-wise processing

Two images f and g (maybe also h)

defined @ same domain

(same size & $\in \mathbb{R}^2$)



Independent computation per pixel!

Goal:

Iterate over ***all*** f ***pixel locations*** (or of 2 imgs)

Per-pixel 'processing' of corresponding ***value***

Store result @ ***same location*** on g (or a 3rd img)

Examples:

Point-wise image negation → $g(x) = 1 - f(x)$

Point-wise addition of a scalar → $g = f + 1$

Point-wise log of pixels values → $g = \log(1 + f)$

Point-wise addition of 2 images → $h = f + g$

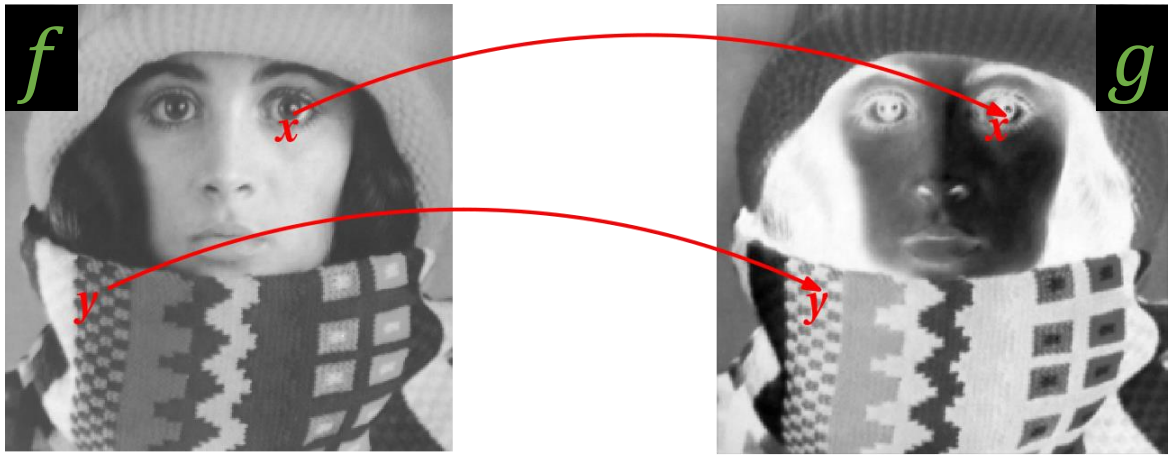
Point Operators

Point-wise processing

Image $f : \mathcal{D} \rightarrow \mathcal{R}$

maps from: spatial domain $\mathcal{D}$

to: range $\mathcal{R}$



Point Operator Ψ

transforms image f → new image g

$$g = \Psi f$$

$$\forall x \in \mathcal{D}: \quad g(x) = \psi\big(f(x)\big)$$

Function Ψ:

- Works on values in range $\mathcal{R}$ of f
- The range of Ψ might differ from its domain
  (so, resulting image will have a new range)

  Example: Color img --> Grayscale img
- Input can be ≥ 1 images

Point Operators

Image Arithmetic

Two Images $f: \mathbb{R}^2 \rightarrow \mathbb{R}_+$ (2-Dimensional, Grayscale)

$g: \mathbb{R}^2 \rightarrow \mathbb{R}_+$

Sampled on same grid B

Discrete version → F and G

Point-wise addition $h = f + g$

$\forall x \in R^2$: $h(x) = (f + g)(x) = f(x) + g(x)$

$\forall k \in Z^2$: $H(k) = F(k) + G(k)$

```
def addImage(f, g):
    h = f.copy()
    for p in domainIterator(f):
        h[p] = f[p] + g[p]
    return h
```

Domain Iterator (tuple)

*Can also **Lift** Operator (+)* — *Automatic in Python* — *Infix notation:* **H=F+G**

Point Operators

Image Arithmetic

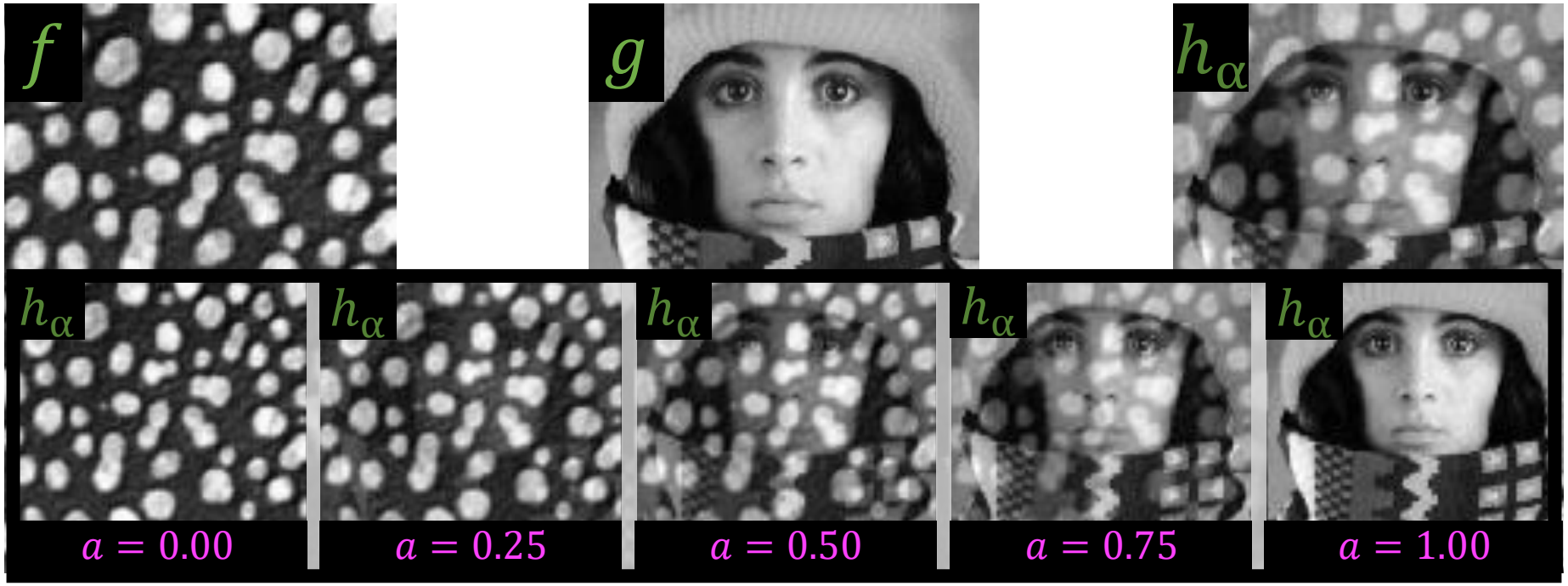

```
def addImage(f, g):
    h = f.copy()
    for p in domainIterator(f)
        h[p] = (f[p] + 2 g[p]) / 3
    return h
```

*Can also **Lift** Operators **+** and **/***

H = (f + 2g) / 3

Point Operators

A-Blending



a.k.a.
weighted average
of 2 images

$$h_\alpha = (1 - \alpha) f + \alpha g$$

Steering/mixing weights

```
def alphaBlend(f, g, alpha):
        return (1 - alpha) * f + alpha * g
```

Lifted *Operators*

*+ and - and **

Steering/mixing weights

Point Operators

Unsharp Masking

Two Images:

f: Original image

g: ***Blurred*** version of f
(will discuss next weeks)

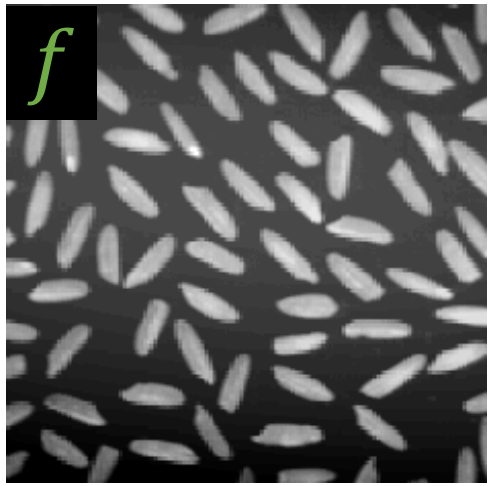
$$h_\beta = f + \underbrace{\beta}_{\text{Steering weight}}(f - g)$$

https://en.wikipedia.org/wiki/Unsharp_masking

Point Operators

Thresholding / Binarizing

Grayscale image



Binary image

$$g = [\, f > thresh \,]$$

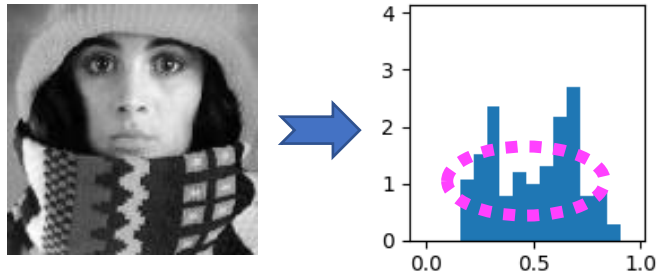
Iverson Bracket → $[c] = \begin{cases} 1 & \text{for } c = \text{true} \\ 0 & \text{for } c = \text{false} \end{cases}$

In Python **not** needed!

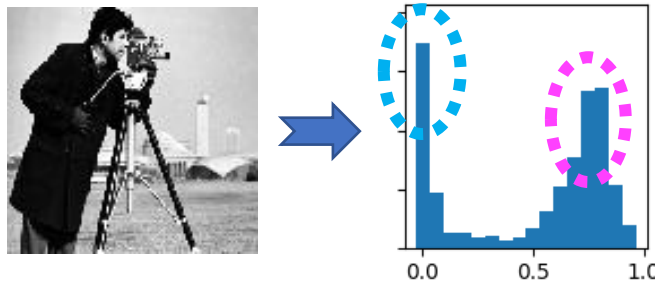
Scalar Threshold

Point Operators

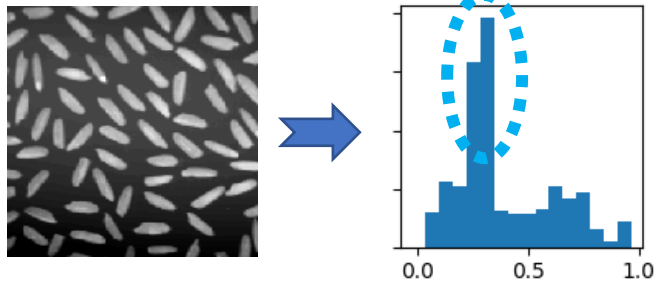
Histograms (issues)



Only *mid-range* gray
No extremes



Spikes for *bright* and *very-dark* areas



Mostly *dark-ish* areas

Not using full range
of luminance
might be sub-optimal

Image might be … :

Missing *large (bright)* values? → … too *dark*

Missing *low (dark)* values? → … too *bright*

Changing illumination conditions
for same scene → very different histograms!

Point Operators

Histogram Contrast Stretching

$f(x)$: $\mathbb{R}^2 \rightarrow [a, b] \subset [0,1] \in \mathbb{R}$

Contrast Stretching { Construct new image using full range of luminance

Monadic Operator

$$g(x) = \varphi(f(x)) = \frac{f(x) - a}{b - a}$$

'Distance' from min luminance in f

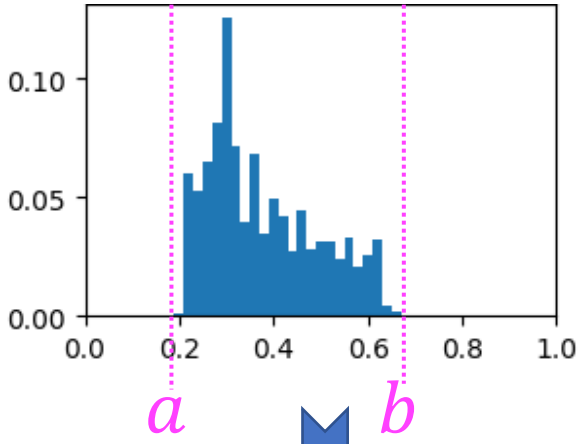
Stretches luminance range to [0,1]

$$g = \frac{f - f_{min}}{f_{max} - f_{min}}$$

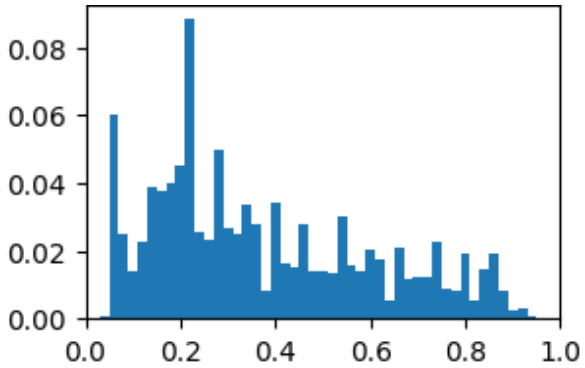
Lifted arithmetical operators

Input: **Low Contrast**

X axis: luminance values (bins)
Y axis: frequency for each value



Output: **Contrast Stretched**



Point Operators

Histogram Equalization (idealized)

Produce image $g = \varphi f$ with ***'constant' (flat) histogram*** h_g

All luminance values should be 'equally' probable
(as 'equal' as possible for finite number of pixels)

Useful consequence: ***cumulative histogram*** $H_g(u) = u$

Why? Think: integrate a constant

Luminance value: $u = f(x)$

Preserve ***order***: $u_1 \leq u_2 \Rightarrow \varphi(u_1) \leq \varphi(u_2)$

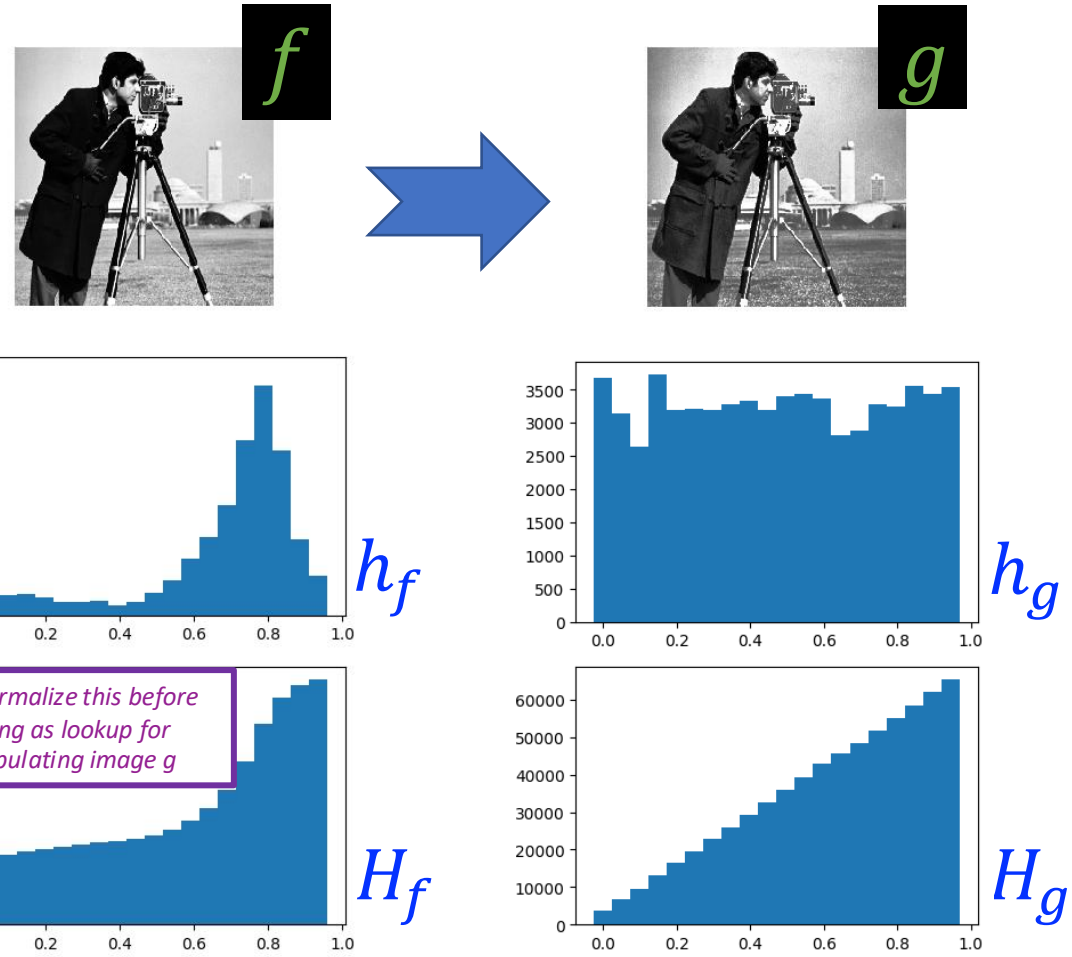
So, the **p^{th} percentile** of f & g should be the same:

$$H_f(u) = H_g(\varphi(u)) = \varphi(u)$$

Visit each pixel of f
replace its value u with:

$$\varphi(u) = H_f^{norm}(u)$$

To populate a pixel of g:
- visit the corresponding pixel location of f
- read its value $u = f(x)$
- use u as key to 'query' (the *normalized*) H_f as "lookup table"
- Copy value from "lookup table" as value of pixel of g



← [Source] Read more & get and *run* code

Point Operators

Histogram Equalization / Self-study ← ← ←

Take a deep breath -- implementation details:

(you need to run the code and print/visualize intermediate variables if you want to get every detail)
(grayed-out things are not too important for the "big picture")

```
import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt
imgF = cv.imread('wiki.jpg', cv.IMREAD_GRAYSCALE)

hist,bins = np.histogram(imgF.flatten(),256,[0,256])
cdf = hist.cumsum()

cdf_normalized = cdf * float(hist.max()) / cdf.max()
plt.plot(cdf_normalized, color = 'b')
plt.hist(img.flatten(),256,[0,256], color = 'r')
plt.xlim([0,256])
plt.legend(('cdf','histogram'), loc = 'upper left')
plt.show()
```

h_f

H_f

"You can see the histogram lies in brighter region. We need the full spectrum. For that, we need a transformation function which maps the input pixels in brighter region to output pixels in full region. That is what histogram equalization does."

```
cdf_m = np.ma.masked_equal(cdf,0)
cdf_m = (cdf_m - cdf_m.min())  /  (cdf_m.max()-cdf_m.min())  * 255
cdf = np.ma.filled(cdf_m ,0).astype('uint8')
```

Ignore 0 values in histogram

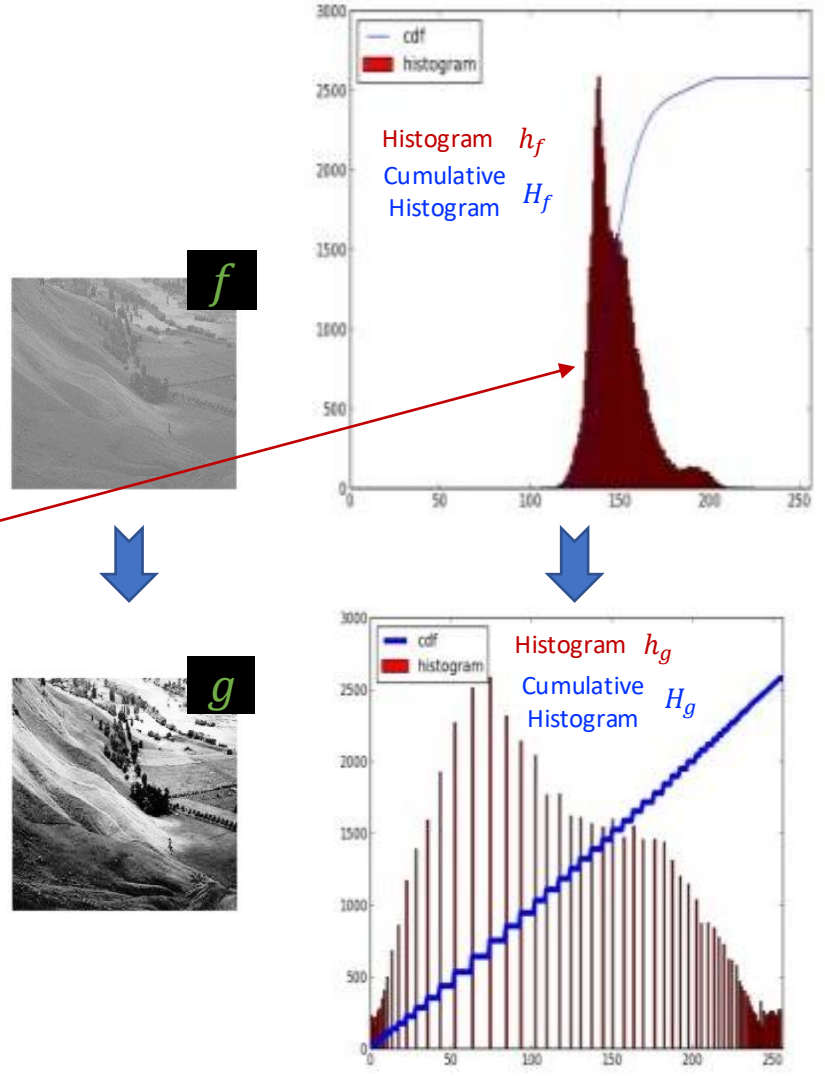
Normalized H_f^{norm}

"Now we have the look-up table that gives us the information on what is the output pixel value for every input pixel value."

```
imgG = cdf[imgF]
```

In previous slide, bottom-left corner, this is shown as in the black box

$$\varphi(u) = H_f^{norm}(u)$$



Point Operators

Histogram Thresholding

Segment foreground (objects of interests) from background

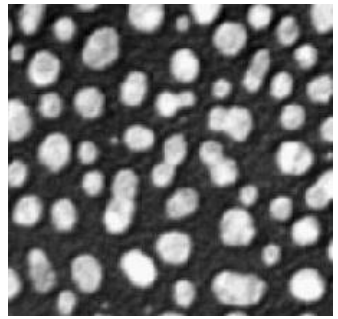
- Manually ☹ choose threshold t → looking at histogram (bimodal distribution)
- ***Automatically*** choose threshold t → ***IsoData*** thresholding

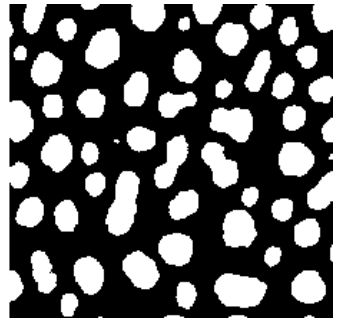
$m_L(t)$ → Mean of pixel values that are: $\leq t$

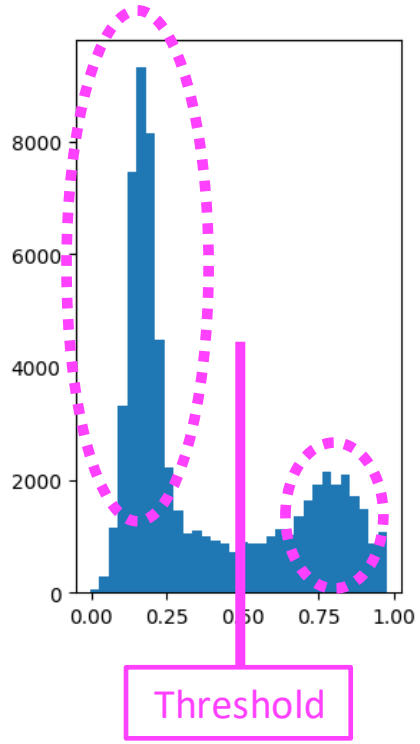
$m_H(t)$ → Mean of pixel values that are: $> t$

$$t = \frac{m_L(t) + m_H(t)}{2} = m(t)$$

Function of itself!







Iterative (Recursive) Solution

- Start with initial guess t_i for $i = 0$
- Compute $t_{i+1} = m(t_i)$
- Repeat until convergence
  (i.e. repeat as long as t_{i+1} and t_i are not too similar)

Resources for these slides

https://rvdboomgaard.github.io/ComputerVision_LectureNotes/LectureNotes/IP/Images/index.html

https://rvdboomgaard.github.io/ComputerVision_LectureNotes/LectureNotes/IP/PointOperators/index.html

→ If you have any suggestions, please post on Canvas ←