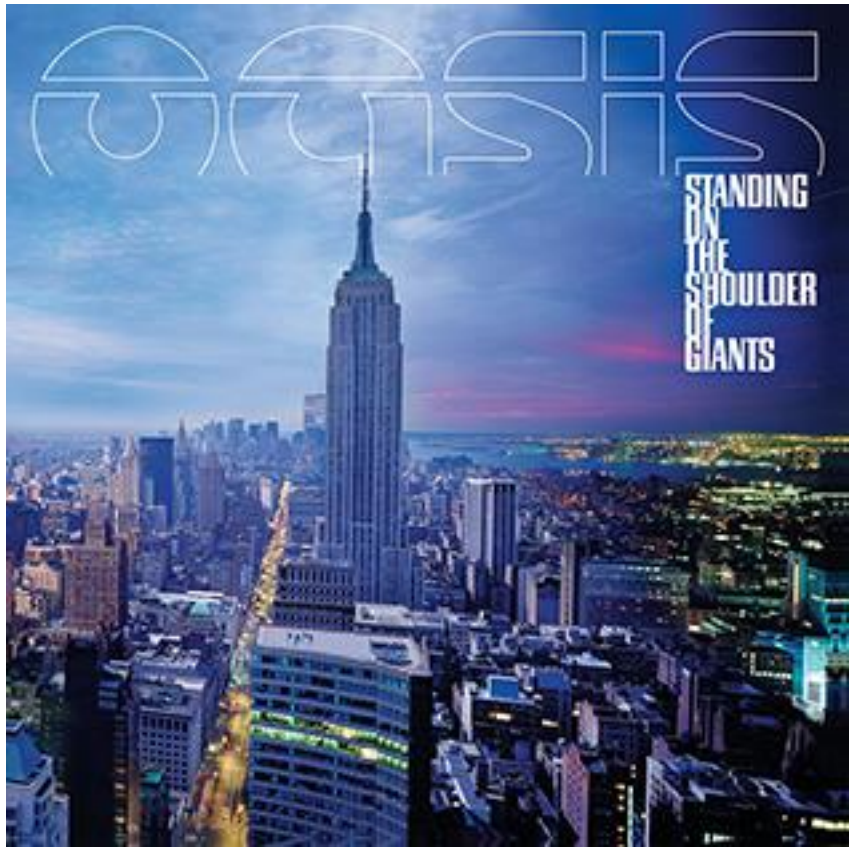
Image Transformations and Enhancement

Computer Vision - Lecture 02

Further reading

- Some slides adapted from Alyosha Efros, Derek Hoiem, Svetlana Lazebnik

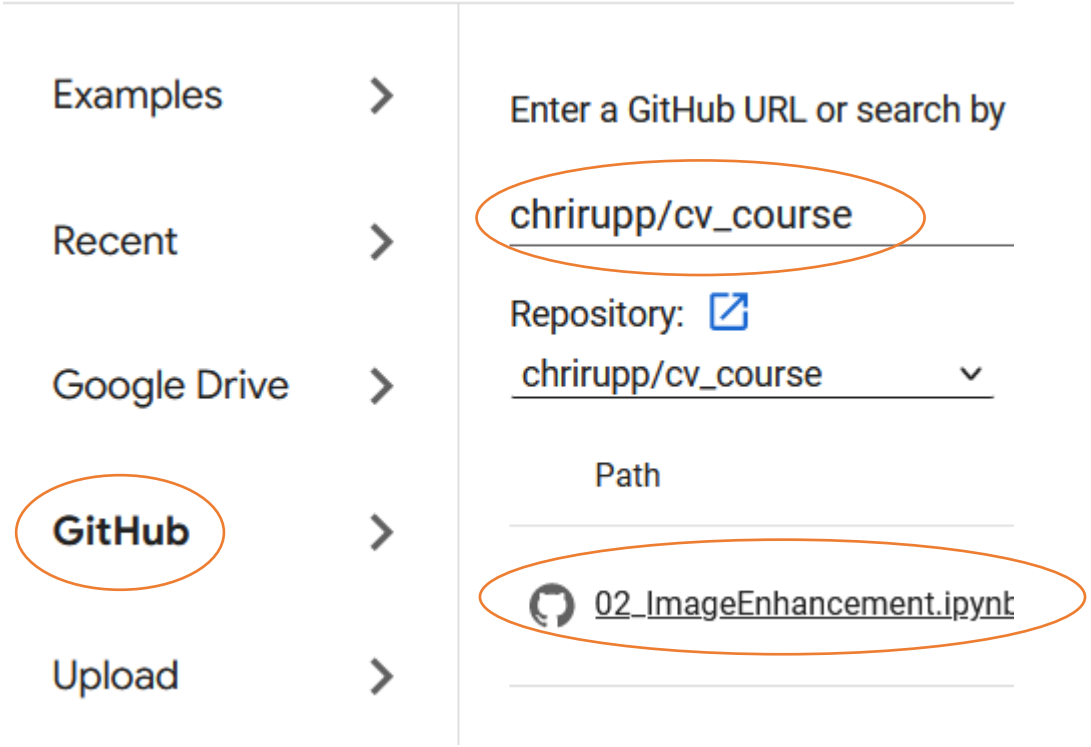
- Almost all examples on the slides today come from python code



By Scan, Fair use, https://en.wikipedia.org/w/index.php?curid=11751438

Code

https://colab.research.google.com/



Warning: This notebook was not authored by Google

This notebook is being loaded from GitHub. It may request access to your data stored with Google, or read data and credentials from other sessions. Please review the source code before executing this notebook.

Cancel Run anyway

Using Colab

- Cells separate code blocks.
- Cells can be run with the play button.
- A notebook is stateful! Things might use outputs from previous cells.
- When you change things, make sure to run all necessary cells.

Using the code

- Can be helpful to understand the connection between the ideas and their implementation.
- Very useful for the practicals (and also for classes).
- The exam will not ask for implementation details.

Overview

- Images as functions
- Subsampling & upsampling
- Point-wise transformations
- Geometric transformations
- Image filtering

Why is Computer Vision hard?

What we see

```
import cv2
image = cv2.imread('02/image.jpg')
print(image)
```

```
[[[ 82 100 129]
  [ 83 101 130]
  [ 84 102 133]
  ...
  [ 23  34  54]
  [ 20  34  57]
  [ 17  33  56]]
 ...
```

What a computer sees

Digital Images

red channel

green channel

blue channel

Images as Pixels

height, width, channels (B, G, R)

(854, 1280, 3)

grayscale images have only one channel

(854, 1280)

top left pixel

```
[[107 108 109 ...  39  39  38]
 [112 113 114 ...  70  67  64]
 [117 118 120 ... 174 168 164]
 ...
 [184 183 182 ...  25  26  26]
 [184 183 182 ...  25  26  26]
 [184 183 182 ...  25  25  26]]
```

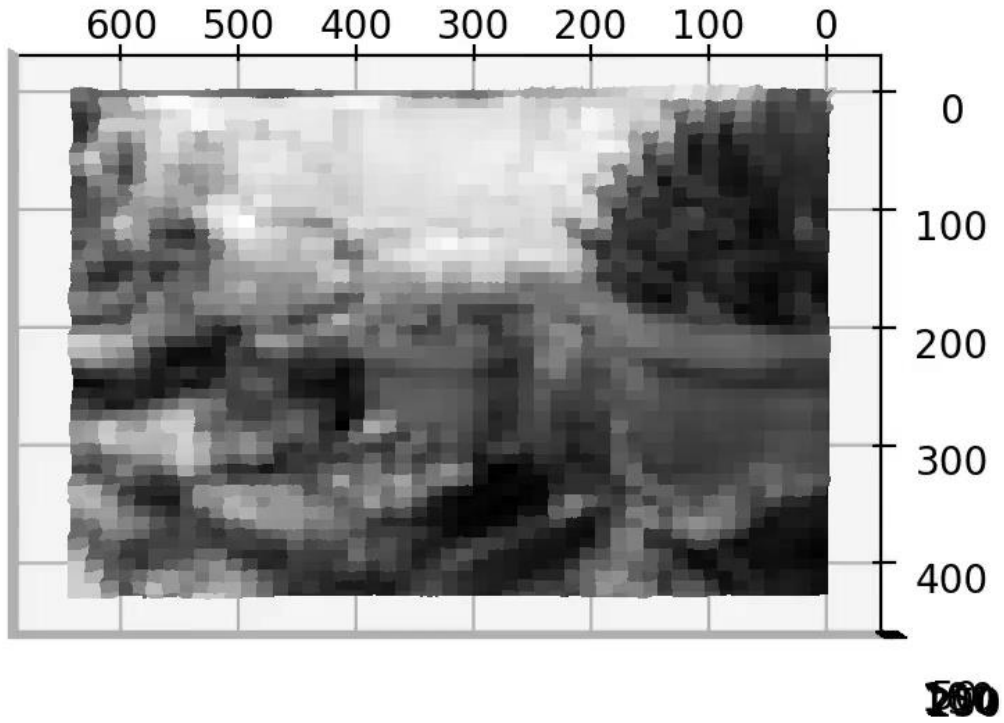
bottom right pixel

2D array of pixel intensities

```
print(image.shape)
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
print(gray.shape)
print(gray)
```

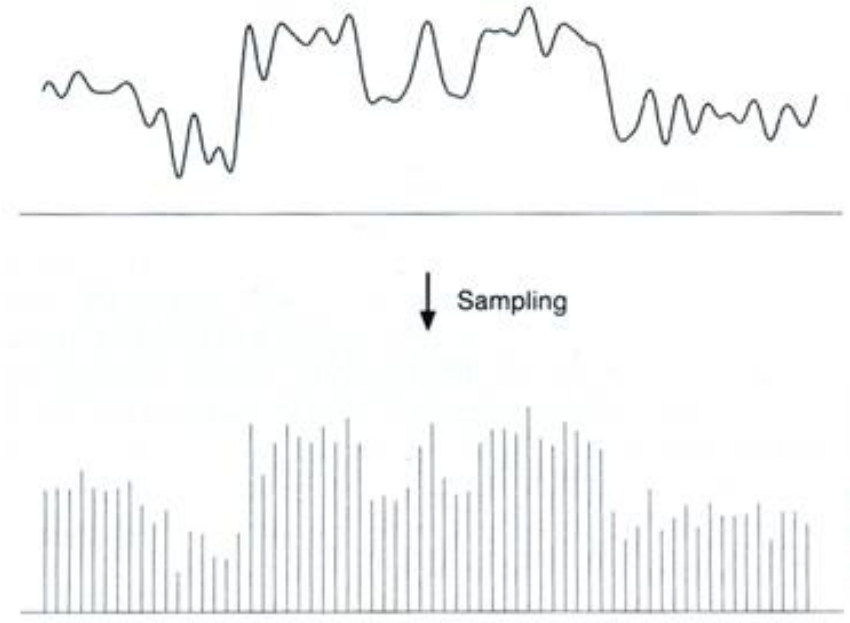
Images as Functions

- We can interpret an image as samples from a continuous 2D function $f(x, y)$
- f maps from 2D coordinates to image intensities
- The functional representation is very useful to express operations on images (e.g. filtering, transformations, etc.)

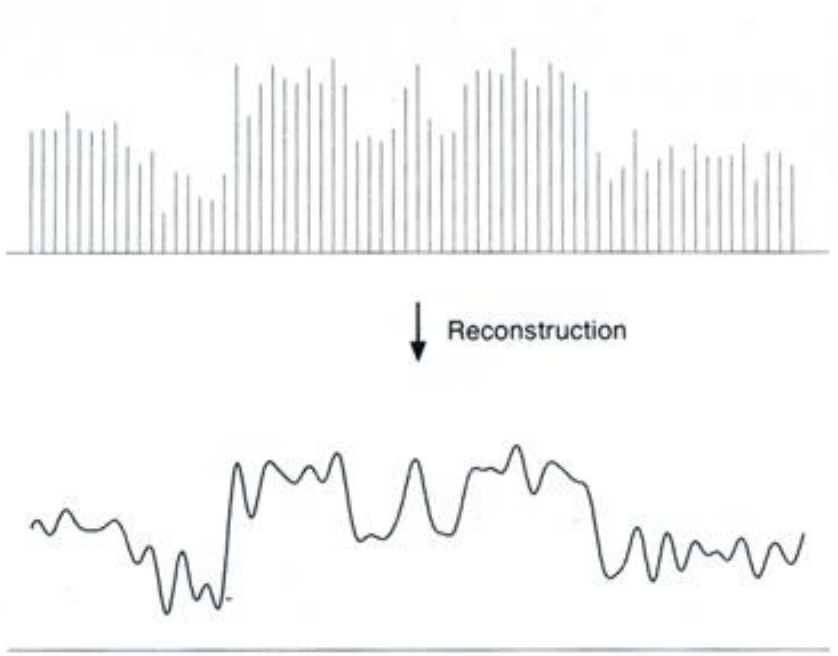


Sampling and reconstruction

Sampling: recording the function's values at a discrete set of locations



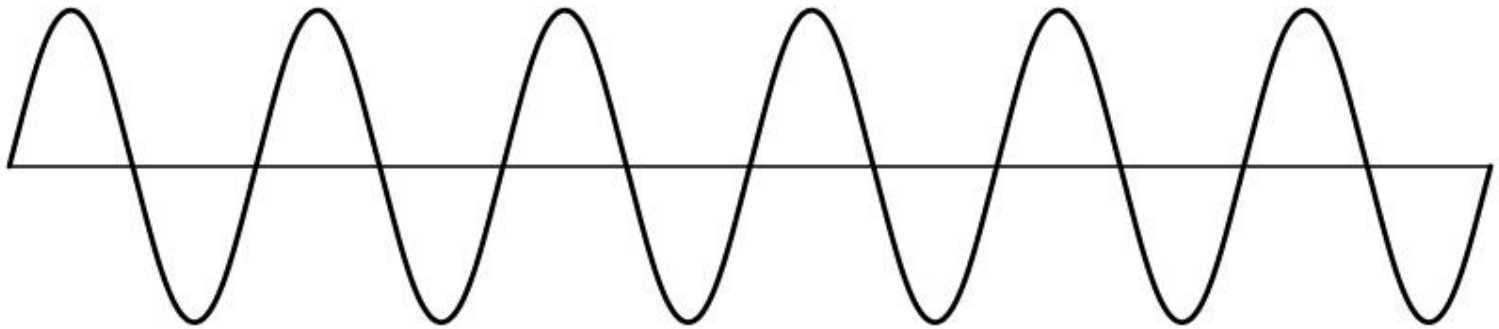
Reconstruction: converting a sampled representation back into a continuous function by "guessing" what happens between the samples



Source: S. Marschner (via A. Efros)

Sampling and reconstruction

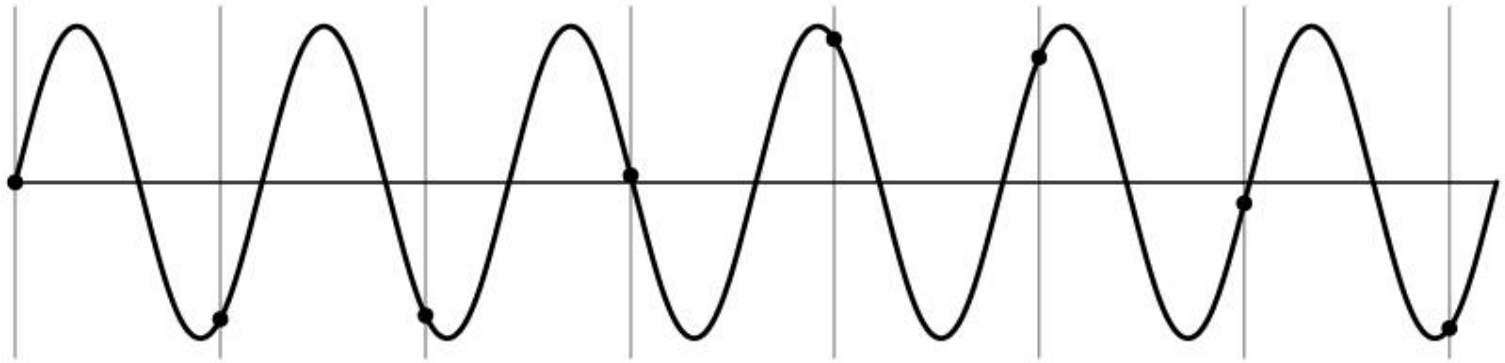
- Simple example: a sine wave



Source: S. Marschner (via A. Efros)

Sampling and reconstruction

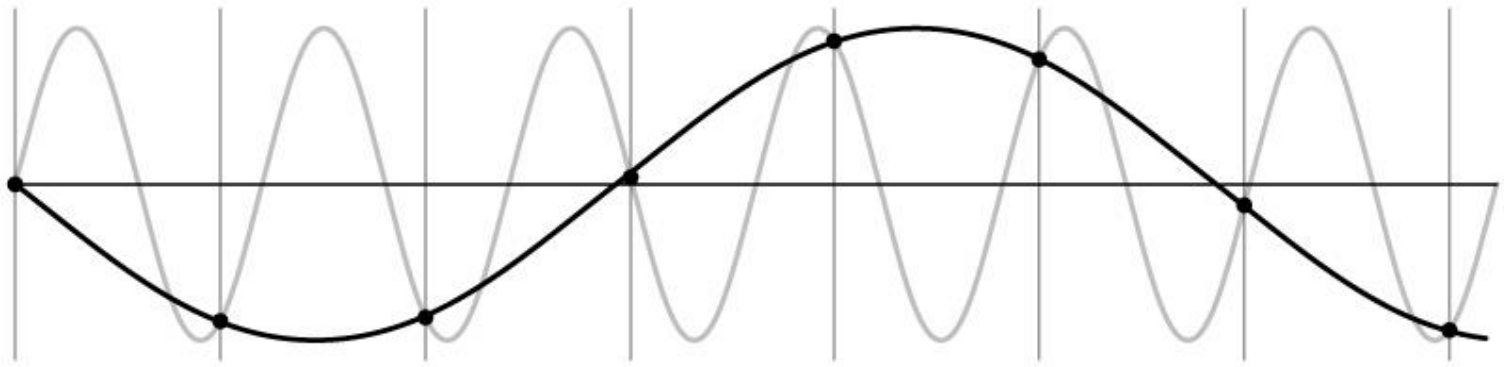
- Simple example: a sine wave
- What if we “missed” things between the samples?
  - Unsurprising result: information is lost



Source: S. Marschner (via A. Efros)

Sampling and reconstruction

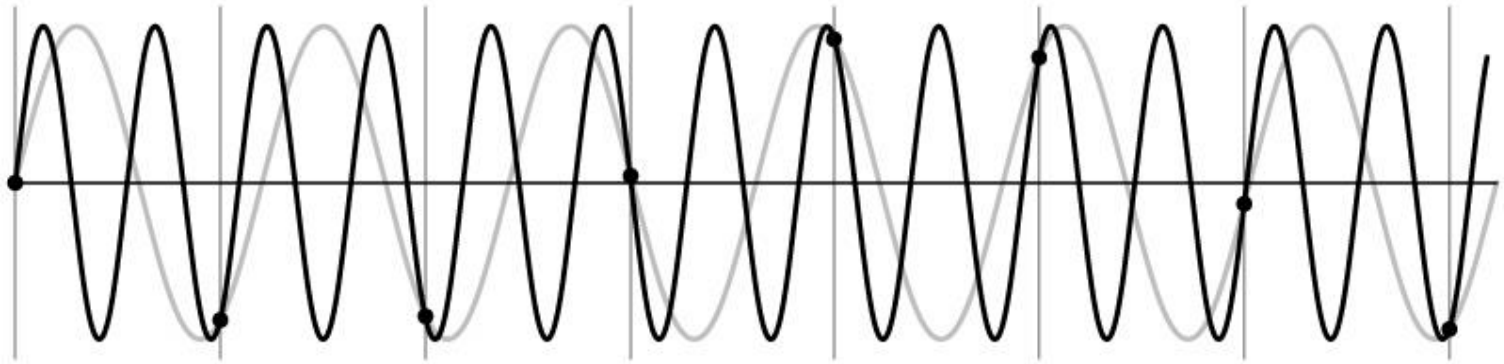
- Simple example: a sine wave
- What if we “missed” things between the samples?
  - Unsurprising result: information is lost
  - Surprising result: indistinguishable from lower frequencies



Source: S. Marschner (via A. Efros)

Sampling and reconstruction

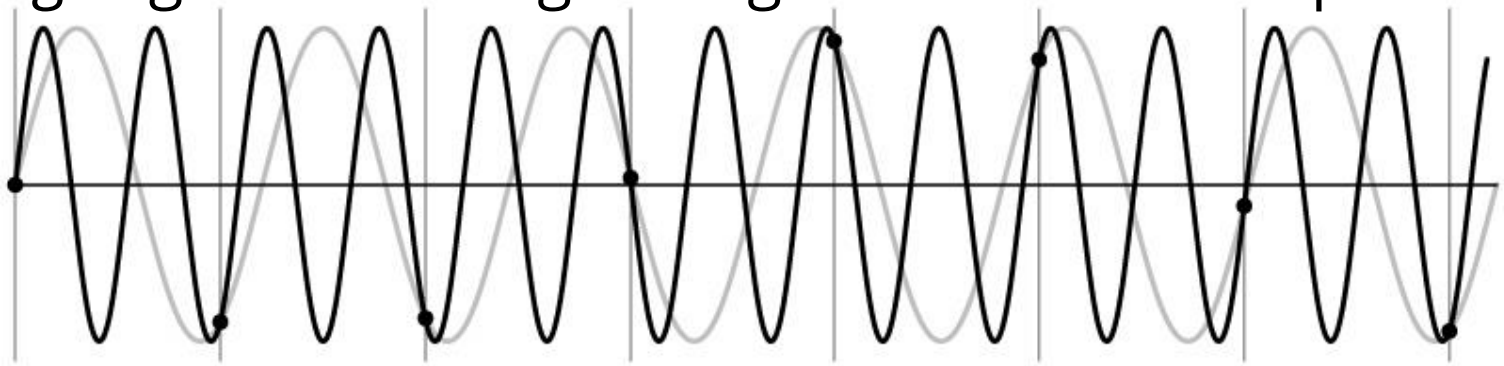
- Simple example: a sine wave
- What if we “missed” things between the samples?
  - Unsurprising result: information is lost
  - Surprising result: indistinguishable from lower frequencies (or even higher frequencies)



Source: S. Marschner (via A. Efros)

Sampling and reconstruction

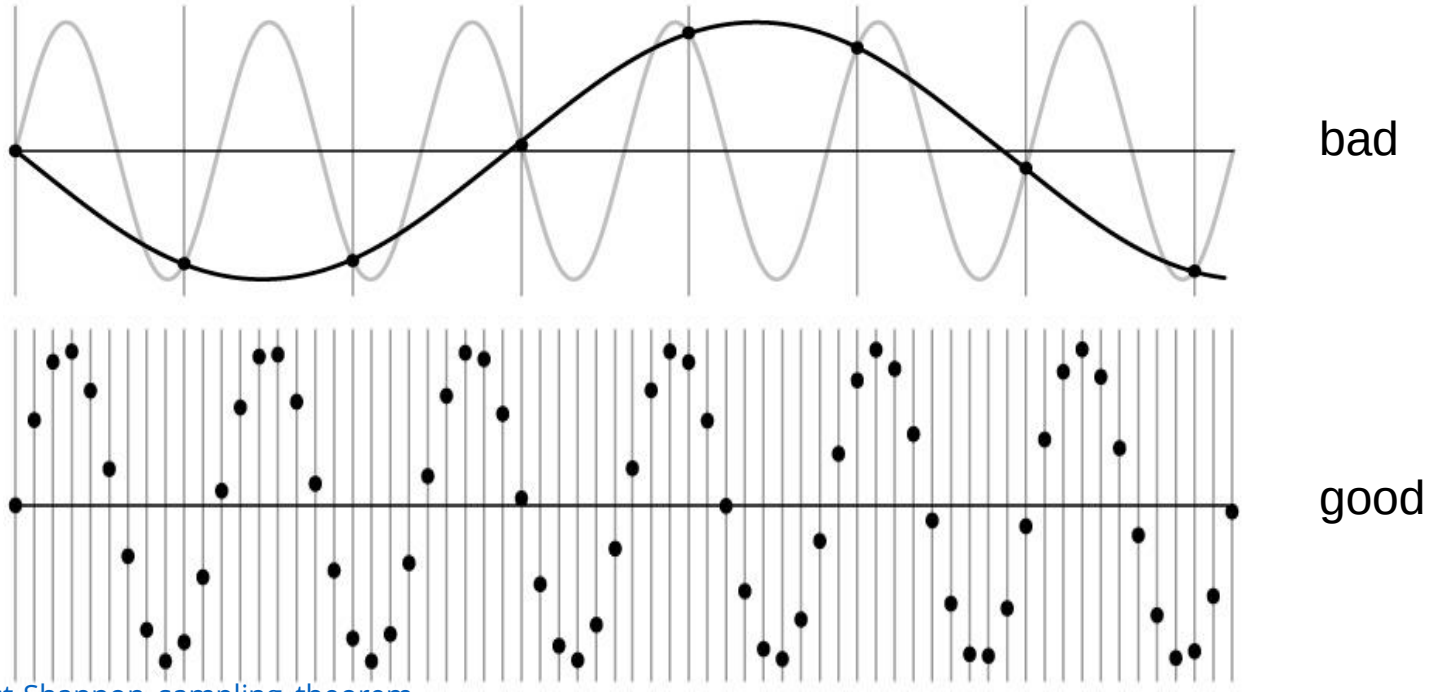
- Simple example: a sine wave
- What if we “missed” things between the samples?
  - Unsurprising result: information is lost
  - Surprising result: indistinguishable from lower frequencies (or even higher frequencies)
  - *Aliasing*: signal “traveling in disguise” as other frequencies



Source: S. Marschner (via A. Efros)

Nyquist-Shannon sampling theorem

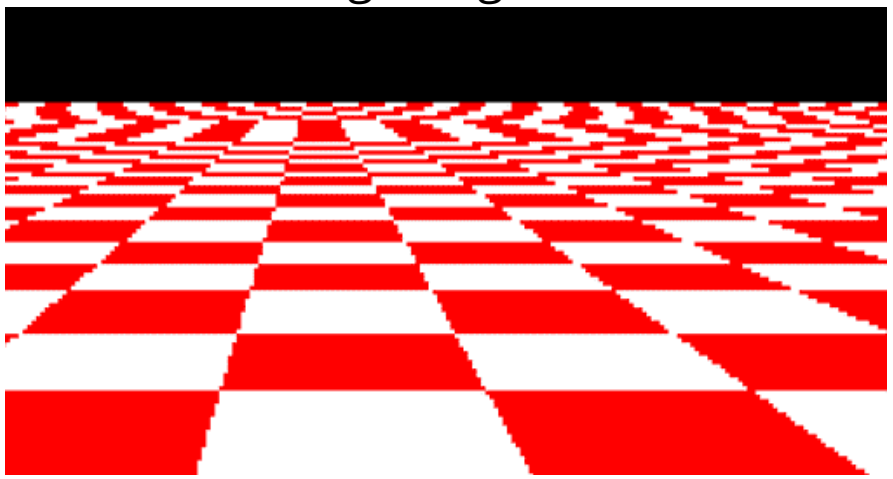
When sampling a signal at discrete intervals, the sampling frequency must be at least *twice* the maximum frequency of the input signal to allow us to reconstruct the original perfectly from the sampled version

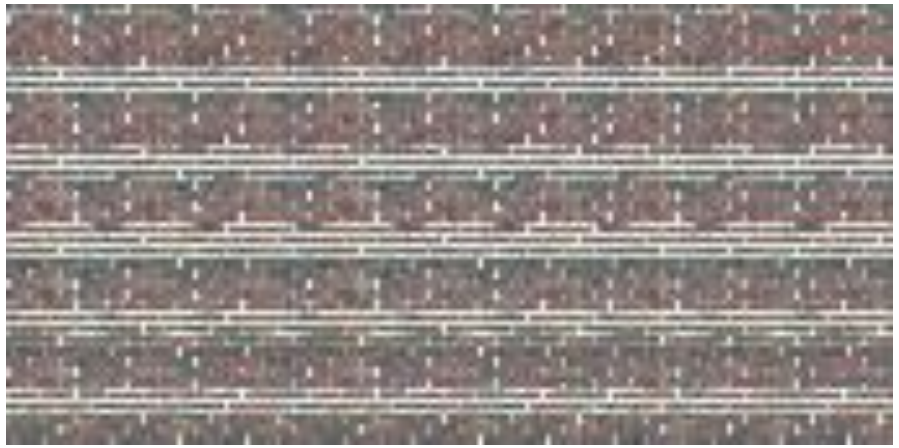


https://en.wikipedia.org/wiki/Nyquist-Shannon_sampling_theorem

Aliasing “in the wild”

Disintegrating textures

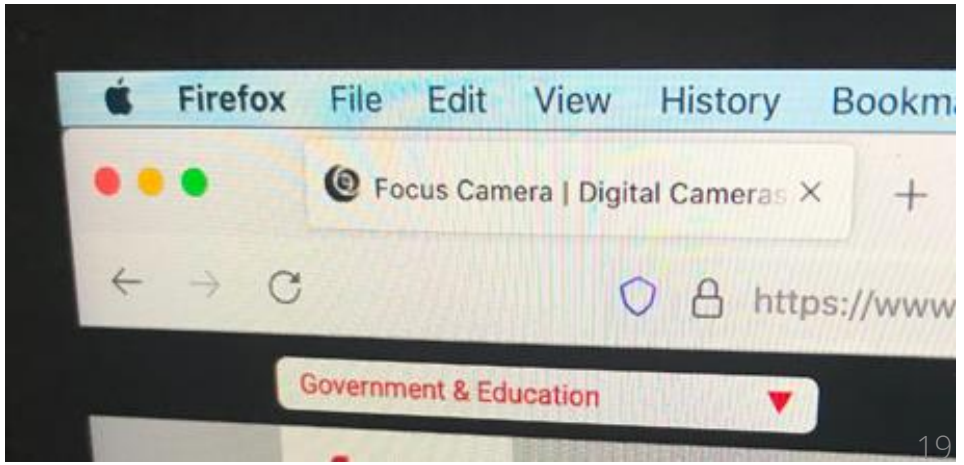




Source

Moiré patterns, false color

Source



Source

Anti-aliasing

What are possible solutions?

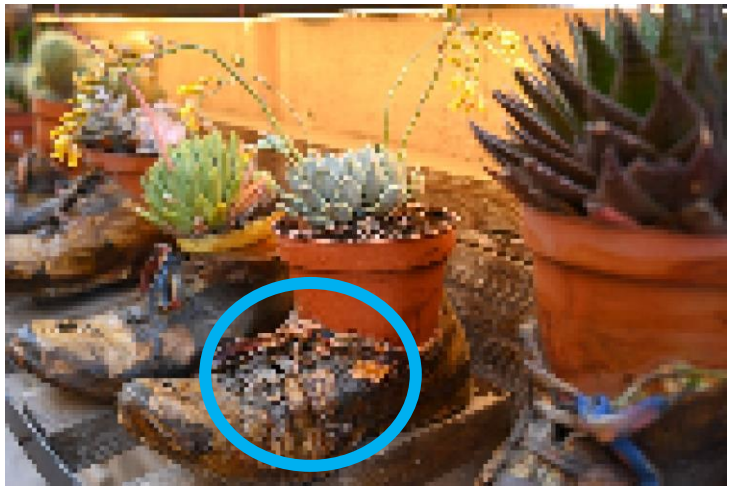
- Sample more often (if you can)

- Before sampling: get rid of all frequencies that are greater than half the new sampling frequency
  - Will lose information, but still better than aliasing
  - How to get rid of high frequencies?
    - Apply a smoothing or *low-pass* filter

Subsampling Images

- Goal: reduce the resolution of an image by a factor of 2^n
- Idea: let's delete every pixel with coordinates that are not a multiple of 2^n

factor 4



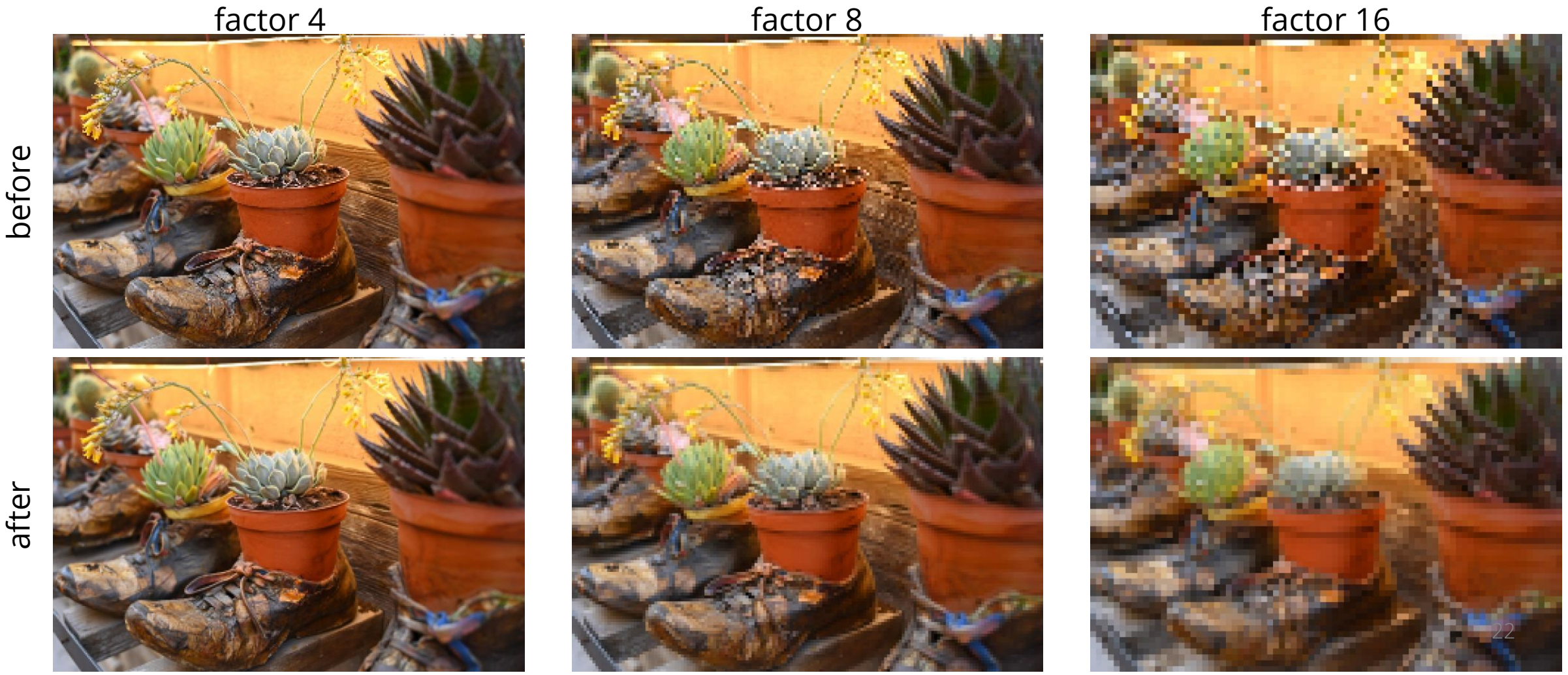
factor 8

factor 16

- Aliasing problems in high-frequency regions!

Subsampling Images

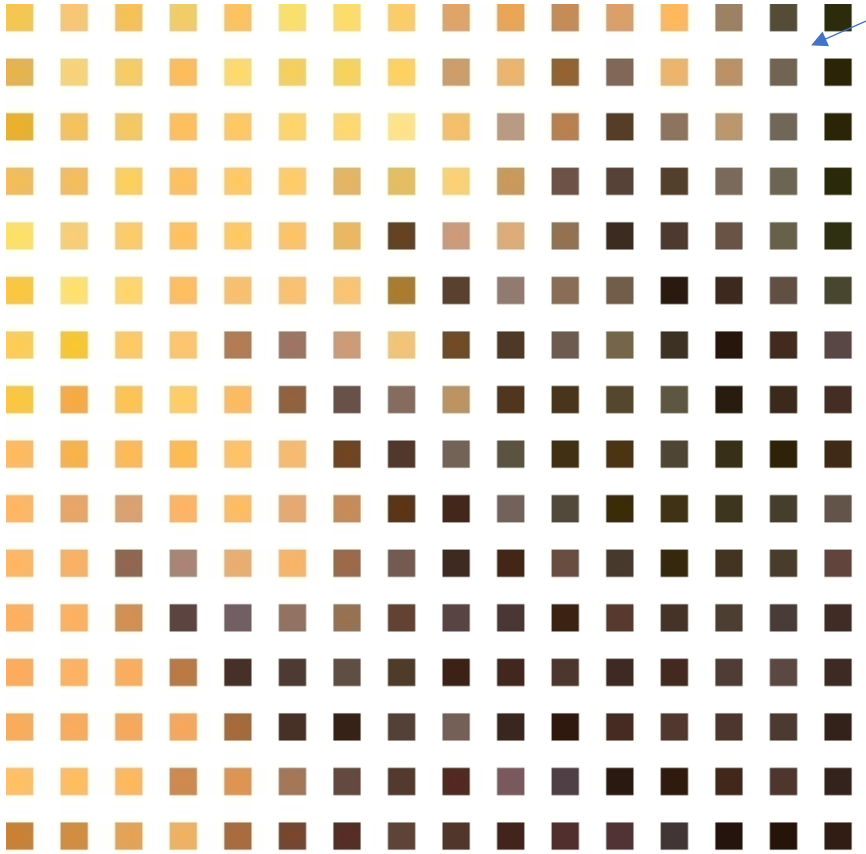
Idea: remove high-frequency details first (by blurring [later])



Upsampling Images

How to increase the resolution by a factor of 2

upsample

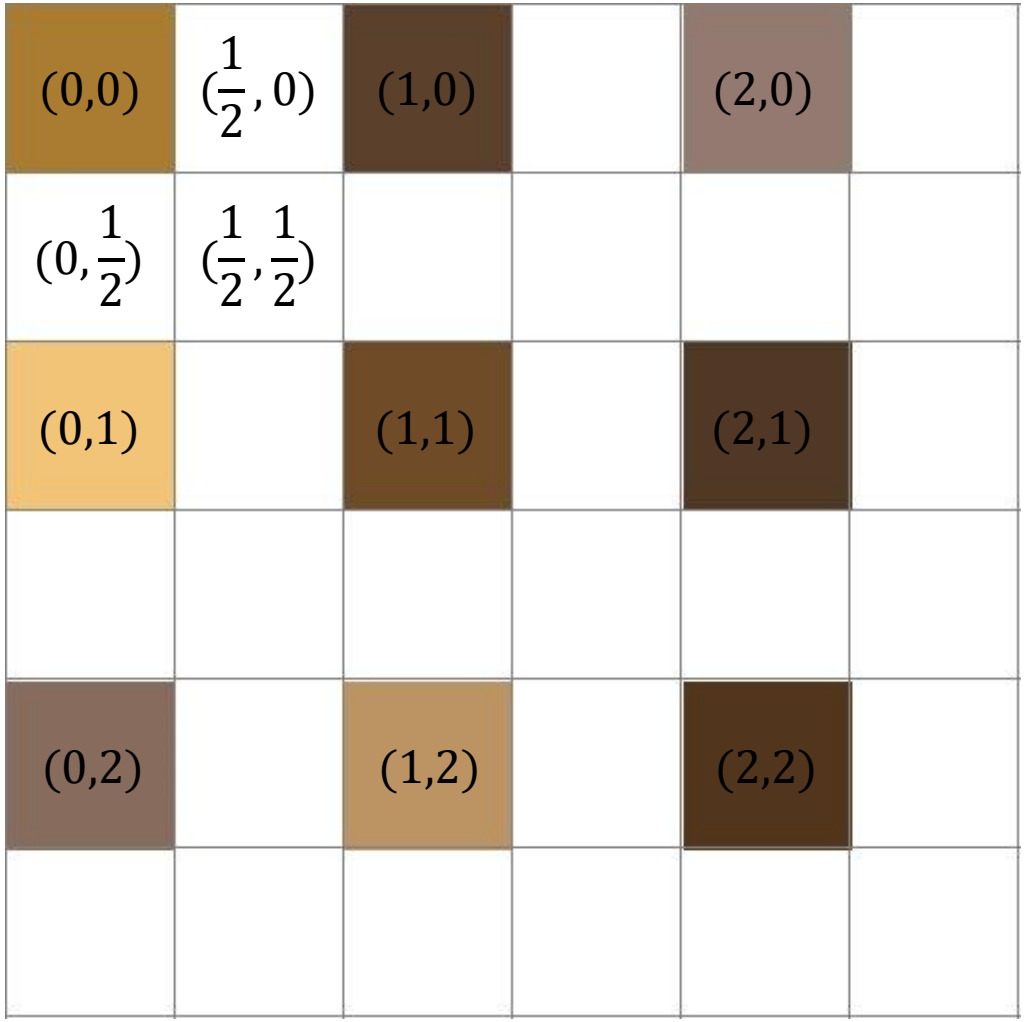


how do we determine the colours of the missing pixels?

Interpolation!

Interpolation

- We will express the image as a function $f(x, y)$ given at integer coordinates
- Upsampling by 2 means finding values for $f(x + \delta_x, y + \delta_y)$ with $\delta_x, \delta_y \in \{0, \frac{1}{2}\}$
- Define some shorthands:
  $\mathrm{A} \coloneqq f(0,0)$ $\mathrm{B} \coloneqq f(1,0)$
  $\mathrm{C} \coloneqq f(0,1)$ $\mathrm{D} \coloneqq f(1,1)$
- Here: math will be with grayscale images only. In practice: process each channel separately



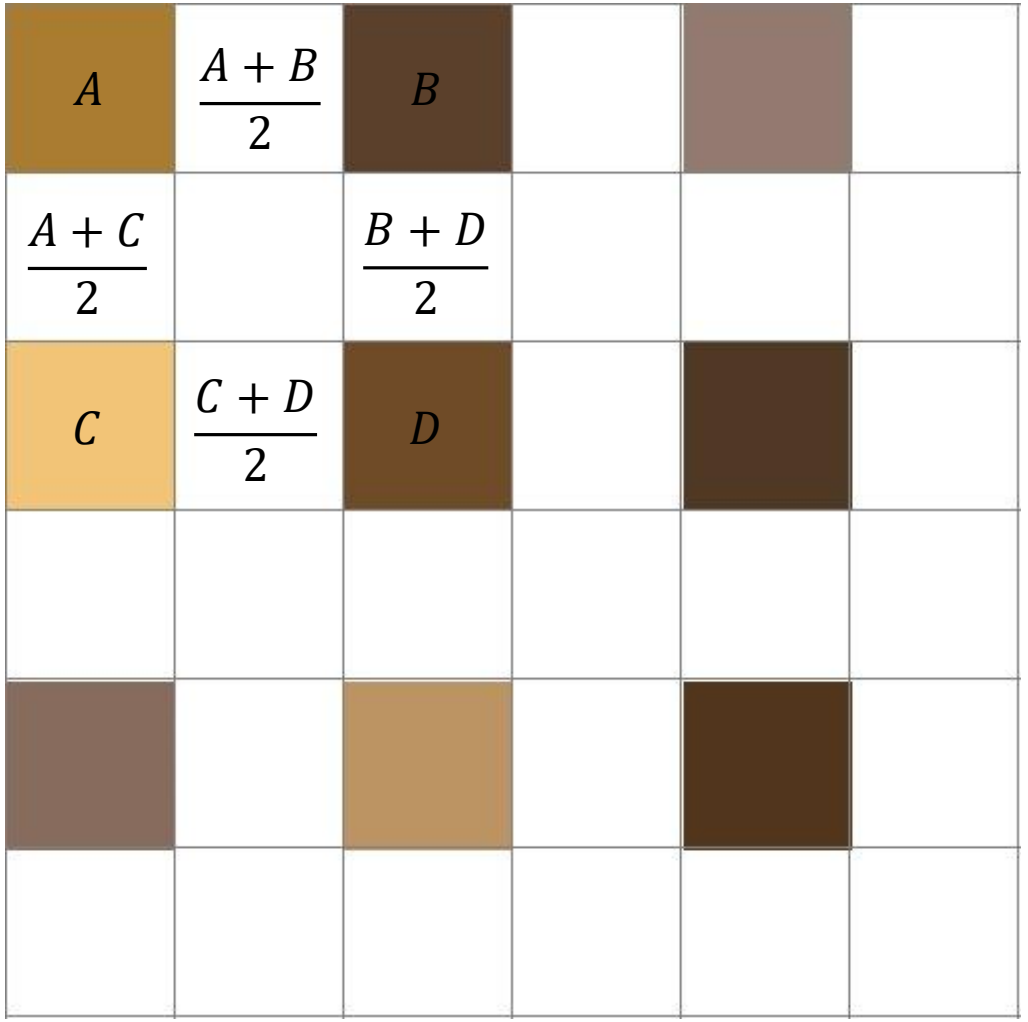
Bilinear Interpolation

- Intuition: new values should lie *between* existing ones:

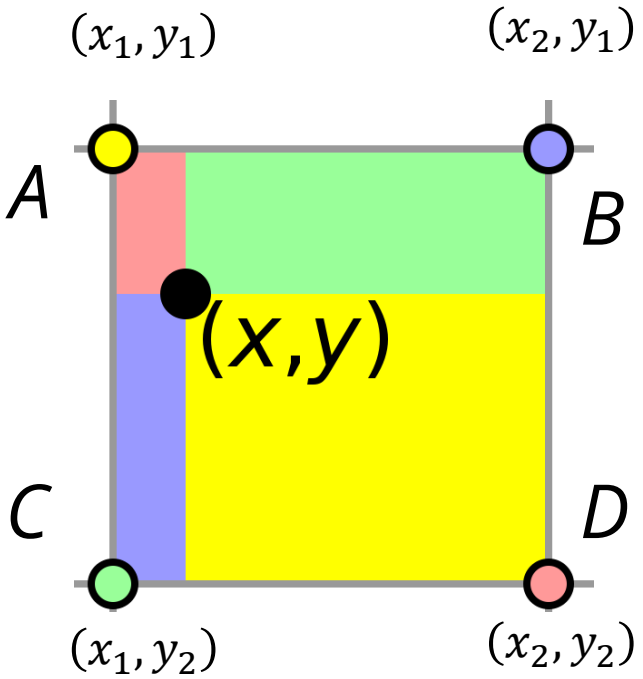
$$f\left(\frac{1}{2},0\right) = \frac{f(0,0) + f(1,0)}{2}$$

- After filling in the middles between known points, centres between 4 pixels can be filled. Well defined:

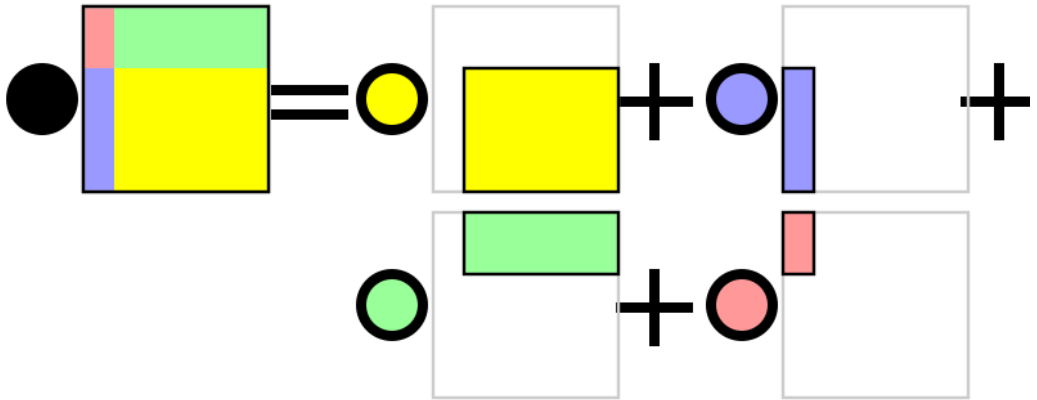
$$\frac{\frac{A+B}{2} + \frac{C+D}{2}}{2} = \frac{A+B+C+D}{4} = \frac{\frac{A+C}{2} + \frac{B+D}{2}}{2}$$



Generalized Bilinear Interpolation



Intuition:



Formalized:

$$f(x,y) = w_{11}A + w_{21}B + w_{12}C + w_{22}D$$

$$w_{11} = \frac{(x_2 - x)(y_2 - y)}{(x_2 - x_1)(y_2 - y_1)} \qquad w_{21} = \frac{(x - x_1)(y_2 - y)}{(x_2 - x_1)(y_2 - y_1)}$$

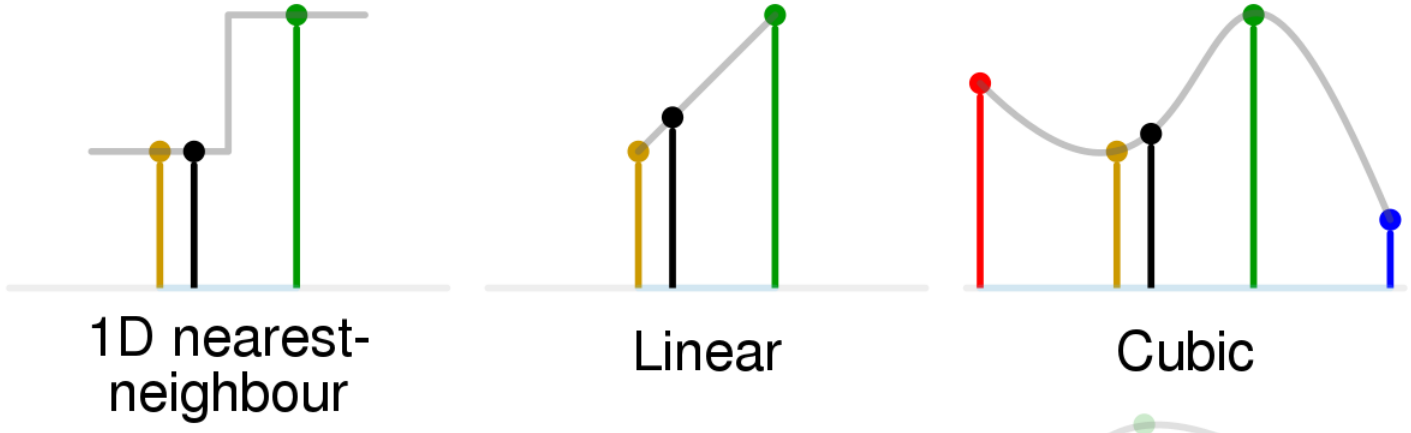
$$w_{12} = \frac{(x_2 - x)(y - y_1)}{(x_2 - x_1)(y_2 - y_1)} \qquad w_{22} = \frac{(x - x_1)(y - y_1)}{(x_2 - x_1)(y_2 - y_1)}$$

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

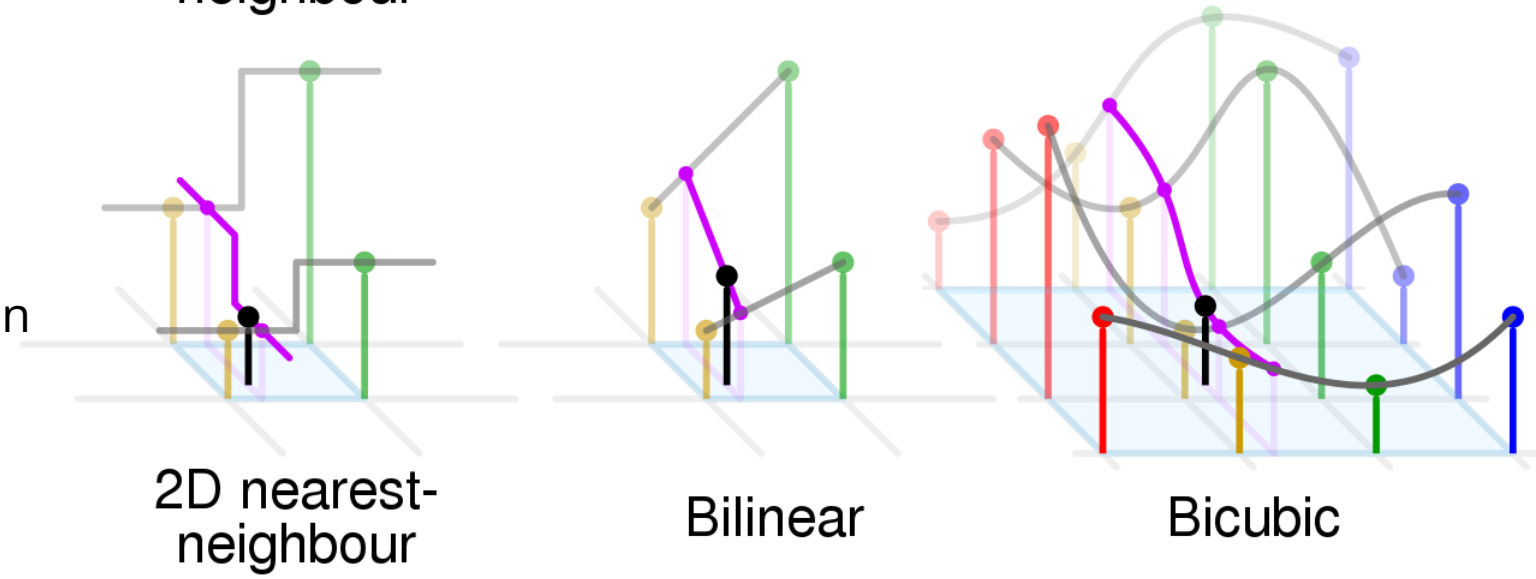
adapted from from S. Lazebnik

Other Interpolation Schemes

1D Interpolation



2D Interpolation



Source: Wikipedia

Useful Interpolation Properties

- Nearest Neighbour Interpolation
  - Does only use values already in the data
- (Bi-) Linear Interpolation
  - Does not create samples outside of the range of interpolants
- (Bi-) Cubic Interpolation
  - Is smooth (differentiable) everywhere

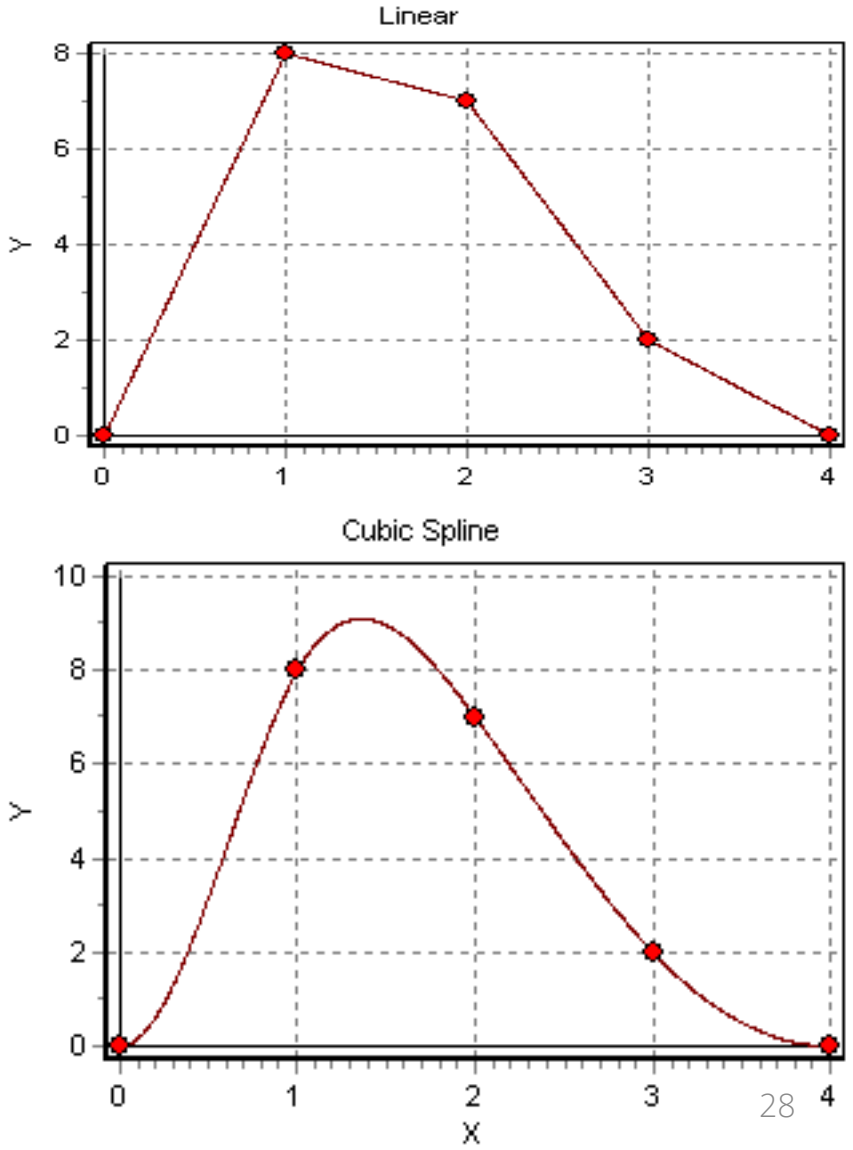


Image source

Image Transformations

- Images as functions can help formulating resampling.
- The functional representation allows us to do other transformations too!
- A transformation creates a new image f' from f.

- Point-wise transformation: $f'(x, y) = t(f(x, y))$
- Geometric transformation: $f'(x, y) = f(T(x, y))$
- Image filtering: $f'(x, y) = F(N(x, y))$, for a neighbourhood $N(x, y) = \{f(u, v) | \text{“}(u, v) \text{ is a neighbour of } (x, y)\text{”}\}$

Point-Wise Transformations

- $f'(x, y) = t(f(x, y))$
- Changes the **range** of the image

- Negative: $f' = 1 - f$

Point-Wise Transformations

- $f'(x, y) = t(f(x, y))$
- Changes the **range** of the image

- Negative: $f' = 1 - f$
- Contrast: $f' = af + b$

Point-Wise Transformations

- $f'(x,y) = t(f(x,y))$
- Changes the **range** of the image

- Negative: $f' = 1 - f$
- Contrast: $f' = af + b$
- Gamma correction:

Geometric Transformations

- $f'(x, y) = f(T(x, y))$
- Changes the **domain** of the image

- Translation:

$T(x, y) = (x + \delta_x, y + \delta_y)$

dx=0.0, dy=0.0

Geometric Transformations

- $f'(x, y) = f(T(x, y))$
- Changes the **domain** of the image

- Translation:
  $T(x, y) = (x + \delta_x, y + \delta_y)$
- Scaling: $T(x, y) = (sx, sy)$

s=1.0

Geometric Transformations

- $f'(x, y) = f(T(x, y))$
- Changes the **domain** of the image

- Translation:
  $T(x, y) = (x + \delta_x, y + \delta_y)$
- Scaling: $T(x, y) = (sx, sy)$
- Rotation:
  $T(x, y) = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$

angle=0.0

General Geometric Transformations

- We can express these (and more) geometric transformations in a unified manner
- Homogeneous coordinates: $\begin{pmatrix} x \\ y \end{pmatrix} \rightarrow \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$

- Affine transformations:

$$T(x, y) = A \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

Affine Transformation Examples

- Translation: $T(x, y) = (x + \delta_x, y + \delta_y) = \begin{pmatrix} 1 & 0 & \delta_x \\ 0 & 1 & \delta_y \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$

- Scaling: $T(x, y) = (sx, sy) = \begin{pmatrix} s & 0 & 0 \\ 0 & s & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$

- Rotation: $T(x, y) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$

Affine Transformation Examples

- Horizontal Shearing:

$$T(x,y) = \begin{pmatrix} 1 & m & 0 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

m=0.0

- Vertical Shearing:

$$T(x,y) = \begin{pmatrix} 1 & 0 & 0 \\ m & 1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

m=0.0

Combining Transformations

- How can we chain transformations? E.g. rotation and translation.
- Same idea: Homogeneous coordinates.
- Add a row to the matrix to make it square.

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix} \rightarrow \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{pmatrix}$$

Affine Transformation Matrix

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} a_{11}x + a_{12}y + a_{13} \\ a_{21}x + a_{22}y + a_{23} \\ 1 \end{pmatrix}$$

We can essentially ignore the additional row and 1. (for now!)

An affine transformation preserves:

- **Collinearity**: three or more points which lie on the same line (called collinear points) continue to be collinear after the transformation.
- **Parallelism**: two or more lines which are parallel, continue to be parallel after the transformation.
- **Convexity** of sets: a convex set continues to be convex after the transformation.

Combining Transformations

- Affine transformations in 2D are 3x3 matrices with 6 variables.

- It is a group under composition of functions:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ 0 & 0 & 1 \end{pmatrix}$$

Combining Transformations

- Simply multiply transformation matrices.
- Order matters! (Applied right-to-left because we multiply points right)

Example – Rotation & Translation:

$$\mathrm{R} = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \qquad T = \begin{pmatrix} 1 & 0 & \delta_x \\ 0 & 1 & \delta_y \\ 0 & 0 & 1 \end{pmatrix}$$

$$RT = \begin{pmatrix} \cos\theta & -\sin\theta & \delta_x\cos\theta - \delta_{\mathrm{y}}\sin\theta \\ \sin\theta & \cos\theta & \delta_x\sin\theta + \delta_y\cos\theta \\ 0 & 0 & 1 \end{pmatrix} \neq TR = \begin{pmatrix} \cos\theta & -\sin\theta & \delta_x \\ \sin\theta & \cos\theta & \delta_y \\ 0 & 0 & 1 \end{pmatrix}$$

Rotation and Translation

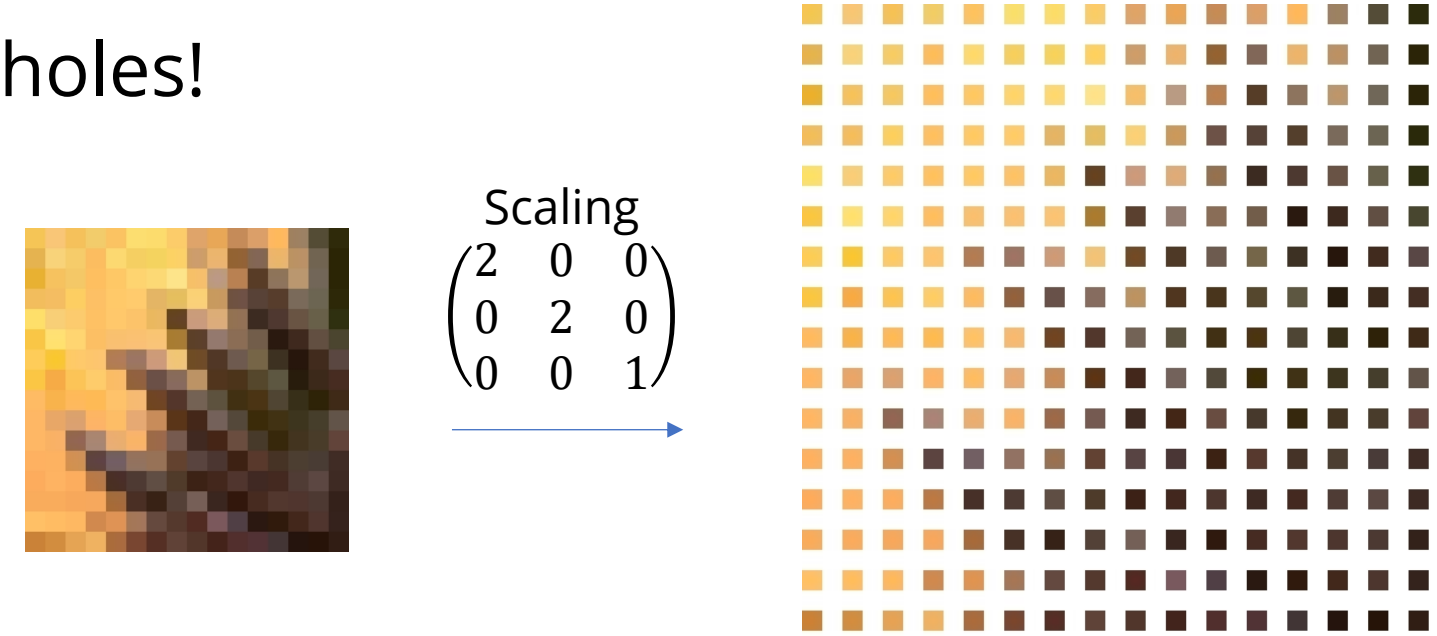
Animating rotation, fixed x-translation

TR: rotate then translate

RT: translate then rotate

How did I create these animations?

- Transformations define a *forward warp.*
- $T(x, y) = (x', y')$ maps source pixels to target locations
- A naïve implementation takes every source pixel and draws it at its new location.
- Problem: holes!



Backward Warps

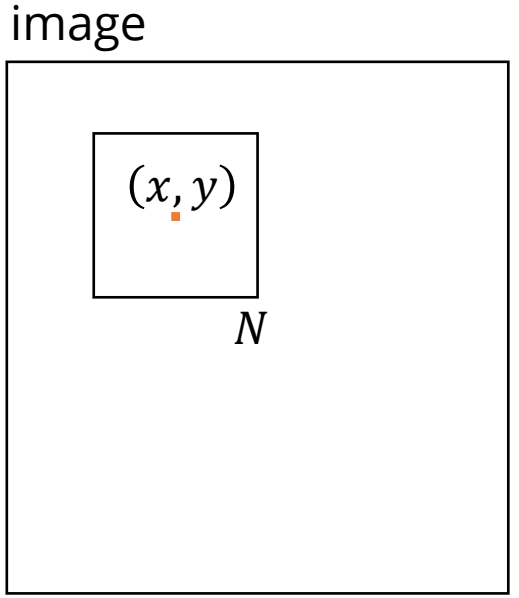
- Iterating over source pixels and drawing them at the target location is called a *forward warp*.
- Often better: for every target-image pixel: look up where it comes from – this is a *backwards warp*.
- We can compute backwards warps with a matrix inverse:

$$T^{-1}(x, y) = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{pmatrix}^{-1} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

- If the source location is between pixels: use interpolation!

Image Filtering

Idea of filtering: $f'(x,y) = F(N(x,y))$
for a set $N(x,y) = \{(u,v) |$ "(u,v) is neighbour of (x,y)"$\}$



- We replace each point with some computation on its neighbourhood.
- Neighbours often in a square: $|x-u| \leq n$ and $|y-v| \leq n$. (total size is $(2n+1)^2$ pixels)
- F is called a *filter*.
- Usually, the same filter is applied everywhere.

Averaging

- Earlier we removed high frequency details by blurring.
- The easiest way to blur an image is by averaging a neighbourhood.

- Simple blur: $F(N) = \frac{1}{|N|} \sum_{(u,v) \in N} f(u, v)$

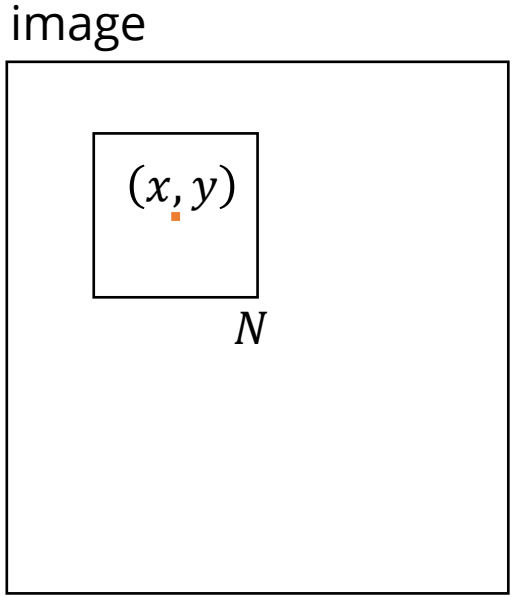
5x5

17x17

33x33

Gaussian Blur

- Averaging gives the same weight to every pixel in the neighbourhood.
- It is better to discount pixels when they are further away.



weight

$$F(N) = \underbrace{\frac{1}{\sum_{(u,v)\in N} w(u,v)}}_{\text{normalization}} \sum_{(u,v)\in N} w(u,v) f(u,v)$$

Gaussian Blur: $w(u,v) = e^{-\frac{(x-u)^2+(y-v)^2}{2\sigma^2}}$

Averaging vs. Gaussian Blur

averaging

Gaussian Blur

Discrete Convolution

Given discrete functions $f: \mathbb{Z} \mapsto \mathbb{R}$ and $g: \mathbb{Z} \mapsto \mathbb{R}$, their convolution is defined as

$$h[x] = (f * g)[x] = \sum_{u=-\infty}^{+\infty} f[u]g[x-u]$$

Discrete Convolution

If f has finite support in the set $\mathcal{M} = \{0, \dots, M-1\}$,
i.e., $f[m] = 0, \forall\, m \in \mathbb{Z} \backslash \mathcal{M}$, then:

$$h[x] = (f * g)[x] = \sum_{u=0}^{M-1} f[u] g[x - u + \frac{M-1}{2}]$$

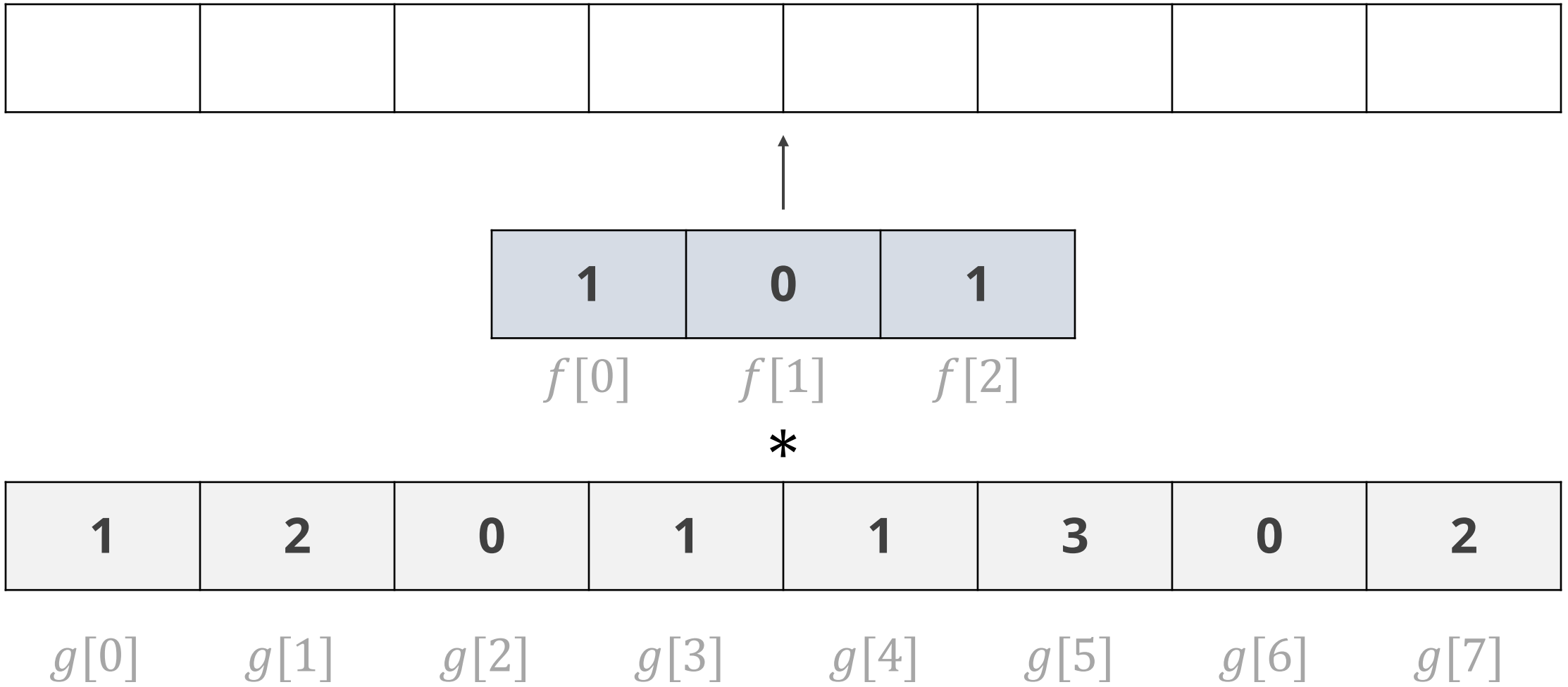
f: Kernel or filter

g: Input function

Example

$$h[x] = (f * g)[x] = \sum_{u=0}^{M-1} f[u]g[x-u+1]$$

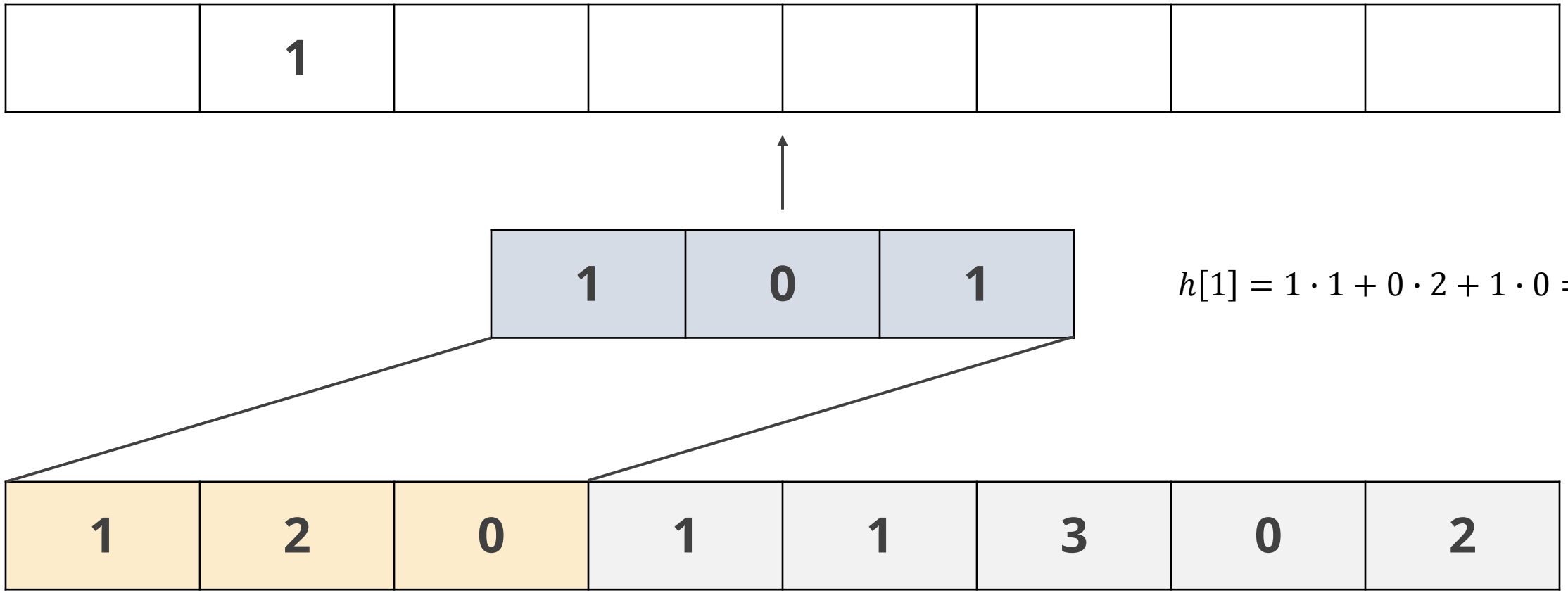
$$M = 3, x \in [0,7]$$



Example

$$h[1] = (f * g)[1] = \sum_{u=0}^{2} f[u]g[1-u+1]$$

$$M = 3, x \in [0,7]$$

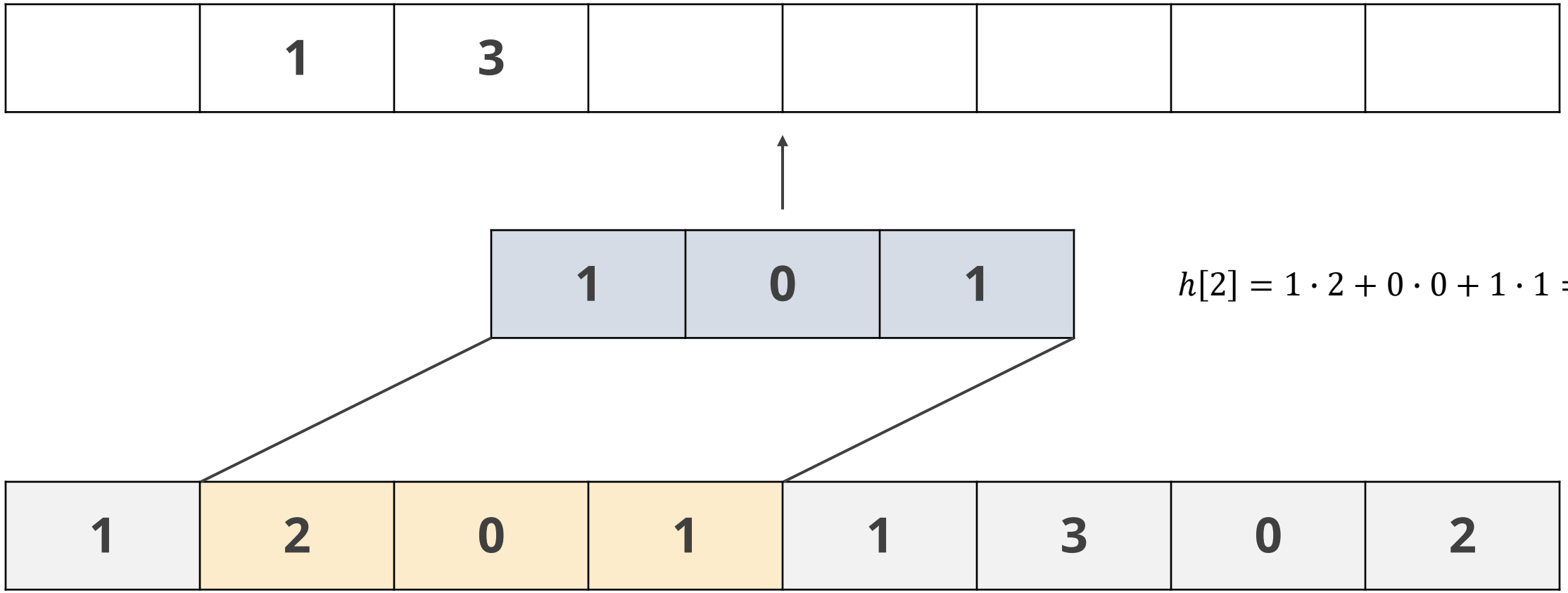


$$h[1] = 1 \cdot 1 + 0 \cdot 2 + 1 \cdot 0 = 1$$

Example

$$h[2] = (f * g)[2] = \sum_{u=0}^{2} f[u]g[2-u+1]$$

$$M = 3, x \in [0,7]$$

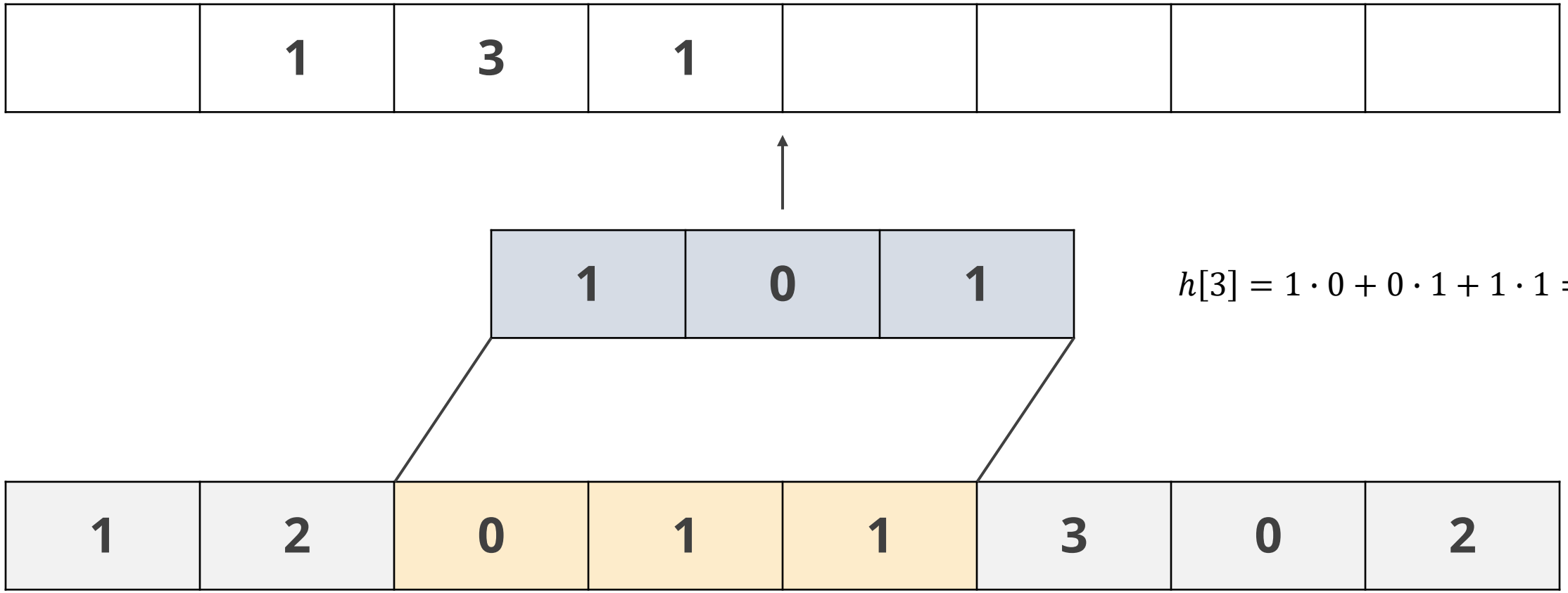


$$h[2] = 1 \cdot 2 + 0 \cdot 0 + 1 \cdot 1 = 3$$

Example

$$h[3] = (f * g)[3] = \sum_{u=0}^{2} f[u]g[3-u+1]$$

$$M = 3, x \in [0,7]$$

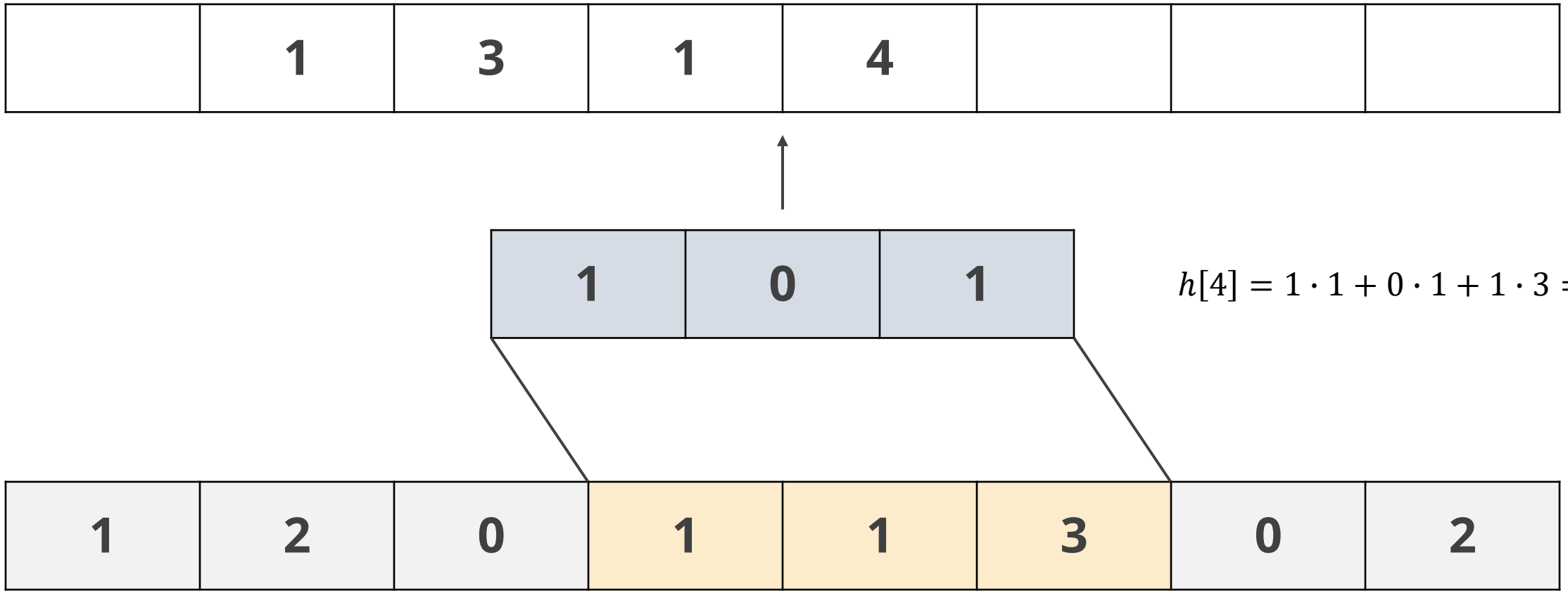


$$h[3] = 1 \cdot 0 + 0 \cdot 1 + 1 \cdot 1 = 1$$

Example

$$h[4] = (f * g)[4] = \sum_{u=0}^{2} f[u]g[4-u+1]$$

$$M = 3, x \in [0,7]$$

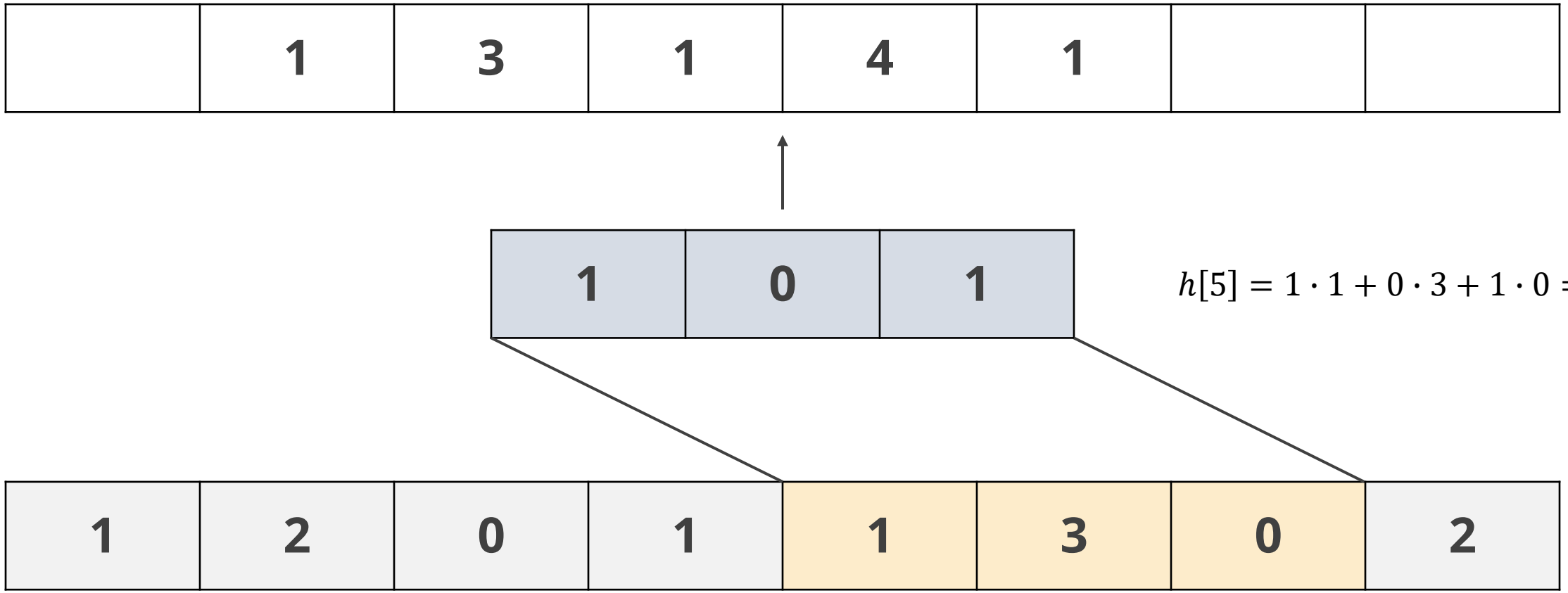


$$h[4] = 1 \cdot 1 + 0 \cdot 1 + 1 \cdot 3 = 4$$

Example

$$h[5] = (f * g)[5] = \sum_{u=0}^{2} f[u]g[5-u+1]$$

$$M = 3, x \in [0,7]$$

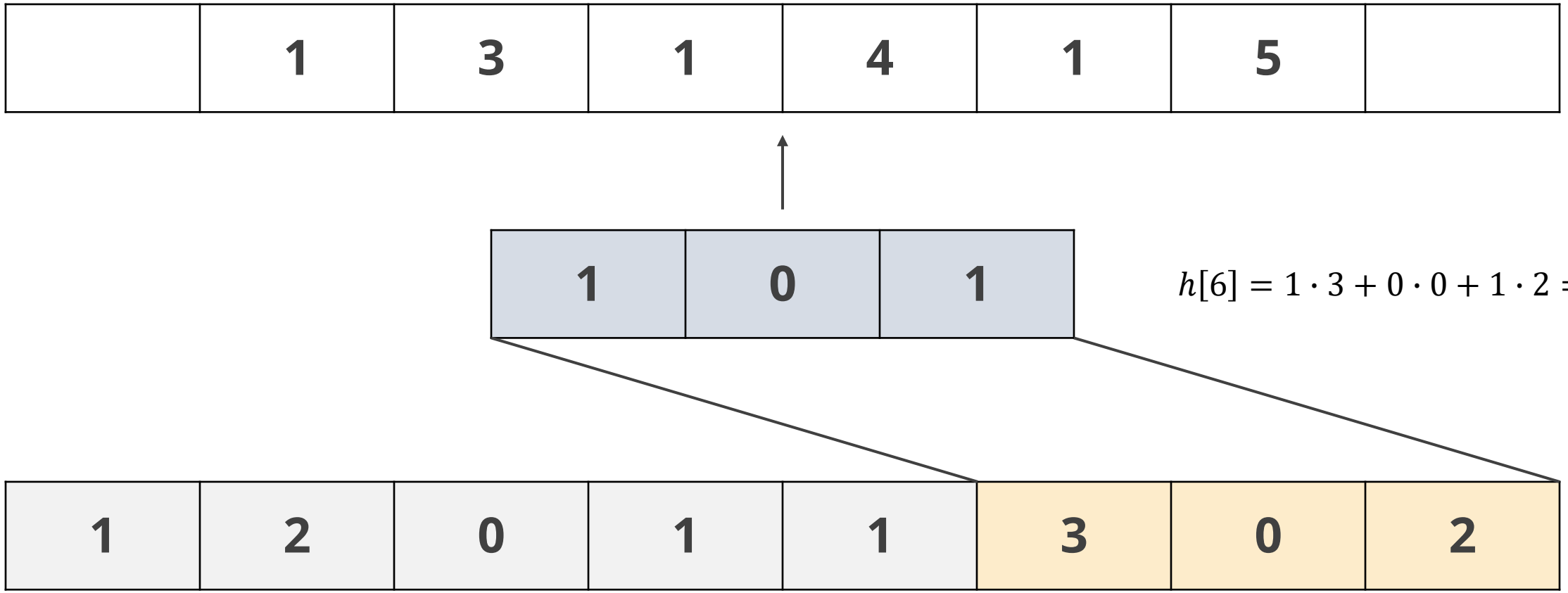


$$h[5] = 1 \cdot 1 + 0 \cdot 3 + 1 \cdot 0 = 1$$

Example

$$h[6] = (f * g)[6] = \sum_{u=0}^{2} f[u]g[6-u+1]$$

$$M = 3, x \in [0,7]$$

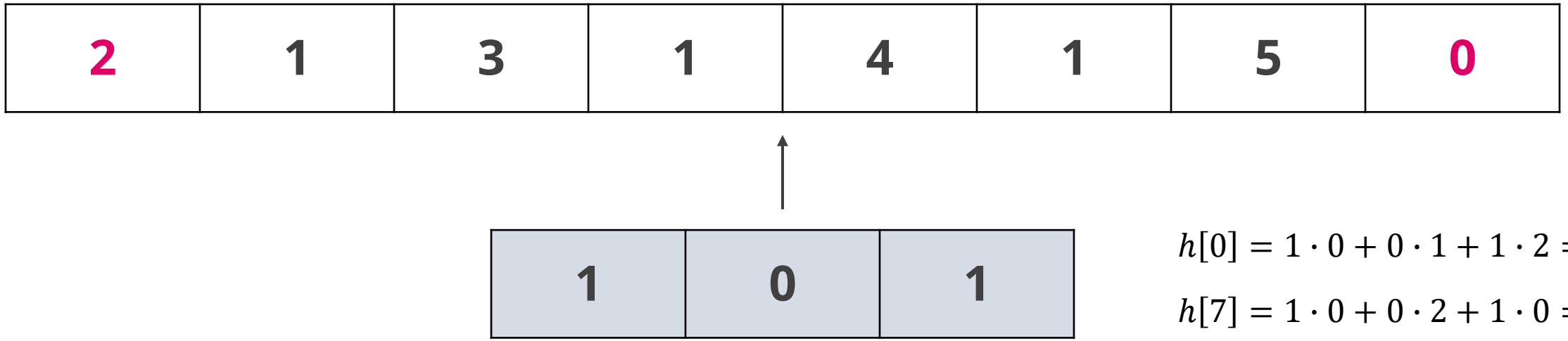


$$h[6] = 1 \cdot 3 + 0 \cdot 0 + 1 \cdot 2 = 5$$

Example

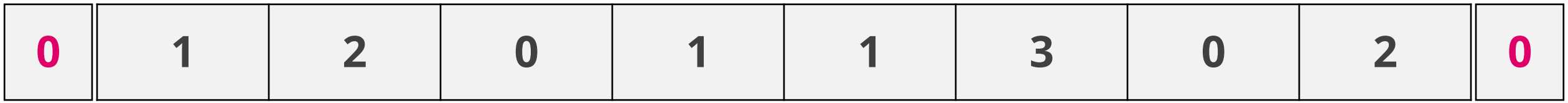
$$h[x] = (f * g)[x] = \sum_{u=0}^{2} f[u]g[x-u+1]$$

$$M = 3, x \in [0,7]$$



$$h[0] = 1 \cdot 0 + 0 \cdot 1 + 1 \cdot 2 = 2$$

$$h[7] = 1 \cdot 0 + 0 \cdot 2 + 1 \cdot 0 = 0$$



Boundary case: zero-padding

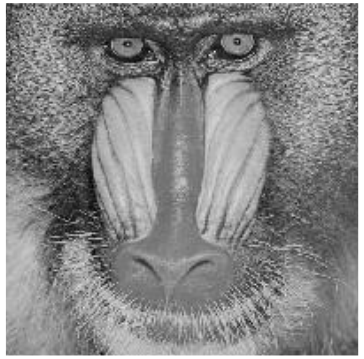
Multidimensional Discrete Convolution

If $g: \mathbb{Z} \times \mathbb{Z} \mapsto \mathbb{R}$ and $f: \{0, \dots, M-1\} \times \{0, \dots, N-1\} \mapsto \mathbb{R}$ are discrete functions of two variables, then:

$$h[x, y] = (f * g)[x, y] = \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} f[u, v] g[x - u + \frac{M-1}{2}, y - v + \frac{M-1}{2}]$$

Example

We can think of convolution as performing an image operation defined by the filter

$g =$ 

$f * g =$ 

$f =$

$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$
$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$
$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$

Mean filter
(Smoothing)

Median Filtering

- We can apply more complex functions than simply weighted averages.
- Median Filtering: $F(N) = \text{median}_{(u,v)\in N}\,(f(u,v))$
- Median: sort all elements and take “the middle one”
- Good for outlier removal

input: 20% corrupted pixels

3x3 median filtering

17x17 median filtering

Bilateral Filter

- Gaussian Blur blurs everything equally. Sometimes we want to preserve edges/boundaries.
- Use two weights: $w(u,v) = w_g(u,v)w_s(u,v)$

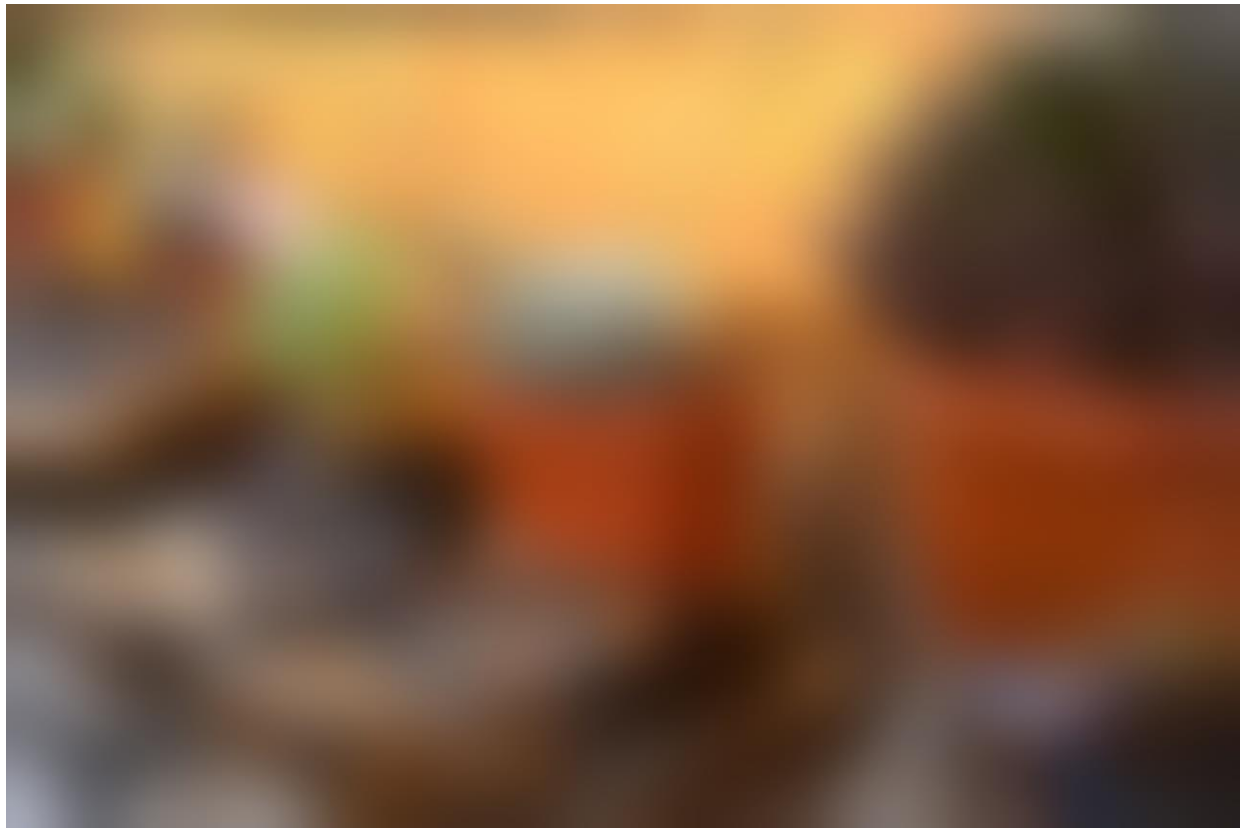
$w_g(u,v) = e^{-\frac{(x-u)^2+(y-v)^2}{2\sigma_g^2}}$ (like Gaussian Blur)

$w_s(u,v) = e^{-\frac{\left(f(u,v)-f(x,y)\right)^2}{2\sigma_s^2}}$ (weigh similar pixels higher)

- Bilateral Filter: $F(N) = \frac{1}{\sum_{(u,v)\in N} w(u,v)} \sum_{(u,v)\in N} w(u,v)f(u,v)$

Bilateral Filter

Bilateral 35x35



Gaussian Blur 33x33