What is an image?

Matthew Bryan

CEA-Leti, Grenoble, France

matthew.bryan@cea.fr

GitHub: @matbryan52 / microscopy-images-qem

Slides: matbryan52.github.io/microscopy-images-qem

Who I am

Matthew Bryan

@matbryan52 on GitHub

Research Software Engineer

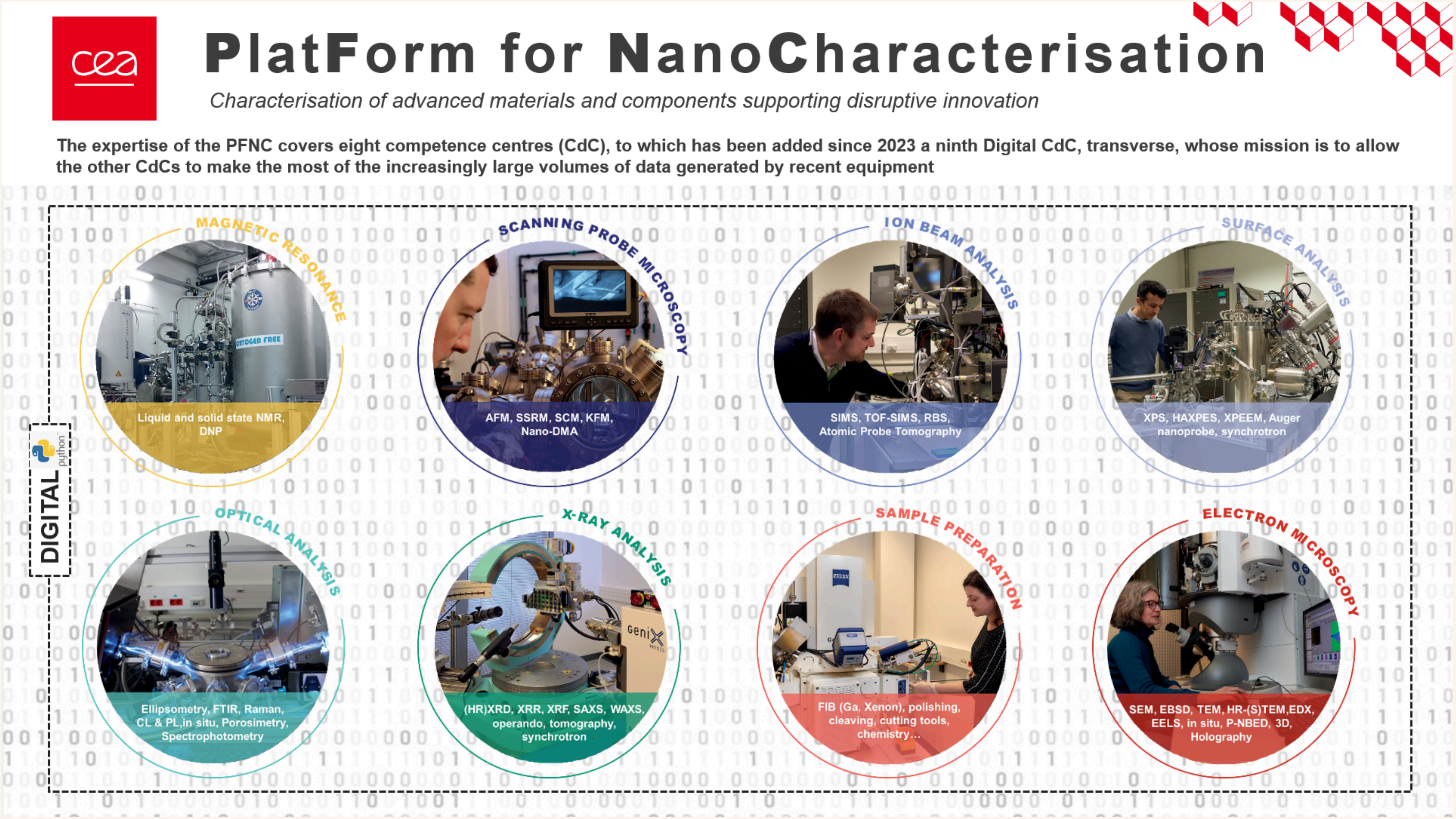
cea leti   Grenoble 🇫🇷   Alps 🏔️

Background:

- fluids + engineering
- image processing
- computer vision

Not really a Microscopist!

Developer on the LiberTEM project



PlatForm for NanoCharacterisation

Characterisation of advanced materials and components supporting disruptive innovation

The expertise of the PFNC covers eight competence centres (CdC), to which has been added since 2023 a ninth Digital CdC, transverse, whose mission is to allow the other CdCs to make the most of the increasingly large volumes of data generated by recent equipment

MAGNETIC RESONANCE: Liquid and solid state NMR, DNP

SCANNING PROBE MICROSCOPY: AFM, SSRM, SCM, KFM, Nano-DMA

ION BEAM ANALYSIS: SIMS, TOF-SIMS, RBS, Atomic Probe Tomography

SURFACE ANALYSIS: XPS, HAXPES, XPEEM, Auger nanoprobe, synchrotron

OPTICAL ANALYSIS: Ellipsometry, FTIR, Raman, CL & PL,in situ, Porosimetry, Spectrophotometry

X-RAY ANALYSIS: (HR)XRD, XRR, XRF, SAXS, WAXS, operando, tomography, synchrotron

SAMPLE PREPARATION: FIB (Ga, Xenon), polishing, cleaving, cutting tools, chemistry…

ELECTRON MICROSCOPY: SEM, EBSD, TEM, HR-(S)TEM,EDX, EELS, in situ, P-NBED, 3D, Holography

DIGITAL python

Content

- Images
- Digital Images
- Visualisation
- Signals
- Geometry
- Filtering
- Segmentation
- Enhancement
- Alignment
- Summary

Images and Photographs

krvarshney.github.io

Images

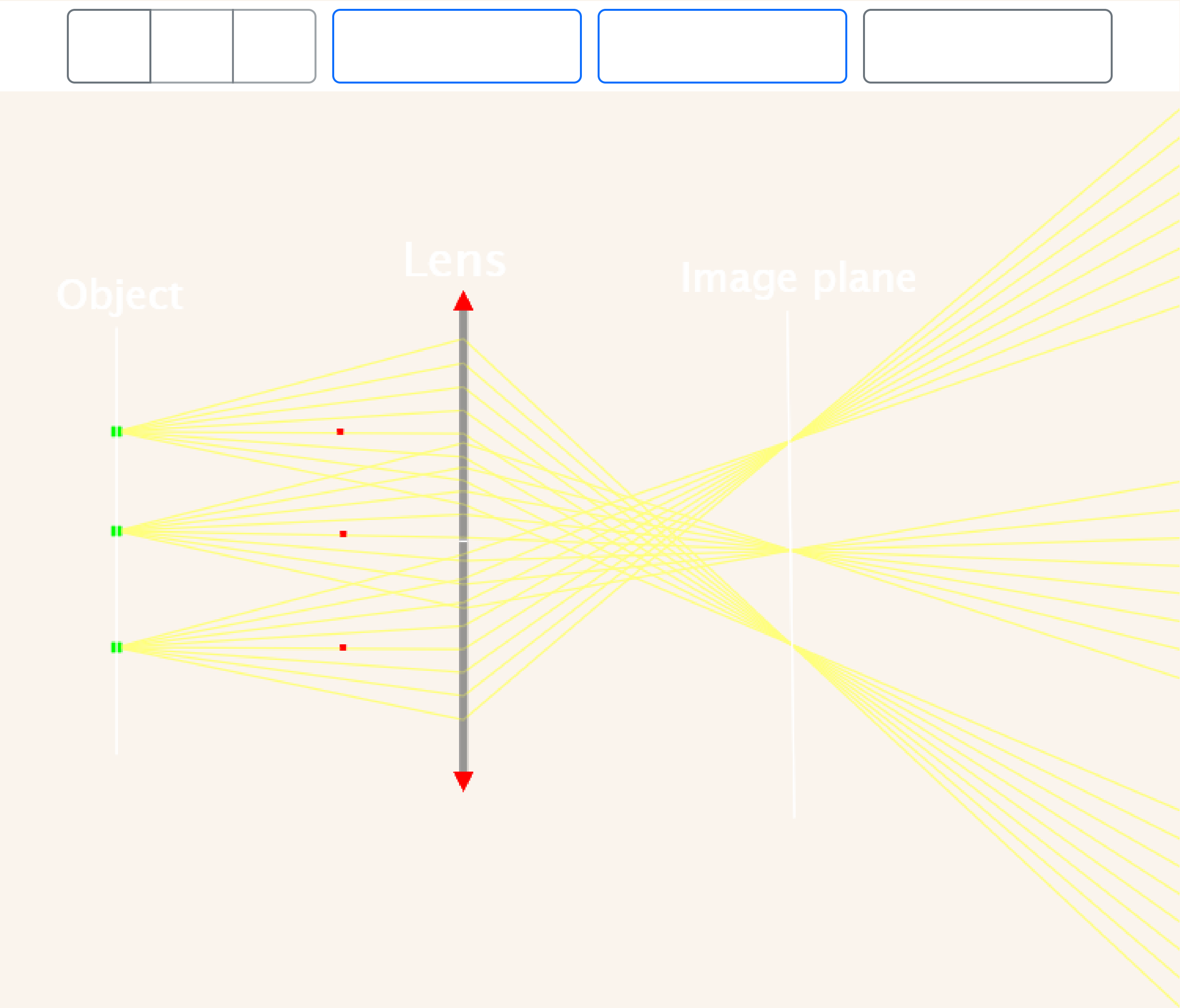
In optics, an Image is a plane with a **one-to-one** mapping between ray origin points and ray destinations

> All rays leaving one point on an object arrive at the same point in the image

Recording the rays on this plane will give a **spatially correct representation** of the object

> Of course, this becomes *much* more complex when the optical system is imperfect, or an object has depth!

phydemo.app

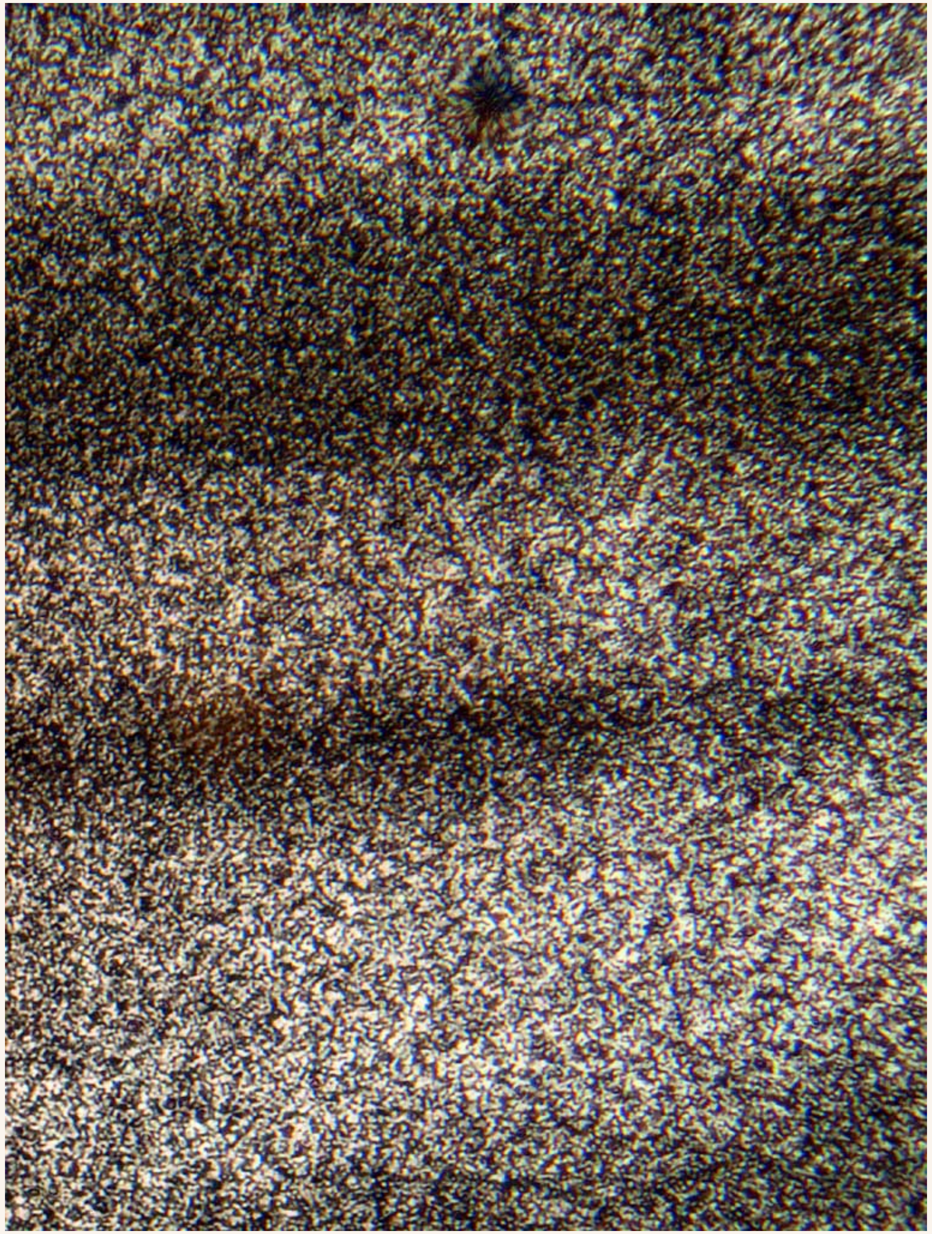


Analogue images - Film

Recording light in chemical reactions

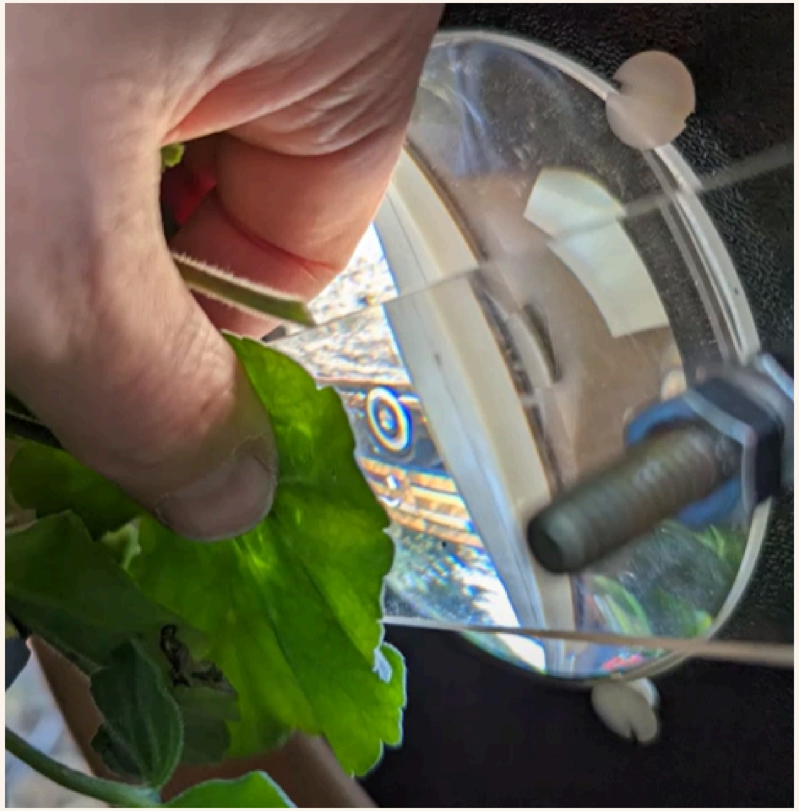
- Light- (or electron-) sensitive coatings that transform when **exposed**
- Sensitivity determined by (chemical) reaction rate (temperature, wavelength etc.)
- **Resolution** determined by average particle size - randomly distributed!
    - In practice film is extremely densely coated

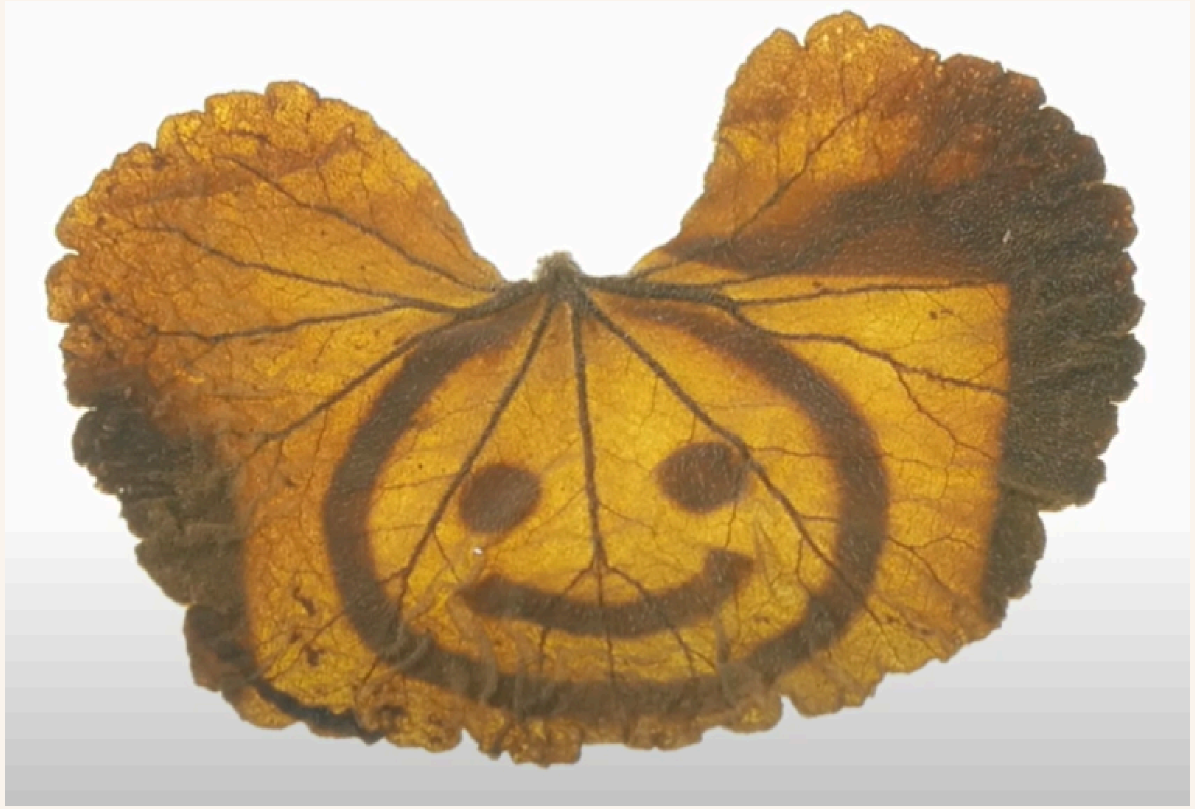
Film grain



Analogue images with plants

Any light-reacting chemistry could be used to record a photograph, even photosynthesis!



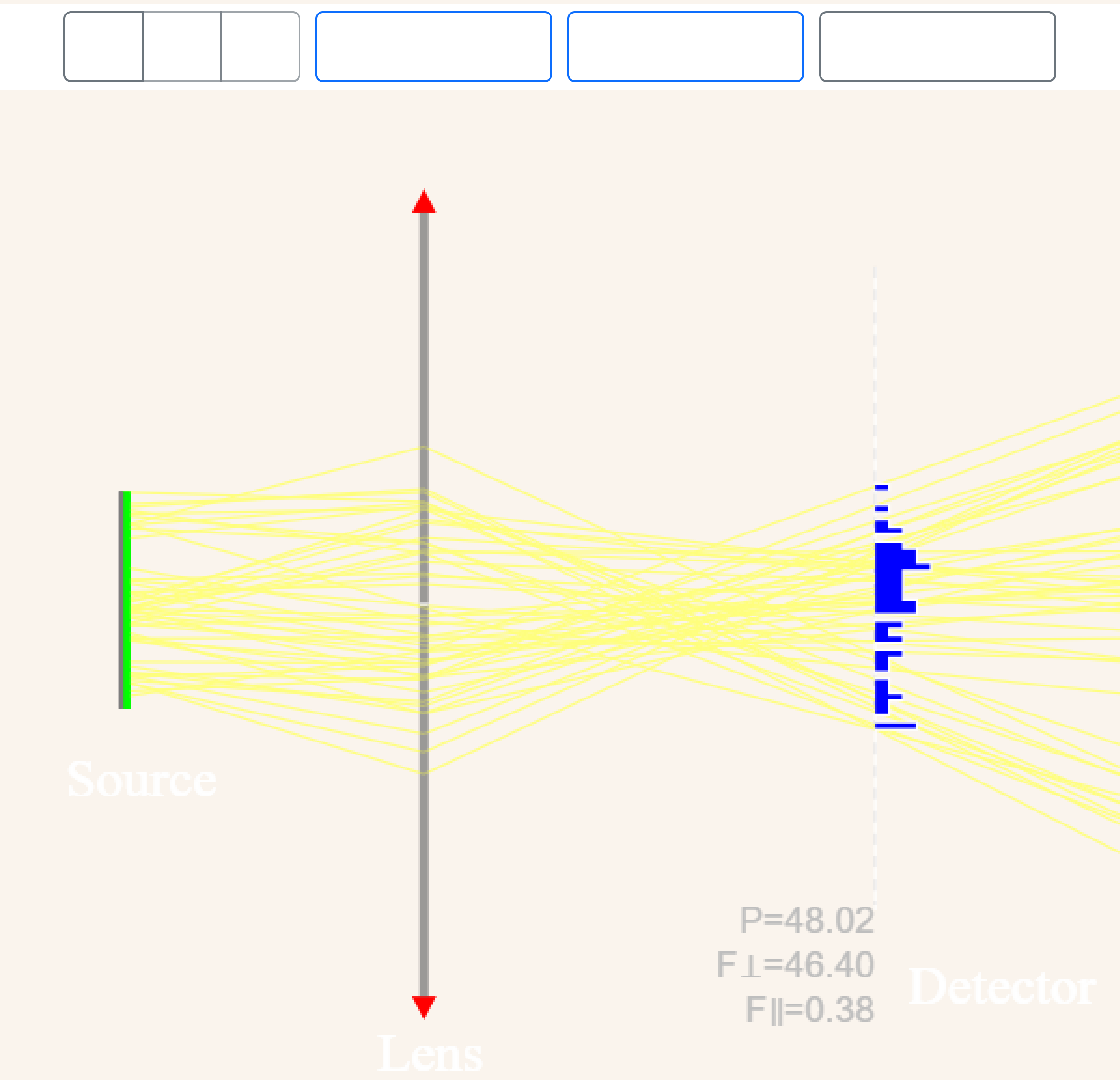


Digital images

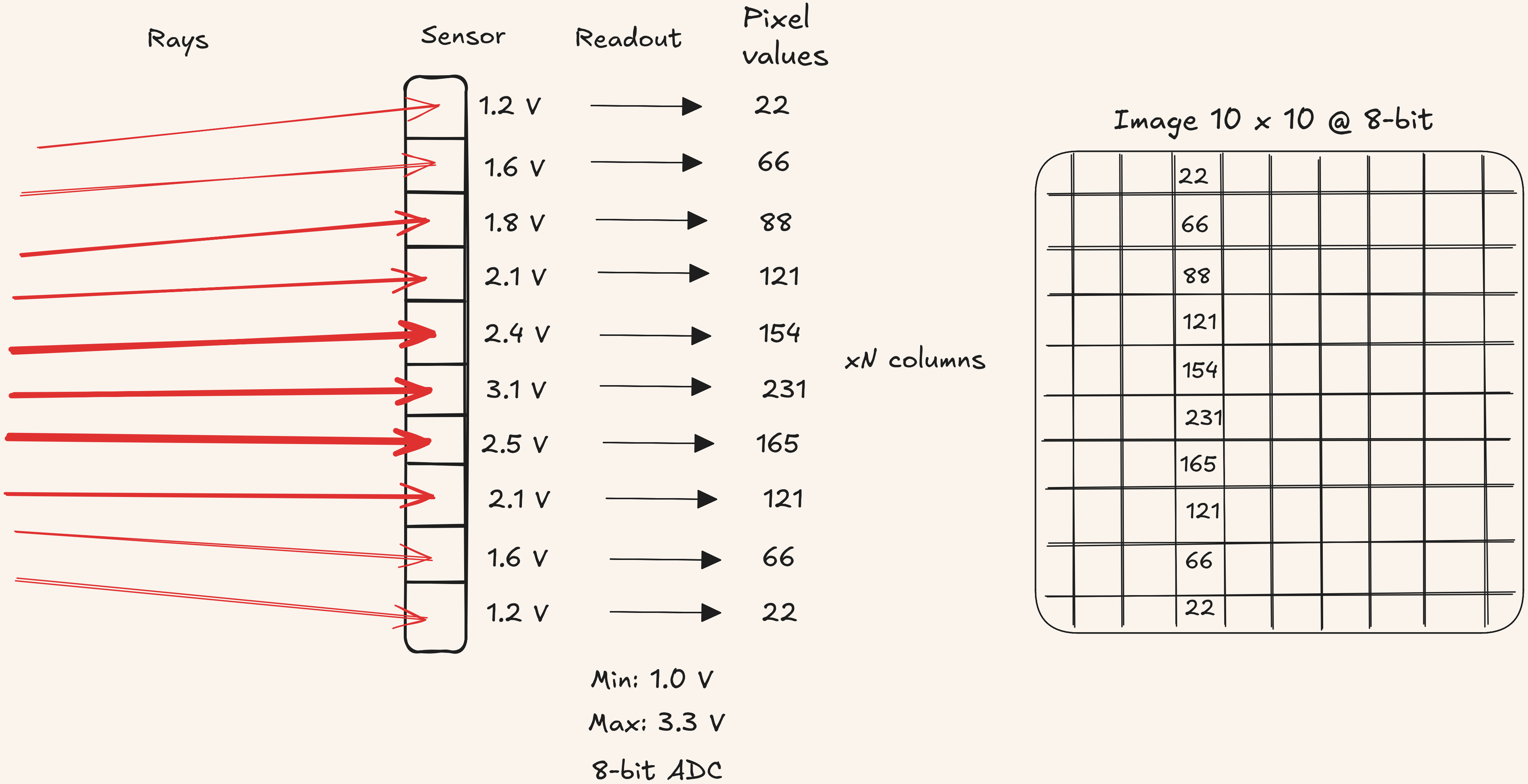
Recording images with numbers

- Convert local light intensity to an electrical signal, then **digitize** it
- Sensors have physical limits, noise etc, so our digitization is always imperfect

At the most basic, a digital image is a list of numbers representing recorded *values*, and a way to structure these numbers into a shape we can interpret as the physical image



Rays to Digital Image



Rays

Sensor	Readout	Pixel values
1.2 V	→	22
1.6 V	→	66
1.8 V	→	88
2.1 V	→	121
2.4 V	→	154
3.1 V	→	231
2.5 V	→	165
2.1 V	→	121
1.6 V	→	66
1.2 V	→	22

Min: 1.0 V

Max: 3.3 V

8-bit ADC

xN columns

Image 10 x 10 @ 8-bit

			22						
			66						
			88						
			121						
			154						
			231						
			165						
			121						
			66						
			22						

What are pixels? Resolution?

A *pixel* is an **el**ement of a **pi**cture. In acquisition it is the sampled value at a given position.

- Represents a single, discrete intensity from the wavefront that was recorded

You may also encounter the term *voxel*, which is an **el**ement of a **vo**lume in 3D

Resolution, depending on the context, can be pixel spatial **density** (i.e. how well we can *resolve* two adjacent peaks), or total pixel **count**, usually as a 2D shape e.g. `(height, width)`.

Python: xarray, HyperSpy

Calibrations

Digital images are discrete, both in space and value

- Position within a digital image is given by an integer coordinate:
    - `[3, 5]` not dimension `[0.2 cm, 0.8 cm]`
- Intensity is usually recorded as an integer value
    - `530` not a physical quantity like $3.2\ J$

Interpretation of digital images in physical units requires a calibration, accounting for (amongst others):

- Pixel size, spacing, shape
- Sensor response, readout characteristics

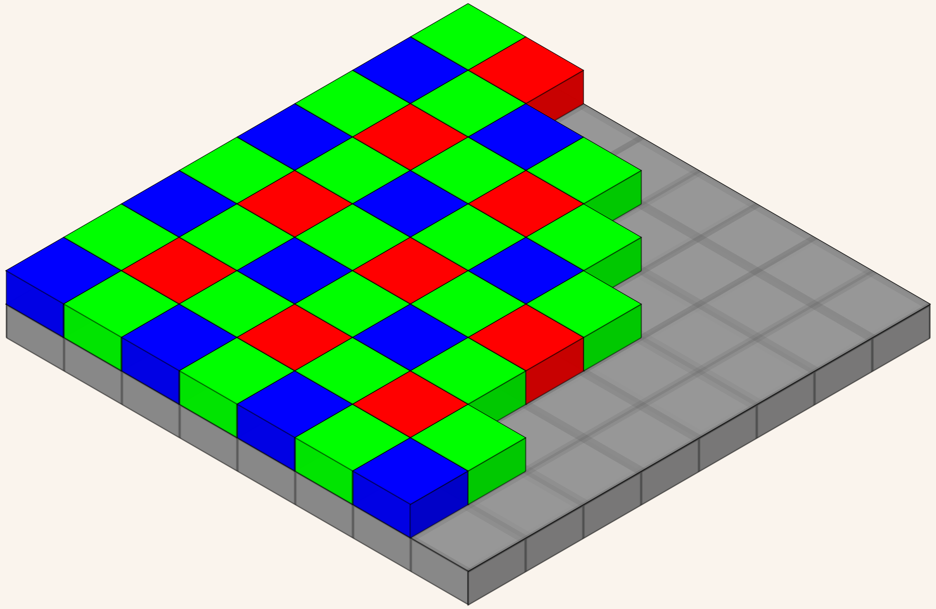
These values *may* be found in the image metadata, if you're lucky!

Colour images

A colour image is a stack of images of the same wavefront, each sampling one part of the spectrum

- We are most familiar with RedGreenBlue (RGB) images
- These are usually made with a pre-sensor **Bayer filter**, which samples colour differently in adjacent pixels
- The recorded values are split into separate R, G, and B intensity images

The three **channels** in the `[heigh, width, colour]` stack are spatially offset, but with intelligent recombination we can display them without artefacts



Bayer-filter, Wiki - Cburnett

Almeida et al (2022)

Spectral images

Spectral images are a generalisation of colour images, where each channel represents a well-defined band of energy.

- Ideally spectral channels don't overlap in energy, unlike many colour image filters
- We normally can't sample both spatially and spectrally simultaneously, so we create images *channel-by-channel* (e.g. EFTEM), or *position-by-position* (e.g STEM-EELS)

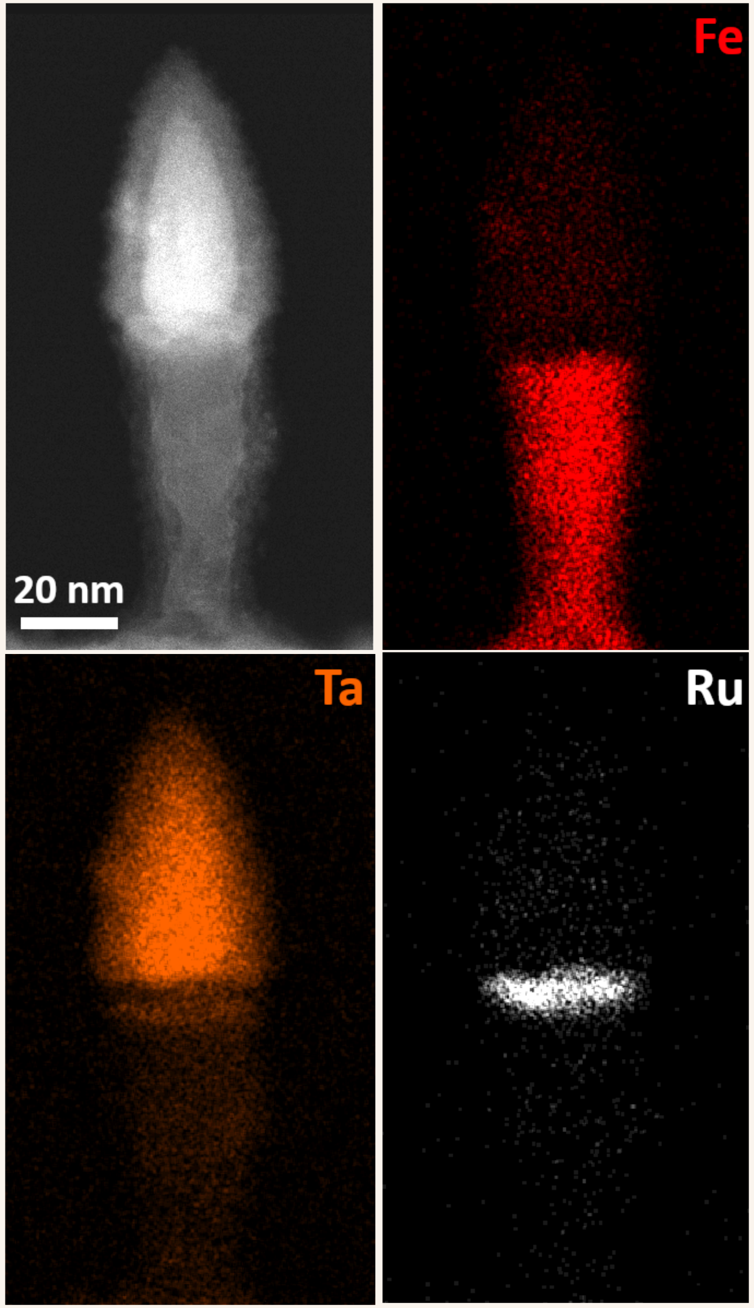
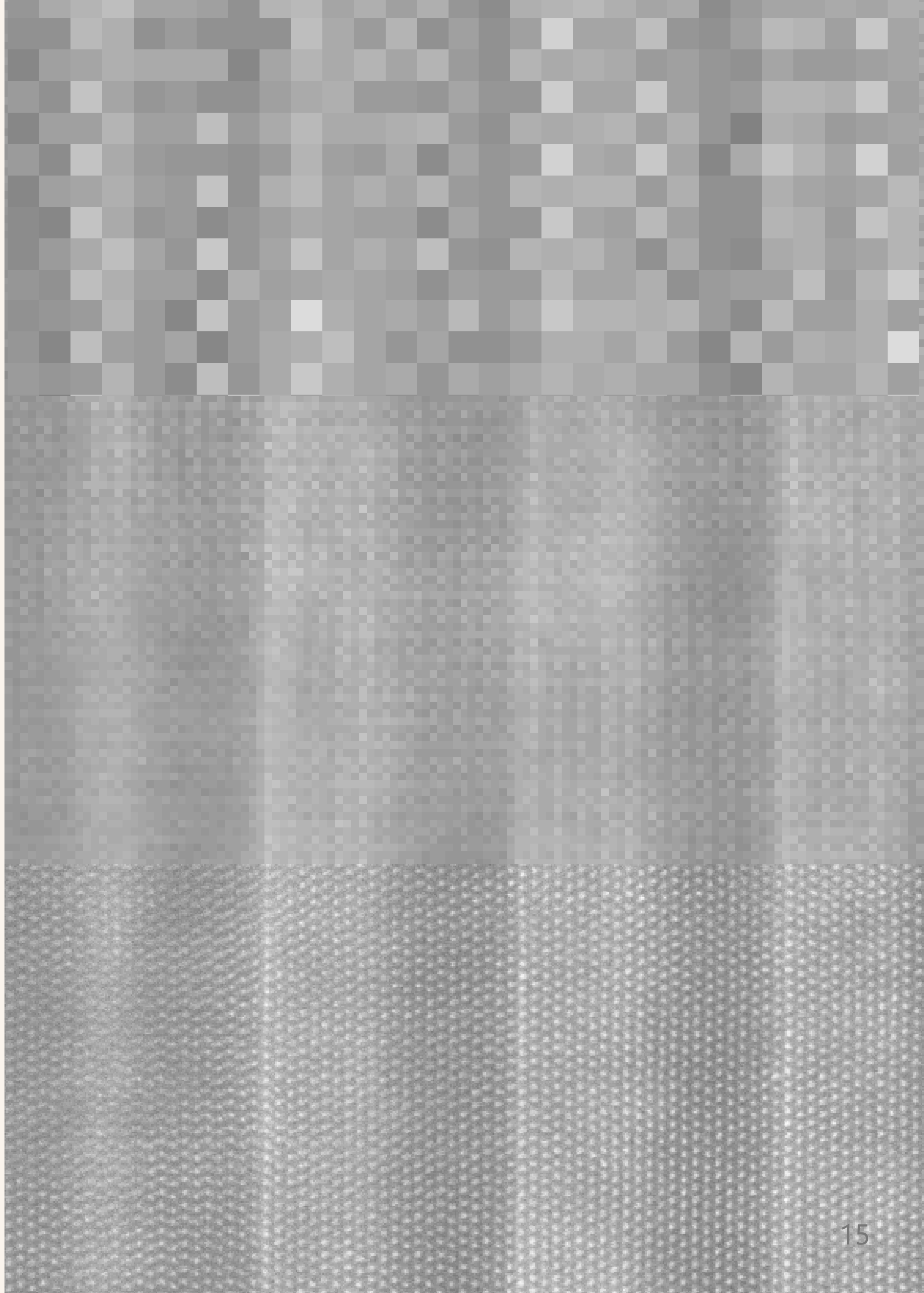


Image: Jean-Luc Rouvière

Images and Computers



Arrays of numbers

Computers store numbers long sequences of binary digits (`0`, `1`), which we can interpret to reproduce an image with a given shape

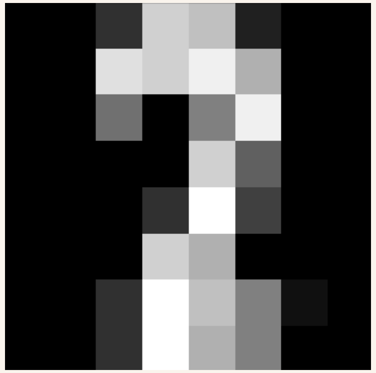
> Images are 1-dimensional **sequences** of numbers to a computer, there is no hardware-level concept of `height`, `width`, `channel` etc.

Numbers can also be stored using different rules, leading to even more ways to (mis-)interpret an image.

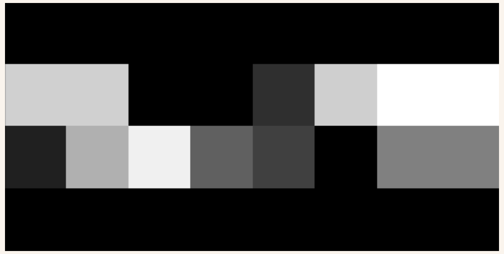
The numbers:

```
0000000000000000000000110000110100001100000000100000000000000000
0000000000000000000011100000110100001111000010110000000000000000
0000000000000000000001110000000000001000000011110000000000000000
0000000000000000000000000000000000001101000001100000000000000000
0000000000000000000000000000001100010000000001000000000000000000
0000000000000000000000000000110100001011000000000000000000000000
0000000000000000000000110001000000001100000010000000000100000000
0000000000000000000000110001000000001011000010000000000000000000
```

Can be equally interpreted as:



uint8, C-order, 8x8



uint16, F-order, 4x8

Number types

There are many conventions for storing numbers as binary, here are some common ones used in images. Usage depends on your camera electronics and what processing you do.

	Names	Size (bits or digits)	Min	Max	`0100000001001001`
					`0000111111010000`

Binary	`bool`	8	0	1	`-`
Unsigned Integer	`uint8`, `ubyte`	8	0	255	`64`, `73`, `15`, `208`
	`uint16`	16	0	65,535	`18752`, `53263`
Integer	`int16`, `short`	16	-32,768	32,767	`18752`, `-12273`
	`int32`, `long`	32	-2,147,483,648	2,147,483,647	`-804304576`
Floating (Decimal)	`float32`, `float`	32	-3.40E+38	-3.40E+38	`3.14159`
	`float64`, `double`	64	-1.70E+308	1.70E+308	`-`
Complex	`complex64`	64	-3.40E+38	-3.40E+38	`-`

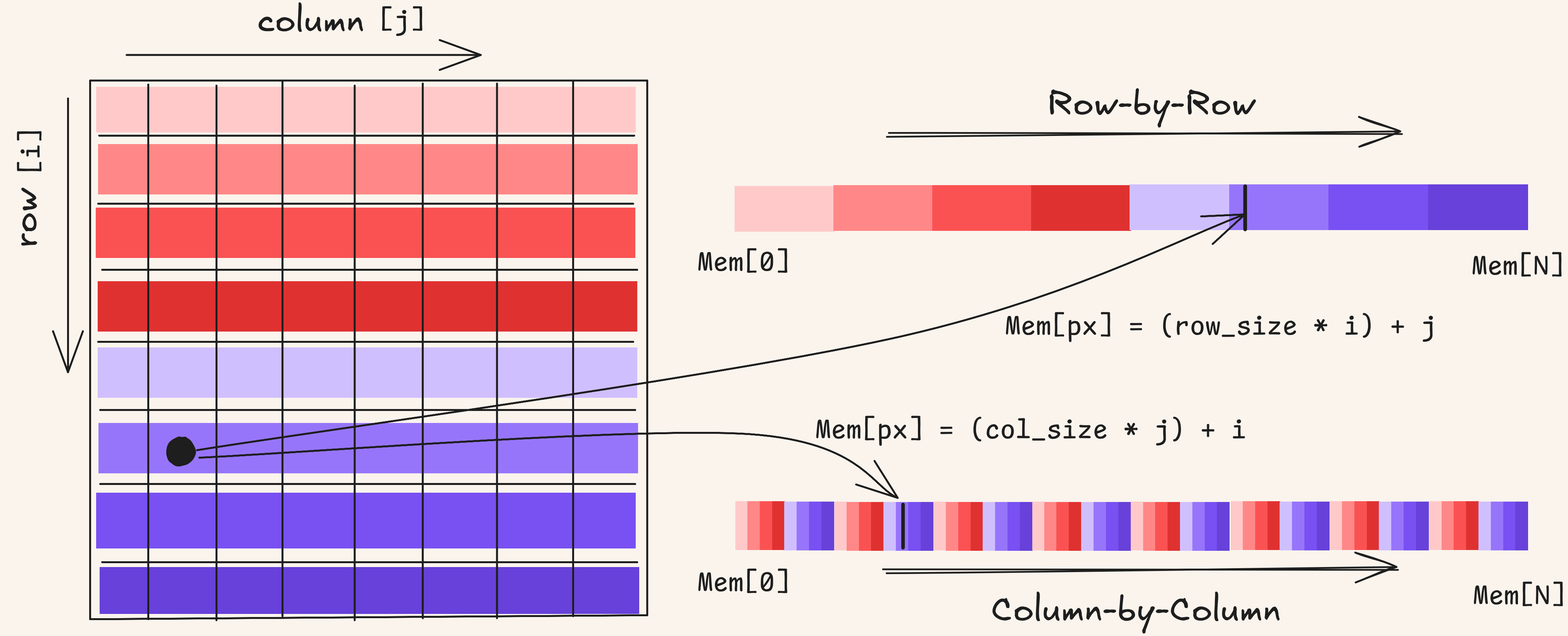
Number types - Notes

- Digital numbers are stored in a fixed amount of space - exceeding the min or max for a type can cause **wrapping**, e.g. `200 + 100 = 44`!
    - `uint8` has a maximum of `256`, so $200 + 100 = 300 \Rightarrow$ `300 mod 256 = 44`.
- The size of the number `=` the space it requires in memory and on disk
    - No reason to store 8-byte `float64` if values are only `0` or `1`
    - Often larger types ⇒ slower computation
- Floating point numbers have **variable precision**, i.e. they can represent *very large* or *very small* values, but cannot represent a large number with a small fraction:
    - `float32(324,000) + float32(0.0055) = float(324,000.0)` and not `float32(324,000.0055)`

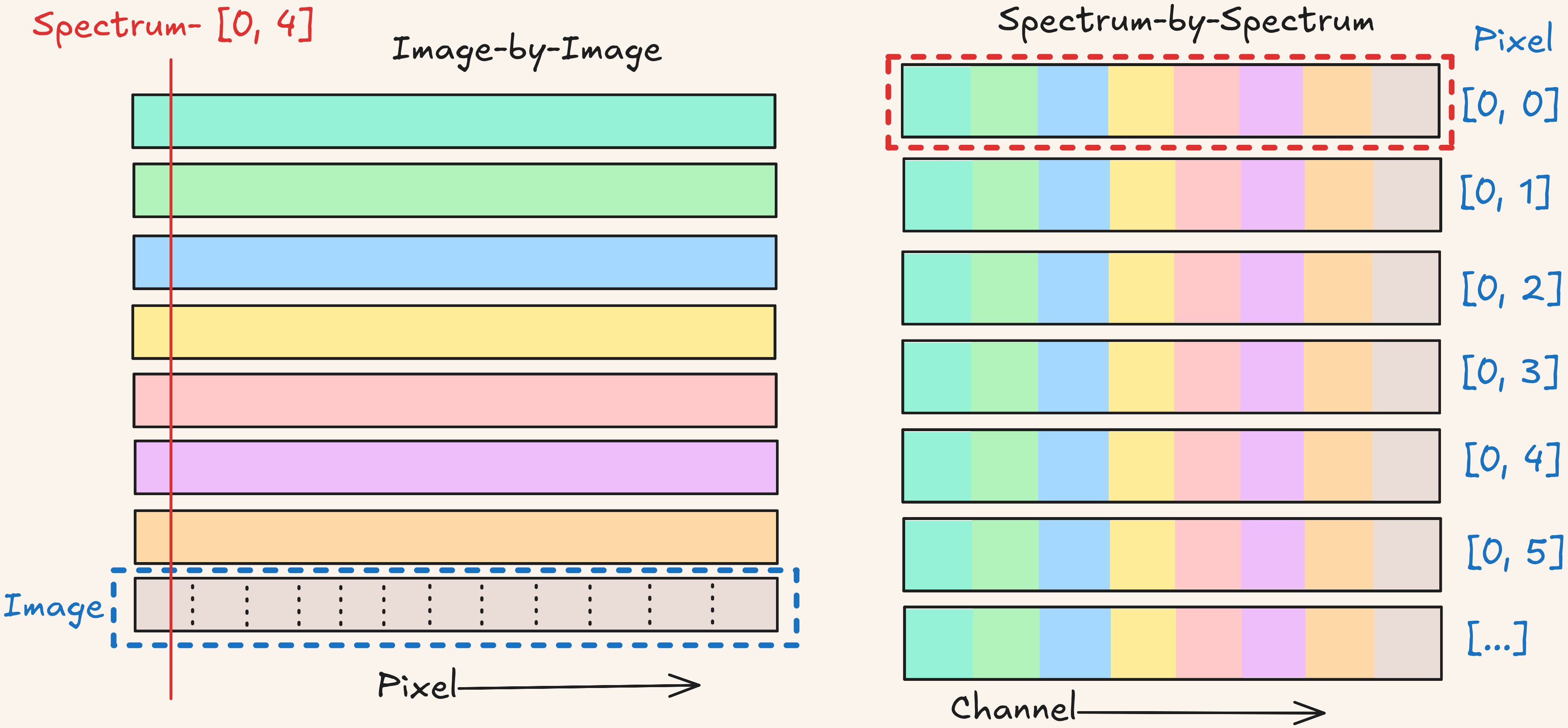
⇤ ↞

Memory layout

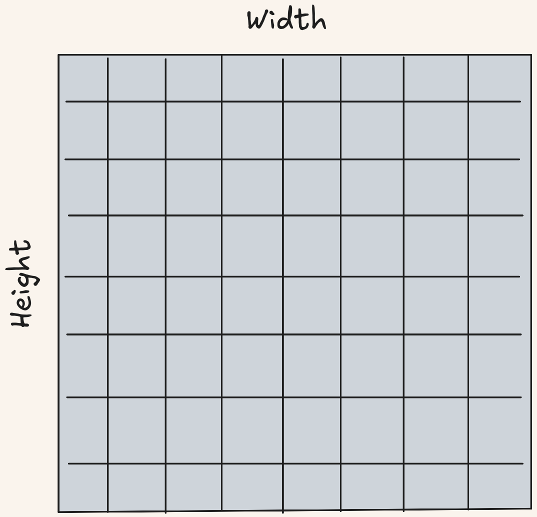
A 2D image is usually ordered **row-by-row**, or **column-by-column**, by convention. As each number occupies a fixed number of bits in the sequence, we can find the value of any pixel by *row/col arithmetic* and then **indexing** according to the layout in memory.



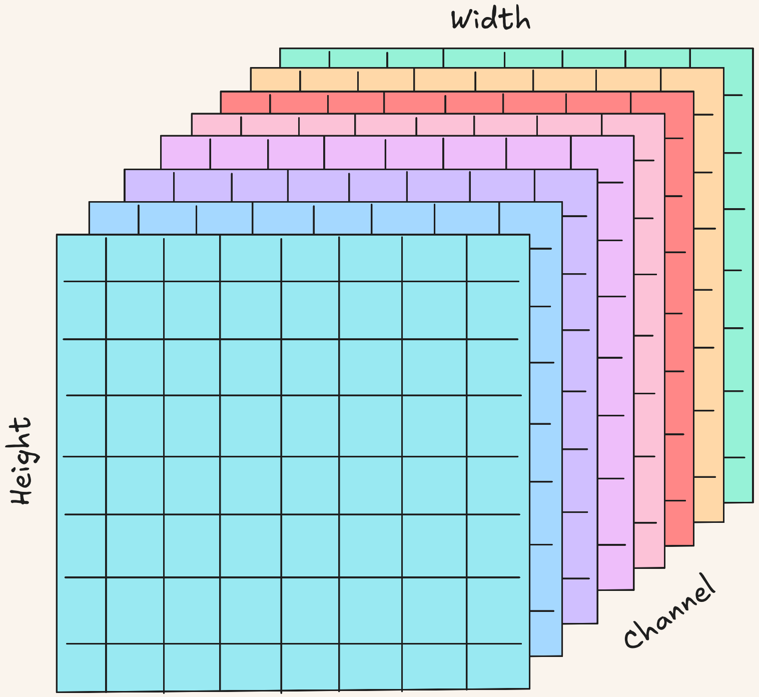
If an image is large and >2D, e.g. a spectrum image, then memory layout can heavily affect processing time. **Jumping** between memory locations is very slow compared to sequentially reading memory, so it pays to store data in the way it will be processed.



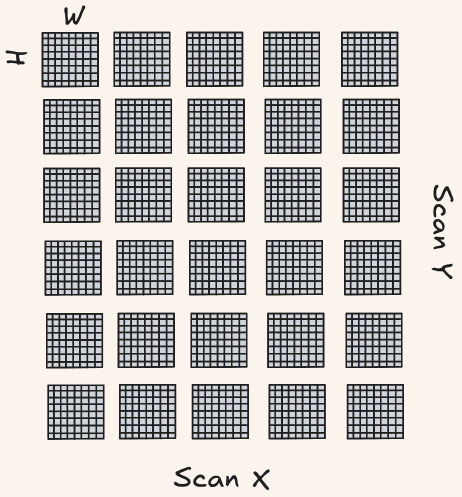
Multi-image data, stacks, 4D-STEM



Shape [1024, 1024] ≈ 2 MB



Shape [64, 1024, 1024] ≈ 128 MB



Shape [100, 100, 1024, 1024] ≈ 20 GB

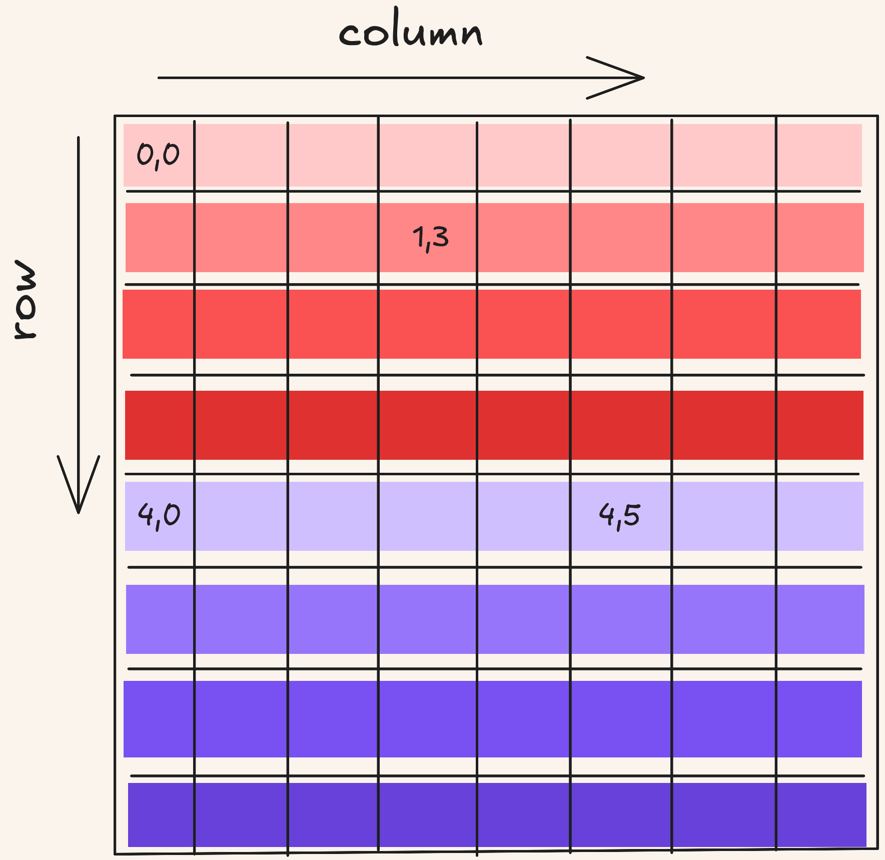
Tomography can an add an extra `[tilt]` dimension to all of the above!

Coordinate systems

Depending on the tool or programming language, image coordinate systems vary

- Matrix notation in 2D: `[row, column]`
- Python is **0-indexed**
  - `image[0, 0]` is the first pixel
- MATLAB is **1-indexed**:
  - `image[1, 1]` is the first pixel

Extra dimensions e.g. `channel`, `scan` are according to convention (and sometimes also memory-layout).



Typically `row == 0/1` at the **top** when displayed, with positive-**down**

Maths with images

As an image is just a list of numbers, so we can do arithmethic or more complex operations on images to yield new images or other results. For example:

```
image = image - image.min()  # subtract the minimum value in the image from every pixel
px_sum = image[5, 7] + image[2, 4]  # sum the values in two pixels
image = log(image)  # take the natural logarithm of every pixel
wavefront = exp(-1j * image)  # interpret the image as phase and create a complex wavefront
sum_image = image + other_image  # sum two images together
```

Note that when operating on pairs of images they must have the same `shape` for the elementwise calculation to be defined.

Maths with images

Image: Hanako Okuno / Tescan + Wikipedia

```
def image_function(img):
    return -1 * img


#  img ** 2  |  np.log(0.01 + img)  |  img * range(w)

```

RUN

Image file formats

Images can be stored in many ways, depending on how they are used

- `.jpg`, `.png`, `.gif`: colour RGB `uint8` images, compressed for small file size, open anywhere without special software, not for scientific *data*, just **visualisation**
- `.tif`: a **general-purpose** image format, can hold most number types and shapes
  - TIFF files with strange data (floating point) may need special software
  - Can hold additional metadata (e.g. calibrations), can be compressed
- Proprietary formats like `.dm3/4`, `.mib`, `.emd`, `.blo`: specific to a certain camera or software, not always readable elsewhere
- General *array* formats: `.mat`, `.npy`, `.hdf5`, `.zarr`: flexible, can be compressed, can hold stacks / nD data and metadata, need compatible code/software

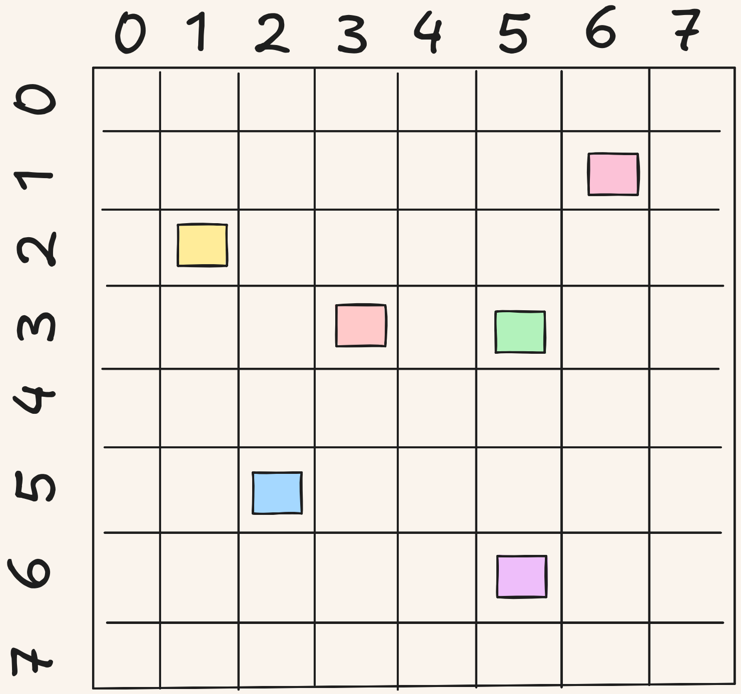
Python: `sparse`, `scipy.sparse`

Sparse images

In very low dose conditions (e.g. EDX), most image pixels contain a **zero value**. This is good use case for *sparse* images.

> Storing only the **non-zero values** can achieve enormous space savings

- Simplest strategy is store non-zero values and their coordinates, but more intelligent schemes exist
- Many operations are $f(a, 0) \in \{0, a\}$ so also avoid wasted computation



Full image with zeros ≈ 128 bytes

6 values + (x,y) coordinates ≈ 24 bytes

Stored as: [1, 6, 🟥, 2, 1, 🟨, 3, 3, 🟧,
3, 5, 🟩, 5, 2, 🟦, 6, 5, 🟪]

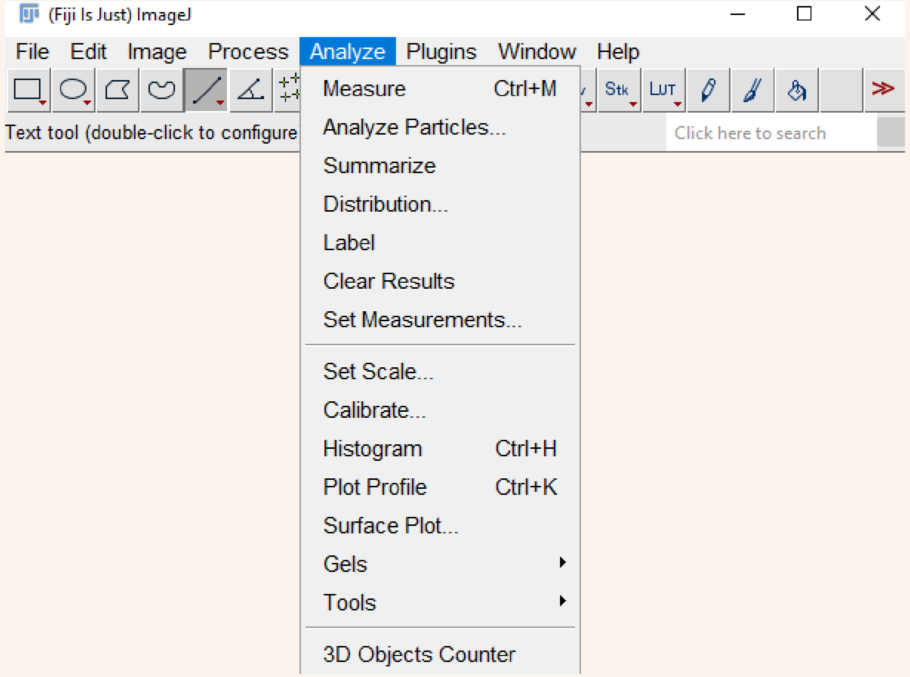
Image software

Useful software packages to work with images in microscopy

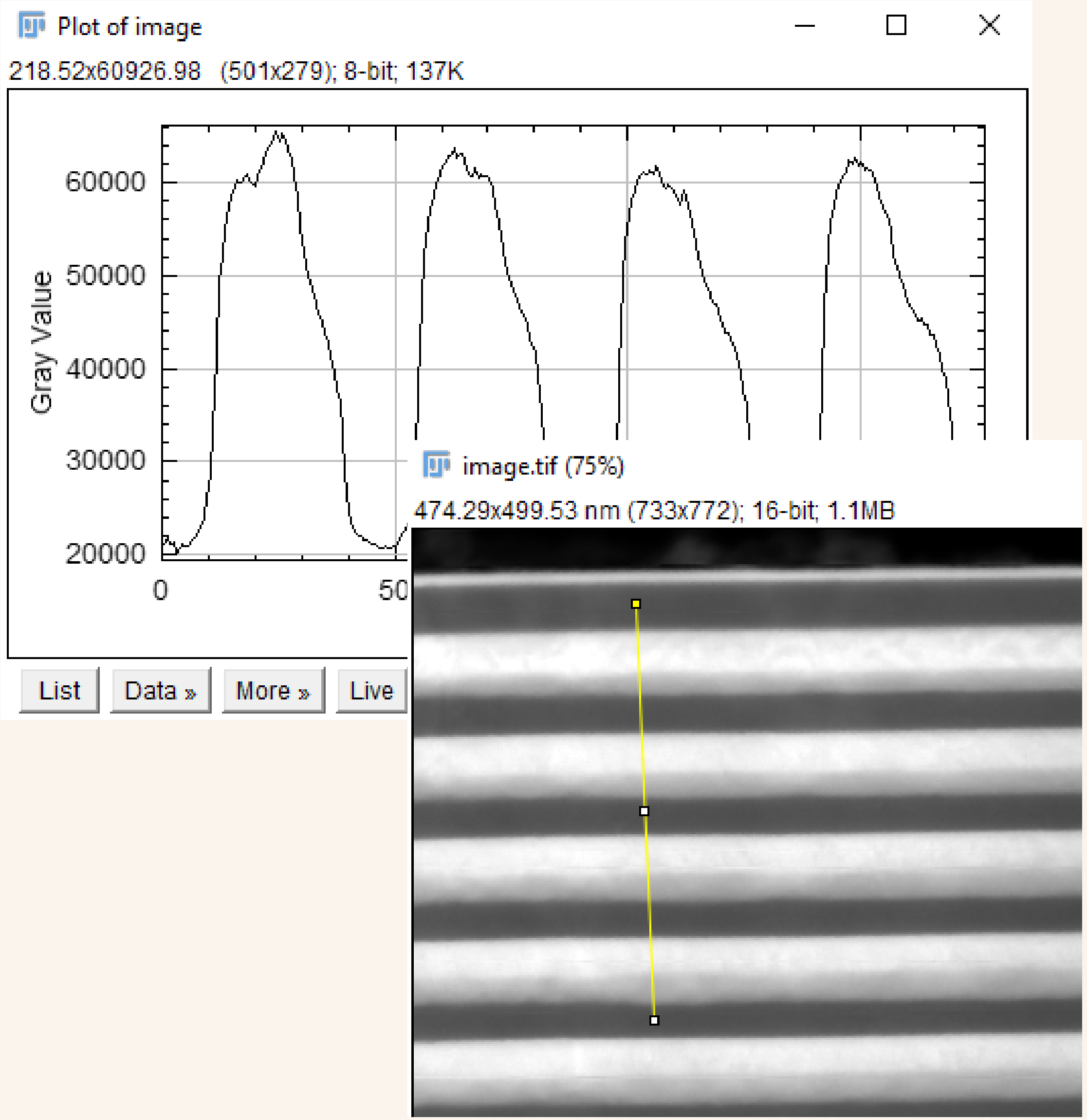
Image: Lou Denaix

Fiji (imagej.net)

Widely used in scientific imaging, plugins…



Text tool (double-click to configure

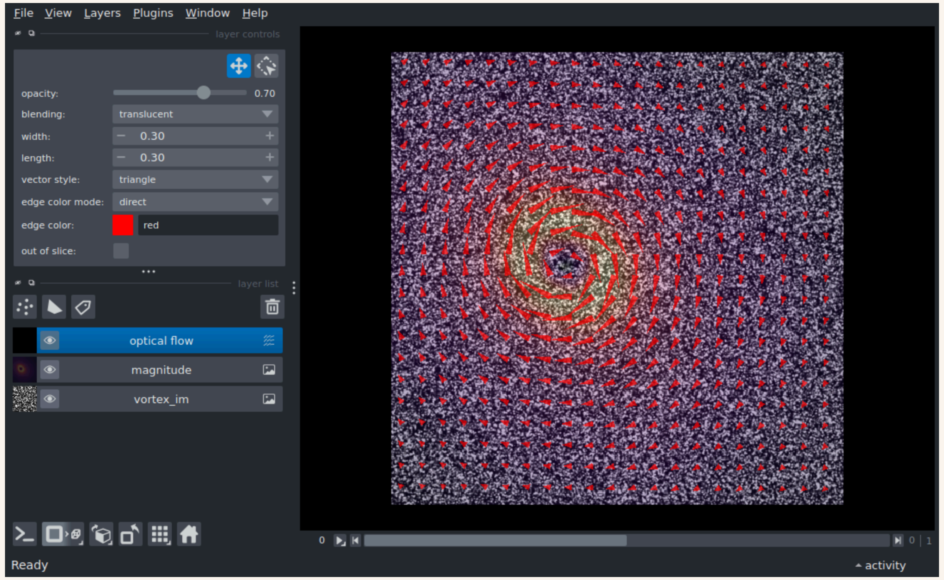


Calibrations, stacks, measurements, math, segmentation…

Napari (napari.org)

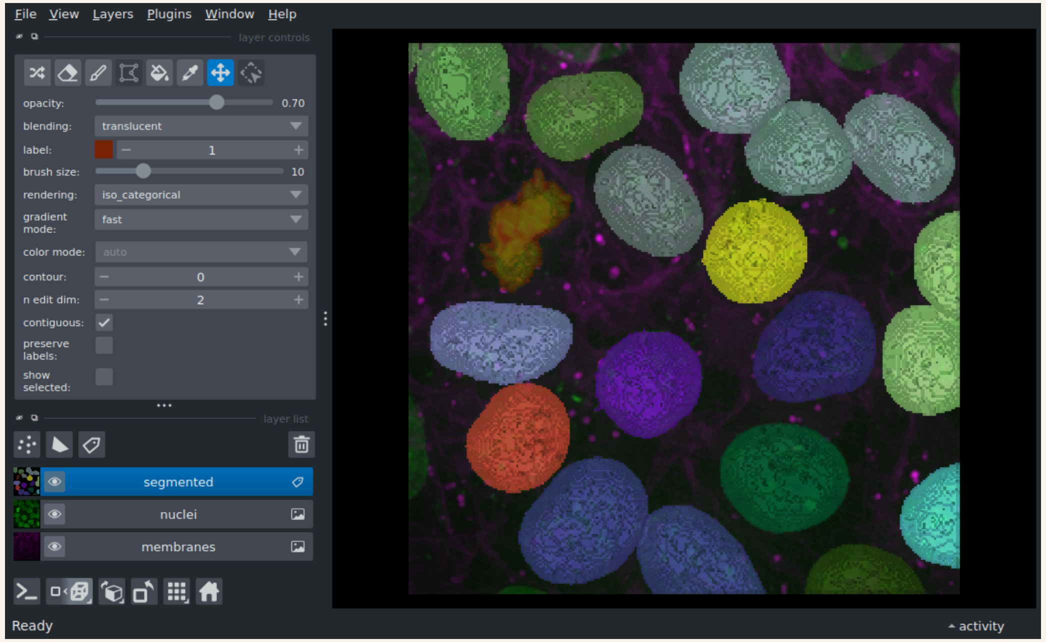
Multi-D data viewer, annotations

Python-based, easy to add analysis



opacity: 0.70
blending: translucent
width: 0.30
length: 0.30
vector style: triangle
edge color mode: direct
edge color: red
out of slice: □

optical flow
magnitude
vortex_im



opacity: 0.70
blending: translucent
label: 1
brush size: 10
rendering: iso_categorical
gradient mode: fast
color mode: auto
contour: 0
n edit dim: 2
contiguous: ☑
preserve labels: □
show selected: □

segmented
nuclei
membranes

Good support for 3D volumes

Gatan Digital Micrograph (gatan.com)

Well-known, feature-rich GUI even when using the free license

Python scripting enables any analysis with GMS display

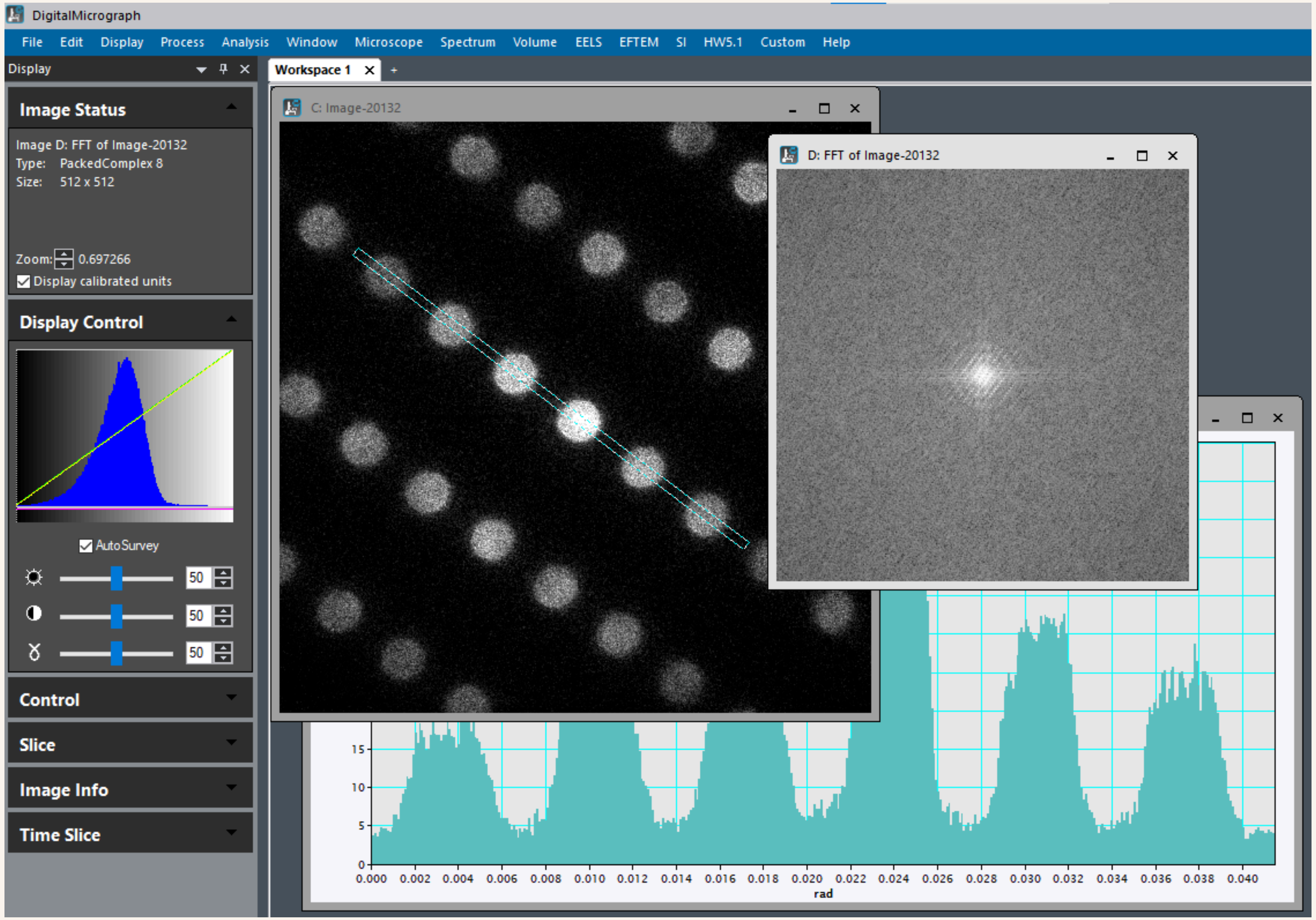


Image D: FFT of Image-20132
Type: PackedComplex 8
Size: 512 x 512

Zoom: 0.697266
☑ Display calibrated units

☑ AutoSurvey

Python libraries for images

The Python scientific ecosystem is vast - once an image is loaded as array data, typically under `numpy`, it can be interpreted in many ways.

`numpy` is the general **array** manipulation library for Python. It provides:

- The data structure for multi-dimensional arrays, including images
- Fast implementations of basic operations on these arrays

```
random_image = = np.random.uniform(size=(64, 64))  # random image 0..1 of shape [64, 64]
theta = np.arctan(random_image)  # array of radian values computed from image
phase_image = np.exp(1j * theta)  # phase image from theta values
```

`scipy-ndimage` (docs.scipy.org)

- Low-level tools for images (e.g. convolve, interpolate, measurements)

`scikit-image` / `skimage` (scikit-image.org)

- High-level tools for images (e.g. resizing, alignment, segmentation, filtering)

Edge detection

Image segmentation

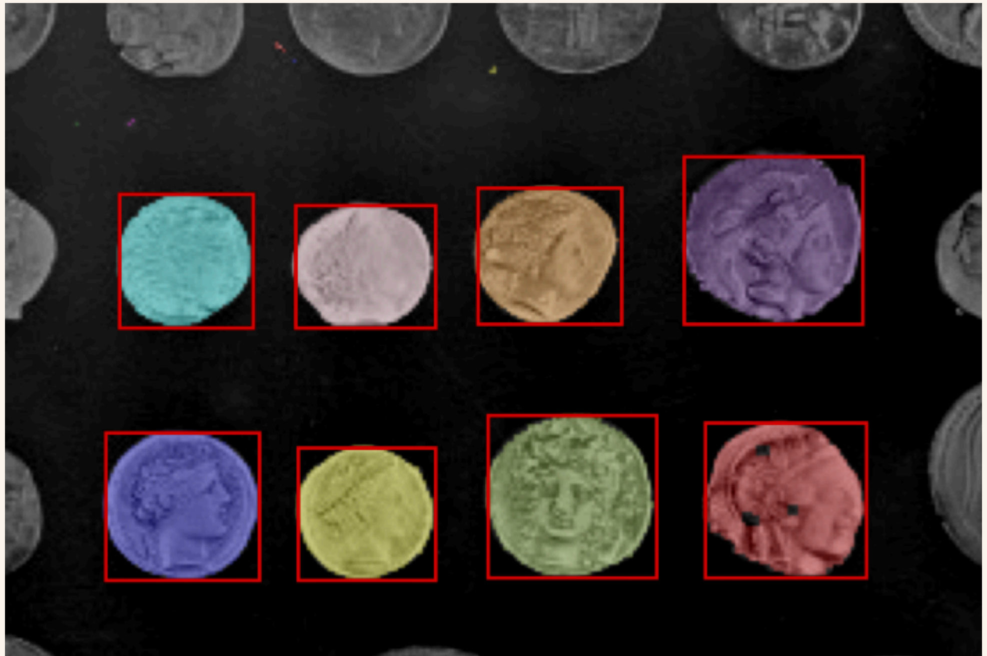


Image: Gustav Persson

`matplotlib` (matplotlib.org)

- General plotting library
- Can directly `imread` + `imshow` images
- Good for combining images with results + annotations

```
plt.imshow(image, cmap="gray")  # plot an image with gray colourmap
plt.scatter([32, 43], [16, 25])  # annotate with some points
plt.show()
```

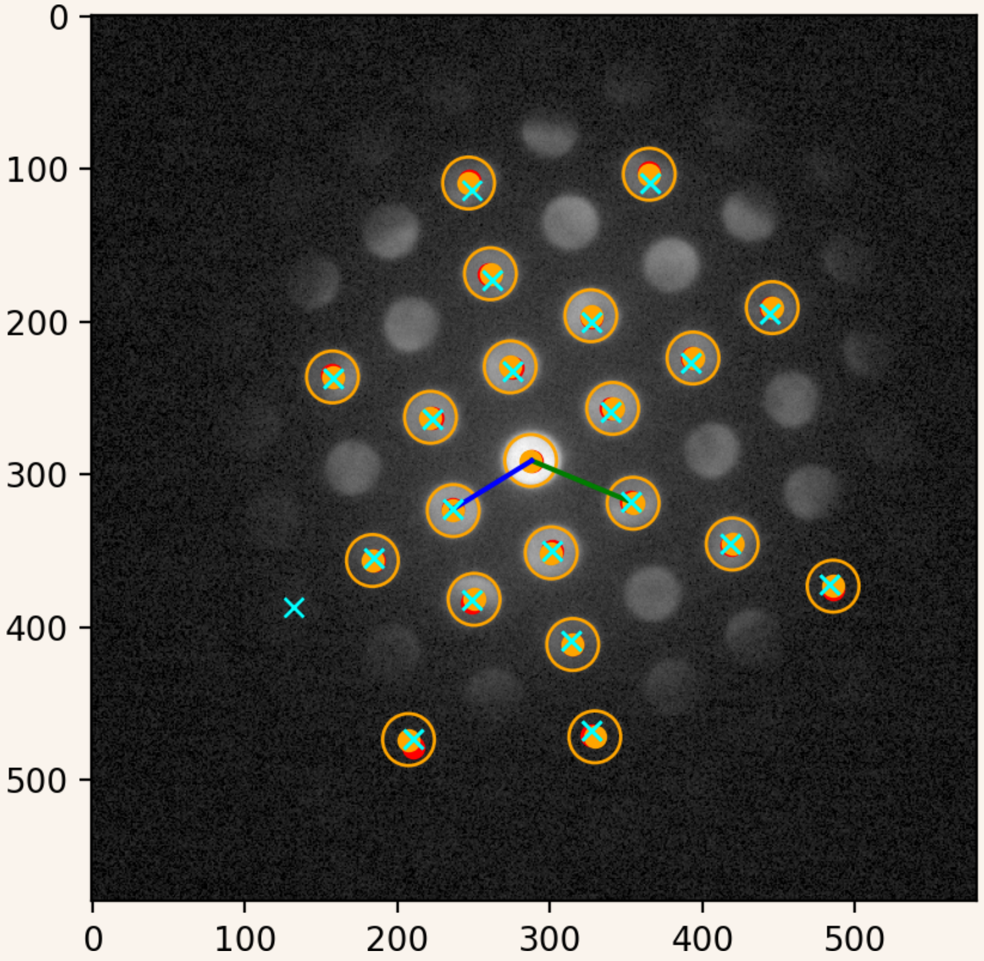
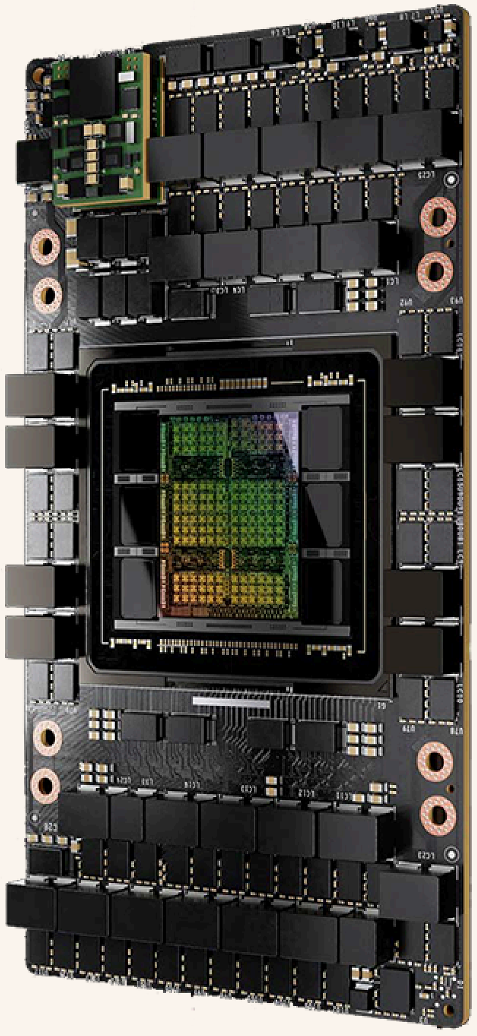


Image: nVidia

Graphics Processing Units (GPUs)

A Graphics Processing Unit (GPU) is a computation **accelerator** which can be added to computers. They can be used to speed up many forms of scientific computation.

- GPUs are specialised to perform simple math **operations in parallel** on multi-dimensional arrays of data (such as images)
- Operations for 3D graphics (coordinate transformations, filtering, raytracing), the original usage for GPUs, are frequently identical to math needed in scientific computing (FFTs, convolutions, matrix algebra and inversion).



GPU code in Python

At least in Python, it is **reasonably trivial** to make code run on GPU rather than CPU, thanks to the standardisation efforts behind the teams behind numpy and cupy.

> Many math functions have been re-implemented on GPU, and are used nearly the same as the CPU equivalent

For example the following using CPU:

```python
assert large_image.shape == (4096, 4096)

import numpy as np
img_fft = np.fft.fft2(large_image)
```

runs in `~1 s`, but is equivalent to:

```python
import cupy as cp
img_fft = cp.fft.fft2(large_image)
```

and runs in `~1 ms` on a large GPU.

Visualising images

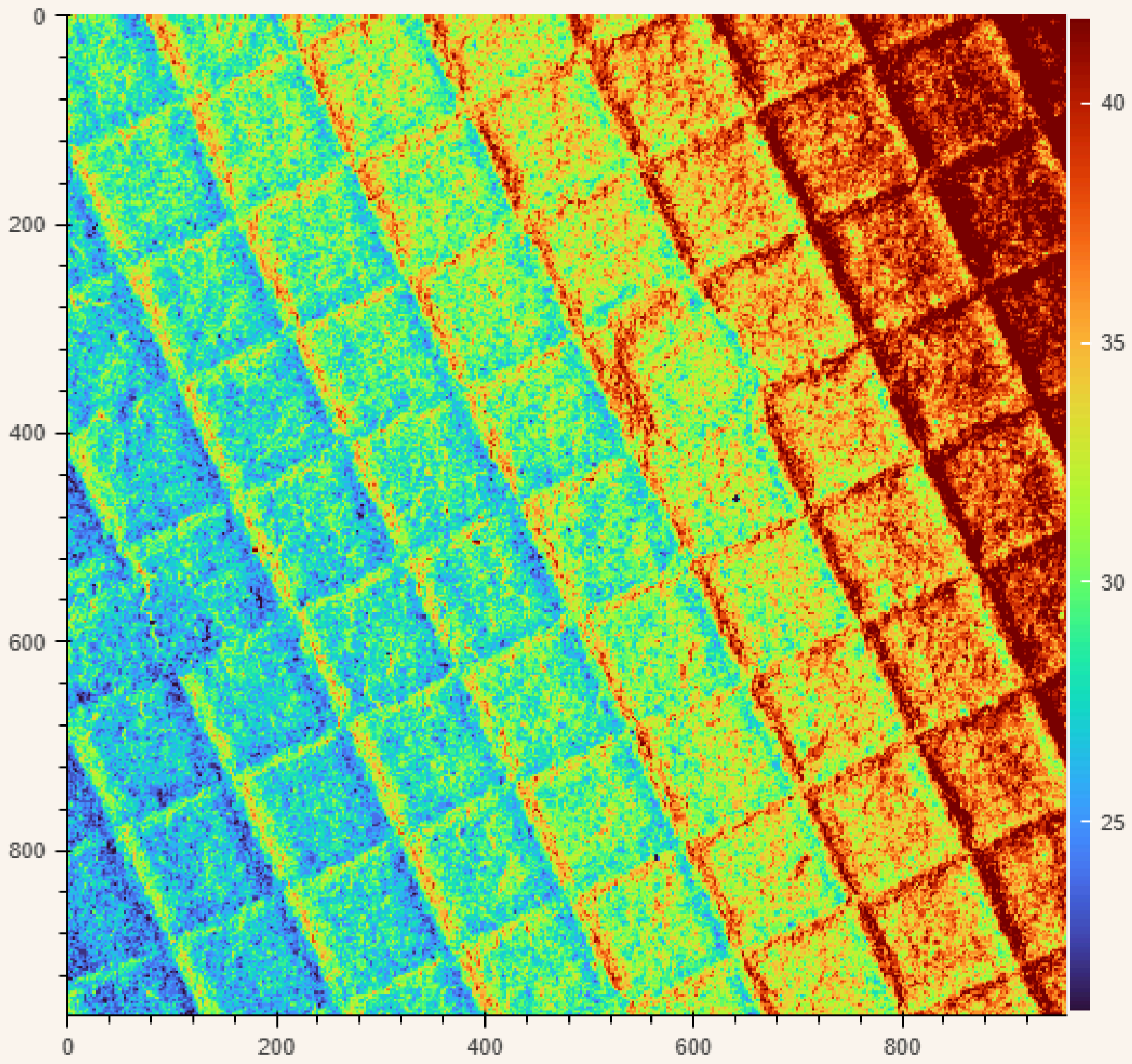


Image histograms

An image histogram represents the **frequency of intensity** values in an image. It is a useful way to visualise the separation between background and content, and to see outlier pixels.

1	2	2	4	3	1	2	7
4	5	3	6	4	5	1	4
3	2	7	2	5	2	6	2
5	1	4	3	6	7	1	7
3	7	3	6	2	1	4	2
6	4	5	1	3	5	1	5
2	5	4	5	2	4	3	4

9x - 1

11x - 2

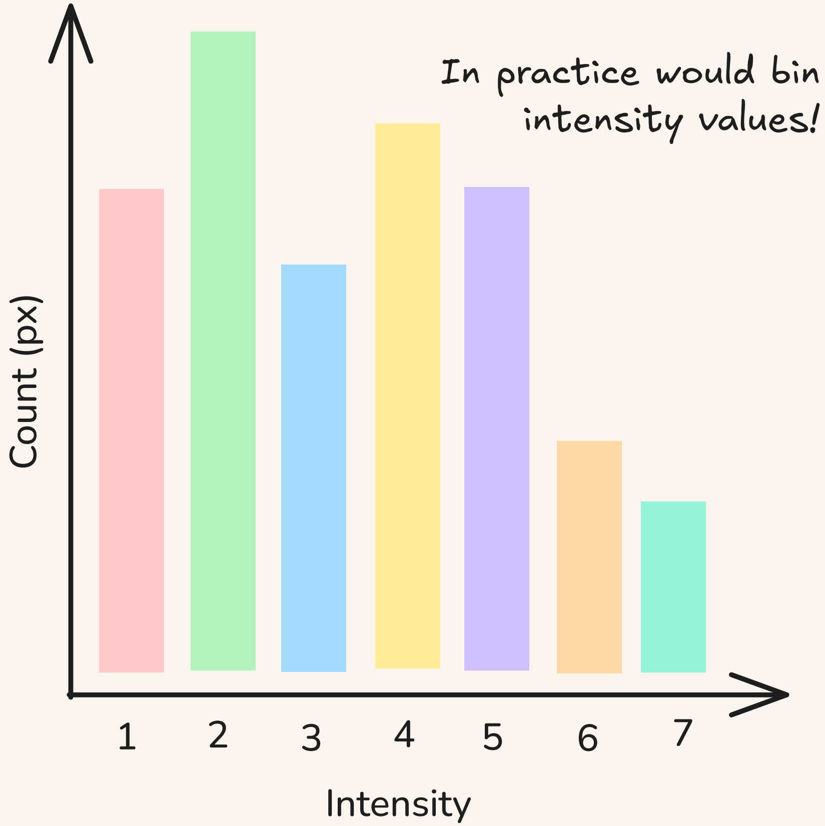
8x - 3

10x - 4

9x - 5

5x - 6

4x - 7



In practice would bin intensity values!

Data → Screen

Screens usually display `uint8` RGB colour (3 values of `0-255` per-pixel, also known as 24-bit colour). Unless our image was acquired in these three channels then we need to **transform** our data from recorded intensities to screen RGB.

- If we recorded intensity, then setting R = G = B on a screen gives colourless **Gray**
  - This limits us to **only 256 levels** of intensity to display all of the data range
  - If our data are more than 8-bit, need to sacrifice detail or clip values
- We can use artificial colour to achieve more on-screen contrast, known as a *lookup table* or **colormap**, of which there are many choices for different applications.

Choice of data transformation or colormap can massively influence how data are perceived.

Transformation: Brightness + Contrast

Image: Jean-Luc Rouvière

The basic data-to-screen transformation is linear: `[img.min(), img.max()] → [0, 255]`.

The **brightness/constrast** transform chooses two other values and **clips** pixels outside their range to `0` or `255`. This removes detail in some regions while increasing it in others.

Brightness: 0.50

Contrast: 0.75

Reset

Dynamic range

Dynamic range usually refers to the difference between the minimum and maximum value that an image could possibly represent, i.e. how much **depth of intensity** we can store without saturating at the top-end, or recording only zeros at the bottom.

> In microscopy we often have data which span orders of magnitude in intensity (e.g. diffraction patterns).

When brightness/contrast cannot cover the dynamic range of an image, a **non-linear mapping** between data and colour can be used, trading local for global contrast:

- Log-colour rescales data by its magnitude → $I_{disp} = \log(I_{img})$
- Gamma-colour scales the data with a power law → $I_{disp} = I_{img}^{\gamma}$

Image: Gustav Persson

Transformation: Gamma + Logarithmic

Linear Gamma Log

Reset

Gamma: 1

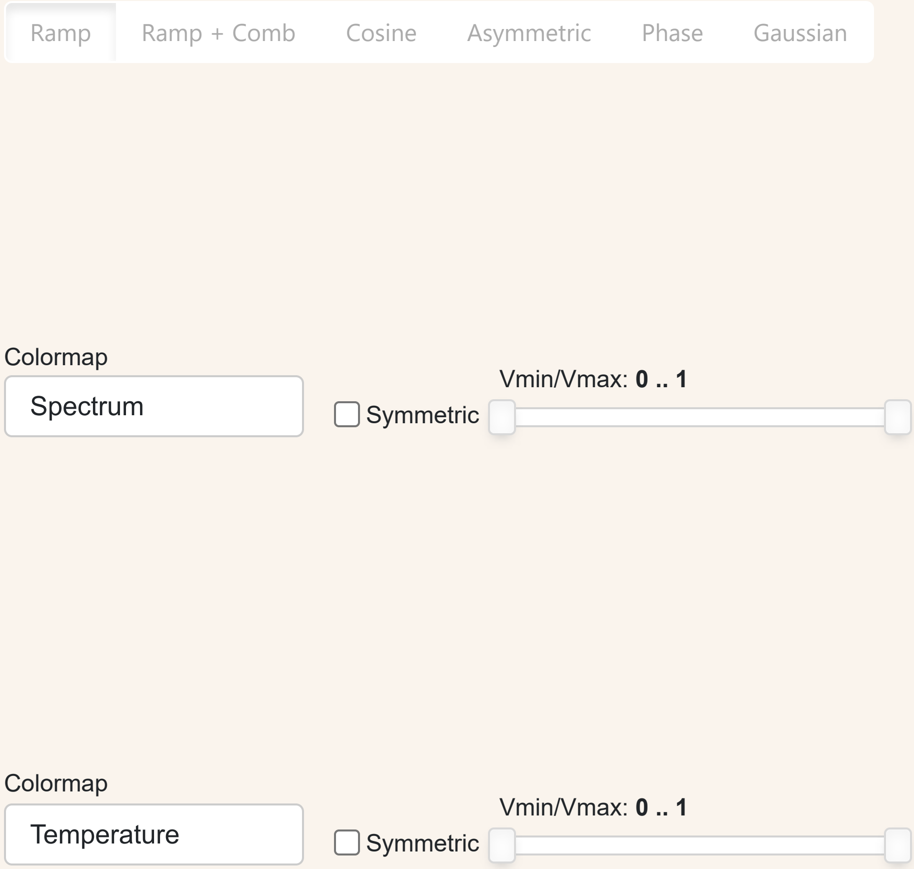
Kovesi (2015), Python: `colorcet`

Colourmaps

Colourmaps are critical to how we interpret visual data. It is important that features we see are from the data and not the map.

> Some colourmaps are made to be **perceptually uniform** - a Δ in the data is perceived as the same *visual* Δ to our eyes, across the whole range of the colourmap.

Non-uniform colourmaps can create visual features which do not exist, or hide real information.



Colormap: Spectrum

□ Symmetric

Vmin/Vmax: **0 .. 1**

Colormap: Temperature

□ Symmetric

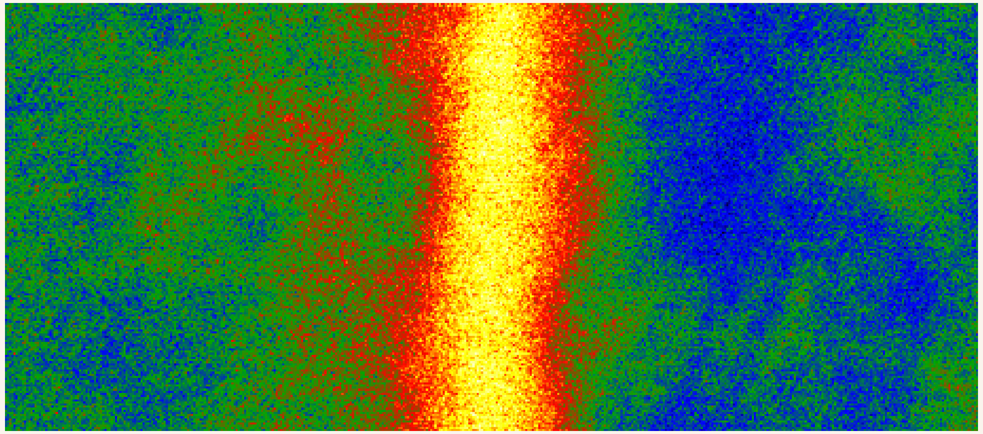
Vmin/Vmax: **0 .. 1**

davidmathlogic.com/colorblind

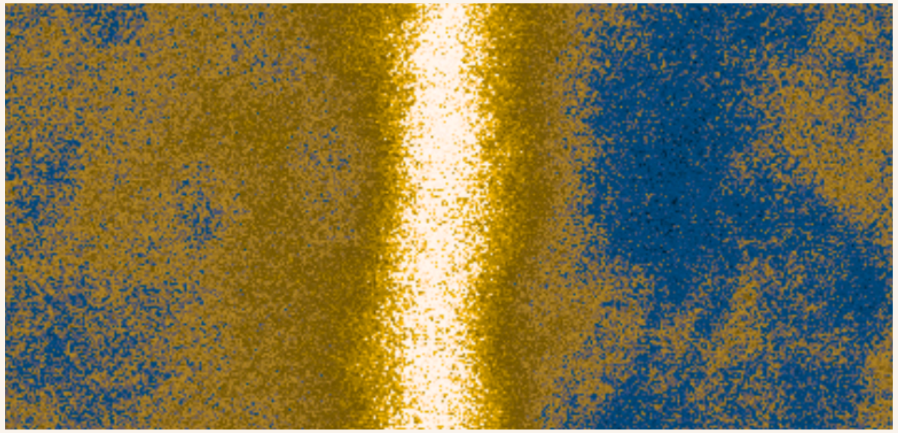
Colour blindness

Certain colour blindness forms are experienced in 1-5% of the population (biased towards males). Choice of colourmap can hugely impact the perception of data for these groups.

- In particular try to *avoid* using Red–Green to draw distinctions, as this is the most common form of colour bindness



Trichromatic ("normal") vision



Dichromatic "green-blind" vision

Python: Matplotlib, Image: Tescan / CEA

Transparency (Alpha)

Digital images can also be combined or overlaid using transparency, called **alpha**.

> Transparency can be defined on a **per-pixel** basis

For example a low-count area in an EDS map can let the HAADF show through.

- When working with colour images you may see `RGBA` where `A` is a 4th "colour" channel used for alpha

EDS Alpha: **0.50**

Complex and 2D-vector images

For complex images we must choose how convert real + imaginary into an standard image.

- A typical example is in holography, where the reconstruction is complex
  - The `abs()` of the wave represents the amplitude
  - The `angle()` of the wave displays the phase

We also need to be careful about how to display periodic phase with a colourmap:

- We can use a **cyclic** map → lose visualisation of phase ramps.
- A common technique to work around this is **phase unwrapping**

Python: `skimage.restoration.unwrap_phase`, Image: Grillo et al. (2020)

Complex image display

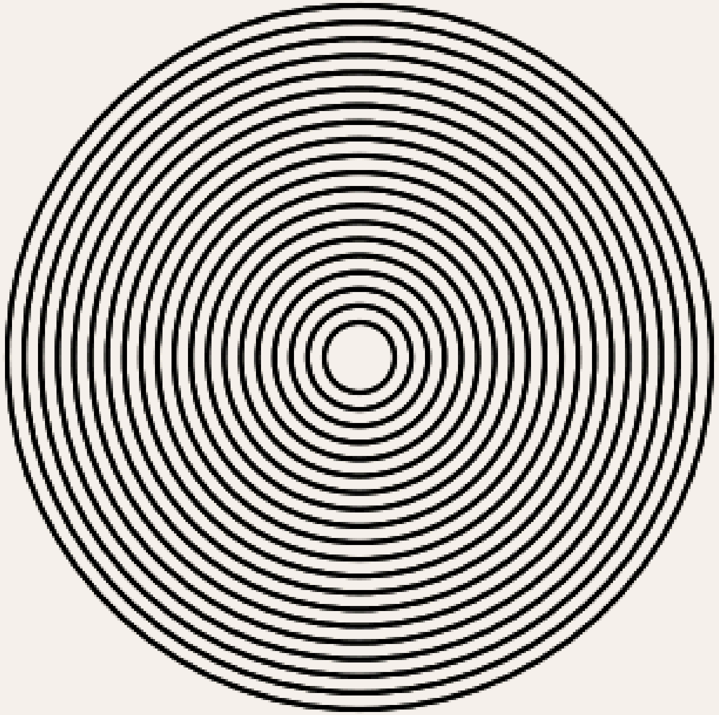
Display channel Real Imaginary

⋮

Display channel Amplitude Phase Unwrapped

⋮

Images as signals



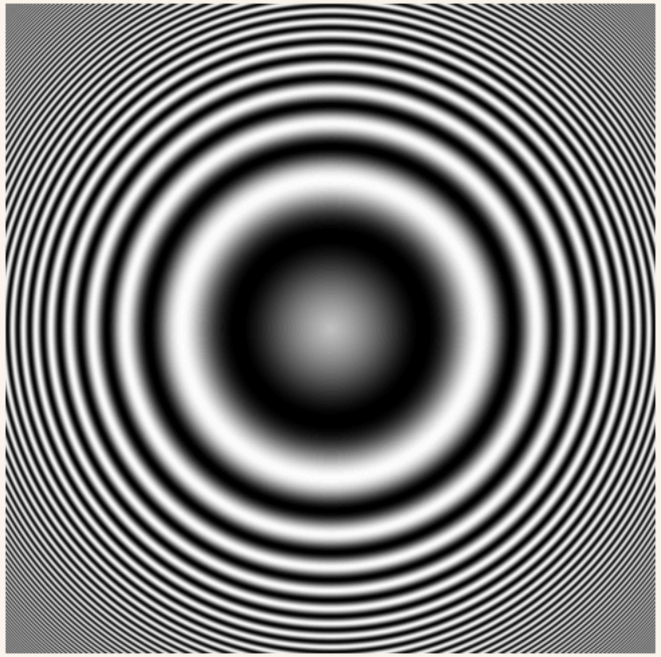
Images as signals

A digital image is a sampling of a continuous world onto a discrete grid. The *step-* or *pixel size* limits what information can be captured by the image.

Conversely, increasing pixel density adds value only if the information is there to sample:

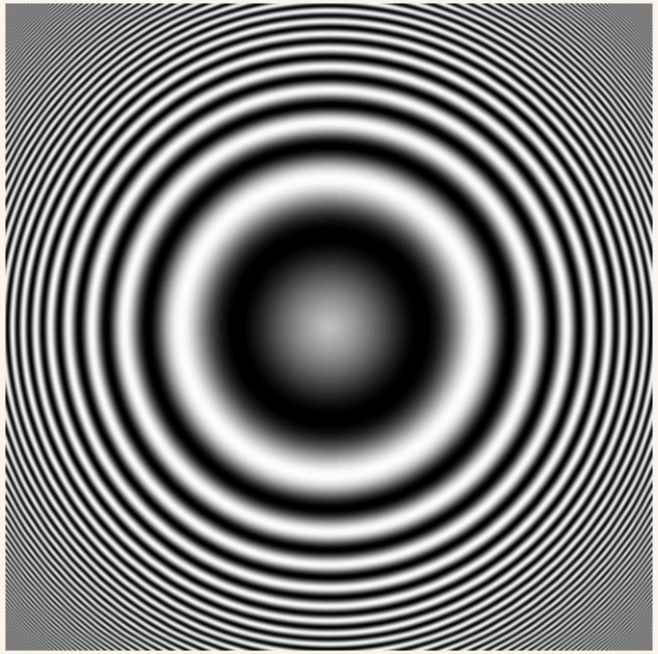
- A smooth ramp in intensity is fully defined by two points - we can **interpolate** between them to get the same quality as a densely sampled image
- If the optics of the microscope cannot cleanly resolve the detail we want to see, more camera pixels will not help, we'll just have better sampled blur
- For a periodic feature like atomic columns, **2 samples-per-period** are sufficient according to Nyqist-Shannon, if we impose the right model when displaying the data. Though this wouldn't make very interesting images!

With reduced sampling, the denser areas of the signal are not resolved.



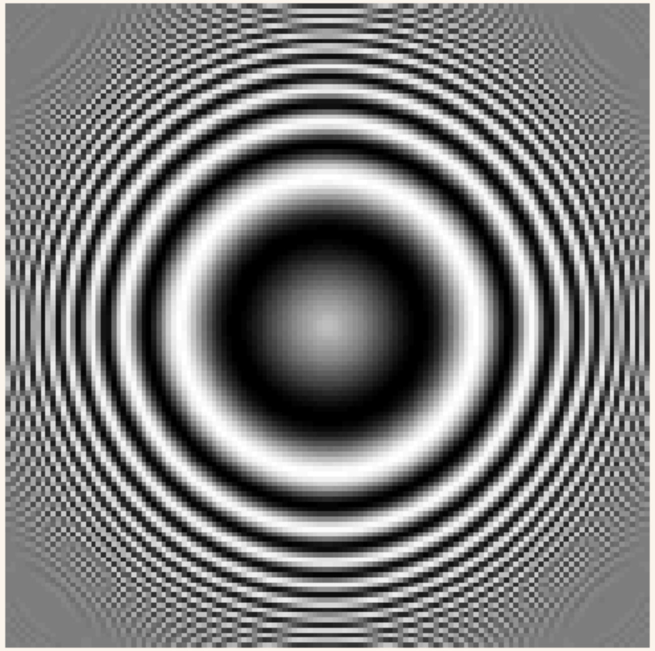
1024x1024

Oversampled



512x512

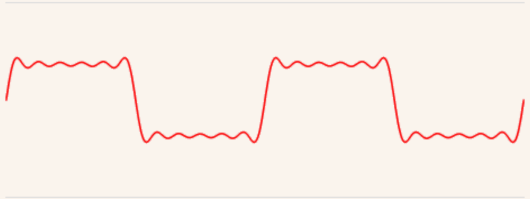
Correctly sampled



128x128

Undersampled

With extra sampling, no additional detail is added



Frequencies in 2D signals

In 1D we can perform a Fourier *transform* to describe a function $f(x)$ as a sum of periodic components each $A_u e^{-\mathrm{i}2\pi ux}$ i.e. $A_u \cos(2\pi ux + \theta) + \mathrm{i}A_u \sin(2\pi ux)$. We can evaluate the coefficients A_u:

$$A_u = \int_{-\infty}^{\infty} f(x)\, e^{-\mathrm{i}2\pi ux}\, dx, \quad \forall u \in \mathbb{R}$$

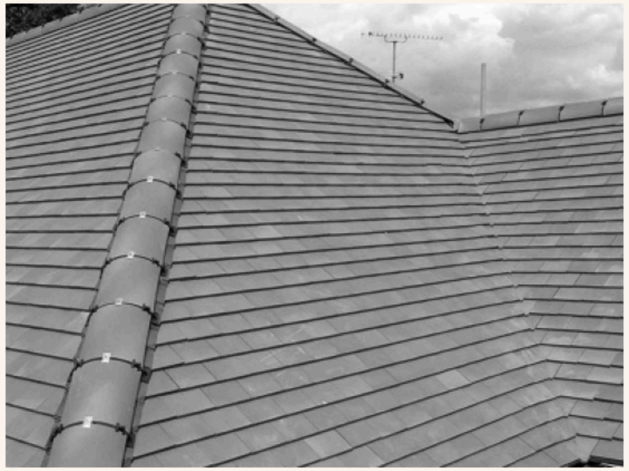
each A_u represents a contribution to $f(x)$ by a particular *frequency* u.

On an image $f(x, y)$ we can do the same, but we must use two *spatial frequencies* e.g. u, v.

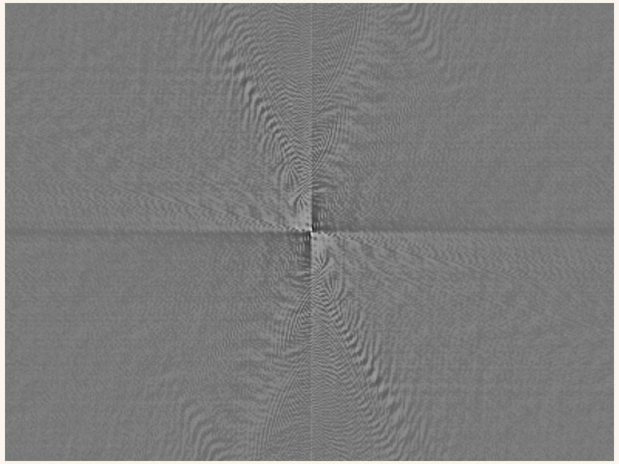
A Fourier transform can be computed efficiently with a Fast Fourier Transform (**FFT**).

Image: safeguardroofingandbuildingltd.co.uk

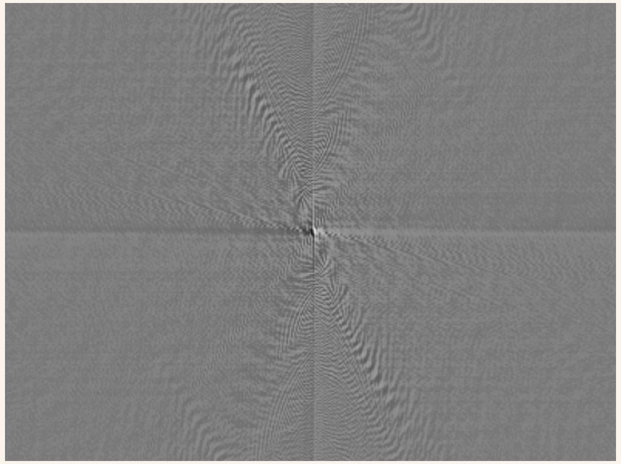
Fourier transforms are **complex-valued**, representing the $i \sin$ and $\cos$ terms.



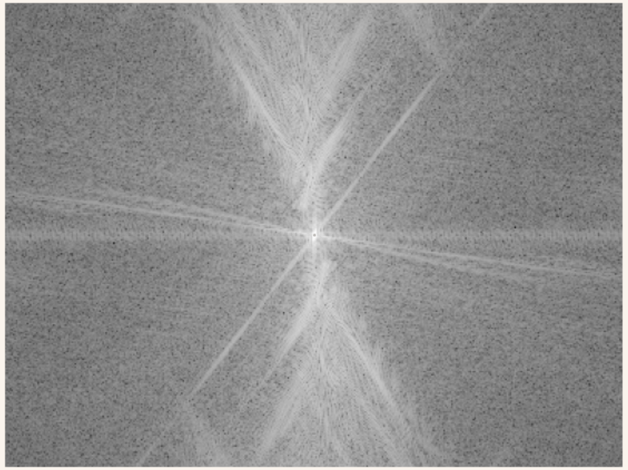
`Input image`



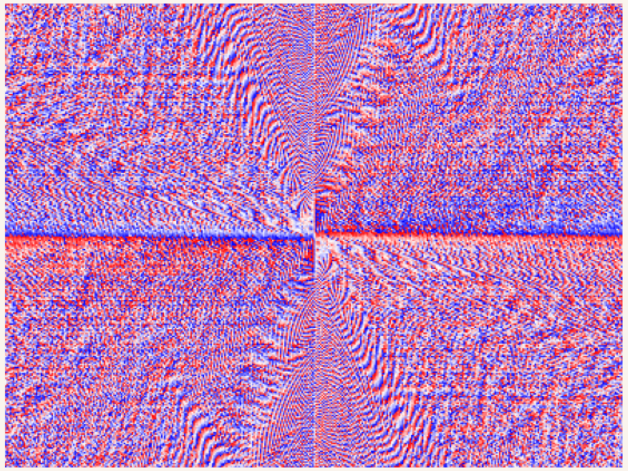
`real(FFT)`



`imag(FFT)`

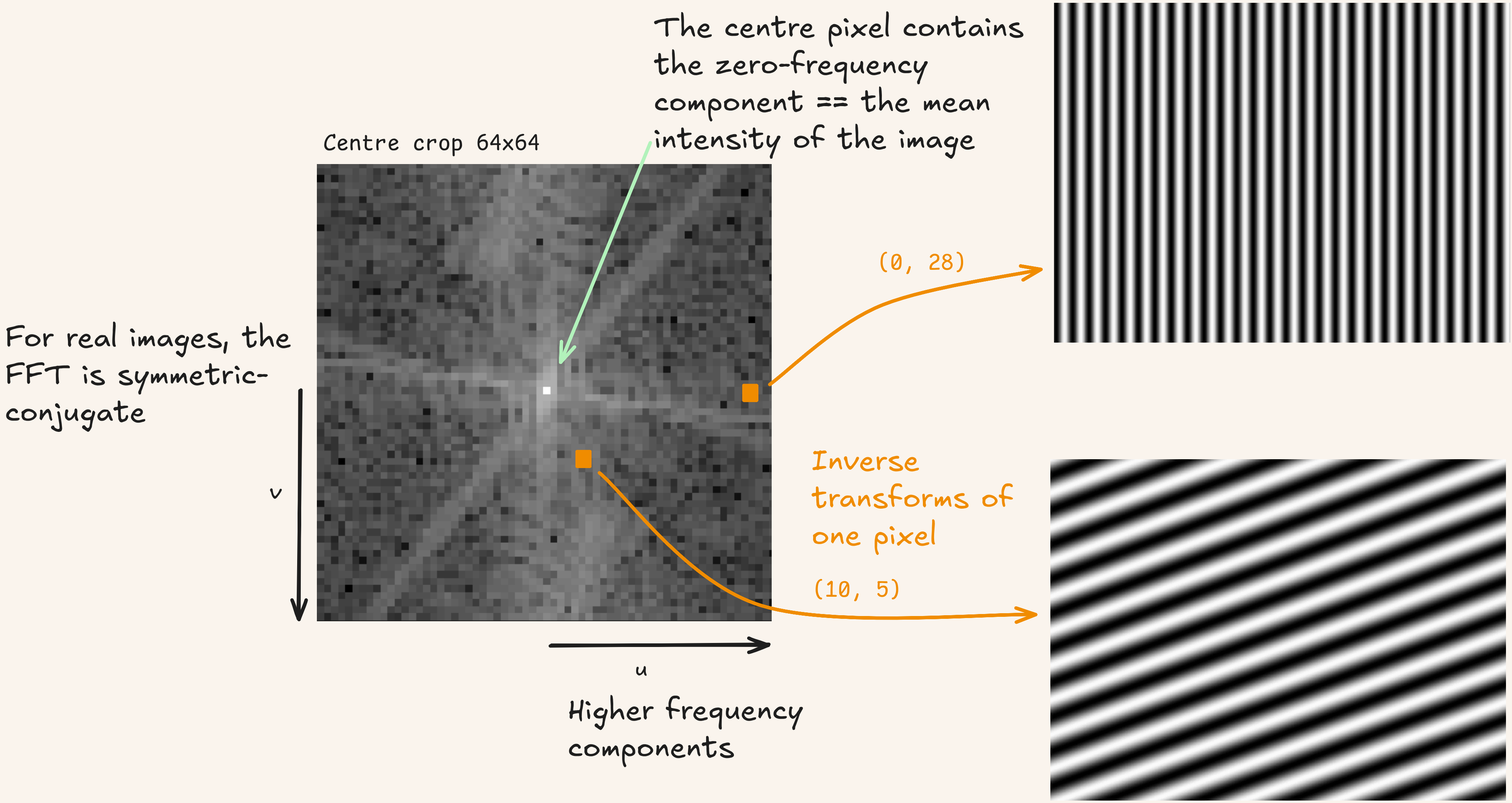


`log(abs(FFT))`



`phase(FFT)`

Fourier components



Centre crop 64x64

The centre pixel contains the zero-frequency component == the mean intensity of the image

For real images, the FFT is symmetric-conjugate

Higher frequency components

Inverse transforms of one pixel

The zero-frequency (mean value) is normally a **much** larger component than the rest!

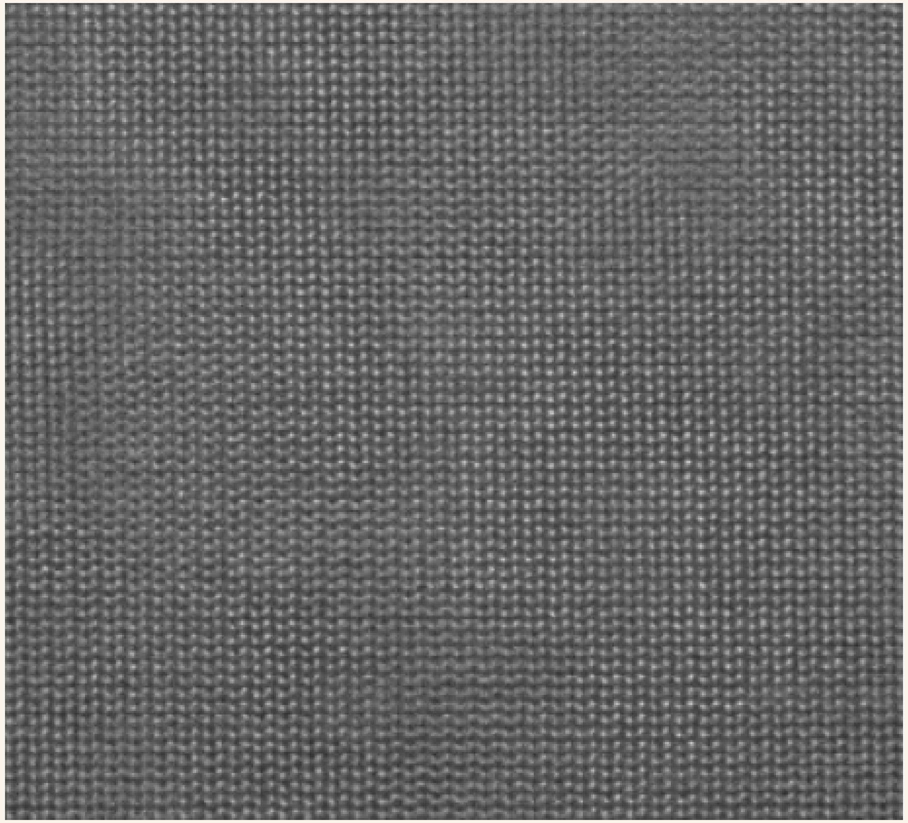
Uses of image Fourier transforms

- The transform is reversible, it contains exactly the same information as the image
- We can performing **filtering** by modifying the FFT, e.g. remove high-frequency noise while leaving the main content intact
- Many mathematical operations are much more efficient in frequency space than direct space, for example **correlation** and **convolution**

Image: Jean-Luc Rouvière

Fourier transforms in Microscopy

High-resolution images of atomic columns are naturally periodic, and lattice spacings appear clearly in the amplitude of an FFT.



FFT →

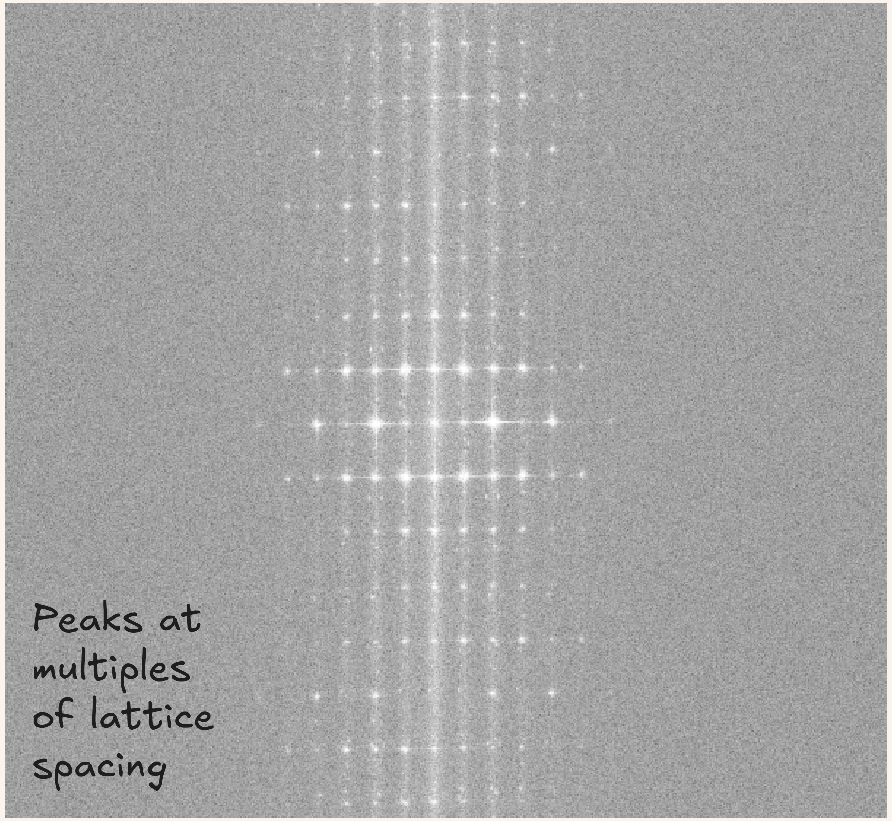


Image: Grillo et al. (2020)

Fourier transforms in Microscopy

Electron holography uses FFTs to extract information from the interference pattern created by the biprism:

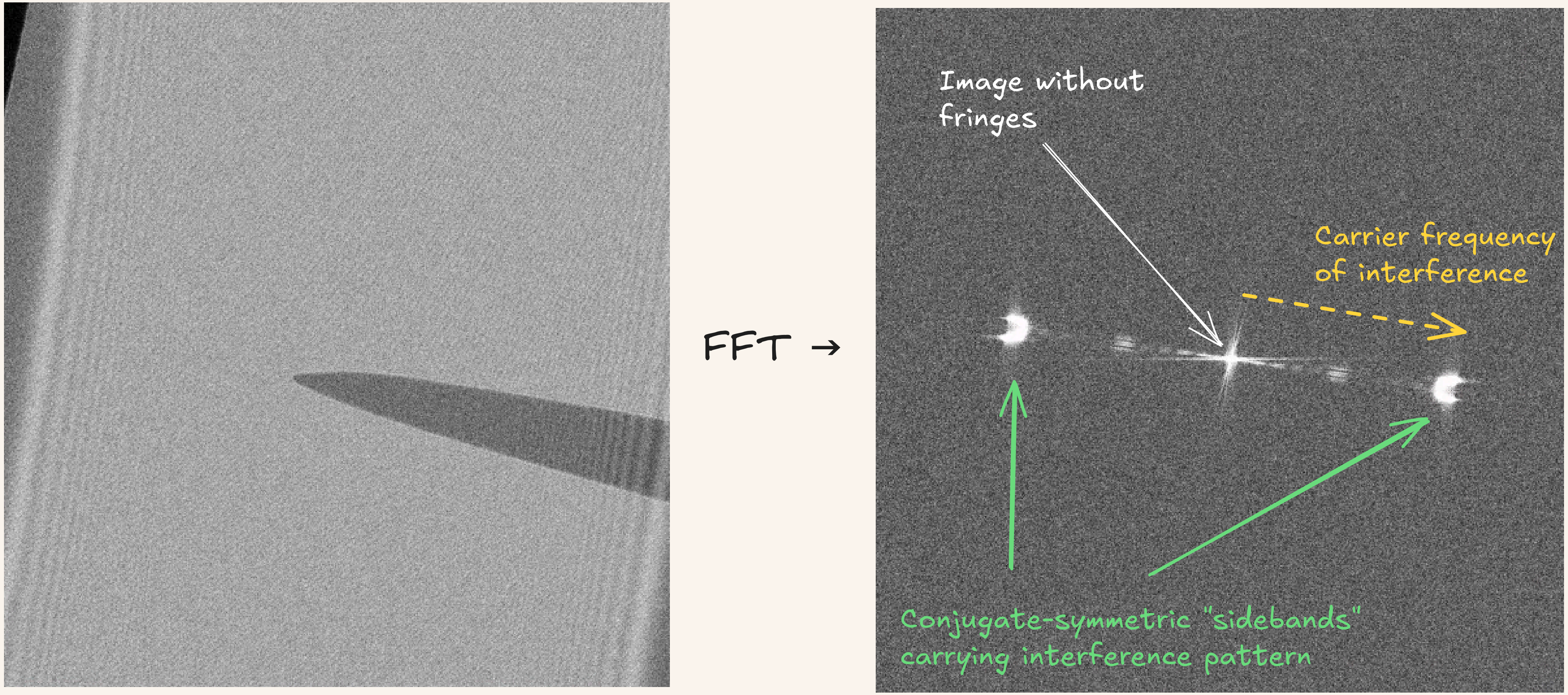
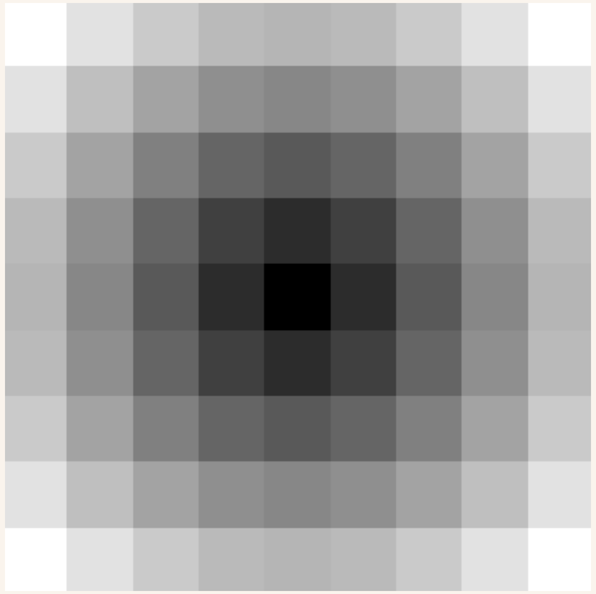

FFT →
Image without fringes
Carrier frequency of interference
Conjugate-symmetric "sidebands" carrying interference pattern

Image interpolation

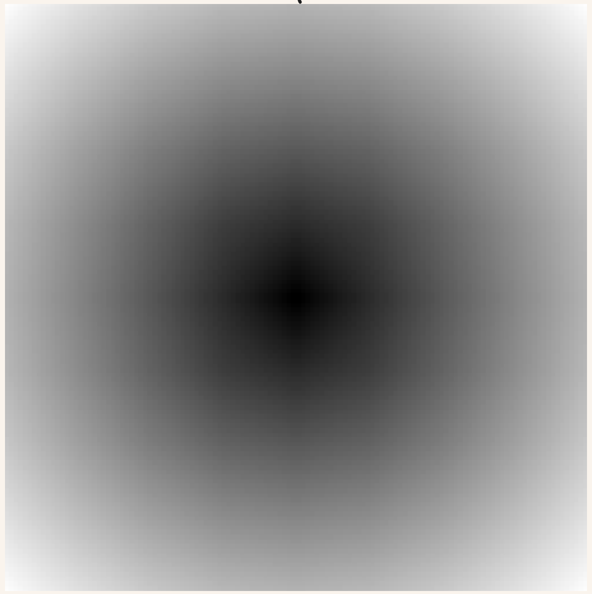
A discrete image can be *interpolated* into a continuous coordinate system so that it can be **re-sampled** at new coordinates.

- Interpolation does not add additional information, but can reconstruct a higher-fidelity version of the image if we have a good model of the true signal.
- Interpolation is one method to acheive **sub-pixel** resolution in measurements, for example finding the position of intensity peaks in an image

9 x 9 Raw data



128 x 128 Interpolation

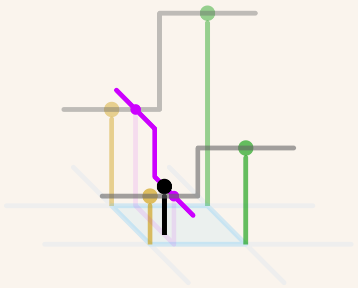


Adapted from Cmglee CC BY-SA 4.0

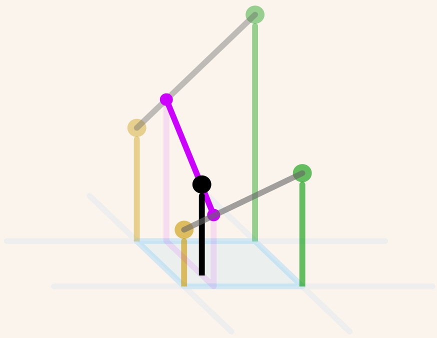
Interpolation schemes

Interpolating schemes can be very basic (e.g.piecewise constant) or very flexible (polynomial splines).

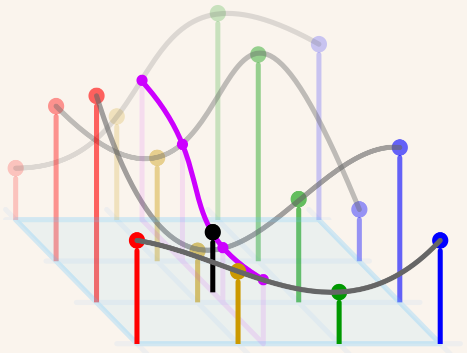
- Interpolation can **smooth** an image if desired → the interpolant doesn't perfectly reproduce values at input positions.
- Also possible to interpolate an image from unstructured samples (i.e. not originally on a grid).



2D nearest-neighbour



Bilinear



Bicubic

Python: `libertem_ui`, Image: Gustav Persson

Interpolated line profile from image

Sampling (pt-per-px): 1.5 Clear Remove pt

...

Averaging (px): 0

...

Aliasing

A signal sampled at lower than its highest frequency can be subject to **aliasing**. The samples will ambiguously fit both the true signal and other signals at combinations of the true and sampling frequency.

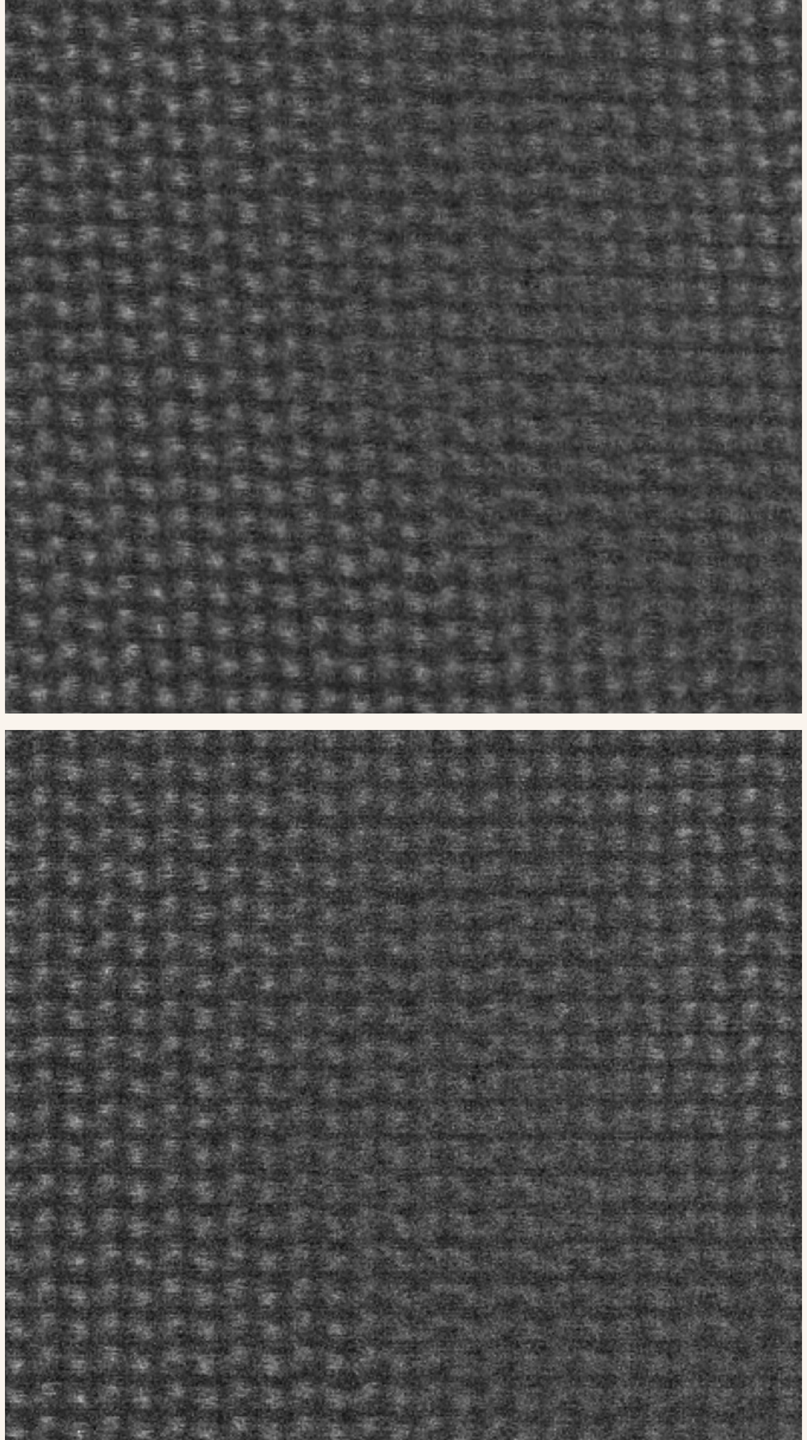
...

☑ Sin(f) □ Cos(f) □ Sin(1/2f)

Interpolation: ○ Linear ● Sinc()

Sampling rate: **34**

Image: Jean-Luc Rouvière

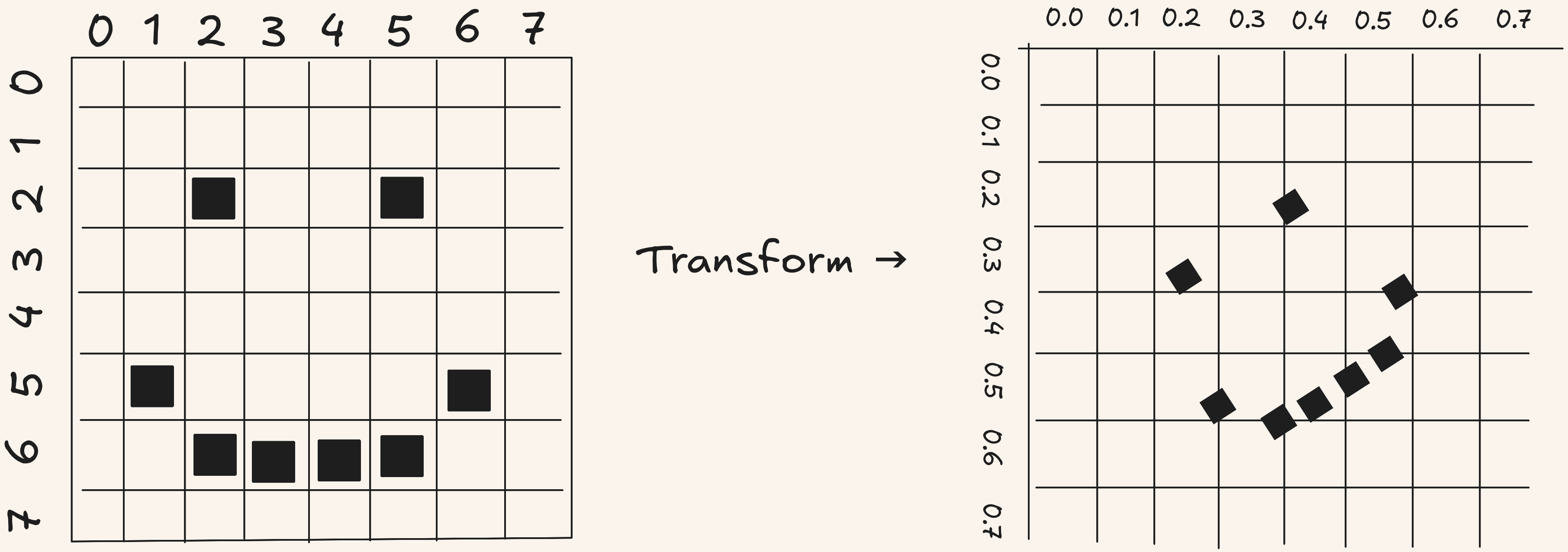


Images and Geometry

Python: `skimage.transform`

Geometric transforms of images

The information in an image exists on a coordinate grid. We can map it onto a new grid using a **transform**, and so translate, stretch, rotate, shear, or generally **warp** the data.

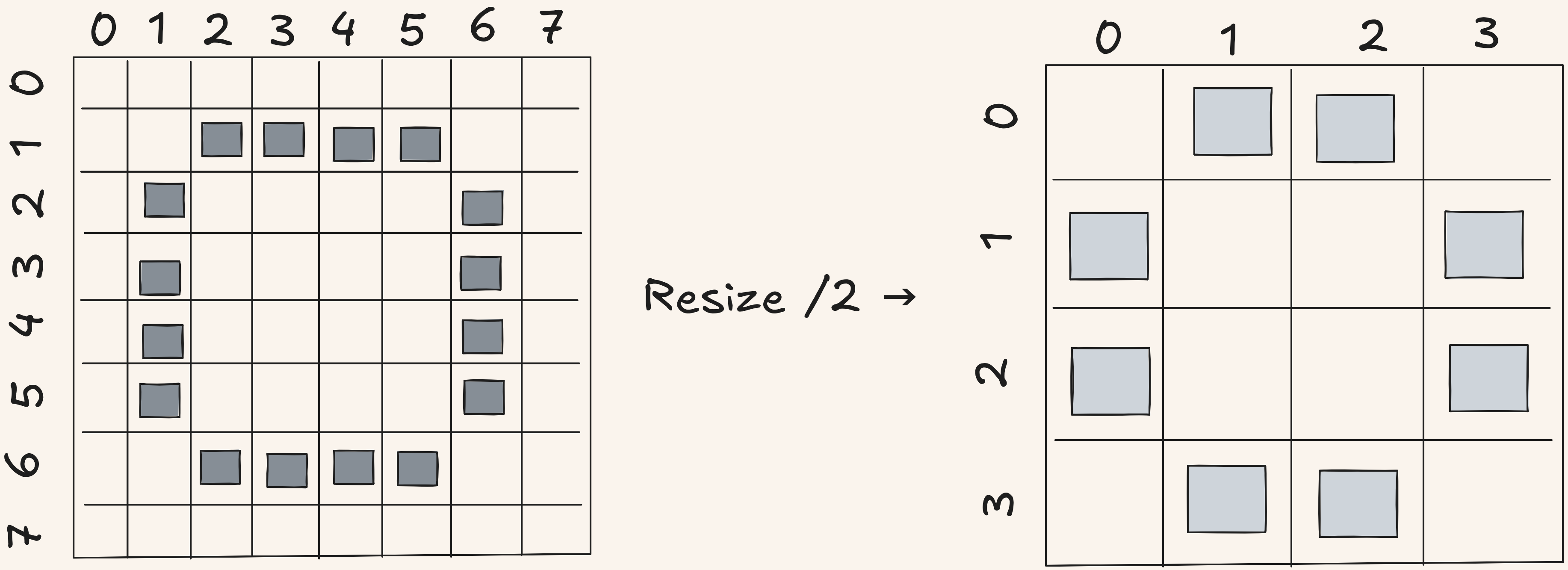


Python: `skimage.transform.resize`

Resizing

Image rescaling maps, for example, pixel `[5, 3]` to `[5 * scale, 3 * scale]`, for all pixels.

The new image is generated by sampling new pixel coordinates via **interpolation**.



Distinct from **binning** as we are not limited to integer scale factors.

Matrix transforms

Many coordinate transformations can be represented as a matrix multiplication.

- We just saw rescaling, which can be represented as:

$$\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}$$

After which we can *interpolate* on the grid x', y' to create the transformed image.

(Affine) Matrix transforms

Other uniform transformations include:

$\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & \lambda & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}$
Scaling x + y	Rotation by θ	Flip-y	Shear x	Shift x + y

These can be chained to create more complex transforms e.g.

$$Scale \times Shear \times Shift \times \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}$$

> The row / column `0, 0, 1` is called a *homogeneous coordinate* and allows **translation**.

Image: Jean-Luc Rouvière

(Affine) Matrix transforms

Scale up	Rotate ↶	Shear-X +	Shear-Y +	Shift-X →	Shift-Y ↓
Scale down	Rotate ↷	Shear-X -	Shear-Y -	Shift-X ←	Shift-Y ↑

Clear

Python `skimage.transform.PolynomialTransform`, Image: Jean-Luc Rouvière

Polynomial transform

Run Reset Show original Load

Affine transforms preserve straight lines and parallelism - but in some cases we may need to **correct curves**, e.g in STEM with sample drift.

A very flexible transform is a polynomial transform, which has the general form:

$$x' = \sum_{j}^{N} \sum_{i}^{j} a_{ij} x^{j-i} y^{j}$$

mapping x, y to x' (equivalently to y' with additional b_{ij}).

Python `skimage.transform.warp_polar`

Polar image transform

Some images, e.g. diffraction patterns, can be interpreted in **polar coordinates** (r, θ).

This can be acheived another type of non-affine coordinate transform:

$$\sqrt{(x-c_x)^2+(y-c_y)^2} \rightarrow r$$

$$\arctan\left(\frac{y-c_y}{x-c_x}\right) \rightarrow \theta$$

We generate this mapping for all (x, y) in the image, then interpolate at the (r, θ) we want to display a new image for.

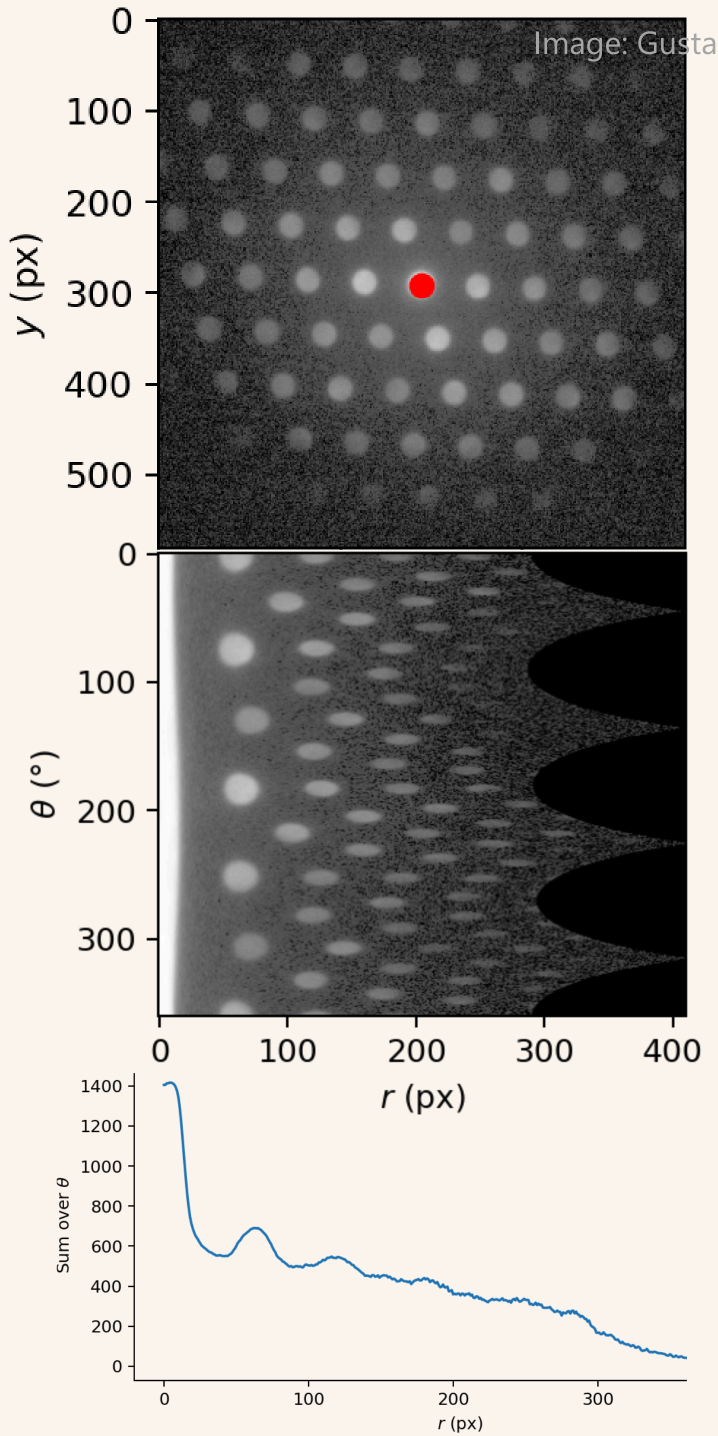


Image: Gustav Persson

Image Filtering

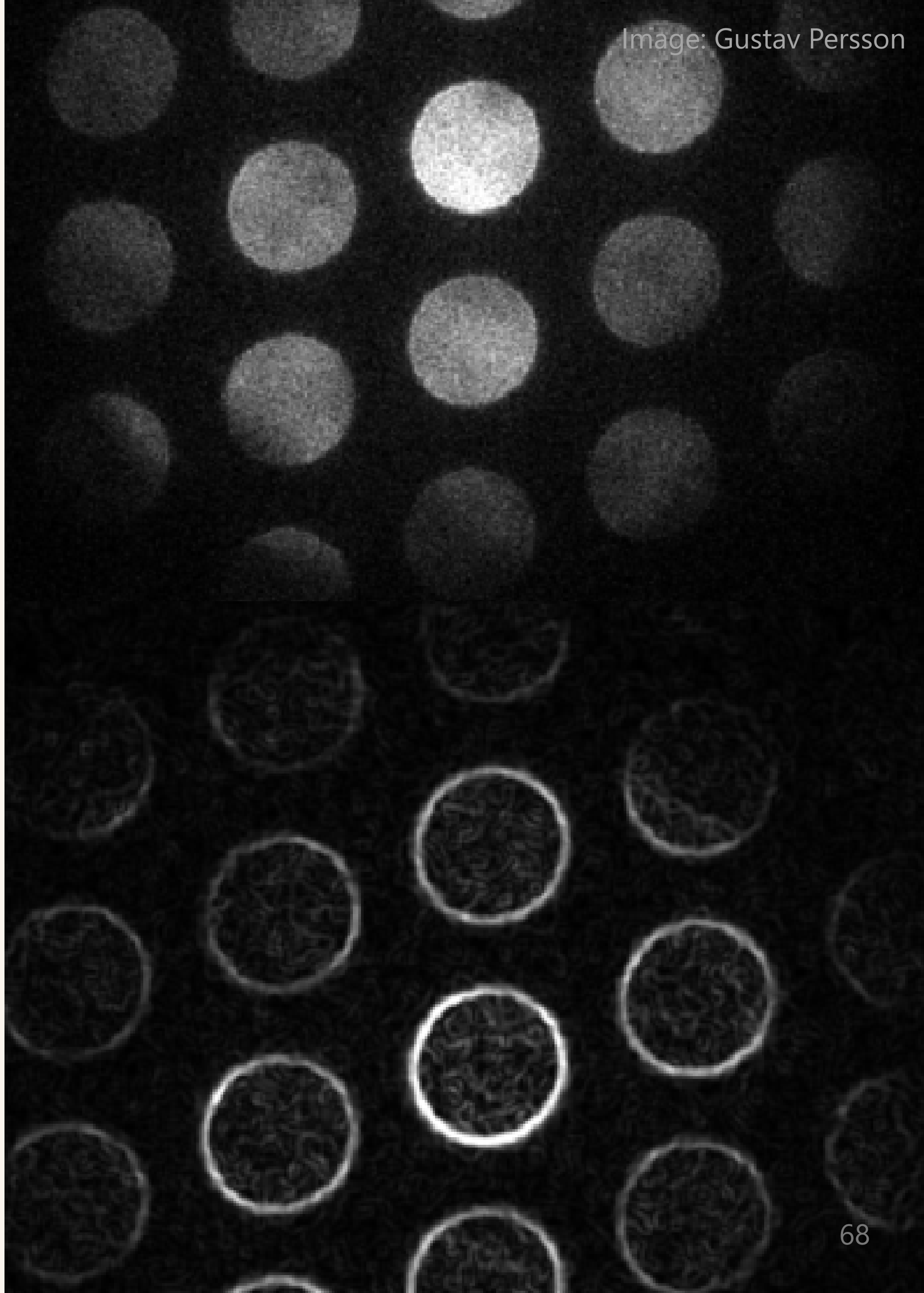
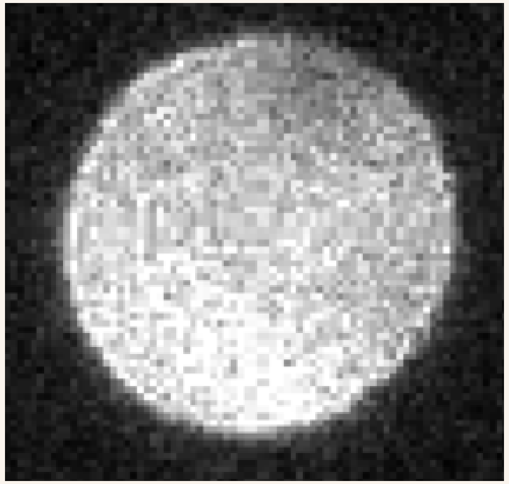


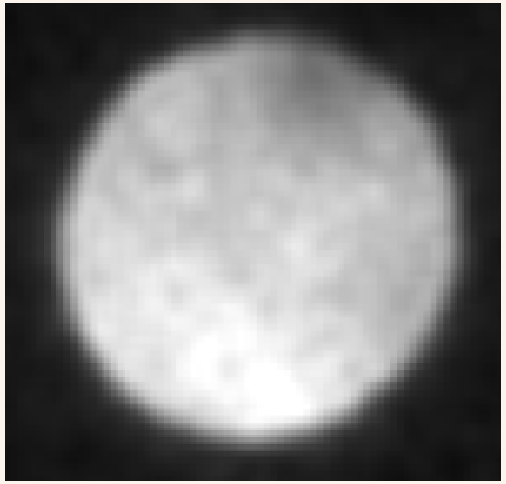
Image: Gustav Persson

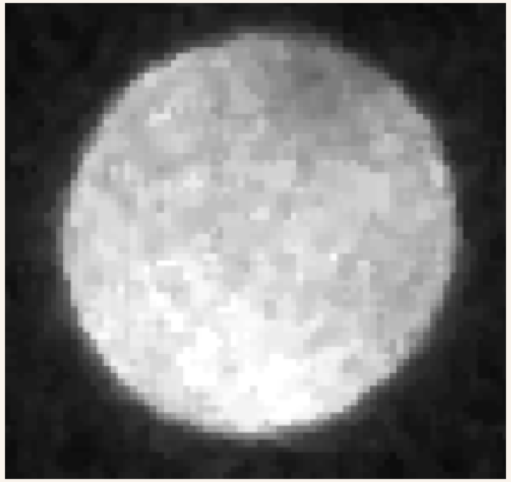
Python `skimage.filters`

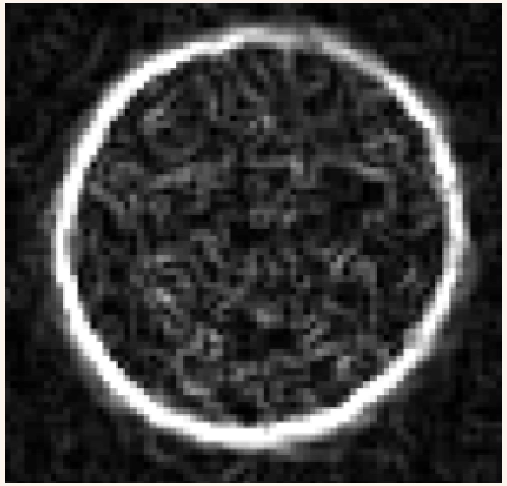
Filtering

Filters enhance certain information in an image, compensate for issues in the imaging system or highlight properties of the image which are beyond a simple intensity distribution.


Raw


5x5 Gaussian

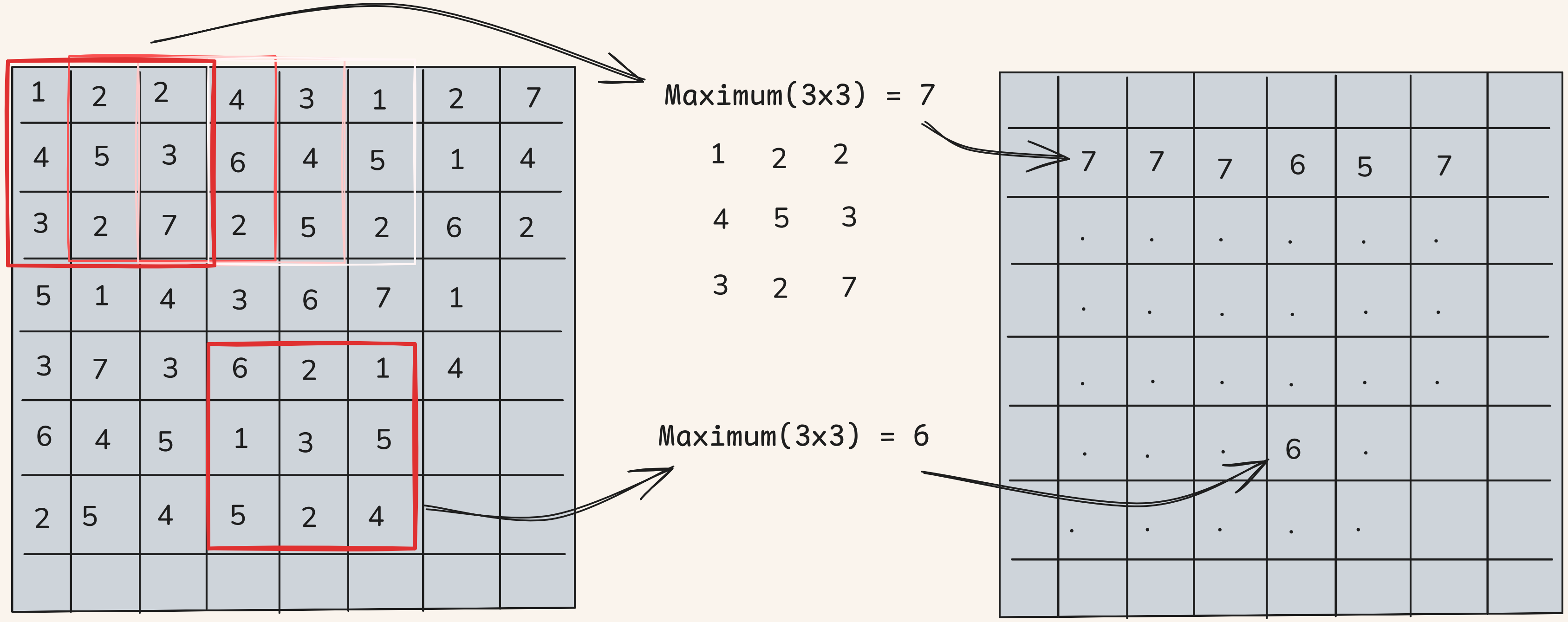

3x3 Median


3x3 Sobel

Filtering is usually a **pre-processing step** before applying other methods.

Patch-based filters

A simple type of filter is *patch-based*. These run a procedure in the vicinity of each pixel to generate a new value for that pixel.



1	2	2	4	3	1	2	7
4	5	3	6	4	5	1	4
3	2	7	2	5	2	6	2
5	1	4	3	6	7	1	
3	7	3	6	2	1	4	
6	4	5	1	3	5		
2	5	4	5	2	4		

Maximum(3x3) = 7

1	2	2
4	5	3
3	2	7

Maximum(3x3) = 6

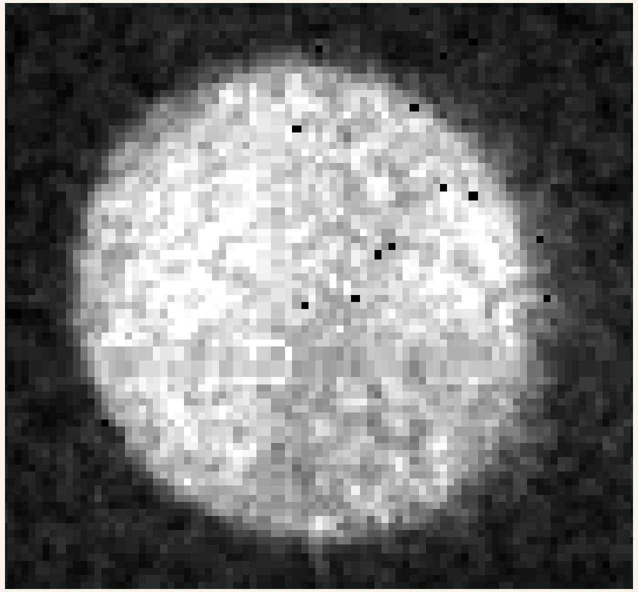
	7	7	7	6	5	7	
	.	.	.	.	.	.	
	.	.	.	.	.	.	
	.	.	.	.	.	.	
	.	.	.	6	.		
	.	.	.	.	.		

> Edges need special treatment as their neighbourhood is limited, else the filtered image becomes smaller. Padding with zeros, periodic boundaries or reflecting the boundary are common ways to handle this.

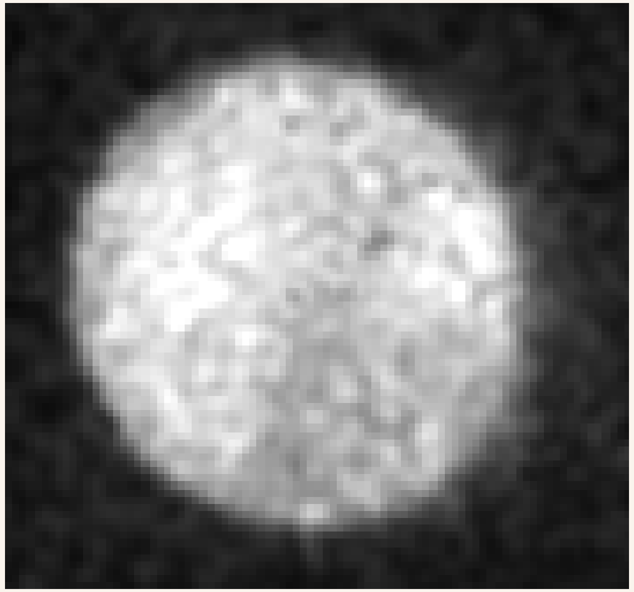
Python `skimage.filters.gaussian`

Gaussian blur

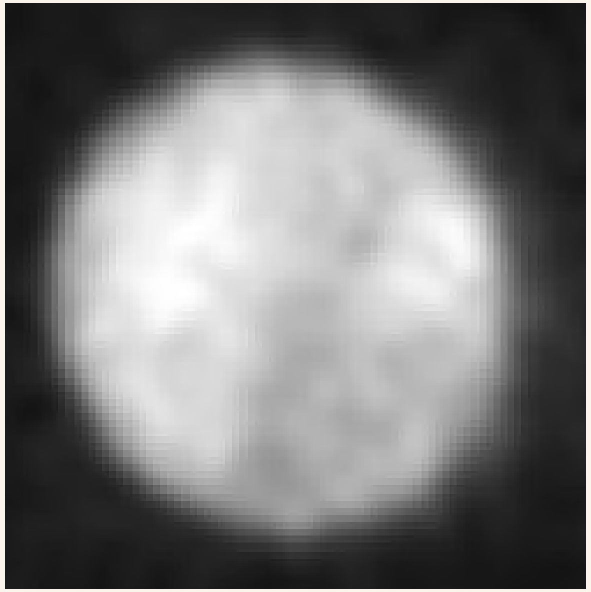
Gaussian blur is a patch-based filter which computes a local **Gaussian-weighted average** of values in each pixel's local neighbourhood.



Raw



Blur 3x3

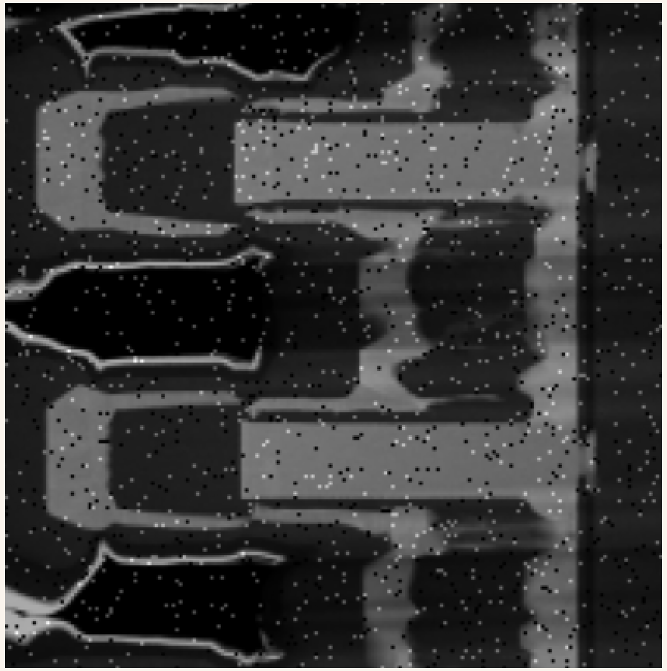


Blur 5x5

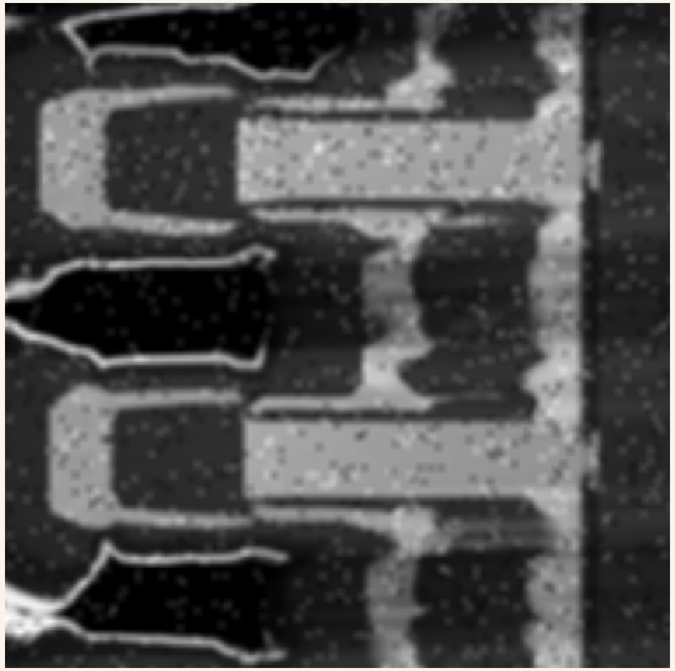
Python `skimage.filters.median`, Image: Jean-Luc Rouvière

Median filter

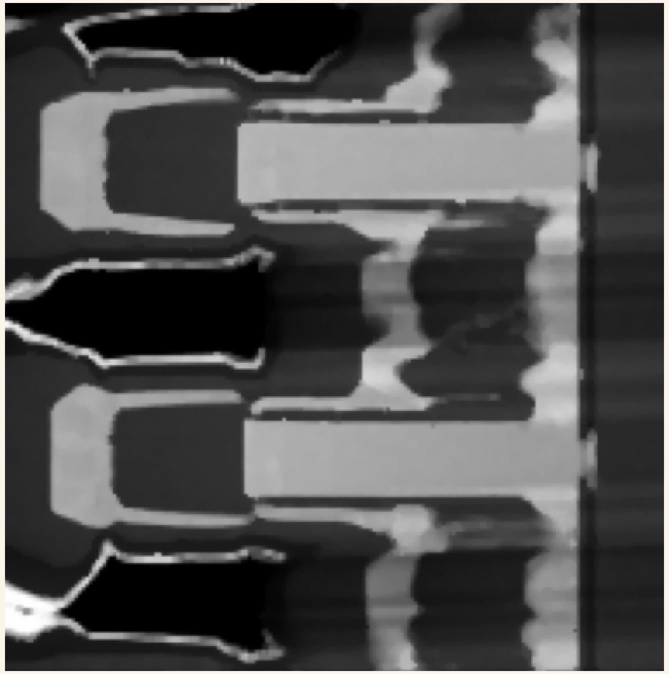
The Median filter is a patch-based filter which is quite useful for removing **extreme values**, for example hot or dead pixels. A Gaussian blur would incorporate these unwanted extremes into the blurred image.



Noise- Corrupted



Blur 3x3



Median 3x3

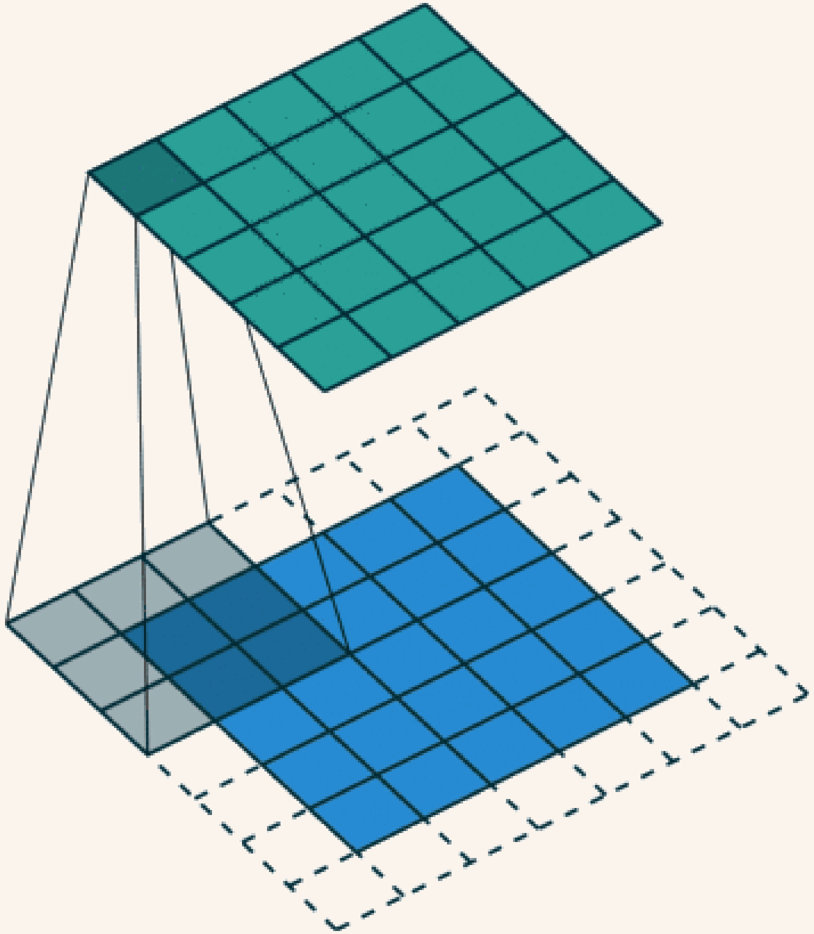
Convolution filters

Convolutional filters are a class of patch-based filters using elementwise multiplication and summation with a small *kernel* to compute each new pixel value.

- They can be efficiently computed using a Fourier transform since

$$\hat{F}(a * b) = \hat{F}(a)\hat{F}(b)$$

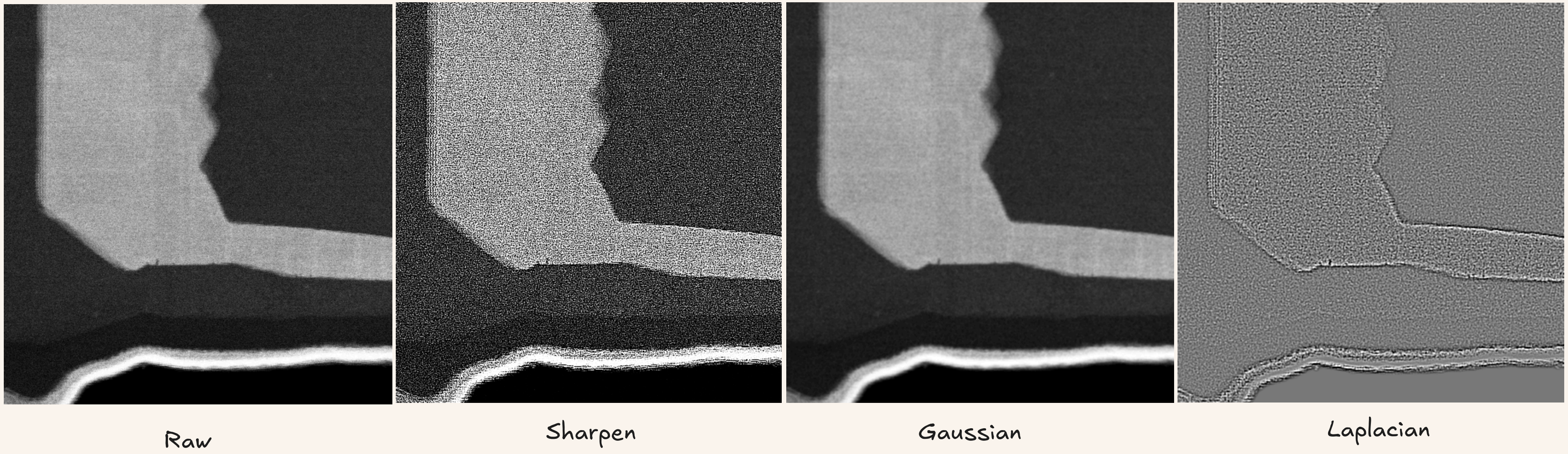
GPUs are *very* efficient at computing image convolutions.



Dumoulin and Visin (2016)

Example kernels

$$\begin{bmatrix}-1 & -1 & -1\\ -1 & 17 & -1\\ -1 & -1 & -1\end{bmatrix} / 9 \qquad \begin{bmatrix}0.02 & 0.04 & 0.02\\ 0.04 & 0.81 & 0.04\\ 0.02 & 0.04 & 0.02\end{bmatrix} \qquad \begin{bmatrix}0 & -1 & 0\\ -1 & 4 & -1\\ 0 & -1 & 0\end{bmatrix}$$



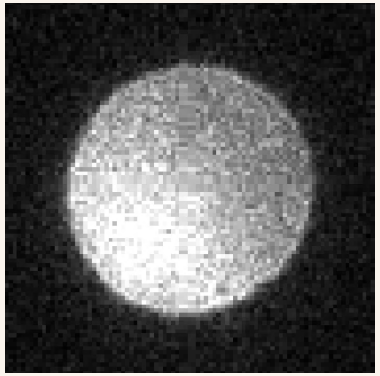
Kernels can be designed to respond to arbitrary features, e.g. corners or textures.

- Convolution underpins many image neural networks, filters guide classification

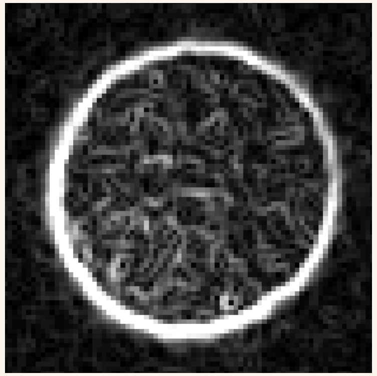
Python `skimage.filters.sobel`

Edge filters (Sobel filter)

Edge filters respond to **sharp transitions** in image intensity, or large image gradient, and are useful in applications like peak finding or contour detection for metrology.



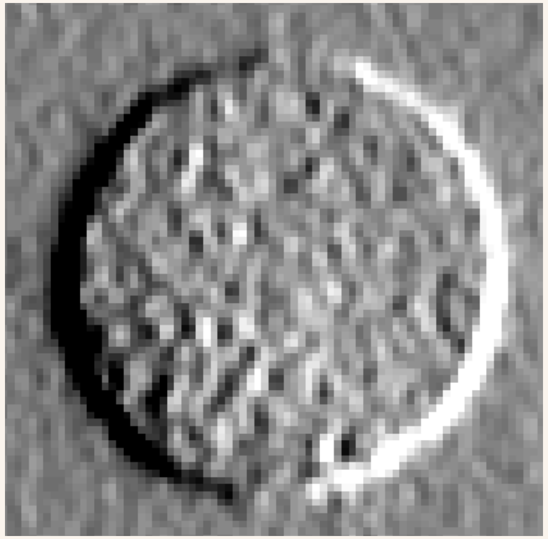
Raw



Horizontal^2
+
Vertical^2

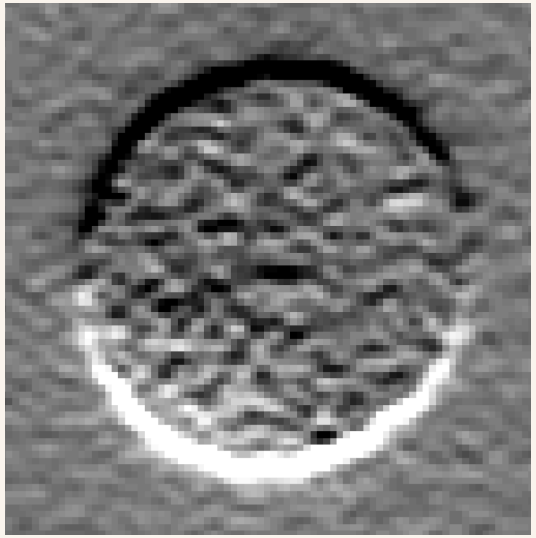
$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

Horizontal edges



$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Vertical edges



> The size of the filter influences whether it catches sharp edges or soft edges.

Frequency space filtering

Zero-ing or modifying frequencies in the FFT of an image acts as a filtering process.

The most well-known are:

- *Low-pass* or *high-cut*, which **retain low-frequency** information like gradients
  - Block the FFT far from the centre
- *High-pass* or *low-cut*, which **retain high-frequency** information like edges
  - Block the central part of the FFT
- *Band-pass* → **cut both** high-frequency and low-frequency information
  - Block everything except a ring of frequencies

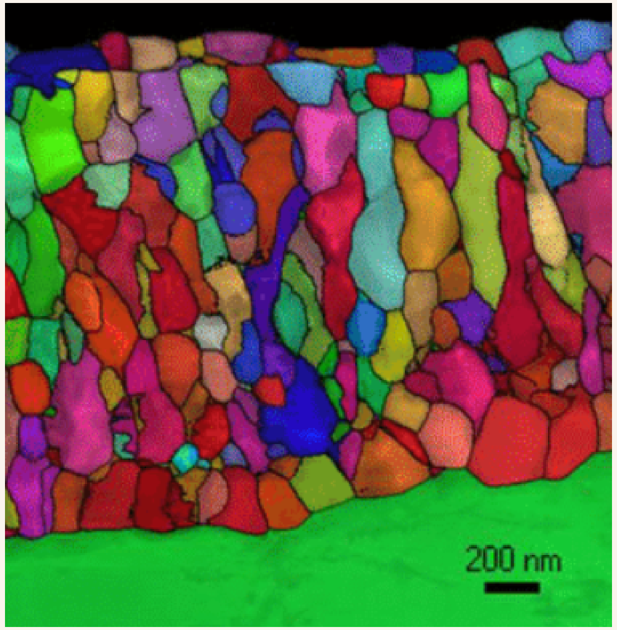
Image: safeguardroofingandbuildingltd.co.uk

Frequency space filtering

[✓ Low-pass] High-pass Band-pass

Filter radius: **400**

Image Segmentation



Latu-Romain (2021)

Python `skimage.segmentation`

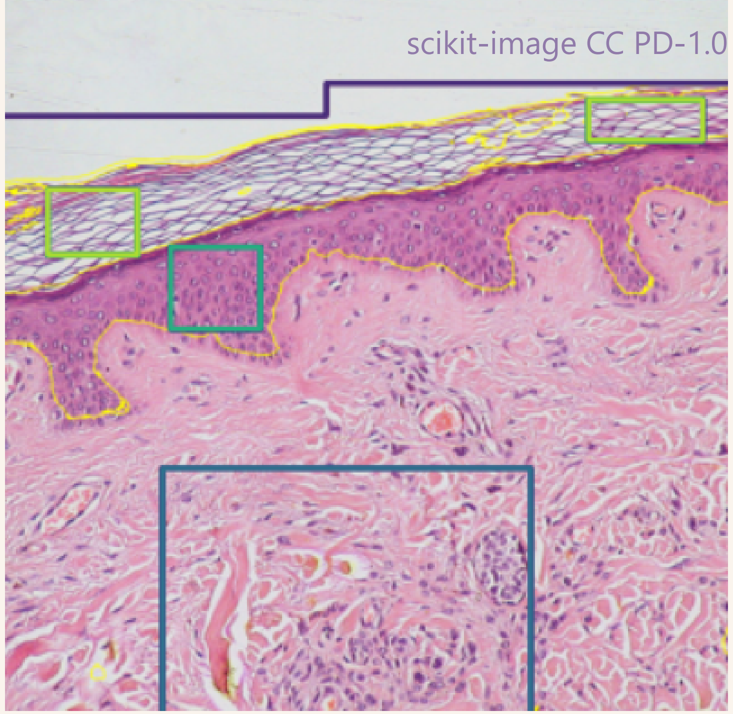
Image segmentation

Image segmentation algorithms label pixels of an image based on what they each represent

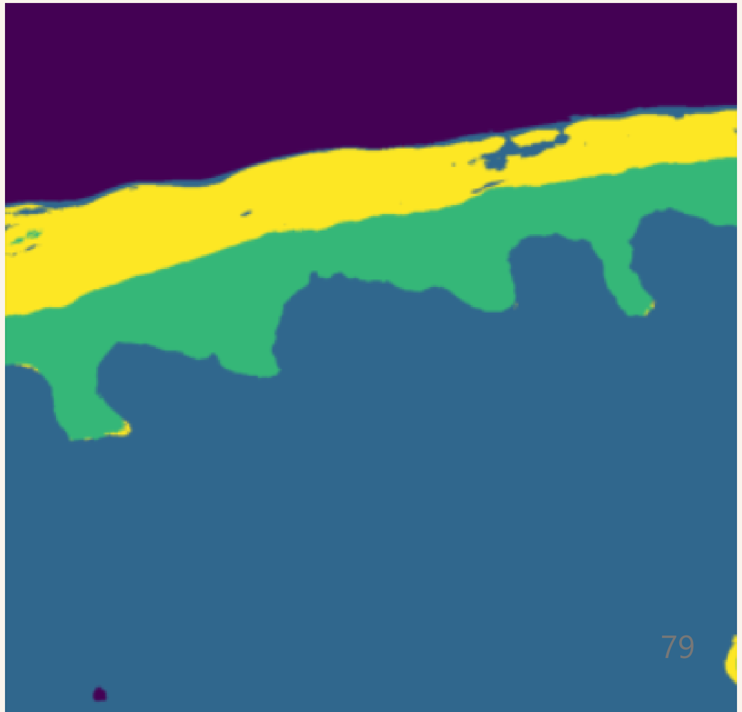
- Poly-crystal *phase and orientation mapping* is a form of image segmentation, for example to measure a grain size distribution

Segmentation algorithms can use *local*- and *non-local* information to label a pixel:

- Intensity of the pixel and its neighbours
- Location of a pixel with respect to edges / shapes
- Texture in the region of the pixel



scikit-image CC PD-1.0



Binary thresholding

The simplest segmentation is a **hard cut in intensity**: above the cut is assigned category `1` or `True`, below a `0` or `False`. For simple, high-contrast data this is often sufficient.

⋮ …

Python: `skimage.morphology`, Diagram: Mardiris (2016)

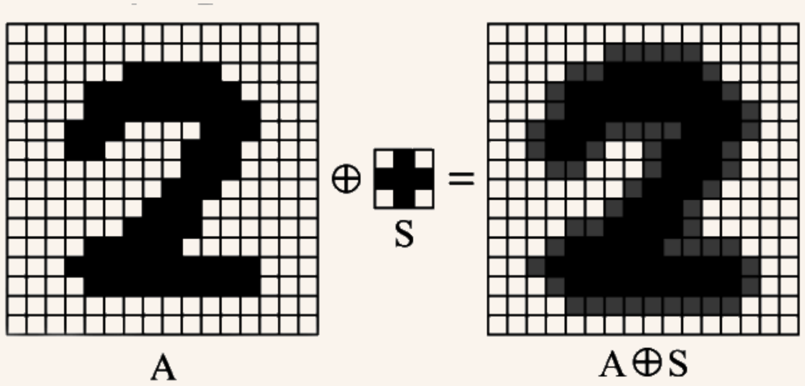
Binary image operations

Reset | Skeletonize | Remove small regions

Erode | Dilate | Fill small holes

A binary image can be modified using **morphological operations**, which shrink or expand a region, or fill holes.

A "footprint" array is convolved with the binary image, where this overlaps `True` pixels we modify according to some rule.



`skimage.measure.label`

Image Labelling - Connected Components

The **connected components** algorithm can be used to number isolated regions in a binary image, allowing us to count and measure properties like *area* and *diameter*.

The algorithm propagates the label of adjacent `True` pixels, or creates a new label, until no unlabelled pixels remain.

`skimage.measure.label`, `regionprops`, Image: NISE

Connected Components example

Image Threshold Components

⋮

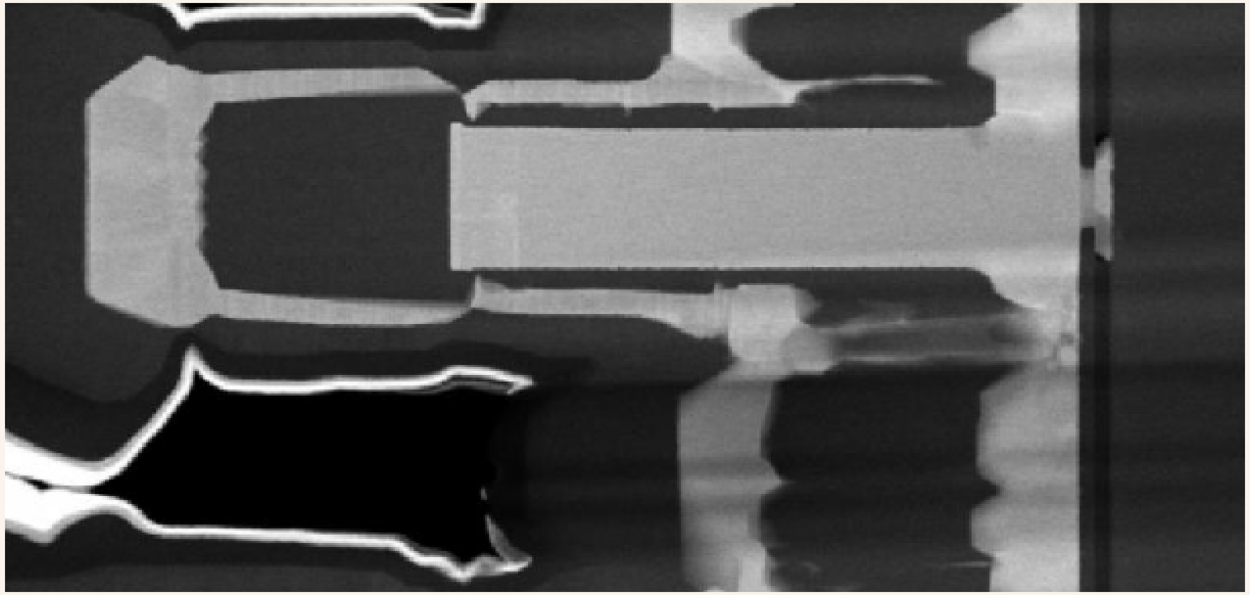
Threshold: **0.50**

…

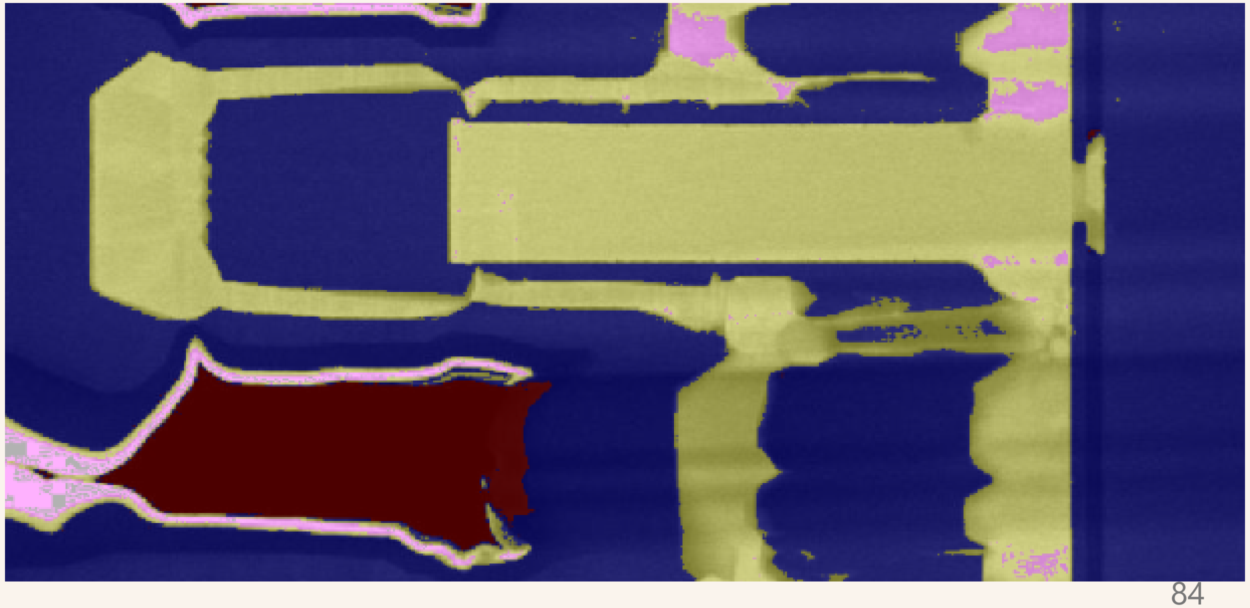
Image: Jean-Luc Rouvière

Multi-level thresholding

If the image contains multiple regions at different intensity levels then we can repeatedly apply intensity thresholding to segment it.



4 intensity threshold bands



Python: `skimage.feature`

Image features

When an image contains *intensity gradients or noise* then threshold-based segmentation can be impossible.

More advanced algorithms compute **feature vectors** on the data - combining intensity, edges, textures etc. - to distinguish categories which share properties.

Classical approaches include:

- Gabor filters
- Gray-level co-occurrence matrices
- Local binary patterns

In practice probably use deep learning!

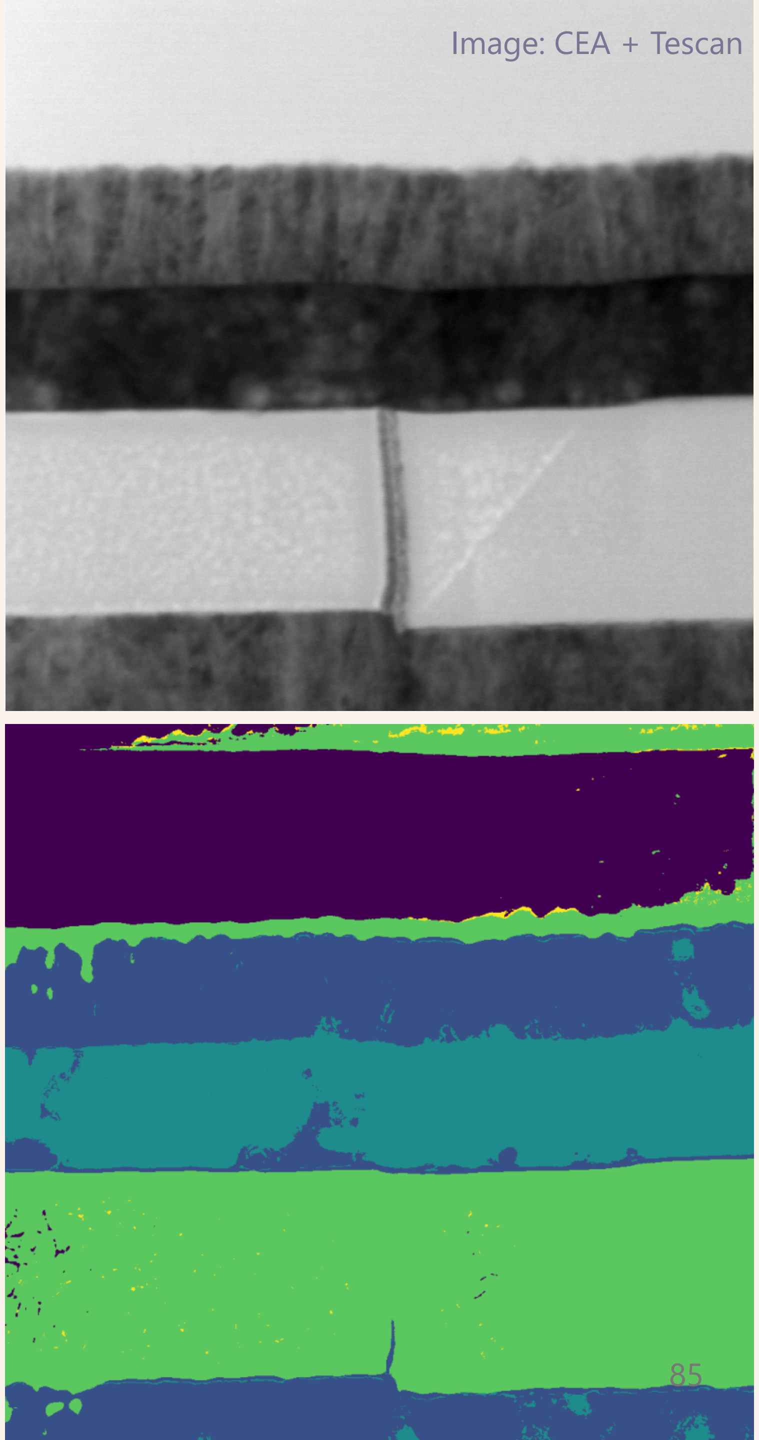


Image: CEA + Tescan

Deep learning for image segmentation

Image segmentation was an early application of **convolutional neural networks** (CNNs), particularly as image features are difficult to construct. The model can instead learn optimal features for the data it is trained on.

The most well-know, albeit now quite old architecture are the **U-Nets**, which are designed to combine information at multiple image scales to inform the segmentation.

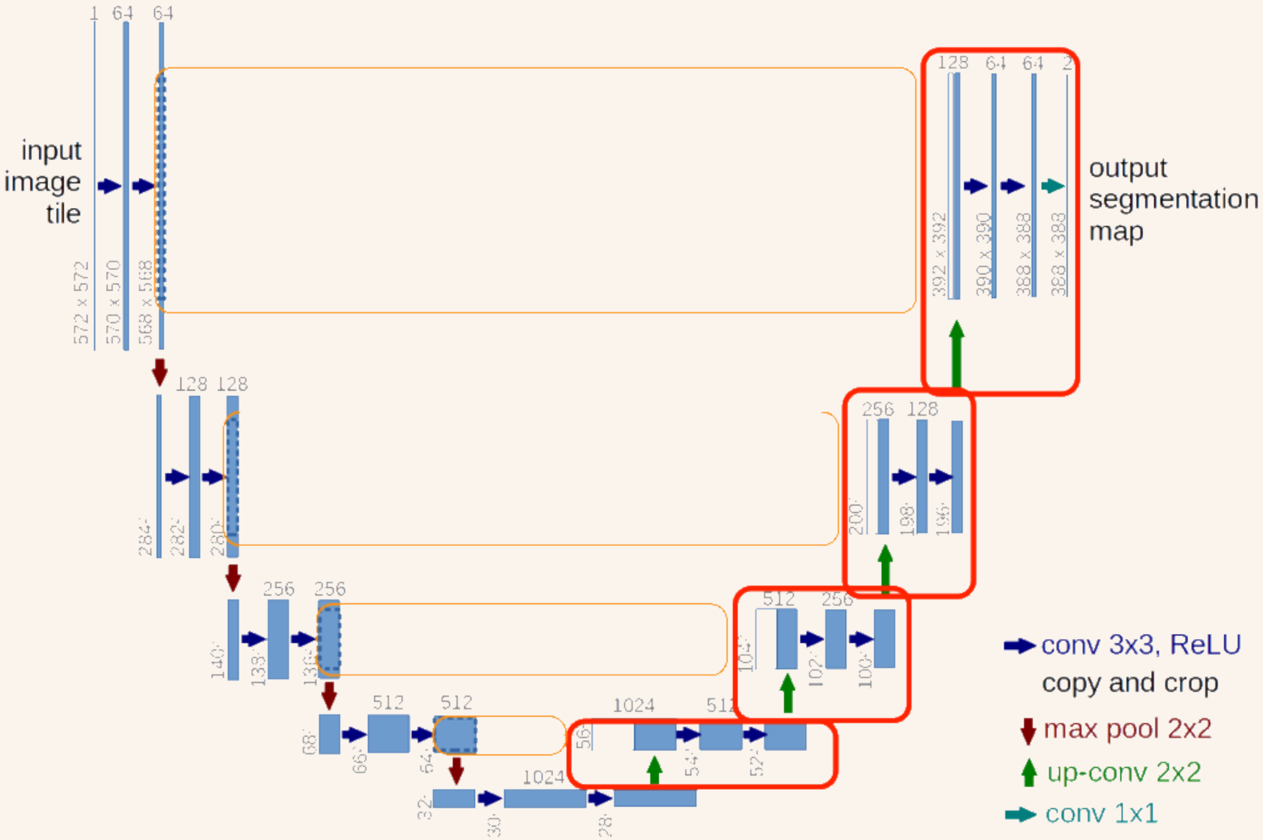
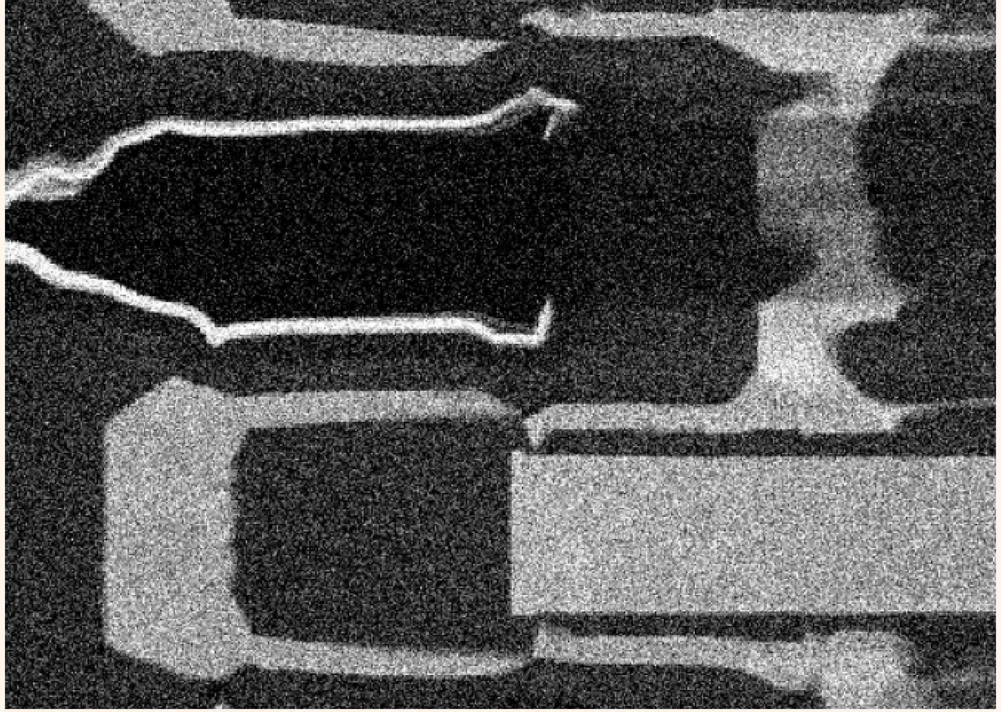


Image Restoration



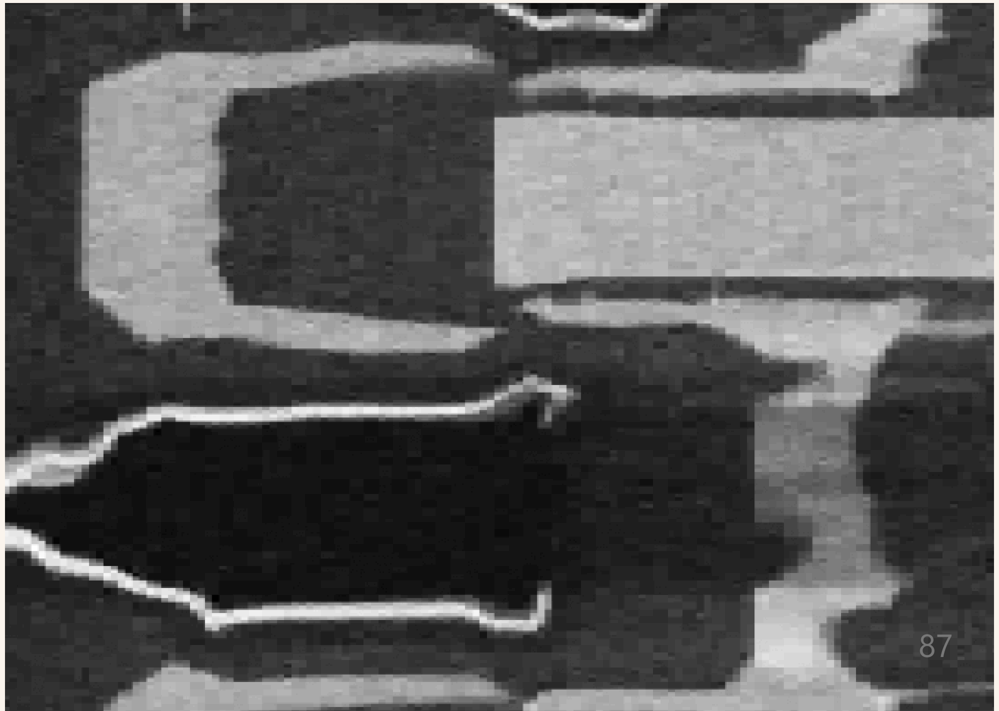


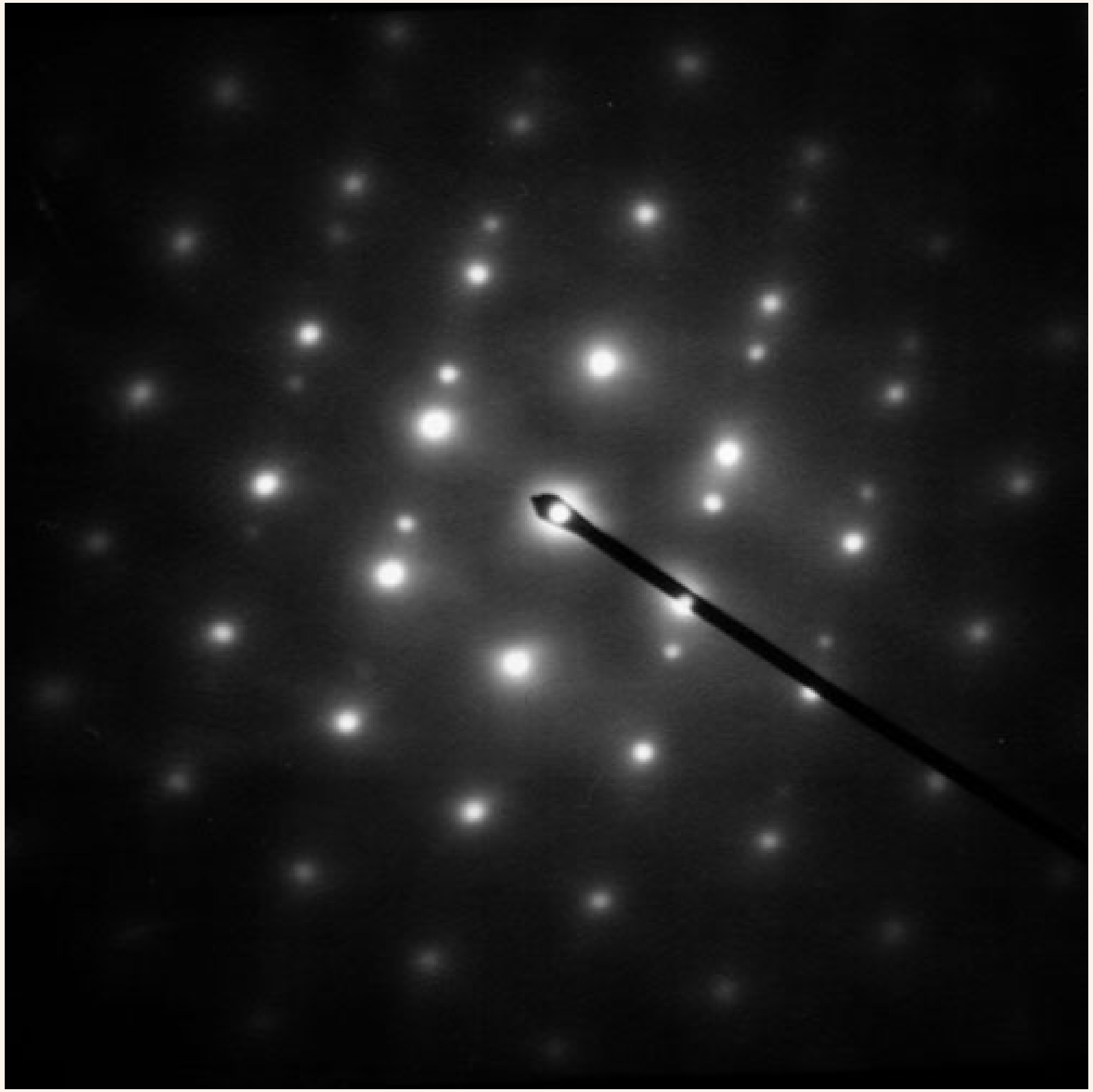
Image: Wikipedia, PD

Image restoration

Image restoration refers to techniques to **remove artefacts or noise** from an image while preserving the content.

> Filtering is a type of image restoration, but is simpler. Restoration frequently estimates how an image would have been without its artefacts.

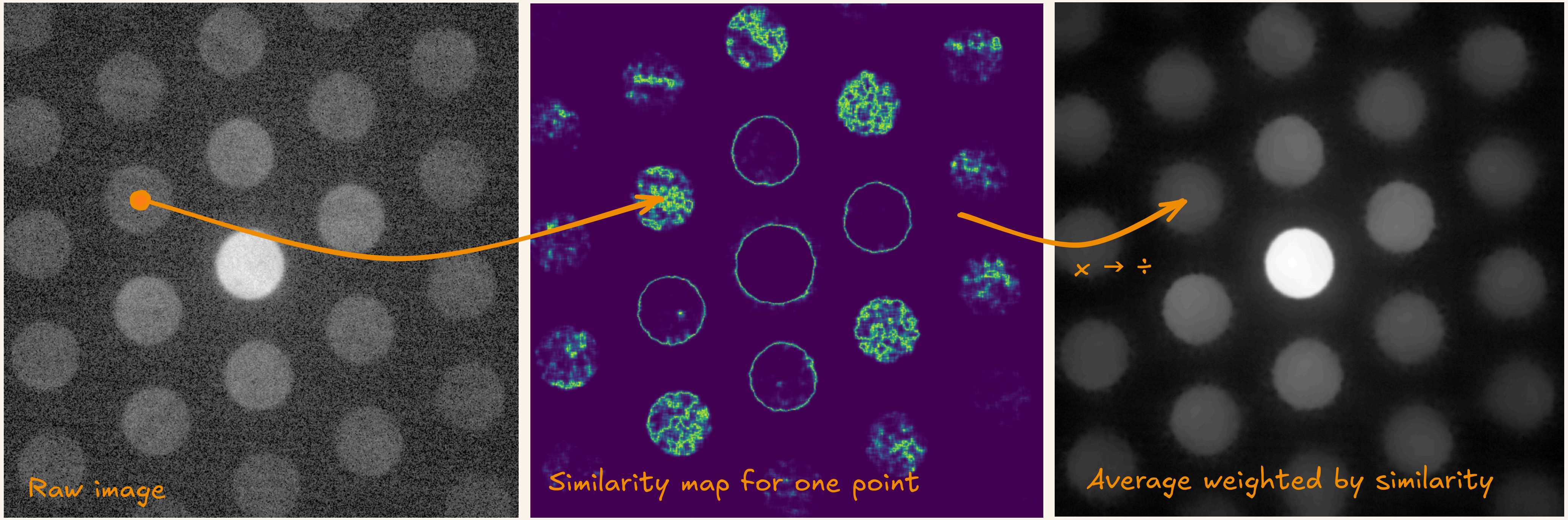
In low-dose, low signal-to-noise data, **denoising** is of particular interest.



Python: `skimage.restoration.denoise_nl_means`

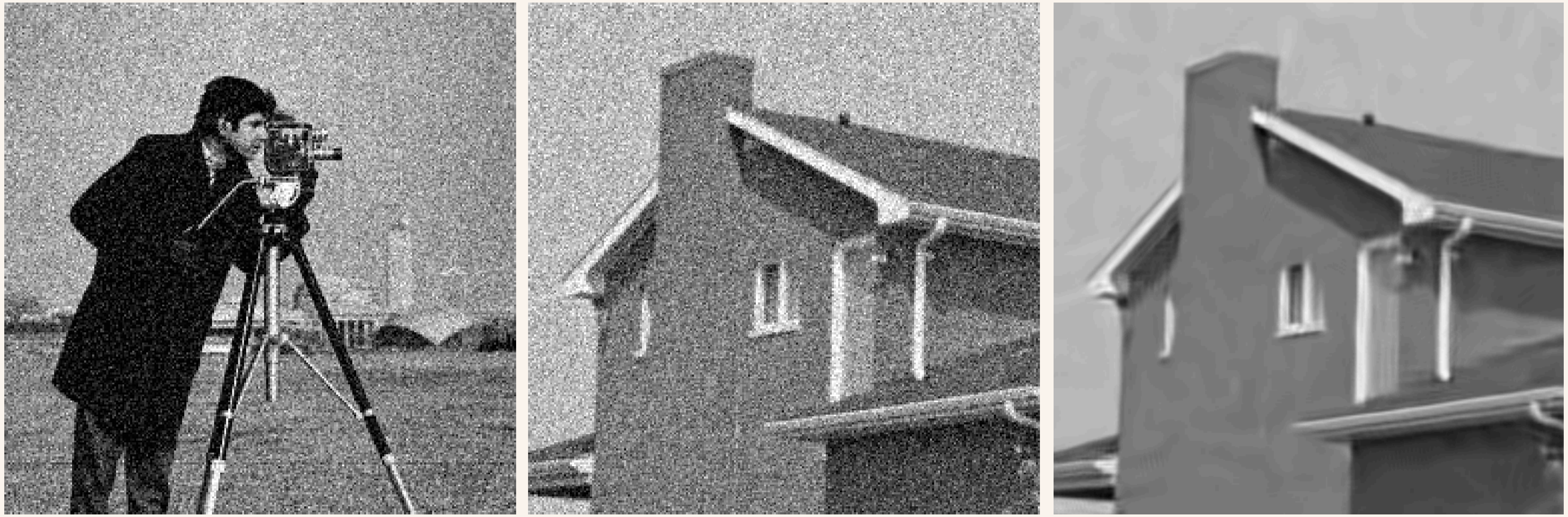
Denoising: Non-Local means

Rather than a simple average of local patches around each pixel, instead average all pixels in the image **weighted by their similarity** to the pixel being denoised.



Denoising: Block-Matching 3D (BM3D)

The BM3D algorithm improves non-local means by **grouping similar image patches** and filtering them together. Going beyond a simple weighted average greatly improves edge and texture preservation. Extensions exist for denoising **time-series** and **hyperspectral** data.



Dabov et al. (2006)

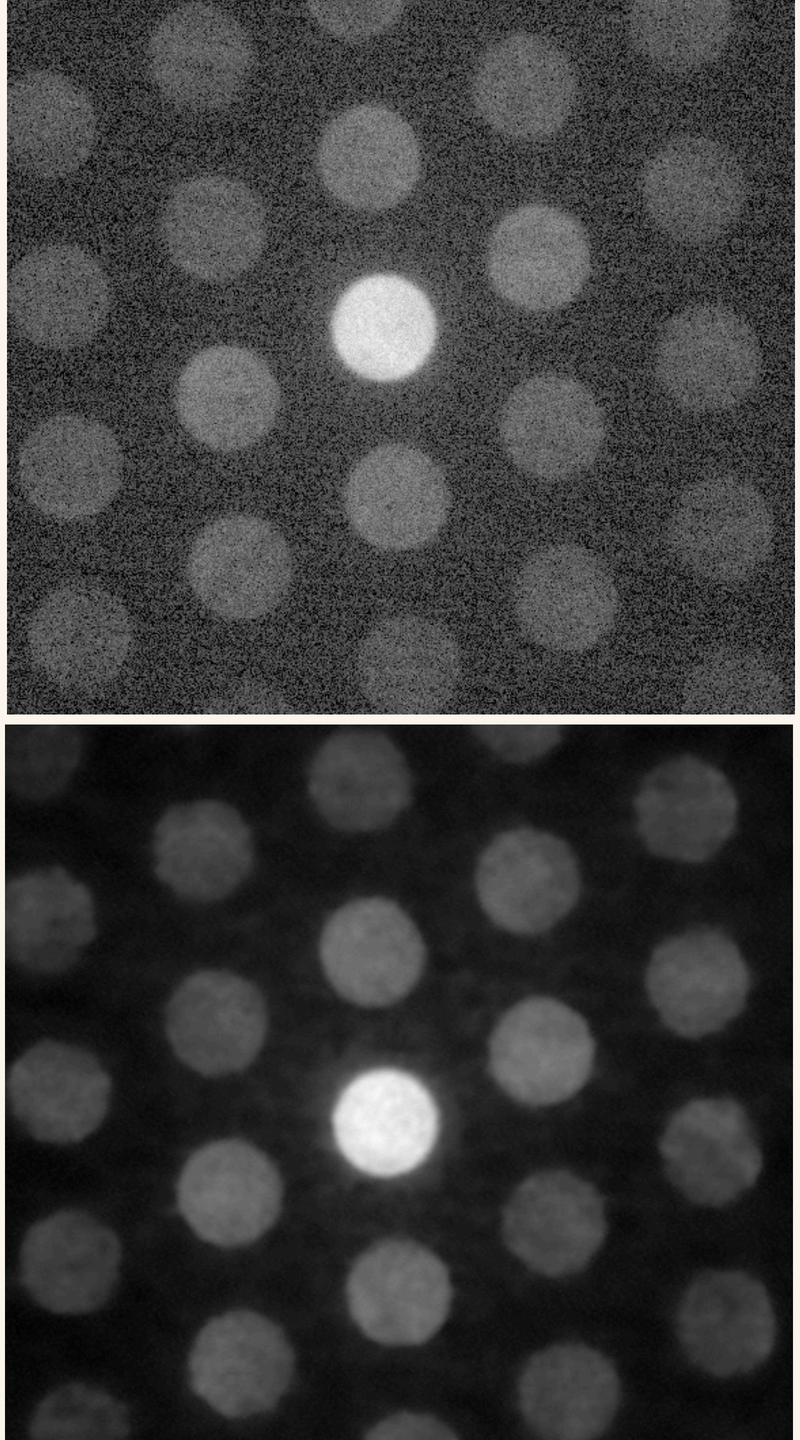
Python: `careamics`

Deep-learning for denoising

Denoising is a problem which is well-suited to *unsupervised* deep learning, because noise has simple statistics compared to image content.

A well-known architecture are the **Noise2-models**, e.g. Noise2Noise, which can efficiently denoise images without clean data to train from.

- These models are available as command-line tools, no programming required:



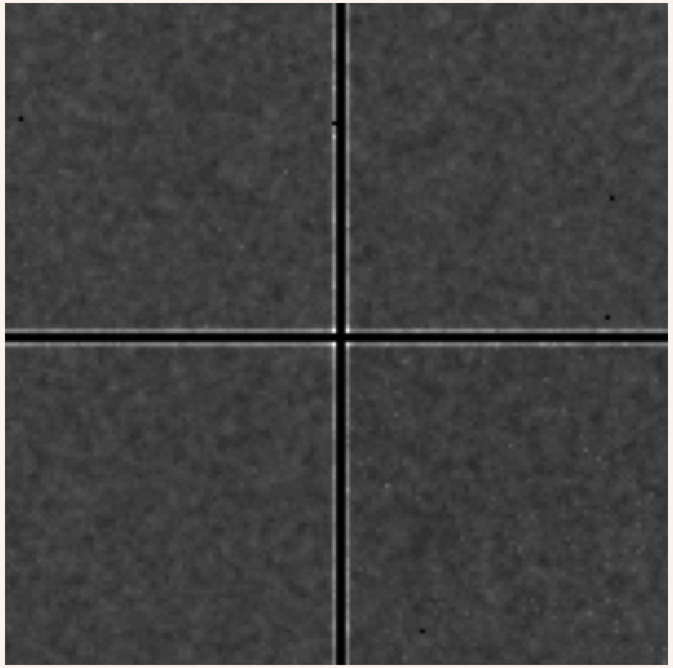
Inpainting

Inpainting replaces corrupted or **missing data** with a best-estimate. Some examples are to infill:

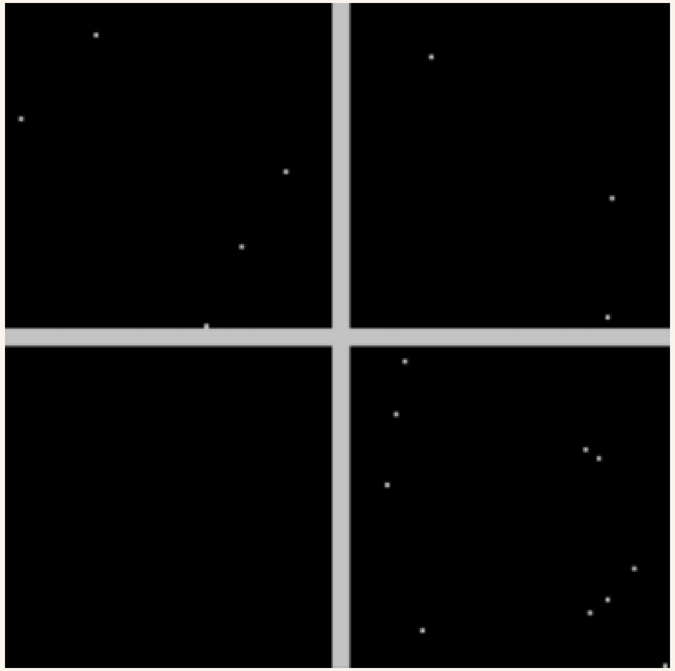
- dead pixels
- image area covered by a beamstopper
- a sensor bonding gap.

Inpainting - Interpolation

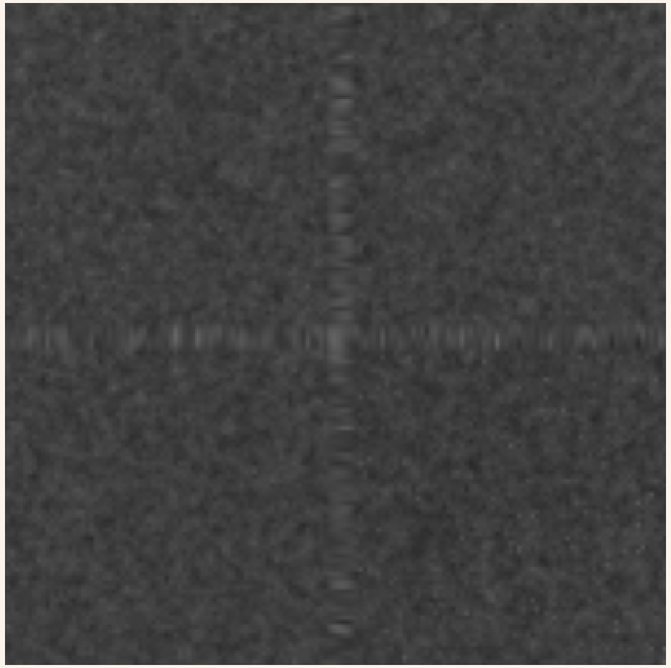
Simple *interpolation* is a good approach for small defects such as dead pixels.



Raw



Invalid mask



Interpolated

Deep Learning Inpainting

Inpainting is a very active field in deep learning, notably for natural images (e.g. background modification on smartphones).

An example is Large Mask Inpainting - LaMa (Suvorov et al., 2022).

Take care with scientific images as common models are not trained on these domains, and the "invented" data are likely misleading!

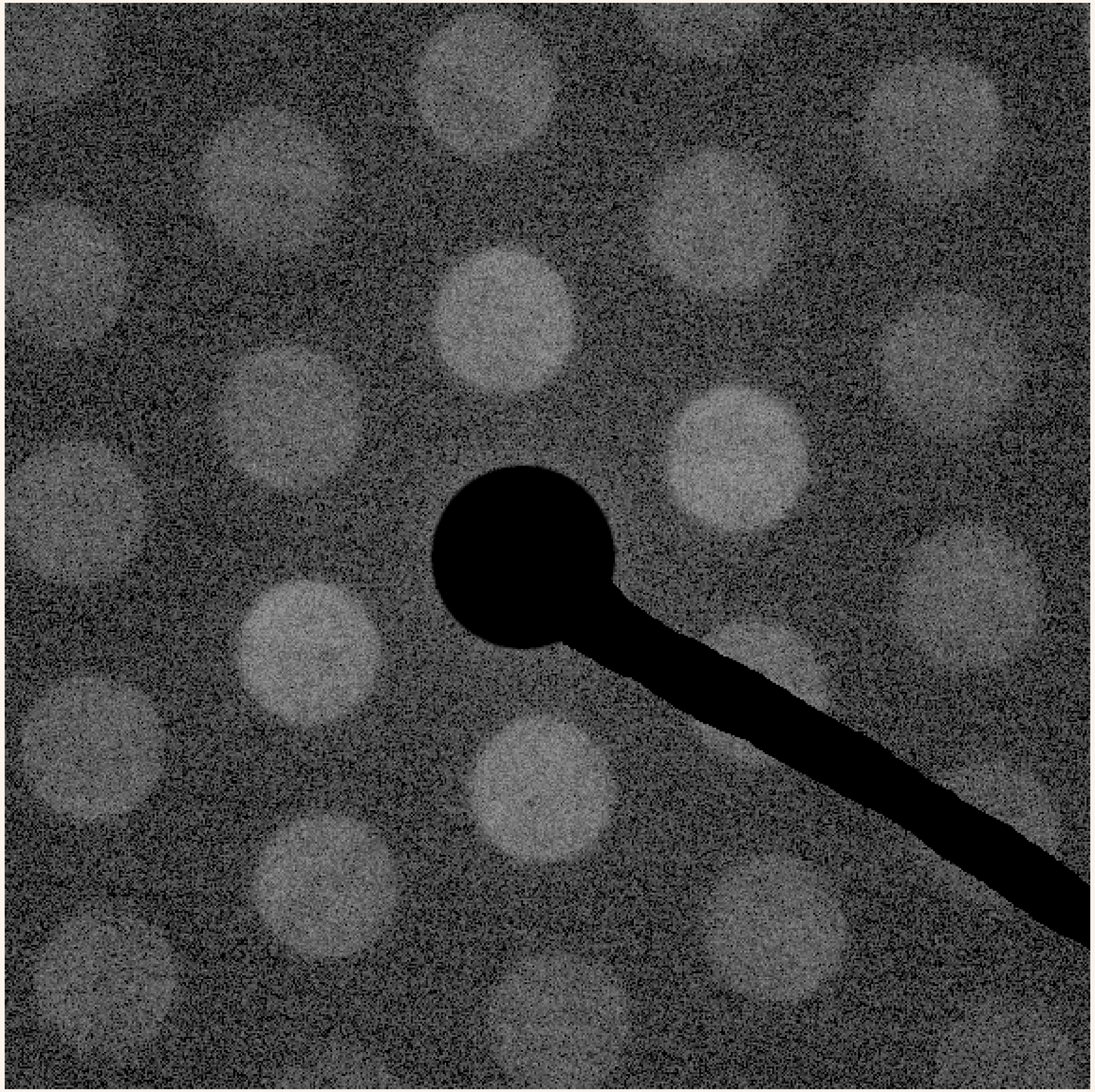
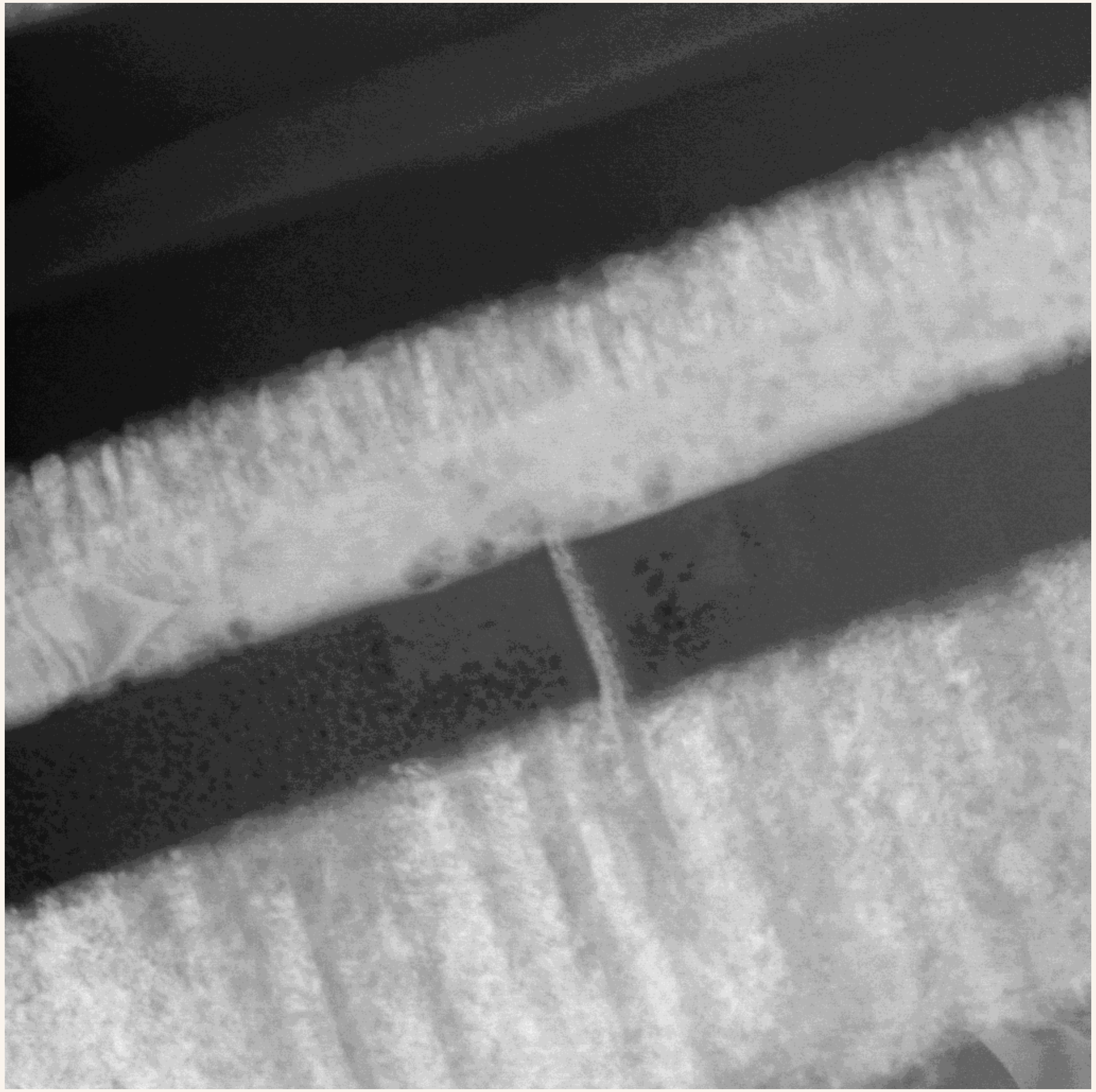


Image: CEA + Tescan

Pattern matching and image alignment



Pattern matching and image alignment

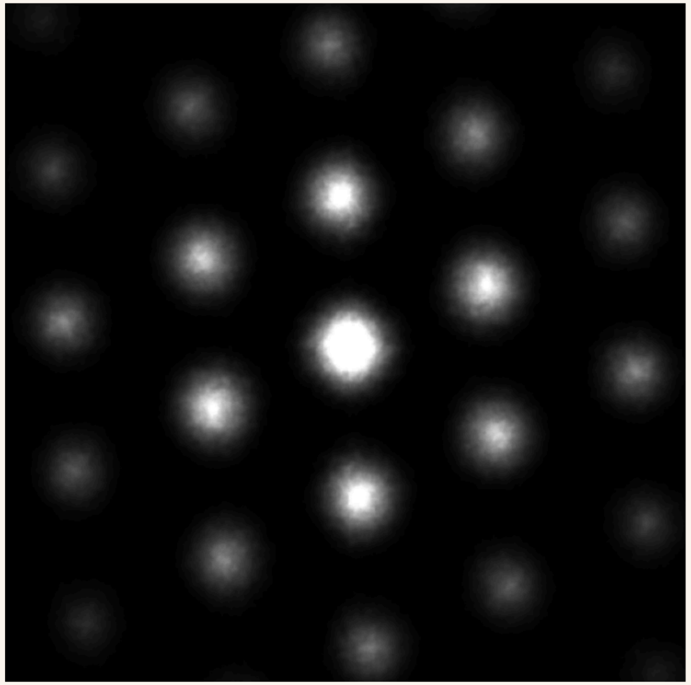
A common need in microscopy is to locate some image feature: an edge, a spot a corner - in order to measure somthing about it. This is an application of **pattern matching**.

A related problem is **image alignment**, where two-or-more images are separated by acquisition drift or change of scale, but we would like to compare the data from both images on the same grid or plot, requiring us to transform one image into the coordinate system of the other(s). Image alignment is also often referred to as image **registration**.

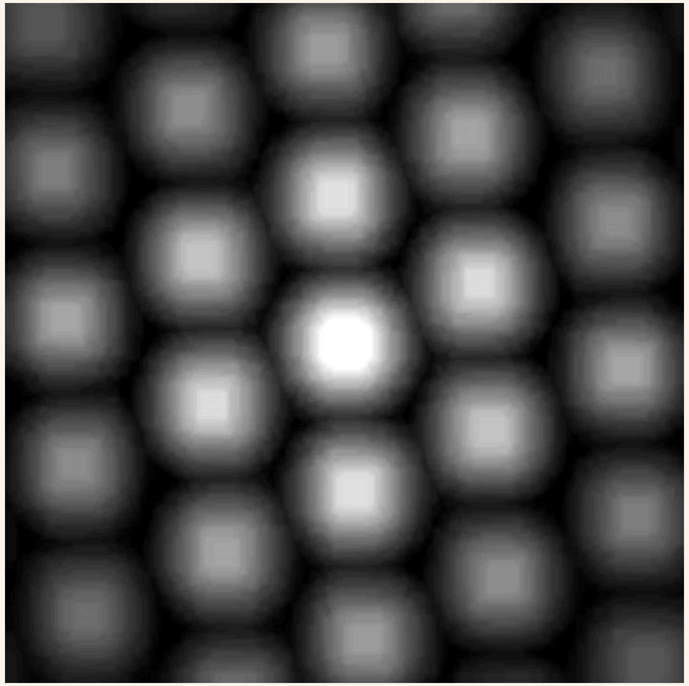
Python: `skimage.feature.peak_local_max`

Peak-finding

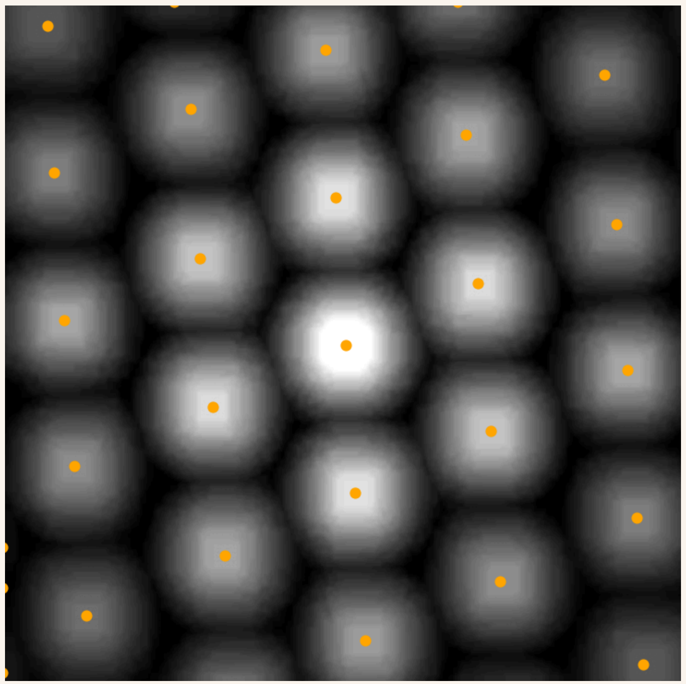
When the feature to detect is a local minimum or maximum in the intensity image, we can use *peak-finding* to locate it. A simple algorithm uses a *maximum filter*:



Raw data



Maximum filter 20x20

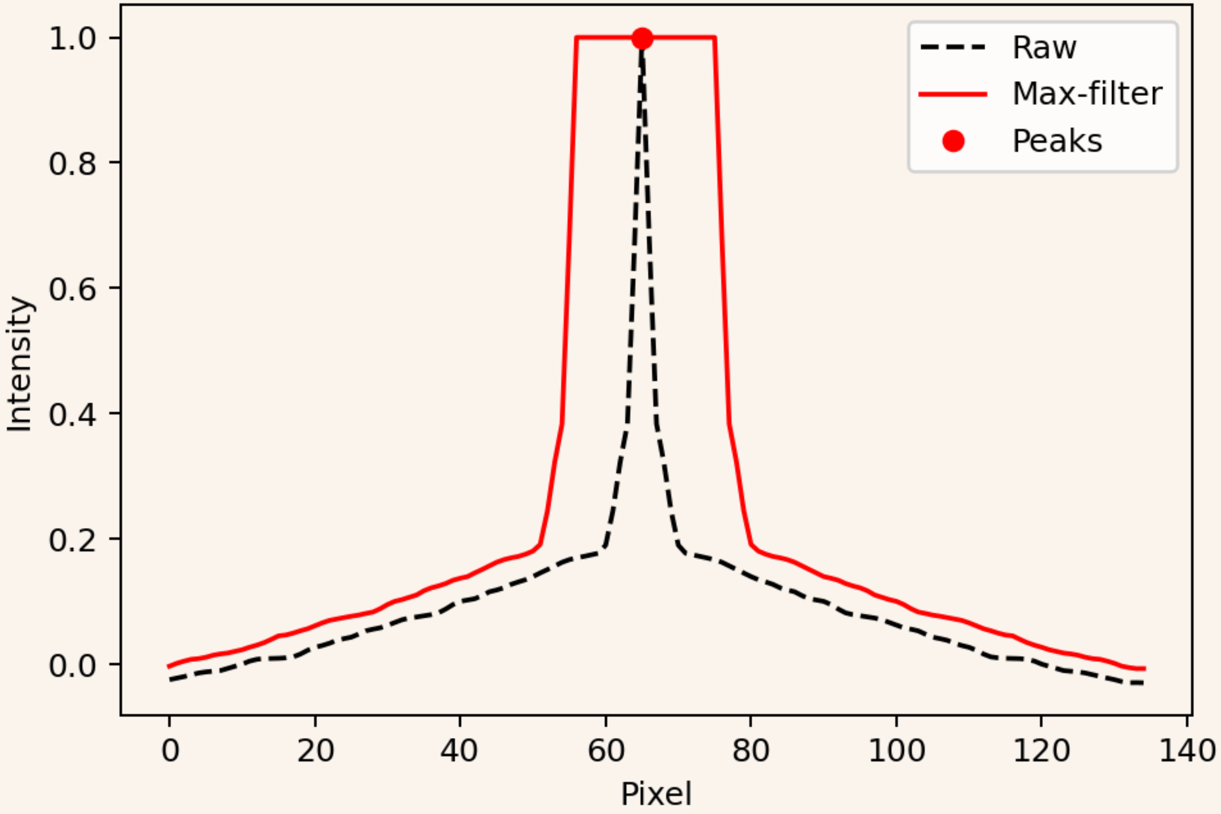


Filtered == Raw

Peak-finding in 1D, example

In practice with noisy data it is also necessary to:

- optimise the maximum filter window
- sort the peaks by value and perform a cut
- filter any peaks which are too similar

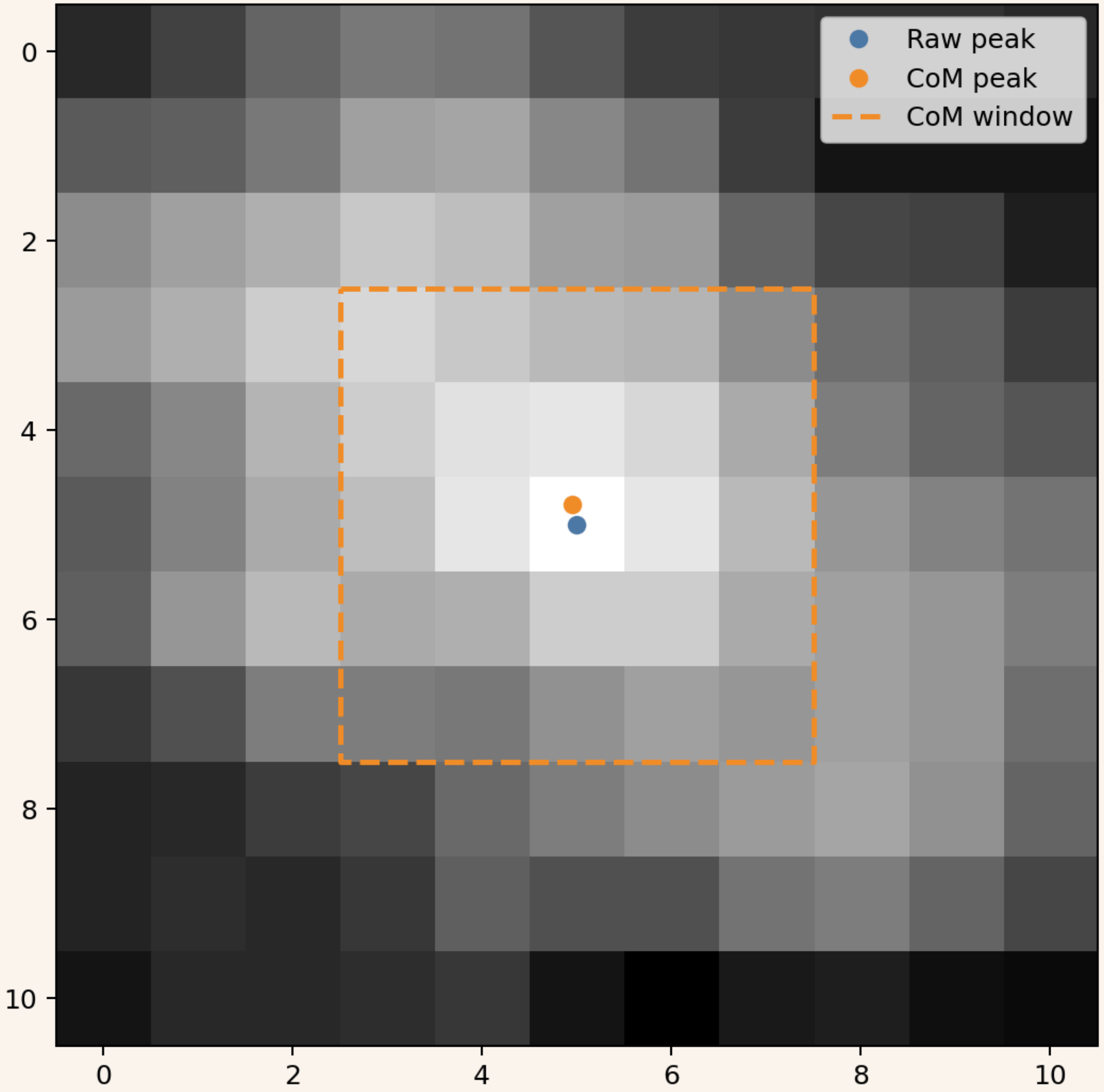


Python: scipy.ndimage.center_of_mass

Subpixel refinement with Centre-of-Mass

The simple peak finding algorithm only returns maxima at integer pixel coordinates.

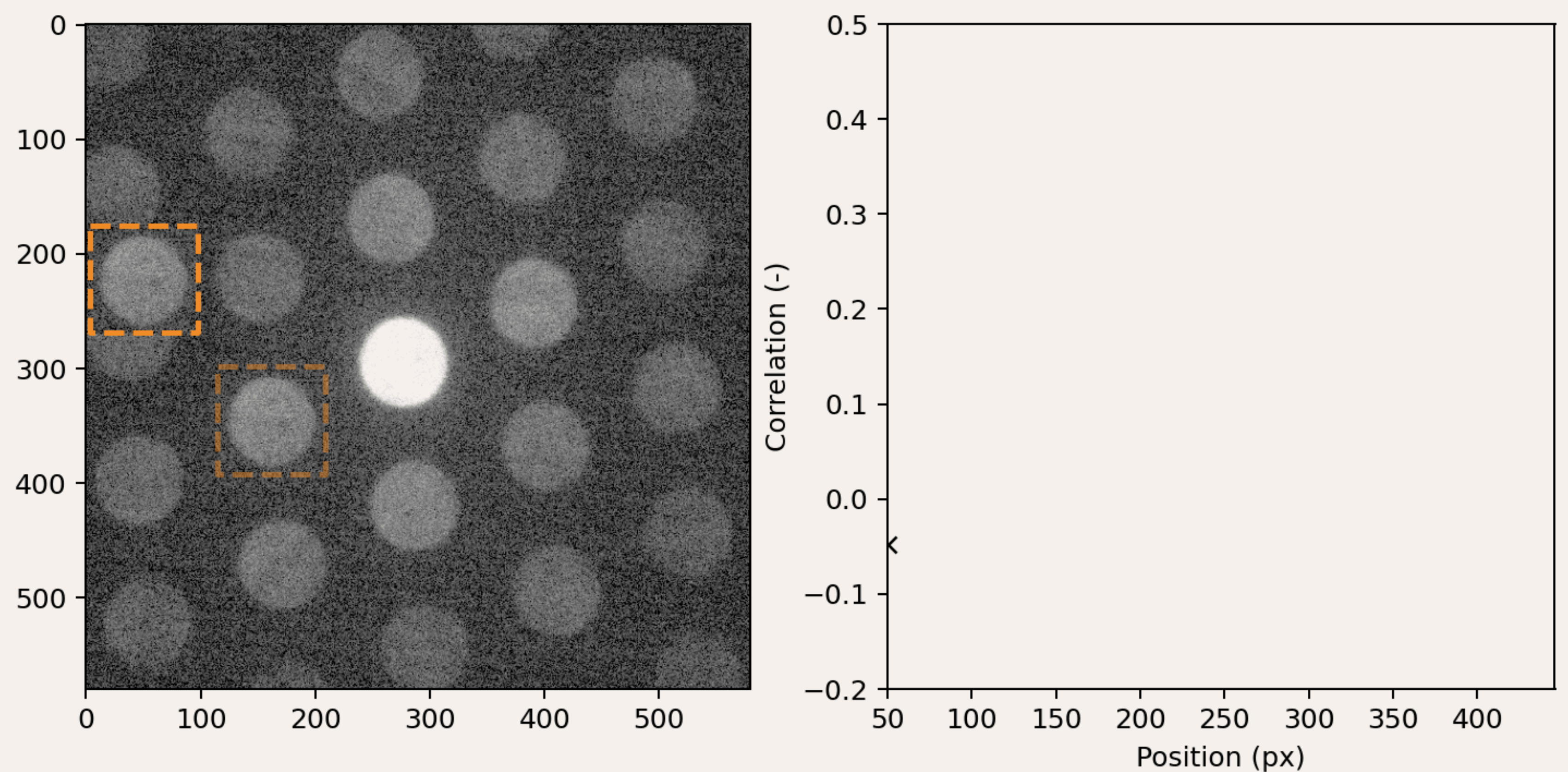
We can acheive greater precision by performing intensity-weighted local **centre-of-mass** around each peak.



Python: `skimage.feature.match_template`

Template matching

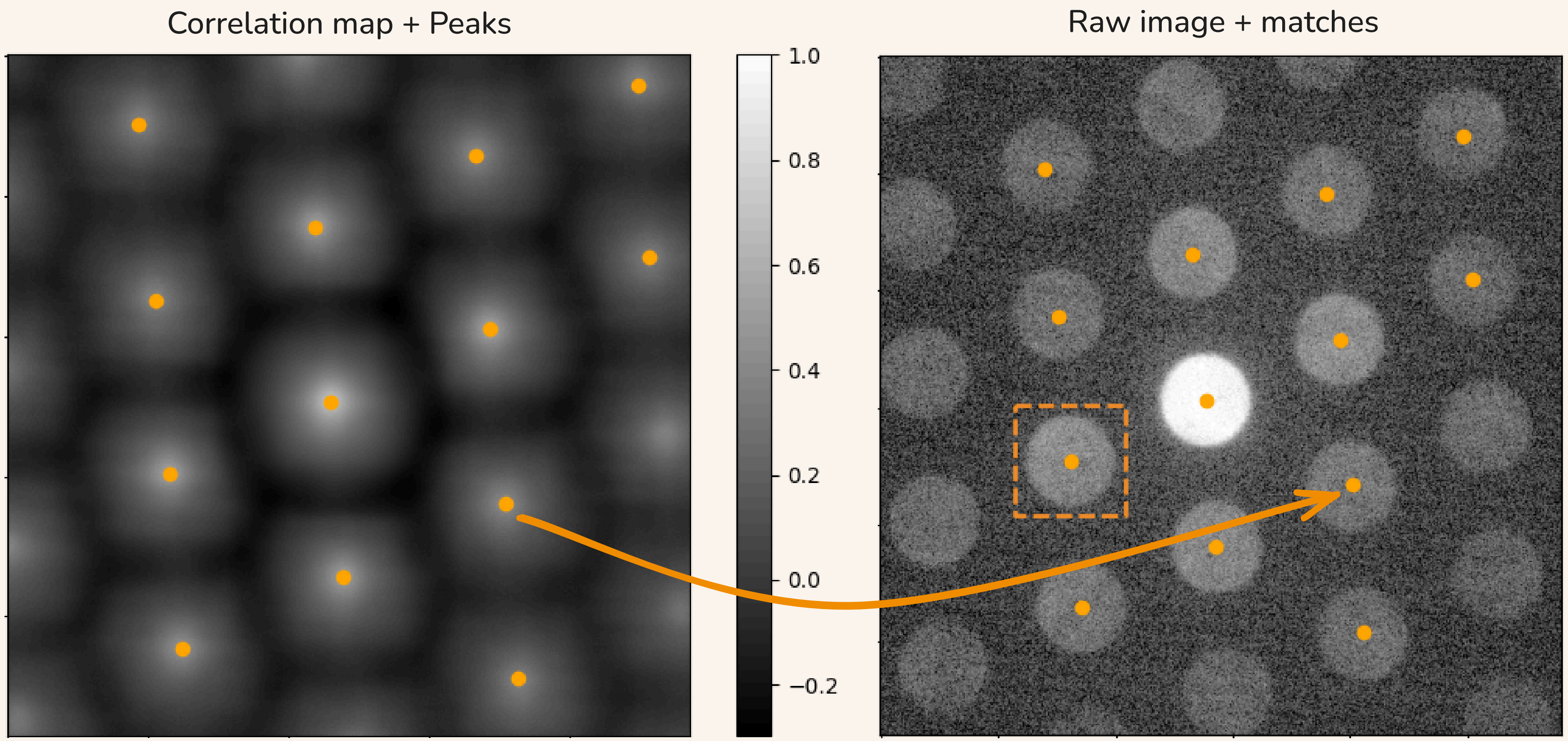
When the feature to find is not a local maximum, or we need to detect a particular pattern in the intensity rather than a point, one approach is *template matching*, based on the **correlation** between our target image and the *template* or pattern that we want to find.



Python: `skimage.feature.peak_local_max`

Template matching: locate matches

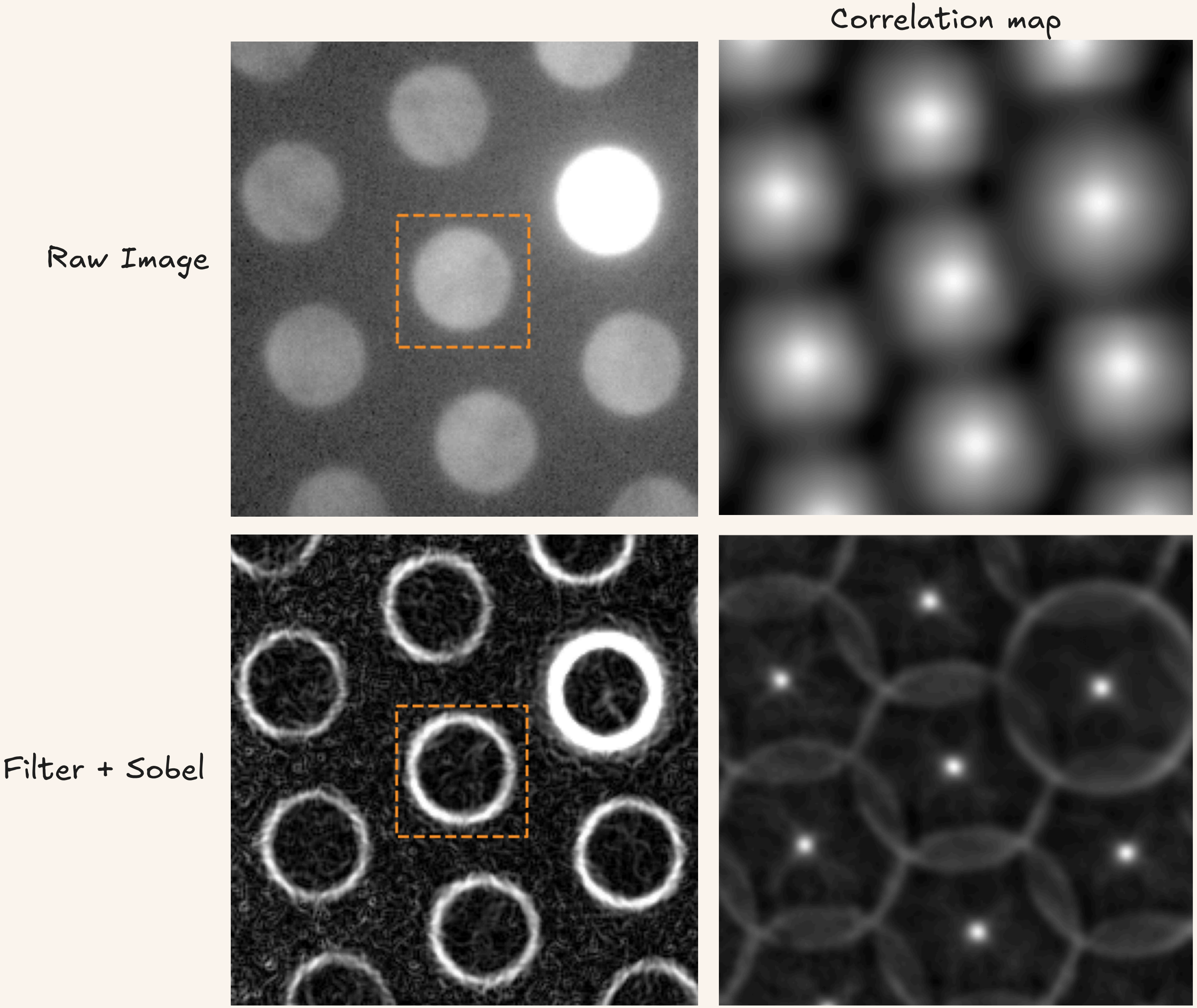
Checking all template positions generates a 2D correlation map with peaks at all "good" matches. Then use a peak-finding algorithm (with refinement) to locate the best positions.



Template matching: filtering

Template matching is **very sensitive** to both template choice and image quality.

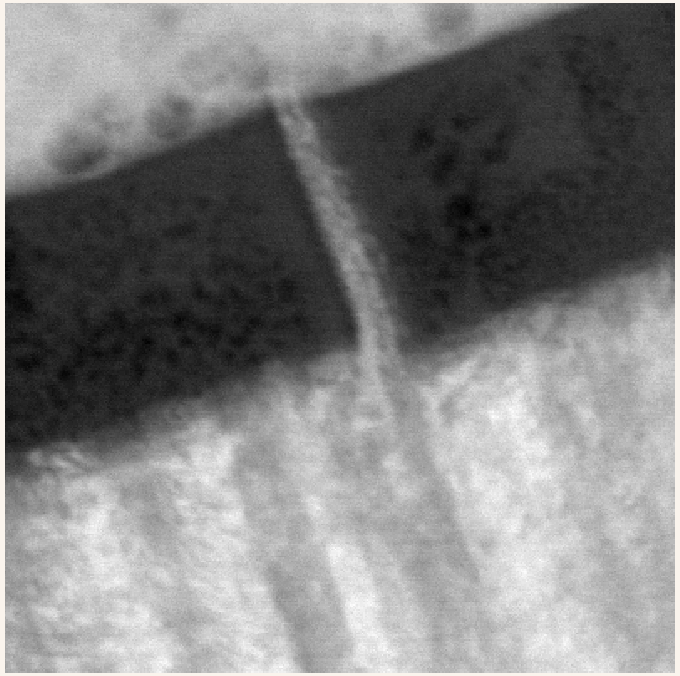
Often useful to **filter** the target image to acheive sharper peaks in the correlation image, leading to more precise results.



Python: `skimage.registration.phase_cross_correlation`

Image alignment

If we want to align whole images in translation we can compute the **cross-correlation** between them.



The maximum in the correlation map can be found using peak-finding.

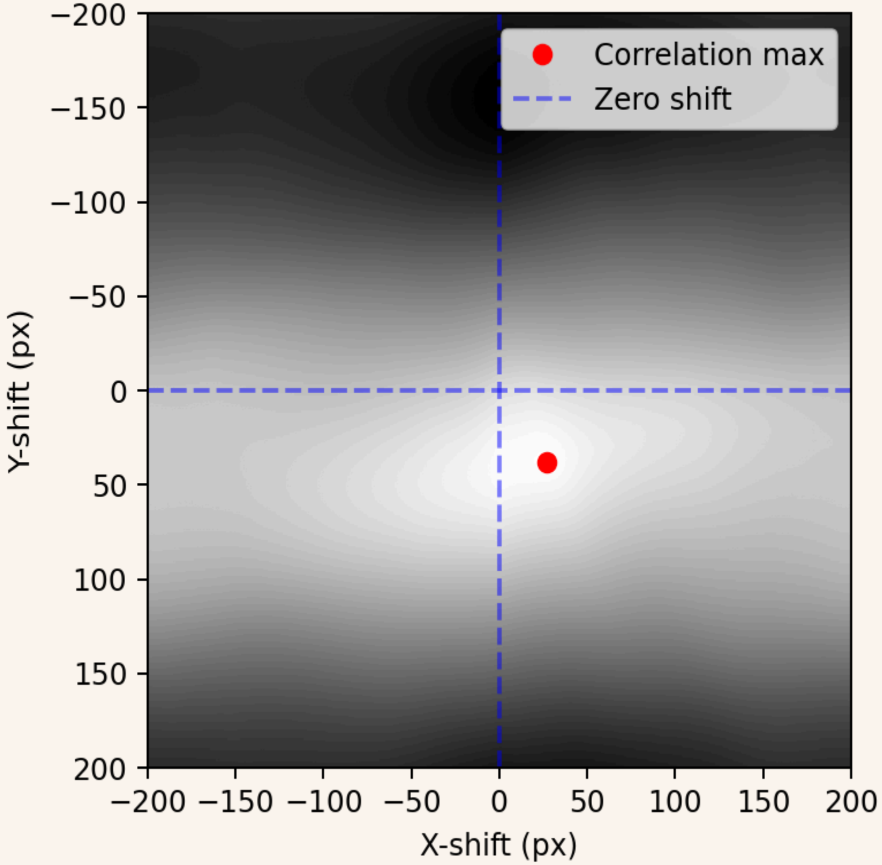


Image alignment, correlation-based

In practice whole-image correlation-based alignment is not very robust, and will fail for changes of scale or image rotation.

- In some cases, aligning on just a subset of the image simplifies the correlation map
- Downscaling the images can improve results, as noise is minimised and the alignment uses only "large" features of the image
    - Multi-scale or "pyramid" alignment first aligns at a large scale, then progressively increases resolution while constraining the maximum shift.
- Image filtering and pre-processing (e.g. normalisation) can also hugely affect the reliability

Python: `scipy.ndimage.fourier_shift`

Fourier image shifting

A useful property of a Fourier transform:

> shifting a signal in real space is equivalent to multiplication by a complex exponential in the transformed space, i.e. a phase shift

This can be used to shift an image even by **sub-pixel** distances.

X-Shift

Python: `skimage.feature.SIFT`, `pystackreg`

Image alignment, automatic point-based

An alternative approach is to fit a geometric transform between the two images based on **corresponding points** visible in both.

These points can be estimated automatically using a feature extractor like `SIFT` (Scale Invariant Feature Transform) or chosen manually.

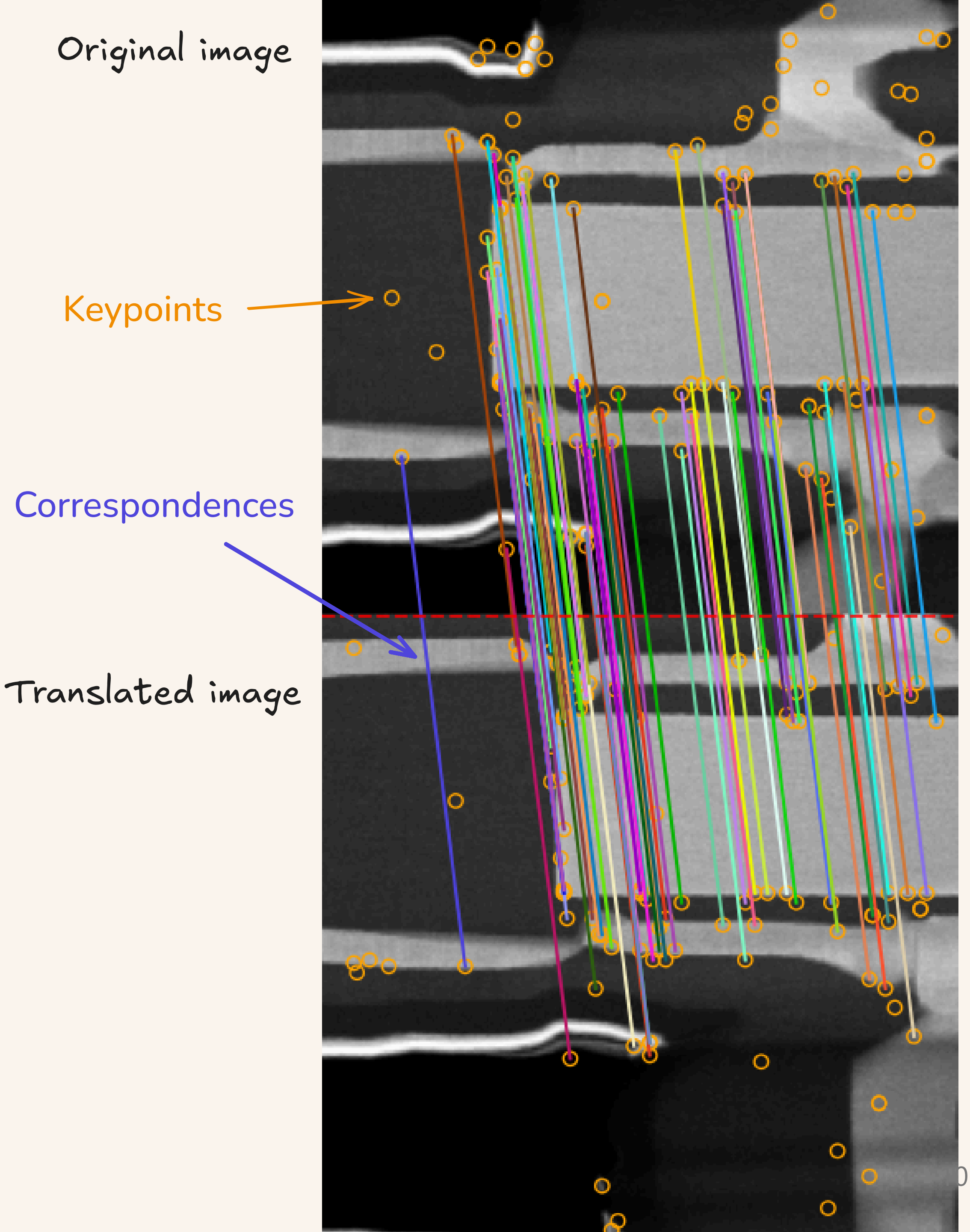


Image alignment, manual point-based

Run

Transformation type

Affine

Overlay alpha: **0**

Clear points

..

...

Summary

Summary

Digital images underpin almost all of modern microscopy, and influence how data are acquired, interpreted and perceived.

This presentation was a very rapid overview of a lot of topics, and should be seen as **a starting point** for what could be done with your data.

Please reach out if you have questions or ideas at GitHub: @matbryan52 or at matthew.bryan@cea.fr

About the slides

These slides are written in Marp using Markdown.

The interactive components are based on Panel and Bokeh, which can be used both in standalone web-pages and within Jupyter to put interactivity in-line with your analysis.

Diagrams were drawn with Excalidraw.

The source, figures and code for everything is on Github: matbryan52/microscopy-images-qem.

Thank you for listening