Deblurring images, in Python

A companion to Hansen, Nagy & O'Leary, ending on real telescope data

Hugues Talbot

November 2026

Table of contents

Preface 4
Contents . . . 4
Running the code . . . 5
Sources and credits . . . 5

I The book, chapter by chapter 6

1 The image deblurring problem 7
1.1 A sharp image, a PSF, a blurred image . . . 7
1.2 Blurring is linear . . . 8
1.3 Seeing the matrix once: a one-dimensional signal . . . 9
1.4 The naive solution . . . 10
1.5 Why: look at the spectrum . . . 10
1.6 A first cure: do not divide where K is small . . . 11
1.7 The real thing: one exposure of our telescope . . . 13
Your turn . . . 14
Where to read more . . . 14

2 Images as arrays 15
2.1 An image is an array . . . 15
2.2 What you see is not what you have . . . 16
2.3 Data types: integers lie . . . 18
2.4 Arithmetic on images . . . 19
2.5 Colour images . . . 21
Your turn . . . 21
Where to read more . . . 22

3 The blurring function 23
3.1 The PSF is the image of a point . . . 23
3.2 The matrix behind the convolution . . . 24
3.3 Where PSFs come from: three models . . . 25
3.4 Measuring the PSF on the data . . . 26
3.5 Noise . . . 27
3.6 Boundary conditions . . . 29
Your turn . . . 30
Where to read more . . . 30

4 Structured computations 31
4.1 One dimension: three boundary conditions, three structures . . . 31
4.2 Periodic boundaries: the FFT diagonalises A . . . 32
4.3 Reflexive boundaries: the DCT diagonalises A . . . 33
4.4 Separable blurs: Kronecker products . . . 34
4.5 Summary: which tool for which problem . . . 35
4.6 Realistic test data, and what to do at the edges . . . 36
Your turn . . . 38
Where to read more . . . 38

5 The SVD and spectral analysis 39
5.1 The SVD of a blur . . . 39
5.2 The naive solution, channel by channel . . . 40
5.3 Spectral filtering . . . 41

5.4 Two dimensions: the patterns are images . . . 42
5.5 Real data: what stacking buys . . . 44
Your turn . . . 45
Where to read more . . . 46

6 Regularisation by spectral filtering **47**
6.1 Two filters . . . 47
6.2 Tikhonov regularisation is a least-squares problem . . . 48
6.3 Deblurring an image with both filters . . . 49
6.4 Two errors that pull in opposite directions . . . 50
6.5 Choosing α without knowing the truth . . . 51
6.6 Real data, and a fourth rule . . . 54
Your turn . . . 55
Where to read more . . . 56

7 Colour images and smoothing norms **57**
7.1 Colour: blurring across channels . . . 57
7.2 Our case: no mixing, but three different PSFs . . . 58
7.3 Tikhonov revisited: penalise the derivative . . . 59
7.4 Other norms: keeping the edges . . . 61
Your turn . . . 62
Where to read more . . . 63

II Towards the research project **64**

8 From one frame to many: proper coaddition **65**
8.1 Forty-five exposures, forty-five PSFs . . . 65
8.2 The model and its solution . . . 67
8.3 A controlled experiment . . . 68
8.4 Where the gain comes from . . . 69
8.5 The real exposures . . . 71
8.6 The weak point: every exposure is trusted . . . 73
Your turn . . . 74
Where to read more . . . 74

9 Iterative methods: when there is no closed form **75**
9.1 Operators, not matrices . . . 75
9.2 Gradient descent . . . 76
9.3 Conjugate gradient . . . 77
9.4 Majorise-minimise . . . 78
9.5 Robust proper coaddition . . . 80
9.6 Real data: the robust loss as a diagnostic . . . 81
9.7 Where the project starts . . . 82
Your turn . . . 83
Where to read more . . . 83

References **84**

Appendices **85**

A The helper module `hnotools.py` **85**

Preface

These nine chapters teach how to restore a blurred image, from the first principles to the edge of a research project. They accompany a textbook,

> P. C. Hansen, J. G. Nagy and D. P. O'Leary, *Deblurring Images: Matrices, Spectra, and Filtering*, SIAM, 2006,

which teaches with MATLAB. Here the same mathematics is taught in Python, in our own words, and every chapter ends on real data: narrowband images of the Eagle Nebula (M16) taken with a 14 cm refractor, with Hubble Space Telescope images of the same field as a reference.

This is a companion, not a substitute. Read the chapter of the book, then work through the corresponding chapter here.

Contents

The book, chapter by chapter

1. The image deblurring problem: why the obvious solution fails.
2. Images as arrays: NumPy in place of MATLAB; display is not data.
3. The blurring function: the PSF, measured on a star; noise; boundaries.
4. Structured computations: never building the matrix.
5. The SVD and spectral analysis: the Picard plot.
6. Regularisation by spectral filtering: Tikhonov, and how much of it.
7. Colour images and smoothing norms: beyond the basic method.

Towards the research project

8. From one frame to many: proper coaddition.
9. Iterative methods: when there is no closed form; robust coaddition.

followed by the references and a listing of the helper module.

Chapters 1 to 7 follow the seven chapters of Hansen et al. (2006). Chapter 8 is not in that book: it extends the method from one image to many, which is the situation of every astronomer, and introduces *proper coaddition* (Zackay and Ofek 2017). Chapter 9 follows chapter 4 of Talbot (2024) and provides the iterative methods needed when no formula exists. Its last section lists the open questions of the research project that these notes prepare for.

The chapters were written as Jupyter notebooks, and the text calls them "notebook 1", "notebook 2" and so on: notebook n is chapter n.

Each chapter has the same shape:

1. **What you will take away**, in a few lines;
2. the ideas, each stated in plain words before any formula, each checked numerically as soon as it is stated;
3. **real data** from our telescope;
4. **Your turn**: exercises. Predict the answer before you run the cell;
5. **Where to read more**.

Do the chapters in order. Each uses what the previous ones built.

Running the code

Every chapter is a Jupyter notebook; this book is those notebooks, rendered with their outputs. To run and modify them:

```
cd notebooks/hno
python3 -m venv .venv
.venv/bin/pip install -r requirements.txt
.venv/bin/jupyter lab
```

The few functions shared by the chapters live in `hnotools.py`, listed in the appendix. It is short on purpose: read it. The data are in `data/m16_sample.npz`, about 4.5 MB, described at the top of `make_sample.py`.

Sources and credits

- The progression of chapters 1 to 7 is that of Hansen, Nagy and O'Leary. The text, the code and the examples are ours; the book is cited for the full treatment of each topic.
- The one-dimensional signal of chapters 1, 5, 6 and 7, the variational view of Tikhonov regularisation, and the iterative methods of chapter 9 come from H. Talbot, *Lecture Notes on Optimisation: applications to computer vision, image and signal processing, and machine learning*, CentraleSupélec.
- Proper coaddition is due to B. Zackay and E. Ofek, *The Astrophysical Journal* 836, 188 (2017). The declared target resolution is due to P. Magain, F. Courbin and S. Sohy, *The Astrophysical Journal* 494, 472 (1998).
- The telescope data were acquired by H. Talbot. The Hubble images are from the Hubble Heritage project (NASA, ESA, STScI).

Part I

The book, chapter by chapter

1 The image deblurring problem

Companion notebook to Hansen, Nagy & O'Leary, ***Deblurring Images: Matrices, Spectra, and Filtering*** *(SIAM, 2006), chapter 1. Read the chapter first; this notebook lets you run the ideas in Python, and ends on our own telescope data. The one-dimensional example is taken from H. Talbot's lecture notes on optimisation.*

What you will take away

1. Blurring is a *linear* operation: $b = A\,x + e$.
2. For a spatially invariant blur with periodic boundaries, A is diagonal in the Fourier basis, so applying it and inverting it cost one FFT each.
3. The obvious solution $x = A^{-1}b$ is useless: it amplifies noise without bound. This is what "ill-posed" means in practice.
4. A usable solution must give up some frequencies. Choosing which ones is *regularisation*, the subject of the whole book.

Notation. The book writes x for the sharp image, b for the blurred one, e for the noise and A for the blurring matrix. Our project notes write $y = k * x + n$: same thing, with $b \leftrightarrow y$, $e \leftrightarrow n$, and A = "convolve with the PSF k". Capital letters (X, B, K) are Fourier transforms.

```
import numpy as np
from numpy.fft import fft2, ifft2
import matplotlib.pyplot as plt
import hnotools as hno

hno.style()
rng = np.random.default_rng(0)        # fixed seed: your figures match ours
```

1.1 A sharp image, a PSF, a blurred image

Our "true sky" x is a piece of the Pillars of Creation seen by the Hubble Space Telescope (oxygen filter). We blur it ourselves, so that for once we *know* the answer and can measure how well each method recovers it.

The **point-spread function** (PSF) is the image of a single point of light: what one star becomes after the atmosphere and the telescope. Blurring replaces every point of the sky by a copy of the PSF.

```
data = hno.load_sample()
x = data['hubble'].astype(float)                # the true image, 256 x 256
psf = hno.gaussian_psf(x.shape, fwhm=6.0)       # a Gaussian blur, 6 pixels wide

noise_level = 0.005                             # 0.5 % of the brightest pixel
e = noise_level * rng.standard_normal(x.shape)
b_exact = hno.blur(x, psf)                      # A x
b = b_exact + e                                 # what a camera records

hno.show([x, psf[108:149, 108:149], b_exact, b],
         ['true image $x$', 'PSF (zoom)', 'blurred $A x$', 'blurred + noise $b$']);
```

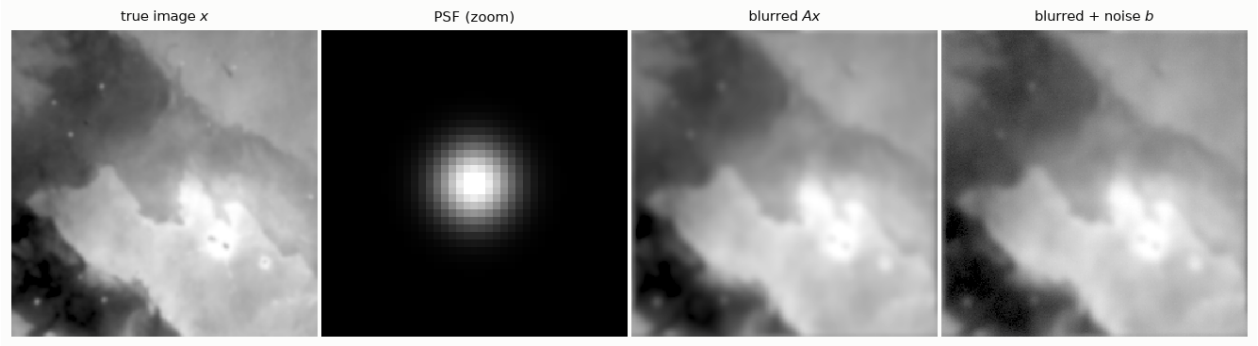


Figure 1.1: The experiment of this chapter: a true image, the PSF, the blurred image, and the blurred image with noise.

The noise is barely visible. Keep that in mind: it is about to ruin everything.

1.2 Blurring is linear

"Linear" means two things: blurring a sum gives the sum of the blurs, and blurring twice as bright an image gives twice as bright a result. Check it:

```
x1, x2 = x, np.roll(x, 40, axis=1)              # two different images
lhs = hno.blur(x1 + 2 * x2, psf)
rhs = hno.blur(x1, psf) + 2 * hno.blur(x2, psf)
print('largest difference:', np.abs(lhs - rhs).max())
```

```
largest difference: 1.7763568394002505e-15
```

Zero, up to rounding. So if we stack the pixels of an image into one long vector, blurring is a matrix product $b = Ax$.

How big is A? Our image has $256 \times 256 = 65\,536$ pixels, so A is $65\,536 \times 65\,536$:

```
n = x.size
print(f'A has {n**2:.2e} entries = {n**2 * 8 / 1e9:.0f} GB in double precision')
```

```
A has 4.29e+09 entries = 34 GB in double precision
```

We will never build A. We do not need to: when the PSF is the same everywhere in the image and the boundaries are periodic, the **convolution theorem** says that blurring is a plain multiplication in Fourier space,

$$B(\nu) = K(\nu)\, X(\nu),$$

where K, the Fourier transform of the PSF, is called the *optical transfer function*. In other words A is **diagonal in the Fourier basis**, and the values of K are its eigenvalues. This is all `hno.blur` does: open `hnotools.py` and read its three lines.

Chapters 3 and 4 of the book explain when this is exactly true and what to do at the image boundaries. For now, accept periodic boundaries.

1.3 Seeing the matrix once: a one-dimensional signal

Before inverting anything, let us look at a blurring matrix. For an image it is too big; for a one-dimensional signal it is not. The example below comes from H. Talbot's *Lecture Notes on Optimisation* (chapter 2): a signal of 250 samples with steps and slow waves, blurred by a Gaussian kernel 25 samples long, with a tiny amount of noise.

```
N = 250
s = np.linspace(0, 3 * np.pi, N)
signal = 1.0 * (np.cos(2 * s) > 0) + 2 * np.sin(s / 4) + 0.1 * np.cos(3 * s + 1)

half = 12                                          # the kernel covers 2*12 + 1 = 25 samples
offsets = np.arange(-half, half + 1)
kernel = np.exp(-offsets**2 / (2 * 4.0**2))
kernel /= kernel.sum()

# Row i of A holds the kernel centred on sample i: one shifted copy per row.
A1 = sum(w * np.eye(N, k=o) for o, w in zip(offsets, kernel))
observed = A1 @ signal + 2e-4 * rng.standard_normal(N)
naive = np.linalg.solve(A1, observed)

fig, ax = plt.subplots(1, 3, figsize=(11, 3.0), gridspec_kw={'width_ratios': [1, 1.5, 1.5]})
ax[0].imshow(A1[:60, :60], cmap='gray_r'); ax[0].grid(False)
ax[0].set_title('top-left corner of $A$', loc='left'); ax[0].set_xticks([]); ax[0].set_yticks([])
ax[1].plot(s, signal, color=hno.INK2, lw=1.2, label='true signal')
ax[1].plot(s, observed, color=hno.BLUE, label='blurred + noise')
ax[1].set_title('the data', loc='left'); ax[1].legend(loc='lower right', fontsize=8)
ax[2].plot(s, naive, color=hno.ORANGE, lw=0.8)
ax[2].set_title('naive solution of $A x = b$', loc='left')
fig.tight_layout()
print(f'condition number of A: {np.linalg.cond(A1):.1e}')
print(f'largest value of the true signal: {np.abs(signal).max():.1f}   of the naive solution:
 ↪  {np.abs(naive).max():.1e}')
```

```
condition number of A: 1.1e+08
largest value of the true signal: 3.1   of the naive solution: 2.9e+03
```

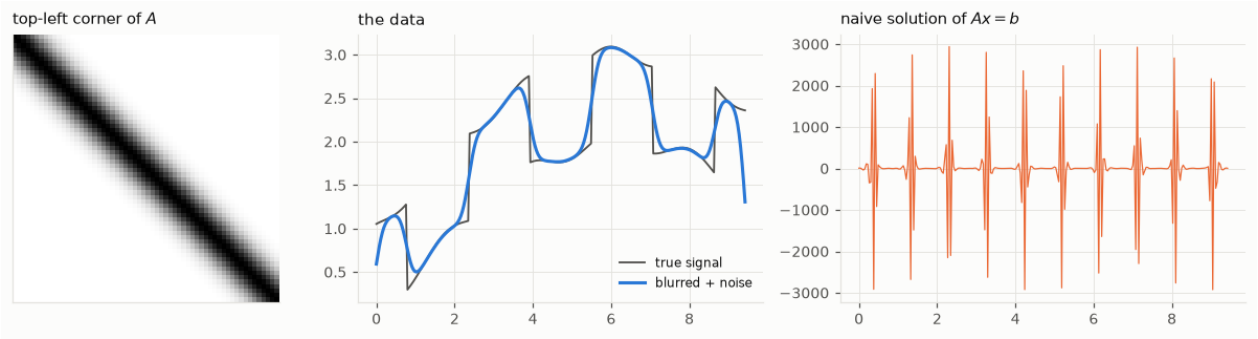


Figure 1.2: A one-dimensional blur: the band structure of its matrix, the data, and the naive solution of the linear system.

Three things to notice.

- A is a **band**: each row is the same kernel, shifted by one sample. All the structure of a blurring matrix is in that sentence, and chapter 4 of the book exploits it.
- The **condition number** of A measures how much solving $Ax = b$ can amplify a relative error in b. Here it is 10^8: an error of one part in ten thousand in the data may become an error ten thousand times larger than the signal.

- And it does. The noise we added is invisible on the middle plot, about one part in ten thousand. The naive solution is wrong by three orders of magnitude.

The same thing will now happen with images. We cannot build their A, but the convolution theorem gives us its eigenvalues for free.

1.4 The naive solution

If $B = KX$, then surely $X = B/K$. First with **no noise at all**:

```
K = hno.psf2otf(psf)
x_naive_exact = np.real(ifft2(fft2(b_exact) / K))
print('relative error, noise-free data:', hno.rel_error(x_naive_exact, x))
```

```
relative error, noise-free data: 20.753292470784793
```

A relative error of 1 would mean "as wrong as a black image". We are far beyond that, **without any noise**: the only errors in `b_exact` are rounding errors in the sixteenth digit. Now with the camera noise:

```
x_naive = np.real(ifft2(fft2(b) / K))
print('relative error, noisy data:', hno.rel_error(x_naive, x))
hno.show([x, b, x_naive], ['true image $x$', 'data $b$', 'naive solution $B/K$']);
```

```
relative error, noisy data: 650076042681940.1
```

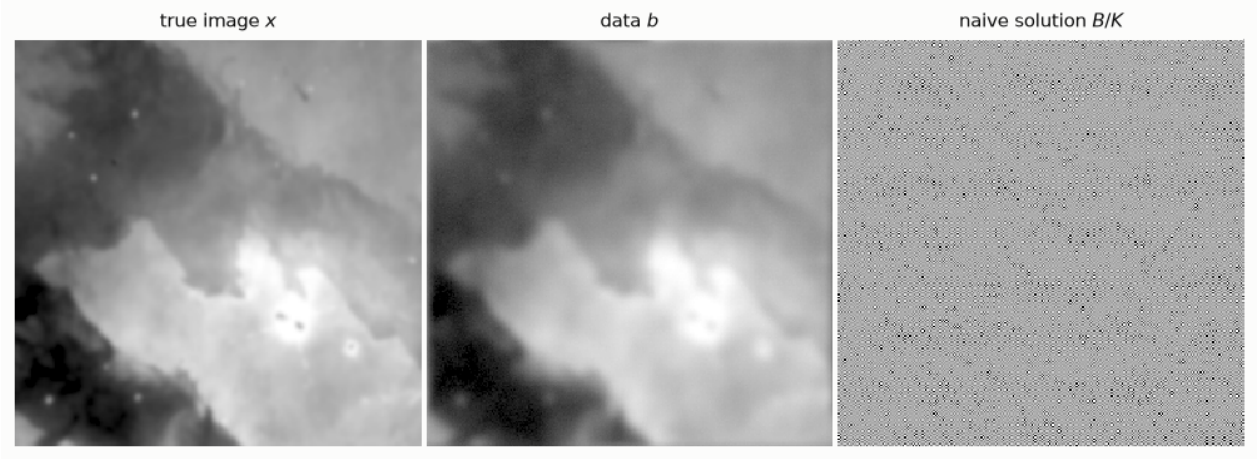


Figure 1.3: The naive solution of the image problem.

The naive solution is pure garbage. Not "a bit noisy": garbage.

1.5 Why: look at the spectrum

Write the noisy data in Fourier space, $B = KX + E$, and divide by K:

$$\frac{B}{K} = X + \frac{E}{K}.$$

The first term is what we want. The second is the **inverted noise**. Noise has roughly the same strength at all frequencies, but K falls towards zero at high frequencies, because a blur erases fine detail. Dividing a constant by something tiny gives something enormous.

Figure 1.4 shows the three actors, averaged over all directions:

```
nu, K_r = hno.radial_profile(K)
_, X_r = hno.radial_profile(fft2(x))
_, E_r = hno.radial_profile(fft2(e))

fig, ax = plt.subplots(figsize=(6.2, 3.4))
ax.semilogy(nu, X_r * K_r, color=hno.BLUE, label='blurred signal $|K X|$')
ax.semilogy(nu, E_r, color=hno.ORANGE, label='noise $|E|$')
ax.semilogy(nu, K_r, color=hno.INK2, lw=1.2, ls=(0, (4, 2)), label='transfer function $|K|$')
ax.set_ylim(1e-18, 1e5); ax.set_xlim(0, 0.5)
ax.set_xlabel('spatial frequency (cycles / pixel)')
ax.set_title('Signal, noise and transfer function', loc='left')
ax.legend(loc='lower left');
```

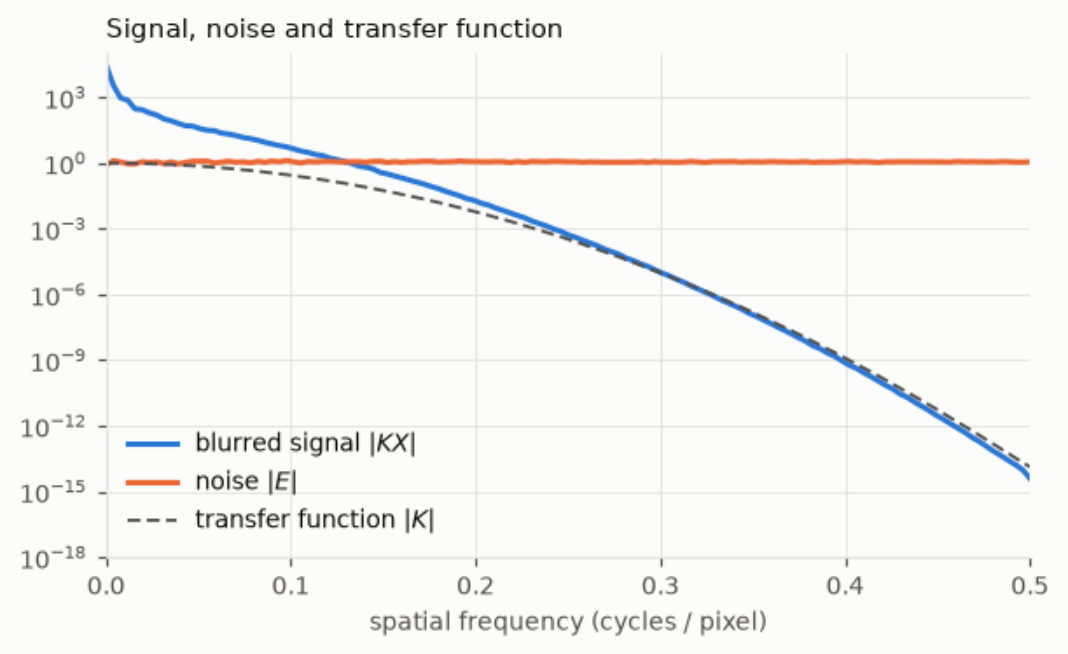


Figure 1.4: Blurred signal, noise and transfer function against spatial frequency, averaged over all directions.

Read it from left to right.

- At low frequencies the blurred signal is far above the noise: the data tell us about X there.
- The two curves cross at about 0.13 cycles per pixel. Beyond that point the data are **only noise**: the signal that was there has been pushed below it.
- $|K|$ keeps falling, to 10^{-14} at the right edge of the plot and lower still in the corners of the spectrum. There it meets the rounding floor of the computer, which is why even the noise-free inversion failed.

Dividing by K multiplies the noise by 10^{5} at frequency 0.3 and by 10^{11} at frequency 0.45. No measurement is good enough to survive that. The information is **gone**, and no algorithm can bring it back. What an algorithm *can* do is refrain from pretending.

1.6 A first cure: do not divide where K is small

The simplest honest method: invert where $|K|$ is larger than a threshold τ, and put zero elsewhere.

```
def truncated_inverse(b, K, tau):
    """Invert only the frequencies where |K| > tau; zero the others."""
    keep = np.abs(K) > tau
    X = np.zeros_like(K)
```

```
    X[keep] = fft2(b)[keep] / K[keep]
    return np.real(ifft2(X))

taus = np.logspace(-4, -0.1, 40)
errors = [hno.rel_error(truncated_inverse(b, K, t), x) for t in taus]
best = taus[int(np.argmin(errors))]

fig, ax = plt.subplots(figsize=(6.2, 3.2))
ax.loglog(taus, errors, color=hno.BLUE)
ax.axhline(hno.rel_error(b, x), color=hno.INK2, lw=1.0, ls=(0, (4, 2)))
ax.text(taus[0], hno.rel_error(b, x) * 1.08, 'error of the blurred data itself', color=hno.INK2,
 ↪  fontsize=8)
ax.plot(best, min(errors), 'o', color=hno.ORANGE)
ax.set_xlabel(r'threshold $\tau$'); ax.set_ylabel('relative error')
ax.set_title('Too small: noise wins.  Too large: blur stays.', loc='left');
print(f'best threshold {best:.3f}: error {min(errors):.3f}  (blurred data: {hno.rel_error(b,
 ↪  x):.3f})')
```

```
best threshold 0.100: error 0.028  (blurred data: 0.041)
```

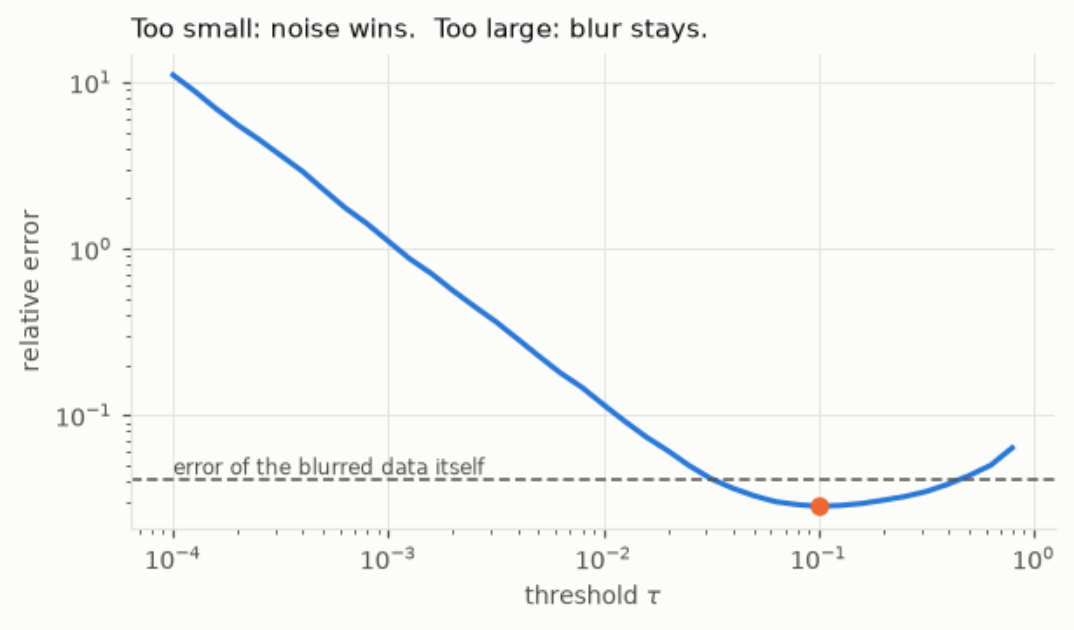


Figure 1.5: Relative error of the truncated inverse against the threshold. The dashed line is the error of the blurred data.

The U-shaped curve of Figure 1.5 is the single most important picture of the subject.

- **Left side** (τ too small): we invert frequencies that contain only noise, and the error explodes.
- **Right side** (τ too large): we throw away frequencies that did contain signal, and the result stays blurred.
- **In between** there is a best compromise, and it is clearly better than doing nothing.

```
hno.show([b, truncated_inverse(b, K, best), x],
         ['data $b$', f'truncated inverse, $\\tau$ = {best:.3f}', 'true image $x$'],
 ↪  same_scale=True);
```

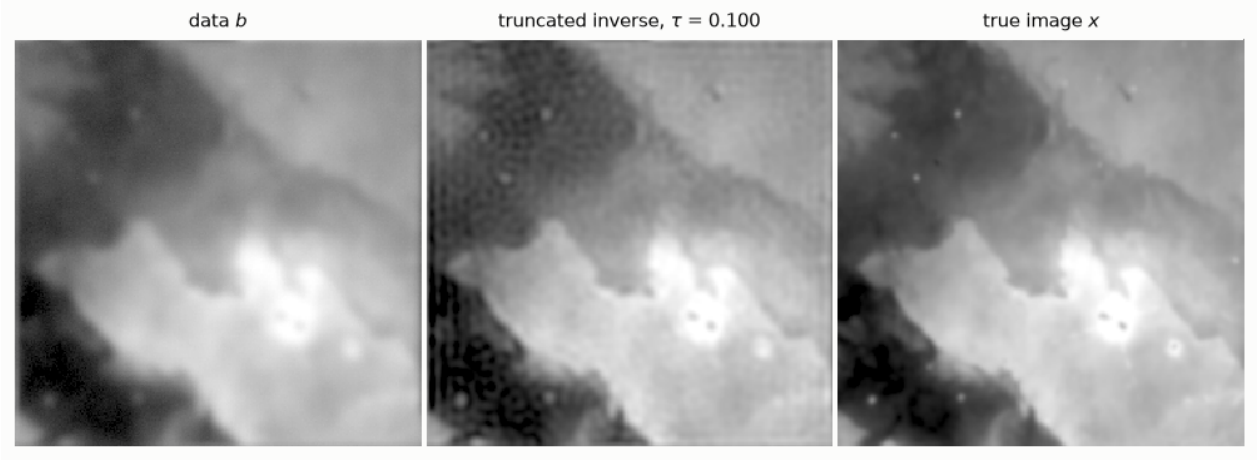


Figure 1.6: The data, the best truncated inverse, and the true image.

Sharper than the data, and no garbage. You can also see ripples, most clearly in the dark regions: cutting the spectrum abruptly creates them. Chapter 6 replaces the brutal cut by smoother *filters* (Tikhonov is the famous one), and chapter 5 explains the U-curve with the singular value decomposition.

One problem remains, and it is a serious one: we found the best τ **by comparing with the true image**. With real data there is no true image.

1.7 The real thing: one exposure of our telescope

Here is a single 5-minute exposure of the same region, taken with a 14 cm refractor, and its PSF. We did not choose this PSF: it was **measured on the stars of that very exposure**, which are natural point sources. It is not a Gaussian but a *Moffat* profile, with wider wings.

```
y = data['subs'][1].astype(float)                        # an exposure in median seeing
fmaj, fmin, pa, beta = data['sub_psf'][1]
k = hno.moffat_psf(y.shape, fmaj, fmin, pa, beta)
Ky = hno.psf2otf(k)
scale = float(data['pixel_scale'])
print(f'PSF width {np.sqrt(fmaj * fmin) * scale:.2f} arcsec  ({np.sqrt(fmaj * fmin):.2f} pixels),
 ↪  Moffat beta {beta:.2f}')

hno.show([y, np.real(ifft2(fft2(y) / Ky)), truncated_inverse(y, Ky, 0.3), truncated_inverse(y, Ky,
 ↪  0.05)],
         ['one exposure $y$', 'naive inverse', r'truncated, $\tau$ = 0.3', r'truncated, $\tau$ =
 ↪  0.05'],
         same_scale=True);
```

```
PSF width 2.89 arcsec  (3.78 pixels),  Moffat beta 3.06
```

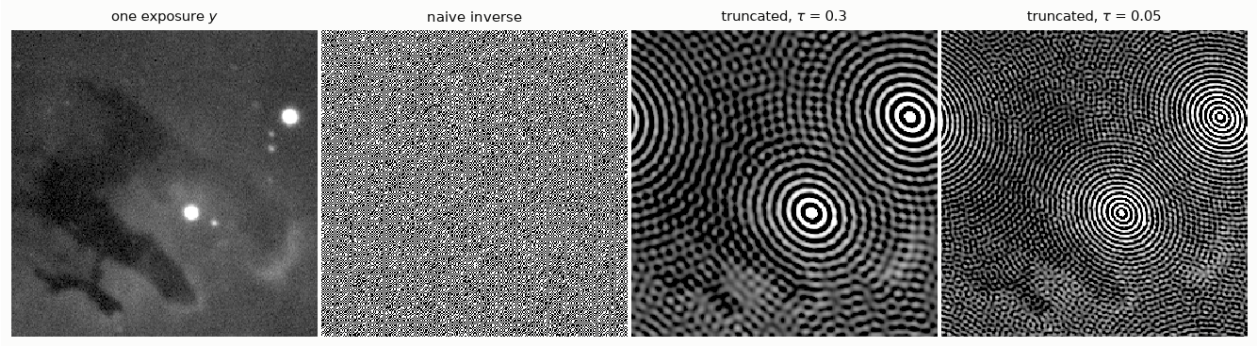


Figure 1.7: One real exposure, its naive inverse, and two truncated inverses.

Three things to see, all of them true of real data in general.

- The naive inverse is garbage, as before.
- With a threshold, the garbage is gone but **the whole image is covered with rings** centred on the two brightest stars. Cutting the spectrum abruptly is the same as convolving the image with an oscillating kernel, and the brighter the source, the stronger the oscillation: a star hundreds of times brighter than the nebula buries it. Worse, these two stars are *saturated*: the camera clipped them, so they do not obey $y = k * x + n$ at all. A model violation is punished much harder than noise.
- A bolder threshold adds amplified noise on top of the rings. And this time **there is no U-curve to tell us where to stop**, because nobody knows the true sky.

Three remarks that lead to the rest of the series and to our project.

- *A single exposure is mostly noise.* That is why astronomers take hundreds and combine them. Notebook 8 shows that combining and deblurring are one and the same linear operation, if you do it right.
- *Choosing the amount of regularisation without the truth* is the practical heart of the matter. Chapter 6 gives general-purpose rules. Our project uses a different idea: we **declare in advance** the resolution we will deliver, and then verify it on the stars of the result.
- *Stars and nebula want different treatments.* In the project we measure the PSF on the stars, then remove them before deblurring the nebula.

Your turn

Change the cells above, or add new ones below. Predict the answer before you run anything.

1. **Blur width.** Redo sections 1.1, 1.4, 1.5 and 1.6 with `fwhm=3` and with `fwhm=12`. How do the crossing frequency in the spectrum plot, the best τ and the best error change? Explain the trend in one sentence.
2. **Less noise.** Divide `noise_level` by 100. How many more frequencies can you recover? Is a hundred times less noise worth a hundred times more detail?
3. **Gaussian or Moffat?** Replace the Gaussian by `hno.moffat_psf(x.shape, 6.0, beta=2.5)`: same width, heavier wings. Is the problem easier or harder to invert? Look at $|K|$ before you answer.
4. **Periodic boundaries.** Make an image that is black except for one bright pixel near the right edge, and blur it. Where does the light go? Why is this wrong for a real camera, and why did it not hurt us here?
5. **Harder.** In section 1.5 we wrote $B/K = X + E/K$. For white noise of standard deviation σ per pixel, show that the expected squared error of the truncated inverse is $\sum_{|K|\le\tau} |X|^2 \; + \; \sigma^2 N \sum_{|K|>\tau} 1/|K|^2$ up to the normalisation of your FFT, and identify the two sides of the U-curve in this formula.

Where to read more

- Hansen et al. (2006), chapter 1 (this notebook), then chapters 3 and 4 for the structure of A and the boundary conditions.
- Gonzalez and Woods (2018), chapter 4 (Fourier transform, convolution theorem) and sections 5.5 to 5.7 (degradation model, inverse filtering).

2 Images as arrays

Companion notebook to Hansen, Nagy & O'Leary, ***Deblurring Images****, chapter 2. The chapter shows how to handle images in MATLAB. This notebook covers the same ground with NumPy, and you can read it instead of the chapter.*

What you will take away

1. An image is an array of numbers. Its shape, its data type and its index order are things you must always know.
2. **Displaying an image and computing with it are different jobs.** What you see on screen has been stretched; the numbers must stay untouched.
3. Integer types silently give wrong answers. Convert to floating point first.
4. Simple arithmetic on images already does useful science: averaging, differencing, measuring noise, masking.

```
import os, tempfile
import numpy as np
import matplotlib.pyplot as plt
import hnotools as hno

hno.style()
data = hno.load_sample()
```

2.1 An image is an array

Our working image is the average of 45 five-minute exposures of the Pillars of Creation through an oxygen filter, taken with a 14 cm refractor.

```
img = data['mean'].astype(float)
print('type :', type(img).__name__)
print('shape:', img.shape, ' (rows, columns)')
print('dtype:', img.dtype)
print(f'values from {img.min():.4f} to {img.max():.4f}, median {np.median(img):.5f}')
```

```
type : ndarray
shape: (256, 256)  (rows, columns)
dtype: float64
values from -0.0003 to 0.6288, median 0.00075
```

Three conventions to fix in your mind, because every bug in image code comes from one of them.

- **Index order is `[row, column]`**, that is `[y, x]`, not `[x, y]`.
- **Row 0 is at the top** when displayed, as in a matrix, not at the bottom as on a graph.
- **Slices are half-open**: `img[100:160]` takes rows 100 to 159.

Cropping is slicing:

```
crop = img[131:191, 138:198]                # 60 rows, 60 columns around a bright star
row, col = np.unravel_index(np.argmax(crop), crop.shape)
print('crop shape:', crop.shape, '  brightest pixel at row', row, 'column', col, 'of the crop')
hno.show([img, crop], ['the whole image', 'img[131:191, 138:198]'], width=3.4);
```

```
crop shape: (60, 60)   brightest pixel at row 21 column 12 of the crop
```

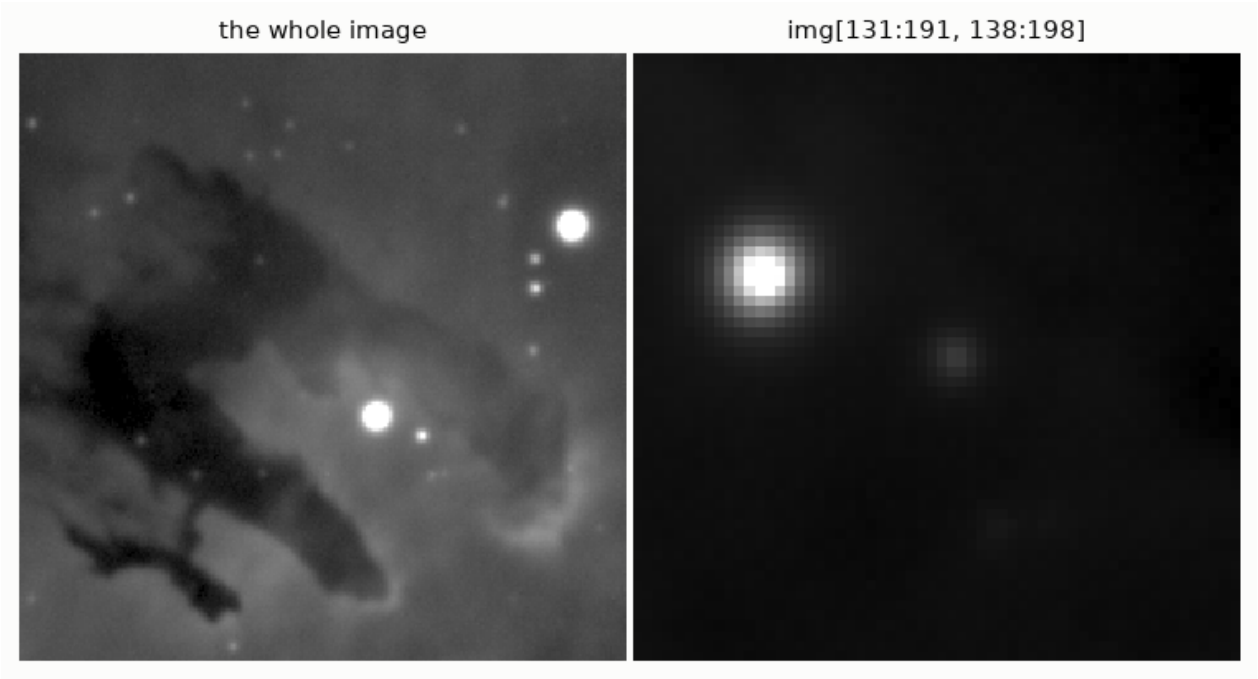


Figure 2.1: The mean of 45 exposures, and a crop obtained by slicing.

A slice is a **view**, not a copy: modifying `crop` would modify `img`. Write `img[131:191, 138:198].copy()` when you want an independent array.

2.2 What you see is not what you have

`hno.show` made the nebula visible. Here is the same array displayed honestly, black for the smallest value and white for the largest:

```
fig, ax = plt.subplots(1, 2, figsize=(9, 3.6), gridspec_kw={'width_ratios': [1, 1.5]})
ax[0].imshow(img, vmin=img.min(), vmax=img.max()); ax[0].grid(False)
ax[0].set_xticks([]); ax[0].set_yticks([]); ax[0].set_title('linear display, min to max')
ax[1].hist(img.ravel(), bins=200, color=hno.BLUE, log=True)
ax[1].set_xlabel('pixel value'); ax[1].set_ylabel('number of pixels')
ax[1].set_title('histogram of the pixel values', loc='left')
fig.tight_layout()
p1, p99 = np.percentile(img, [1, 99])
print(f'99 % of the pixels lie between {p1:.4f} and {p99:.4f}: {(p99 - p1) / (img.max() -
↪  img.min()):.1%} of the full range')
```

```
99 % of the pixels lie between -0.0001 and 0.0029: 0.5% of the full range
```

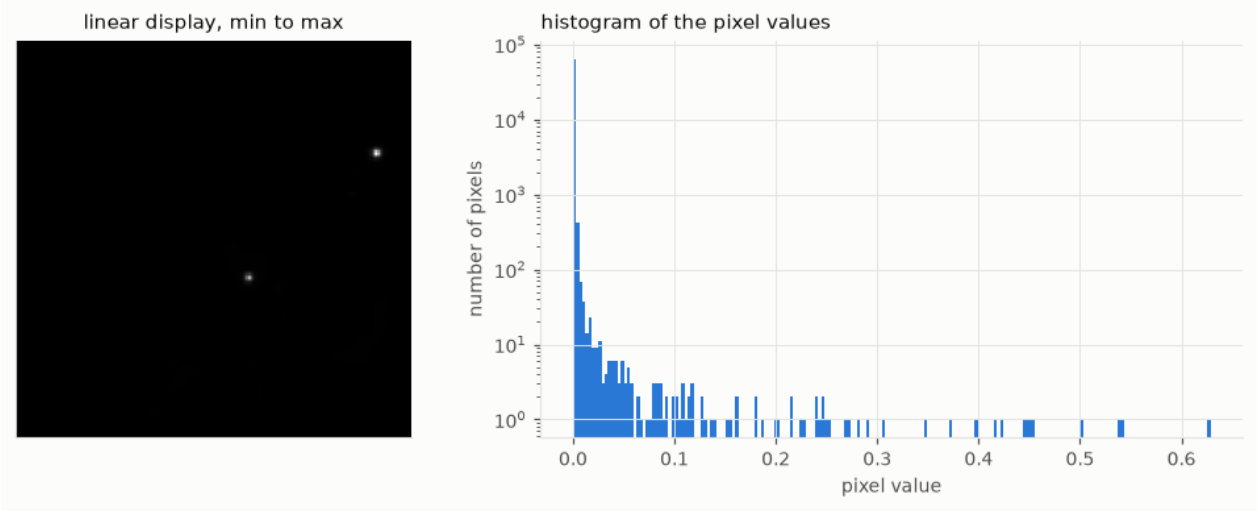


Figure 2.2: The same image displayed linearly from its minimum to its maximum, and the histogram of its values.

Two stars and nothing else. The nebula is all there, but it occupies less than one percent of the range of values, squeezed against zero. Astronomical images have an enormous **dynamic range**, and a screen does not.

To look at such an image we **stretch** it: we choose which values become black and white, and how the values in between are spread. Three common choices:

```
def stretch(a, lo, hi, kind='linear'):
    """Map values lo..hi to 0..1 for display. Returns a NEW array."""
    z = np.clip((a - lo) / (hi - lo), 0, 1)
    if kind == 'sqrt':
        z = np.sqrt(z)
    elif kind == 'asinh':
        z = np.arcsinh(30 * z) / np.arcsinh(30)
    return z

lo, hi = np.percentile(img, [1, 99.9])
fig, ax = plt.subplots(1, 3, figsize=(10, 3.6))
for a, kind in zip(ax, ('linear', 'sqrt', 'asinh')):
    a.imshow(stretch(img, lo, hi, kind), vmin=0, vmax=1); a.grid(False)
    a.set_xticks([]); a.set_yticks([]); a.set_title(f'percentiles 1 to 99.9, {kind}')
fig.tight_layout()
```

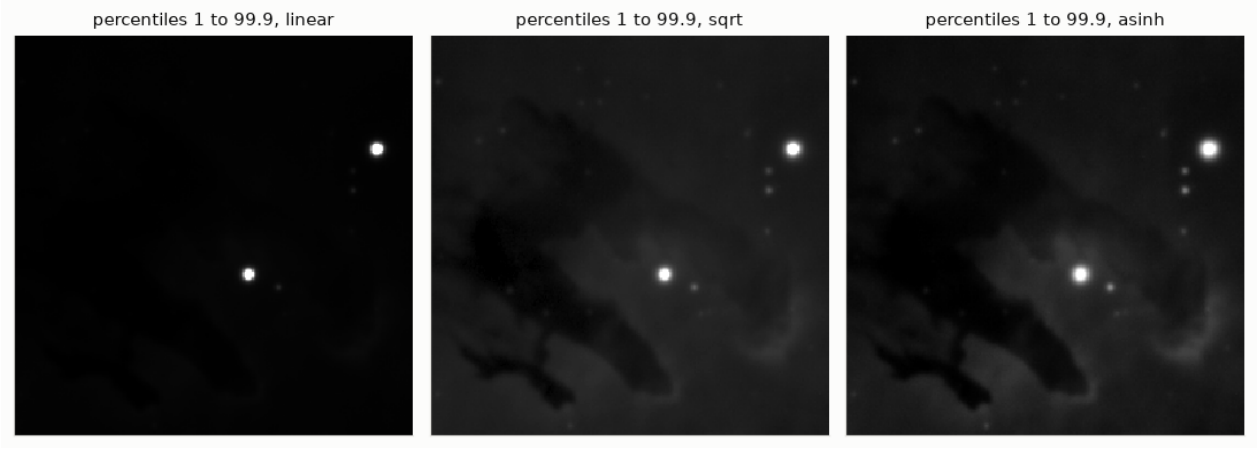


Figure 2.3: Three stretches of the same array.

The three panels show **the same array**. None is more "true" than the others. The stretch is a choice made for the eye, and it should never leak into the numbers.

2.2.1 Why it matters: measure a star on stretched data and you are wrong

The width of a star image, its *full width at half maximum* (FWHM), is the standard measure of sharpness. Let us measure it on a synthetic star whose width we know, before and after a square-root stretch.

```
def fwhm_by_area(a):
    """FWHM of a single round blob: diameter of the disc of pixels above half maximum."""
    return 2 * np.sqrt(np.sum(a >= 0.5 * a.max()) / np.pi)

star = hno.moffat_psf((513, 513), 40.0, beta=2.5)       # a wide star, so that pixels are small
print(f'true FWHM            : 40.0')
print(f'measured, linear data: {fwhm_by_area(star):.1f}')
print(f'measured, sqrt data  : {fwhm_by_area(np.sqrt(star)):.1f}')
```

```
true FWHM            : 40.0
measured, linear data: 40.0
measured, sqrt data  : 60.9
```

Fifty percent too wide. The half maximum of the stretched star is not the half maximum of the star. An early version of our own study compared two telescopes on stretched images and got widths wrong by a factor of two; the mistake was found only when the measurement was redone on linear data.

Rule: stretch for the eye, compute on the linear data.

2.3 Data types: integers lie

Image files usually store integers: 8 bits (0 to 255) for ordinary pictures, 16 bits (0 to 65 535) for cameras. NumPy keeps that type when it reads them, and integer arithmetic **wraps around without warning**:

```
a = np.array([200, 100, 50], dtype=np.uint8)
b = np.array([100, 100, 100], dtype=np.uint8)
print('a + b =', a + b, '   (200 + 100 should be 300)')
print('a - b =', a - b, '   (50 - 100 should be -50)')
print('as floats:', a.astype(float) + b, a.astype(float) - b)
```

```
a + b = [ 44 200 150]    (200 + 100 should be 300)
a - b = [100   0 206]    (50 - 100 should be -50)
as floats: [300. 200. 150.] [100.   0. -50.]
```

No error, no warning, wrong numbers. **Convert to floating point as soon as you have read an image**: `img = img.astype(float)`.

The second danger of integers is **quantisation**. Let us save our image as an 8-bit PNG, as one would for a web page, and read it back.

```
tmp = os.path.join(tempfile.mkdtemp(), 'nebula.png')
plt.imsave(tmp, img, cmap='gray', vmin=img.min(), vmax=img.max())     # linear, 8 bits
back = plt.imread(tmp)[:, :, 0].astype(float)                         # values k/255
back = back * (img.max() - img.min()) + img.min()                     # same units as img

levels = (p99 - p1) / (img.max() - img.min()) * 255
print(f'the nebula (99 % of the pixels) spans {levels:.1f} grey levels out of 255')
hno.show([img, back], ['original, floating point', 'after an 8-bit round trip'], width=3.6);
```

```
the nebula (99 % of the pixels) spans 1.2 grey levels out of 255
```

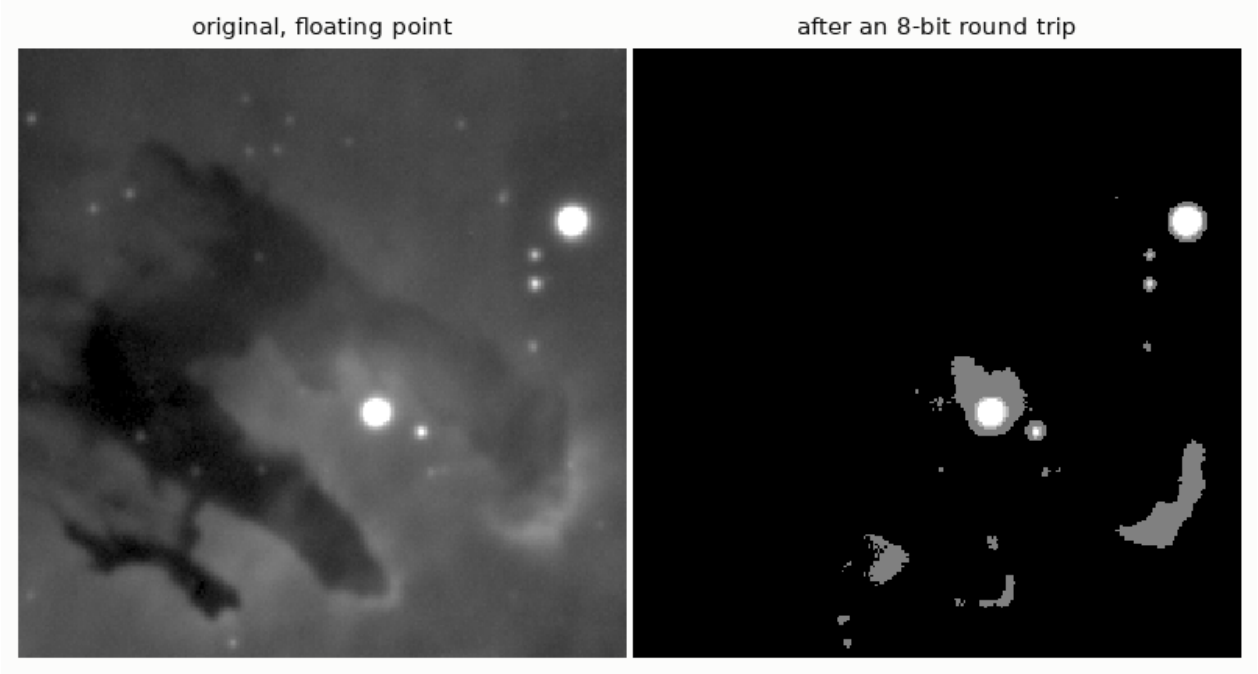


Figure 2.4: The image before and after being saved as an 8-bit file.

The whole nebula had to fit in about one grey level, and it is gone. This is why astronomers store images as 16-bit or floating-point **FITS** files, never as PNG or JPEG. (In Python, FITS files are read with the `astropy` package; our sample is stored as a NumPy `.npz` archive to keep the setup small.)

2.4 Arithmetic on images

With floating-point arrays, image arithmetic is array arithmetic. Three examples that we will use again and again.

Averaging exposures. `data['cube']` holds all 45 exposures of the central part of the field, as an array of shape `(45, 128, 128)`.

```
cube = data['cube'].astype(float)
one = cube[0]
mean45 = cube.mean(axis=0)                    # average along the first axis: over the exposures
hno.show([one, mean45], ['one exposure', 'mean of 45'], same_scale=True, width=3.4);
```

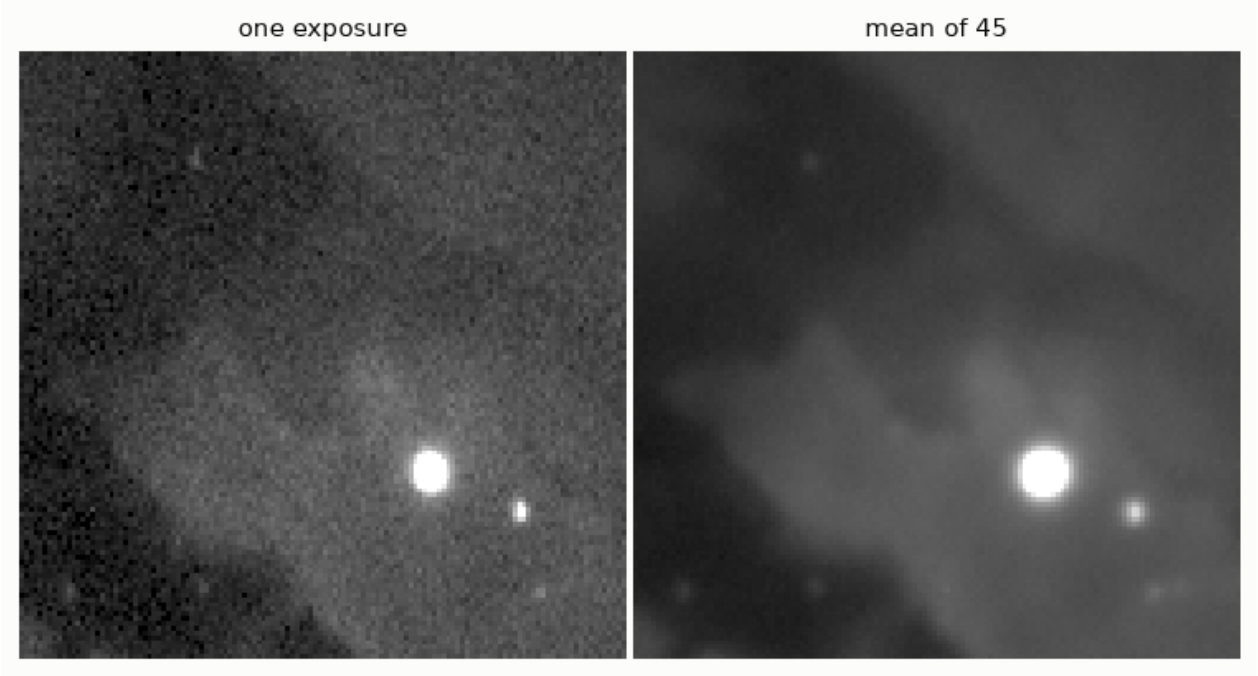


Figure 2.5: One exposure and the mean of 45, on the same display scale.

Measuring noise by differencing. The sky does not change between two exposures, so their difference contains no sky: only noise. If each exposure has noise σ, the difference has noise $\sigma\sqrt{2}$.

The same trick measures the noise of the *average*: split the exposures in two halves, average each, and take half the difference of the two averages.

We estimate the spread with the **median absolute deviation** (MAD) rather than the standard deviation, because stars are not noise and a few bright pixels would dominate a standard deviation.

```
def robust_sigma(a):
    """Standard deviation estimated from the median absolute deviation."""
    return 1.4826 * np.median(np.abs(a - np.median(a)))

sigma_one = robust_sigma(cube[0] - cube[1]) / np.sqrt(2)
sigma_mean = robust_sigma(cube[0::2].mean(axis=0) - cube[1::2].mean(axis=0)) / 2
print(f'noise of one exposure : {sigma_one:.2e}')
print(f'noise of the mean     : {sigma_mean:.2e}')
print(f'ratio                 : {sigma_one / sigma_mean:.1f}    (sqrt(45) = {np.sqrt(45):.1f})')
```

```
noise of one exposure : 2.69e-04
noise of the mean     : 4.01e-05
ratio                 : 6.7    (sqrt(45) = 6.7)
```

Averaging N exposures divides the noise by about $\sqrt{N}$. This is the whole reason for taking many exposures, and this "split in two halves" test is how our project puts error bars on its results.

Masks. A comparison gives an array of booleans, which selects pixels.

```
saturated = one > 0.5 * one.max()           # the camera clipped these pixels
print('saturated pixels in this exposure:', saturated.sum())
print(f'median of the sky, stars excluded: {np.median(one[~saturated]):.2e}')
```

```
saturated pixels in this exposure: 5
median of the sky, stars excluded: 9.12e-04
```

2.5 Colour images

A colour image is three images stacked along a third axis: shape `(rows, columns, 3)`, in the order red, green, blue.

Our telescope camera is monochrome. We photograph the nebula three times through narrow filters that isolate the light of sulphur (S II), hydrogen (Hα) and oxygen (O III), and *assign* them to red, green and blue. This is the "Hubble palette" of the famous picture.

```
sho = data['mean_sho'].astype(float)         # shape (3, 256, 256): S II, H-alpha, O III
print('stored as', sho.shape)

rgb = np.stack([stretch(c, *np.percentile(c, [1, 99.8]), 'asinh') for c in sho], axis=-1)
print('for display', rgb.shape, ' values', rgb.min(), 'to', rgb.max())

fig, ax = plt.subplots(1, 4, figsize=(12, 3.3))
for a, c, name in zip(ax, sho, ('S II  (red)', 'Hα  (green)', 'O III  (blue)')):
    a.imshow(stretch(c, *np.percentile(c, [1, 99.8]), 'asinh'), vmin=0, vmax=1); a.set_title(name)
ax[3].imshow(rgb); ax[3].set_title('colour composite')
for a in ax:
    a.set_xticks([]); a.set_yticks([]); a.grid(False)
fig.tight_layout()
```

```
stored as (3, 256, 256)
for display (256, 256, 3)  values 0.0 to 1.0
```

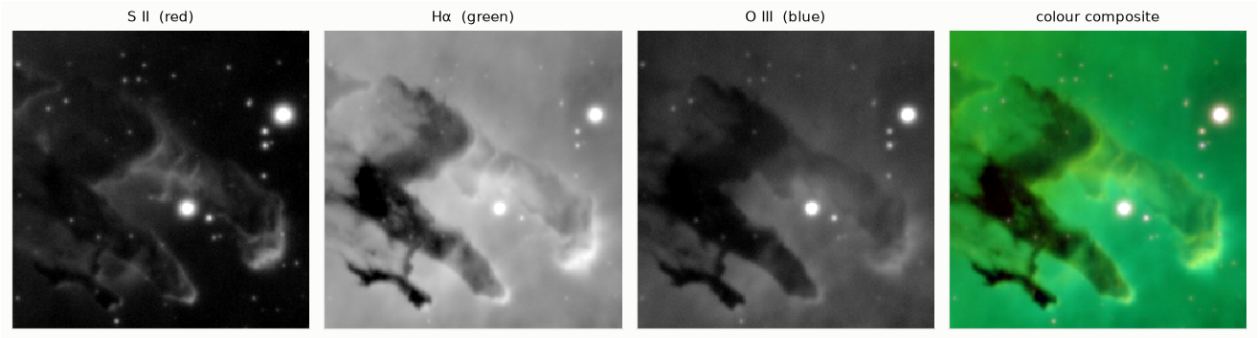


Figure 2.6: The three narrowband channels and their colour composite.

`imshow` expects a colour array of floats between 0 and 1, which is why each channel was stretched separately before stacking. The result is green almost everywhere, because hydrogen is by far the brightest gas in this nebula. Published pictures rebalance the three channels to taste: the colours are a display choice too. Chapter 7 comes back to colour, where each channel has its own blur.

Your turn

1. **Index order.** Without running it, say what `img[10, 200]`, `img[:, 0]` and `img[::2, ::2]` are. Check the shape of each.
2. **View or copy?** Set `c = img[0:5, 0:5]`, then `c[:] = 0`. Look at `img[0, 0]`. Reload the data and repeat with `.copy()`.
3. **Sixteen bits.** In section 2.3, how many levels would the nebula get in a 16-bit file? Compare that step with the noise of one exposure measured in section 2.4. Is 16 bits enough for a single exposure? For the mean of 45?
4. **Stretch and width.** Redo the star measurement of section 2.2 with the `asinh` stretch. Then explain, with the formula of the Moffat profile in `hnotools.py`, why the square-root stretch gives exactly this factor.
5. **Noise in practice.** Compute `robust_sigma(cube[i] - cube[i + 1])` for all consecutive pairs and plot it against `i`. Did every exposure have the same noise? What could make one night noisier than another?

Where to read more

- Hansen et al. (2006), chapter 2, for the same ideas in MATLAB.
- Gonzalez and Woods (2018), chapter 2 (sampling, quantisation, arithmetic on images) and section 3.2 (intensity transformations, which is what a stretch is).
- The NumPy user guide, “Indexing on ndarrays” and “Copies and views”.

3 The blurring function

Companion notebook to Hansen, Nagy & O'Leary, ***Deblurring Images****, chapter 3.*

What you will take away

1. The **point-spread function** (PSF) is the image of a point. It contains everything there is to know about a spatially invariant blur.
2. The columns of the blurring matrix A are shifted copies of the PSF.
3. A PSF can come from a physical model (defocus, atmosphere) or, better, be **measured on the data**: in astronomy every star is a free point source.
4. Noise has a physical origin too, and it tells us how much to trust each measurement.
5. A picture is a window on a larger scene. What we assume about the scene *outside* the window, the **boundary condition**, changes the matrix A.

```
import numpy as np
from scipy import ndimage, optimize
import matplotlib.pyplot as plt
import hnotools as hno

hno.style()
rng = np.random.default_rng(0)
data = hno.load_sample()
```

3.1 The PSF is the image of a point

Blur an image that is black except for one bright pixel. What comes out is, by definition, the PSF, centred where the pixel was.

```
point = np.zeros((64, 64)); point[20, 40] = 1.0
psf_small = hno.gaussian_psf((15, 15), fwhm=5.0)          # a 15 x 15 array holding the PSF

blurred_point = ndimage.convolve(point, psf_small, mode='constant')
two_points = point.copy(); two_points[45, 15] = 0.5
blurred_two = ndimage.convolve(two_points, psf_small, mode='constant')

hno.show([point, blurred_point, two_points, blurred_two],
         ['one point', 'its image: the PSF', 'two points', 'two PSFs, added'], lo=0, hi=100,
 ↪  gamma=1);
```

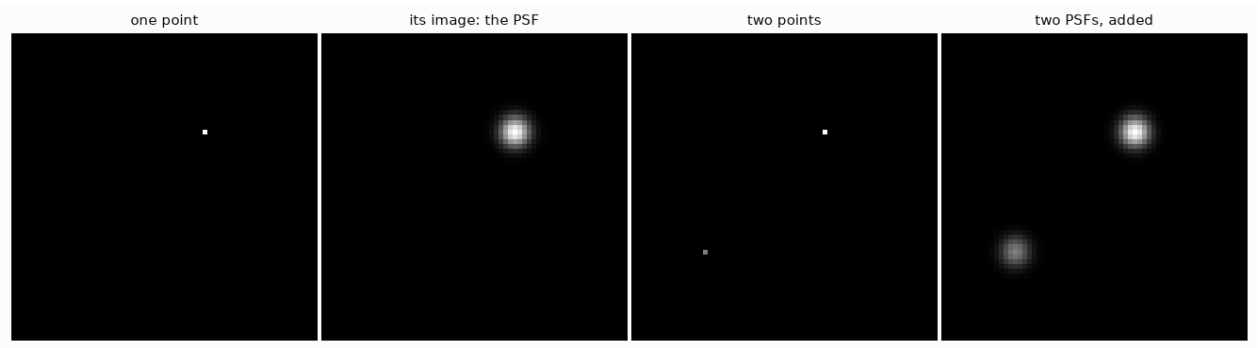


Figure 3.1: A point and its image; two points and their image.

Because blurring is linear, the image of any scene is the sum of the PSFs of all its points, each weighted by the brightness of the point. That operation has a name: **convolution**, written $b = k * x$.

Two properties of a physical PSF that we will rely on:

- it is **non-negative** (light cannot be subtracted);
- it **sums to one** if the blur neither creates nor loses light.

```
print('minimum of the PSF:', psf_small.min(), '   sum of the PSF:', psf_small.sum())
print('total light before:', two_points.sum(), '   after:', round(blurred_two.sum(), 12))
```

```
minimum of the PSF: 6.729172487219203e-07    sum of the PSF: 1.0000000000000002
total light before: 1.5    after: 1.5
```

3.2 The matrix behind the convolution

Notebook 1 showed the matrix of a 1-D blur. Let us build the matrix of a 2-D blur, on an image small enough to afford it: 12×12 pixels, hence A of size 144×144.

To write $b = Ax$ the image must become a vector. **Here the book and NumPy differ.** MATLAB stacks the *columns* of the image; NumPy's `ravel()` stacks the *rows*. Both are fine as long as you are consistent. We use NumPy's order.

Column j of A is what A does to the j-th unit vector, that is, to an image with a single bright pixel. So column j is the PSF centred on pixel j, flattened.

```
n = 12
psf_tiny = hno.gaussian_psf((7, 7), fwhm=3.0)

def build_matrix(n, psf, mode):
    """Blurring matrix of an n x n image, one column per pixel."""
    A = np.zeros((n * n, n * n))
    for j in range(n * n):
        unit = np.zeros(n * n); unit[j] = 1.0
        A[:, j] = ndimage.convolve(unit.reshape(n, n), psf, mode=mode).ravel()
    return A

A = build_matrix(n, psf_tiny, 'constant')

fig, ax = plt.subplots(1, 3, figsize=(10.5, 3.6))
ax[0].imshow(A, cmap='gray_r'); ax[0].set_title('the matrix $A$ (144 x 144)')
ax[1].imshow(A[:36, :36], cmap='gray_r'); ax[1].set_title('its top-left corner')
ax[2].imshow(A[:, 5 * n + 6].reshape(n, n), cmap='gray'); ax[2].set_title('column 66, reshaped: a
 ↪  PSF')
for a in ax:
    a.set_xticks([]); a.set_yticks([]); a.grid(False)
fig.tight_layout()

x_test = rng.random((n, n))
print('A x equals the convolution:', np.allclose(A @ x_test.ravel(), ndimage.convolve(x_test,
 ↪  psf_tiny, mode='constant').ravel()))
```

```
A x equals the convolution: True
```

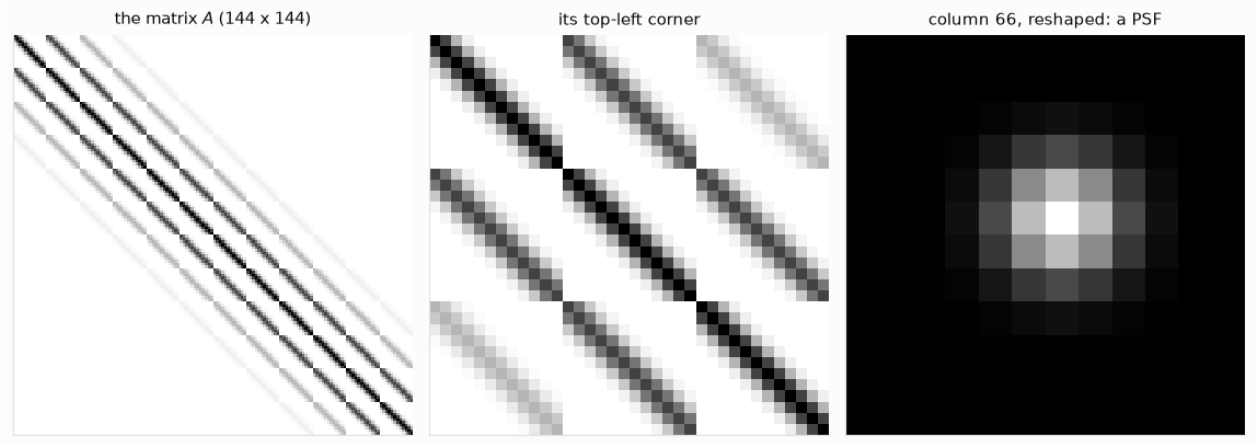


Figure 3.2: The blurring matrix of a 12 x 12 image, a detail of it, and one of its columns reshaped as an image.

A is made of **blocks**, each block is a **band**: bands inside bands. This regular structure is why we will never need to store A; notebook 4 turns it into fast algorithms.

3.3 Where PSFs come from: three models

- **Out-of-focus blur**: a point becomes a uniform disc.
- **Gaussian blur**: a convenient mathematical model, and a fair one for the sum of many small independent blurs.
- **Moffat blur**: the shape of a star seen through the turbulent atmosphere with a long exposure. It has a Gaussian-like core and much wider wings, $k(r) \propto (1 + r^2/\alpha^2)^{-\beta}$. This is the PSF of our telescope images.

The three PSFs below have the same full width at half maximum, 8 pixels.

```
shape = (129, 129)
yy, xx = np.indices(shape) - 64
disc = (np.hypot(xx, yy) <= 4.0).astype(float); disc /= disc.sum()
gauss = hno.gaussian_psf(shape, 8.0)
moffat = hno.moffat_psf(shape, 8.0, beta=2.5)

fig, ax = plt.subplots(1, 2, figsize=(10.5, 3.3))
r = np.arange(0, 40)
for psf, name, col in ((disc, 'disc (defocus)', hno.AQUA), (gauss, 'Gaussian', hno.BLUE), (moffat,
 ↪  'Moffat', hno.ORANGE)):
    ax[0].semilogy(r, psf[64, 64:64 + 40] / psf[64, 64], color=col, label=name)
    nu, otf = hno.radial_profile(hno.psf2otf(psf))
    ax[1].semilogy(nu, otf, color=col, label=name)
ax[0].set_ylim(1e-6, 2); ax[0].set_xlabel('distance from the centre (pixels)')
ax[0].set_title('the PSF, cut through its centre', loc='left'); ax[0].legend()
ax[1].set_ylim(1e-8, 2); ax[1].set_xlabel('spatial frequency (cycles / pixel)')
ax[1].set_title('its transfer function $|K|$', loc='left')
fig.tight_layout()
```

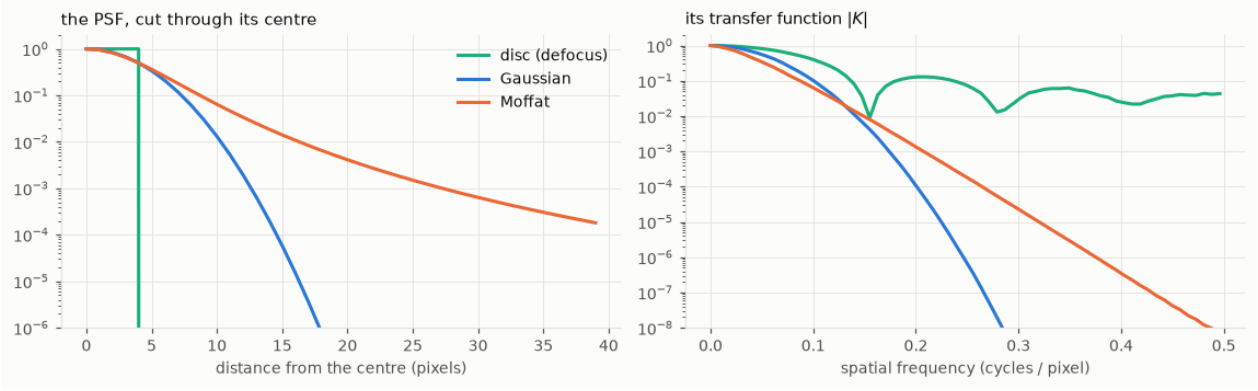


Figure 3.3: Three PSF models of the same width: a cut through the PSF (left) and the transfer function (right).

Same width, three very different inverse problems. Read the right-hand plot of Figure 3.3 with Figure 1.4 in mind: wherever $|K|$ is small, the data hold no usable information.

- The **Gaussian** kills high frequencies fastest: the hardest to invert.
- The **Moffat** keeps a little of everything: its wide wings in image space are the price of a transfer function that decays slowly.
- The **disc** has a transfer function that oscillates and dips to zero at regular intervals: some frequencies are erased exactly. (The dips do not quite reach zero on the plot because a disc drawn on a pixel grid is not perfectly round.)

3.4 Measuring the PSF on the data

Models are useful, measurements are better. A star is so far away that it is a perfect point source, so **each star in an image is a picture of the PSF**, with some noise. Let us find stars in one of our exposures, cut one out, and fit a Moffat profile to it.

```
frame = data['subs'][1].astype(float)                    # an exposure in median seeing
stars = hno.find_stars(frame, n=8)
for row, col, peak in stars:
    print(f'star at row {row:3d}, column {col:3d}, peak {peak:.3f}')
```

```
star at row  72, column 233, peak 0.548
star at row 152, column 150, peak 0.336
star at row 161, column 169, peak 0.010
star at row  98, column 217, peak 0.006
star at row  86, column 217, peak 0.005
star at row 124, column 216, peak 0.004
star at row 202, column 223, peak 0.004
star at row 210, column  88, peak 0.003
```

The two brightest are saturated: the camera reached its maximum and clipped them, so they no longer show the true shape of the PSF. We take the brightest star that is safely below saturation.

```
row, col, peak = next(s for s in stars if s[2] < 0.25 * stars[0][2])
h = 10
stamp = frame[row - h:row + h + 1, col - h:col + h + 1]

def moffat_model(p, yy, xx):
    """Circular Moffat on a constant background. p = amplitude, y0, x0, alpha, beta, background."""
    amp, y0, x0, alpha, beta, bg = p
    return bg + amp * (1 + ((yy - y0)**2 + (xx - x0)**2) / alpha**2) ** (-beta)
```

```
yy, xx = np.indices(stamp.shape) - h
fit = optimize.least_squares(lambda p: (moffat_model(p, yy, xx) - stamp).ravel(),
                             x0=[stamp.max(), 0, 0, 3.0, 2.5, np.median(stamp)],
                             bounds=([0, -3, -3, 0.5, 1.1, -1], [10, 3, 3, 30, 10, 1]))
amp, y0, x0, alpha, beta, bg = fit.x
fwhm_fit = 2 * alpha * np.sqrt(2 ** (1 / beta) - 1)

fmaj, fmin, pa, beta_ref = data['sub_psf'][1]
print(f'our fit on one star   : FWHM {fwhm_fit:.2f} px, beta {beta:.2f}')
print(f'the project, 60 stars : FWHM {np.sqrt(fmaj * fmin):.2f} px, beta {beta_ref:.2f}')

model = moffat_model(fit.x, yy, xx)
dist = np.hypot(yy - y0, xx - x0).ravel()
fig, ax = plt.subplots(1, 4, figsize=(12, 3.0), gridspec_kw={'width_ratios': [1, 1, 1, 1.5]})
for a, im, name in zip(ax, (stamp, model, bg + 3 * (stamp - model)), ('the star', 'the Moffat fit',
 ↪  'star minus fit (x 3)')):
    a.imshow(im, vmin=bg - 0.3 * amp, vmax=bg + amp); a.set_title(name)
    a.set_xticks([]); a.set_yticks([]); a.grid(False)
ax[3].plot(dist, stamp.ravel() - bg, '.', color=hno.INK2, ms=3, label='pixels of the star')
rr = np.linspace(0, dist.max(), 200)
ax[3].plot(rr, amp * (1 + rr**2 / alpha**2) ** (-beta), color=hno.ORANGE, label='Moffat fit')
ax[3].set_xlabel('distance from the centre (pixels)'); ax[3].set_title('radial profile',
 ↪  loc='left'); ax[3].legend()
fig.tight_layout()
```

```
our fit on one star   : FWHM 3.65 px, beta 3.15
the project, 60 stars : FWHM 3.78 px, beta 3.06
```

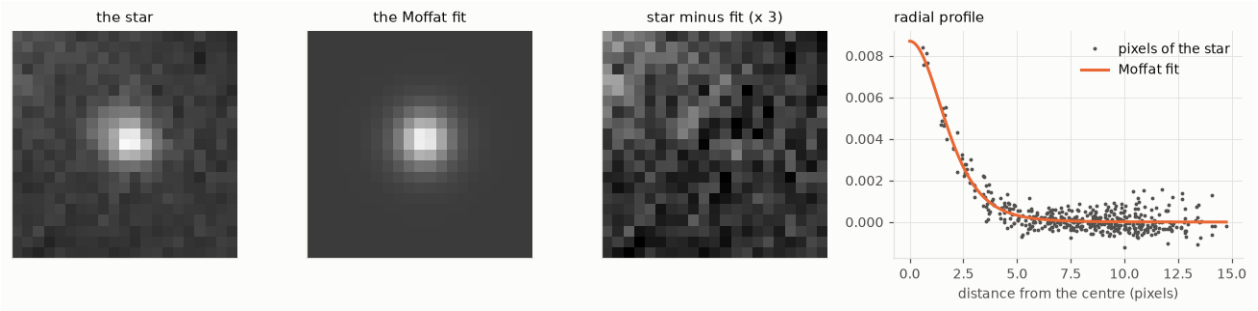


Figure 3.4: A star of a real exposure, the Moffat profile fitted to it, the residual, and the radial profile.

One star, twenty lines of code, and we recover the width that the project measures by fitting sixty stars per exposure with an elliptical model. The remaining difference is honest: one faint star is a noisy estimate, and a round model cannot follow a slightly elongated PSF. The exponent β is the hardest parameter to pin down, because it is set by the faint wings.

This is the foundation of our project: **the blur is not assumed, it is measured**, exposure by exposure.

3.5 Noise

The noise in an astronomical image has two main sources.

- **Photon noise.** Light arrives as photons, at random. If a pixel expects λ photons, it receives a Poisson-distributed number, with mean λ and standard deviation $\sqrt{\lambda}$. Brighter means noisier in absolute terms, but cleaner in relative terms.
- **Read noise.** The electronics add a small Gaussian error to every pixel, the same everywhere.

```
levels = np.array([4, 25, 100, 400, 2500])
counts = rng.poisson(levels, size=(20000, len(levels)))
for lam, c in zip(levels, counts.T):
    print(f'expected {lam:5d} photons: measured mean {c.mean():7.1f}, std {c.std():5.1f}  (sqrt =
     ↪  {np.sqrt(lam):5.1f}),  relative noise {c.std() / lam:5.1%}')
```

```
expected     4 photons: measured mean     4.0, std   2.0  (sqrt =   2.0),  relative noise 50.3%
expected    25 photons: measured mean    25.0, std   5.0  (sqrt =   5.0),  relative noise 19.9%
expected   100 photons: measured mean   100.0, std  10.0  (sqrt =  10.0),  relative noise 10.0%
expected   400 photons: measured mean   399.9, std  20.0  (sqrt =  20.0),  relative noise  5.0%
expected  2500 photons: measured mean  2500.2, std  50.2  (sqrt =  50.0),  relative noise  2.0%
```

For our images there is a simplification. Through a narrow filter the nebula is faint, and most photons come from the **sky background**, which is the same everywhere. So the noise has nearly the same standard deviation in every pixel, and since it is the sum of many photons it is very nearly Gaussian. Check it on real data, using the difference of two exposures (notebook 2):

```
cube = data['cube'].astype(float)
diff = (cube[0] - cube[1]) / np.sqrt(2)
sigma = 1.4826 * np.median(np.abs(diff - np.median(diff)))

fig, ax = plt.subplots(figsize=(6.2, 3.2))
ax.hist(diff.ravel() / sigma, bins=np.linspace(-6, 6, 97), density=True, color=hno.BLUE, alpha=0.55,
 ↪  label='difference of two exposures')
t = np.linspace(-6, 6, 300)
ax.plot(t, np.exp(-t**2 / 2) / np.sqrt(2 * np.pi), color=hno.ORANGE, label='Gaussian')
ax.set_yscale('log'); ax.set_ylim(1e-5, 1); ax.set_xlabel('value / $\\sigma$')
ax.set_title('Is the noise Gaussian?', loc='left'); ax.legend(loc='lower center');
```

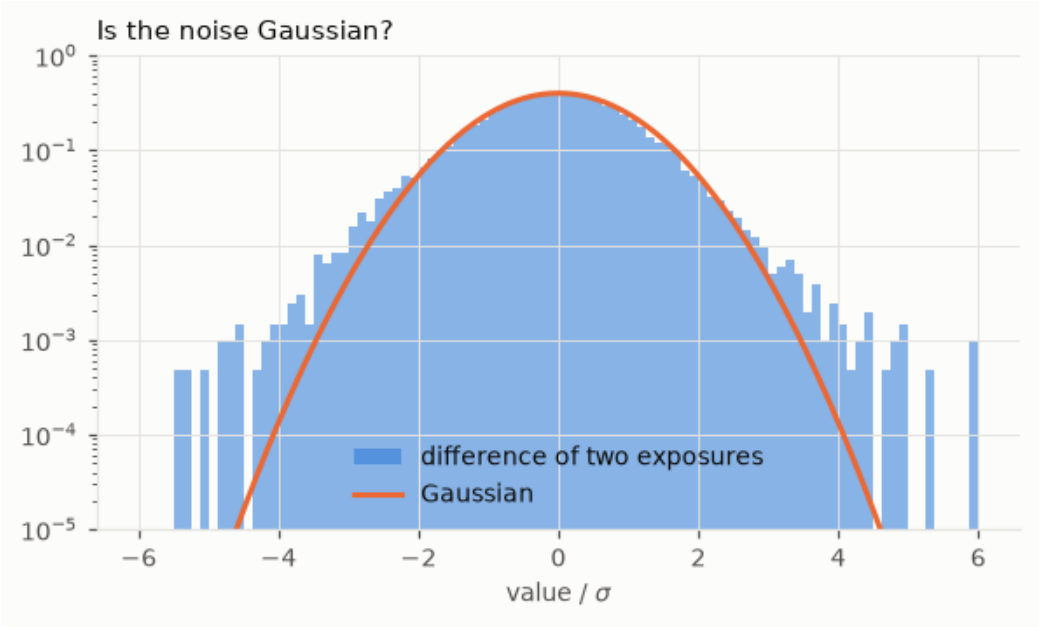


Figure 3.5: Histogram of the difference of two exposures, in units of its standard deviation, with a Gaussian.

Gaussian over several orders of magnitude in the centre, with **heavier tails**: those are the stars, which are not identical in the two exposures because the blur changed. Keep this picture in mind for the rest of the series. “White Gaussian noise” is an excellent model of the sky and a poor model of bright stars.

3.6 Boundary conditions

A blurred pixel near the edge of a picture received light from points *outside* the picture. To write $b = Ax$ for the picture alone, we must say what the scene looks like out there. Three classical assumptions:

- **zero**: the scene is black outside;
- **periodic**: the scene repeats itself like a tiling;
- **reflexive**: the scene outside is the mirror image of the scene inside.

None of them is true. To see how wrong each one is, we blur a large image (the truth), keep only its central part, and compare with what each assumption predicts from the central part alone.

```
big = data['hubble'].astype(float)                          # 256 x 256: the "whole scene"
psf_bc = hno.gaussian_psf((33, 33), fwhm=8.0)
inner = slice(64, 192)                                      # the camera sees the central 128 x 128

truth = ndimage.convolve(big, psf_bc, mode='reflect')[inner, inner]     # light from outside
 ↪  included
x_in = big[inner, inner]
panels, names = [], []
for mode, name in (('constant', 'zero'), ('wrap', 'periodic'), ('reflect', 'reflexive')):
    predicted = ndimage.convolve(x_in, psf_bc, mode=mode)
    err = np.abs(predicted - truth)
    panels.append(err); names.append(f'{name}: largest error {err.max():.3f}')
    print(f'{name:10s} error at the border {err[0, :].mean():.4f}   in the centre {err[48:80,
     ↪  48:80].mean():.1e}')

fig, ax = plt.subplots(1, 3, figsize=(10.5, 3.7))
for a, e, name in zip(ax, panels, names):
    a.imshow(e, vmin=0, vmax=0.15, cmap='gray_r'); a.set_title(name)
    a.set_xticks([]); a.set_yticks([]); a.grid(False)
fig.tight_layout()
```

```
zero       error at the border 0.1576   in the centre 0.0e+00
periodic   error at the border 0.1818   in the centre 0.0e+00
reflexive  error at the border 0.0041   in the centre 0.0e+00
```

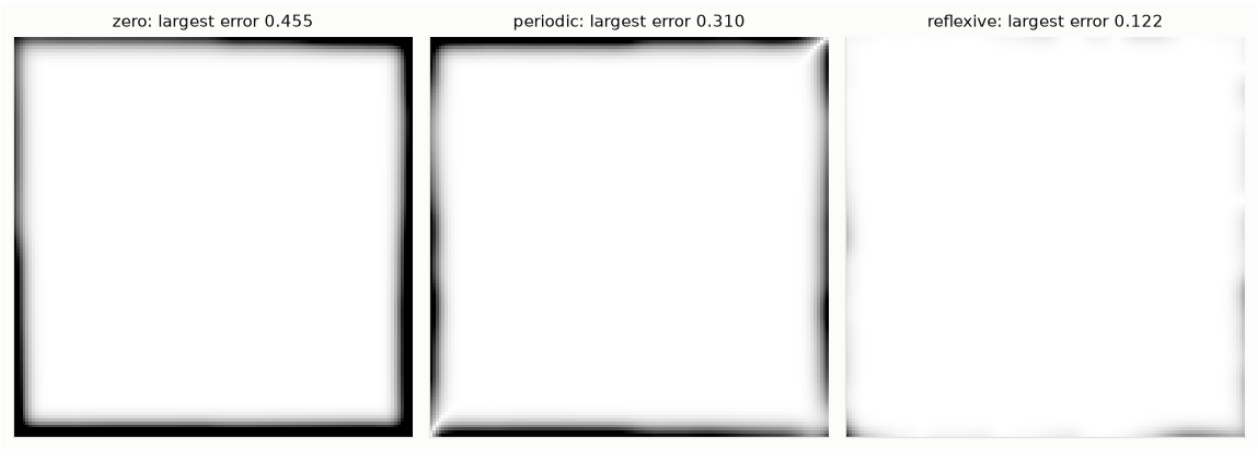


Figure 3.6: Error of the three boundary conditions: white where the assumption is right, dark where it is wrong.

The error maps are white where the assumption is right and dark where it is wrong. All three are exact in the centre and wrong in a frame as wide as the PSF. The zero and periodic conditions are badly wrong here: the sky is never black, and the left edge of this scene does not resemble its right edge. The reflexive condition is forty times better, because a mirror image at least continues the scene without a jump.

An error in the model of the blur is not like noise: it is **structured**, and the inversion amplifies it just as eagerly. Deblurring with a wrong boundary condition produces ripples that start at the edges and travel inwards. Notebook 4 shows how each boundary condition leads to its own fast algorithm, and what to do in practice.

Your turn

1. **Columns and rows.** In section 3.2, display *row* 66 of A reshaped as a 12×12 image. Why does it look like the column? Would it for a PSF that is not symmetric? Try with a PSF shifted by one pixel.
2. **Matrices for other boundaries.** Build A with `mode='wrap'` and with `mode='reflect'` and display them next to the zero-boundary one. Where do the new non-zero entries appear, and why there?
3. **A defocused star.** Blur `big` with the disc PSF of section 3.3 and plot the radial profile of the Fourier transform of the result. Can you see the frequencies that the disc erased?
4. **Another star.** Fit the Moffat model to two other unsaturated stars of `frame`. How much do the three widths differ? Now fit one of the saturated stars: what happens to the fit, and to the residual image?
5. **Which exposure is sharpest?** `data['subs']` holds three exposures. Fit the same star in each and rank them. Compare with `data['sub_psf']`.
6. **Noise and brightness.** In section 3.5, at what number of photons does the relative noise fall below 1 %? Below 0.1 %? What does that say about the exposure time needed to measure a faint nebula to 1 %?

Where to read more

- Hansen et al. (2006), chapter 3.
- Gonzalez and Woods (2018), sections 5.2 (noise models) and 5.6 (estimating the degradation function).
- Talbot (2024), section 2.2, for well-posed and ill-posed problems.

4 Structured computations

Companion notebook to Hansen, Nagy & O'Leary, ***Deblurring Images****, chapter 4.*

The blurring matrix of a megapixel image has 10^{12} entries. This notebook is about never building it.

What you will take away

1. Each boundary condition gives A a recognisable structure.
2. **Periodic** boundaries: A is diagonalised by the Fourier transform (FFT).
3. **Reflexive** boundaries with a symmetric PSF: A is diagonalised by the cosine transform (DCT).
4. **Separable** PSF: A is a Kronecker product of two small matrices.
5. Test data must be created *without* assuming your favourite boundary condition, or you will fool yourself.

```
import numpy as np
from numpy.fft import fft, ifft, fft2, ifft2
from scipy import ndimage
from scipy.fft import dctn, idctn
import matplotlib.pyplot as plt
import hnotools as hno

hno.style()
rng = np.random.default_rng(0)
data = hno.load_sample()
```

4.1 One dimension: three boundary conditions, three structures

Take a signal of 12 samples and a kernel of 5. The matrix of the blur depends on what we assume beyond the two ends.

```
N = 12
kern = np.array([1.0, 2.0, 3.0, 2.0, 1.0]); kern /= kern.sum()

def matrix_1d(kern, N, mode):
    """Blurring matrix of a signal of length N: one column per sample."""
    return np.column_stack([ndimage.convolve1d(np.eye(N)[j], kern, mode=mode) for j in range(N)])

T = matrix_1d(kern, N, 'constant')       # zero boundary
C = matrix_1d(kern, N, 'wrap')           # periodic boundary
R = matrix_1d(kern, N, 'reflect')        # reflexive boundary

fig, ax = plt.subplots(1, 3, figsize=(10, 3.5))
for a, M, name in zip(ax, (T, C, R), ('zero: Toeplitz', 'periodic: circulant', 'reflexive: Toeplitz
 ↪  + Hankel')):
    a.imshow(M, cmap='gray_r', vmin=0, vmax=0.5); a.set_title(name)
    a.set_xticks([]); a.set_yticks([]); a.grid(False)
fig.tight_layout()
```

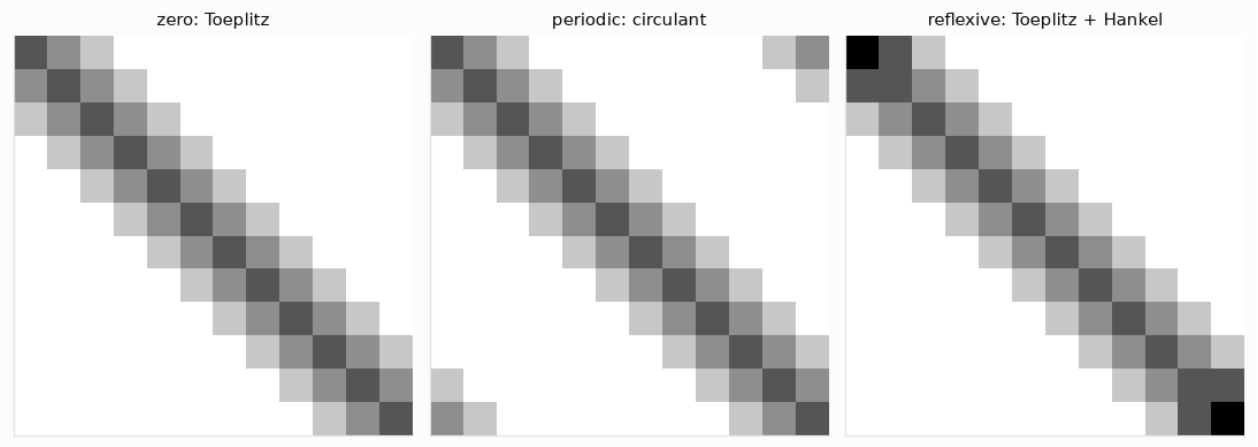


Figure 4.1: The blurring matrix of a one-dimensional signal under three boundary conditions.

- **Zero boundary.** The same values along every diagonal: a *Toeplitz* matrix. Light that falls off the ends is lost.
- **Periodic boundary.** What leaves on one side comes back on the other: the diagonals wrap around the corners. Every row is the previous one shifted by one step, cyclically. This is a *circulant* matrix.
- **Reflexive boundary.** What would leave is folded back: the corners gain entries along the *anti*-diagonals. Constant anti-diagonals define a *Hankel* matrix, so this one is Toeplitz plus Hankel.

In two dimensions the same three structures appear block-wise; the book names them BCCB, BTTB and so on. The names matter less than what each structure buys.

4.2 Periodic boundaries: the FFT diagonalises A

Fact: **every circulant matrix has the same eigenvectors**, the complex sinusoids of the discrete Fourier transform. Its eigenvalues are the Fourier transform of its first column. Check both statements on C:

```
eigvals_fft = fft(C[:, 0])
eigvals_num = np.linalg.eigvals(C)
print('eigenvalues from the FFT of the first column :', np.sort(eigvals_fft.real).round(4)[:6],
 ↪  '...')
print('eigenvalues from a general-purpose routine   :', np.sort(eigvals_num.real).round(4)[:6],
 ↪  '...')

x = rng.random(N)
print('C x  equals  ifft( fft(c) * fft(x) ):', np.allclose(C @ x, ifft(eigvals_fft * fft(x)).real))
```

```
eigenvalues from the FFT of the first column : [0.     0.     0.0595 0.0595 0.1111 0.1111] ...
eigenvalues from a general-purpose routine   : [-0.     0.     0.0595  0.0595  0.1111  0.1111] ...
C x  equals  ifft( fft(c) * fft(x) ): True
```

So multiplying by C costs two FFTs instead of a matrix product, and *solving* $Cx = b$ costs the same: divide by the eigenvalues instead of multiplying.

In two dimensions nothing changes except that `fft` becomes `fft2`. The first column of A, reshaped as an image, is the PSF **with its centre moved to pixel (0, 0)** and wrapped around the corners. Verify on an 8×8 image, where we can still afford the 64×64 matrix:

```
n = 8
psf = hno.gaussian_psf((5, 5), fwhm=2.5)

def matrix_2d(n, psf, mode):
    A = np.zeros((n * n, n * n))
    for j in range(n * n):
```

```
        unit = np.zeros(n * n); unit[j] = 1.0
        A[:, j] = ndimage.convolve(unit.reshape(n, n), psf, mode=mode).ravel()
    return A

A_per = matrix_2d(n, psf, 'wrap')
first_column = A_per[:, 0].reshape(n, n)            # PSF centred on pixel (0, 0), wrapped
K = fft2(first_column)                              # the n*n eigenvalues, as an n x n array

X = rng.random((n, n))
print('A x equals ifft2(K * fft2(X)):', np.allclose((A_per @ X.ravel()).reshape(n, n), ifft2(K *
 ↪  fft2(X)).real))
print('eigenvalues agree            :', np.allclose(np.sort(np.linalg.eigvals(A_per).real),
 ↪  np.sort(K.real.ravel())))

fig, ax = plt.subplots(1, 3, figsize=(8.4, 2.9))
for a, im, name in zip(ax, (psf, first_column, np.abs(K)), ('the PSF', 'first column of $A$',
 ↪  'eigenvalues $|K|$')):
    a.imshow(im); a.set_title(name); a.set_xticks([]); a.set_yticks([]); a.grid(False)
fig.tight_layout()
```

```
A x equals ifft2(K * fft2(X)): True
eigenvalues agree            : True
```

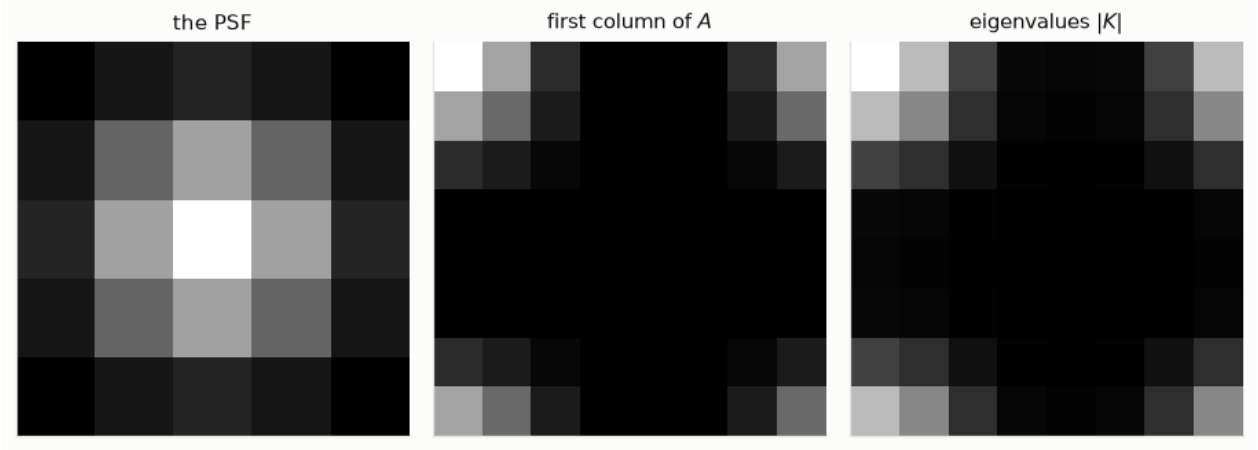


Figure 4.2: The PSF, the first column of the periodic blurring matrix, and its eigenvalues.

This is precisely what `hno.psf2otf` computes, and what notebook 1 used without proof. The array K has a name in optics: the *optical transfer function*.

4.3 Reflexive boundaries: the DCT diagonalises A

Notebook 3 showed that the reflexive boundary condition is the least wrong of the three. It has its own fast transform, on one condition: the PSF must be **symmetric in both directions** (left-right and up-down). Then A is diagonalised by the two-dimensional *discrete cosine transform*, the real-valued cousin of the FFT.

The eigenvalues are obtained from the first column of A again, with one extra division: if e_1 is the image with a single 1 in the top-left corner,

$$S = \frac{\mathrm{dct2}(A\,e_1)}{\mathrm{dct2}(e_1)}.$$

```
A_ref = matrix_2d(n, psf, 'reflect')
e1 = np.zeros((n, n)); e1[0, 0] = 1.0
dct2 = lambda a: dctn(a, type=2, norm='ortho')
idct2 = lambda a: idctn(a, type=2, norm='ortho')

S = dct2(ndimage.convolve(e1, psf, mode='reflect')) / dct2(e1)      # no matrix needed
print('A x equals idct2(S * dct2(X)):', np.allclose((A_ref @ X.ravel()).reshape(n, n), idct2(S *
 ↪  dct2(X))))
print('eigenvalues agree            :', np.allclose(np.sort(np.linalg.eigvalsh(A_ref)),
 ↪  np.sort(S.ravel())))
```

```
A x equals idct2(S * dct2(X)): True
eigenvalues agree            : True
```

At full size, with no matrix anywhere: blur a 256×256 image with reflexive boundaries, then undo the blur exactly by dividing in the DCT domain. (No noise here; this only checks the algebra.)

```
x_true = data['hubble'].astype(float)
psf9 = hno.gaussian_psf((9, 9), fwhm=3.0)
b = ndimage.convolve(x_true, psf9, mode='reflect')

e1 = np.zeros_like(x_true); e1[0, 0] = 1.0
S = dct2(ndimage.convolve(e1, psf9, mode='reflect')) / dct2(e1)
x_back = idct2(dct2(b) / S)
print(f'smallest eigenvalue {np.abs(S).min():.1e},  relative error of the recovered image
 ↪  {hno.rel_error(x_back, x_true):.1e}')
```

```
smallest eigenvalue 8.8e-07,  relative error of the recovered image 3.7e-11
```

4.4 Separable blurs: Kronecker products

A PSF is **separable** when it is the product of a vertical profile and a horizontal profile: $P = c\,r^{\mathsf{T}}$, a rank-one array. Blurring then splits into two independent 1-D blurs, columns first, rows next:

$$B = A_c\, X\, A_r^{\mathsf{T}},$$

where A_c and A_r are small 1-D blurring matrices, of the size of the image *side*. In vector form this is a **Kronecker product**. With NumPy's row-by-row flattening,

$$\operatorname{vec}(A_c X A_r^{\mathsf{T}}) = (A_c \otimes A_r)\operatorname{vec}(X).$$

(The book flattens column by column and therefore writes $A_r \otimes A_c$. Same mathematics, different bookkeeping.)

```
c = np.array([1.0, 4.0, 6.0, 4.0, 1.0]); c /= c.sum()          # vertical profile
r = np.array([1.0, 2.0, 1.0]); r /= r.sum()                    # horizontal profile
P = np.outer(c, r)                                             # a separable 5 x 3 PSF

Ac, Ar = matrix_1d(c, n, 'constant'), matrix_1d(r, n, 'constant')
A_sep = matrix_2d(n, P, 'constant')
print('A equals kron(Ac, Ar)      :', np.allclose(A_sep, np.kron(Ac, Ar)))
print('A x equals Ac X Ar^T       :', np.allclose((A_sep @ X.ravel()).reshape(n, n), Ac @ X @ Ar.T))
print('storage: A has', A_sep.size, 'entries; Ac and Ar together', Ac.size + Ar.size)
```

```
A equals kron(Ac, Ar)      : True
A x equals Ac X Ar^T       : True
storage: A has 4096 entries; Ac and Ar together 128
```

The gain is that **everything about A follows from the two small matrices**: its inverse, its singular value decomposition (notebook 5), and so its regularised solutions, for *any* boundary condition, at the cost of working on $n \times n$ matrices instead of $n^2 \times n^2$.

Is a given PSF separable? Compute its singular values: a separable PSF has exactly one that is not zero.

```
fmaj, fmin, pa, beta = data['sub_psf'][1]
candidates = {'Gaussian': hno.gaussian_psf((33, 33), 6.0),
              'Moffat (round)': hno.moffat_psf((33, 33), 6.0, beta=2.5),
              'our telescope': hno.moffat_psf((33, 33), fmaj, fmin, pa, beta)}

fig, ax = plt.subplots(figsize=(6.2, 3.2))
for (name, p), col in zip(candidates.items(), (hno.BLUE, hno.ORANGE, hno.AQUA)):
    sv = np.linalg.svd(p, compute_uv=False)
    ax.semilogy(np.arange(1, 13), sv[:12] / sv[0], 'o-', color=col, label=name, ms=4)
    print(f'{name:15s} second singular value / first = {sv[1] / sv[0]:.1e}')
ax.set_ylim(1e-18, 3); ax.set_xlabel('index'); ax.set_title('singular values of the PSF array',
 ↪  loc='left'); ax.legend();
```

```
Gaussian        second singular value / first = 3.9e-17
Moffat (round)  second singular value / first = 5.6e-02
our telescope   second singular value / first = 6.6e-02
```

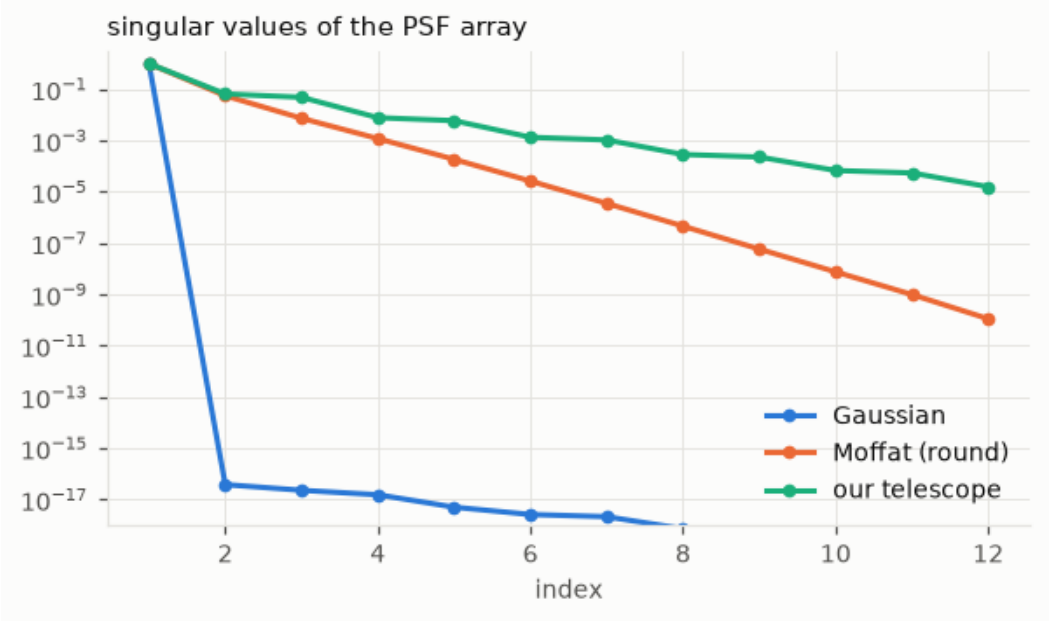


Figure 4.3: Singular values of three PSF arrays. A separable PSF has a single non-zero singular value.

The Gaussian is separable to machine precision: that is a special property of the Gaussian, and one reason for its popularity. A Moffat PSF is not, and neither is the real PSF of our telescope, although one separable term already captures most of it. The book explains how to use the best rank-one approximation when the PSF is nearly separable.

4.5 Summary: which tool for which problem

Boundary condition	Condition on the PSF	Structure of A	Fast tool
periodic	none	circulant blocks	FFT: `fft2`, `ifft2`
reflexive	symmetric in both directions	Toeplitz + Hankel blocks	DCT: `dctn`, `idctn`

Boundary condition	Condition on the PSF	Structure of A	Fast tool
any	separable, $P = c\,r^{\mathsf{T}}$	Kronecker product	two small matrices
zero	general	Toeplitz blocks	none: iterative methods (notebook 9)

In every fast case the pattern is the same three steps: **transform** the data, **divide** by the eigenvalues (with care: that is regularisation), **transform back**.

4.6 Realistic test data, and what to do at the edges

A warning before the experiments of the next notebooks. If you create test data by blurring an image *with periodic boundaries* and then deblur it *assuming periodic boundaries*, the edges are perfectly consistent and you will see no edge problem. Real pictures are not that kind. The honest recipe:

1. blur an image **larger** than the one you will work with;
2. **cut out** the central part, so that light really did come from outside;
3. add noise.

```
def make_test_problem(big, psf, noise, margin=64, seed=1):
    """Blur a large scene, keep its centre, add noise. Returns (x_true, b)."""
    inner = slice(margin, big.shape[0] - margin)
    blurred = ndimage.convolve(big, psf, mode='reflect')[inner, inner]
    return big[inner, inner], blurred + noise *
     ↪  np.random.default_rng(seed).standard_normal(blurred.shape)

psf17 = hno.gaussian_psf((17, 17), fwhm=4.0)
x_in, b_in = make_test_problem(data['hubble'].astype(float), psf17, noise=0.002)
```

Now deblur this honest test problem three ways, each with the simple "truncated inverse" of notebook 1 (invert where the eigenvalue is larger than τ, zero elsewhere):

- assuming **periodic** boundaries, with the FFT;
- assuming **reflexive** boundaries, with the DCT;
- assuming periodic boundaries, after **tapering** the data: fading the image smoothly towards its mean value near the edges, so that the left edge matches the right one. This is what our project's code does.

```
def truncated(coeffs, eig, tau):
    out = np.zeros_like(coeffs)
    keep = np.abs(eig) > tau
    out[keep] = coeffs[keep] / eig[keep]
    return out

def taper(b, width):
    """Fade the image to its mean over `width` pixels along every edge."""
    ramp = np.ones(b.shape[0])
    ramp[:width] = 0.5 - 0.5 * np.cos(np.pi * np.arange(width) / width)
    ramp[-width:] = ramp[:width][::-1]
    return b.mean() + (b - b.mean()) * np.outer(ramp, ramp)

tau = 0.05
m = x_in.shape[0]
big_psf = np.zeros((m, m)); big_psf[m//2 - 8:m//2 + 9, m//2 - 8:m//2 + 9] = psf17
K = hno.psf2otf(big_psf)
e1 = np.zeros((m, m)); e1[0, 0] = 1.0
S = dct2(ndimage.convolve(e1, psf17, mode='reflect')) / dct2(e1)
```

```
x_per = ifft2(truncated(fft2(b_in), K, tau)).real
x_ref = idct2(truncated(dct2(b_in), S, tau))
x_tap = ifft2(truncated(fft2(taper(b_in, 16)), K, tau)).real

core = slice(24, m - 24)                                    # judge away from the edges
for name, xr in (('periodic', x_per), ('reflexive', x_ref), ('periodic + taper', x_tap)):
    print(f'{name:17s} relative error: whole image {hno.rel_error(xr, x_in):.3f},   centre only
     ↪  {hno.rel_error(xr[core, core], x_in[core, core]):.3f}')
print(f'{"blurred data":17s} relative error: whole image {hno.rel_error(b_in, x_in):.3f},   centre
 ↪  only {hno.rel_error(b_in[core, core], x_in[core, core]):.3f}')

hno.show([x_per, x_ref, x_tap, x_in], ['periodic', 'reflexive', 'periodic + taper', 'true image'],
 ↪  same_scale=False, width=2.9);
```

```
periodic          relative error: whole image 0.232,   centre only 0.091
reflexive         relative error: whole image 0.015,   centre only 0.011
periodic + taper  relative error: whole image 0.172,   centre only 0.010
blurred data      relative error: whole image 0.014,   centre only 0.014
```

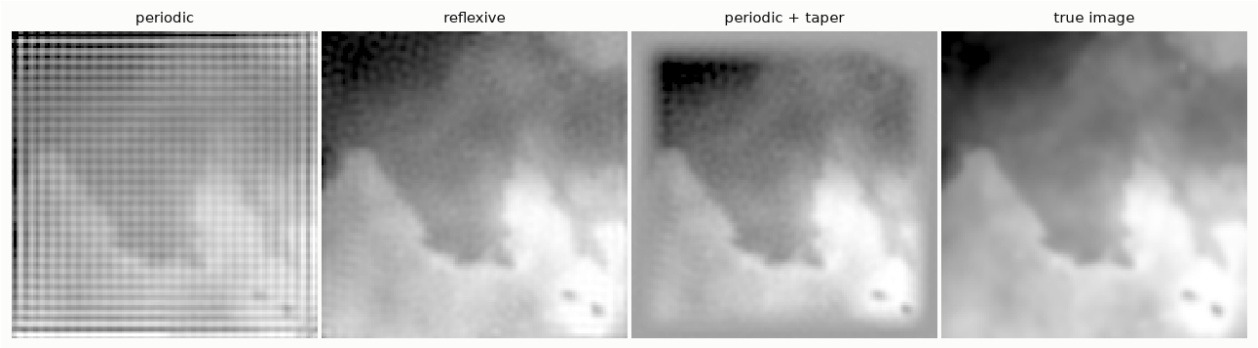


Figure 4.4: An honest test problem deblurred under three treatments of the edges.

Read the numbers together with the pictures.

- **Periodic**, untreated: ripples parallel to the edges invade the picture. They come from the jump between opposite edges, which the periodic model takes for a real feature of the scene and tries to "deblur".
- **Reflexive**: the ripples are gone, at no extra cost. This is the right choice whenever the PSF is symmetric in both directions.
- **Tapering** sacrifices a frame around the picture and saves the centre. It works with any PSF.

Our telescope PSF is an ellipse tilted at an arbitrary angle. It is symmetric through its centre, but not left-right and up-down separately, so the DCT route is closed to us:

```
k = hno.moffat_psf((33, 33), fmaj, fmin, pa, beta)
print('symmetric through the centre :', np.allclose(k, k[::-1, ::-1]))
print('symmetric left-right         :', np.allclose(k, k[:, ::-1]))
print(f'axis ratio {fmin / fmaj:.2f}, tilted by {pa:.0f} degrees')
```

```
symmetric through the centre : True
symmetric left-right         : False
axis ratio 0.87, tilted by 45 degrees
```

That is why the project works with the FFT and tapered edges, on images much larger than the PSF so that the sacrificed frame is cheap.

Your turn

1. **An asymmetric kernel.** In section 4.1 replace the kernel by `[3, 2, 1, 0, 0] / 6`. Which of the three matrices are still symmetric? Is the circulant one still diagonalised by the FFT? (Check as in 4.2.)
2. **Why the first column?** For the periodic case, explain in two sentences why knowing one column of A is knowing all of A.
3. **Break the DCT.** In section 4.3, replace `psf` by a PSF shifted by one pixel (use `np.roll`). Which of the two checks fails? Why does the FFT route not care?
4. **Cost.** Time `A_per @ X.ravel()` against `ifft2(K * fft2(X))` for `n = 8, 16, 32, 64` with `%timeit` (build A only while you can afford it). Plot both on logarithmic axes. What exponents do you find?
5. **Nearly separable.** Compute the best rank-one approximation of the telescope PSF from its SVD and display the difference with the PSF itself. How large is the error relative to the peak?
6. **The inverse crime.** In section 4.6, create the data with `ndimage.convolve(x_in, psf17, mode='wrap')` instead of `make_test_problem`, and deblur with the periodic model. What do the numbers say now, and why should you not believe them?
7. **Taper width.** Try `taper(b_in, w)` for `w` $= 4, 8, 16, 32$. What is the trade-off?

Where to read more

- Hansen et al. (2006), chapter 4.
- Gonzalez and Woods (2018), sections 4.4 to 4.6 (the discrete Fourier transform and its properties).

5 The SVD and spectral analysis

Companion notebook to Hansen, Nagy & O'Leary, ***Deblurring Images****, chapter 5.*

Notebook 1 showed *that* the naive solution fails. This one shows *why*, with the one tool that explains every linear inverse problem: the **singular value decomposition** (SVD).

What you will take away

1. The SVD splits a blur into independent channels, each with its own gain σ_i. Small gain means: that component of the scene barely reaches the data.
2. The naive solution divides each channel by its gain. Where the gain is below the noise, it divides noise by almost zero.
3. The **Picard plot** shows, on your own data, how many channels carry information. It is the first thing to draw for any inverse problem.
4. A regularised solution is a **filtered** sum over the channels.
5. For images with periodic boundaries, the channels are spatial frequencies and the SVD is the Fourier transform you already know.

```
import numpy as np
from numpy.fft import fft2, ifft2
import matplotlib.pyplot as plt
import hnotools as hno

hno.style()
rng = np.random.default_rng(0)
data = hno.load_sample()
```

5.1 The SVD of a blur

We start in one dimension, with the signal of notebook 1 and a 250×250 blurring matrix, because there everything can be computed and plotted. (The Gaussian kernel is the same as in notebook 1, except that we no longer cut it short at 25 samples; exercise 4 shows what the cut does.)

Any matrix can be written

$$A = U \Sigma V^{\mathsf{T}} = \sum_i \sigma_i \, u_i \, v_i^{\mathsf{T}},$$

with orthonormal *right singular vectors* v_i (patterns in the scene), orthonormal *left singular vectors* u_i (patterns in the data), and *singular values* $\sigma_1 \geq \sigma_2 \geq \ldots \geq 0$. Read it as a three-step machine: decompose the scene on the patterns v_i, multiply the i-th coefficient by σ_i, rebuild with the patterns u_i.

```
N = 250
s, signal = hno.lecture_signal(N)
A = hno.blur_matrix_1d(hno.gaussian_kernel_1d(4.0, 40), N)

U, sv, Vt = np.linalg.svd(A)
print('A equals U diag(sv) Vt:', np.allclose(A, U @ np.diag(sv) @ Vt))
print(f'largest singular value {sv[0]:.3f}, smallest {sv[-1]:.1e}, ratio (condition number) {sv[0]
 ↪  / sv[-1]:.1e}')

fig, ax = plt.subplots(1, 2, figsize=(11, 3.2), gridspec_kw={'width_ratios': [1, 1.4]})
```

```
ax[0].semilogy(np.arange(1, N + 1), sv, color=hno.BLUE)
ax[0].set_xlabel('index $i$'); ax[0].set_title('singular values $\\sigma_i$', loc='left')
for i, col in zip((0, 4, 19, 79), (hno.INK2, hno.BLUE, hno.ORANGE, hno.AQUA)):
    ax[1].plot(s, Vt[i] + 0.25 * (3 - (0, 4, 19, 79).index(i)), color=col, lw=1.2)
    ax[1].text(s[-1] + 0.1, 0.25 * (3 - (0, 4, 19, 79).index(i)), f'$v_{{{i + 1}}}$', va='center')
ax[1].set_yticks([]); ax[1].set_xlim(0, s[-1] + 0.9)
ax[1].set_title('four right singular vectors (shifted vertically)', loc='left')
fig.tight_layout()
```

```
A equals U diag(sv) Vt: True
largest singular value 0.999, smallest 8.4e-18, ratio (condition number) 1.2e+17
```

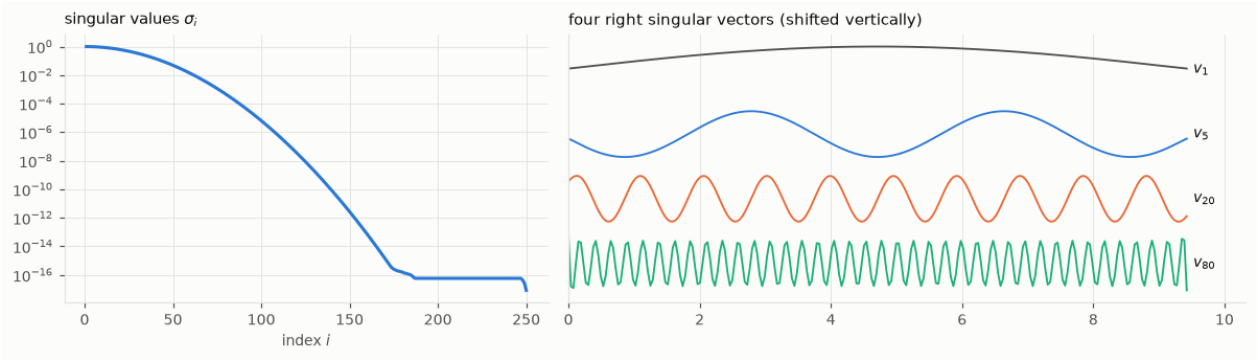


Figure 5.1: Singular values of a one-dimensional blur, and four of its right singular vectors.

Two facts, true for every blur:

- the singular values **decay** steadily, over many orders of magnitude, with no gap, until they reach the rounding level of the computer near 10^{-17};
- the singular vectors **oscillate more and more** as the index grows.

Together they say: *a blur transmits smooth patterns well and fine patterns badly.* That is all a blur is.

5.2 The naive solution, channel by channel

Since U and V are orthogonal, the solution of $Ax = b$ is

$$x_{\text{naive}} = \sum_i \frac{u_i^{\mathsf{T}} b}{\sigma_i}\, v_i.$$

Each coefficient of the data, $u_i^{\mathsf{T}} b$, is divided by the gain of its channel. With noisy data $b = b_{\text{exact}} + e$, each coefficient is the sum of a signal part and a noise part. Plot both, and the singular values:

```
noise = 2e-4
b_exact = A @ signal
b = b_exact + noise * rng.standard_normal(N)

coef_exact = np.abs(U.T @ b_exact)
coef = np.abs(U.T @ b)
idx = np.arange(1, N + 1)

fig, ax = plt.subplots(figsize=(7.5, 3.8))
ax.semilogy(idx, sv, color=hno.INK2, lw=1.2, ls=(0, (4, 2)), label='singular values $\\sigma_i$')
ax.semilogy(idx, coef_exact, '.', color=hno.AQUA, ms=4, label='exact data $|u_i^T b_{exact}|$')
ax.semilogy(idx, coef, '.', color=hno.BLUE, ms=4, label='noisy data $|u_i^T b|$')
```

```
ax.semilogy(idx, coef / sv, '.', color=hno.ORANGE, ms=4, label='naive coefficients $|u_i^T b| /
 ↪  \\sigma_i$')
ax.axhline(noise, color=hno.INK2, lw=0.8)
ax.text(N, noise * 1.6, 'noise level', ha='right', fontsize=8, color=hno.INK2)
ax.set_ylim(1e-19, 1e14); ax.set_xlabel('index $i$')
ax.set_title('The Picard plot', loc='left'); ax.legend(loc='lower left', fontsize=8);
```

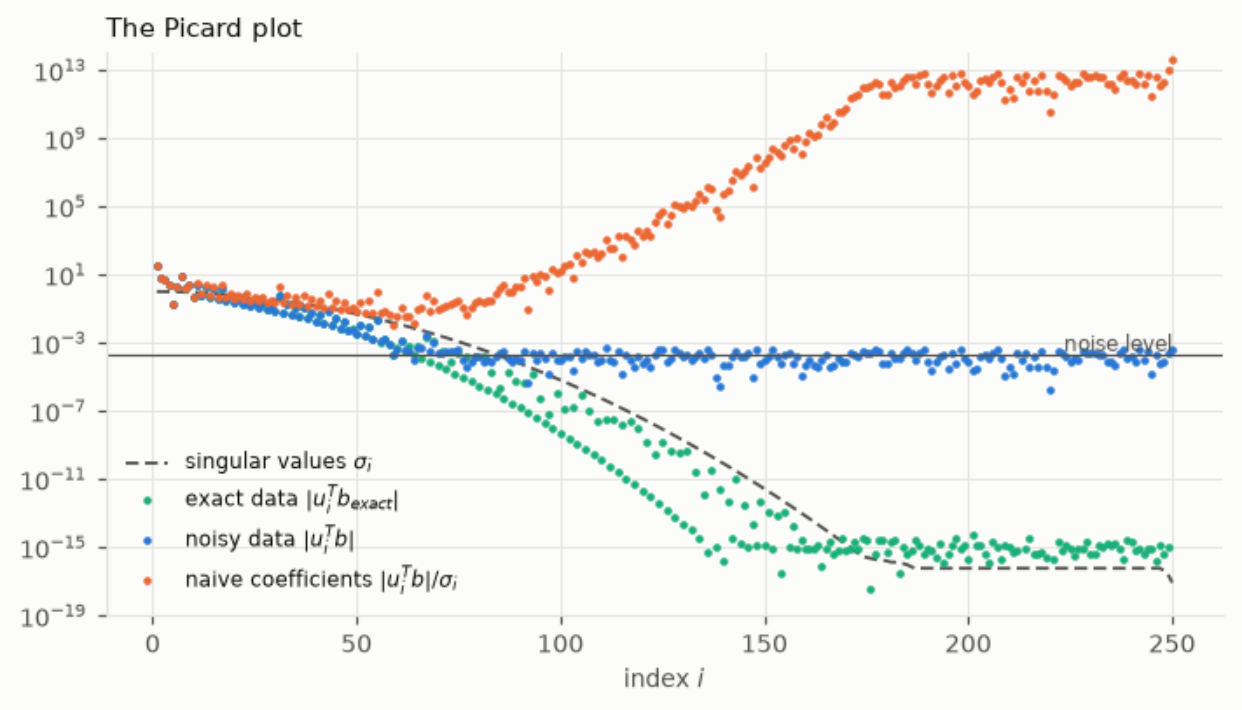


Figure 5.2: The Picard plot of the one-dimensional problem.

Figure 5.2 is the **Picard plot**. Take the time to read it.

- **Green, exact data.** The coefficients fall *faster* than the singular values (dashed). This is the *discrete Picard condition*: it means the exact problem has a decent solution, because $|u_i^{\mathsf{T}} b|/\sigma_i$ stays bounded.
- **Blue, noisy data.** The coefficients follow the green ones at first, then **level off at the noise level**. From there on, they *are* noise: the signal that used to be there is far below.
- **Orange, what the naive solution uses.** Fine on the left. On the right, noise divided by smaller and smaller σ_i: the coefficients grow without limit. The naive solution is dominated by its last, meaningless terms.

The index where the blue points meet the noise floor is the number of channels through which the data tell us anything.

5.3 Spectral filtering

The cure is now obvious: keep the terms that carry information, damp or drop the others. Every method of chapter 6 has the form

$$x_{\text{filt}} = \sum_i \phi_i \, \frac{u_i^{\mathsf{T}} b}{\sigma_i} \, v_i,$$

with *filter factors* ϕ_i close to 1 for the good channels and close to 0 for the bad ones. The simplest choice keeps the first k terms and drops the rest: the **truncated SVD** (TSVD).

```python
def tsvd(U, sv, Vt, b, k):
    """Truncated SVD solution: the first k terms of the naive sum."""
    return Vt[:k].T @ ((U[:, :k].T @ b) / sv[:k])

ks = np.arange(1, 140)
errors = [hno.rel_error(tsvd(U, sv, Vt, b, k), signal) for k in ks]
k_best = int(ks[np.argmin(errors)])
k_floor = int(np.argmax(coef < 3 * noise)) + 1              # first coefficient at the noise floor
print(f'best k = {k_best} (error {min(errors):.3f});   first coefficient within 3x of the noise
 ↪  floor: i = {k_floor}')

fig, ax = plt.subplots(1, 2, figsize=(11, 3.3), gridspec_kw={'width_ratios': [1, 1.5]})
ax[0].semilogy(ks, errors, color=hno.BLUE); ax[0].plot(k_best, min(errors), 'o', color=hno.ORANGE)
ax[0].set_xlabel('number of terms $k$'); ax[0].set_title('relative error of the TSVD solution',
 ↪  loc='left')
ax[1].plot(s, signal, color=hno.INK2, lw=1.0, label='true signal')
for k, col in ((8, hno.AQUA), (k_best, hno.BLUE), (k_best + 12, hno.ORANGE)):
    ax[1].plot(s, tsvd(U, sv, Vt, b, k), color=col, lw=1.3, label=f'$k$ = {k}')
ax[1].set_ylim(-0.5, 4.2); ax[1].set_title('three TSVD solutions', loc='left');
 ↪  ax[1].legend(loc='upper left', fontsize=8, ncol=2)
fig.tight_layout()
```

```
best k = 71 (error 0.044);   first coefficient within 3x of the noise floor: i = 59
```

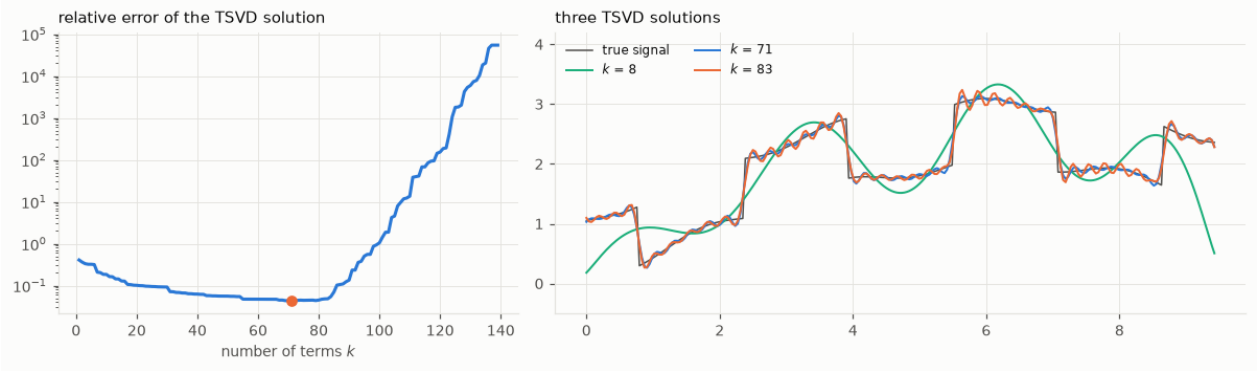


Figure 5.3: Error of the truncated SVD against the number of terms, and three solutions.

The U-curve of notebook 1 (Figure 1.5) again, now explained term by term.

- Too few terms: only smooth patterns are allowed, and the steps of the signal are rounded off (*regularisation error*).
- Too many: noise-dominated terms enter with huge weights (*perturbation error*).
- The best k sits close to where the Picard plot said the data hit the noise floor. **You can read a good regularisation off the Picard plot without knowing the true signal.**

Note what even the best solution cannot do: the steps stay rounded, with small oscillations next to them. Sharp steps need the fine patterns, and the data no longer contain them.

5.4 Two dimensions: the patterns are images

For an image, the singular vectors are images too. With a separable blur (notebook 4) they are products of the 1-D patterns: if $A = A_c \otimes A_r$, its singular vectors are $v_i^c \otimes v_j^r$ and its singular values $\sigma_i^c \, \sigma_j^r$. Here are a few of these *basis images*, for a Gaussian blur on a 32×32 image:

```
n = 32
A1 = hno.blur_matrix_1d(hno.gaussian_kernel_1d(1.5, 5), n)
U1, s1, V1t = np.linalg.svd(A1)

pairs = [(0, 0), (1, 0), (1, 2), (4, 3), (9, 9), (20, 22)]
fig, ax = plt.subplots(1, len(pairs), figsize=(12, 2.4))
for a, (i, j) in zip(ax, pairs):
    a.imshow(np.outer(V1t[i], V1t[j]), cmap='gray')
    a.set_title(f'$\\sigma$ = {s1[i] * s1[j]:.1e}', fontsize=9)
    a.set_xticks([]); a.set_yticks([]); a.grid(False)
fig.tight_layout()
```

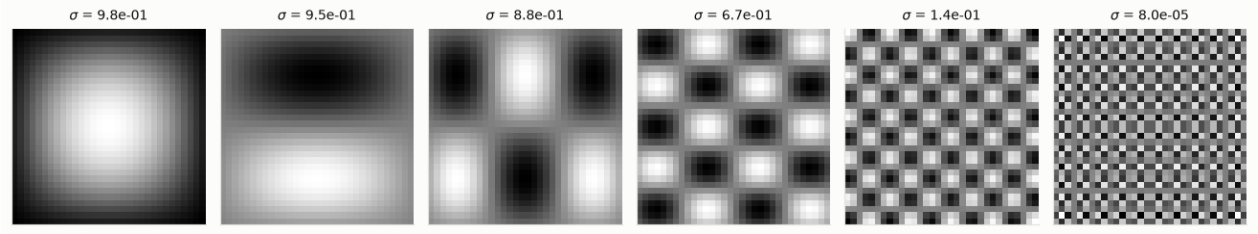


Figure 5.4: Six basis images of a separable Gaussian blur, with their singular values.

Same story as in 1-D: large singular values go with smooth images, small ones with fine checkerboards.

With **periodic** boundaries we do not even need to compute an SVD. Notebook 4 showed that A is diagonalised by the Fourier transform: the patterns are the sinusoids of each spatial frequency, and the singular values are the moduli of the transfer function,

$$\sigma_i \longleftrightarrow |K(\nu)|.$$

So for images, "index i" becomes "spatial frequency ν", and the Picard plot becomes the spectrum plot of notebook 1 (Figure 1.4). Here it is again in Picard form, the 65 536 frequencies sorted by decreasing $|K|$:

```
x_true = data['hubble'].astype(float)
psf = hno.gaussian_psf(x_true.shape, fwhm=6.0)
K = hno.psf2otf(psf)
noise2 = 0.005
B_exact = fft2(hno.blur(x_true, psf))
B = B_exact + fft2(noise2 * rng.standard_normal(x_true.shape))

order = np.argsort(-np.abs(K).ravel())
def running_median(a, w=400):
    a = a[: len(a) // w * w].reshape(-1, w)
    return np.median(a, axis=1)

i_axis = running_median(np.arange(K.size, dtype=float))
fig, ax = plt.subplots(figsize=(7.5, 3.6))
ax.semilogy(i_axis, running_median(np.abs(K).ravel()[order]), color=hno.INK2, lw=1.2, ls=(0, (4,
 ↪  2)), label='$|K|$, sorted')
ax.semilogy(i_axis, running_median(np.abs(B_exact).ravel()[order]), color=hno.AQUA, label='exact
 ↪  data')
ax.semilogy(i_axis, running_median(np.abs(B).ravel()[order]), color=hno.BLUE, label='noisy data')
ax.set_ylim(1e-18, 1e4); ax.set_xlabel('index (frequencies sorted by decreasing $|K|$)')
ax.set_title('Picard plot of the image problem of notebook 1', loc='left'); ax.legend(loc='upper
 ↪  right');
usable = int(np.sum(running_median(np.abs(B).ravel()[order]) > 2 *
 ↪  np.median(np.abs(B).ravel()[order][-20000:])) * 400)
print(f'frequencies clearly above the noise floor: about {usable} of {K.size}  ({usable /
 ↪  K.size:.1%})')
```

```
frequencies clearly above the noise floor: about 2800 of 65536  (4.3%)
```

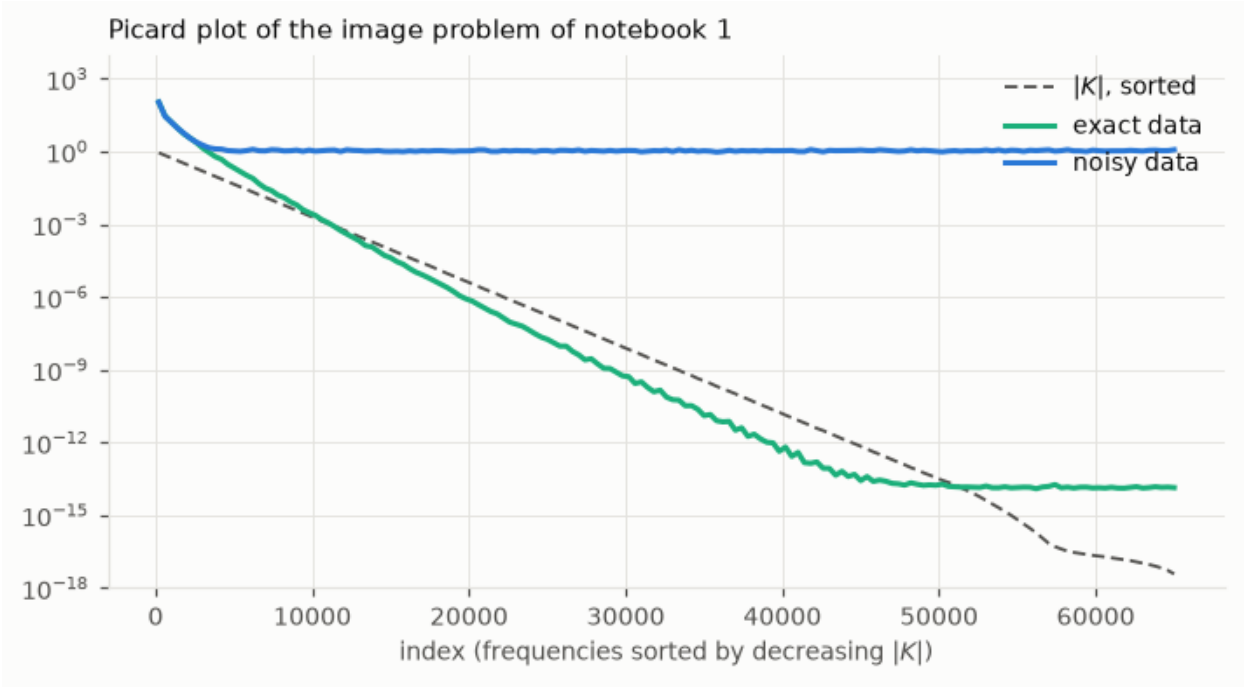


Figure 5.5: Picard plot of the image problem of chapter 1.

Out of 65 536 unknowns, the data determine a few thousand. Everything else in a deblurred image comes from the *method*, not from the data. This is the number to keep in mind when someone shows you a spectacularly sharp restoration.

5.5 Real data: what stacking buys

For real data there is no "exact" curve. But we can still draw the two things that matter: the spectrum of the image, and the spectrum of its noise, which we measure by differencing as in notebook 2. Where the first sinks into the second, the information ends.

We do this for one exposure of our telescope and for the average of all 45, on a corner of the field that contains nebula but no bright star. (A star is a point, so its spectrum is flat and would hide what we want to see.)

```
cube = data['cube'].astype(float)[:, 2:66, 2:66]           # a 64 x 64 star-free corner, 45
 ↪  exposures
scale = float(data['pixel_scale'])

def taper(b, width=8):
    ramp = np.ones(b.shape[0])
    ramp[:width] = 0.5 - 0.5 * np.cos(np.pi * np.arange(width) / width)
    ramp[-width:] = ramp[:width][::-1]
    return b.mean() + (b - b.mean()) * np.outer(ramp, ramp)

cases = (('one exposure', cube[0], (cube[0] - cube[1]) / np.sqrt(2), hno.ORANGE),
         ('mean of 45 exposures', cube.mean(axis=0), (cube[0::2].mean(axis=0) -
 ↪  cube[1::2].mean(axis=0)) / 2, hno.BLUE))

fig, ax = plt.subplots(figsize=(7.5, 3.8))
for name, img, noise_img, col in cases:
```

```
    nu, spec = hno.radial_profile(fft2(taper(img)))
    _, nspec = hno.radial_profile(fft2(taper(noise_img)))
    knee = nu[1 + np.argmax(spec[1:] < 2 * nspec[1:])]
    ax.semilogy(nu / scale, spec, color=col, label=name)
    ax.semilogy(nu / scale, nspec, color=col, lw=1.0, ls=(0, (2, 2)))
    print(f'{name:22s} noise level {np.median(nspec[5:]):.2e},  image within 2x of its noise from
     ↪ {knee / scale:.2f} cycles/arcsec')
ax.plot([], [], color=hno.INK2, lw=1.0, ls=(0, (2, 2)), label='its noise (measured by
 ↪ differencing)')
ax.set_xlabel('spatial frequency (cycles / arcsecond)')
ax.set_title('Spectrum of real images, and of their noise', loc='left'); ax.legend();
```

```
one exposure           noise level 1.12e-02,  image within 2x of its noise from 0.10 cycles/arcsec
mean of 45 exposures   noise level 1.65e-03,  image within 2x of its noise from 0.20 cycles/arcsec
```

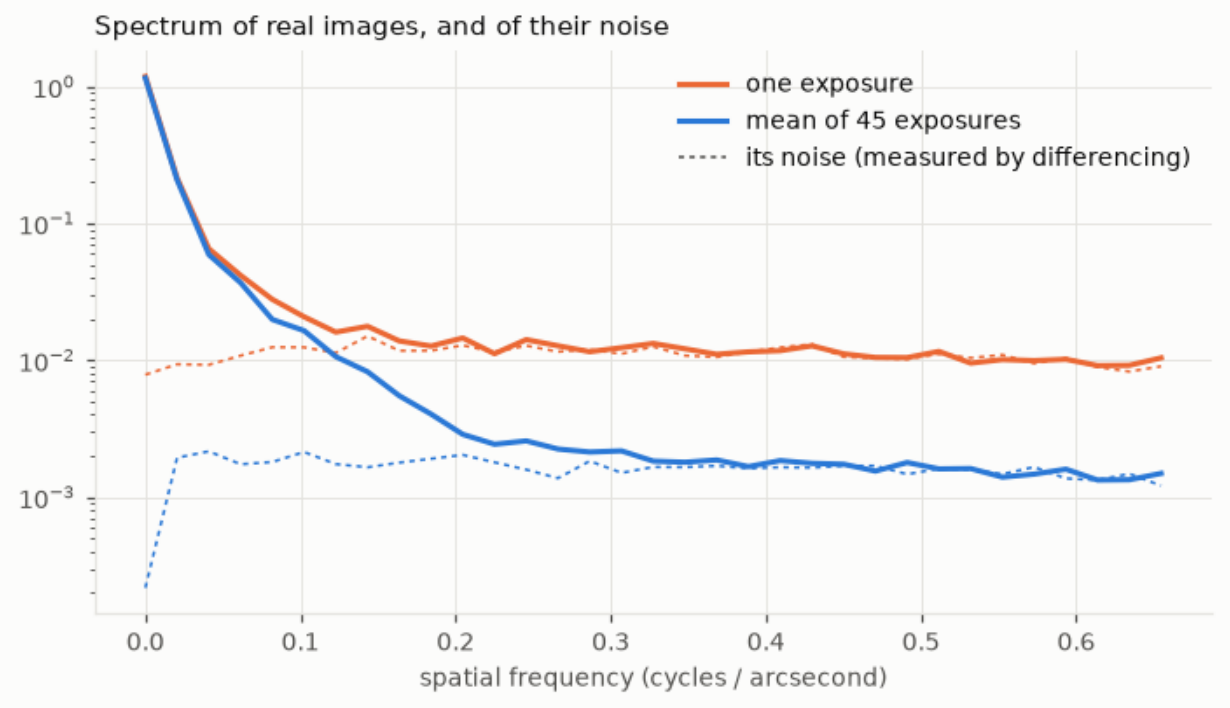


Figure 5.6: Spectrum of one exposure and of the mean of 45 (solid lines), and of their noise (dotted lines).

Averaging 45 exposures lowers the noise by a factor close to $\sqrt{45} \approx 7$ and leaves the signal where it was. The point where the image sinks into its noise moves from about 0.10 to about 0.20 cycles per arcsecond. In two dimensions, twice the radius means **four times as many channels carrying information**, so four times as much of the scene that can honestly be restored. No deblurring algorithm can do this; only more photons can.

This is why our project never deblurs single exposures. Notebook 8 shows how to combine them so that not one of these newly usable channels is wasted.

Your turn

1. **Counting sign changes.** For the 1-D problem, count the sign changes of v_i for $i = 1, \dots, 60$ and plot the count against i. What simple law do you find?
2. **More noise, fewer channels.** Redo sections 5.2 and 5.3 with noise 10^{-2} and 10^{-6}. Tabulate the noise level, the index where the coefficients reach the floor, and the best k. How many channels does a factor 100 in noise cost?

3. **A wider blur.** Replace the kernel by `hno.gaussian_kernel_1d(8.0, 80)`. What happens to the singular values, and to the best k?
4. **A kernel cut short.** Use the kernel of notebook 1, `hno.gaussian_kernel_1d(4.0, 12)`, which stops at 25 samples. The singular values now stall on a plateau near 10^{-3}. Why? (Cutting a kernel multiplies it by a box; what does that do to its Fourier transform?) Is the problem easier or harder to invert, and is that good news?
5. **Picard violated.** Take b = pure noise (no signal at all) and draw its Picard plot. How would you recognise, on real data, that an image contains nothing but noise?
6. **2-D basis images.** In section 5.4, reconstruct the top-left 32×32 corner of `x_true` from only the 100 basis images with the largest $\sigma_i^c \sigma_j^r$. (Hint: the coefficients are `V1t @ X @ V1t.T`.) What is lost?
7. **How many exposures?** From section 5.5, estimate how many exposures would be needed to push that point to 0.4 cycles per arcsecond. Is it a realistic observing programme?

Where to read more

- Hansen et al. (2006), chapter 5.
- Talbot (2024), section 2.2, on conditioning.
- For the SVD itself: Strang (2016), the chapter on the singular value decomposition.

6 Regularisation by spectral filtering

Companion notebook to Hansen, Nagy & O'Leary, ***Deblurring Images****, chapter 6, with the optimisation point of view of H. Talbot's* ***Lecture Notes on Optimisation****, chapter 3.*

Notebook 5 ended on a formula: a regularised solution is a filtered sum, $x_{\text{filt}} = \sum_i \phi_i \, (u_i^{\mathsf{T}} b/\sigma_i) \, v_i$. This notebook is about choosing the filter factors ϕ_i, and above all about choosing **how much** to filter when the true image is unknown.

What you will take away

1. Two classical filters: the truncated SVD and **Tikhonov** regularisation.
2. Tikhonov regularisation is also the solution of a penalised least-squares problem. Filtering and penalising are two views of the same thing.
3. The error of a regularised solution has two parts that pull in opposite directions.
4. Three rules for choosing the regularisation parameter without the truth: the discrepancy principle, generalised cross-validation, and the L-curve.
5. A fourth rule, the one our project uses: declare the resolution you will deliver, then verify it.

```
import numpy as np
from numpy.fft import fft2, ifft2
from scipy import optimize
import matplotlib.pyplot as plt
import hnotools as hno

hno.style()
rng = np.random.default_rng(0)
data = hno.load_sample()
```

6.1 Two filters

- **Truncated SVD**: keep the k largest singular values, drop the others. $\phi_i = 1$ if $\sigma_i > \tau$, else 0.
- **Tikhonov**: damp smoothly, $\phi_i = \dfrac{\sigma_i^2}{\sigma_i^2 + \alpha^2}$.

Both leave the large singular values alone ($\phi \approx 1$) and suppress the small ones ($\phi \approx 0$). The parameter, τ or α, is the singular value at which the filter switches.

```
sig = np.logspace(-6, 0, 400)
alpha = 1e-3
fig, ax = plt.subplots(figsize=(6.4, 3.0))
ax.semilogx(sig, (sig > alpha).astype(float), color=hno.ORANGE, label='truncated SVD, $\\tau =
↪  10^{-3}$')
ax.semilogx(sig, sig**2 / (sig**2 + alpha**2), color=hno.BLUE, label='Tikhonov, $\\alpha =
↪  10^{-3}$')
ax.set_xlabel('singular value $\\sigma_i$'); ax.set_title('filter factors $\\phi_i$', loc='left');
↪  ax.legend();
```

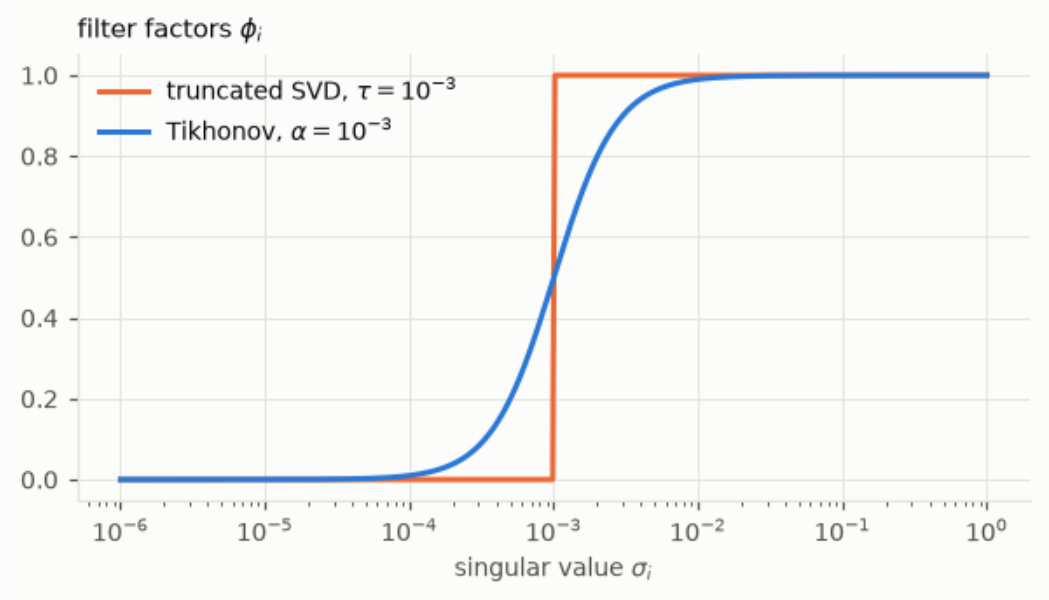


Figure 6.1: Filter factors of the truncated SVD and of Tikhonov regularisation.

Note the Tikhonov factor for a small singular value: $\phi_i/\sigma_i \approx \sigma_i/\alpha^2$, which goes to zero instead of infinity. The division by a tiny number has been replaced by a multiplication by a tiny number.

6.2 Tikhonov regularisation is a least-squares problem

The lecture notes introduce Tikhonov regularisation from a different door. Instead of solving $Ax = b$, look for the x that explains the data **and** stays small:

$$x_\alpha = \arg\min_x \ \|Ax - b\|_2^2 \ + \ \alpha^2 \|x\|_2^2.$$

The first term asks for fidelity to the data, the second penalises the wild oscillations that the naive solution is made of, and α sets the exchange rate between the two. Setting the gradient to zero gives a linear system, the *regularised normal equations*,

$$(A^\mathsf{T} A + \alpha^2 I)\, x = A^\mathsf{T} b,$$

whose matrix is symmetric positive definite, hence always invertible.

Is this the same solution as the Tikhonov *filter*? Check on the 1-D problem of notebook 5, where we have both the matrix and its SVD.

```
N = 250
s, signal = hno.lecture_signal(N)
A = hno.blur_matrix_1d(hno.gaussian_kernel_1d(4.0, 40), N)
b = A @ signal + 2e-4 * rng.standard_normal(N)
U, sv, Vt = np.linalg.svd(A)

alpha = 3e-3
x_filter = Vt.T @ (sv / (sv**2 + alpha**2) * (U.T @ b))                    # filtered SVD sum
x_normal = np.linalg.solve(A.T @ A + alpha**2 * np.eye(N), A.T @ b)        # normal equations
print('largest difference between the two solutions:', np.abs(x_filter - x_normal).max())
print(f'relative error of the Tikhonov solution: {hno.rel_error(x_filter, signal):.3f}')
```

```
largest difference between the two solutions: 2.196867132653324e-10
relative error of the Tikhonov solution: 0.044
```

Identical. Insert $A = U\Sigma V^{\mathsf{T}}$ in the normal equations and the filter factors $\sigma_i^2/(\sigma_i^2 + \alpha^2)$ fall out in two lines (exercise 1).

The two views are worth keeping side by side.

- The **filter** view is the one that *computes fast*, whenever a fast transform diagonalises A.
- The **optimisation** view is the one that *generalises*: change the penalty, add constraints, replace the square by a robust loss, and the minimisation problem still makes sense when no SVD can help. Notebooks 7 and 9 go down that road, and so does our project.

6.3 Deblurring an image with both filters

For images we use a fast transform. The Gaussian PSF is symmetric, so we take reflexive boundaries and the cosine transform (notebook 4); the eigenvalues S of the blur play the role of the singular values.

In this section and the next two, the test data are created **with the same reflexive boundaries that the solver assumes**. Notebook 4 called this the "inverse crime", and it is one. We commit it on purpose: the rules of section 6.5 are derived for data that obey the model exactly, plus white noise, and we first want to see them work under their own assumptions. Section 6.5 ends with what happens when the assumptions fail.

```
from scipy import ndimage

psf = hno.gaussian_psf((25, 25), fwhm=6.0)
noise_level = 0.002
x_true = data['hubble'].astype(float)
b_img = ndimage.convolve(x_true, psf, mode='reflect') + noise_level *
 ↪  rng.standard_normal(x_true.shape)

S = hno.dct_eigenvalues(psf, b_img.shape)
B = hno.dct2(b_img)

def tsvd_image(B, S, tau):
    X = np.zeros_like(B); keep = np.abs(S) > tau
    X[keep] = B[keep] / S[keep]
    return hno.idct2(X)

def tikhonov_image(B, S, alpha):
    return hno.idct2(S * B / (S**2 + alpha**2))

params = np.logspace(-4, 0, 60)
err_tsvd = [hno.rel_error(tsvd_image(B, S, p), x_true) for p in params]
err_tikh = [hno.rel_error(tikhonov_image(B, S, p), x_true) for p in params]
tau_best, alpha_best = params[np.argmin(err_tsvd)], params[np.argmin(err_tikh)]

fig, ax = plt.subplots(figsize=(6.4, 3.2))
ax.loglog(params, err_tsvd, color=hno.ORANGE, label='truncated SVD'); ax.loglog(params, err_tikh,
 ↪  color=hno.BLUE, label='Tikhonov')
ax.axhline(hno.rel_error(b_img, x_true), color=hno.INK2, lw=1.0, ls=(0, (4, 2)))
ax.text(params[0], hno.rel_error(b_img, x_true) * 1.07, 'blurred data', fontsize=8, color=hno.INK2)
ax.set_ylim(0.006, 1); ax.set_xlabel('parameter ($\\tau$ or $\\alpha$)'); ax.set_ylabel('relative
 ↪  error'); ax.legend()
ax.set_title('Error against the regularisation parameter', loc='left');
print(f'blurred data              error {hno.rel_error(b_img, x_true):.4f}')
print(f'truncated SVD, best tau   {tau_best:.3f}: error {min(err_tsvd):.4f}')
print(f'Tikhonov,      best alpha {alpha_best:.3f}: error {min(err_tikh):.4f}')
```

```
blurred data              error 0.0240
truncated SVD, best tau   0.070: error 0.0098
Tikhonov,      best alpha 0.060: error 0.0105
```

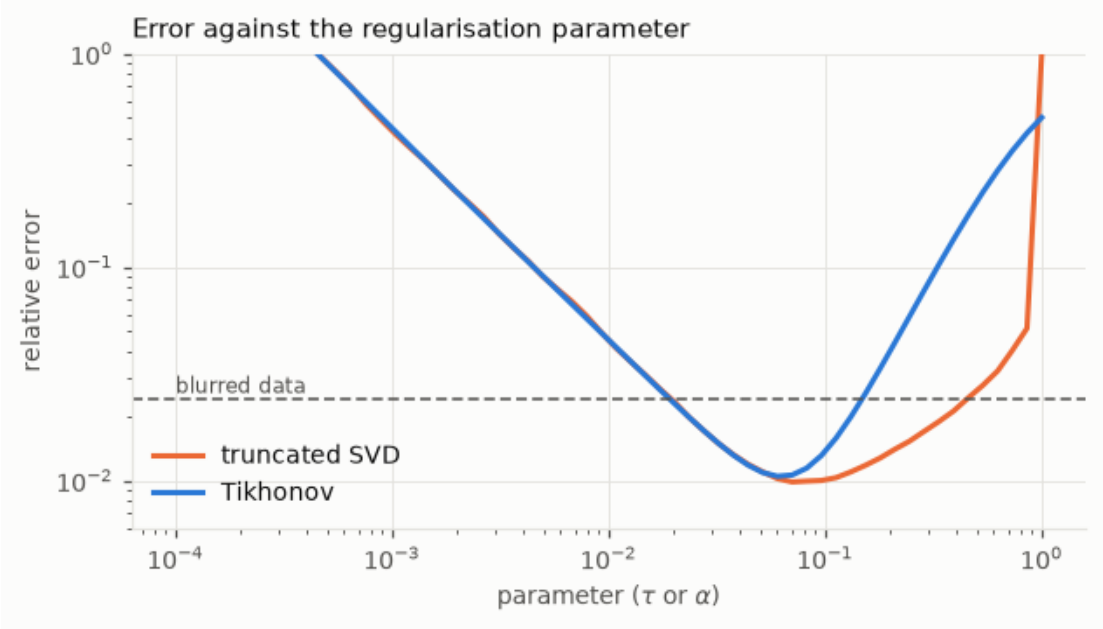


Figure 6.2: Relative error of the two filters against their parameter.

```
hno.show([b_img, tsvd_image(B, S, tau_best), tikhonov_image(B, S, alpha_best), x_true],
         ['blurred + noise', 'truncated SVD (best)', 'Tikhonov (best)', 'true image'],
 ↪  same_scale=True, width=2.9);
```

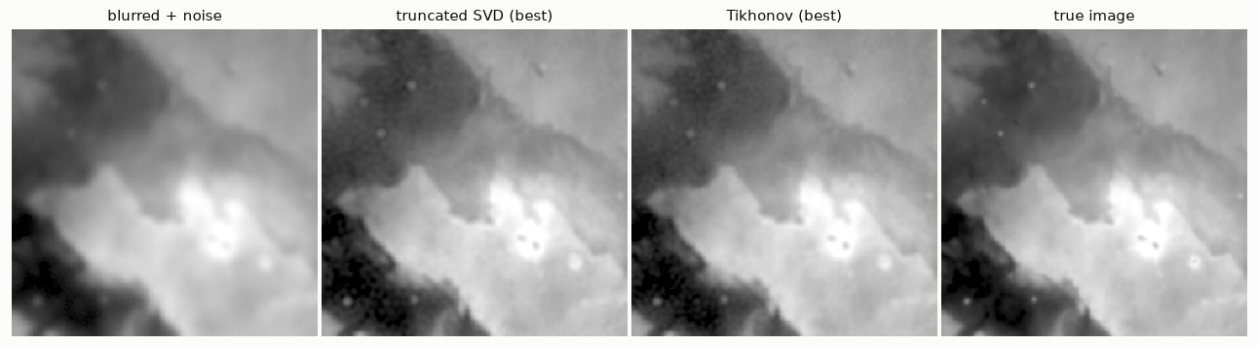


Figure 6.3: The data, the best result of each filter, and the true image.

Both filters do about equally well at their best setting, and both curves have the U shape. For too large a parameter Tikhonov loses sooner, because its filter starts damping well before $\sigma_i = \alpha$. Its advantages lie elsewhere: no ripples from an abrupt cut, and a formulation as a minimisation problem, which is the one that generalises.

6.4 Two errors that pull in opposite directions

With $b = Ax + e$, the filtered solution differs from the truth by

$$x_{\text{filt}} - x \;=\; \underbrace{\sum_i (\phi_i - 1)\,(v_i^{\mathsf{T}} x)\,v_i}_{\text{regularisation error}} \;+\; \underbrace{\sum_i \phi_i \,\frac{u_i^{\mathsf{T}} e}{\sigma_i}\, v_i}_{\text{perturbation error}} .$$

- The **regularisation error** is the part of the true image that the filter removed. It grows with α.
- The **perturbation error** is the noise that the filter let through, amplified. It shrinks with α.

In an experiment we know x and e, so we can compute each part:

```
X_true = hno.dct2(x_true)
E = B - S * X_true                                  # the noise, in the transform domain
reg_err, pert_err = [], []
for a in params:
    phi = S**2 / (S**2 + a**2)
    reg_err.append(np.linalg.norm((phi - 1) * X_true) / np.linalg.norm(X_true))
    pert_err.append(np.linalg.norm(phi * E / S) / np.linalg.norm(X_true))

fig, ax = plt.subplots(figsize=(6.4, 3.2))
ax.loglog(params, reg_err, color=hno.AQUA, label='regularisation error')
ax.loglog(params, pert_err, color=hno.ORANGE, label='perturbation error')
ax.loglog(params, err_tikh, color=hno.BLUE, label='total error')
ax.set_ylim(1e-3, 10); ax.set_xlabel('$\\alpha$'); ax.legend(); ax.set_title('Tikhonov: the two
↪  parts of the error', loc='left');
```

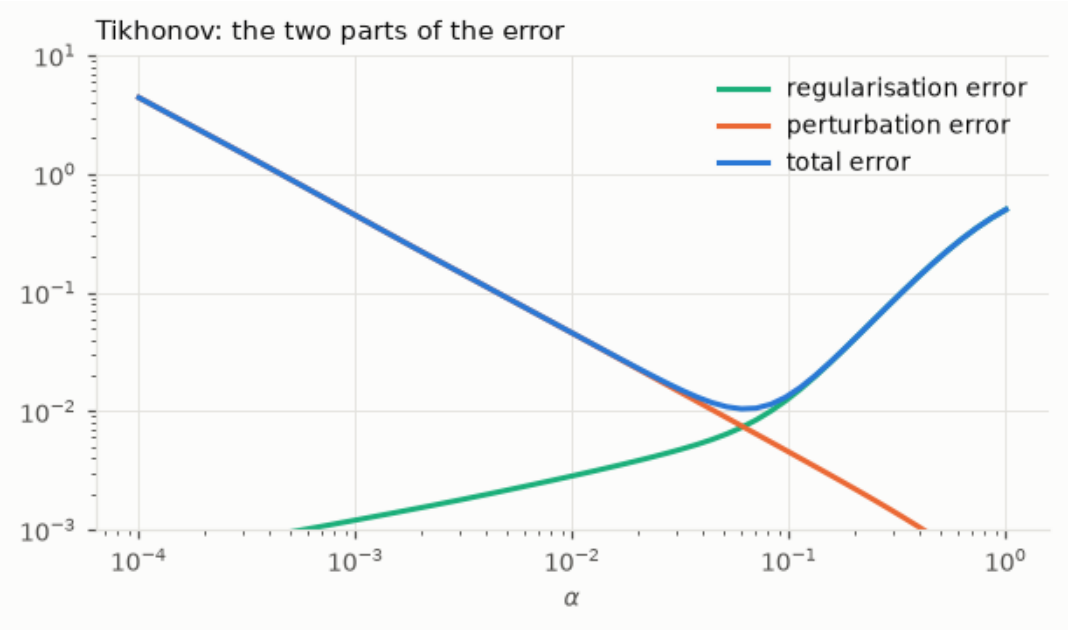


Figure 6.4: Regularisation error, perturbation error, and their total, for Tikhonov regularisation.

In Figure 6.4 the best α is close to where the two curves cross. Every parameter-choice rule below is an attempt to locate that crossing **without seeing either curve**.

6.5 Choosing α without knowing the truth

All three rules use only the data, through the **residual** $r_\alpha = b - Ax_\alpha$, whose coefficients in the transform domain are $(1-\phi_i)\,\hat{b}_i$.

Discrepancy principle. If we knew the truth, the residual would be the noise, of squared norm about $N\sigma^2$ for N pixels of noise σ. So choose the α whose residual has just that size: explain the data down to the noise, and no further. *Needs the noise level.*

Generalised cross-validation (GCV). A good x_α should predict a data value that was left out. Averaged over all pixels this gives a function to minimise,

$$G(\alpha) = \frac{\|r_\alpha\|^2}{\left(\sum_i (1-\phi_i)\right)^2}.$$

Needs nothing.

L-curve. Plot $\log\|x_\alpha\|$ against $\log\|r_\alpha\|$ for all α. The curve looks like an L; the corner is where reducing the residual further starts to cost a large increase in the solution. *Needs nothing.*

```
grid = np.logspace(-5, 0, 200)

def residual_norm(B, S, a):
    return np.linalg.norm(a**2 / (S**2 + a**2) * B)            # (1 - phi) * B

def solution_norm(B, S, a):
    return np.linalg.norm(S * B / (S**2 + a**2))

def gcv(B, S, a):
    one_minus_phi = a**2 / (S**2 + a**2)
    return np.sum((one_minus_phi * B)**2) / np.sum(one_minus_phi)**2

def choose_alpha(B, S, sigma):
    """The three rules. Returns a dict {rule: alpha}."""
    dp = 10 ** optimize.brentq(lambda la: residual_norm(B, S, 10**la) - np.sqrt(B.size) * sigma, -6,
 ↪  1)
    g = grid[np.argmin([gcv(B, S, a) for a in grid])]
    rho = np.log([residual_norm(B, S, a) for a in grid]); eta = np.log([solution_norm(B, S, a) for a
 ↪  in grid])
    d1r, d1e = np.gradient(rho), np.gradient(eta)
    curvature = (d1r * np.gradient(d1e) - np.gradient(d1r) * d1e) / (d1r**2 + d1e**2) ** 1.5
    return {'discrepancy principle': dp, 'GCV': g, 'L-curve': grid[5 + np.argmax(curvature[5:-5])]}

def report(B, S, sigma, x_ref, b_ref):
    errs = [hno.rel_error(tikhonov_image(B, S, a), x_ref) for a in grid]
    rules = {'best possible (uses the truth)': grid[np.argmin(errs)], **choose_alpha(B, S, sigma)}
    print(f'{"rule":32s} {"alpha":>8s}   relative error      (blurred data: {hno.rel_error(b_ref,
     ↪  x_ref):.4f})')
    for name, a in rules.items():
        print(f'{name:32s} {a:8.4f}   {hno.rel_error(tikhonov_image(B, S, a), x_ref):.4f}')
    return rules

rules = report(B, S, noise_level, x_true, b_img)

fig, ax = plt.subplots(1, 2, figsize=(11, 3.3))
ax[0].loglog(grid, [gcv(B, S, a) for a in grid], color=hno.BLUE); ax[0].axvline(rules['GCV'],
 ↪  color=hno.ORANGE, lw=1)
ax[0].set_xlabel('$\\alpha$'); ax[0].set_title('GCV function, and its minimum', loc='left')
ax[1].plot(np.log([residual_norm(B, S, a) for a in grid]), np.log([solution_norm(B, S, a) for a in
 ↪  grid]), color=hno.BLUE)
for (name, a), col in zip(list(rules.items())[1:], (hno.AQUA, hno.ORANGE, hno.INK2)):
    ax[1].plot(np.log(residual_norm(B, S, a)), np.log(solution_norm(B, S, a)), 'o', color=col,
 ↪  label=name)
ax[1].set_xlabel('log residual norm'); ax[1].set_ylabel('log solution norm');
 ↪  ax[1].set_title('L-curve', loc='left'); ax[1].legend()
fig.tight_layout()
```

```
rule                             alpha   relative error      (blurred data: 0.0240)
best possible (uses the truth)    0.0622   0.0104
discrepancy principle             0.0353   0.0138
GCV                               0.0174   0.0264
L-curve                           0.0043   0.1035
```

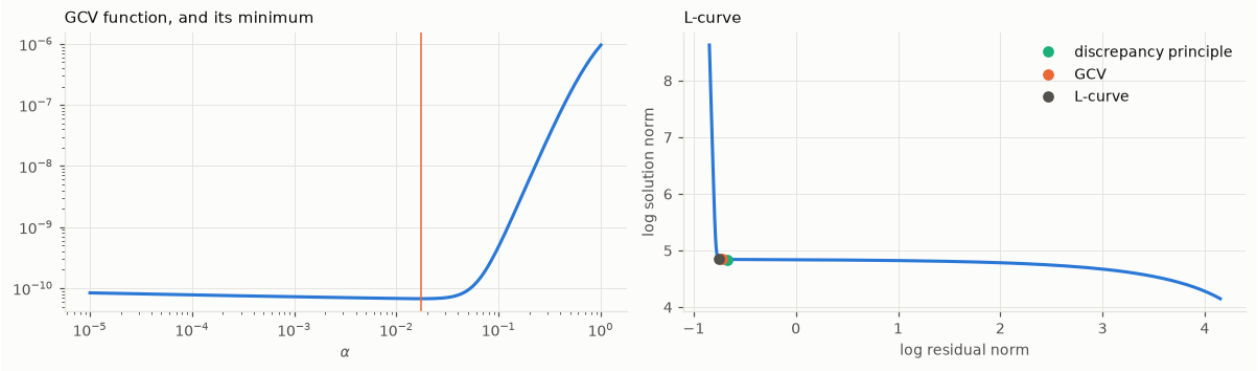


Figure 6.5: The GCV function with its minimum, and the L-curve with the choices of the three rules.

Compare each rule with the first line, the only one that used the true image.

- The **discrepancy principle** lands close to the best possible choice. It was told the noise level, and here the noise is exactly what it assumes.
- **GCV**, which was told nothing, regularises too little, and its result is no better than the blurred data.
- The **L-curve** corner is badly placed on this problem: it regularises far too little and the result is worse than the data. The L-curve is known to struggle when the true image is very smooth, which is our case.

None of the rules is infallible, and on another problem their ranking can change. This is the honest state of the art; the book discusses the strengths and failure modes of each.

Estimating the noise level. The discrepancy principle needs σ. It can be read off the data: at the highest frequencies the blurred signal has vanished (notebook 5), so those coefficients are pure noise.

```
m = B.shape[0]
high = B[m // 2:, m // 2:]                                    # the high-frequency quarter
sigma_hat = 1.4826 * np.median(np.abs(high))
print(f'noise estimated from the data: {sigma_hat:.5f}     true value: {noise_level}')
```

```
noise estimated from the data: 0.00201     true value: 0.002
```

6.5.1 When the assumptions fail

Now the honest version of the same experiment (notebook 4): the scene is larger than the picture, so light really came from outside, and the reflexive boundary is only an approximation. Same PSF, same noise, same rules.

```
x_h, b_h = hno.make_test_problem(x_true, psf, noise=noise_level)
S_h, B_h = hno.dct_eigenvalues(psf, b_h.shape), hno.dct2(b_h)
report(B_h, S_h, noise_level, x_h, b_h);
```

```
rule                                 alpha    relative error      (blurred data: 0.0238)
best possible (uses the truth)      0.0740   0.0164
discrepancy principle               0.0237   0.0291
GCV                                 0.0104   0.0550
L-curve                             0.0034   0.1392
```

All three rules now regularise too little, and the "restored" images are worse than the blurred one. Nothing is wrong with the code. The residual $b - Ax_\alpha$ now contains two things: the noise, and the **error of the model** at the edges, which is not white noise at all. The rules cannot tell them apart. They see structure in the residual, conclude that the solution does not yet explain the data, and push α down to chase an error that no x can explain.

This is the most practical lesson of the notebook: **parameter-choice rules are only as good as the model of the data**. With real images the model is never exact: the PSF is measured with some error, it varies across the field, bright stars saturate. The next section shows a rule that asks less of the model.

6.6 Real data, and a fourth rule

Let us deblur a real image: the average of our 45 exposures. First measure its PSF on a star (notebook 3), then apply Tikhonov regularisation with the parameter chosen by GCV. The PSF is not symmetric in general, so we use the FFT with tapered edges (notebook 4). With complex eigenvalues K, the Tikhonov solution reads $X = \bar{K}\,Y/(|K|^2 + \alpha^2)$.

```
y = data['mean'].astype(float)
star = hno.fit_star(y, 161, 169)                                  # an unsaturated star
print(f'PSF of the mean image: FWHM {star["fwhm"]:.2f} px = {star["fwhm"] *
 ↪  float(data["pixel_scale"]):.2f} arcsec, Moffat beta {star["beta"]:.2f}')

K = hno.psf2otf(hno.moffat_psf(y.shape, star['fwhm'], beta=star['beta']))
Y = fft2(hno.taper(y))

def gcv_fft(a):
    one_minus_phi = a**2 / (np.abs(K)**2 + a**2)
    return np.sum(np.abs(one_minus_phi * Y)**2) / np.sum(one_minus_phi)**2

alpha_real = grid[np.argmin([gcv_fft(a) for a in grid])]
x_gcv = ifft2(np.conj(K) * Y / (np.abs(K)**2 + alpha_real**2)).real
print(f'GCV chooses alpha = {alpha_real:.4f};   star FWHM after deblurring: {hno.fit_star(x_gcv,
 ↪  161, 169)["fwhm"]:.2f} px')
```

```
PSF of the mean image: FWHM 3.23 px = 2.46 arcsec, Moffat beta 2.72
GCV chooses alpha = 0.0018;   star FWHM after deblurring: 0.72 px
```

A star narrower than one pixel is not a restoration. GCV has chosen a tiny α, and the result is close to the naive inverse: this is the failure of the previous section, on real data, where the model is never exact (the PSF is itself a noisy measurement, and the bright stars are saturated).

There is a deeper problem. Even when a rule behaves, *how sharp is the result supposed to be?* Tikhonov regularisation does not say: the resolution of the output is whatever comes out, and it differs from image to image.

Our project turns the question around, following Magain et al. (1998). **Decide first what PSF the result should have**: a narrower one than the data, the *target* t, but not infinitely narrow. Then aim at the sky seen through that target, $t * x$, instead of the sky itself:

$$X_t = T\,\frac{\bar{K}\,Y}{|K|^2 + \lambda}.$$

It is the Tikhonov filter (with $\lambda = \alpha^2$) multiplied by the transfer function T of the target, which removes on purpose the highest frequencies, those we could never have restored. And the choice of λ becomes a **contract**: take the largest λ, hence the least noise, for which a star of the result really has the declared width.

```
target_fwhm = 0.8 * star['fwhm']                                  # declare: 20 % sharper than the data
T = hno.psf2otf(hno.gaussian_psf(y.shape, target_fwhm))

print(f'declared target: {target_fwhm:.2f} px')
chosen = None
for lam in (1e-1, 3e-2, 1e-2, 3e-3, 1e-3, 3e-4, 1e-4):
    x_t = ifft2(T * np.conj(K) * Y / (np.abs(K)**2 + lam)).real
    delivered = hno.fit_star(x_t, 161, 169)['fwhm']
    ok = delivered <= 1.05 * target_fwhm
    print(f'  lambda {lam:7.0e}: delivered {delivered:.2f} px  {"<- contract met" if ok and chosen
     ↪  is None else ""}')
    if ok and chosen is None:
        chosen = (lam, x_t)
```

```
crop = (slice(100, 228), slice(100, 228))
hno.show([y[crop], x_gcv[crop], chosen[1][crop]],
         ['mean of 45 exposures', f'Tikhonov, GCV', f'target {target_fwhm:.2f} px, $\\lambda$ =
 ↪  {chosen[0]:.0e}'], same_scale=True, width=3.4);
```

```
declared target: 2.58 px
  lambda   1e-01: delivered 4.24 px
  lambda   3e-02: delivered 3.65 px
  lambda   1e-02: delivered 3.27 px
  lambda   3e-03: delivered 2.98 px
  lambda   1e-03: delivered 2.82 px
  lambda   3e-04: delivered 2.70 px  <- contract met
  lambda   1e-04: delivered 2.63 px
```

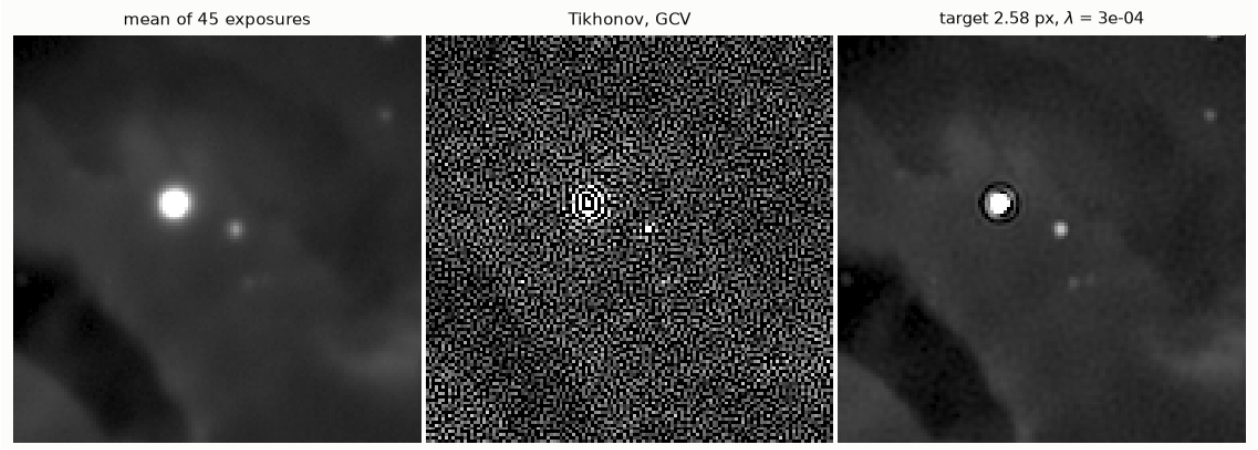


Figure 6.6: The mean of 45 real exposures, its Tikhonov restoration with the GCV parameter, and its restoration to a declared target.

Look at the three images.

- The GCV result is noise, as the sub-pixel star had warned.
- The target version is a credible, slightly sharper image, and it states what it delivers: we checked it on a star. That check is the whole difference between a *restoration* and a *measurement*.
- It also shows a **dark ring** around the saturated star: the model violation of notebook 1, still with us. The project removes the stars before deblurring the nebula for exactly this reason.

One limitation remains. The average of 45 exposures has **one** PSF, a blend of 45 different ones. The sharp exposures have been diluted by the blurred ones before we even started. Notebook 8 removes that limitation.

Your turn

1. **Derive the filter.** Insert $A = U\Sigma V^{\mathsf{T}}$ into $(A^{\mathsf{T}}A + \alpha^2 I)\,x = A^{\mathsf{T}}b$ and show that $x = \sum_i \frac{\sigma_i^2}{\sigma_i^2+\alpha^2}\frac{u_i^{\mathsf{T}}b}{\sigma_i}v_i$.
2. **Gradient.** Write $F(x) = \|Ax - b\|^2 + \alpha^2\|x\|^2$ and compute its gradient. Check numerically, on the 1-D problem, that the gradient vanishes at `x_normal` and not at the true signal. What does that tell you about what Tikhonov regularisation "believes"?
3. **More noise.** Redo sections 6.3 to 6.5 with `noise_level = 0.02`. How do the best α and the three rules move? Which rule tracked the change best?
4. **Wrong noise level.** Give the discrepancy principle a noise level that is 30 % too small, then 30 % too large. Which mistake is worse? Why do practitioners multiply σ by a safety factor slightly above 1?
5. **Coloured noise.** GCV assumes white noise. Smooth the noise with a small Gaussian before adding it to the blurred image, and look at the GCV function. What goes wrong?
6. **Ask for too much.** In section 6.6, declare a target of half the width of the data. Can the contract be met? At what cost in the background of the image? What is the honest conclusion?

Where to read more

- Hansen et al. (2006), chapter 6.
- Talbot (2024), chapter 3: Tikhonov regularisation as penalised least squares, with other penalties.
- Gonzalez and Woods (2018), sections 5.8 and 5.9 (Wiener and constrained least-squares filtering).
- Magain et al. (1998), for the target-PSF idea.

7 Colour images and smoothing norms

Companion notebook to Hansen, Nagy & O'Leary, ***Deblurring Images****, chapter 7, with the penalties of H. Talbot's* ***Lecture Notes on Optimisation****, chapter 3.*

Chapter 7 of the book opens three doors beyond the basic method. This notebook walks through each of them.

What you will take away

1. A colour image adds a second, small blurring matrix that mixes the channels; the two combine as a Kronecker product.
2. In our telescope images the channels do not mix, but **each has its own PSF**. Bringing them to a common, declared PSF removes colour fringes.
3. The Tikhonov penalty $\|x\|^2$ can be replaced by $\|Lx\|^2$, where L takes derivatives: the result prefers *smooth* images to *small* ones.
4. Squared penalties round off edges whatever L is. Penalising $\|Lx\|_1$ instead keeps them, at a price: no closed-form solution any more.

```
import numpy as np
from numpy.fft import fft2, ifft2
from scipy import ndimage
import matplotlib.pyplot as plt
import hnotools as hno

hno.style()
rng = np.random.default_rng(0)
data = hno.load_sample()
```

7.1 Colour: blurring across channels

A colour image has three channels. Two kinds of blur can affect it.

- **Within** each channel: the spatial blur we have studied, matrix A.
- **Across** channels: some of the red light is recorded as green, and so on. This happens in colour sensors and in film. It is described by a 3×3 matrix A_{color} whose row i says how the three true colours contribute to recorded colour i.

If every channel has the same spatial blur, the full model is a Kronecker product, $(A_{\text{color}} \otimes A)\, x = b$: mixing and blurring act on different "axes" of the data and do not interfere.

```
scene = np.stack([c / np.percentile(c, 99.5) for c in data['mean_sho'].astype(float)])   # 3 x 256 x
↪  256
A_color = np.array([[0.70, 0.20, 0.10],
                    [0.15, 0.70, 0.15],
                    [0.10, 0.20, 0.70]])                        # rows sum to 1: no light lost
psf = hno.gaussian_psf(scene.shape[1:], fwhm=3.0)

mix = lambda M, img: np.einsum('ij,jhw->ihw', M, img)           # apply a 3 x 3 matrix to the
↪  channels
blur3 = lambda img: np.stack([hno.blur(c, psf) for c in img])  # blur each channel

b1 = blur3(mix(A_color, scene))                                 # mix, then blur
```

```
b2 = mix(A_color, blur3(scene))                                  # blur, then mix
print('the two orders give the same image:', np.allclose(b1, b2))

unmixed = mix(np.linalg.inv(A_color), b1)                        # undo the mixing: a 3 x 3 inverse
print('after unmixing, each channel is just a blurred channel:', np.allclose(unmixed, blur3(scene)))
print(f'condition number of A_color: {np.linalg.cond(A_color):.1f}')
```

```
the two orders give the same image: True
after unmixing, each channel is just a blurred channel: True
condition number of A_color: 2.0
```

Because the two operations commute, the inverse problem splits: first undo the colour mixing, which is a well-conditioned 3×3 system, then deblur each channel separately with everything we already know. The hard part is still the spatial blur.

7.2 Our case: no mixing, but three different PSFs

Our colour images are made differently (notebook 2): three separate series of exposures through three narrow filters, on different nights. There is no cross-channel mixing, $A_{\text{color}} = I$. But the atmosphere was not the same on each night, so **each channel has its own PSF**. Measure it on two unsaturated stars:

```
sho = data['mean_sho'].astype(float)
names = ('S II  (red)', 'Hα  (green)', 'O III  (blue)')
stars = ((161, 169), (98, 217))
psf_params = []
for name, c in zip(names, sho):
    fits = [hno.fit_star(c, r, col) for r, col in stars]
    fwhm, beta = np.mean([f['fwhm'] for f in fits]), np.mean([f['beta'] for f in fits])
    psf_params.append((fwhm, beta))
    print(f'{name:14s} FWHM {fwhm:.2f} px = {fwhm * float(data["pixel_scale"]):.2f} arcsec')
```

```
S II  (red)    FWHM 2.68 px = 2.04 arcsec
Hα  (green)    FWHM 2.73 px = 2.09 arcsec
O III  (blue)  FWHM 3.21 px = 2.45 arcsec
```

The oxygen channel is about 20 % blurrier than the other two. In a colour composite, every star is then a sharp red-green core inside a wider blue disc: a **colour fringe**. The radial profiles of a star in the three channels, each scaled to its peak, show it directly:

```
def star_profile(c, row=161, col=169, half=10):
    """Radial profile of a star, background removed, scaled to the fitted peak."""
    f = hno.fit_star(c, row, col)
    stamp = c[row - half:row + half + 1, col - half:col + half + 1]
    yy, xx = np.indices(stamp.shape) - half
    dist = np.hypot(yy - (f['row'] - row), xx - (f['col'] - col)).ravel()
    values = ((stamp - f['bg']) / f['amp']).ravel()
    edges = np.arange(0, 7.5, 1.0)
    return 0.5 * (edges[:-1] + edges[1:]), np.array([np.median(values[(dist >= a) & (dist < b)])
     ↪  for a, b in zip(edges[:-1], edges[1:])])

def plot_profiles(ax, channels, title):
    for c, name, colour in zip(channels, names, (hno.ORANGE, hno.AQUA, hno.BLUE)):
        r, prof = star_profile(c)
        ax.plot(r, prof, 'o-', color=colour, ms=4, label=name)
    ax.set_xlabel('distance from the centre of the star (pixels)'); ax.set_title(title, loc='left')
```

The remedy uses the target idea of notebook 6. Choose **one** target PSF for the three channels, and filter each channel with *its own* measured PSF K_c:

$$X_c = T\,\frac{\bar{K}_c\,Y_c}{|K_c|^2 + \lambda}.$$

After this, all three channels show the sky through the same PSF T. We declare the width of the sharpest channel as the common target.

```
target = min(f for f, _ in psf_params)
T = hno.psf2otf(hno.gaussian_psf(sho.shape[1:], target))
matched = []
for (fwhm, beta), c in zip(psf_params, sho):
    K = hno.psf2otf(hno.moffat_psf(c.shape, fwhm, beta=beta))
    matched.append(ifft2(T * np.conj(K) * fft2(hno.taper(c)) / (np.abs(K)**2 + 1e-3)).real)
matched = np.stack(matched)

print(f'declared common target: {target:.2f} px')
for name, c in zip(names, matched):
    print(f'{name:14s} delivered {np.mean([hno.fit_star(c, r, col)["fwhm"] for r, col in
     ↪  stars]):.2f} px')

fig, ax = plt.subplots(1, 2, figsize=(10, 3.2), sharey=True)
plot_profiles(ax[0], sho, 'before: three different PSFs'); ax[0].legend()
plot_profiles(ax[1], matched, 'after: one common target')
fig.tight_layout()
```

```
declared common target: 2.68 px
S II  (red)    delivered 2.75 px
Hα  (green)    delivered 2.79 px
O III  (blue)  delivered 2.87 px
```

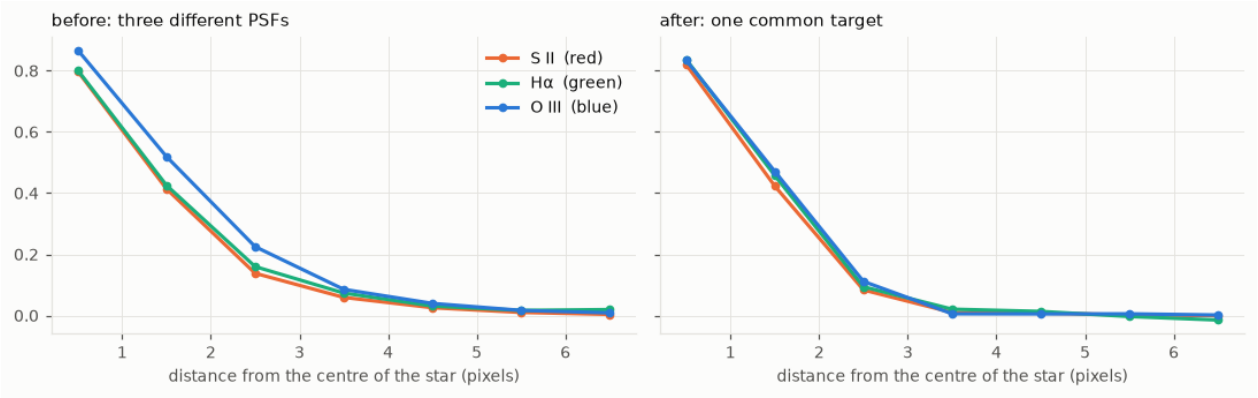


Figure 7.1: Radial profile of a star in the three channels, before and after matching to a common target.

The three widths now agree to within 4 %, against 20 % before, and the profiles fall on top of each other: the fringe is gone. (The widths are a few percent above the target because of the regularisation λ; notebook 6 showed how to trade that against noise.) For a scientist this matters far beyond looks: the *ratio* of two channels measures the physical state of the gas, and a ratio of two images taken through different PSFs is wrong near every sharp feature.

This "PSF matching to a declared target" is one of the things our project does to every image.

7.3 Tikhonov revisited: penalise the derivative

Notebook 6 penalised the size of the solution, $\alpha^2\|x\|^2$. But why should a good image be *small*? A more sensible belief is that a good image is **smooth**: neighbouring pixels have similar values. The lecture notes write the general form

$$x_\alpha = \arg\min_x \ \|Ax - b\|^2 + \alpha^2 \|Lx\|^2, \qquad (A^\mathsf{T}A + \alpha^2 L^\mathsf{T}L)\, x = A^\mathsf{T} b,$$

with L the identity (small solution), the **gradient** (small variations) or the **Laplacian** (small curvature).

With a fast transform nothing changes but the filter. For reflexive boundaries, $L^\mathsf{T}L$ is diagonalised by the same cosine transform as A, and the filter factors become

$$\phi = \frac{S^2}{S^2 + \alpha^2\, \ell^2}, \qquad \ell^2_{\text{gradient}}(k,l) = (2 - 2\cos\tfrac{\pi k}{n}) + (2 - 2\cos\tfrac{\pi l}{n}), \qquad \ell^2_{\text{Laplacian}} = (\ell^2_{\text{gradient}})^2.$$

ℓ^2 is zero at frequency zero and grows with frequency: **the penalty leaves the smooth part of the image alone and charges only for the fine detail**, which is where the noise lives.

```
x_true = data['hubble'].astype(float)
psf25 = hno.gaussian_psf((25, 25), fwhm=6.0)
b_img = ndimage.convolve(x_true, psf25, mode='reflect') + 0.002 * rng.standard_normal(x_true.shape)
S, B = hno.dct_eigenvalues(psf25, b_img.shape), hno.dct2(b_img)

n = b_img.shape[0]
one_d = 2 - 2 * np.cos(np.pi * np.arange(n) / n)
grad2 = one_d[:, None] + one_d[None, :]
penalties = {'identity': np.ones_like(grad2), 'gradient': grad2, 'Laplacian': grad2**2}

alphas = np.logspace(-4, 1, 61)
fig, ax = plt.subplots(figsize=(6.4, 3.2))
best = {}
for (name, L2), col in zip(penalties.items(), (hno.INK2, hno.BLUE, hno.ORANGE)):
    errs = [hno.rel_error(hno.idct2(S * B / (S**2 + a**2 * L2 + 1e-300)), x_true) for a in alphas]
    best[name] = (alphas[np.argmin(errs)], min(errs))
    ax.loglog(alphas, errs, color=col, label=name)
    print(f'{name:10s} best alpha {best[name][0]:.3f}: relative error {best[name][1]:.4f}')
ax.set_ylim(0.006, 1); ax.set_xlabel('$\\alpha$'); ax.set_ylabel('relative error'); ax.legend()
ax.set_title('Three penalties on the image problem of notebook 6', loc='left');
```

```
identity   best alpha 0.068: relative error 0.0105
gradient   best alpha 0.100: relative error 0.0089
Laplacian  best alpha 0.121: relative error 0.0088
```

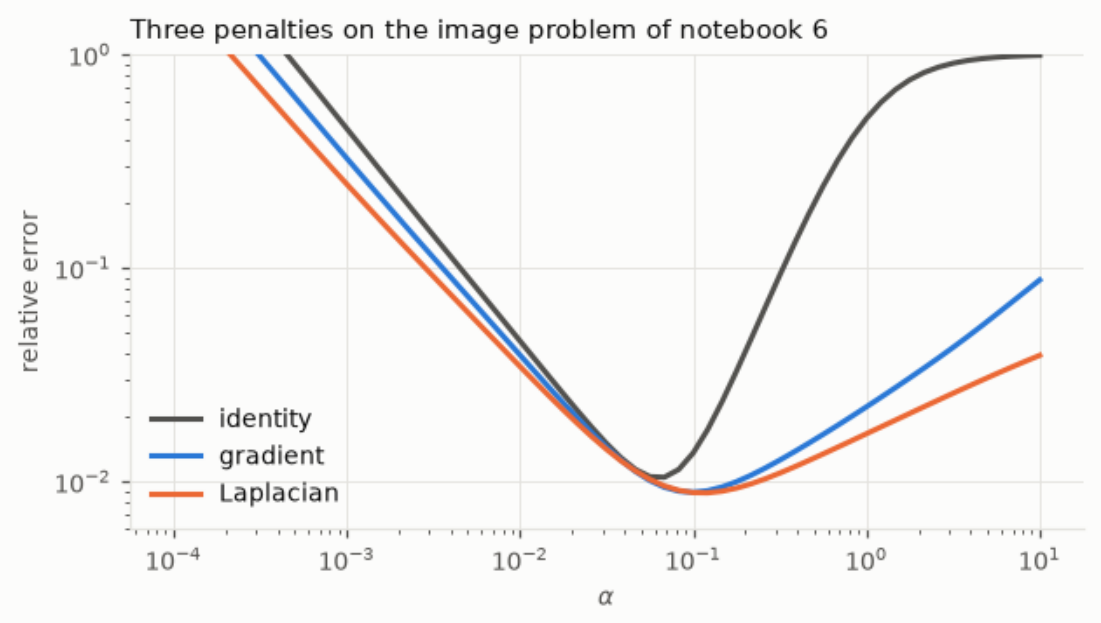


Figure 7.2: Relative error of three Tikhonov penalties against the regularisation parameter.

The derivative penalties do better than the identity, by about 15 % here, and the two of them are close to each other. A real gain, obtained for free, but not a revolution: all three are **squared** penalties, and they share a limitation that the next section makes visible.

7.4 Other norms: keeping the edges

Back to the 1-D signal, which has sharp steps. Solve the three Tikhonov problems with explicit matrices, each at its best α:

```
N = 250
s, signal = hno.lecture_signal(N)
A = hno.blur_matrix_1d(hno.gaussian_kernel_1d(4.0, 40), N)
b = A @ signal + 2e-4 * rng.standard_normal(N)

D = np.diff(np.eye(N), axis=0)                                  # first differences: (N-1) x N
operators = {'identity': np.eye(N), 'gradient': D, 'Laplacian': np.diff(np.eye(N), 2, axis=0)}

def tikhonov_general(A, b, L, alpha):
    return np.linalg.solve(A.T @ A + alpha**2 * L.T @ L, A.T @ b)

solutions = {}
for name, L in operators.items():
    err, a = min((hno.rel_error(tikhonov_general(A, b, L, a), signal), a) for a in np.logspace(-5,
 ↪  1, 61))
    solutions[name] = tikhonov_general(A, b, L, a)
    print(f'{name:10s} best alpha {a:.4f}: relative error {err:.4f}')
```

```
identity   best alpha 0.0010: relative error 0.0419
gradient   best alpha 0.0010: relative error 0.0421
Laplacian  best alpha 0.0013: relative error 0.0422
```

Three penalties, the same error to within one percent. On this signal the choice of L changes nothing, because the error is entirely in the **steps**, and a squared penalty hates steps: one jump of height h costs h^2, while the same rise spread over ten samples costs ten times $(h/10)^2$, that is ten times less. Squared penalties always prefer the ramp to the step.

The cure is to penalise the **absolute value** of the derivative instead of its square:

$$x = \arg\min_x \ \|Ax - b\|^2 + \alpha \|Dx\|_1, \qquad \|Dx\|_1 = \sum_i |x_{i+1} - x_i|.$$

Now a jump of height h costs h however it is spread: steps are no longer punished. This is **total variation** regularisation.

The price: the problem is not quadratic any more, so there is no linear system and no filter. We need an iterative algorithm. Here is one in a few lines, of the *majorise-minimise* family described in chapter 4 of the lecture notes: replace $|t|$ by a quadratic that touches it at the current solution, solve that Tikhonov problem, repeat.

```
def total_variation(A, b, alpha, iterations=60, eps=1e-3):
    """Total-variation deblurring by majorise-minimise: a sequence of
    weighted Tikhonov problems. Notebook 9 explains why this works."""
    x = tikhonov_general(A, b, D, 1e-2)                         # any reasonable start
    for _ in range(iterations):
        w = 1.0 / np.sqrt((D @ x)**2 + eps**2)                  # large weight where the signal is
 ↪  flat
        x = np.linalg.solve(A.T @ A + alpha * D.T @ (w[:, None] * D), A.T @ b)
    return x
```

```
x_tv = total_variation(A, b, alpha=1e-3)
print(f'total variation: relative error {hno.rel_error(x_tv, signal):.4f}')

fig, ax = plt.subplots(1, 2, figsize=(11, 3.3))
for a, (lo, hi) in zip(ax, ((0, N), (50, 120))):
    a.plot(s[lo:hi], signal[lo:hi], color=hno.INK2, lw=1.0, label='true signal')
    a.plot(s[lo:hi], solutions['gradient'][lo:hi], color=hno.ORANGE, lw=1.5, label='Tikhonov,
 ↪  gradient penalty')
    a.plot(s[lo:hi], x_tv[lo:hi], color=hno.BLUE, lw=1.5, label='total variation')
ax[0].set_title('the whole signal', loc='left'); ax[0].legend(fontsize=8)
ax[1].set_title('zoom on two steps', loc='left')
fig.tight_layout()
```

```
total variation: relative error 0.0215
```



Figure 7.3: Tikhonov regularisation with a gradient penalty, and total variation, on the one-dimensional signal.

Half the error, and the steps are steps again. The same data, the same blur, the same noise: only our **belief about the unknown** changed, from "smooth" to "piecewise smooth, with a few jumps".

Two warnings before you fall in love with it.

- The result is exactly as good as the belief. Total variation turns gentle slopes into small staircases, and would do damage to an image that really is smooth, such as a nebula.
- We have left the world of filters. There are no filter factors to plot and no Picard plot to tell us what came from the data and what came from the penalty. That loss of transparency is the price of every powerful prior, and it is the reason our project insists on *verifying* its results.

Your turn

1. **A badly conditioned colour matrix.** In section 7.1, make the channels mix almost completely (every entry of `A_color` close to $1/3$). What happens to the condition number, and to the unmixed image if you add a little noise to `b1` first?
2. **Ratios.** Compute the ratio image O III / S II before and after the PSF matching of section 7.2, and display both around the star. Where does the unmatched ratio go wrong?
3. **Which target?** Redo the matching with the *widest* channel's width as target, then with 70 % of the sharpest. Compare the delivered widths and the noise in a dark region of the image. Which channels pay, and how?
4. **Why zero at zero?** Explain why $\ell^2_{\text{gradient}}(0,0) = 0$ means that the gradient penalty does not care about the mean brightness of the image. Why is that an advantage for astronomical images, which sit on a bright sky background?
5. **Verify the eigenvalues.** For $n = 16$, build the 1-D first-difference matrix with reflexive boundaries and check numerically that $D^{\mathsf{T}}D$ is diagonalised by the cosine transform, with eigenvalues $2 - 2\cos(\pi k/n)$.
6. **Staircases.** Apply `total_variation` to a blurred, noisy *ramp* (a straight slope with no step). What does the solution look like? Now you know when not to use it.

Where to read more

- Hansen et al. (2006), chapter 7.
- Talbot (2024), chapter 3 (choice of the regularising operator) and section 4.5 (majorise-minimise).
- Rudin et al. (1992): the paper that introduced total variation to imaging.

Part II

Towards the research project

8 From one frame to many: proper coaddition

This notebook has no chapter in Hansen, Nagy & O'Leary. It applies everything the book taught to the situation of every astronomer: not one blurred image, but many, each with its own blur. It is the starting point of our research project. Reference: B. Zackay & E. Ofek, "How to COADD images. II", The Astrophysical Journal 836, 188 (2017).

What you will take away

1. With N exposures of the same sky, the least-squares problem still has a closed-form solution, and it costs one FFT per exposure.
2. That solution **stacks and deblurs in one step**. It is not the same as averaging first and deblurring afterwards, and it is never worse.
3. The gain comes from the exposures being *different*: each one is trusted at the spatial frequencies where it is good. No exposure is thrown away.
4. How much is gained can be computed in advance from the PSFs alone.
5. The estimator trusts every exposure blindly. One defective exposure contaminates the result: this is where the project begins.

```
import numpy as np
from numpy.fft import fft2, ifft2, fftfreq
import matplotlib.pyplot as plt
import hnotools as hno

hno.style()
data = hno.load_sample()
scale = float(data['pixel_scale'])                    # arcseconds per pixel
```

8.1 Forty-five exposures, forty-five PSFs

`data['cube']` holds 45 exposures of five minutes each, taken on several nights. `data['all_psf']` holds, for each one, the PSF measured on its stars (notebook 3), the small registration error left after aligning the exposures, and the noise level.

```
cube = data['cube'].astype(float)                     # 45 x 128 x 128
P = data['all_psf'].astype(float)                     # columns: fmaj, fmin, pa, beta, dx, dy,
 ↪  sigma, f
fwhm = np.sqrt(P[:, 0] * P[:, 1]) * scale             # arcseconds
noise = P[:, 6] / P[:, 7]                             # noise level of each exposure (estimated on
 ↪  whole frames,
                                                      # nebula included: only the ratios between
                                                       ↪  exposures matter)
N = len(cube)

order = np.argsort(fwhm)
fig, ax = plt.subplots(figsize=(6.4, 2.9))
ax.plot(np.arange(1, N + 1), fwhm[order], 'o', color=hno.BLUE, ms=4)
ax.set_xlabel('exposures, sorted from sharpest to blurriest'); ax.set_ylabel('PSF width (arcsec)')
ax.set_title('The seeing of the 45 exposures', loc='left');
print(f'PSF width from {fwhm.min():.2f} to {fwhm.max():.2f} arcsec, median {np.median(fwhm):.2f}')
print(f'noise from {noise.min():.1e} to {noise.max():.1e}')
```

```
PSF width from 1.46 to 3.56 arcsec, median 2.89
noise from 4.1e-04 to 4.4e-04
```

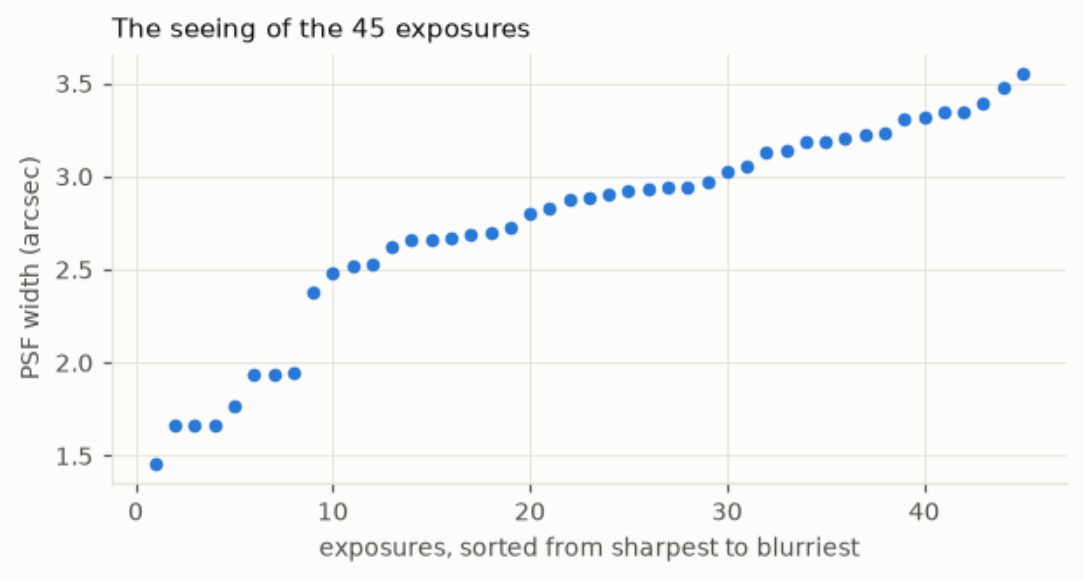


Figure 8.1: PSF width of the 45 exposures, sorted.

```
sharp, blurry = order[1], order[-2]
hno.show([cube[sharp], cube[blurry], cube.mean(axis=0)],
         [f'a sharp exposure ({fwhm[sharp]:.1f}")', f'a blurry one ({fwhm[blurry]:.1f}")', 'the mean
 ↪  of all 45'], same_scale=True);
```

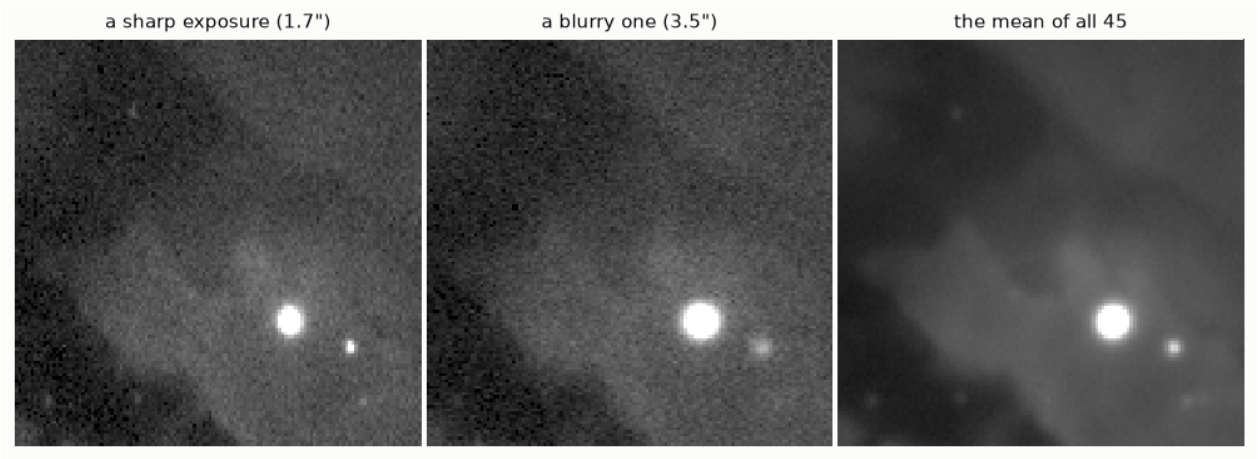


Figure 8.2: A sharp exposure, a blurry one, and the mean of all 45.

The width of the PSF varies by more than a factor of two. The mean of all exposures (right) has little noise, but it is blurrier than the best exposures: averaging has **diluted the sharp exposures in the blurry ones**.

Astronomers know two classical answers, and both waste something.

- *Average everything, then deblur.* Simple, but the fine detail that only the sharp exposures recorded has been diluted before the deblurring starts.
- *Keep only the sharpest exposures* ("lucky imaging"). The result is sharper, but most of the photons, collected at great cost, go in the bin.

8.2 The model and its solution

Write the model of notebook 1 once per exposure. All exposures see the same sky x; exposure i has its own PSF k_i and its own noise level s_i:

$$y_i = k_i * x + n_i, \qquad n_i \sim \mathcal{N}(0, s_i^2 I), \qquad i = 1, \dots, N.$$

Least squares, each exposure weighted by its noise (a noisy exposure counts less):

$$\hat{x} = \arg\min_x \sum_{i=1}^{N} \frac{1}{2s_i^2} \|y_i - k_i * x\|^2.$$

In Fourier space every convolution is a product, the sum of squares separates frequency by frequency, and each frequency is a one-unknown least-squares problem. Setting the derivative to zero gives

$$S(\nu)\, X(\nu) = R(\nu), \qquad R = \sum_i \frac{\bar{K}_i\, Y_i}{s_i^2}, \qquad S = \sum_i \frac{|K_i|^2}{s_i^2}.$$

S goes to zero at high frequencies, as $|K|^2$ did for one image, so we regularise exactly as in notebook 6: a declared target PSF t and a small λ,

$$\hat{X}_t = T \frac{R}{S + \lambda\, S(0)}.$$

This is **proper coaddition**. With $N = 1$ it is the target filter of notebook 6.

One detail before coding it. The exposures are aligned to a fraction of a pixel, not perfectly. A shift by (d_x, d_y) multiplies the Fourier transform by a phase, $e^{-2\pi i(\nu_x d_x + \nu_y d_y)}$ (the *shift theorem*), so the residual shift of each exposure simply goes into its K_i.

```
def frame_otf(shape, p, zoom=1.0):
    """Transfer function K_i of one exposure: its Moffat PSF, shifted by its
    registration error. p = (fmaj, fmin, pa, beta, dx, dy, ...)."""
    K = hno.psf2otf(hno.moffat_psf(shape, p[0] * zoom, p[1] * zoom, p[2], p[3]))
    nu_y, nu_x = fftfreq(shape[0])[:, None], fftfreq(shape[1])[None, :]
    return K * np.exp(-2j * np.pi * (nu_x * p[4] * zoom + nu_y * p[5] * zoom))

def proper_coadd(Y, K, s, T, lam):
    """Y, K: arrays (N, h, w) of Fourier transforms and transfer functions;
    s: noise levels (N,); T: transfer function of the target; lam: regularisation."""
    w = (1.0 / s**2)[:, None, None]
    R = np.sum(w * np.conj(K) * Y, axis=0)
    S = np.sum(w * np.abs(K)**2, axis=0)
    return ifft2(T * R / (S + lam * S.flat[0])).real

def stack_then_deblur(Y, K_mean, T, lam):
    """The classical pipeline: average the exposures, then one target filter."""
    return ifft2(T * np.conj(K_mean) * Y.mean(axis=0) / (np.abs(K_mean)**2 + lam)).real
```

Two functions of three lines each. Everything else in this notebook is measurement.

8.3 A controlled experiment

First with a known truth. We take the Hubble image as the sky and simulate 45 exposures of it with **the 45 real PSFs and the real relative noise levels** of our data. (The Hubble image has pixels half the size of ours, so the PSF widths are doubled to keep the same sky scale.)

The three contenders, all aiming at the same target of 1.8 arcsec:

- the plain mean, not deblurred;
- the mean, then deblurred, using the exact PSF of the mean (a generous assumption: in real life it has to be fitted);
- proper coaddition.

```
x = data['hubble'].astype(float)
rng = np.random.default_rng(3)
K_sim = np.array([frame_otf(x.shape, p, zoom=2.0) for p in P])
s_sim = 0.05 * noise / np.median(noise)                # 5 % noise in a typical exposure
Y_sim = np.array([K * fft2(x) + fft2(si * rng.standard_normal(x.shape)) for K, si in zip(K_sim,
 ↪  s_sim)])

T_sim = hno.psf2otf(hno.gaussian_psf(x.shape, 2 * 2.36)).real          # target: 1.8 arcsec
truth = ifft2(T_sim * fft2(x)).real                                     # the sky seen through the
 ↪  target
core = (slice(24, -24), slice(24, -24))
error = lambda img: hno.rel_error(img[core] - truth[core].mean(), truth[core] - truth[core].mean())

lams = np.logspace(-6, 0, 25)
K_mean = K_sim.mean(axis=0)
err_classic = [error(stack_then_deblur(Y_sim, K_mean, T_sim, l)) for l in lams]
err_proper = [error(proper_coadd(Y_sim, K_sim, s_sim, T_sim, l)) for l in lams]
mean_img = ifft2(Y_sim.mean(axis=0)).real

fig, ax = plt.subplots(figsize=(6.4, 3.2))
ax.loglog(lams, err_classic, color=hno.ORANGE, label='mean, then deblur')
ax.loglog(lams, err_proper, color=hno.BLUE, label='proper coaddition')
ax.axhline(error(mean_img), color=hno.INK2, lw=1.0, ls=(0, (4, 2)))
ax.text(lams[0], error(mean_img) * 1.07, 'plain mean', fontsize=8, color=hno.INK2)
ax.set_xlabel('$\\lambda$'); ax.set_ylabel('relative error against the sky at 1.8"');
 ↪  ax.legend(loc='upper left')
ax.set_title('Simulation with the 45 real PSFs', loc='left');
print(f'plain mean          {error(mean_img):.4f}')
print(f'mean, then deblur   {min(err_classic):.4f}   (best lambda
 ↪  {lams[np.argmin(err_classic)]:.0e})')
print(f'proper coaddition   {min(err_proper):.4f}   (best lambda
 ↪  {lams[np.argmin(err_proper)]:.0e})')
print(f'proper coaddition lowers the error by {1 - min(err_proper) / min(err_classic):.0%}')
```

```
plain mean          0.0600
mean, then deblur   0.0248   (best lambda 3e-03)
proper coaddition   0.0228   (best lambda 3e-03)
proper coaddition lowers the error by 8%
```

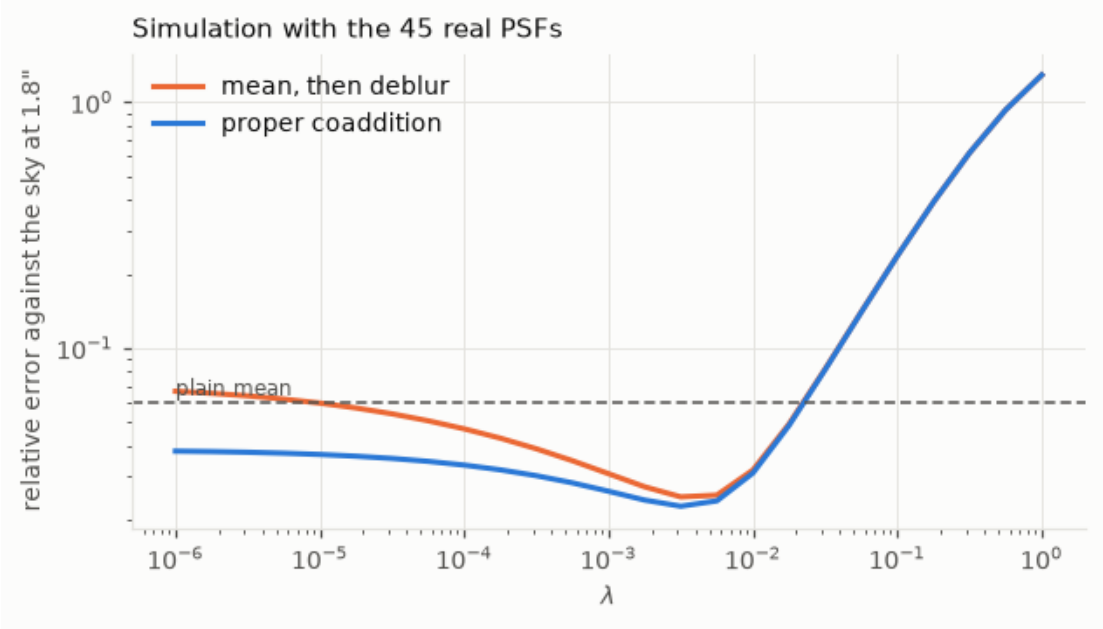


Figure 8.3: Simulation: relative error of the two pipelines against the regularisation parameter.

Both deblurring pipelines beat the plain mean by a wide margin. Proper coaddition is **never above** the classical pipeline, clearly below it for small λ, and at the best setting of each it has the lower error. The gain is modest here, and it was obtained against an opponent that was given the exact PSF of the mean. What matters is where it comes from.

8.4 Where the gain comes from

Look at one spatial frequency ν. In the sum R, exposure i enters with the weight $\bar{K}_i/s_i^2$: **it counts in proportion to how well it transmits that frequency**. A blurry exposure has $K_i(\nu) \approx 0$ at a high frequency and simply does not vote there; at low frequencies, where it is as good as any other, it votes fully.

Here is the share of each exposure in S, at a low and at a high frequency:

```
shape = cube.shape[1:]
K_all = np.array([frame_otf(shape, p) for p in P])
weights = np.abs(K_all)**2 / noise[:, None, None]**2

fig, ax = plt.subplots(figsize=(7.5, 3.0))
for (iy, ix), name, col in (((0, 4), '0.04 cycles/arcsec (large structures)', hno.AQUA), ((0, 32),
 ↪  '0.33 cycles/arcsec (fine detail)', hno.ORANGE)):
    share = weights[:, iy, ix] / weights[:, iy, ix].sum()
    ax.plot(np.arange(1, N + 1), 100 * share[order], 'o-', color=col, ms=3.5, label=name)
ax.axhline(100 / N, color=hno.INK2, lw=0.8); ax.text(N, 100 / N * 1.15, 'equal shares', ha='right',
 ↪  fontsize=8, color=hno.INK2)
ax.set_xlabel('exposures, sorted from sharpest to blurriest'); ax.set_ylabel('share of the
 ↪  information (%)')
ax.set_title('Who is trusted, at two spatial frequencies', loc='left'); ax.legend();
```

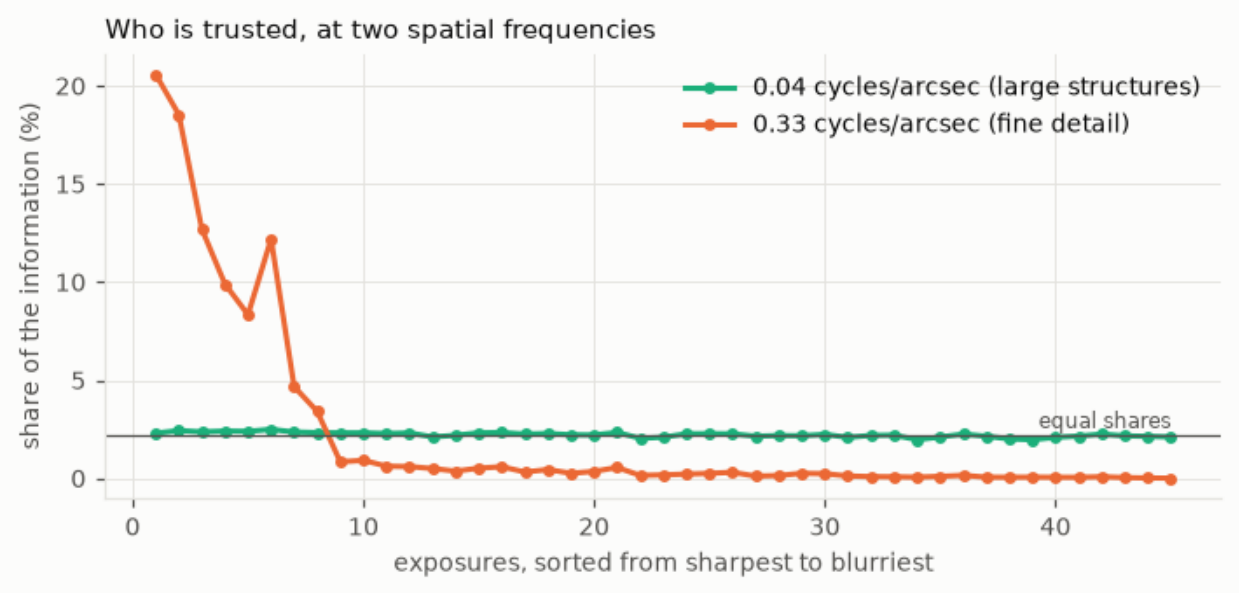


Figure 8.4: Share of each exposure in the information, at a low and at a high spatial frequency.

Figure 8.4 tells the story. For the large structures every exposure contributes about equally: nothing is wasted. For the fine detail, a handful of sharp exposures carry almost everything and the blurry ones are ignored. Proper coaddition is lucky imaging decided **frequency by frequency**, without throwing a single photon away.

The size of the gain can be written down. At each frequency, the ratio of the squared signal-to-noise ratios of proper coaddition and of the plain mean is

$$G(\nu) = \frac{S(\nu)\,\sum_i s_i^2}{\left|\sum_i K_i(\nu)\right|^2} \;\geq\; 1,$$

by the Cauchy-Schwarz inequality, with equality only if all exposures have the same PSF and the same noise. Deblurring the mean afterwards cannot change its signal-to-noise ratio at any frequency (it multiplies signal and noise by the same number), so G is also the advantage over the classical pipeline. And G needs only the PSFs and the noise levels: **it can be computed before any coaddition is done.**

```
S_real = weights.sum(axis=0)
G = S_real * np.sum(noise**2) / np.abs(K_all.sum(axis=0))**2
nu, G_r = hno.radial_profile(G)

fig, ax = plt.subplots(figsize=(6.4, 3.0))
ax.semilogy(nu / scale, G_r, color=hno.BLUE)
ax.axhline(1, color=hno.INK2, lw=0.8)
ax.set_yticks([1, 2, 3, 5, 7]); ax.set_yticklabels(['1', '2', '3', '5', '7']); ax.minorticks_off()
ax.set_xlabel('spatial frequency (cycles / arcsecond)'); ax.set_title('Predicted gain $G$ for these
↪  45 exposures', loc='left');
print(f'G at 0.3 cycles/arcsec: {G_r[np.argmin(np.abs(nu / scale - 0.3))]:.2f}')
```

```
G at 0.3 cycles/arcsec: 1.84
```

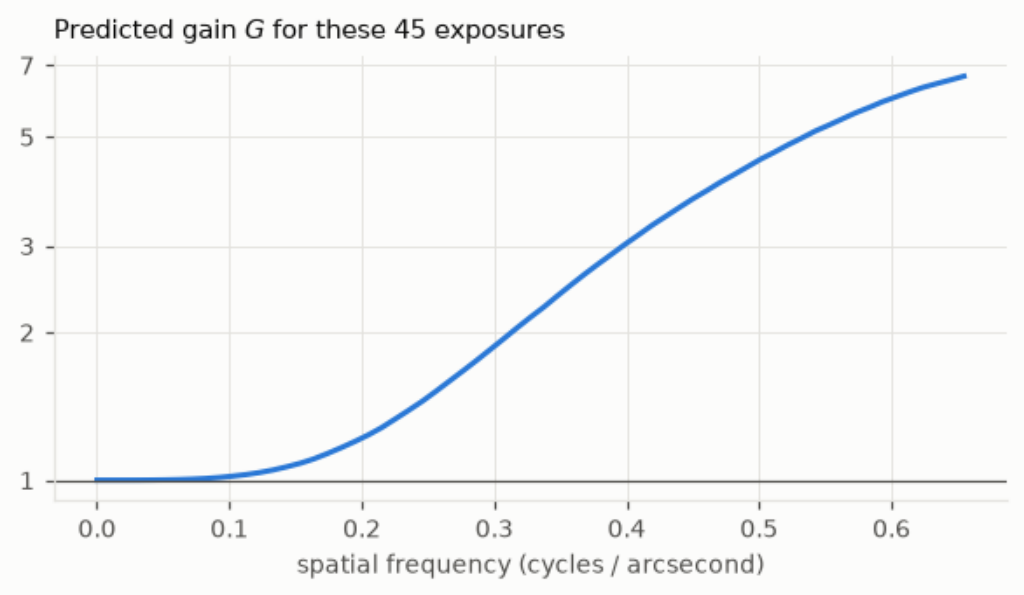


Figure 8.5: Predicted gain of proper coaddition over the plain mean, for the 45 real exposures.

Figure 8.5 predicts no gain for the large structures, a factor of almost two at 0.3 cycles per arcsecond, more beyond. For the two other colour channels of this nebula, whose exposures were taken in steadier conditions, the same computation gives 1.15: hardly any gain to expect. Whether this prediction matches the gain actually measured, across many fields and instruments, is the question of our research project.

8.5 The real exposures

Now the real cube. There is no true image, so we judge as in notebook 6: the **width of a star** in the result, and the **noise**, measured by processing the even and the odd exposures separately and taking half the difference (notebook 2).

For the classical pipeline we do what one does in practice: measure the PSF of the mean on a star, and deblur the mean with it.

```
Y_all = np.array([fft2(hno.taper(f, 12)) for f in cube])
T = hno.psf2otf(hno.gaussian_psf(shape, 2.36)).real                    # target: 2.36 px = 1.8
 ↪  arcsec
star = (97, 105)                                                       # an unsaturated star
fit_mean = hno.fit_star(cube.mean(axis=0), *star)
K_of_mean = hno.psf2otf(hno.moffat_psf(shape, fit_mean['fwhm'], beta=fit_mean['beta']))
even, odd = np.arange(0, N, 2), np.arange(1, N, 2)
quiet = (slice(4, 44), slice(4, 44))                                   # a corner without bright
 ↪  stars
robust_sigma = lambda a: 1.4826 * np.median(np.abs(a - np.median(a)))

def measure(pipeline, lam):
    """Star width (arcsec) of the full result, and its noise from the even/odd split."""
    full = pipeline(np.arange(N), lam)
    half_diff = (pipeline(even, lam) - pipeline(odd, lam)) / 2
    return hno.fit_star(full, *star)['fwhm'] * scale, robust_sigma(half_diff[quiet]), full

run_proper = lambda idx, lam: proper_coadd(Y_all[idx], K_all[idx], noise[idx], T, lam)
run_classic = lambda idx, lam: stack_then_deblur(Y_all[idx], K_of_mean, T, lam)

print(f'plain mean: star width {fit_mean["fwhm"] * scale:.2f} arcsec')
print(f'{"lambda":>8s} | proper: width  noise   | classic: width  noise')
```

```
results = {}
for lam in (1e-1, 3e-2, 1e-2, 3e-3, 1e-3, 3e-4):
    wp, n_p, img_p = measure(run_proper, lam); wc, n_c, img_c = measure(run_classic, lam)
    results[lam] = (wp, n_p, img_p, wc, n_c, img_c)
    print(f'{lam:8.0e} |        {wp:5.2f}  {n_p:.2e} |         {wc:5.2f}  {n_c:.2e}')
```

```
plain mean: star width 2.46 arcsec
  lambda | proper: width  noise    | classic: width  noise
   1e-01 |         2.95  1.01e-05 |          3.16  9.68e-06
   3e-02 |         2.50  1.65e-05 |          2.70  1.68e-05
   1e-02 |         2.22  2.39e-05 |          2.40  2.49e-05
   3e-03 |         2.02  3.18e-05 |          2.17  3.64e-05
   1e-03 |         1.90  3.88e-05 |          2.03  4.96e-05
   3e-04 |         1.84  4.45e-05 |          1.92  6.51e-05
```

Read the table along each row first: at every λ proper coaddition is **sharper**, and from $\lambda = 3 \times 10^{-2}$ downwards, which is where either method starts to sharpen the image at all, it is also **less noisy**.

Then compare at equal sharpness, which is the fair comparison. Proper coaddition at $\lambda = 3 \times 10^{-3}$ and the classical pipeline at $\lambda = 10^{-3}$ deliver the same star width:

```
wp, n_p, img_p = results[3e-3][:3]
wc, n_c, img_c = results[1e-3][3:]
print(f'proper coaddition : star width {wp:.2f} arcsec, noise {n_p:.2e}')
print(f'mean, then deblur : star width {wc:.2f} arcsec, noise {n_c:.2e}')
print(f'same sharpness, {1 - n_p / n_c:.0%} less noise')

inside = (slice(14, -14), slice(14, -14))                              # leave out the tapered frame
lo, hi = np.percentile(img_p[inside], [1, 99.0])
fig, ax = plt.subplots(1, 4, figsize=(11.6, 3.2))
panels = ((cube.mean(axis=0)[inside], 'mean of 45 exposures'), (img_c[inside], 'mean, then deblur'),
          (img_p[inside], 'proper coaddition'), (data['hubble'].astype(float)[28:-28, 28:-28],
 ↪  'Hubble, same field'))
for a, (img, name) in zip(ax, panels):
    l, h = (lo, hi) if 'Hubble' not in name else np.percentile(img, [1, 99.7])
    a.imshow(np.clip((img - l) / (h - l), 0, 1) ** 0.5, vmin=0, vmax=1); a.set_title(name)
    a.set_xticks([]); a.set_yticks([]); a.grid(False)
fig.tight_layout()
```

```
proper coaddition : star width 2.02 arcsec, noise 3.18e-05
mean, then deblur : star width 2.03 arcsec, noise 4.96e-05
same sharpness, 36% less noise
```

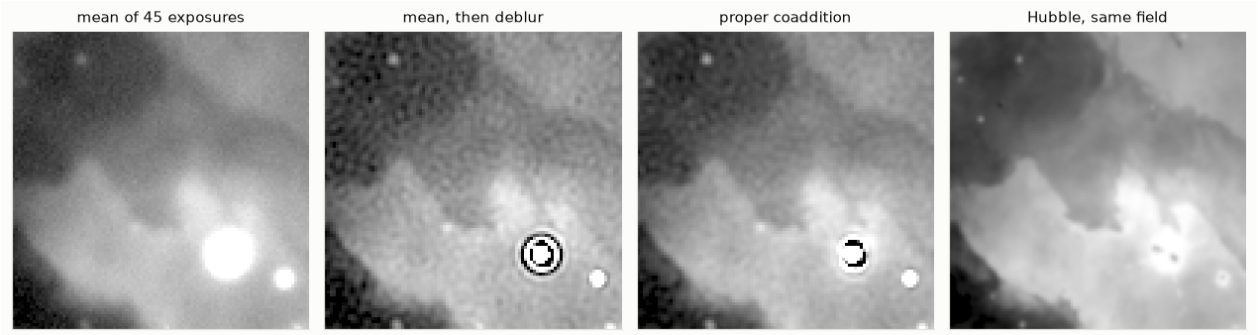


Figure 8.6: The real exposures: plain mean, mean then deblurred, proper coaddition at the same delivered star width, and Hubble.

A third less noise at the same delivered resolution, from the same 45 exposures, with a method that fits in

three lines and uses no prior knowledge about nebulae: only the measured PSFs. The Hubble image of the same field, on the right, shows what there is to be found.

Look also at the bright star: in both deblurred images it has a dark ring. It is saturated, it violates the model, and no amount of clever weighting changes that. The project measures the PSFs on the stars, then removes the stars from the exposures before coadding the nebula.

8.6 The weak point: every exposure is trusted

Least squares has no notion of a *wrong* measurement. Go back to the simulation and spoil **one** exposure out of 45 with a satellite trail, a bright line that the model knows nothing about.

```
bad = 10
yy, xx = np.indices(x.shape)
trail = 3.0 * np.exp(-0.5 * ((yy - 0.45 * xx - 60) / 1.5)**2)          # a bright oblique line
Y_bad = Y_sim.copy()
Y_bad[bad] = Y_sim[bad] + fft2(trail)

lam = lams[np.argmin(err_proper)]
clean = proper_coadd(Y_sim, K_sim, s_sim, T_sim, lam)
spoiled = proper_coadd(Y_bad, K_sim, s_sim, T_sim, lam)
print(f'error without the trail: {error(clean):.4f}     with one spoiled exposure:
 ↪  {error(spoiled):.4f}')
hno.show([ifft2(Y_bad[bad]).real, clean, spoiled], ['the spoiled exposure', 'coadd of 45 clean
 ↪  exposures', 'coadd with the spoiled one'], width=3.3);
```

```
error without the trail: 0.0228     with one spoiled exposure: 0.0425
```

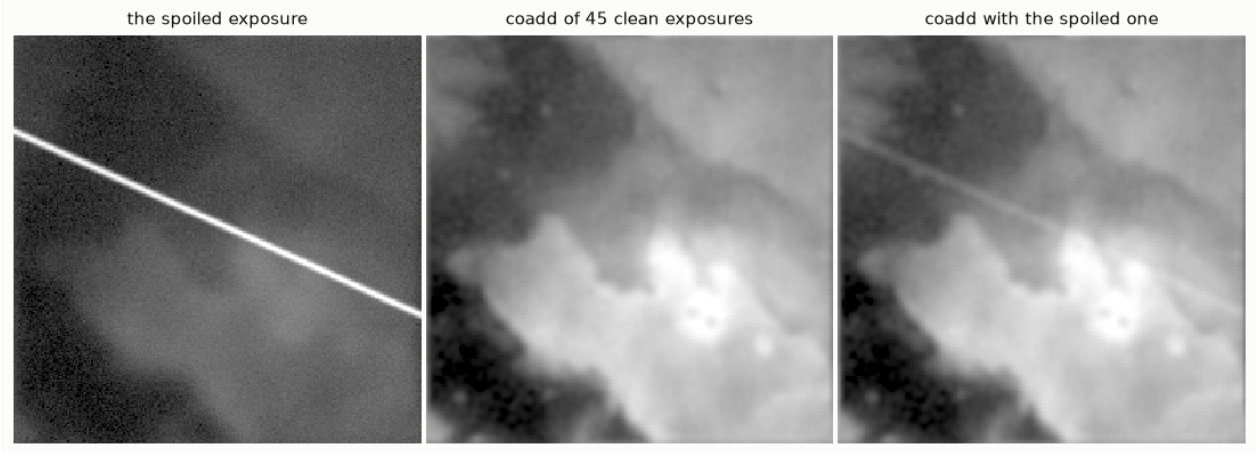


Figure 8.7: One spoiled exposure, the coadd of the clean exposures, and the coadd that includes the spoiled one.

One exposure in forty-five, and the trail is in the final image, and the error has almost doubled. The classical average at least has a standard defence, *outlier rejection*: discard, pixel by pixel, the values that disagree with the other exposures. Proper coaddition as written has none.

This is not a thought experiment. In our real data one exposure was taken during a gust of wind and shows every star twice. Its PSF fit looked perfectly normal, it was coadded with the others, and a ghost star appeared in the result.

Making the estimator **robust**, so that an exposure or a pixel that contradicts the others is detected and set aside, without giving up what proper coaddition gains, is the first task of the project. It requires replacing the square in the least-squares criterion by something more tolerant, and then there is no closed form any more. Notebook 9 provides the tools.

Your turn

1. **Derive it.** Starting from the weighted least-squares criterion of section 8.2, write the criterion in Fourier space (Parseval), differentiate with respect to $X(\nu)$ and obtain $SX = R$.
2. **One exposure.** Check numerically that `proper_coadd` with a single exposure equals the target filter of notebook 6.
3. **Identical exposures.** Replace all 45 PSFs by the same one in the simulation of section 8.3. What happens to the two error curves, and to G? Explain with the Cauchy-Schwarz inequality.
4. **Lucky imaging.** Coadd only the 10 sharpest exposures of the real cube, properly. Compare star width and noise with the full coadd at the same λ. Who wins on each axis, and why does the full coadd never lose on sharpness at equal noise?
5. **The shift theorem.** Set the registration errors (columns 4 and 5 of `P`) to zero in `frame_otf` and redo section 8.5. How much star width does ignoring them cost?
6. **A wrong PSF.** Multiply the PSF widths of all exposures by 1.2 before coadding the real cube (you "measured" them badly). What happens to the delivered star width? What does the dark ring around ordinary stars tell you? This is why the project *audits* its results.
7. **Before the project.** Read sections 1 to 3 of Zackay & Ofek (2017) and find our R and S in their equations. What do they call the image $R/\sqrt{S}$, and what is special about its noise?

Where to read more

- Zackay and Ofek (2017), the paper that introduced proper coaddition.
- Magain et al. (1998), for the declared target.
- The project's own notes: `docs/formulation/` and `docs/PSF-STUDY.md` in this repository.

9 Iterative methods: when there is no closed form

This notebook follows chapter 4 of H. Talbot's ***Lecture Notes on Optimisation*** *(gradient descent, conjugate gradient, majorise-minimise) and applies it to the problems of the previous notebooks. It is the bridge from the textbook material to the research project.*

Every solution so far was a formula: transform, divide with care, transform back. That works when a fast transform diagonalises the problem and the criterion is a sum of squares. Change either, and the formula is gone: a blur that varies across the image, boundaries that are neither periodic nor reflexive, a penalty that is not quadratic (notebook 7), a loss that must resist outliers (notebook 8).

What you will take away

1. An iterative method needs only two functions: one that applies A, one that applies A^T. Never a matrix.
2. **Gradient descent** is simple and slow, and stopping it early is itself a regularisation.
3. **Conjugate gradient** solves the Tikhonov problem in a few dozen iterations.
4. **Majorise-minimise** turns a hard criterion into a sequence of easy ones. Applied to a robust loss, it makes proper coaddition immune to a spoiled exposure, and each of its steps is an ordinary proper coaddition.

```
import numpy as np
from numpy.fft import fft2, ifft2, fftfreq
import matplotlib.pyplot as plt
import hnotools as hno

hno.style()
rng = np.random.default_rng(0)
data = hno.load_sample()
```

9.1 Operators, not matrices

An iterative method never asks "what is A?". It asks "what is A times this vector?", and sometimes "what is A^T times that one?". So we write the blur as two *functions*. For a convolution, A^T is the convolution with the PSF flipped upside-down and left-right; in Fourier space, multiplication by the conjugate $\bar{K}$.

We keep the FFT here because it is fast and familiar, but nothing below depends on it: replace the bodies of `A` and `AT` by any linear operation and its transpose, and the algorithms are unchanged.

```
x_true = data['hubble'].astype(float)
K = hno.psf2otf(hno.gaussian_psf(x_true.shape, fwhm=6.0))

A = lambda v: ifft2(K * fft2(v)).real                  # blur
AT = lambda v: ifft2(np.conj(K) * fft2(v)).real        # its transpose

b = A(x_true) + 0.005 * rng.standard_normal(x_true.shape)
```

A habit worth more than any algorithm: test your transpose. By definition, $\langle Au, v\rangle = \langle u, A^\mathsf{T}v\rangle$ for all u, v. Take two random images and compare the two numbers. If they differ, the operator pair is wrong, and every method built on it will quietly converge to nonsense.

```
u, v = rng.standard_normal(x_true.shape), rng.standard_normal(x_true.shape)
print('<A u, v>  =', np.sum(A(u) * v))
print('<u, AT v> =', np.sum(u * AT(v)))
```

```
<A u, v>  = 49.15959985748253
<u, AT v> = 49.15959985748253
```

9.2 Gradient descent

Take the plain least-squares criterion $F(x) = \frac{1}{2}\|Ax-b\|^2$. Its gradient is $\nabla F(x) = A^{\mathsf{T}}(Ax-b)$: blur the current guess, compare with the data, send the difference back through A^{T}. Gradient descent walks downhill,

$$x_{k+1} = x_k - \tau A^{\mathsf{T}}(Ax_k - b),$$

and converges if the step τ is below $2/\|A\|^2$. For a blur that conserves light the largest singular value is 1, so $\tau = 1$ is safe.

```
x = np.zeros_like(b)
err_gd = []
for k in range(1500):
    x = x - 1.0 * AT(A(x) - b)
    err_gd.append(hno.rel_error(x, x_true))
k_best = int(np.argmin(err_gd)) + 1

fig, ax = plt.subplots(figsize=(6.4, 3.1))
ax.loglog(np.arange(1, 1501), err_gd, color=hno.BLUE)
ax.plot(k_best, min(err_gd), 'o', color=hno.ORANGE)
ax.axhline(hno.rel_error(b, x_true), color=hno.INK2, lw=1.0, ls=(0, (4, 2)))
ax.text(1, hno.rel_error(b, x_true) * 1.04, 'blurred data', fontsize=8, color=hno.INK2)
ax.set_xlabel('iteration'); ax.set_ylabel('relative error'); ax.set_title('Gradient descent on the
 ↪  unregularised problem', loc='left');
print(f'smallest error {min(err_gd):.4f} at iteration {k_best};  after 1500 iterations
 ↪  {err_gd[-1]:.4f}')
```

```
smallest error 0.0271 at iteration 56;  after 1500 iterations 0.0556
```

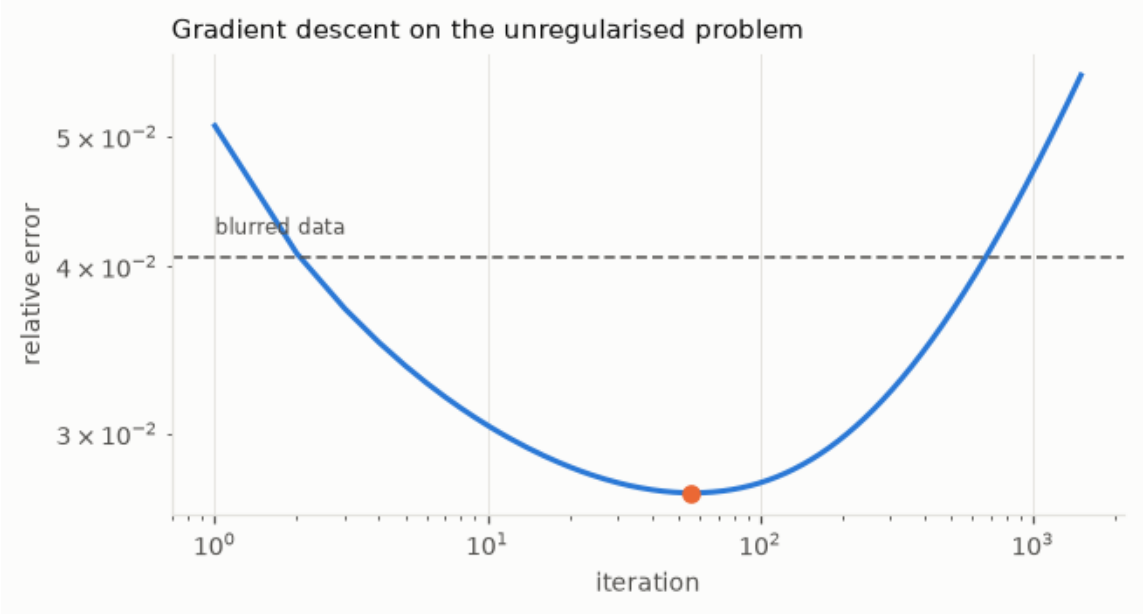


Figure 9.1: Gradient descent on the unregularised problem: the error decreases, then increases again.

In Figure 9.1 the error goes down, reaches a minimum, and **comes back up**. We are minimising the unregularised criterion, whose minimiser is the naive solution of notebook 1; given enough iterations, gradient descent would get there, garbage included.

But it gets there in a particular order. One can show that after k steps the iterate is a filtered solution in the sense of notebook 5, with filter factors $\phi_i = 1 - (1 - \tau\sigma_i^2)^k$: the components with large singular values converge first, the noisy ones last. **Stopping early is a regularisation, and the iteration count plays the role of $1/\alpha$.** This behaviour is called *semi-convergence*. It is the U-curve once more, with the iteration number on the horizontal axis.

9.3 Conjugate gradient

Now a criterion with a proper minimum: Tikhonov, $F(x) = \frac{1}{2}\|Ax - b\|^2 + \frac{1}{2}\alpha^2\|x\|^2$. Its minimiser solves the regularised normal equations of notebook 6,

$$\underbrace{(A^\mathsf{T}A + \alpha^2 I)}_{M}\, x = A^\mathsf{T}b,$$

a linear system whose matrix M is symmetric positive definite. For such systems there is a method far better than gradient descent. Gradient descent forgets where it has been and keeps partially undoing its previous steps. **Conjugate gradient** (CG) chooses each new direction so that it never spoils what the previous directions achieved. It needs one product by M per iteration, like gradient descent, and nothing else.

```
def conjugate_gradient(M, rhs, iterations, callback=None):
    """Solve M x = rhs for a symmetric positive definite operator M (a function)."""
    x = np.zeros_like(rhs)
    r = rhs - M(x)                          # residual
    p = r.copy()                            # search direction
    rs = np.sum(r * r)
    for _ in range(iterations):
        Mp = M(p)
        step = rs / np.sum(p * Mp)
        x = x + step * p
        r = r - step * Mp
        rs_new = np.sum(r * r)
        p = r + (rs_new / rs) * p           # new direction, conjugate to the previous ones
        rs = rs_new
        if callback: callback(x)
    return x

alpha = 0.05
M = lambda v: AT(A(v)) + alpha**2 * v
rhs = AT(b)
x_formula = ifft2(np.conj(K) * fft2(b) / (np.abs(K)**2 + alpha**2)).real    # the filter of notebook
 ↪ 6

dist_cg, dist_gd = [], []
conjugate_gradient(M, rhs, 60, callback=lambda x: dist_cg.append(hno.rel_error(x, x_formula)))
x = np.zeros_like(b)
for k in range(60):
    x = x - (M(x) - rhs) / (1 + alpha**2)
    dist_gd.append(hno.rel_error(x, x_formula))

fig, ax = plt.subplots(figsize=(6.4, 3.1))
ax.semilogy(np.arange(1, 61), dist_gd, color=hno.ORANGE, label='gradient descent')
ax.semilogy(np.arange(1, 61), dist_cg, color=hno.BLUE, label='conjugate gradient')
ax.set_xlabel('iteration'); ax.set_ylabel('distance to the exact Tikhonov solution'); ax.legend()
ax.set_title('Same cost per iteration, very different progress', loc='left');
print(f'after 60 iterations: gradient descent {dist_gd[-1]:.1e},  conjugate gradient
 ↪ {dist_cg[-1]:.1e}')
```

```
after 60 iterations: gradient descent 1.7e-02,  conjugate gradient 1.5e-04
```

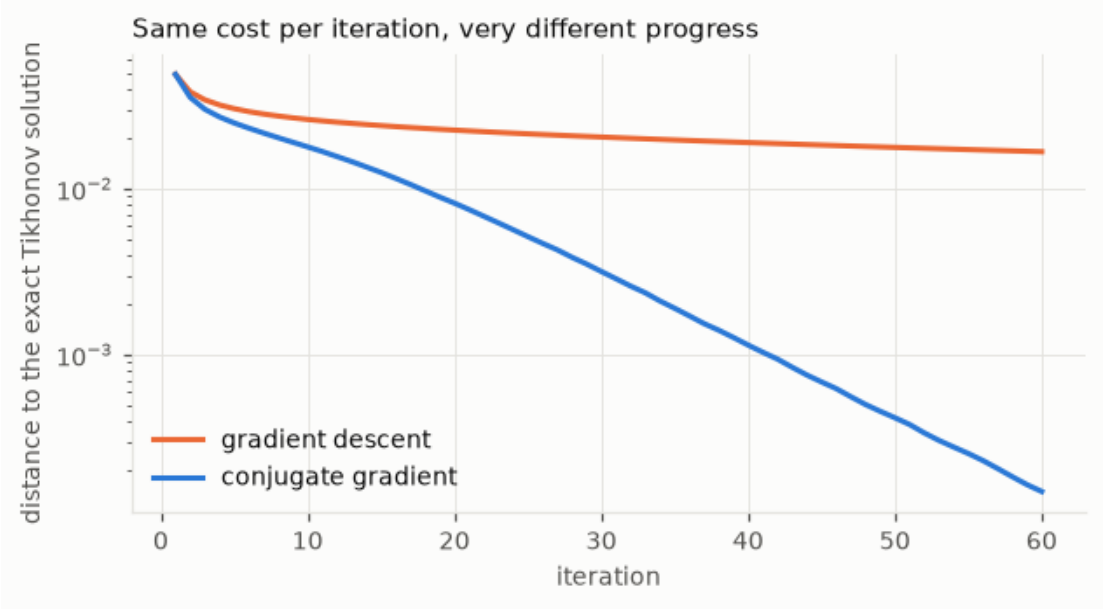


Figure 9.2: Distance to the exact Tikhonov solution, for gradient descent and for conjugate gradient.

Here we could check the answer, because for this problem the formula exists. The point of CG is that it does not need the formula: give it *any* pair `A`, `AT` and it solves the Tikhonov problem, for the penalties of notebook 7 as well. It is the workhorse of large inverse problems.

9.4 Majorise-minimise

Gradient descent and CG handle smooth criteria; CG wants a quadratic one. For anything else there is a strategy of great generality.

> To minimise a difficult function F, replace it by an easy function Q that lies **above** F everywhere and **touches** it at the current point. Minimise Q. Repeat.

Since $Q \geq F$ with equality at the current point, the minimiser of Q can only have a lower (or equal) value of F: **the criterion decreases at every step**, with no step size to tune. The art is in choosing Q quadratic, so that each step is a least-squares problem we already know how to solve.

A one-number example. Notebook 8 ended with a satellite trail. Look at a single pixel on the trail, through the 45 exposures: 44 values scattered around the truth and one far off. We want the "typical" value.

- Least squares, $\sum_i (y_i - m)^2$, gives the **mean**, which the outlier drags away.
- A **robust loss** grows quadratically for small residuals and only linearly for large ones, so an outlier cannot pull as hard. The classical choice is Huber's,

$$\rho(r) = \begin{cases} r^2/2 & |r| \leq \delta \\ \delta\,|r| - \delta^2/2 & |r| > \delta. \end{cases}$$

```
delta = 2.5
huber = lambda r: np.where(np.abs(r) <= delta, 0.5 * r**2, delta * np.abs(r) - 0.5 * delta**2)
clip = lambda r: np.clip(r, -delta, delta)                  # the derivative of the Huber loss

values = 1.0 + 0.1 * rng.standard_normal(45)                # one pixel in 45 exposures, noise 0.1
values[10] += 3.0                                           # ... one of them on the trail
sigma = 0.1

F = lambda m: np.sum(huber((values - m) / sigma))
m = values.mean()                                           # start from the least-squares answer
path = [m]
```

```
for _ in range(6):
    r = (values - m) / sigma
    m = np.mean(m + sigma * clip(r))                    # mean of the "modified" data: one MM
 ↪  step
    path.append(m)

grid_m = np.linspace(0.9, 1.15, 300)
fig, ax = plt.subplots(figsize=(6.6, 3.3))
ax.plot(grid_m, [F(g) for g in grid_m], color=hno.BLUE, label='robust criterion $F(m)$')
for k, col in ((0, hno.ORANGE), (1, hno.AQUA)):
    mk = path[k]; rk = (values - mk) / sigma
    Q = [F(mk) + np.sum(-clip(rk) * (g - mk) / sigma) + 0.5 * len(values) * ((g - mk) / sigma)**2
 ↪  for g in grid_m]
    ax.plot(grid_m, Q, color=col, lw=1.2, ls=(0, (4, 2)), label=f'quadratic majoriser at step {k}')
    ax.plot(mk, F(mk), 'o', color=col)
ax.set_ylim(F(path[-1]) - 3, F(path[0]) + 12); ax.set_xlabel('estimate $m$'); ax.legend(fontsize=8)
ax.set_title('Majorise-minimise on one pixel', loc='left');
print(f'mean (least squares): {values.mean():.3f}     robust estimate: {path[-1]:.3f}     median:
 ↪  {np.median(values):.3f}     truth: 1.000')
print('iterates:', np.round(path, 4))
```

```
mean (least squares): 1.065     robust estimate: 1.004     median: 1.005     truth: 1.000
iterates: [1.0651 1.005  1.0037 1.0036 1.0036 1.0036 1.0036]
```

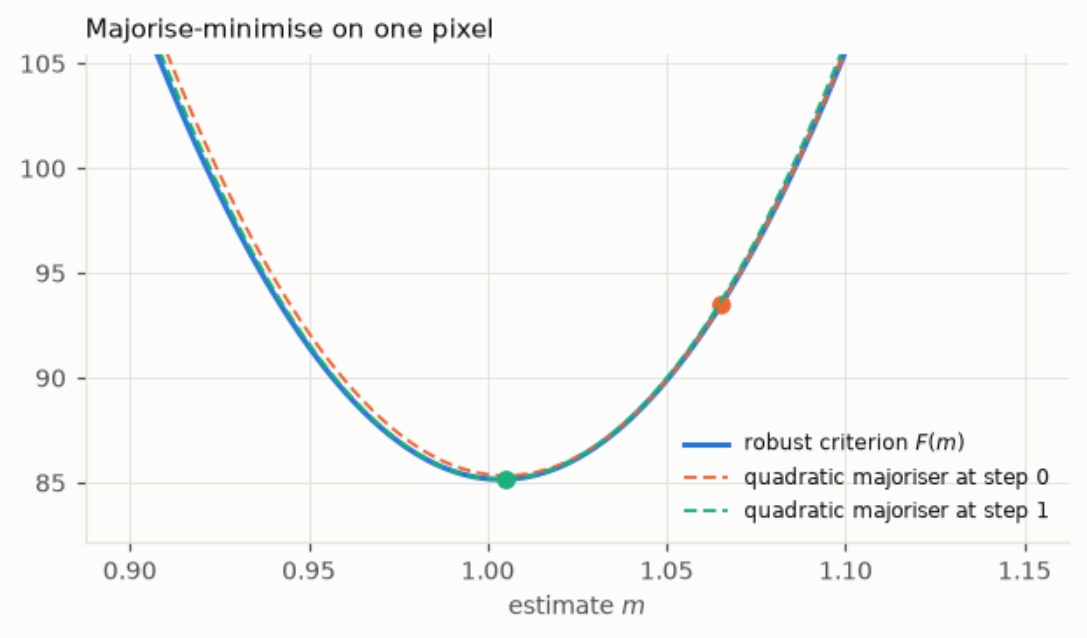


Figure 9.3: Majorise-minimise on one pixel: the robust criterion and two quadratic majorisers.

In Figure 9.3 each dashed parabola lies above the blue curve and touches it at the current estimate; its minimum is the next estimate. Two steps are enough here.

Look at what one step *is*. The Huber loss has curvature at most 1, so the parabola of curvature 1 tangent to it at the current residual r_k is a majoriser. Minimising a sum of such parabolas is a least-squares problem, on **modified data**: each value is kept if its residual is small, and pulled back to within δ noise levels of the current estimate if it is not,

$$y_i' = m_k + \sigma \operatorname{clip}\Big(\frac{y_i - m_k}{\sigma}\Big).$$

The outlier is not deleted by a rule; it is *limited in how much it can disagree*, and it finds its own place as the iterations proceed.

(Notebook 7 used the other common majoriser, for the absolute value: $|t| \le t^2/(2|t_k|) + |t_k|/2$, which gives a *re-weighted* least-squares problem at each step.)

9.5 Robust proper coaddition

Now the whole image. Replace the square by the Huber loss in the criterion of notebook 8:

$$\hat{x} = \arg\min_x \sum_{i=1}^{N} \sum_{\text{pixels}} \rho\left(\frac{y_i - k_i * x}{s_i}\right).$$

The majorise-minimise step is exactly the one above, exposure by exposure and pixel by pixel:

1. predict every exposure from the current estimate: $k_i * x_k$;
2. clip the residuals: $y_i' = k_i * x_k + s_i \,\mathrm{clip}((y_i - k_i * x_k)/s_i)$;
3. solve the least-squares problem for the modified exposures. **That is an ordinary proper coaddition**, the three-line formula of notebook 8.

So the robust estimator costs a few proper coadditions, and reuses all of the code. Here is the simulation of notebook 8 again, satellite trail included.

```
P = data['all_psf'].astype(float)
noise = P[:, 6] / P[:, 7]

def frame_otf(shape, p, zoom=1.0):
    Kf = hno.psf2otf(hno.moffat_psf(shape, p[0] * zoom, p[1] * zoom, p[2], p[3]))
    nu_y, nu_x = fftfreq(shape[0])[:, None], fftfreq(shape[1])[None, :]
    return Kf * np.exp(-2j * np.pi * (nu_x * p[4] * zoom + nu_y * p[5] * zoom))

def coadd_solve(Y, Ks, s, lam):
    """Proper coaddition without the target: the least-squares solution X."""
    w = (1.0 / s**2)[:, None, None]
    S = np.sum(w * np.abs(Ks)**2, axis=0)
    return np.sum(w * np.conj(Ks) * Y, axis=0) / (S + lam * S.flat[0])

def robust_coadd(y, Ks, s, lam, delta=2.5, iterations=5):
    """Huber-loss proper coaddition by majorise-minimise. y: exposures (N, h, w)."""
    X = coadd_solve(fft2(y), Ks, s, lam)                    # start: ordinary proper coaddition
    for _ in range(iterations):
        predicted = ifft2(Ks * X).real
        r = (y - predicted) / s[:, None, None]
        X = coadd_solve(fft2(predicted + s[:, None, None] * np.clip(r, -delta, delta)), Ks, s, lam)
    return X, np.abs(r) > delta                             # solution, and which residuals were
     ↪ clipped

sim_rng = np.random.default_rng(3)
Ks = np.array([frame_otf(x_true.shape, p, zoom=2.0) for p in P])
s = 0.05 * noise / np.median(noise)
y = np.array([ifft2(Kf * fft2(x_true)).real + si * sim_rng.standard_normal(x_true.shape) for Kf, si
 ↪ in zip(Ks, s)])
yy, xx = np.indices(x_true.shape)
y[10] += 3.0 * np.exp(-0.5 * ((yy - 0.45 * xx - 60) / 1.5)**2)          # the satellite trail, in
 ↪ exposure 10

T = hno.psf2otf(hno.gaussian_psf(x_true.shape, 2 * 2.36)).real
truth = ifft2(T * fft2(x_true)).real
core = (slice(24, -24), slice(24, -24))
error = lambda X: hno.rel_error(ifft2(T * X).real[core] - truth[core].mean(), truth[core] -
 ↪ truth[core].mean())
```

```
lam = 3e-3
X_plain = coadd_solve(fft2(y), Ks, s, lam)
X_robust, clipped = robust_coadd(y, Ks, s, lam)
print(f'ordinary proper coaddition : error {error(X_plain):.4f}')
print(f'robust proper coaddition   : error {error(X_robust):.4f}      (without any trail: 0.0228)')
print(f'residuals clipped: {clipped.mean():.2%} of all pixels;  in exposure 10:
 ↪  {clipped[10].mean():.2%}')

hno.show([ifft2(T * X_plain).real, ifft2(T * X_robust).real, clipped[10].astype(float)],
         ['ordinary proper coaddition', 'robust proper coaddition', 'clipped pixels of exposure
 ↪  10'], width=3.3);
```

```
ordinary proper coaddition : error 0.0427
robust proper coaddition   : error 0.0228      (without any trail: 0.0228)
residuals clipped: 1.29% of all pixels;  in exposure 10: 4.16%
```

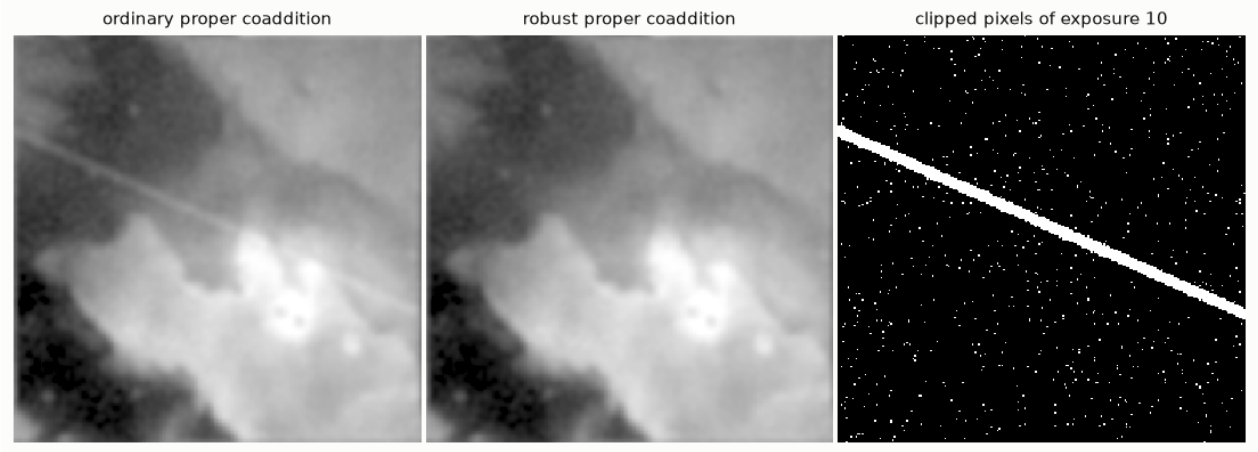


Figure 9.4: Ordinary and robust proper coaddition of the simulation with a satellite trail, and the pixels of the spoiled exposure that were clipped.

The trail is gone and the error is back to its value without any trail. The third panel shows which pixels of the spoiled exposure were limited: **the estimator found the trail by itself**, because those pixels disagreed with what the 44 other exposures said about the sky. Everything else in that exposure was kept and used.

About 1.2 % of the pixels of every clean exposure are clipped too: with $\delta = 2.5$, that is the share of ordinary Gaussian noise beyond 2.5 standard deviations. Clipping them costs a little efficiency; it is the insurance premium.

9.6 Real data: the robust loss as a diagnostic

On the real exposures nobody added a trail. But run the robust estimator anyway and ask *where* the residuals are clipped. Wherever many exposures disagree with the model at the same place, the model is wrong there.

```
cube = data['cube'].astype(float)
N, shape = len(cube), cube.shape[1:]
K_real = np.array([frame_otf(shape, p) for p in P])
pixel_noise = np.array([1.4826 * np.median(np.abs((cube[i] - cube[(i + 1) % N]) - np.median(cube[i]
 ↪  - cube[(i + 1) % N]))) / np.sqrt(2) for i in range(N)])
tapered = np.array([hno.taper(f, 12) for f in cube])

X_real, clipped_real = robust_coadd(tapered, K_real, pixel_noise, 3e-3, iterations=15)
how_often = clipped_real.mean(axis=0)                        # per pixel: fraction of exposures
 ↪  clipped
```

```
print(f'clipped overall: {clipped_real.mean():.2%}   (pure Gaussian noise would give 1.24 %)')
print(f'pixels clipped in more than half of the exposures: {(how_often > 0.5).mean():.2%} of the
 ↪  image')

fig, ax = plt.subplots(1, 2, figsize=(7.6, 3.6))
m = cube.mean(axis=0); lo, hi = np.percentile(m, [1, 99.7])
ax[0].imshow(np.clip((m - lo) / (hi - lo), 0, 1) ** 0.5); ax[0].set_title('mean of the 45
 ↪  exposures')
im = ax[1].imshow(how_often, cmap='gray_r', vmin=0, vmax=0.6); ax[1].set_title('how often each pixel
 ↪  was clipped')
for a in ax:
    a.set_xticks([]); a.set_yticks([]); a.grid(False)
fig.colorbar(im, ax=ax[1], fraction=0.046, label='fraction of exposures'); fig.tight_layout()
```

```
clipped overall: 2.95%   (pure Gaussian noise would give 1.24 %)
pixels clipped in more than half of the exposures: 0.88% of the image
```

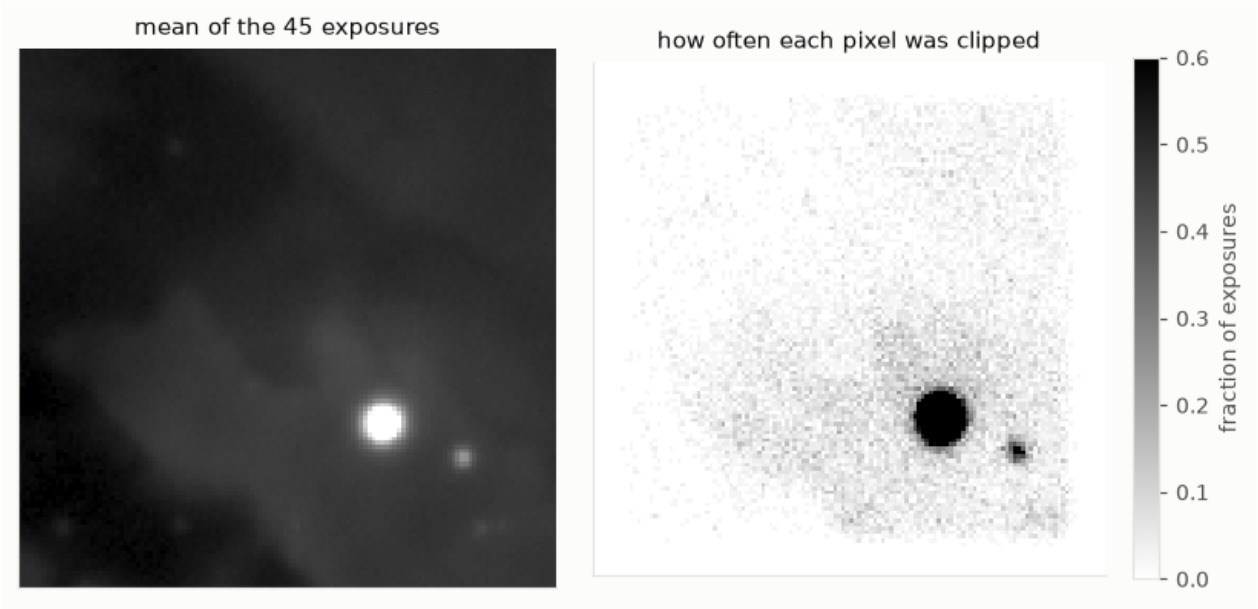


Figure 9.5: The mean of the real exposures, and how often each pixel was clipped by the robust estimator.

The map of Figure 9.5 is nearly white on the nebula, where the model holds and only the expected share of noise is clipped, and dark on the **stars**. This is the model violation we have met in every notebook since the first, now located automatically: a saturated core that is not the sky convolved with the PSF, and PSF wings that a Moffat profile does not describe exactly.

A robust loss is therefore two things at once: a protection against bad data, and an instrument that shows where the model and the data part company.

9.7 Where the project starts

You now have the full chain: the model of a blurred image, its fast solution, the reason regularisation is unavoidable and how to choose it, the extension to many exposures, and the tools to go beyond least squares. The research project begins exactly at the edge of what this notebook showed. Some of the questions it leaves open:

- **A whole exposure can be wrong**, not just some of its pixels: the exposure with doubled stars in our real data has a wrong *PSF*. Is the pixel-wise Huber loss enough, or do we need a test per exposure? What should that test measure?

- **How should δ be chosen**, and can the star cores be protected from being clipped merely because they are bright and imperfectly modelled?
- **Does robustness cost resolution or noise** on clean data? Notebook 8 gave the tools to measure both, on real exposures, with error bars.
- **Does the declared-target contract survive?** The delivered PSF must still be verified on the stars of the result.
- **Convergence and cost** on full-size images of sixty million pixels and two hundred exposures.

None of these has a textbook answer. That is what makes it research.

Your turn

1. **Break the transpose.** Replace `np.conj(K)` by `K` in `AT` and run the test of section 9.1, then conjugate gradient. What happens? Why did it make no difference for the Gaussian PSF, and why would it for our tilted telescope PSF?
2. **Filter factors of gradient descent.** Show that after k steps from $x_0 = 0$, gradient descent gives the filtered solution with $\phi_i = 1 - (1 - \tau\sigma_i^2)^k$. Plot ϕ against σ for $k = 10, 100, 1000$ next to the Tikhonov filter. Which α does each k resemble?
3. **Stopping rule.** Use the discrepancy principle of notebook 6 to stop gradient descent of section 9.2 without looking at the true image. How close to the best iteration do you land?
4. **CG for another penalty.** Solve the Tikhonov problem with the gradient penalty of notebook 7 by conjugate gradient, writing L and L^T as functions with `np.diff`. Test your transpose first.
5. **Choose δ.** Redo section 9.5 with $\delta = 1$, 2.5, 5 and 20. Tabulate the error with and without the trail. What do you lose at each end?
6. **Many trails.** Spoil 5, then 15, then 25 of the 45 exposures with trails at the same position. Where does the robust estimator break down, and why must it?
7. **A wrong PSF.** Spoil one exposure differently: generate it with a PSF twice as wide as the one you tell the estimator. Does the Huber loss notice? Look at the clipped map of that exposure. This is the first open question of section 9.7.

Where to read more

- Talbot (2024), chapter 4: gradient descent, conjugate gradient, majorise-minimise.
- Nocedal and Wright (2006), chapters 3 and 5, for the theory of line search and conjugate gradient.
- Huber and Ronchetti (2009), for robust losses and the "modified residuals" algorithm used here.
- Zackay and Ofek (2017), as in notebook 8, and the project's own notes in `docs/formulation/`.

References

Gonzalez, Rafael C., and Richard E. Woods. 2018. *Digital Image Processing*. 4th ed. Pearson.

Hansen, Per Christian, James G. Nagy, and Dianne P. O'Leary. 2006. *Deblurring Images: Matrices, Spectra, and Filtering*. SIAM. https://doi.org/10.1137/1.9780898718874.

Huber, Peter J., and Elvezio M. Ronchetti. 2009. *Robust Statistics*. 2nd ed. Wiley.

Magain, P., F. Courbin, and S. Sohy. 1998. "Deconvolution with Correct Sampling." *The Astrophysical Journal* 494: 472–77. https://doi.org/10.1086/305187.

Nocedal, Jorge, and Stephen J. Wright. 2006. *Numerical Optimization*. 2nd ed. Springer.

Rudin, Leonid I., Stanley Osher, and Emad Fatemi. 1992. "Nonlinear Total Variation Based Noise Removal Algorithms." *Physica D* 60: 259–68. https://doi.org/10.1016/0167-2789(92)90242-F.

Strang, Gilbert. 2016. *Introduction to Linear Algebra*. 5th ed. Wellesley-Cambridge Press.

Talbot, Hugues. 2024. "Lecture Notes on Optimisation: Applications to Computer Vision, Image and Signal Processing, and Machine Learning." Unpublished manuscript.

Zackay, Barak, and Eran O. Ofek. 2017. "How to COADD Images. II. A Coaddition Image That Is Optimal for Any Purpose in the Background-Dominated Noise Limit." *The Astrophysical Journal* 836 (2): 188. https://doi.org/10.3847/1538-4357/836/2/188.

A The helper module `hnotools.py`

The functions shared by all chapters. Everything here is something you should be able to write yourself.

```
"""Small helpers shared by the notebooks. Read this file: it is short, and
everything in it is something you should be able to write yourself."""
import os
import numpy as np
from numpy.fft import fft2, ifft2, ifftshift, fftfreq
import matplotlib.pyplot as plt

BLUE, ORANGE, AQUA = '#2a78d6', '#eb6834', '#1baf7a'
INK, INK2, SURFACE, GRIDC = '#0b0b0b', '#52514e', '#fcfcfb', '#e4e3df'


def style():
    """One plotting style for the whole series."""
    plt.rcParams.update({
        'figure.facecolor': SURFACE, 'axes.facecolor': SURFACE, 'savefig.facecolor': SURFACE,
        'axes.edgecolor': GRIDC, 'axes.labelcolor': INK2, 'text.color': INK,
        'xtick.color': INK2, 'ytick.color': INK2, 'axes.grid': True, 'grid.color': GRIDC,
        'grid.linewidth': 0.6, 'axes.spines.top': False, 'axes.spines.right': False,
        'font.size': 9, 'axes.titlesize': 9.5, 'lines.linewidth': 2.0,
        'legend.frameon': False, 'figure.dpi': 110, 'image.cmap': 'gray',
        'image.interpolation': 'nearest'})


def load_sample():
    """The real-data sample (see make_sample.py for what each array is)."""
    here = os.path.dirname(os.path.abspath(__file__))
    return dict(np.load(os.path.join(here, 'data', 'm16_sample.npz')))


def gaussian_psf(shape, fwhm):
    """Circular Gaussian PSF, centred on the array, summing to 1."""
    sigma = fwhm / (2 * np.sqrt(2 * np.log(2)))
    y, x = np.indices(shape)
    y, x = y - shape[0] // 2, x - shape[1] // 2
    p = np.exp(-(x**2 + y**2) / (2 * sigma**2))
    return p / p.sum()


def moffat_psf(shape, fwhm_major, fwhm_minor=None, pa_deg=0.0, beta=2.5):
    """Elliptical Moffat PSF, the usual model of a star seen through the
    atmosphere: (1 + r^2)^(-beta), heavier wings than a Gaussian."""
    fwhm_minor = fwhm_minor or fwhm_major
    c = 2 * np.sqrt(2 ** (1 / beta) - 1)          # FWHM = c * alpha
    a, b = fwhm_major / c, fwhm_minor / c
    y, x = np.indices(shape)
    y, x = y - shape[0] // 2, x - shape[1] // 2
    t = np.radians(pa_deg)
    u = (x * np.cos(t) + y * np.sin(t)) / a
    v = (-x * np.sin(t) + y * np.cos(t)) / b
```

```
    p = (1 + u**2 + v**2) ** (-beta)
    return p / p.sum()


def psf2otf(psf):
    """Fourier transform of a centred PSF (its optical transfer function)."""
    return fft2(ifftshift(psf))


def blur(x, psf):
    """Convolve x with psf, with periodic boundaries (one FFT product)."""
    return np.real(ifft2(fft2(x) * psf2otf(psf)))


def radial_profile(a):
    """Azimuthal average of a 2-D Fourier-domain array (fft2 layout).
    Returns frequency in cycles/pixel and the averaged values."""
    n0, n1 = a.shape
    r = np.hypot(fftfreq(n0)[:, None], fftfreq(n1)[None, :])
    idx = np.rint(r * max(n0, n1)).astype(int)
    nb = max(n0, n1) // 2 + 1
    tot = np.bincount(idx.ravel(), weights=np.abs(a).ravel(), minlength=nb)[:nb]
    cnt = np.bincount(idx.ravel(), minlength=nb)[:nb]
    return np.arange(nb) / max(n0, n1), tot / np.maximum(cnt, 1)


def rel_error(estimate, truth):
    """Relative error ||estimate - truth|| / ||truth||."""
    return np.linalg.norm(estimate - truth) / np.linalg.norm(truth)


def show(images, titles=None, lo=1, hi=99.7, same_scale=False, width=3.0, gamma=0.5):
    """Show images side by side with a percentile stretch.
    same_scale=True uses the first image's limits for all of them."""
    images = list(images)
    fig, axes = plt.subplots(1, len(images), figsize=(width * len(images), width * 1.08),
 ↪  squeeze=False)
    lim = None
    for ax, im, k in zip(axes[0], images, range(len(images))):
        if lim is None or not same_scale:
            lim = np.percentile(im, [lo, hi])
        z = np.clip((im - lim[0]) / (lim[1] - lim[0] + 1e-30), 0, 1) ** gamma
        ax.imshow(z, vmin=0, vmax=1)
        ax.set_xticks([]); ax.set_yticks([]); ax.grid(False)
        for sp in ax.spines.values():
            sp.set_visible(False)
        if titles:
            ax.set_title(titles[k])
    fig.tight_layout(pad=0.3)
    return fig


def find_stars(img, n=10, box=9):
    """Positions (row, col) and peak values of the n brightest local maxima,
    brightest first. Good enough to locate stars in a small field."""
    from scipy import ndimage
    smooth = ndimage.gaussian_filter(img, 1.0)
    peak = (smooth == ndimage.maximum_filter(smooth, box))
    peak[:box, :] = peak[-box:, :] = False
```

```
    peak[:, :box] = peak[:, -box:] = False
    rows, cols = np.nonzero(peak)
    order = np.argsort(img[rows, cols])[::-1][:n]
    return [(int(rows[i]), int(cols[i]), float(img[rows[i], cols[i]])) for i in order]


def lecture_signal(N=250):
    """The 1-D test signal of H. Talbot's lecture notes (notebook 1):
    steps plus slow waves, sampled on [0, 3 pi]. Returns (abscissa, signal)."""
    s = np.linspace(0, 3 * np.pi, N)
    return s, 1.0 * (np.cos(2 * s) > 0) + 2 * np.sin(s / 4) + 0.1 * np.cos(3 * s + 1)


def gaussian_kernel_1d(sigma, half):
    """Normalised 1-D Gaussian kernel of 2*half + 1 samples."""
    o = np.arange(-half, half + 1)
    k = np.exp(-o**2 / (2 * sigma**2))
    return k / k.sum()


def blur_matrix_1d(kernel, N, mode='constant'):
    """Blurring matrix of a signal of length N (notebook 4): one column per
    sample. mode = 'constant' (zero), 'wrap' (periodic) or 'reflect'."""
    from scipy import ndimage
    return np.column_stack([ndimage.convolve1d(np.eye(N)[j], kernel, mode=mode) for j in range(N)])


# ---- reflexive boundaries: the cosine transform (notebook 4) ----

def dct2(a):
    from scipy.fft import dctn
    return dctn(a, type=2, norm='ortho')


def idct2(a):
    from scipy.fft import idctn
    return idctn(a, type=2, norm='ortho')


def dct_eigenvalues(psf, shape):
    """Eigenvalues of the blur with reflexive boundaries, for a PSF that is
    symmetric in both directions: dct2(A e1) / dct2(e1)."""
    from scipy import ndimage
    e1 = np.zeros(shape); e1[0, 0] = 1.0
    return dct2(ndimage.convolve(e1, psf, mode='reflect')) / dct2(e1)


def make_test_problem(big, psf, noise, margin=64, seed=1):
    """Honest test data (notebook 4): blur a large scene, keep its centre,
    add noise. Returns (x_true, b)."""
    from scipy import ndimage
    inner = slice(margin, big.shape[0] - margin)
    blurred = ndimage.convolve(big, psf, mode='reflect')[inner, inner]
    return big[inner, inner], blurred + noise *
     ↪  np.random.default_rng(seed).standard_normal(blurred.shape)


def taper(b, width=16):
    """Fade an image to its mean over `width` pixels along every edge, so
```

```
    that periodic boundaries do no harm (notebook 4)."""
    ry, rx = np.ones(b.shape[0]), np.ones(b.shape[1])
    edge = 0.5 - 0.5 * np.cos(np.pi * np.arange(width) / width)
    for r in (ry, rx):
        r[:width] = edge; r[-width:] = edge[::-1]
    return b.mean() + (b - b.mean()) * np.outer(ry, rx)


def fit_star(img, row, col, half=10):
    """Fit a circular Moffat profile on a constant background to the star
    near (row, col) (notebook 3). Returns a dict with the FWHM in pixels."""
    from scipy import optimize
    stamp = img[row - half:row + half + 1, col - half:col + half + 1]
    yy, xx = np.indices(stamp.shape) - half
    def model(p):
        amp, y0, x0, alpha, beta, bg = p
        return bg + amp * (1 + ((yy - y0)**2 + (xx - x0)**2) / alpha**2) ** (-beta)
    span = stamp.max() - np.median(stamp)
    fit = optimize.least_squares(lambda p: (model(p) - stamp).ravel(),
                                 x0=[span, 0, 0, 3.0, 2.5, np.median(stamp)],
                                 bounds=([0, -3, -3, 0.3, 1.1, -np.inf], [10 * span + 1e-30, 3, 3,
↪   30, 50, np.inf]))
    amp, y0, x0, alpha, beta, bg = fit.x
    return {'fwhm': 2 * alpha * np.sqrt(2 ** (1 / beta) - 1), 'beta': beta, 'amp': amp,
            'bg': bg, 'row': row + y0, 'col': col + x0}
```