Modular & Scalable Computational Imaging with Python 🐍

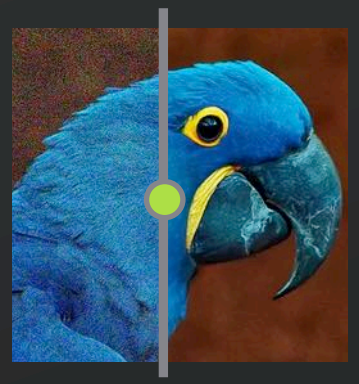
- Matthieu Simeoni

The many facets of Image Reconstruction 💎

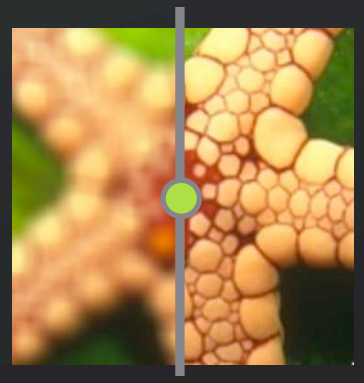
From denoising to tomography, and back

Image reconstruction methods aim at restoring degraded, partially or indirectly observed raw images to produce high-quality ones that are better suited for visualisation or image analysis.

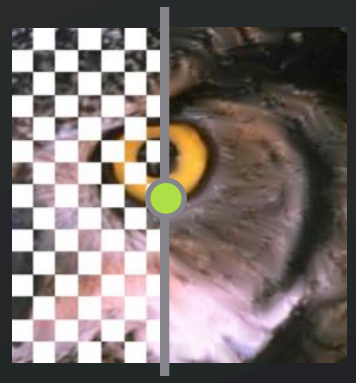
Denoising



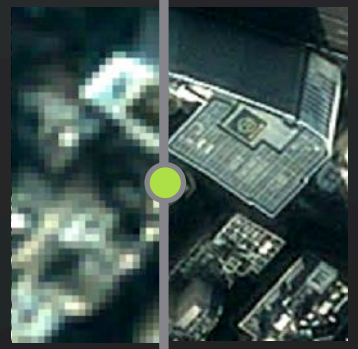
Deblurring



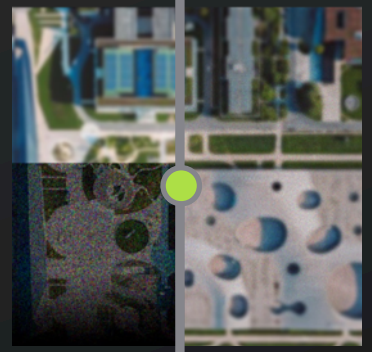
Inpainting



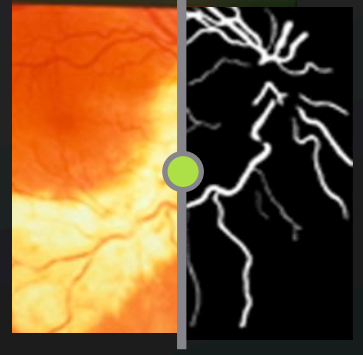
Super-resolution



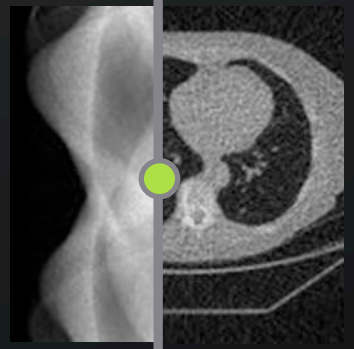
Fusion



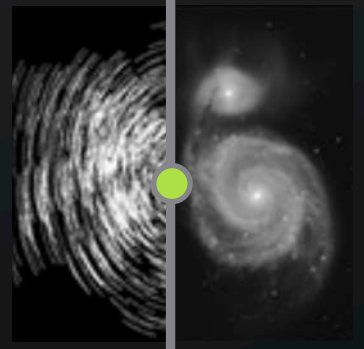
Filtering



Tomography



Interferometry



Demultiplexing

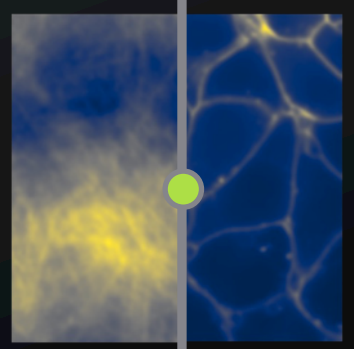
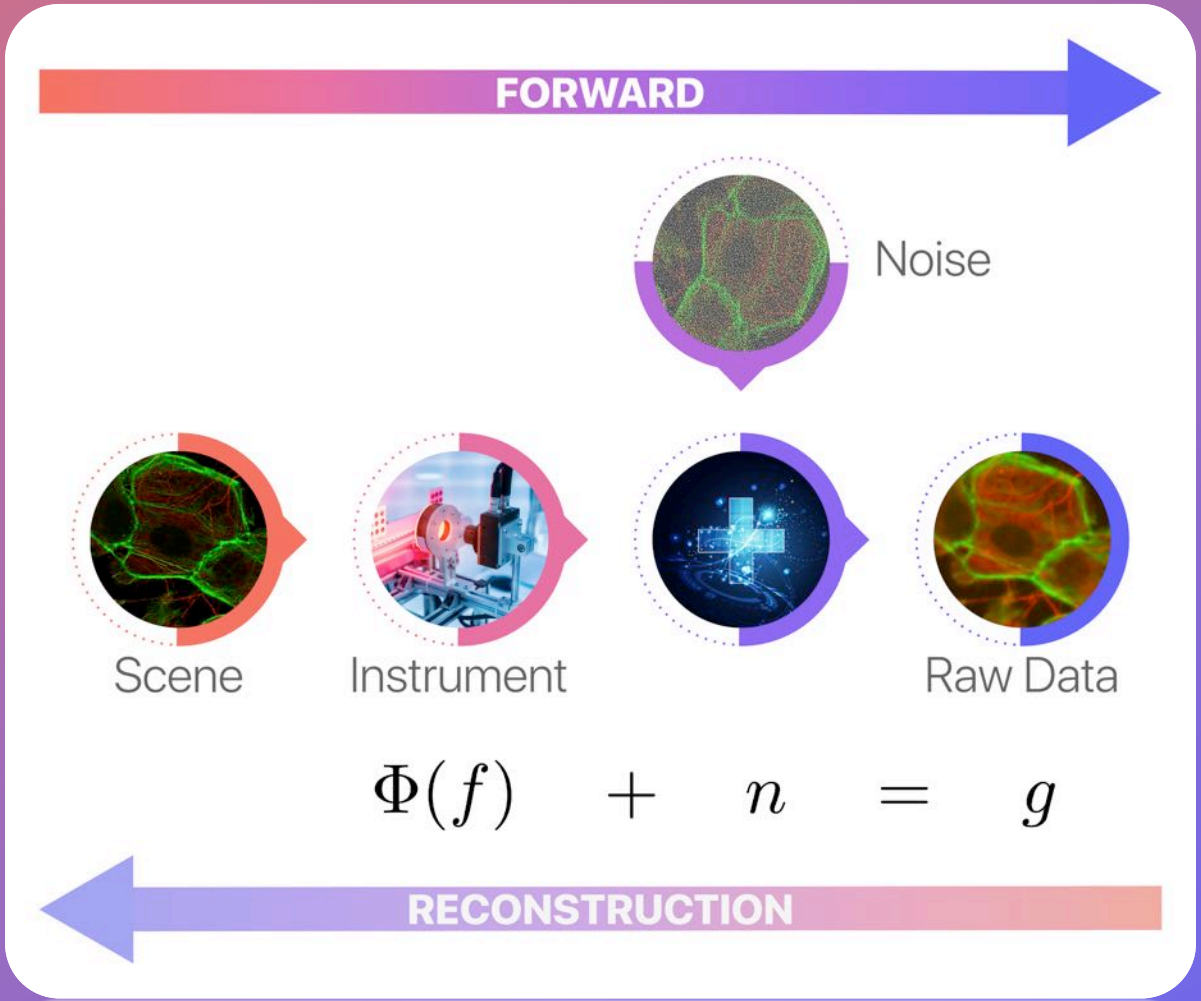


Image Reconstruction as an Inverse Problem 🔄

INVERTING THE ACQUISITION PROCESS

An object of interest — be it a cell, a scene, or something else — is observed through an acquisition system, such as a microscope or camera. This system collects data that is both noisy and blurred.

The goal is to reconstruct the original object from these imperfect measurements.



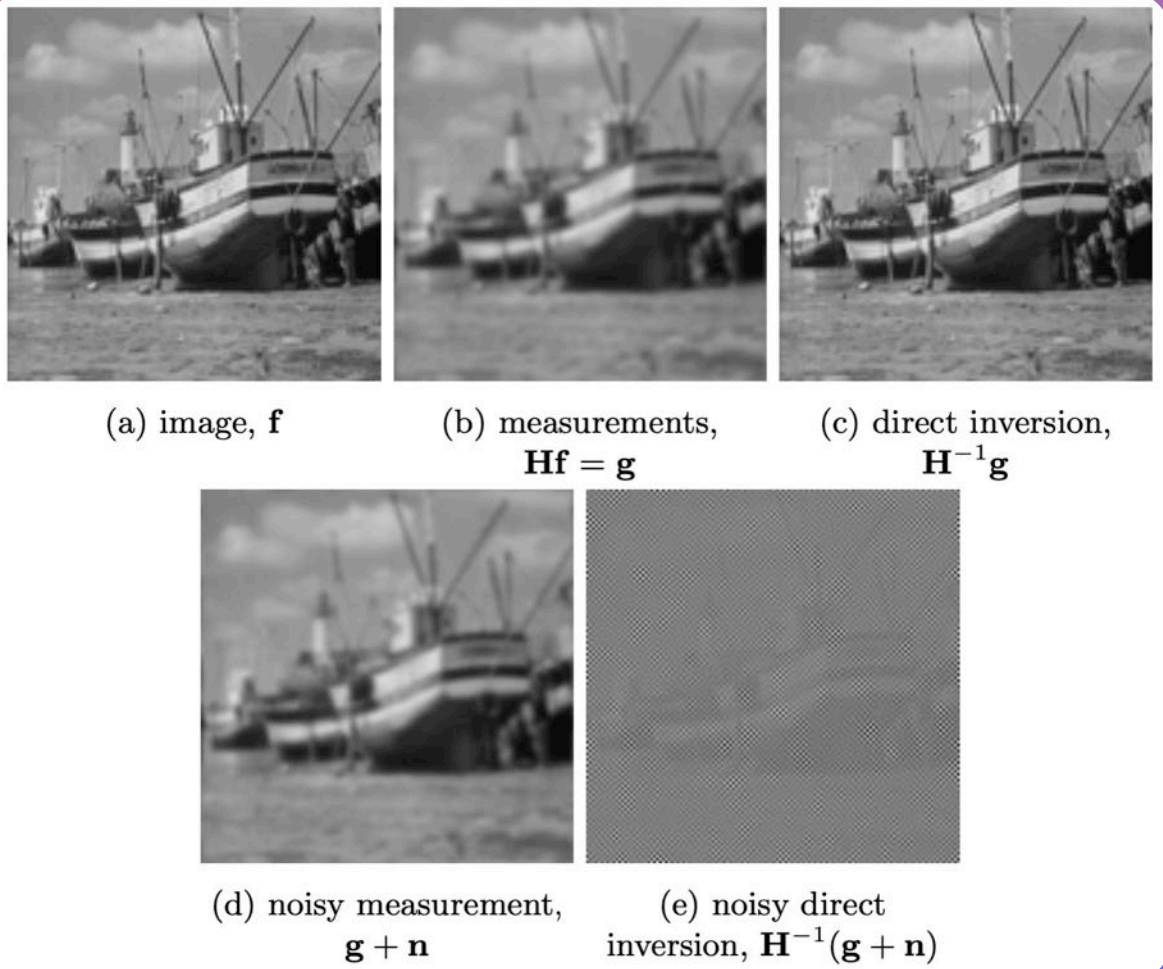
Inverse Problems are Ill-Posed 🧗

COMMON PITFALLS OF DIRECT INVERSION

Inverse problems are often ill-posed:

$$f: \quad g = \Phi(f)$$

There may exist no solution, more than one solution, and solutions may be numerically unstable!



(a) image, $\mathbf{f}$ (b) measurements, $\mathbf{Hf} = \mathbf{g}$ (c) direct inversion, $\mathbf{H}^{-1}\mathbf{g}$

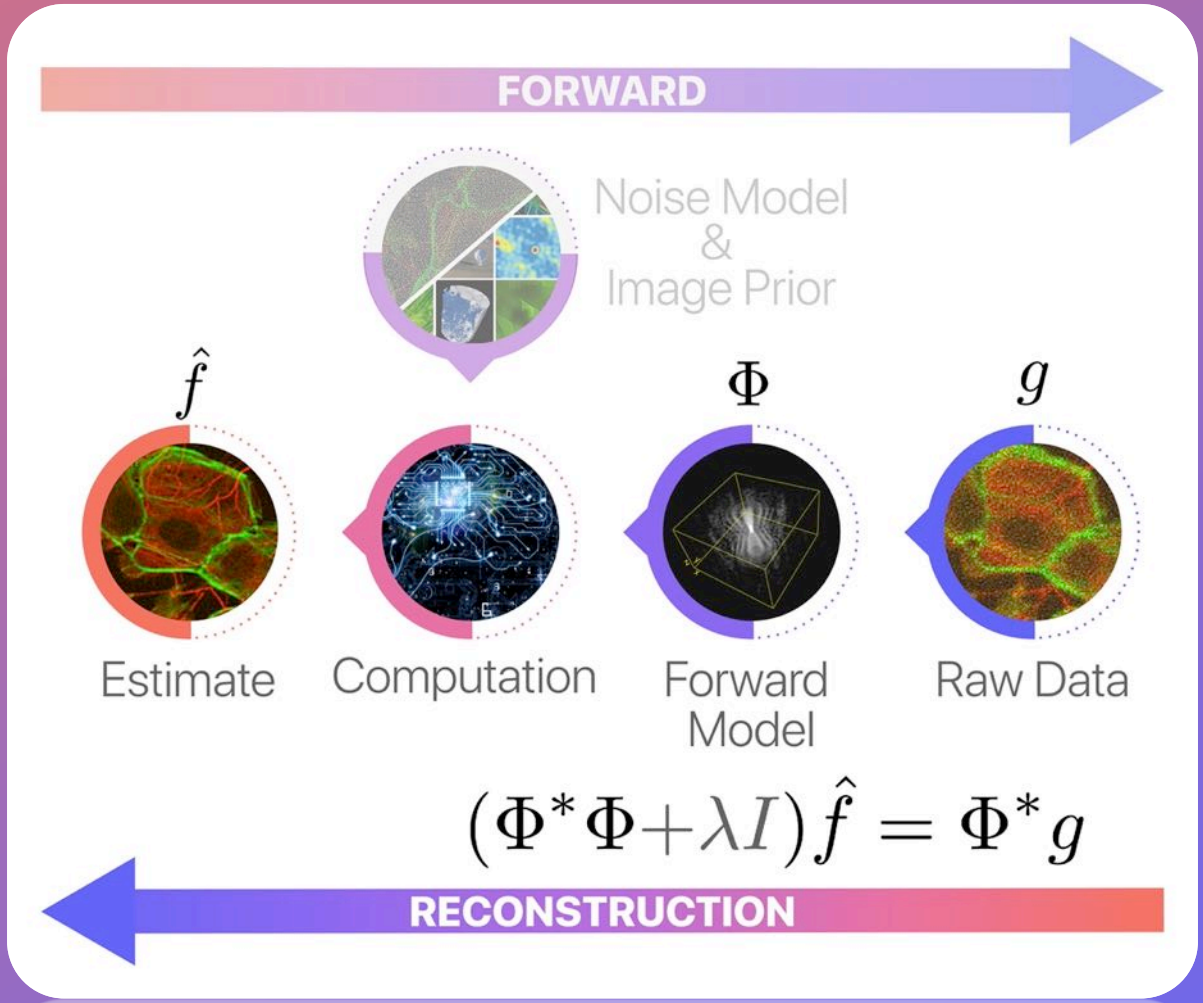
(d) noisy measurement, $\mathbf{g} + \mathbf{n}$ (e) noisy direct inversion, $\mathbf{H}^{-1}(\mathbf{g} + \mathbf{n})$

The Dawn of Computational Imaging 🌅

PSEUDO-INVERSION TECHNIQUES

Traditional imaging pipelines often rely on rough approximations of the pseudo-inverse of the sensing operator.

A key limitation of such schemes is their fixed-function design: they cannot incorporate a priori knowledge about the scene.

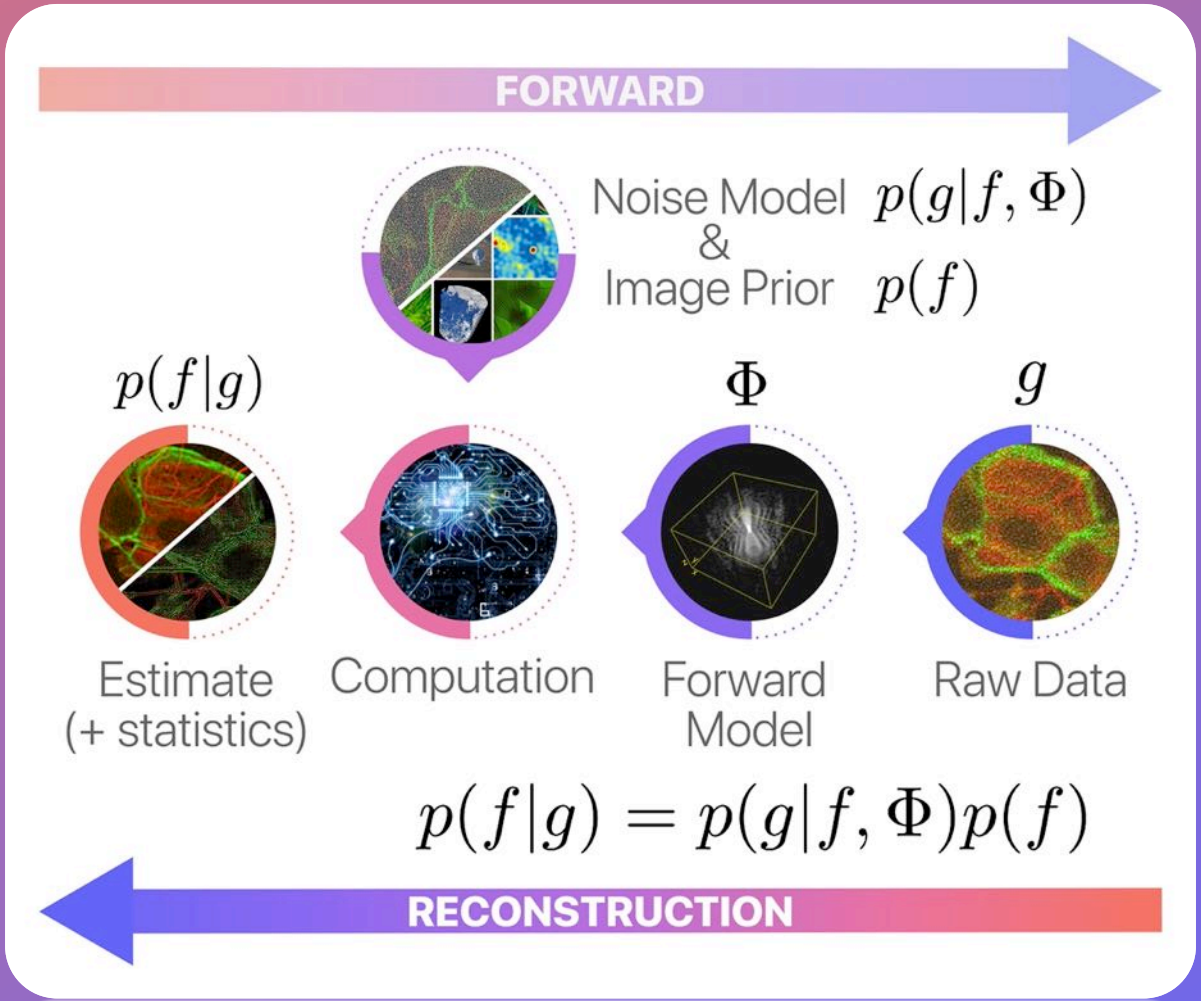


Bayesian Computational Imaging 🧠

LEVERAGING IMAGE PRIORS

Bayesian reconstruction methods assess an optimal trade-off between:

- a likelihood term, which controls the reconstruction fidelity in terms of the data and its statistics,
- a prior term, which promotes physical plausibility of the reconstruction.



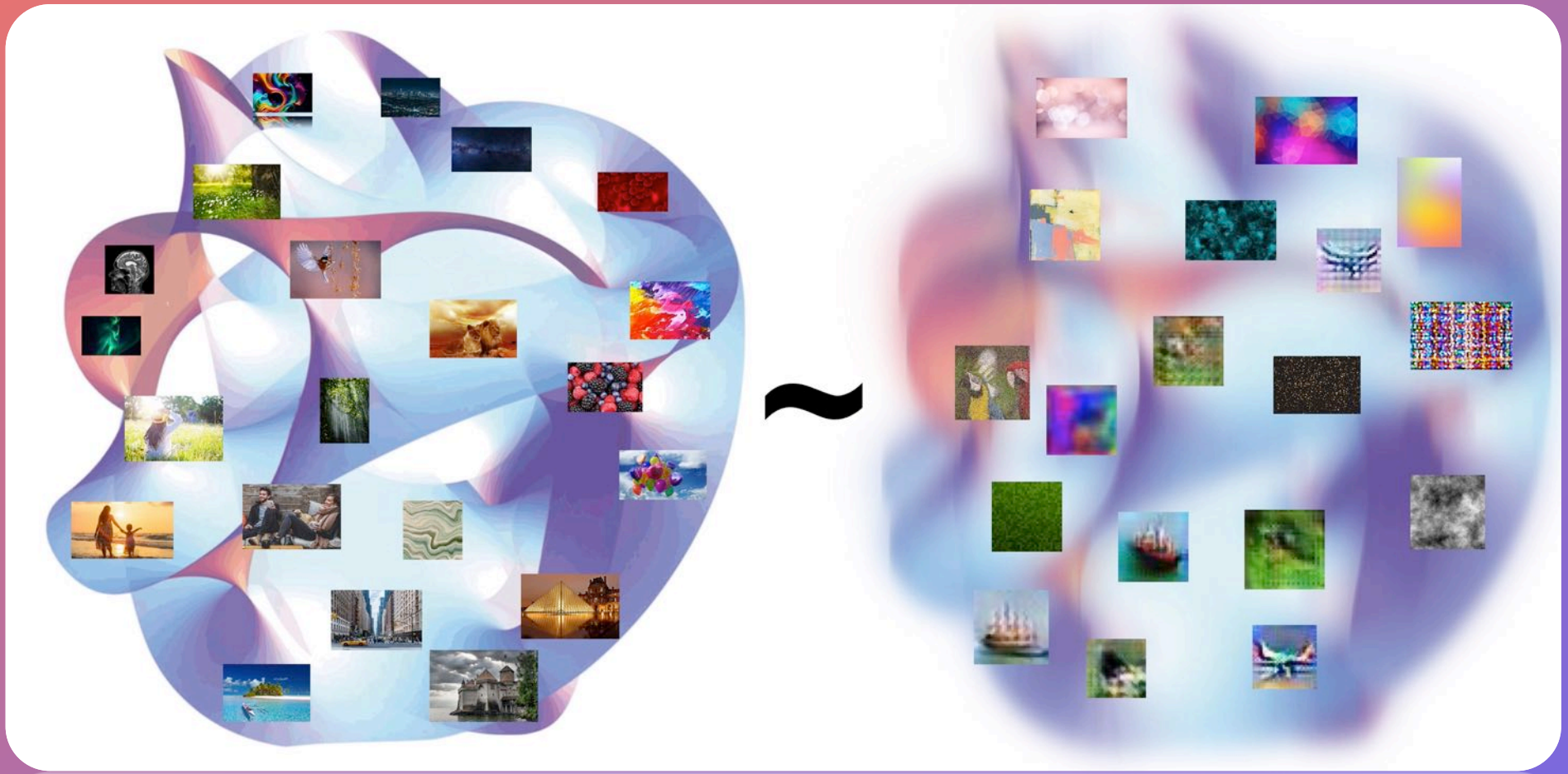
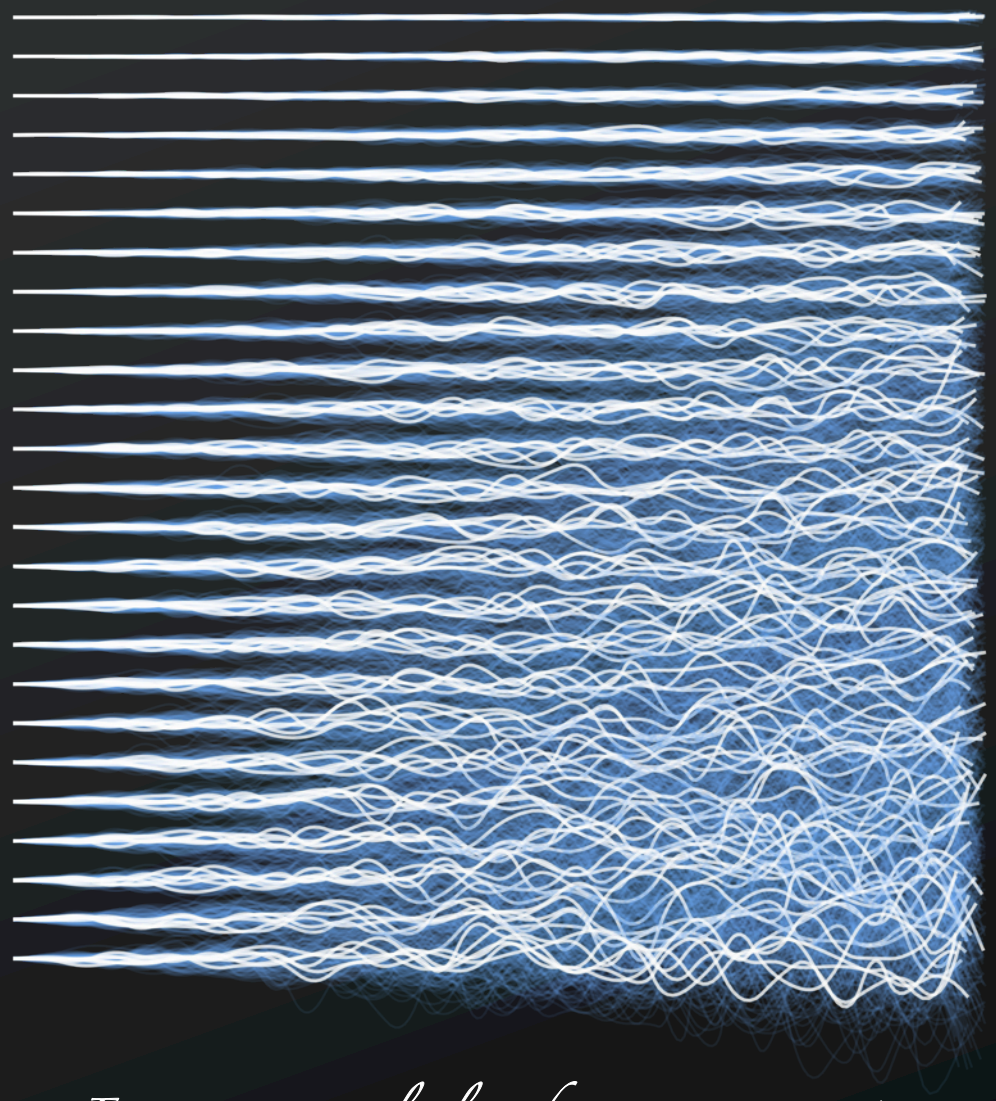


Image priors approximate the
Manifold of Natural Images 🌀

Occam's Razor Principle

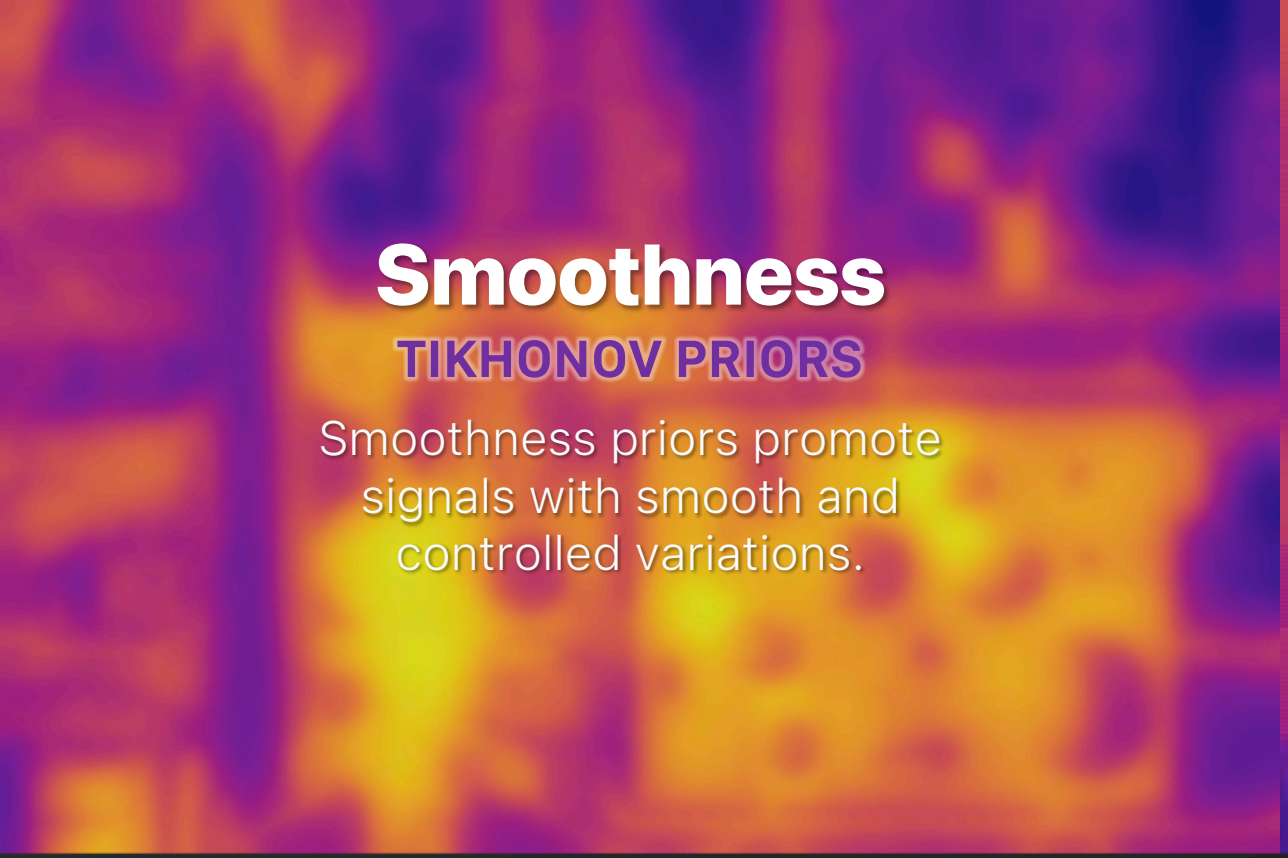
Lex Parsimoniae

When two equally good explanations for a given phenomenon are available, one should always favour the simplest, i.e. the one that introduces the least explanatory variables.



*Entia non sunt multiplicanda praeternecessitate**

**More things should not be used than are necessary*



Smoothness

TIKHONOV PRIORS

Smoothness priors promote signals with smooth and controlled variations.

Sparsity

TOTAL VARIATION PRIORS

Sparse priors encourage signals with few dofs: almost black or mostly flat objects.

Canonical prior:
Generalised Tikhonov.

$$R(f) = \| \mathcal{D} f \|_2^2$$

Canonical prior:
Generalised Total Variation.

$$R(f) = \| \mathcal{D} f \|_{TV}$$

$$\mathcal{D} \in \{\boldsymbol{\nabla}, \Delta, \Delta^n, (\alpha I - \Delta)^\beta, \dots\}$$

Image priors are more performant than ever... 💪

OCCAM'S RAZOR ON STEROIDS

> [Recent] documented results have led researchers to the educated guess that [modern image denoising] methods are getting very close to the optimally possible denoising performance.

Romano et al. 2017

RawNeRF in the Dark
Mildenhall et al. 2021, Google Research

...but the field suffers from an *adoptability crisis!* 🚧

BEYOND PROOF-OF-CONCEPTS AND HERO EXPERIMENTS?

The vast majority of the computational imaging methods proposed in the literature remain at the bare *proof-of-concept stage*, requiring expert knowledge to tune, use and/or deploy in production pipelines.

What are the main challenges and bottlenecks?

- Large ND images
- GPU/distributed computing
- Integration in GUI-based workflows
- Monolithic software
- Barriers to entry (e.g. maths, programming)
- Lack of cross-fertilisation
- Lack of trust, interpretability, intuition
- Large community with diverse needs

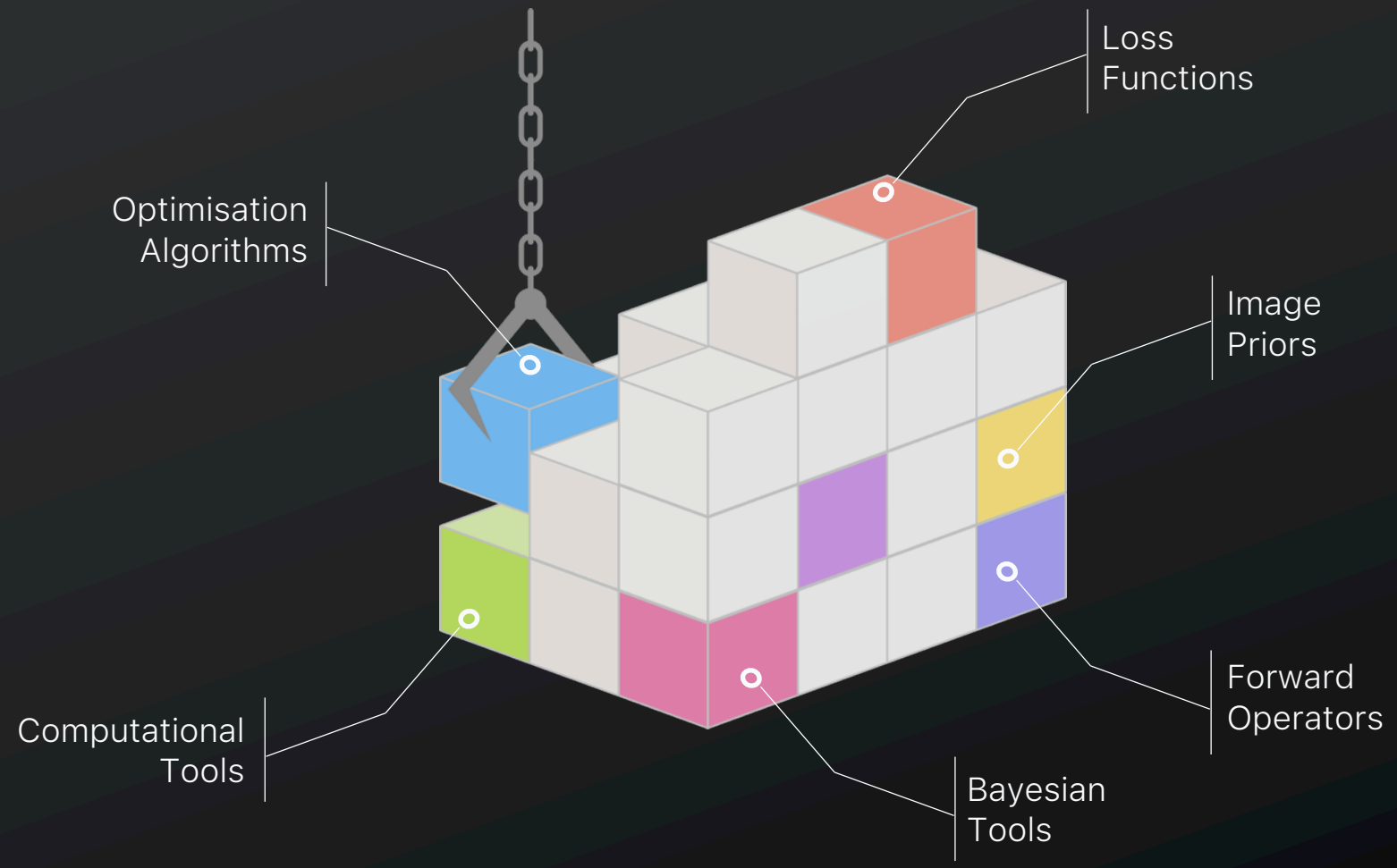
Tight integration of sensing and computing:
difficult to achieve in practice!

Capitalising on *modularity!* 🧩

UNIFYING COMPUTATIONAL IMAGING...BLOCKS BY BLOCKS

Most image reconstruction methods can be expressed in terms of a few fundamental *mathematical building blocks* (e.g., losses, priors, operators, etc). Thanks to their universality, such components can easily be reused across modalities.

Towards microservice imaging solutions...

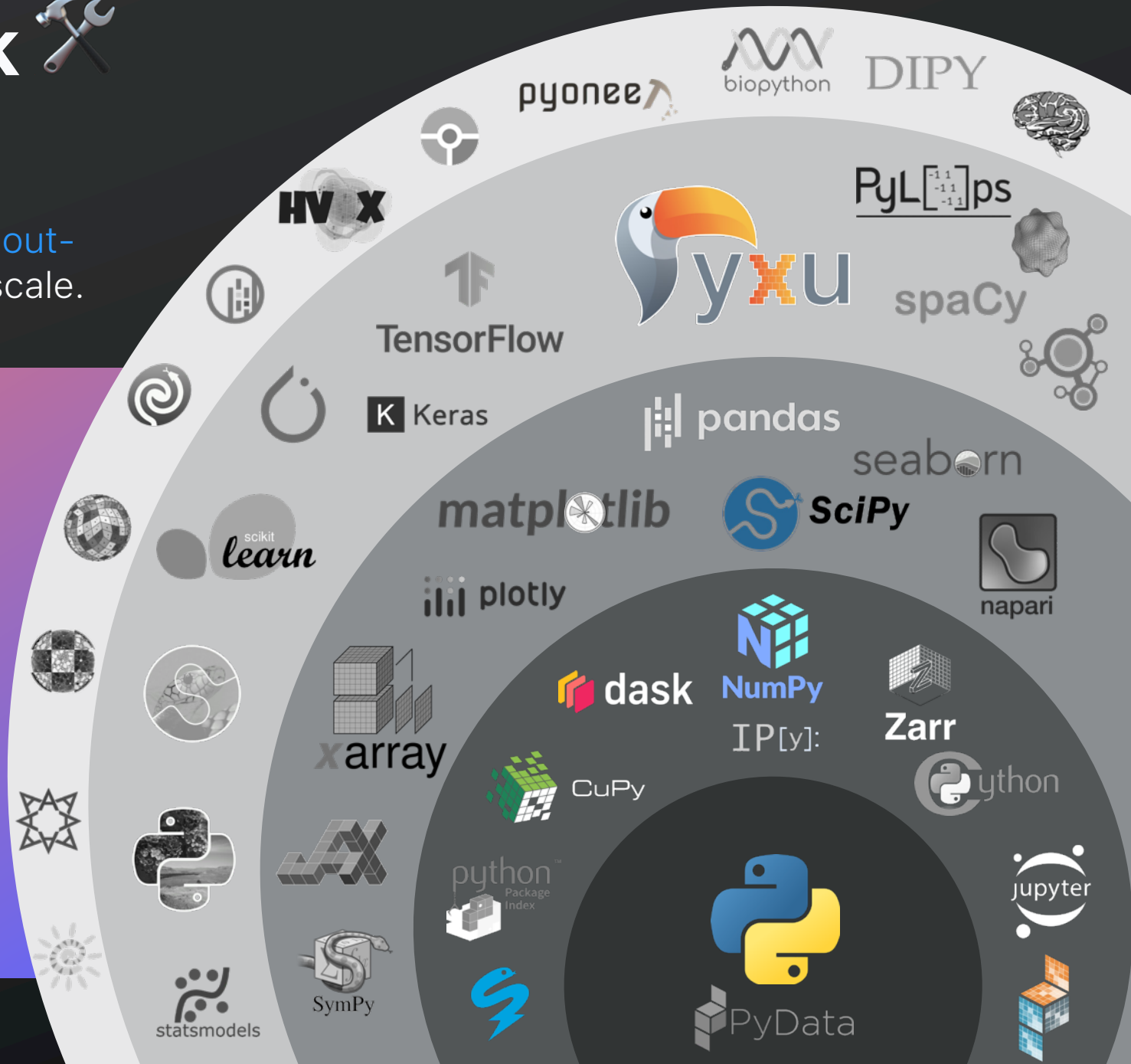


Maintainable, testable, optimised, loosely coupled, hardware-agnostic and highly applicable software components.

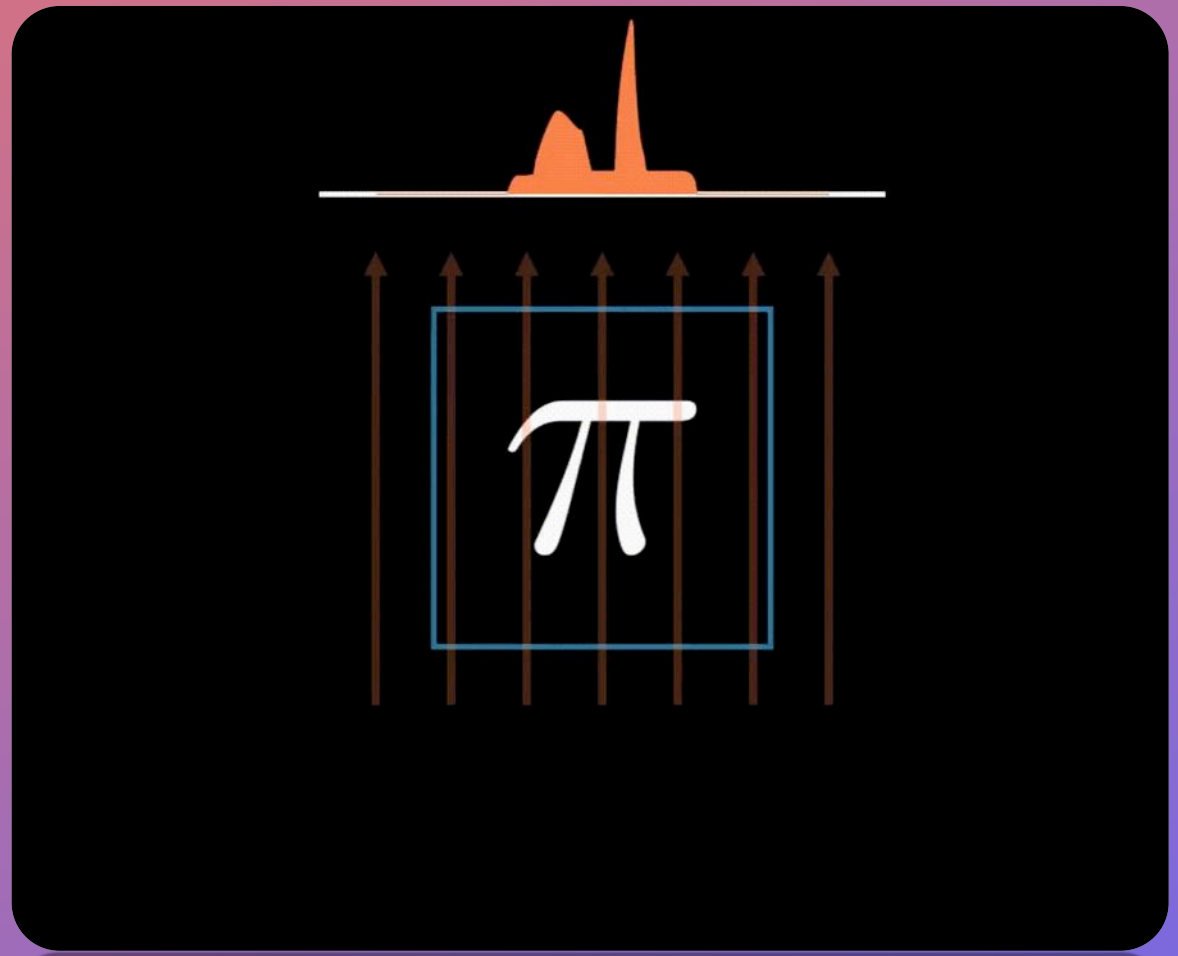
The Pyxu Framework

Pyxu is an open-source Python framework allowing scientists at any level to quickly prototype/deploy hardware accelerated and out-of-core computational imaging pipelines at scale.

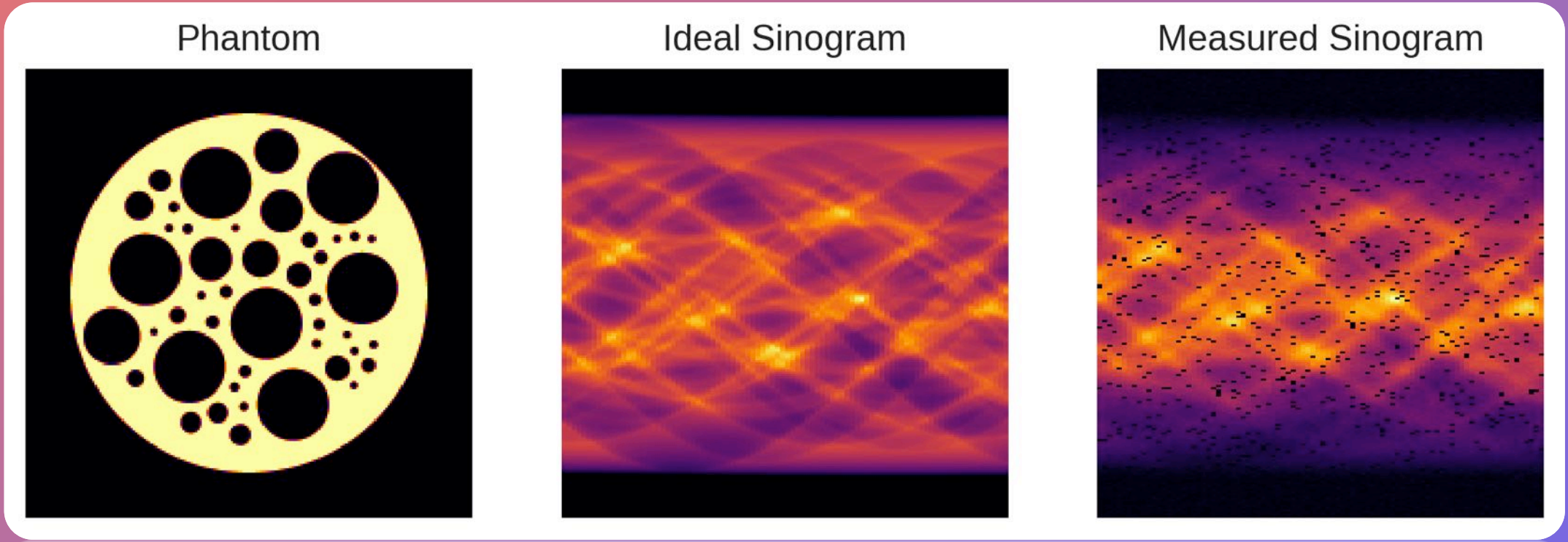
- Microservice architecture
- Plug-and-play API
- Application agnostic
- Backend agnostic
- Bottom-up HPC features
- Interoperable with PyData
- Test suite, docs, CI/CD
- Available on GitHub & PyPI
- Extendable (Pyxu FAIR)



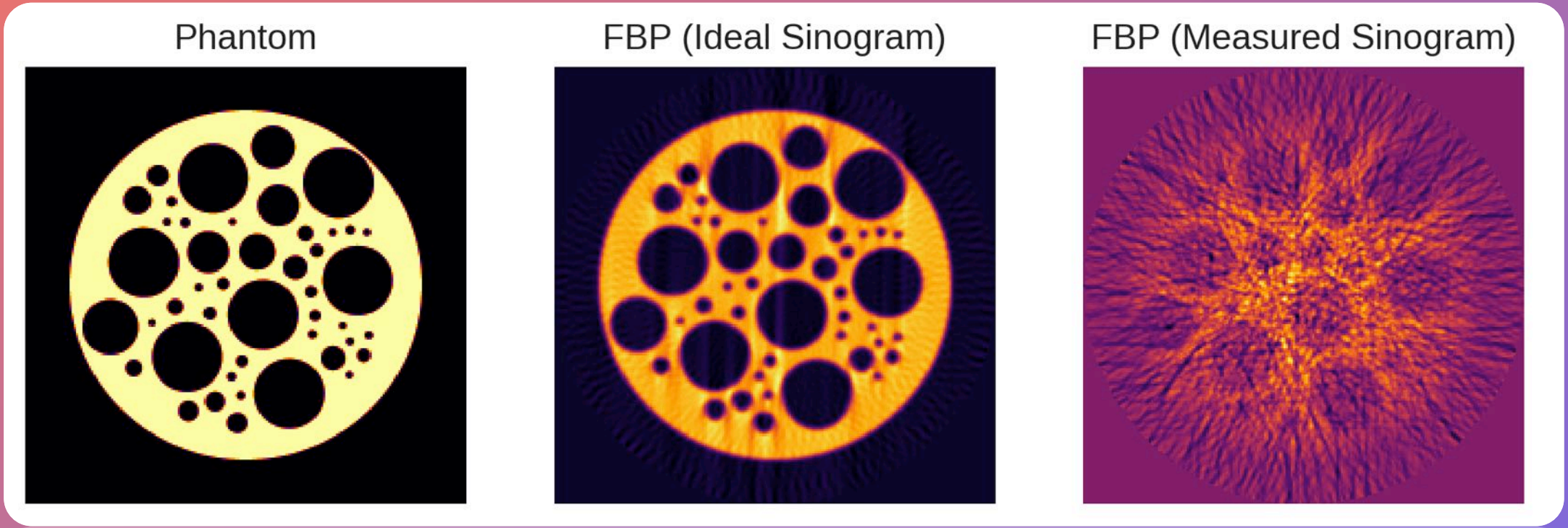
Example: Non-ideal Tomography 🔬



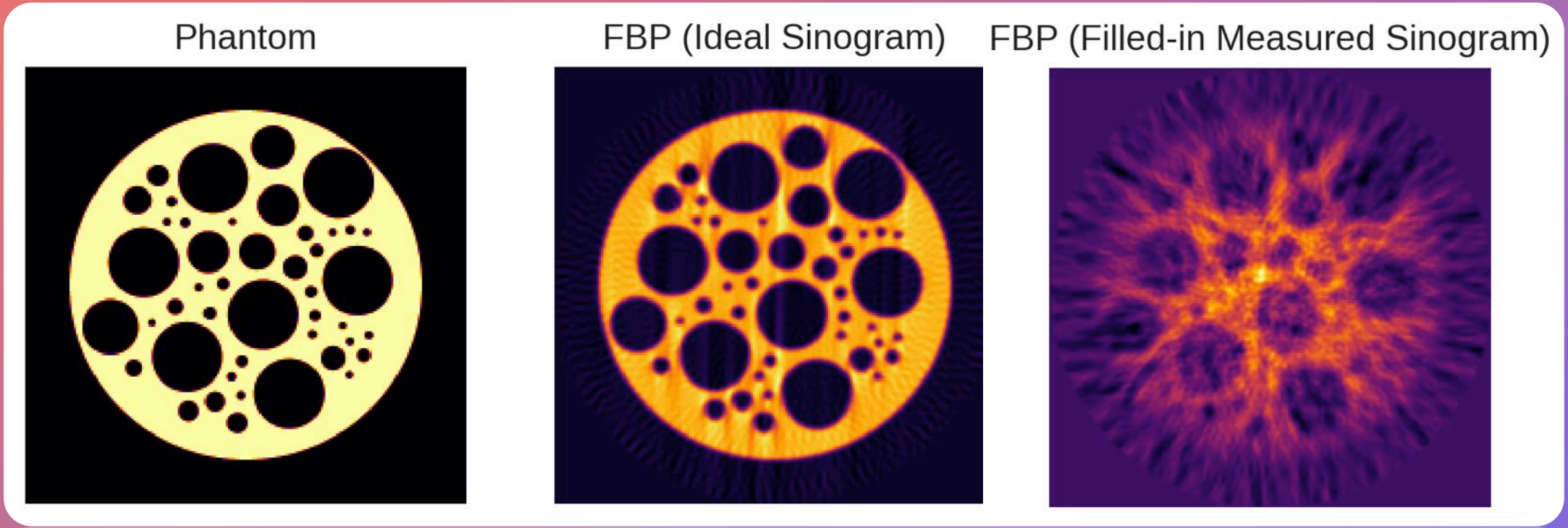
Example: Non-ideal Tomography 🔬



Many non-idealities: detector width, beam tapering and faulty detectors 🚨



Filtered Back-Projection does not account for the non-idealities of the tomographic setup: severe artefacts! 🚫



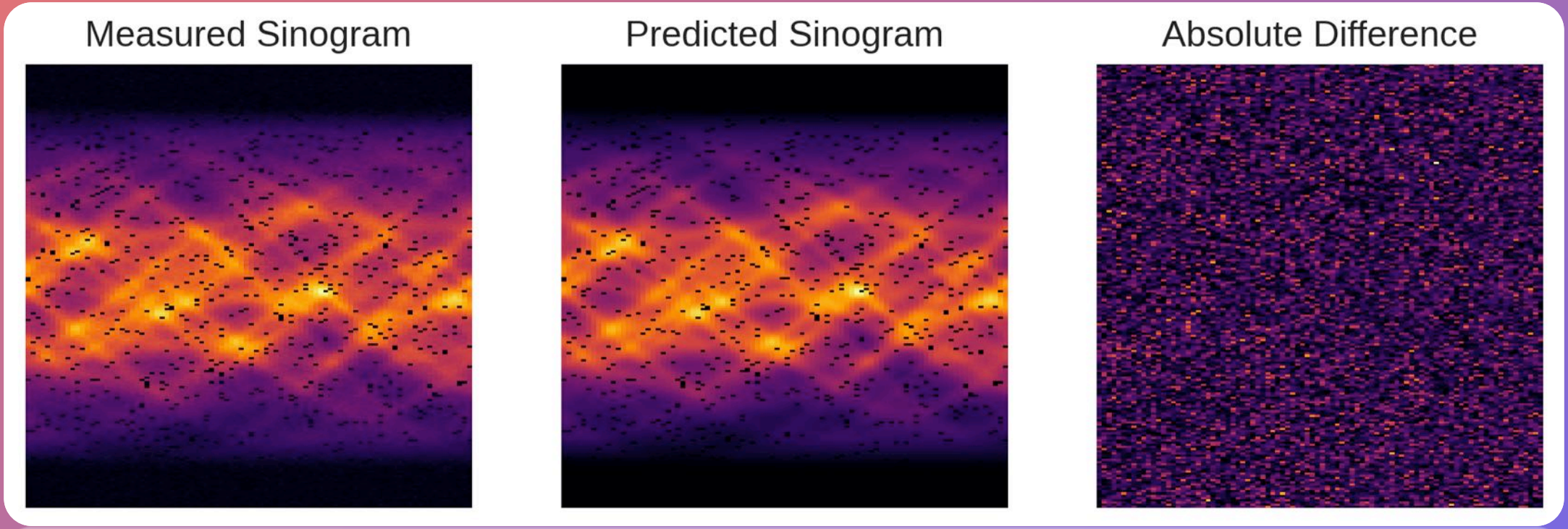
Filtered Back-Projection does not account for the non-idealities of the tomographic setup: severe artefacts! 🚫

Pyxu's Modularity to the rescue! 🦸

```python
import pyxu.abc as pxa
import pyxu.operator as pxo
from pyxu.operator.interop import from_source
import skimage.transform as skt

# Radon Operator (imported from `skt` since Pyxu did not ship with a Radon transform at the time of this writing.)
Radon = from_source(cls=pxa.LinOp,
                    shape=(sino.size, phantom.size),
                    apply=lambda _, arr: skt.radon(arr.reshape(phantom.shape),
                                                   theta=np.linspace(0, 180, angles),
                                                   circle=True).ravel(),
                    adjoint=lambda _, arr: skt.iradon(arr.reshape(sino.shape),
                                                      filter_name=None,
                                                      circle=True).ravel(),
                    vectorize=["apply", "adjoint"],
                    vmethod="scan",
                    enforce_precision=["apply", "adjoint"])

# 1D Filtering
boxcar = np.asarray(sp.signal.get_window("boxcar", wsize)); boxcar /= wsize
BoxCar1D = pxo.Stencil(kernel=[boxcar, np.array([1.0])], center=(wsize // 2, 0), arg_shape=sino.shape,)

# Partial Masking
Mask = pxo.DiagonalOp(mask.ravel())

# Tapering
taper = np.outer(sp.signal.get_window("hamming", sino.shape[0]), np.ones(sino.shape[1]))
Taper = pxo.DiagonalOp(taper.ravel())

# Compose operators
Phi = Taper * Mask * BoxCar1D * Radon
```

$$\Phi = T \circ M \circ F \circ R$$

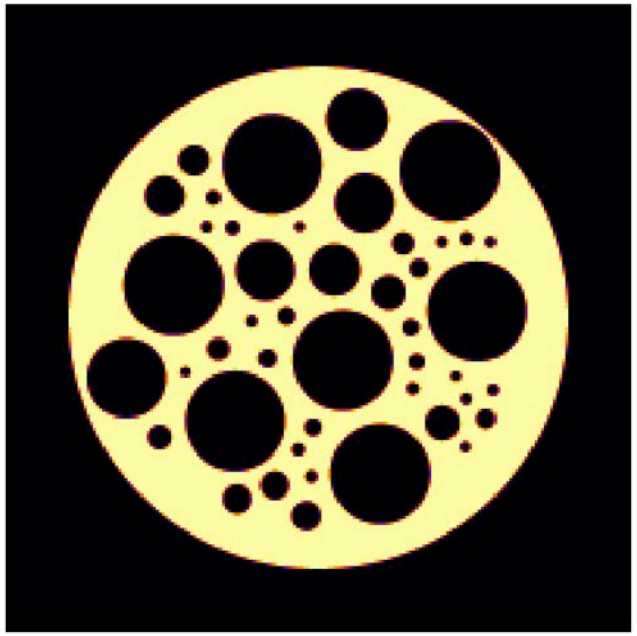
```
predicted_sino = Phi(phantom.ravel()).reshape(sino.shape)
```



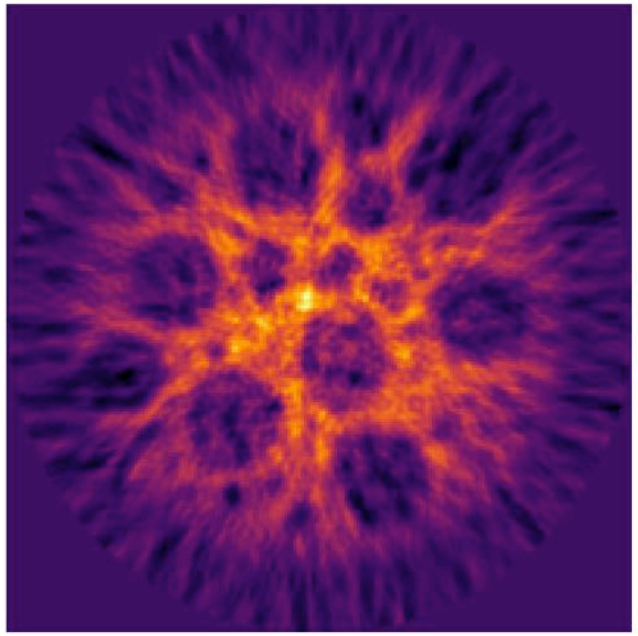
The predicted and measured sinograms are nearly indistinguishable, aside from minor noise variations ✅.

```python
import pyxu.opt.stop as pxst

stop_crit = pxst.RelError(eps=1e-3, var="x", f=None, norm=2, satisfy_all=True) || pxst.MaxIter(500)
pinv_solution = Phi.pinv(sino.ravel(), damp=4,
                         kwargs_init=dict(show_progress=False, verbosity=50),
                         kwargs_fit=dict(stop_crit=stop_crit))
```

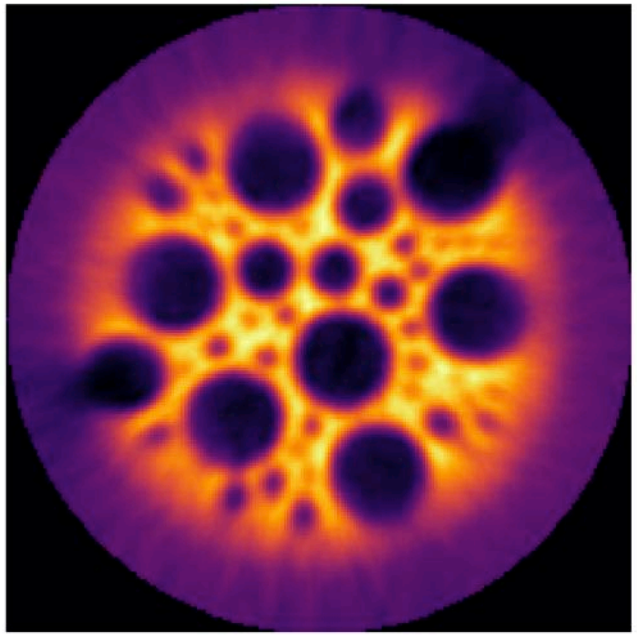
Phantom



Filtered Back Projection



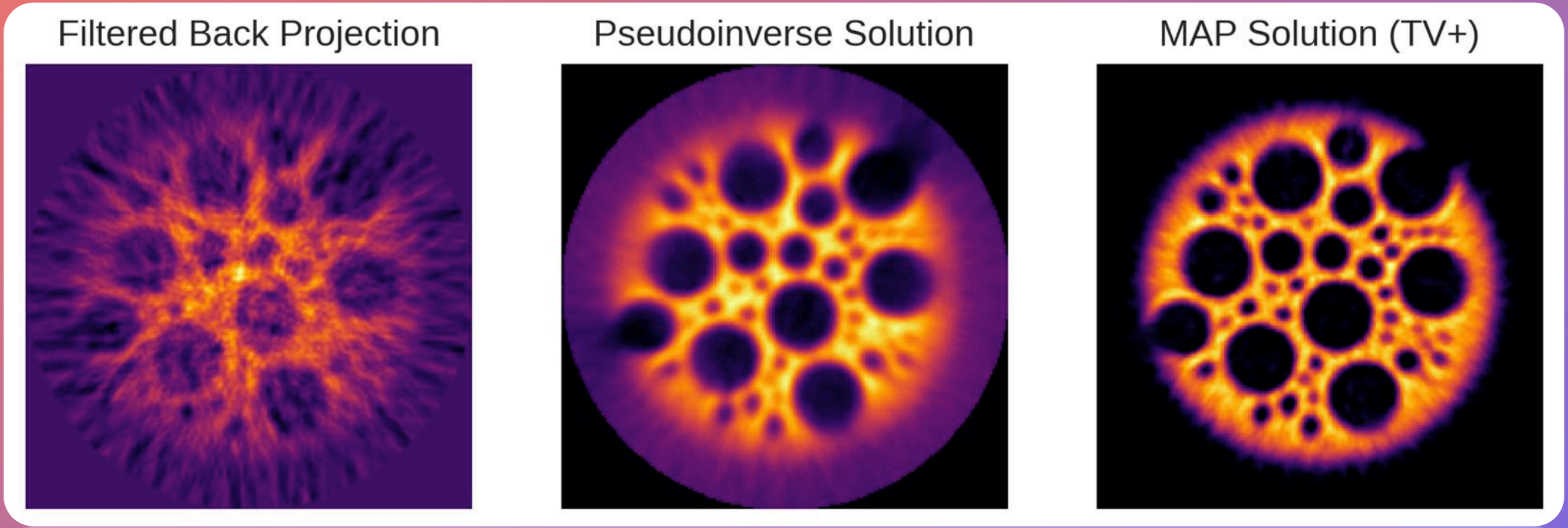
Pseudoinverse Solution



Bayesian Inversion

$$\hat{f} \in \arg\min_{f \geq 0} \frac{1}{2\sigma^2} \|g - \Phi(f)\|_2^2 + \lambda \|\nabla f\|_1 + \mu \|f\|_1$$

```python
from pyxu.operator import Gradient, SquaredL2Norm, L1Norm, PositiveL1Norm
from pyxu.opt.solver import PGD

# TV prior
grad = Gradient(arg_shape=phantom.shape, accuracy=4, mode='constant', diff_method='fd')
lambda_= 0.05 / (2 * sigma**2)
huber_norm = L1Norm(grad.shape[0]).moreau_envelope(0.01)  # We smooth the L1 norm to facilitate optimisation
tv_prior = lambda_ * huber_norm * grad

# Positivity + L1 norm
posL1 = 0.05 * PositiveL1Norm(phantom.size)

# Loss
loss = (1/ (2 * sigma**2)) * SquaredL2Norm(dim=sino.size).asloss(sino.ravel()) * Phi

# Smooth part of the posterior
smooth_posterior = loss + tv_prior
smooth_posterior.diff_lipschitz = smooth_posterior.estimate_diff_lipschitz(method="svd", tol=0.1)

# Define the solver
solver = PGD(f=smooth_posterior, g=posL1, show_progress=False, verbosity=250)

# Call fit to trigger the solver
solver.fit(x0=0*pinv_solution.ravel(), acceleration=True, stop_crit=stop_crit)
recon_tv = solver.solution().squeeze()
```



With only minimal coding, we have substantially improved various facets of image quality, including resolution, contrast, and artifact removal! ✨

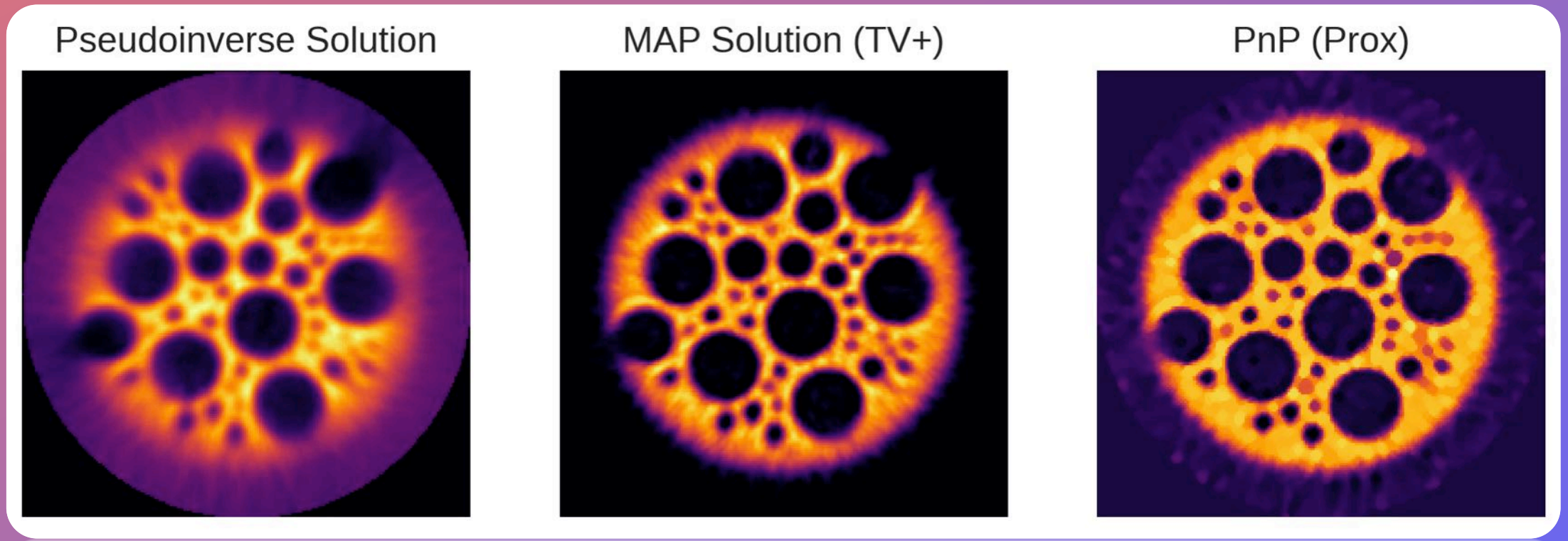
Plug-and-Play (PnP) Priors 🎮

$$f^{k+1} = f^k - \tau\left[\nabla F(f^k) + \lambda(f^k - \mathrm{Denoiser}_\sigma(f^k))\right],$$

$$f^{k+1} = \mathrm{Denoiser}_\sigma(f^k - \tau\nabla F(f^k)),$$

```python
from scipy.ndimage import median_filter

def denoiser(arr, arg_shape, wsize):
    arr = arr.reshape(arg_shape)
    return median_filter(arr, size=wsize).ravel()

class MedianFilterPnP(pxa.ProxDiffFunc):
    def __init__(self, arg_shape, wsize):
        super().__init__((1, np.prod(arg_shape)))
        self._arg_shape = arg_shape
        self.wsize= wsize
        self.diff_lipschitz = 1

    def apply(self, arr):
        return NotImplemented  # PnP priors are implicit: they do not have an apply method!

    def grad(self, arr):
        return arr - denoiser(arr, self._arg_shape, self.wsize)  # Differential form

    def prox(self, arr, tau=None):  # `tau`` is not used since this is not a proper proximal operator.
        return denoiser(arr, self._arg_shape, self.wsize)  # Proximal form

medpnp = MedianFilterPnP(phantom.shape, (3,3))
```

```python
# Define the solver
solver = PGD(f=loss, g=medpnp, show_progress=False, verbosity=75)

# Call fit to trigger the solver
solver.fit(x0=pinv_solution.ravel(), acceleration=True, stop_crit=pxst.MaxIter(250))
recon_pnp_prox = solver.solution().squeeze()
```



Key Takeaways & Acknowledgements 🙏

Universal & Modular 🌐

General purpose and flexible framework: no more wrestling with one-size-fits-all solutions that don't quite fit!

Plug-and-Play 🎮

Craft imaging pipelines effortlessly with advanced operator algebra logic. Pyxu automates the tedious bits.

High-Performance 🚀

Whether you're using CPUs or GPUs, Pyxu works with both. It employs Duck arrays and JIT compilation via Numba.

DOC

Distributed Computing 🔗

Got a lot of data? No worries! Pyxu works at scale: it supports Dask arrays for distributed/out-of-core computing needs.

Deep Learning 🤖

Pyxu is interoperable with major deep learning frameworks like PyTorch and JAX, making it easy to use learnt PnP priors.

REPO

M. Simeoni

S. Kashani

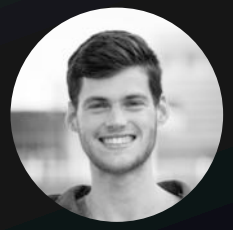
J. Rué-Q.

T. Debarre

D. Hamm

N. Salim

A. Jarret


A. Flowers

K. Okumus