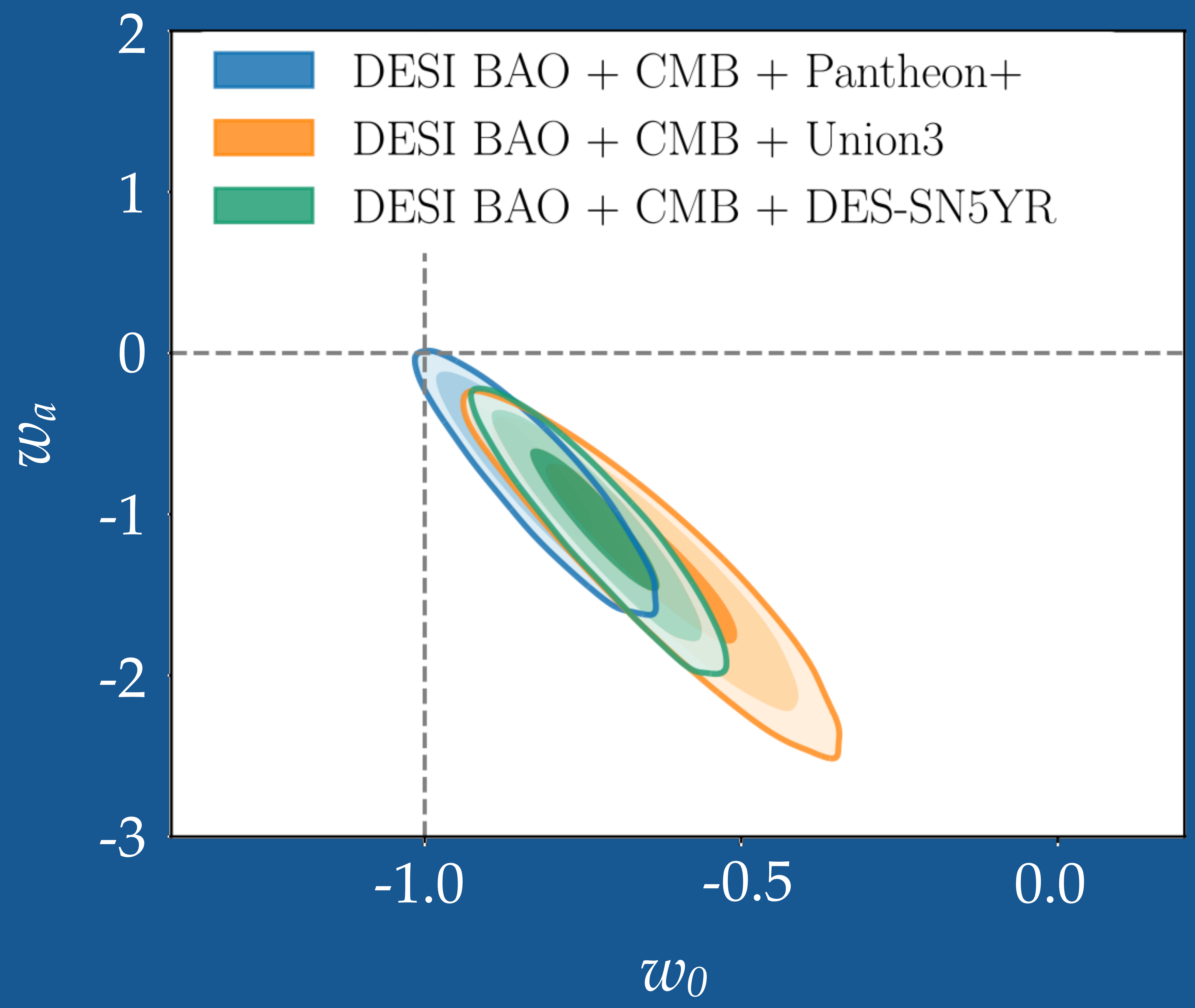
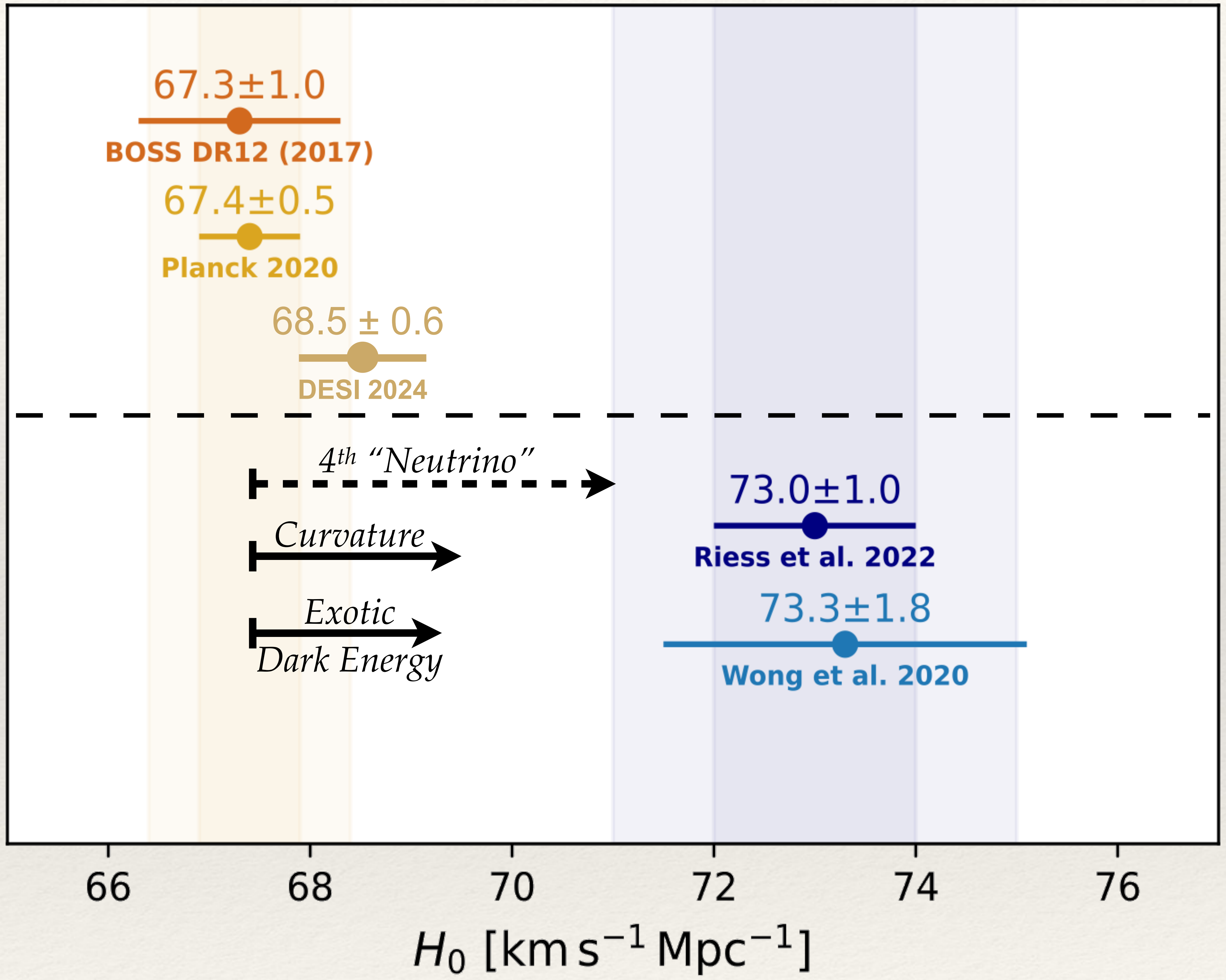
ZTF &
An applied data-science tasks force

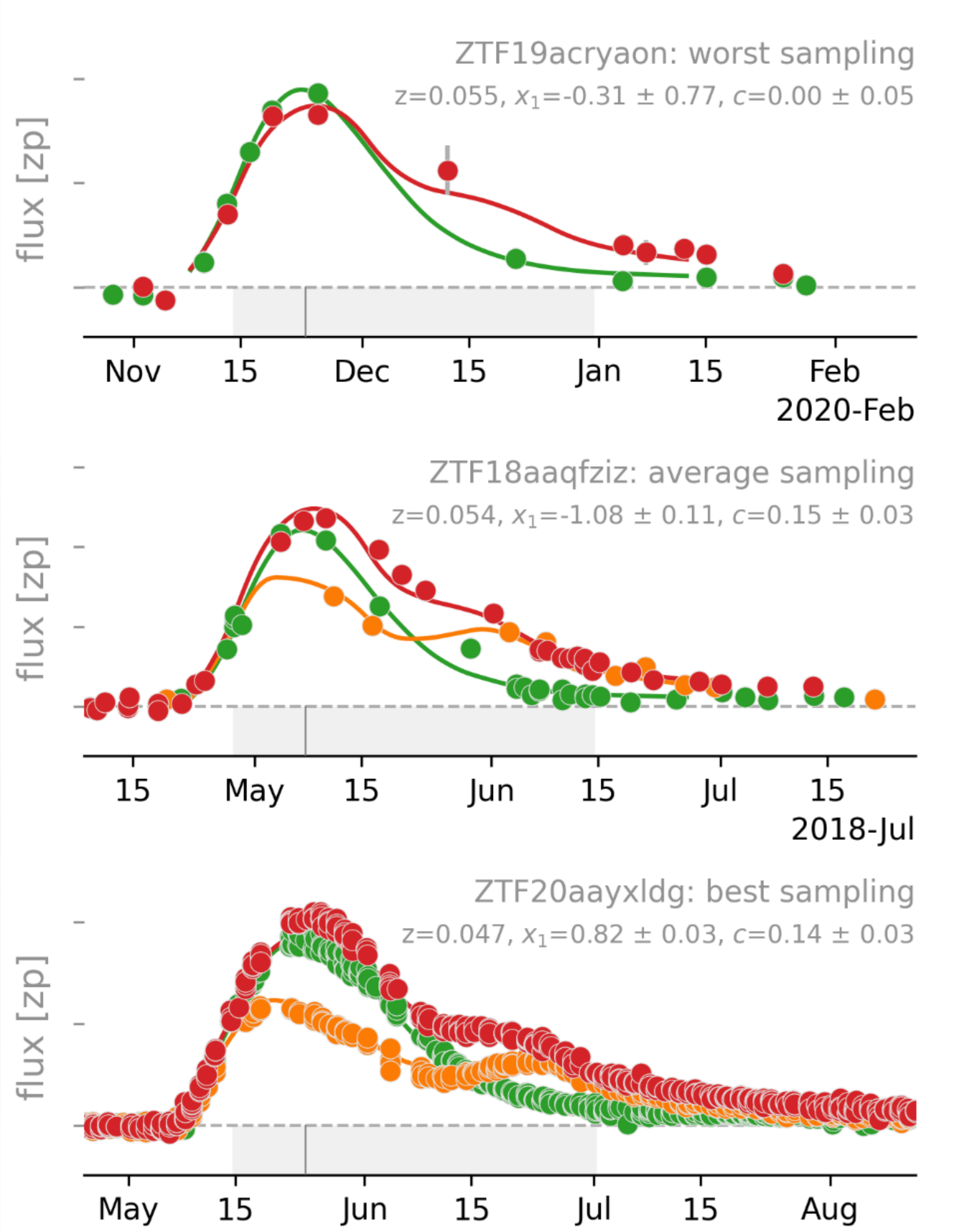
m.rigault@ipnl.in2p3.fr

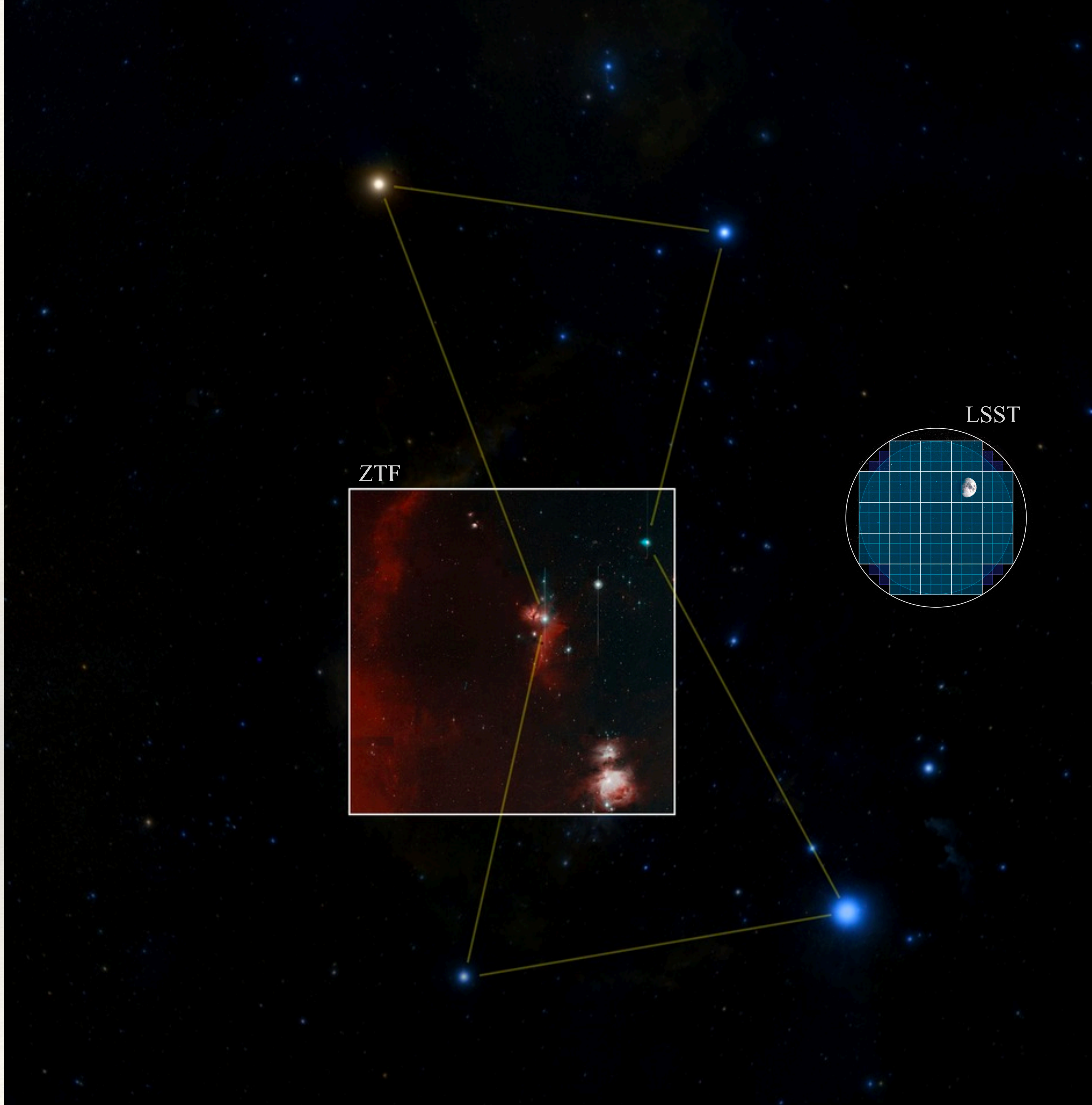
The new face of precision cosmology





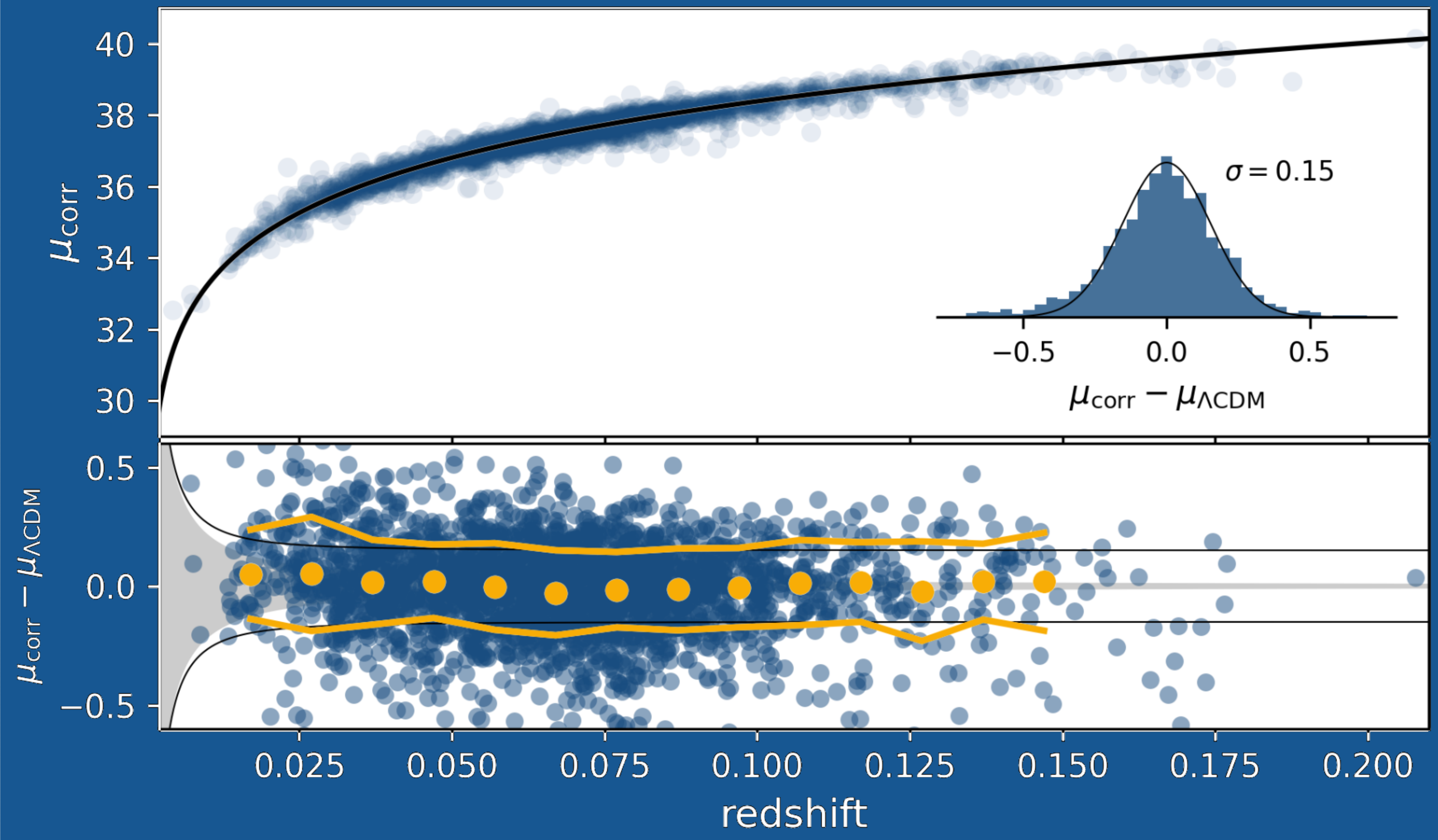
ZTF has impressive data





Survey 3750 deg² per hour

ZTF SN Ia DR2 | ~2700 Cosmology SNe Ia

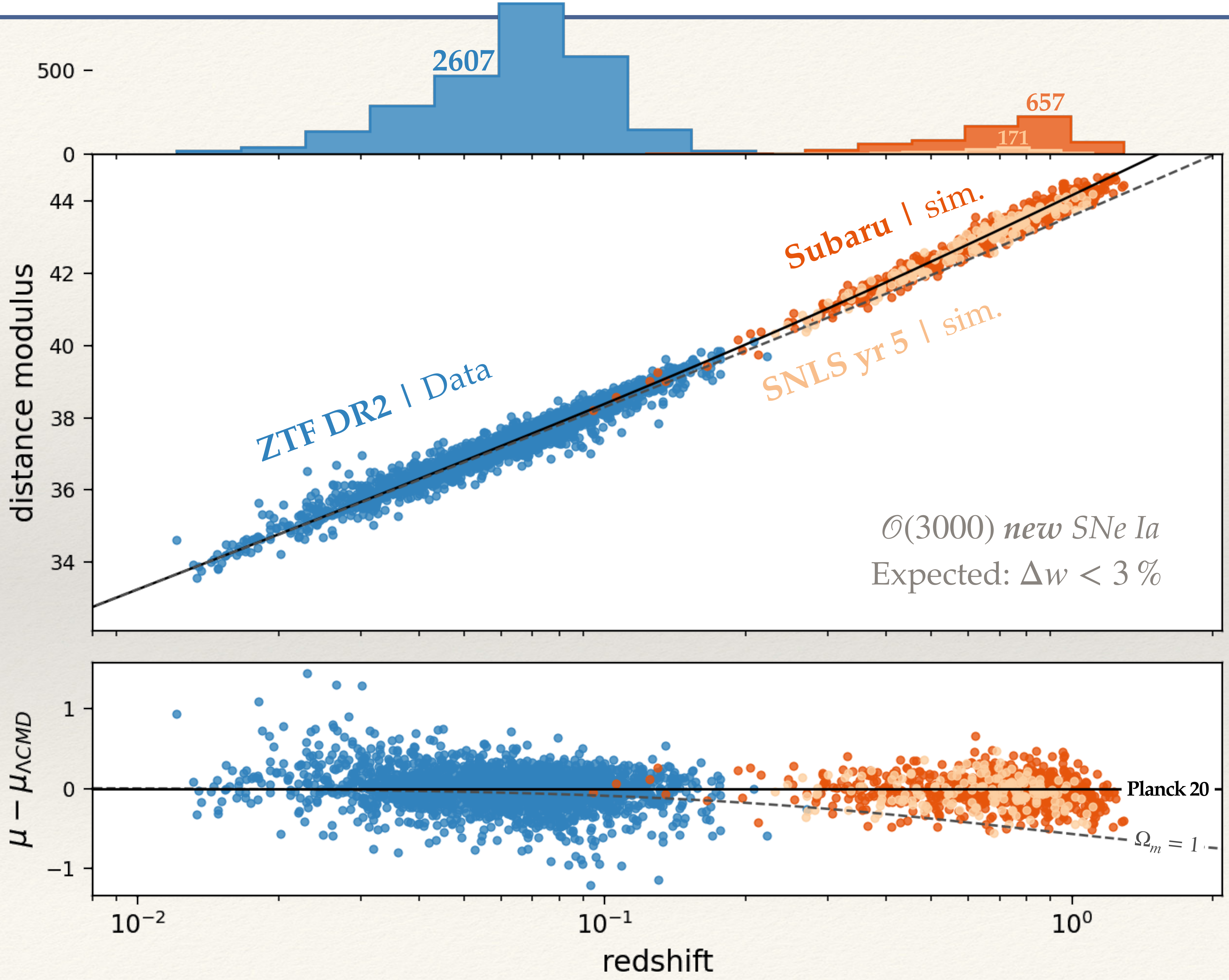


A&A Special Issue

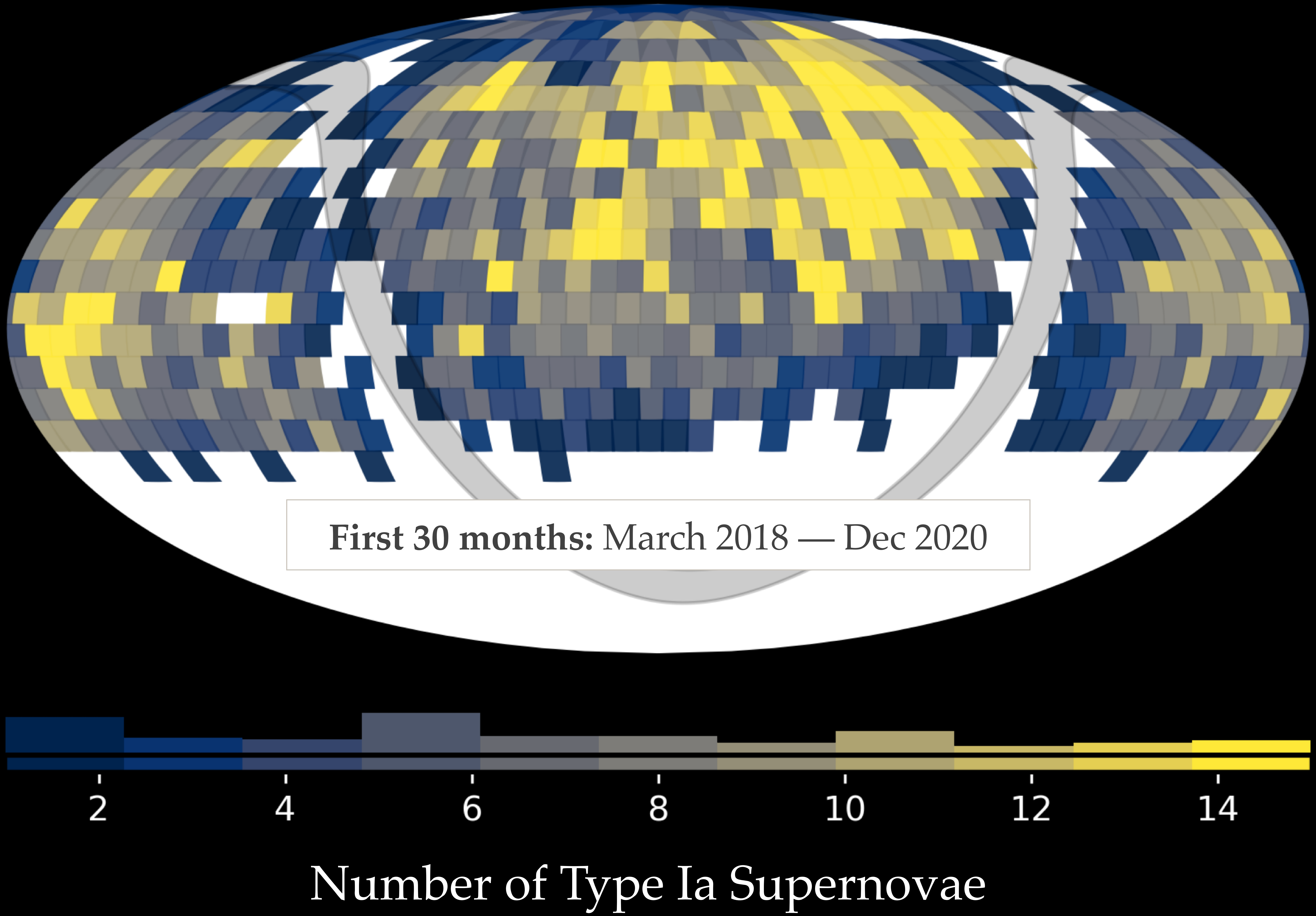
First Author	Short title
Rigault (a, this work)	DR2 overview
Smith	DR2 data review
Lacroix	DR2 photometry
Johansson	DR2 spectra review
Rigault (b)	Light-curve residuals
Kenworthy	Light-curve modeling
Amenouche	DR2 sample simulations
Ginolin (a)	Host, stretch & steps
Ginolin (b)	Host, color & bias origin
Popovic	Host & color evolution
Dhawan	SNe Ia siblings
Ruppin	SNe Ia in clusters
Aubert	SNe Ia in voids
Carreres	Velocity systematics
Burgaz (a)	SN Ia spectral diversity
Dimitriadis	Thermonuclear SN diversity
Terwel	Late-time CSM interaction
Harvey	High-velocity features
Deckers	Secondary maxima
Burgaz (b)	SNe Ia in low-mass hosts
Senzel	Bulge vs. Disk SNe Ia

The *Lemaitre* Diagram

ZTF DR2.5 | In prep



None of these SNe Ia have ever been used for cosmology

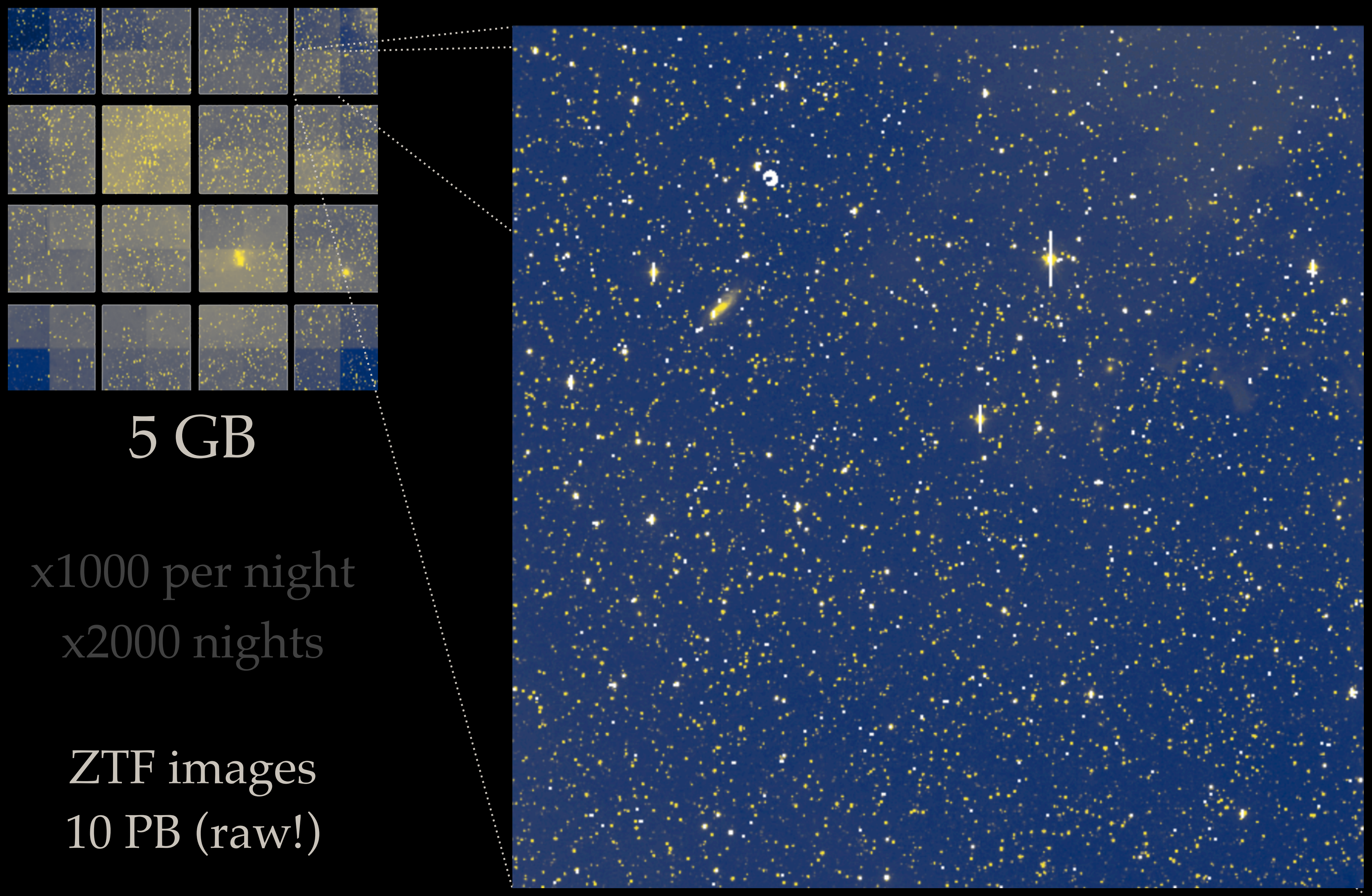

First 30 months: March 2018 — Dec 2020
2
4
6
8
10
12
14
Number of Type Ia Supernovae

Rigault et al. in prep
2
4
6
8
10
12
14
Number of Type Ia Supernovae
ZTF III | 10k SNe Ia
9 000
Planned
7 500
On disc
6 000
DR2
Spectroscopically typed « cosmo » SNe Ia
3 000
Current SNeIa Data (20 years)
Current z<0.15 SNeIa Data
ZTF I
ZTF II
ZTF 04
ZTF III
LSST
→Rolling
2019
2020
2021
2022
2023
2024
2025
2026
2027

mid 2024 | DR2 release
3 000 SNeIa | Fully spectro
(Incl. 5000 Spectra | host properties | etc)
mid 2025 | DR2.5 release
+ Photometric calibration
→ Cosmology | fσ8 | "Lemaitre" Diagram (w)
end 2026 | DR3 release
30 000 SNe Ia | photo-typed
→ Cosmology
Time
40 000 SNe Ia
DR3
30 000
15 000
Photometrically typed SNe Ia
9 000
6 000
3 000
DR2
Spectro SNe Ia
Current SNeIa Data (20 years)
ZTF I
ZTF II
ZTF 04
ZTF III
LSST
→Rolling
2019
2020
2021
2022
2023
2024
2025
2026
2027


ZTF Camera
5 GB
x1000 per night
x2000 nights
ZTF images
10 PB
raw
bias
flat
flat
flat
"Master" Calibration per period (~1 month)
bias
flat
flat
flat
Science Image
+ non linearity model | + pixel bias correction
(table & kernel to implement)
PSF model
wcs
From IPAC
catalogs
PSF
(stars & SN)
–
~300 sources
Aperture
(stars & SN)
–
~300 sources
Aperture
(stars)
x 23 millions images
Solve
Star catalog
raw -> science
science -> stars
Stars -> calib
science -> SN*
SN* -> SN
x 40 000 SNeIa
x 1000 images
PSF cat
(stars & SN)
PSF cat
(stars & SN)
PSF cat
(stars & SN)
Stars
Stars
Stars
SN
SN
SN
Stored
On the fly

ZTF Camera



5 GB

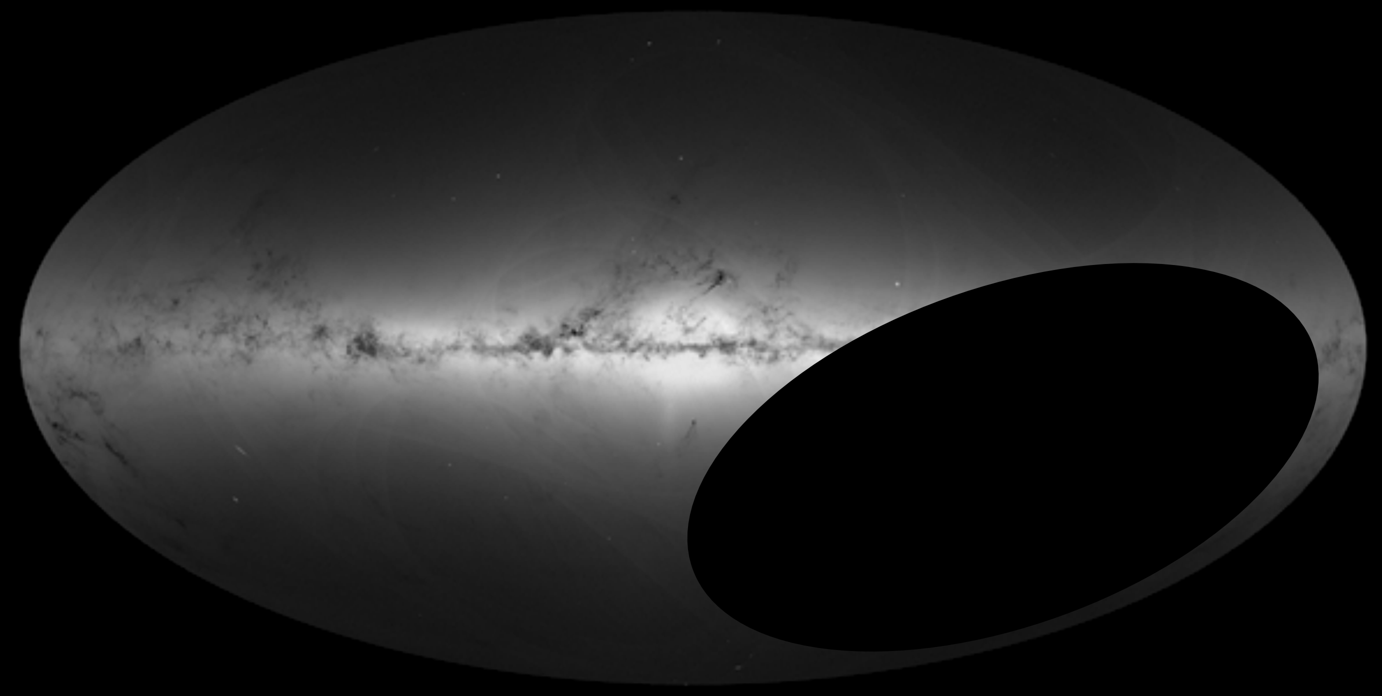
x1000 per night
x2000 nights

ZTF images
10 PB (raw!)

Survey Stability

Matrix 10^7 stars x 10^5 obs

TB of shared RAM ~1 week a year



LSST is ZTFx10

Star catalog

Image Calibration

Merging ~1000x5G images (dask)

10^4 jobs of 10GB (~h)

We will eventually process all 10PB of raw data

dl from IPAC | no storing (eventually)

Images

Light Curve extraction

40 000 SNeIa x 50 starts x 1000 exposures

CPU so far (jax?) | 10^4 30GB jobs

Starts from calibrated images

Signal Extraction x
observing condition deconvolution

Cosmology

Dark energy
Gravity

Multi-messenger

Kilonova (GW)
Cosmic Neutrino | GRB

Astrophysics

Transient Sciences

How to do accurate **simulations** to test my code ?

Is my analysis **scaling** to the full dataset ?

Is that possible to marginalise over **10_000s of parameters** ?

Is there a faster / more accurate **solution** for my problem ?

How to do accurate **simulations** to test my code ?

Is my analysis **scaling** to the full dataset ?

python

Is that possible to marginalise over **10_000s of parameters** ?

Is there a faster / more accurate **solution** for my problem ?

What is possible now that was not 10 years ago
and which might be standard 10 years from now

Pioneered it at CC-IN2P3 for ZTF

(image & spectra processing, Catalog management etc.)

Quickly followed by LSST

Now natively **accessible** through notebook.ccin2p3.fr

Slow function

```
from time import sleep
def slow_computation(x):
    sleep(1)
    return x*2
```

```
%%time
y = slow_computation(4)
```

```
CPU times: user 58.1 ms, sys: 20.2 ms, total: 78.4
ms
Wall time: 1.01 s
```

```
import numpy as np
xx = np.arange(0, 10)
```

```
%%time
yy = [slow_computation(x_) for x_ in xx]
```

```
CPU times: user 705 ms, sys: 245 ms, total: 950 ms
Wall time: 10 s
```

Massively parallelizable

On your laptop

```
from dask.distributed import Client
client = Client()
```

```
%%time
delayed_yy = [dask.delayed(slow_computation)(x_)
              for x_ in xx]
```

```
CPU times: user 603 µs, sys: 178 µs, total: 781 µs
Wall time: 701 µs
```

```
%%time
future_yy = client.compute(delayed_yy)
yy = client.gather(future_yy)
```

```
CPU times: user 81.7 ms, sys: 26.1 ms, total: 108 ms
Wall time: 1.02 s
```

dask

Pioneered it at CC-IN2P3 for ZTF

(image & spectra processing, Catalog management etc.)

Quickly followed by LSST

Now natively **accessible** through notebook.ccin2p3.fr

Slow function

```
from time import sleep
def slow_computation(x):
    sleep(1)
    return x*2
```

```
%%time
y = slow_computation(4)
```

```
CPU times: user 58.1 ms, sys: 20.2 ms, total: 78.4 ms
Wall time: 1.01 s
```

```
import numpy as np
xx = np.arange(0, 10)
```

```
%%time
yy = [slow_computation(x_) for x_ in xx]
```

```
CPU times: user 705 ms, sys: 245 ms, total: 950 ms
Wall time: 10 s
```

Massively parallelizable

At the CC-IN2P3

```
from dask4in2p3.dask4in2p3 import Dask4in2p3
dask4in2p3 = Dask4in2p3()

ncpu= 1_000
client = dask4in2p3.new_client(dask_worker_jobs=ncpu)
```

```
%%time
delayed_yy = [dask.delayed(slow_computation)(x_)
              for x_ in xx]
```

```
CPU times: user 603 µs, sys: 178 µs, total: 781 µs
Wall time: 701 µs
```

```
%%time
future_yy = client.compute(delayed_yy)
yy = client.gather(future_yy)
```

```
CPU times: user 81.7 ms, sys: 26.1 ms, total: 108 ms
Wall time: 1.02 s
```

optax

flax

blackjax

numpyro

jax_cosmo

...

Same code runs on GPU & on CPU

```python
import numpy as np
```

```python
%%time
x = np.ones((10_000, 10_000))
x2 = x*x
```

```
CPU times: user 71.5 ms, sys: 183 ms, total: 255 ms
Wall time: 255 ms
```

```python
import jax.numpy as jnp
```

```python
%%time
y = jnp.ones((10_000, 10_000))
y2 = y*y
```

```
CPU times: user 2.02 ms, sys: 897 µs, total: 2.91 ms
Wall time: 2.87 ms
```

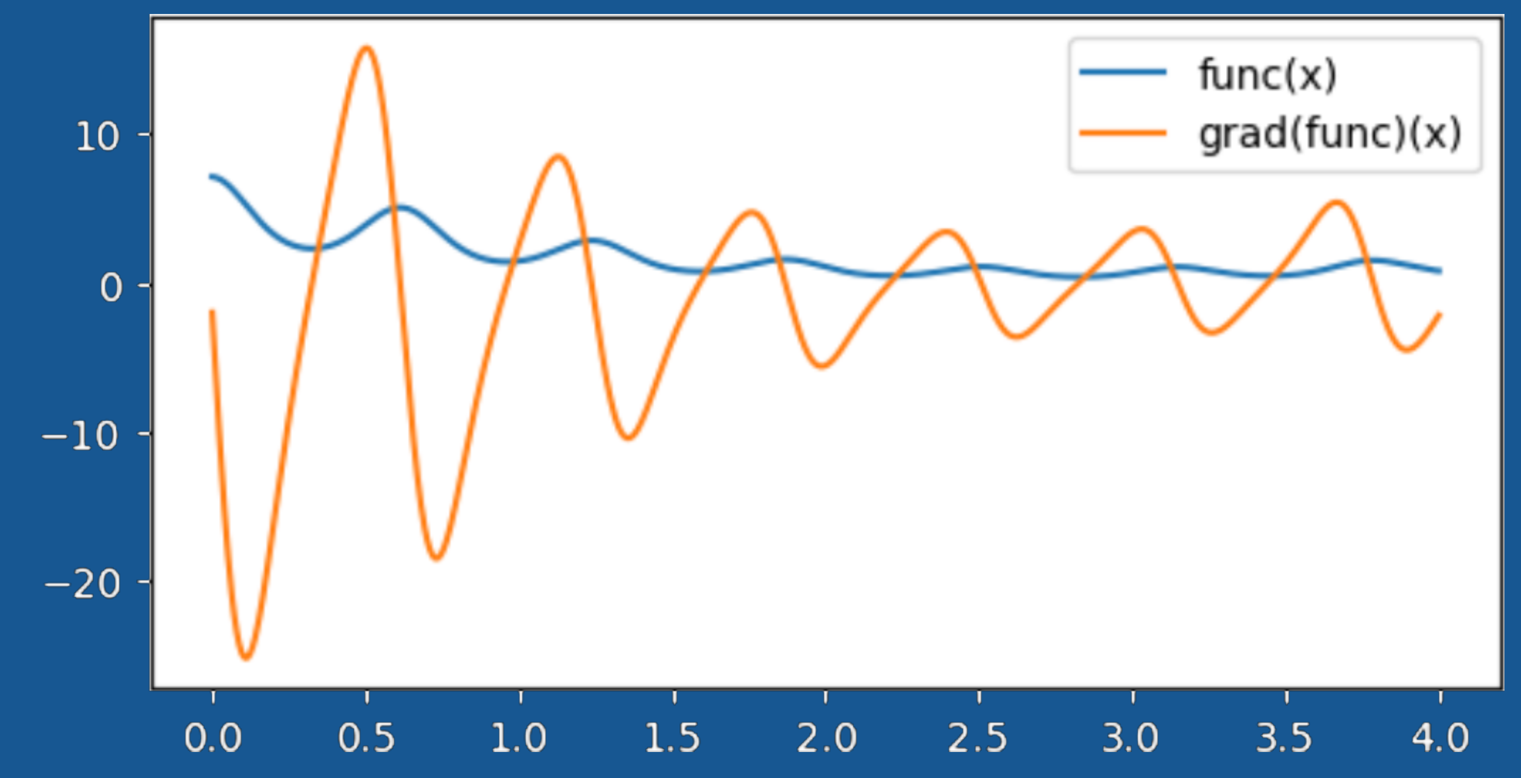
On my laptop (M1)

Automatic analytical gradient

Whatever function

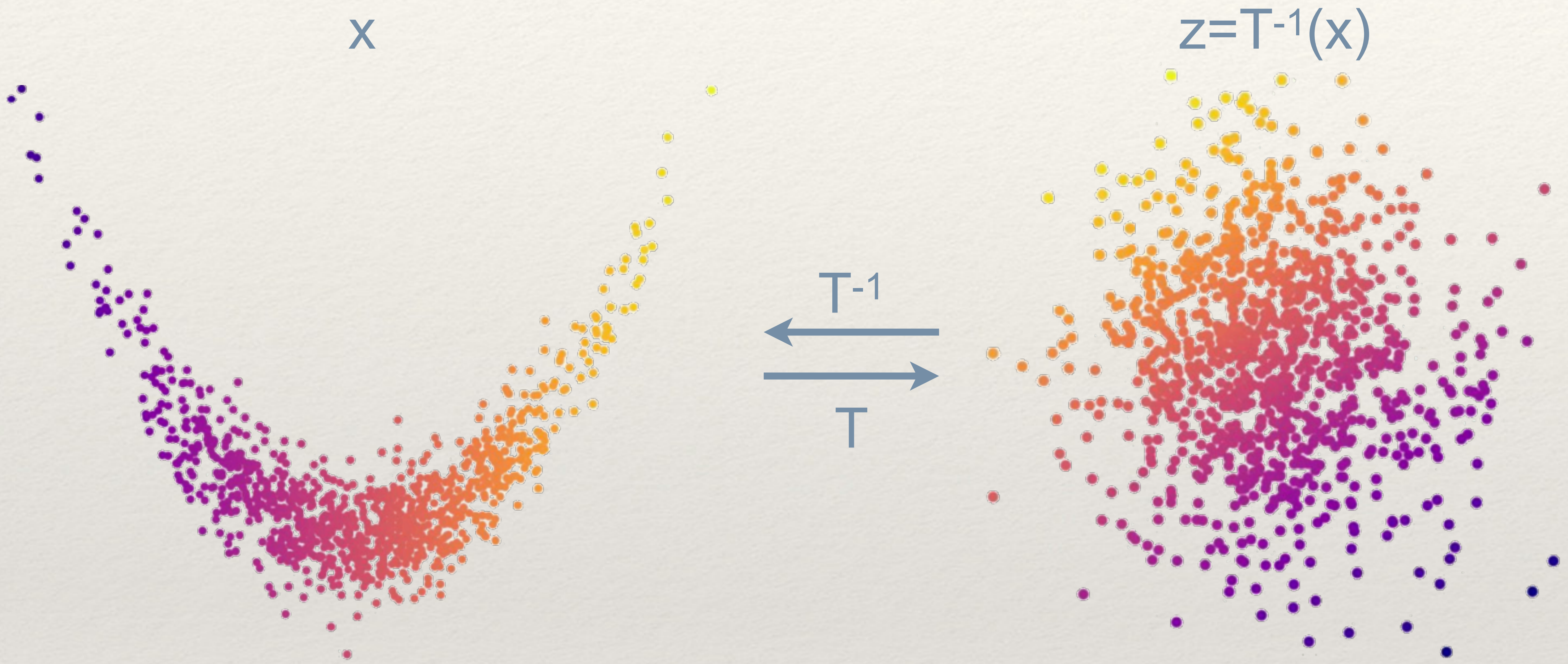
```python
def test_func(x):
    """ """
    base = jnp.cos(x*5)**2
    ref = jnp.sin(x+5)
    return jnp.exp(base-ref)

grad_func = jax.vmap( jax.grad(test_func) )
```

It's gradient



Looks like and feels like numpy & scipy

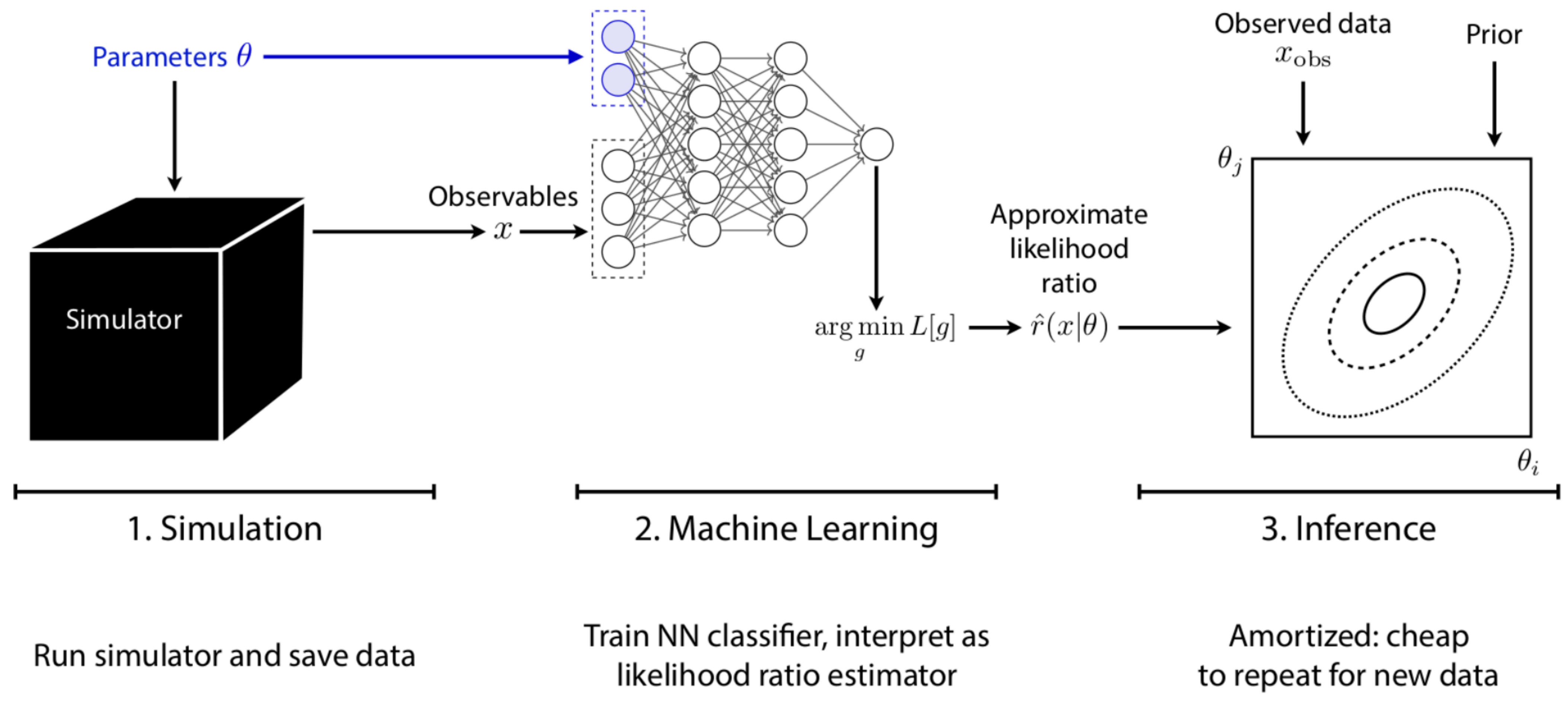
Normalising flow



From C. Doux | SOS 2024

Draw from arbitrary complex distributions

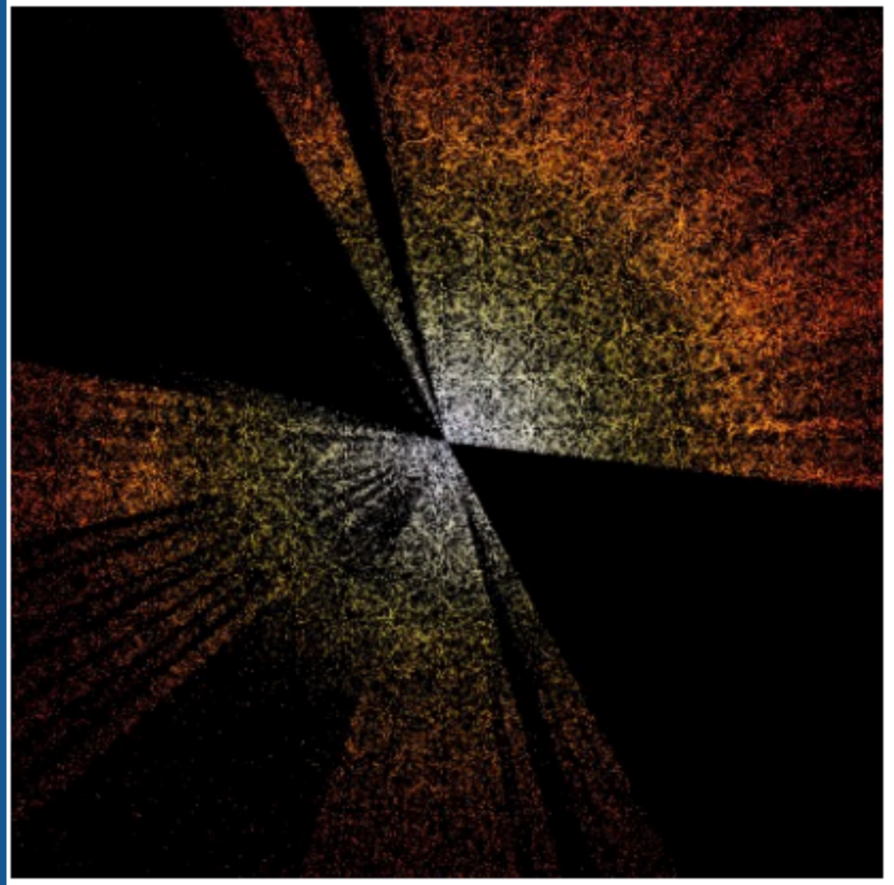
Simulation based inference



$$p(\theta|\mathbf{x}) \approx \hat{r}(\mathbf{x}|\theta)p(\theta)$$

Build your likelihood with a simulator and a neural network

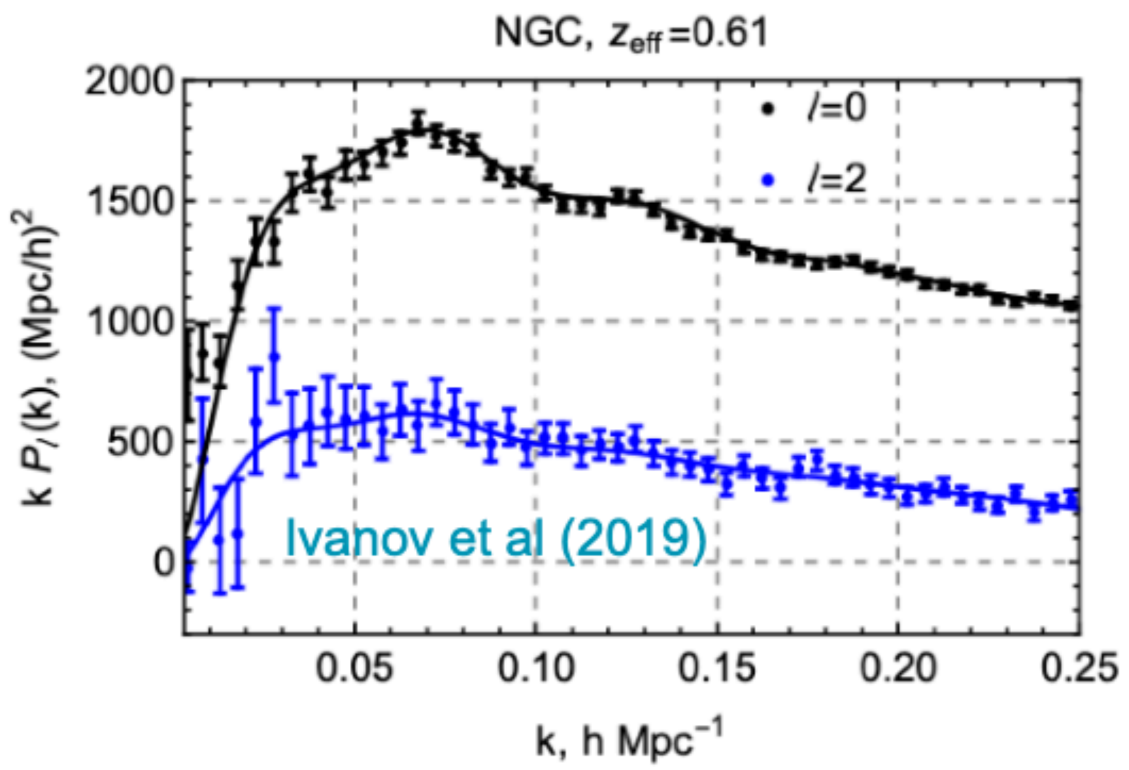
Observed Galaxy Distribution



Data Compression

Summary Statistics

- 2-point correlation function



Theoretical Model

- perturbation theory

Simulation-Based Emulators

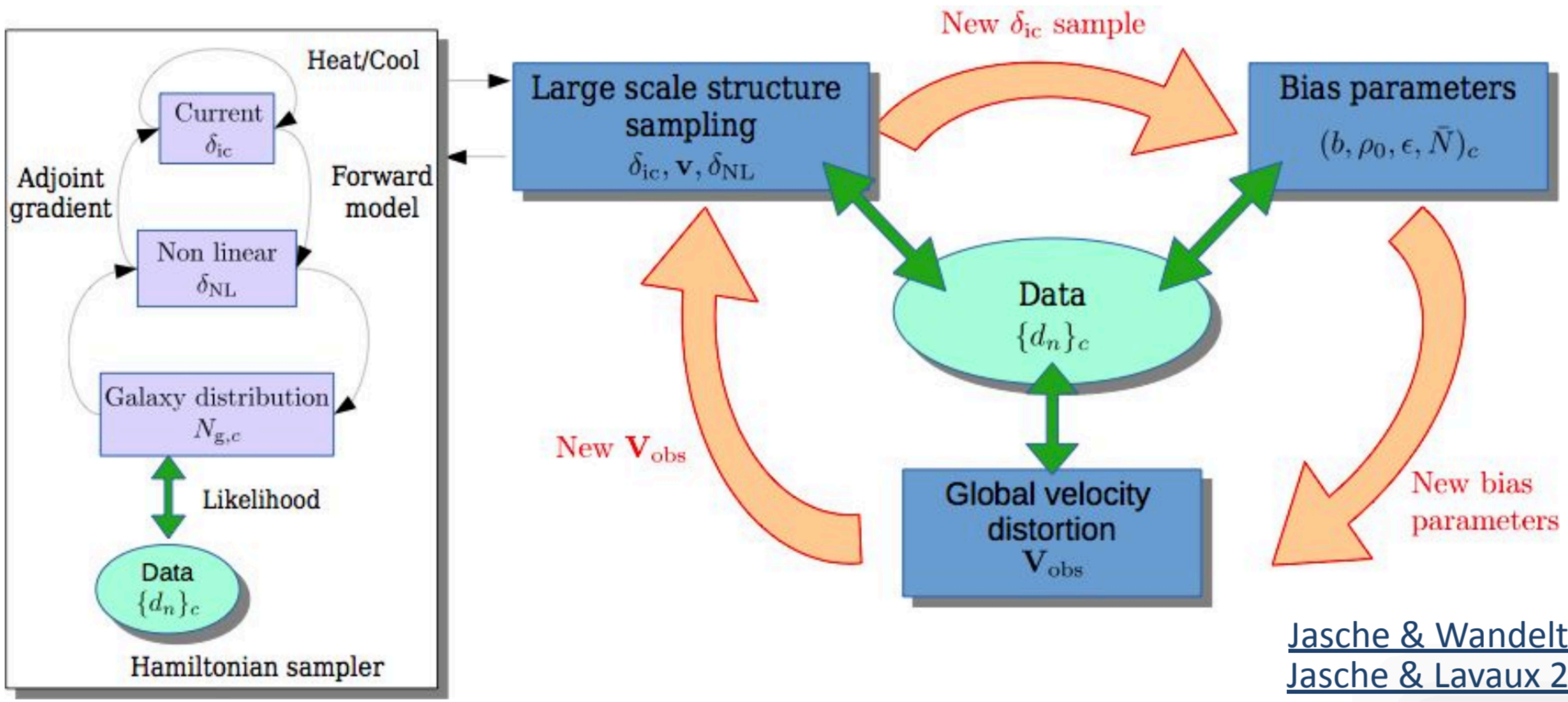
Likelihood

Inferred Cosmology

A_s
n_s
h
Ω_c
Ω_b
N_{eff}
Σm_ν
f_{NL}
$\cdots$

Complex forward modelling

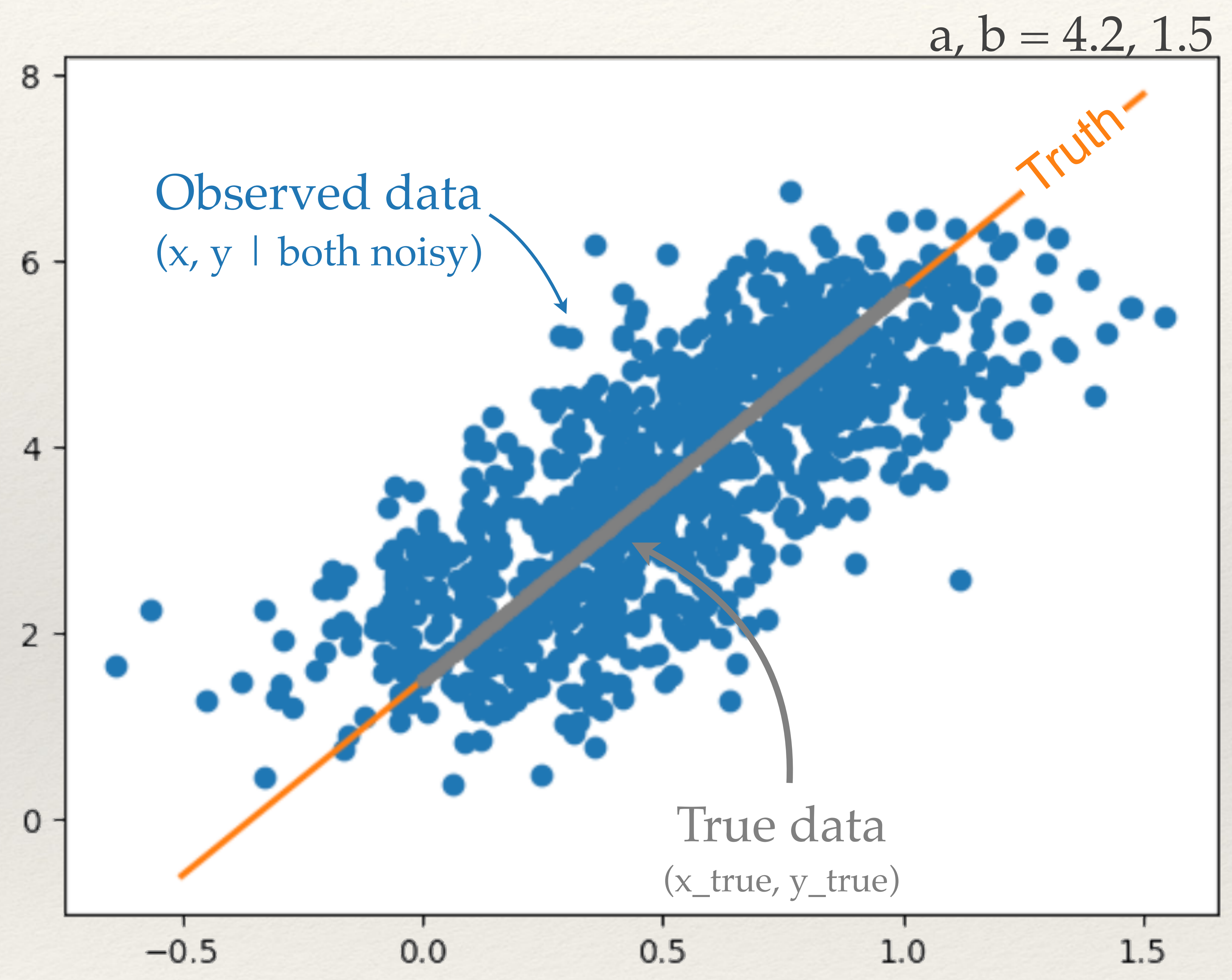
- **Fully differentiable physics forward model**



Jasche & Wandelt 2014
Jasche & Lavaux 2019

From Guilhem Lavaux | GDR CoPhy 2024

Fit a line… requires to fit for all true parameters !




```python
# The model
def get_model(param):
    x_model = param.get("x", x)
    a, b = param["coefs"]
    y_model = x_model * a + b
    return x_model, y_model

# The "total chi2"
def get_chi2(param):
    x_model, y_model = get_model(param)
    chi2_x = jnp.sum((x_model - x)**2 / dx**2)
    chi2_y = jnp.sum((y_model - y)**2 / dy**2)

    return chi2_x + chi2_y

# ============ #
#   fit        #
# ============ #
# guess
params = {"x": x,
        "coefs": jnp.asarray([3., 0.], dtype="float32")
        }

# 1. Setup the opitmizer (optax)
import optax
optimizer = optax.adam(0.001)
# Obtain the `opt_state` that contains statistics for the optimizer.
opt_state = optimizer.init(params)

# 2. the derivative function
grad_func = jax.jit(jax.grad( get_chi2 )) # get the derivative

# 3. the gradient descent
losses = []
niter = 5_000
for i in range(niter):
    current_grads = grad_func(params) # current gradient
    updates, opt_state = optimizer.update(current_grads, opt_state) # update
    params = optax.apply_updates(params, updates) # new params
    losses.append( get_chi2(params) ) # store the loss function

# 4. the result
print(params["coefs"])
```

```
[4.4068894 1.3669752]
```

Organise a task-force

What is possible now that was not 10 years ago
and which might be standard 10 years from now

Python (rust?) oriented | for physicists
packages, methods and statistics

Survey current progress

Provide guidance to others

Join forces for shared projects

Should we ?