# how to **do integrals** the **brute force way** with **dan foreman-mackey**

how to

**do integrals**

using

**differentiation**

with

**dan foreman-mackey**

[who am I?]

**Dan Foreman-Mackey**

[where am I?]

**SF / FI / CCA**

[what do I do?]

**exoplanets?**

[wth am I doing here?]

•••

[what do I do?]

write **software** for astronomy

[what do I do?]

focus on **implementation**

[what do I do?]

try to **learn things** & **share them**

[today's takeaways: #1]

**data analysis** in astrophysics

is getting **ambitious**

[today's takeaways: #1]

**data analysis** in grav waves

is ~~getting~~ **ambitious**

[today's takeaways: #2]

even if you're **not** doing machine learning, there are **useful open source tools**

[today's takeaways: #3]

using them might be **easy**

or **a complete pain**

[today's takeaways: #4]

it might be **worth it**

[1]
# inference

want:

**data ⇒ physics**

have:

**physics ⇒ data**

integral of the form

f(**physics**) p(**physics**|**data**) d**physics**

[one option]

(markov chain) monte carlo

**physics** ~ p(**physics**|**data**)

so, "all" you need is:

[a] a "good" **sampler**

[b] fast p(**data**|**physics**)

[a] a "good" **sampler**

cost per effective sample

[a] a "good" **sampler**
depends on **# of parameters**
and **"geometry"** of the problem

[b] fast p(**data**|**physics**)
should be **interpretable**

[b] fast p(**data**|**physics**)
depends on **size of data**
and **simplifying assumptions**

in **astrophysics**, we want to

[ ] do **rigorous inference**

[ ] with **huge datasets**, and

[ ] **physics-based models**

in **astrophysics**, we want to

[ ] do **rigorous inference**

[ ] with **huge datasets**, and

[ ] **physics-based models**

[ ] finish in **finite time**

so. what do we **do**?

# emcee

**emcee** is an MIT licensed pure-Python implementation of Goodman & Weare's Affine Invariant Markov chain Monte Carlo (MCMC) Ensemble sampler and these pages will show you how to use it.

This documentation won't teach you too much about MCMC but there are a lot of resources available for that (try this one). We also published a paper explaining the emcee algorithm and implementation in detail.

emcee has been used in quite a few projects in the astrophysical literature and it is being actively developed on GitHub.

GitHub dfm/emcee | Tests passing | license MIT | arXiv 1202.3665 | coverage 96%

## Basic Usage

If you wanted to draw samples from a 5 dimensional Gaussian, you would do something like:

```
import numpy as np
import emcee

def log_prob(x, ivar):
    return -0.5 * np.sum(ivar * x ** 2)

ndim, nwalkers = 5, 100
ivar = 1. / np.random.rand(ndim)
p0 = np.random.randn(nwalkers, ndim)

sampler = emcee.EnsembleSampler(nwalkers, ndim, log_prob, args=[ivar])
sampler.run_mcmc(p0, 10000)
```

A more complete example is available in the Quickstart tutorial.

dynesty — dynesty 1.1 documen

https://dynesty.readthedocs.io/en/latest/index.html

dynesty
latest

Search docs

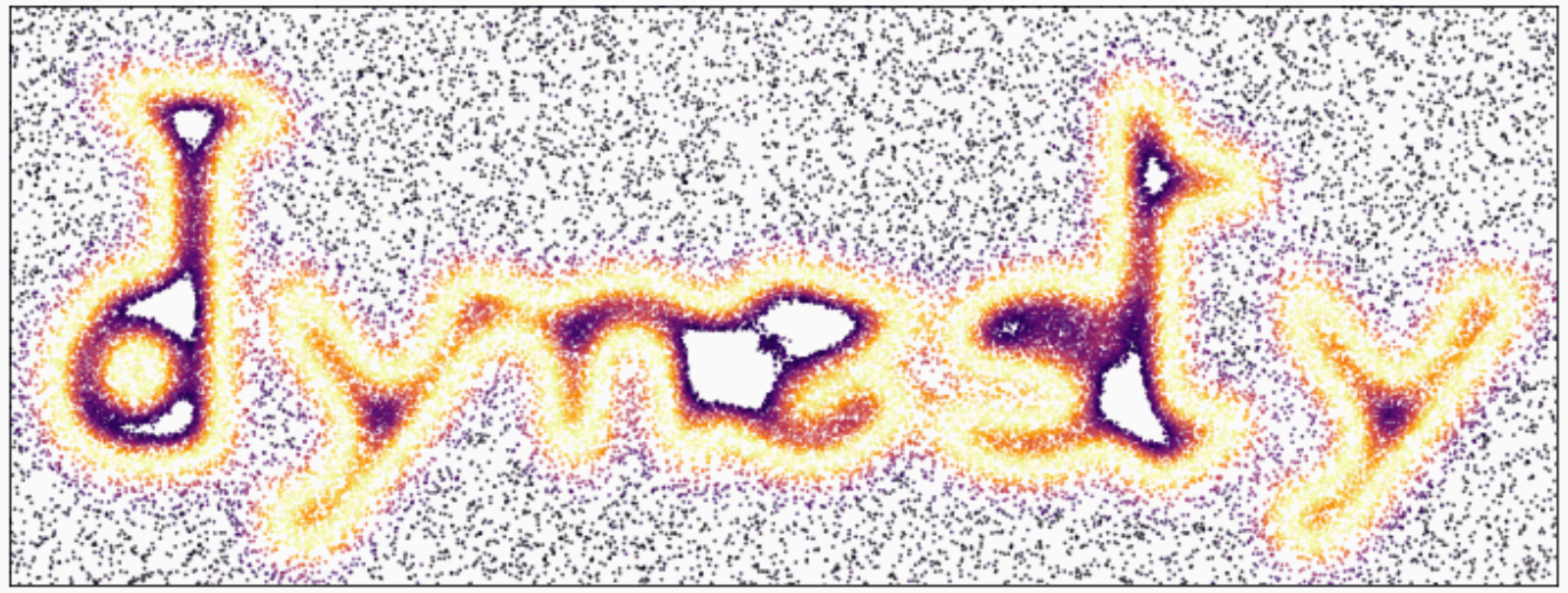
- Crash Course
- Background
- Getting Started
- Dynamic Nested Sampling with dynesty
- Nested Sampling Errors
- Examples
- FAQ
- References and Acknowledgements
- API

Read the Docs v: latest

Docs » dynesty

Edit on GitHub

# dynesty



`dynesty` is a Pure Python, MIT-licensed Dynamic Nested Sampling package for estimating Bayesian posteriors and evidences. See Crash Course and Getting Started for more information. The latest development version can be found here.

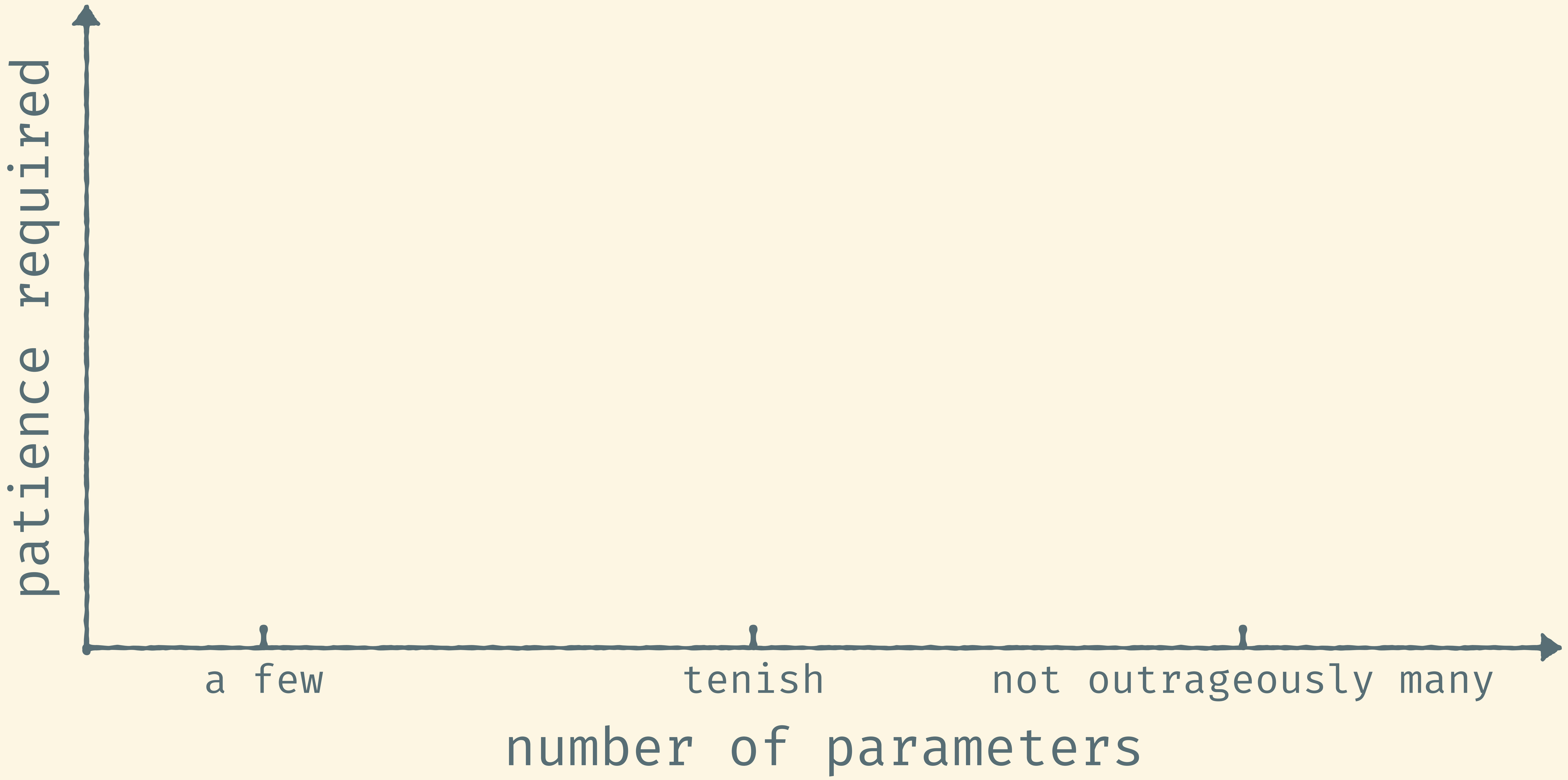
**The release paper describing the code can be found here.**

As a multi-purpose sampler, `dynesty` is designed to perform “reasonably well” across a large array of problems but is not optimized for any single one. In particular, please take caution when applying `dynesty` to estimate Bayesian posteriors and evidences for large-dimensional (>30 dimensions or so) problems.

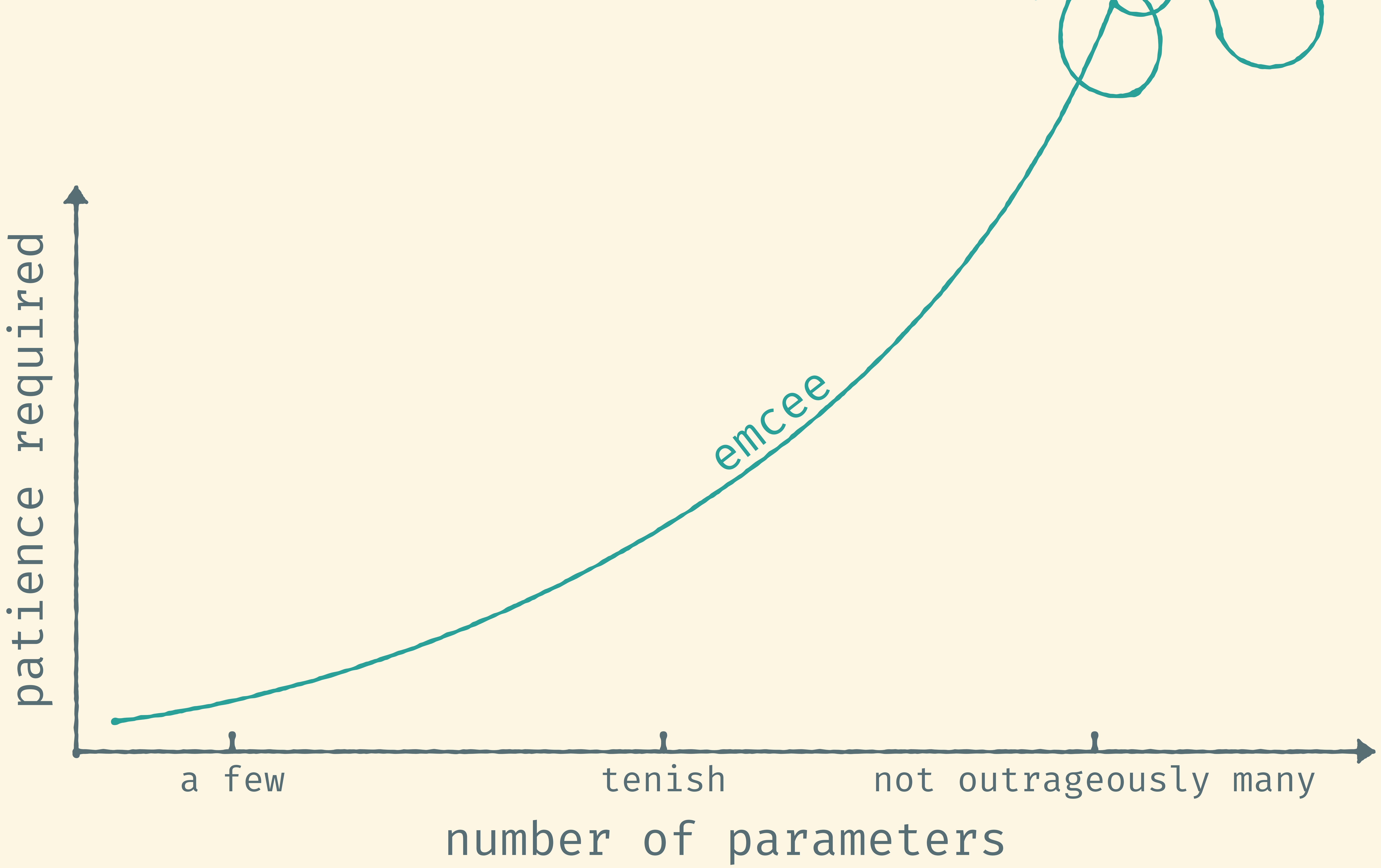
## Installation

`dynesty` is compatible with both Python 2.7 and Python 3.6. It requires `numpy` (for arithmetic)

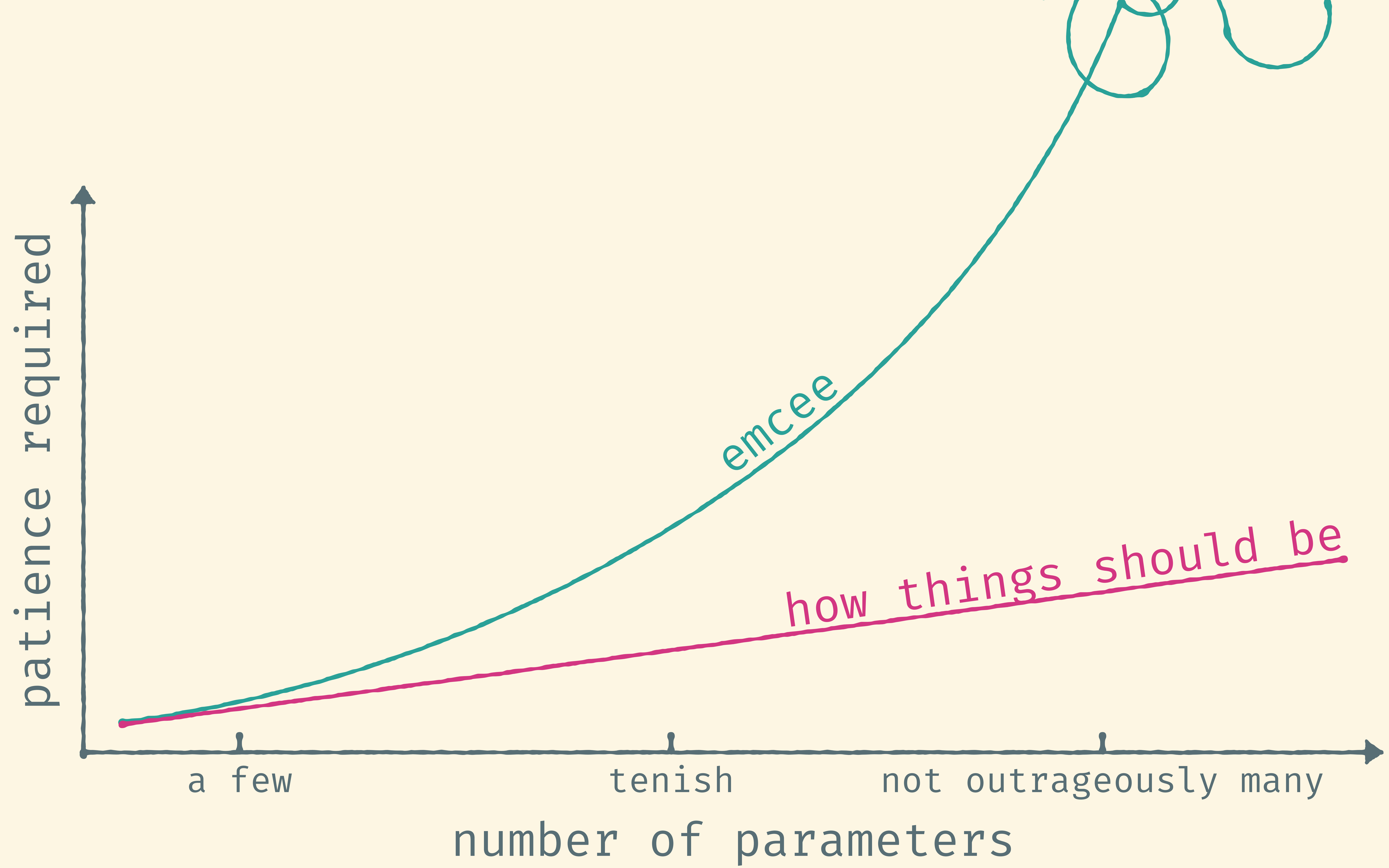
**emcee** isn't a very "good" **sampler**




ref: personal experience




ref: personal experience




ref: personal experience

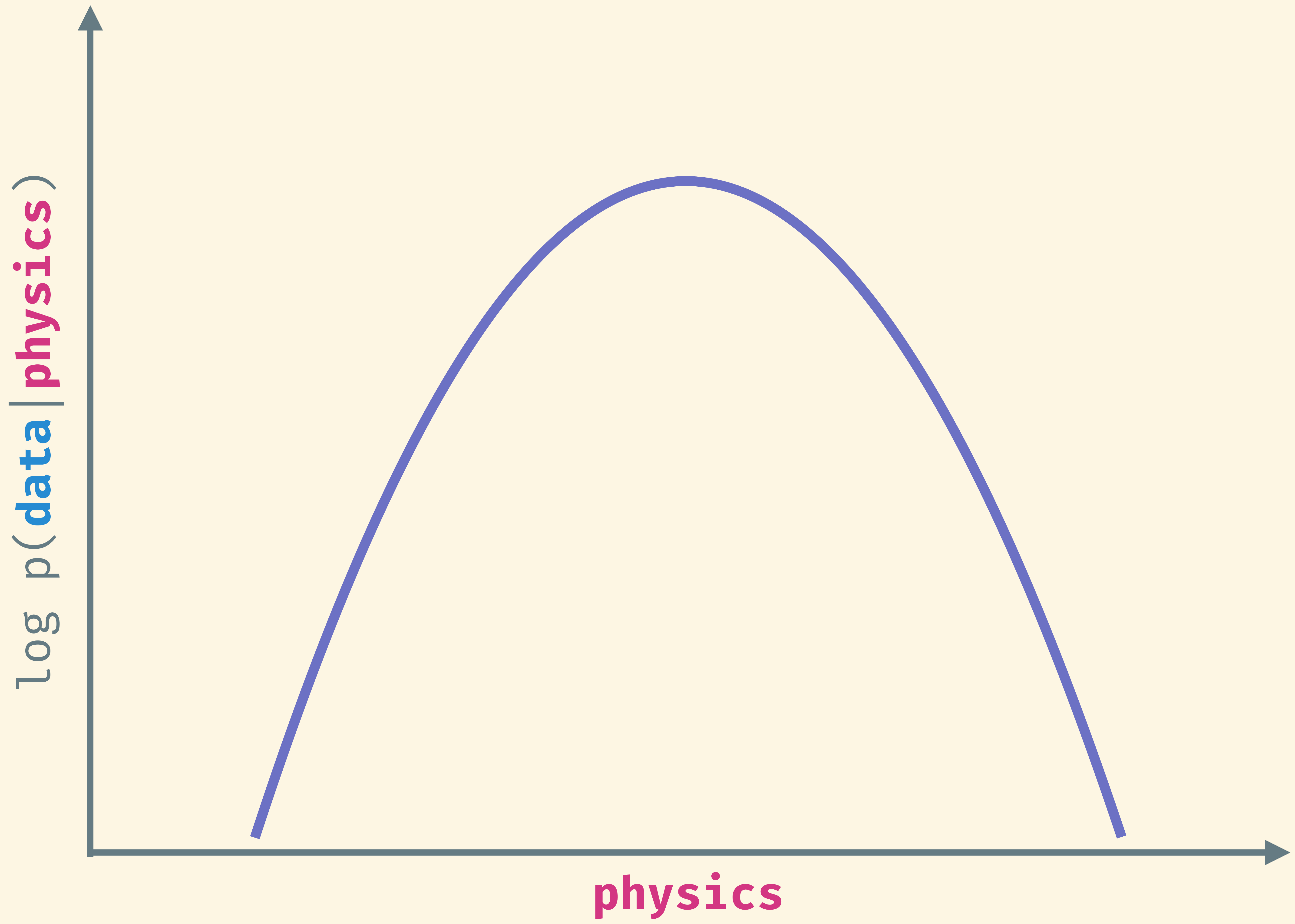
[X] rigorous inference
[X] huge datasets
[X] physics-based models
[?] finite time

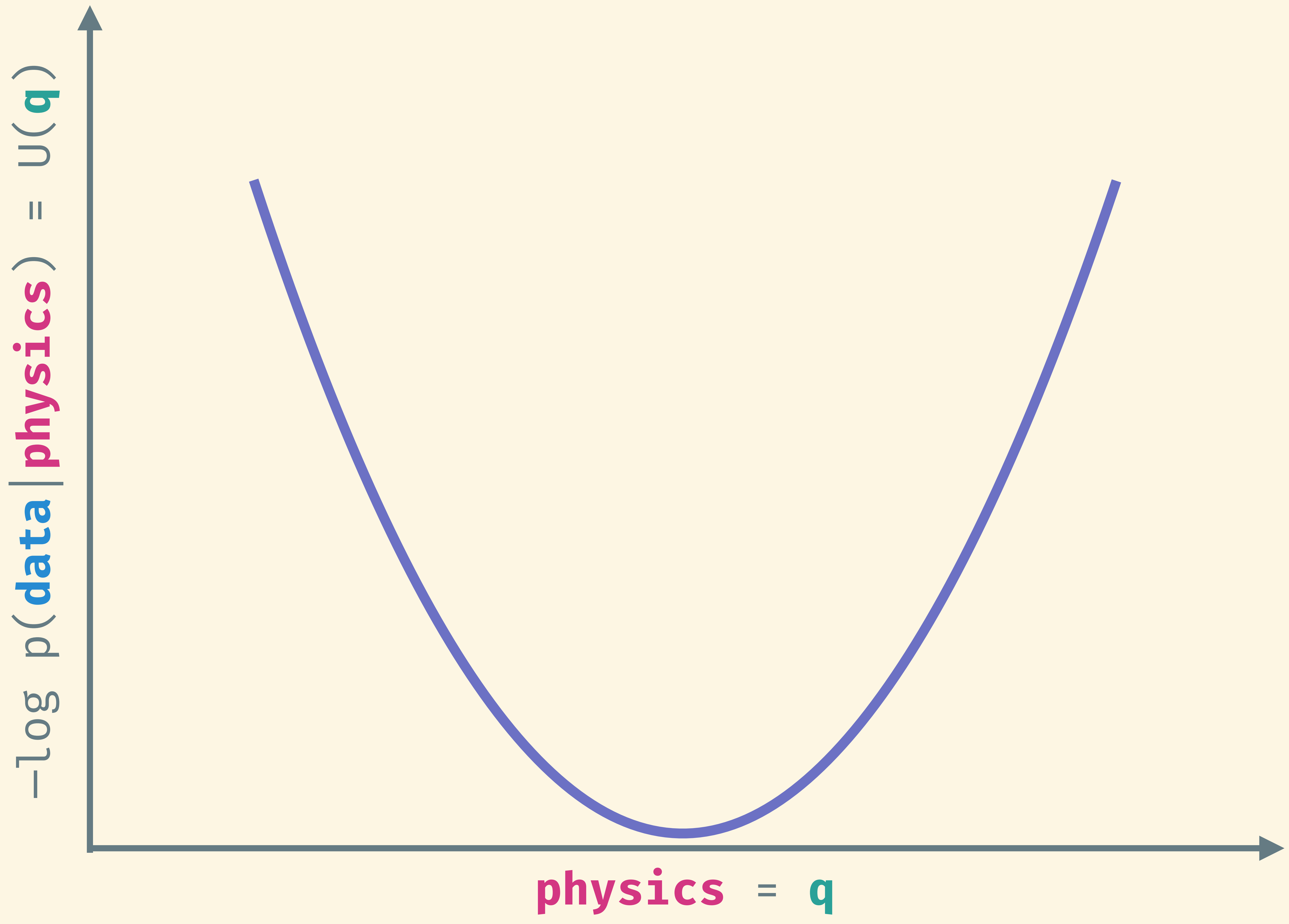
this is **a solved problem***

*sort of

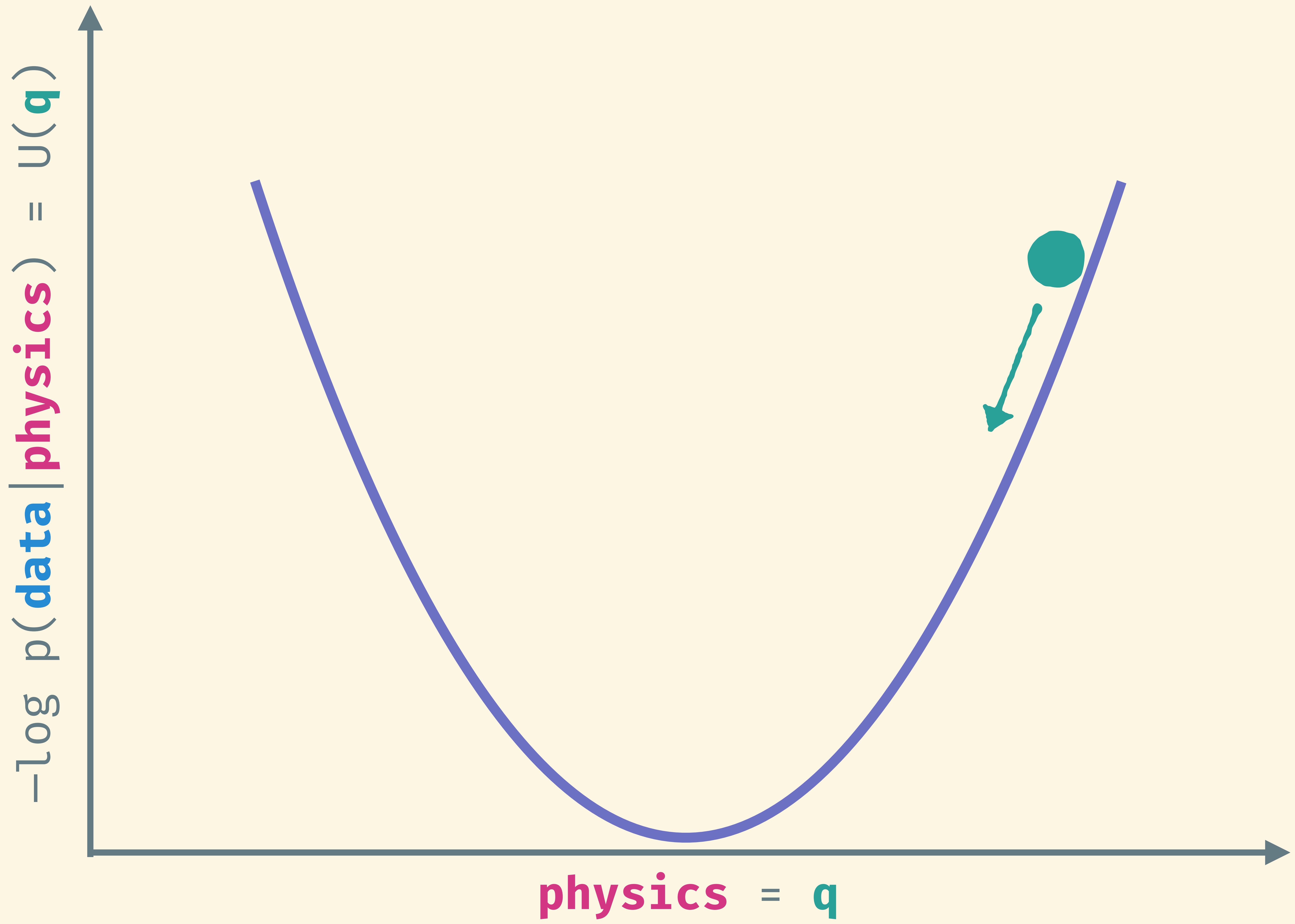
[2]
# gradients

dp(data|physics) / dphysics


log p(data|physics)
physics

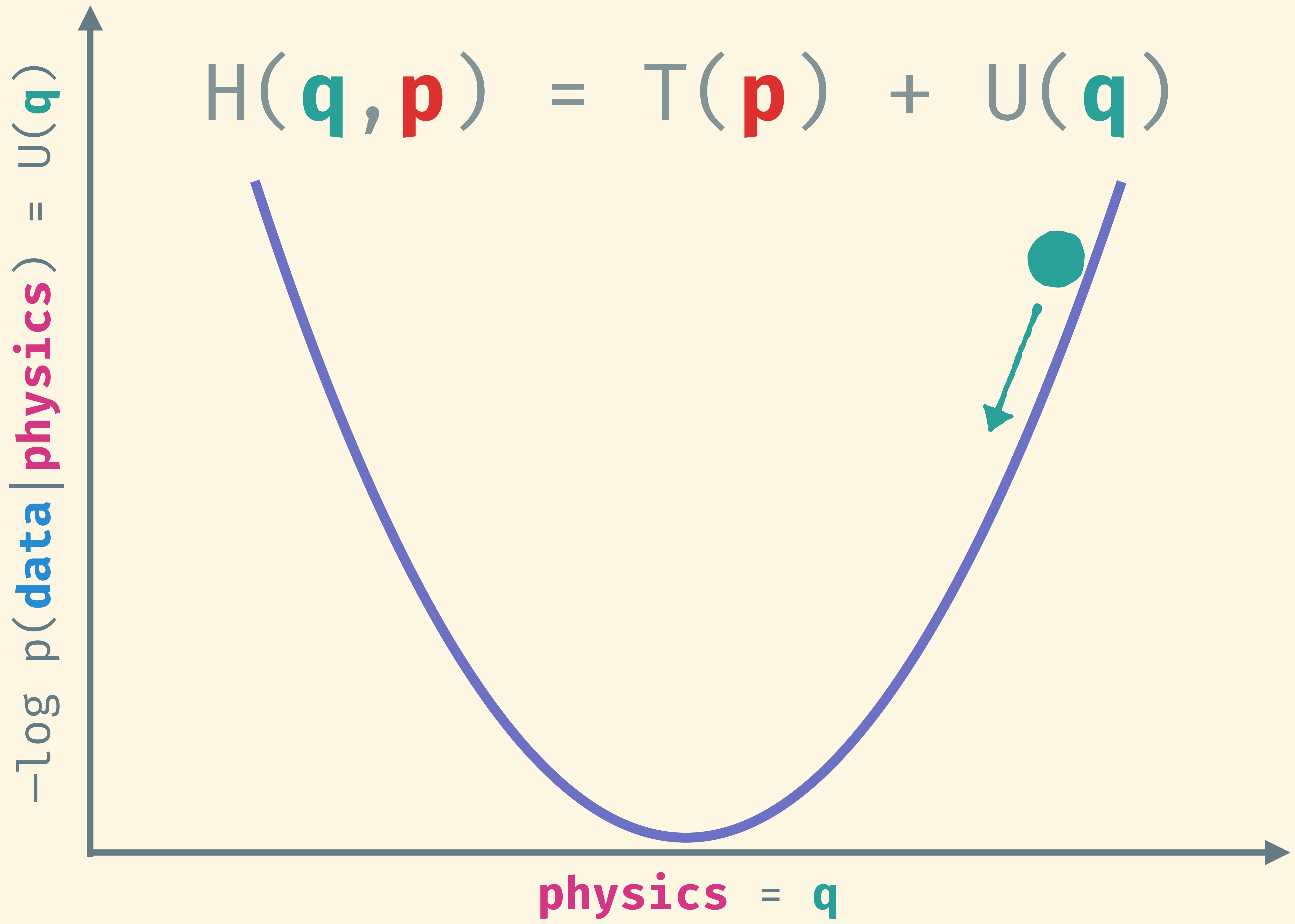


−log p(data|physics) = U(q)
physics = q



−log p(data|physics) = U(q)
physics = q

$$H(\mathbf{q},\mathbf{p}) = T(\mathbf{p}) + U(\mathbf{q})$$



cool. so we're **done**?

maybe!

to name a few...

let's not get ahead of ourselves

but, probably not.

[X] **rigorous inference**
[X] **huge datasets**
[?] **physics-based models**
[X] **finite time**

$$dp(\mathbf{data} \mid \mathbf{physics}) \;/\; d\mathbf{physics}$$

**auto**matic **diff**erentiation

your **model** is just **code**

apply the **chain rule**

apply the **chain rule**
over and over again...

sounds **silly**?

it's **not**! (mostly)

what about things like:

$$M = E - e \sin(E)$$

ref: Kepler (1609)

custom "**ops**"

[a] C/C++/Fortran/CUDA/… code

[b] derivative rules

[3]

# my (current) recommendations

[caveat]

I mostly work in **Python**

[caveat]

this is **a moving target**

[caveat]

this is **not** a **comprehensive** list the classics

**PyMC3**
easy to extend;
"Pythonic"

**Stan**
state-of-the-art
inference algorithms

GPU support, variational inference

**PyTorch + Pyro**
easy to extend

**TensorFlow + TF Probability**
fast; powerful but
poorly documented
inference algorithms

the new kid on the block

**JAX** **+ Numpyro/TF Probability**
all of the above?

my current **recommendation**

(numpyro)

+

```
import numpy as np
```

$\Rightarrow$

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

[4]

# a case study

exoplanet

Search the docs ...



# exoplanet

*exoplanet* is a toolkit for probabilistic modeling of time series data in astronomy with a focus on observations of exoplanets, using PyMC3. *PyMC3* is a flexible and high-performance model building language and inference engine that scales well to problems with a large number of parameters. *exoplanet* extends *PyMC3*'s language to support many of the custom functions and distributions required when fitting exoplanet datasets. These features include:

- A fast and robust solver for Kepler's equation.
- Scalable Gaussian Processes using celerite.
- Fast and accurate limb darkened light curves using starry.
- Common reparameterizations for exoplanet-specific parameters like limb darkening and eccentricity.
- And many others!

All of these functions and distributions include methods for efficiently calculating their *gradients* so that they can be used with gradient-based inference methods like Hamiltonian Monte Carlo, No U-Turns Sampling, and variational inference. These methods tend to be more robust than the methods more commonly used in astronomy (like ensemble samplers and nested sampling) especially when the model has more than a few parameters. For many exoplanet applications, *exoplanet* (the code) can improve the typical performance by orders of magnitude.

*exoplanet* is being actively developed in a public repository on GitHub so if you have any trouble, open an issue there.

**Where to find what you need**

🖥️ For general installation and basic usage, continue scrolling to the table of contents below.

🖼️ For more in depth examples of *exoplanet* used for more realistic problems, go to the Case studies page.



v: latest

celerite2 — celerite2

https://celerite2.readthedocs.io/en/latest/

celerite2

Search

celerite2
40 Stars · 4 Forks

**celerite** \se.le.ʁi.te\ *noun, archaic literary*
A scalable method for Gaussian Process regression in one dimension. From French *célérité*.

celerite2



# celerite2 ¶

*celerite* is an algorithm for fast and scalable Gaussian Process (GP) Regression in one dimension and this library, *celerite2* is a re-write of the original celerite project to improve numerical stability and integration with various machine learning frameworks. This implementation includes interfaces in Python and C++, with full support for Theano/PyMC3 and JAX.

This documentation won't teach you the fundamentals of GP modeling but the best resource for learning about this is available for free online: Rasmussen & Williams (2006). Similarly, the *celerite* algorithm is restricted to a specific class of covariance functions (see the original paper for more information and a recent generalization for extensions to structured two-dimensional data). If you need scalable GPs with more general covariance functions, GPyTorch might be a good choice.

*celerite2* is being actively developed in a public repository on GitHub so if you have any trouble, open an issue there.

User Guide

- Installation
  - Using pip

**Contents**



v: latest

[gaussian process]

a **likelihood function** for

**correlated noise**

[gaussian process]

in my case, caused by

**stochastic stellar variability**

[gaussian process]

in your case, caused by

**systematics** and/or **GW background**

[gaussian process]

the problem:

**$O(N^3)$ scaling**

[gaussian process]

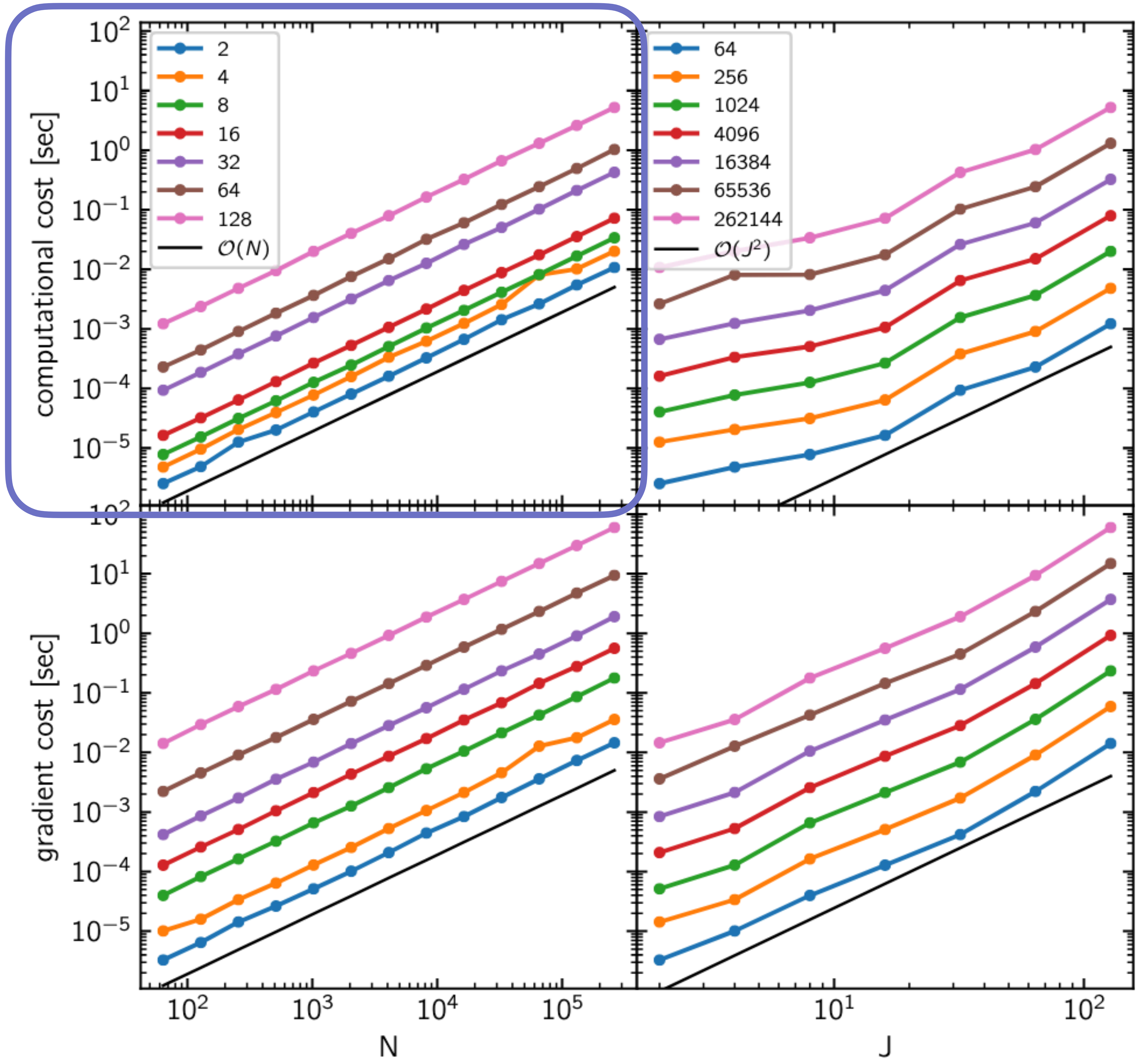
in special cases:

**O(NlogN) scaling**

[gaussian process]

with celerite:

**O(N) scaling**


computational cost [sec]
gradient cost [sec]
2
4
8
16
32
64
128
$\mathcal{O}(N)$
64
256
1024
4096
16384
65536
262144
$\mathcal{O}(J^2)$
$10^2$
$10^1$
$10^0$
$10^{-1}$
$10^{-2}$
$10^{-3}$
$10^{-4}$
$10^{-5}$
$10^3$
$10^4$
$10^5$
N
J

celerite2/forward.hpp at main · 

https://github.com/exoplanet-dev/celerite2/blob/main/c%2B%2B/include/celerite2/forward.hpp

```cpp
/**
 * \brief Compute the Cholesky factorization of the system
 *
 * This computes `d` and `W` such that:
 *
 * `K = L*diag(d)*L^T`
 *
 * where `K` is the celerite matrix and
 *
 * `L = 1 + tril(U*W^T)`
 *
 * This can be safely applied in place: `d_out` can point to `a` and `W_out` can
 * point to `V`, and the memory will be reused. In this particular case, the
 * `celerite2::core::factor_rev` function doesn't use `a` and `V`, but this
 * won't be true for all `_rev` functions.
 *
 * @param t     (N,): The input coordinates (must be sorted)
 * @param c     (J,): The transport coefficients
 * @param a     (N,): The diagonal component
 * @param U     (N, J): The first low rank matrix
 * @param V     (N, J): The second low rank matrix
 * @param d_out (N,): The diagonal component of the Cholesky factor
 * @param W_out (N, J): The second low rank component of the Cholesky factor
 * @param S_out (N, J*J): The cached value of the S matrix at each step
 */
template <bool update_workspace = true, typename Input, typename Coeffs, typename Diag, typename LowRank, typename DiagOut, typename LowRankOut,
          typename Work>
Eigen::Index factor(const Eigen::MatrixBase<Input> &t,          // (N,)
                    const Eigen::MatrixBase<Coeffs> &c,         // (J,)
                    const Eigen::MatrixBase<Diag> &a,           // (N,)
                    const Eigen::MatrixBase<LowRank> &U,        // (N, J)
                    const Eigen::MatrixBase<LowRank> &V,        // (N, J)
                    Eigen::MatrixBase<DiagOut> const &d_out,    // (N,)
                    Eigen::MatrixBase<LowRankOut> const &W_out, // (N, J)
                    Eigen::MatrixBase<Work> const &S_out        // (N, J*J)
) {
  ASSERT_ROW_MAJOR(Work);

  typedef typename Diag::Scalar Scalar;
  typedef typename Eigen::internal::plain_row_type<LowRank>::type RowVector;
  typedef typename Eigen::internal::plain_col_type<Coeffs>::type CoeffVector;

  Eigen::Index N = U.rows(), J = U.cols();
  CAST_VEC(DiagOut, d, N);
  CAST_MAT(LowRankOut, W, N, J);
  CAST_BASE(Work, S);
  if (update_workspace) {
    S.derived().resize(N, J * J);
    S.row(0).setZero();
  }
```

# Posterior inference using emcee

Now, to get a sense for the uncertainties on our model, let's use Markov chain Monte Carlo (MCMC) to numerically estimate the posterior expectations of the model. In this first example, we'll use the emcee package to run our MCMC. Our likelihood function is the same as the one we used in the previous section, but we'll also choose a wide normal prior on each of our parameters.

[9]:

```python
import emcee

prior_sigma = 2.0


def log_prob(params, gp):
    gp = set_params(params, gp)
    return (
        gp.log_likelihood(y) - 0.5 * np.sum((params / prior_sigma) ** 2),
        gp.kernel.get_psd(omega),
    )


np.random.seed(5693854)
coords = soln.x + 1e-5 * np.random.randn(32, len(soln.x))
sampler = emcee.EnsembleSampler(
    coords.shape[0], coords.shape[1], log_prob, args=(gp,)
)
state = sampler.run_mcmc(coords, 2000, progress=True)
sampler.reset()
state = sampler.run_mcmc(state, 5000, progress=True)
```

```
100%|██████████| 2000/2000 [00:32<00:00, 62.37it/s]
100%|██████████| 5000/5000 [01:20<00:00, 62.34it/s]
```

After running our MCMC, we can plot the predictions that the model makes for a handful of samples from the chain. This gives a qualitative sense of the uncertainty in the predictions.

Contents

```
import numpyro.distributions as dist
from numpyro.infer import MCMC, NUTS

import celerite2.jax
from celerite2.jax import terms as jax_terms


def numpyro_model(t, yerr, y=None):
    mean = numpyro.sample("mean", dist.Normal(0.0, prior_sigma))
    log_jitter = numpyro.sample("log_jitter", dist.Normal(0.0, prior_sigma))

    log_sigma1 = numpyro.sample("log_sigma1", dist.Normal(0.0, prior_sigma))
    log_rho1 = numpyro.sample("log_rho1", dist.Normal(0.0, prior_sigma))
    log_tau = numpyro.sample("log_tau", dist.Normal(0.0, prior_sigma))
    term1 = jax_terms.UnderdampedSHOTerm(
        sigma=jnp.exp(log_sigma1), rho=jnp.exp(log_rho1), tau=jnp.exp(log_tau)
    )

    log_sigma2 = numpyro.sample("log_sigma2", dist.Normal(0.0, prior_sigma))
    log_rho2 = numpyro.sample("log_rho2", dist.Normal(0.0, prior_sigma))
    term2 = jax_terms.OverdampedSHOTerm(
        sigma=jnp.exp(log_sigma2), rho=jnp.exp(log_rho2), Q=0.25
    )

    kernel = term1 + term2
    gp = celerite2.jax.GaussianProcess(kernel, mean=mean)
    gp.compute(t, diag=yerr ** 2 + jnp.exp(log_jitter), check_sorted=False)

    numpyro.sample("obs", gp.numpyro_dist(), obs=y)
    numpyro.deterministic("psd", kernel.get_psd(omega))


nuts_kernel = NUTS(numpyro_model, dense_mass=True)
mcmc = MCMC(nuts_kernel, num_warmup=1000, num_samples=1000, num_chains=2)
rng_key = random.PRNGKey(34923)
%time mcmc.run(rng_key, t, yerr, y=y)
```

```
CPU times: user 16.8 s, sys: 122 ms, total: 16.9 s
Wall time: 16.9 s
```

Contents

[recently]
generalized to
**multivariate** data

**celerite2**.readthedocs.io

celeriac/ops.py at main · dfm/ce

https://github.com/dfm/celeriac/blob/main/src/celeriac/ops.py#L93-L102

```python
    if P.ndim == 1:
        if transpose:
            return other * P[None, :]
        return P[:, None] * other
    if transpose:
        return other @ P
    return P.T @ other


Carry = Tuple[Array, Array, Array]
Data = Tuple[Array, Array, Array, Array]
MatmulData = Tuple[Array, Array, Array]


def _factor_impl(
    state: Carry, data: Data
) -> Tuple[Carry, Tuple[Array, Array]]:
    Sp, dp, Wp = state
    an, Un, Vn, Pn = data
    Sn = _pdot(Pn, _pdot(Pn, Sp + dp * jnp.outer(Wp, Wp), transpose=True))
    tmp = Sn @ Un
    dn = an - tmp @ Un
    Wn = (Vn - tmp) / dn
    return (Sn, dn, Wn), (dn, Wn)


def _solve_impl(state: Carry, data: Data) -> Tuple[Carry, Array]:
    Fp, Wp, Zp = state
    Un, Wn, Pn, Yn = data
    Fn = _pdot(Pn, Fp + jnp.outer(Wp, Zp))
    Zn = Yn - Un @ Fn
    return (Fn, Wn, Zn), Zn


def _matmul_impl(state: Carry, data: MatmulData) -> Tuple[Carry, Array]:
    (Fp, Vp, Yp) = state
    Vn, Pn, Yn = data
    Fn = _pdot(Pn, Fp + jnp.outer(Vp, Yp))
    return (Fn, Vn, Yn), Fn
```

celerite2/forward.hpp at main · e

https://github.com/exoplanet-dev/celerite2/blob/main/c%2B%2B/include/celerite2/forward.hpp#L69-L135

50%

```cpp
 * @param V     (N, J): The second low rank matrix
 * @param d_out (N,): The diagonal component of the Cholesky factor
 * @param W_out (N, J): The second low rank component of the Cholesky factor
 * @param S_out (N, J*J): The cached value of the S matrix at each step
 */
template <bool update_workspace = true, typename Input, typename Coeffs, typename Diag, typename LowRank, typename DiagOut, typename LowRankOut,
          typename Work>
Eigen::Index factor(const Eigen::MatrixBase<Input> &t,          // (N,)
                    const Eigen::MatrixBase<Coeffs> &c,         // (J,)
                    const Eigen::MatrixBase<Diag> &a,           // (N,)
                    const Eigen::MatrixBase<LowRank> &U,        // (N, J)
                    const Eigen::MatrixBase<LowRank> &V,        // (N, J)
                    Eigen::MatrixBase<DiagOut> const &d_out,    // (N,)
                    Eigen::MatrixBase<LowRankOut> const &W_out, // (N, J)
                    Eigen::MatrixBase<Work> const &S_out        // (N, J*J)
) {
  ASSERT_ROW_MAJOR(Work);

  typedef typename Diag::Scalar Scalar;
  typedef typename Eigen::internal::plain_row_type<LowRank>::type RowVector;
  typedef typename Eigen::internal::plain_col_type<Coeffs>::type CoeffVector;

  Eigen::Index N = U.rows(), J = U.cols();
  CAST_VEC(DiagOut, d, N);
  CAST_MAT(LowRankOut, W, N, J);
  CAST_BASE(Work, S);
  if (update_workspace) {
    S.derived().resize(N, J * J);
    S.row(0).setZero();
  }

  // This is a temporary vector used to minimize computations internally
  RowVector tmp;
  CoeffVector p;

  // This holds the accumulated value of the S matrix at each step
  Eigen::Matrix<Scalar, LowRank::ColsAtCompileTime, LowRank::ColsAtCompileTime, Eigen::ColMajor> Sn(J, J);

  // This is a flattened pointer to Sn that is used for copying the data
  Eigen::Map<typename Eigen::internal::plain_row_type<Work>::type> ptr(Sn.data(), 1, J * J);

  // First row
  Sn.setZero();
  d(0)                = a(0);
  W.row(0).noalias() = V.row(0) / d(0);

  // The rest of the rows
  for (Eigen::Index n = 1; n < N; ++n) {
    p = exp(c.array() * (t(n - 1) - t(n)));

    // Update S_n = diag(P) * (S_n-1 + d*W*W.T) * diag(P)
    Sn.noalias() += d(n - 1) * W.row(n - 1).transpose() * W.row(n - 1);
    Sn = p.asDiagonal() * Sn;

    // Save the current value of Sn to the workspace
    // Note: This is actually `diag(P) * (S + d*W*W.T)` without the final `* diag(P)`
    internal::update_workspace<update_workspace>::apply(n, ptr, S);

    // Incorporate the second diag(P) that we didn't include above for bookkeeping
    Sn *= p.asDiagonal();

    // Update d = a - U * S * U.T
    tmp  = U.row(n) * Sn;
    d(n) = a(n) - tmp * U.row(n).transpose();
    if (d(n) <= 0.0) return n;

    // Update W = (V - U * S) / d
    W.row(n).noalias() = (V.row(n) - tmp) / d(n);
  }

  return 0;
}

/**
 * \brief Apply a strictly lower matrix multiply
 *
 * This computes:
 *
 * `Z += tril(U * V^T) * Y`
```

# [5]
# summary & remaining issues

[X] rigorous inference
[X] huge datasets
[X] physics-based models
[X] finite time

**but…**

documentation & tutorials?

# General API quickstart¶

[1]:
```
import arviz as az
import matplotlib.pyplot as plt
import numpy as np
import pymc3 as pm
import theano.tensor as tt
import warnings

warnings.simplefilter(action="ignore", category=FutureWarning)
```

[2]:
```
%config InlineBackend.figure_format = 'retina'
az.style.use('arviz-darkgrid')
print('Running on PyMC3 v{}'.format(pm.__version__))
print('Running on ArviZ v{}'.format(az.__version__))
```

```
Running on PyMC3 v3.9.0
Running on ArviZ v0.8.3
```

## 1. Model creation¶

Models in PyMC3 are centered around the `Model` class. It has references to all random variables (RVs) and computes the model logp and its gradients. Usually, you would instantiate it as part of a `with` context:

[3]:
```
with pm.Model() as model:
    # Model definition
    pass
```

We discuss RVs further below but let's create a simple model to explore the `Model` class.

[4]:
```
with pm.Model() as model:
    mu = pm.Normal("mu", mu=0, sigma=1)
    obs = pm.Normal("obs", mu=mu, sigma=1, observed=np.random.randn(100))
```

[5]:
```
model.basic_RVs
```

[5]:
```
[mu, obs]
```

[6]:
```
model.free_RVs
```

[6]:
```
[mu]
```

[7]:
```
model.observed_RVs
```

[7]:
```
[obs]
```

[8]:
```
model.logp({"mu": 0})
```

[8]:
```
array(-136.56820547)
```

It's worth highlighting the design choice we made with `logp`. As you can see above, `logp` is being called with arguments, so it's a method of the model instance. More precisely, it puts

General API quickstart — PyMC

https://docs.pymc.io/notebooks/api_quickstart.html

PyMC3 Tutorials Examples Books + Videos API Developer Guide About PyMC3

# General API quickstart¶

[1]:
```python
import arviz as az
import matplotlib.pyplot as plt
import numpy as np
import pymc3 as pm
import theano.tensor as tt
import warnings

warnings.simplefilter(action="ignore", category=FutureWarning)
```

[2]:
```python
%config InlineBackend.figure_format = 'retina'
az.style.use('arviz-darkgrid')
print('Running on PyMC3 v{}'.format(pm.__version__))
print('Running on ArviZ v{}'.format(az.__version__))
```

```
Running on PyMC3 v3.9.0
Running on ArviZ v0.8.3
```

## 1. Model creation¶

**I'm sorry, what?**

Models in PyMC3 are centered around the `Model` class. It has references to all random variables (RVs) and computes the model logp and its gradients. Usually, you would instantiate it as part of a `with` context:

[3]:
```python
with pm.Model() as model:
    # Model definition
    pass
```

We discuss RVs further below but let's create a simple model to explore the `Model` class.

[4]:
```python
with pm.Model() as model:
    mu = pm.Normal("mu", mu=0, sigma=1)
    obs = pm.Normal("obs", mu=mu, sigma=1, observed=np.random.randn(100))
```

[5]:
```python
model.basic_RVs
```

[5]: `[mu, obs]`

[6]:
```python
model.free_RVs
```

[6]: `[mu]`

[7]:
```python
model.observed_RVs
```

[7]: `[obs]`

[8]:
```python
model.logp({"mu": 0})
```

[8]: `array(-136.56820547)`

It's worth highlighting the design choice we made with `logp`. As you can see above, `logp` is being called with arguments, so it's a method of the model instance. More precisely, it puts `reparameterization`

Fitting TESS data — exoplanet

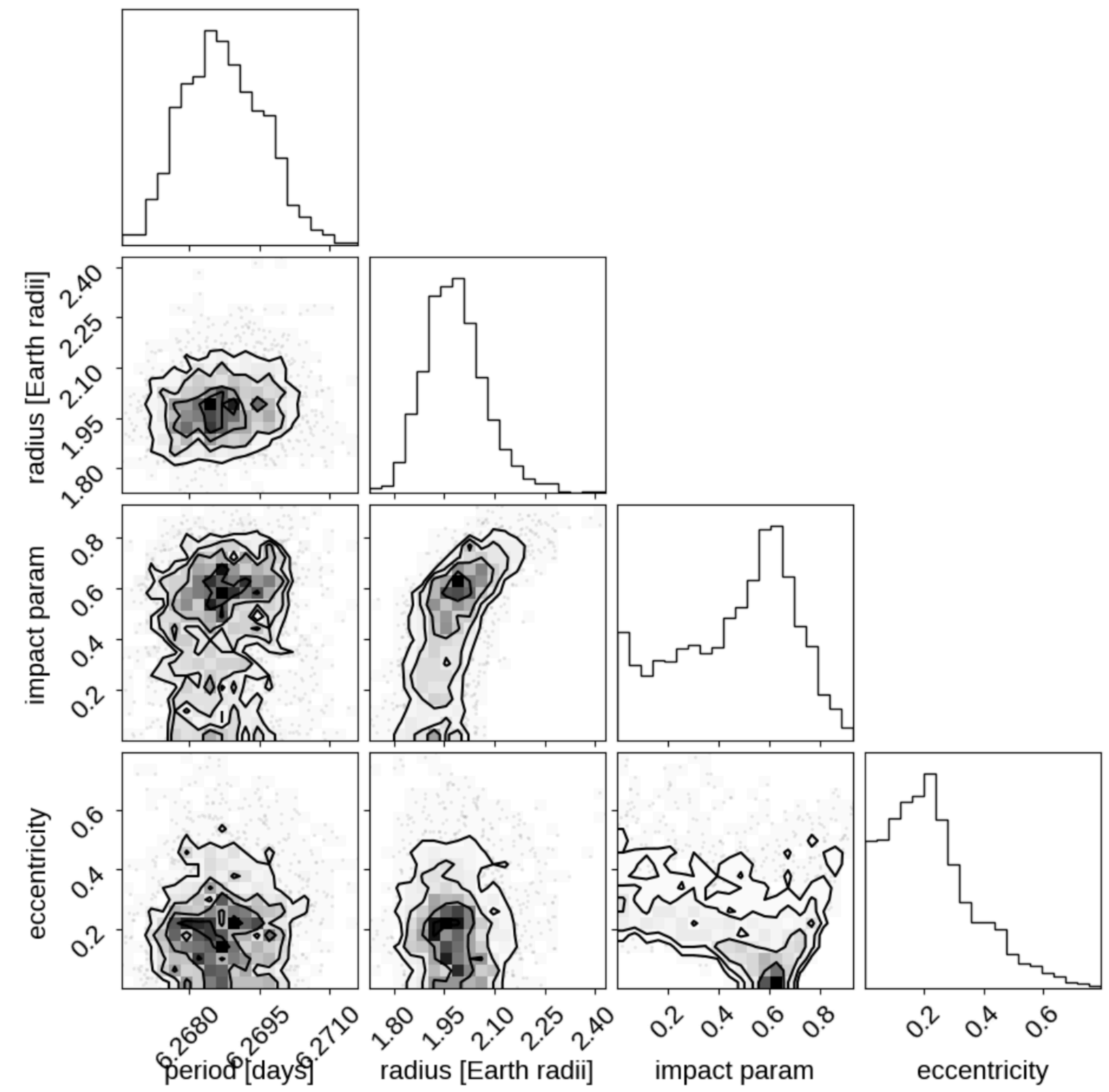
https://gallery.exoplanet.codes/tutorials/tess/

exoplanet

Search the docs ...

Radial velocity fitting
Transit fitting
Astrometric fitting
Gaussian process models for stellar variability
Joint RV & transit fits
**Fitting TESS data**
Quick fits for TESS light curves
Joint RV & transit fits
Simultaneous Fitting of a Transit with Stellar Variability
Fitting transit times
Fitting a detached eclipsing binary
RVs with multiple instruments
Light curves from multiple instruments
Main exoplanet docs

Theme by the Executable Book Project




Contents

Transit search
The transit model in PyMC3
**Results**
Citations

integration with **legacy code**
remains **Hard™**

**ambitious data analysis**

calls for

**powerful tools**

that may (or may not) work

**out of the box**

but we have

**work to do**

I want you to

**keep doing awesome science**

and

**learning about new methods**

then

**share what you learn**

and

**contribute back**

## get in touch!

dfm.io

github.com/dfm

twitter.com/exoplaneteer