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
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**
inference
want:

data  ⇒  physics
have:

physics \Rightarrow \text{data}
integral of the form
\[ f(\text{physics}) \ p(\text{physics} | \text{data}) \ d\text{physics} \]
[one option]

(markov chain) monte carlo

\[\text{physics} \sim p(\text{physics} \mid \text{data})\]
so, "all" you need is:

[a] a "good" sampler

[b] fast $p(\text{data} | \text{physics})$
[a] a "good" **sampler**

**cost per effective sample**
[a] a "good" sampler depends on \# of parameters and "geometry" of the problem
[b] fast $p(\text{data}|\text{physics})$ 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.

Basic Usage

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

```python
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.randn(ndim)
p0 = np.random.randn(nwalkers, ndim)

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

A more complete example is available in the Quickstart tutorial.
**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
The diagram illustrates the relationship between the number of parameters and the patience required. It shows that:

- A few parameters require a minimal amount of patience.
- Tenish (a term not clearly defined in the context) suggests a moderate level of patience.
- Not outrageously many parameters imply that the patience required is not excessively high.

The diagram is based on personal experience.
number of parameters

patience required

emcee

a few

tenish

not outrageously many

ref: personal experience
number of parameters

patience required

emcee

how things should be

ref: personal experience
rigorous inference
huge datasets
physics-based models
finite time
this is a solved problem*
[2] gradients
\[ \frac{dp(data|physics)}{dp\text{physics}} \]
$\log p(\text{data} | \text{physics})$
$-\log p(\text{data} \mid \text{physics}) = u(q)$
$-\log p(data | physics) = u(q)$
\[-\log p(\text{data} \mid \text{physics}) = u(q)\]

\[H(q, p) = T(p) + U(q)\]
cool. so we're done?
maybe!
to name a few ...

let's not get ahead of ourselves
but, probably not.
rigorous inference
huge datasets
physics-based models
finite time
\[ \frac{dp(data|physics)}{dphysics} \]
automatic differentiation
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
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"

GPU support, variational inference

Stan
- state-of-the-art inference algorithms

the new kid on the block

PyTorch + Pyro
- easy to extend

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

JAX
- all of
  + Numpyro/TF Probability
- the above?
my current recommendation
import numpy as np

⇒

import jax.numpy as jnp
a case study
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-Turn 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.
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.
[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:
$O(N\log N)$ scaling
[gaussian process]
with celerite:
\[ O(N) \] scaling
/ * brief Compute the Cholesky factorization of the system
  * This computes 'd' and 'w' such that:
    * 'K = L L^T' where 'K' is the celerite matrix and 'L' = \( \text{triu}(\text{lower}(K)) \)
    * 'd' and 'w' can point to 'a' and 'c_w' can point to 'v' and the memory will be reused. In this particular case, the
      'celerite::factor::factor_vw' function doesn't use 'a' and 'v', but this
      won't be true for all 'factor_vw' functions.
  */

template <bool update_workspace = true, typename Input, typename Coeffs, typename Diag, typename LowRank, typename DiagOut, typename LowRankOut> 

Eigens::Index factor(const Eigens::MatrixBase<Input>& A, // (N, N)
                     const Eigens::MatrixBase<Coeffs>& K,       // (N, N)
                     const Eigens::MatrixBase<Diag>& d,        // (N, 1)
                     const Eigens::MatrixBase<LowRank>& U,     // (N, N)
                     const Eigens::MatrixBase<DiagOut>& d_out, // (N, 1)
                     const Eigens::MatrixBase<LowRankOut>& U_out) // (N, N)
{
    Eigens::MatrixBase<DiagOut> diagOut = std::move(d_out);
    // (N, 1)
    Eigens::MatrixBase<LowRankOut> lowRankOut = std::move(U_out); // (N, N)
    // (N, N)

    if (update_workspace) { 
        // (N, N)
        // (N, N)
    }

    return diagOut, lowRankOut;
}
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.

```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_sqd(omega),
    }

np.random.seed(5659854)
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_mcm(coords, 2000, progress=True)
sampler.reset()
state = sampler.run_mcm(200, progress=True)
```

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.
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_sigma = numpyro.sample("log_sigma", dist.Normal(0.0, prior_sigma))
    log_rho = numpyro.sample("log_rho", dist.Normal(0.0, prior_sigma))
    log_tau = numpyro.sample("log_tau", dist.Normal(0.0, prior_sigma))
    term1 = jax_terms.UnDampedSHOTerm(
        sigma=jnp.exp(log_sigma), rho=jnp.exp(log_rho), 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=10000, num_chains=2)
rng_key = random.PRNGKey(34023)
time mcmc.run(rng_key, t, yerr, y=y)

CPU time: user 14.7 s, sys: 197 ms, total: 16.6 s
Wall time: 16.4 s
[recently] generalized to multivariate data
celerite2.readthedocs.io
if P is None:
    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, Array]

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

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

def _matmul_impl(state: Carry, data: MatmulData) -> Tuple[Carry, Array]:
    (fp, vp, yp) = state
    Wh, Pn, Yn = data
    Fn = _pdpot(Pn, fp + jnp.outer(vp, yp))
    return (Fn, Wh, Yn, Fn)
rigorous inference
huge datasets
physics-based models
finite time
but...
documentation & tutorials?
General API quickstart

```python
[1]:
import arvi as arvi
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'
ax.style.use('arvi-darkgrid')
print('Running on PyMC3 v3.8.0')
print('Running on Arvi v0.6.1')

1. Model creation

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

```python
[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.

```python
[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))

model.basic_RVs
[5]:
[mu, obs]

model.free_RVs
[6]:
[mu]

model.observed_RVs
[7]:
[obs]

model.logp('mu')
[8]: array(-136.56820547)
```

It’s worth highlighting the design choice we made with `pm.Model`. As you can see above, `pm.Model` is being called with arguments, so it’s a method of the model instance. More precisely, it returns an instance of a model class.
General API quickstart

```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)

# config
In [1]: InlineBackend.figure_format = 'retina'
In [2]: as.style.use('arviz-darkgrid')
print('Running on PyMC3 v{}'.format(pm.version))
print('Running on ArviZ v{}'.format(as.__version__))
In [3]: Running on PyMC3 v1.0
Running on ArviZ v0.8.1

1. Model creation

Models in PyMC3 are centered around the Model class. It has references to the model itself, and collects the model’s log and its gradients. Usually, you would instantiate it as part of a with context.

```python
with pm.Model() as model:
    # model definition
    mu = pm.Normal('mu', mu=0, sigma=1)
    obs = pm.Normal('obs', mu=mu, sigma=1, observed=np.random.randn(100))

model.basic_RVs

[1]: [mu, obs]
[2]: model.free_RVs
[3]: [mu]
[4]: model.observed_RVs
[5]: [obs]
[6]: model.logp('mu', 0)
[7]: array(-136.56820547)
```

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