NIFTy.re – A Library for Gaussian Processes & VI for Inferences with Billions of Parameter

Gordian Edenhofer[1,2]

[1]Max Planck Institute for Astrophysics, Garching bei München, Germany, DE
[2]Faculty of Physics, LMU, München, Germany, DE

ERUM-IFT Invited Talk, Monday 20th November, 2023

Why GPs & VI

`NIFTy.re` = `NIFTy` + `jax`

GPs & VI for Astrophysics

Why GPs & VI

`NIFTy.re` = `NIFTy` + `jax`

GPs & VI for Astrophysics

Use Cases

Imaging = (**high dimensional posteriors** → **VI**) + (**smooth structures** → **GPs**)

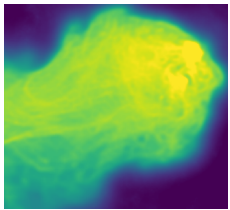


Figure: Cygnus A[1].

Figure: M87*[1].

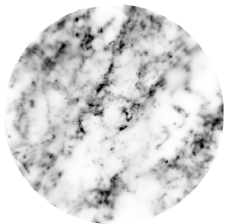


Figure: 3D Dust[1].

[1] Roth et al., "Bayesian radio interferometric imaging with direction-dependent calibration"; Arras et al., "Variable structures in M87* from space, time and frequency resolved interferometry"; Edenhofer, Zucker, et al., "A Parsec-Scale Galactic 3D Dust Map out to 1.2 kpc from the Sun".

Use Cases

- GPs on structured spaces with 500M+ parameters
- Highly structured models: specific likelihoods + physical priors

NIFTy great for GPs on structured spaces and inference of structured models

Why GPs & VI

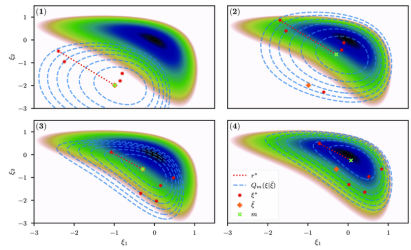
`NIFTy.re` = `NIFTy` + `jax`

GPs & VI for Astrophysics

`NIFTy.re = NIFTy + jax`

NIFTy

- Versatile library for expressing GPs on structured spaces[2]
- Flexible VI using the information about the shape of the model for an accurate yet cheap posterior approximation[2]



[2] Arras et al., "Variable structures in M87* from space, time and frequency resolved interferometry"; Edenhofer, R. H. Leike, et al., "Sparse Kernel Gaussian Processes through Iterative Charted Refinement (ICR)"; Knollmüller and Enßlin, *Metric Gaussian Variational Inference*; Frank, R. Leike, and Enßlin, "Geometric Variational Inference".

JAX

- Flexible library for composing and transforming numerical computations by Google
- Think: NumPy + SciPy + Auto-Differentiation (`jax.jvp`, `jax.vjp`)
- Vectorize code with "broadcasting on steroids"
- Compile code for CPU, GPU and TPU (with constrained set of computations via `jax.jit`)

NIFTy Core Components: NIFTy + JAX = NIFTy.re

NIFTy model

$$\ln P(\theta|d) \propto \text{reduction}(d, f(\theta)) + \text{regularization}(\theta)$$

with parameters θ, a reduction combining data d and model f, and a regularization.

VI using JAX to approximate the posterior using Jacobians

$$J^{\dagger}_{\theta,\bar{\xi}} J^{\dagger}_{f,\bar{\theta}} \; N_{\text{reduction}} \; J_{f,\bar{\theta}} J_{\theta,\bar{\xi}} + \mathbb{1}$$

with matrix $N_{\text{reduction}}$, $J_{\square,\triangle}$ the Jacobian of $\square$ at location $\triangle$, and ξ latent parameters.

Why GPs & VI

`NIFTy.re` = `NIFTy` + `jax`

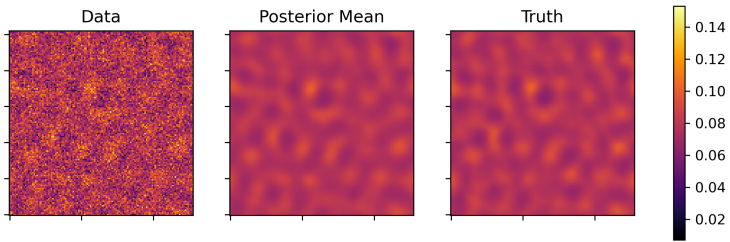
GPs & VI for Astrophysics

Example

Model

$$\tau \sim \mathcal{GP}(\tau \,|\, 0, k)$$
$$\alpha \sim \ln \mathcal{N}(\alpha |\, \bar{\alpha}, \sigma_\alpha)$$
$$s = \alpha \cdot \exp(\tau)$$
$$d \sim \mathcal{N}(d \,|\, s, N)$$

with unknown kernel k, given data d, and given noise covariance N.



Example: Priors

```
dims = (128, 128)
cf_zm = {"offset_mean": 0., "offset_std": (1e-3, 1e-4)}
cf_fl = {
  "fluctuations": (1e-1, 5e-3),
  "loglogavgslope": (-1., 1e-2),
  "flexibility": (1e+0, 5e-1),
  "asperity": (5e-1, 5e-2),
  "harmonic_domain_type": "Fourier",
}
cfm = jft.CorrelatedFieldMaker("cf")
cfm.set_amplitude_total_offset(**cf_zm)
cfm.add_fluctuations(
  dims, distances=1. / dims[0], **cf_fl,
  prefix="ax1", non_parametric_kind="power"
)
correlated_field = cfm.finalize()
```

Example: Priors

```
scaling = jft.LogNormalPrior(3., 1., name="scaling", shape=(1, ))
# Equivalently for InvGammaPrior, LaplacePrior, LogNormalPrior,
# and NormalPrior
```

Example: Priors

```
# All of these are simple transformations from standard normal
# parameters to the desired distribution (c.f. Inverse Transform Sampling)
log_mean, log_std = ...

def scaling_call(x):
  return jnp.exp(log_mean + log_std * x["scaling"])
```

Example: Priors

```
class CustomLogNormalPrior(jft.Model):
  def __init__(self, log_mean=0., log_std=1., name="scaling", shape=(1, )):
    self.log_mean = log_mean
    self.log_std = log_std
    self.name = name
    super().__init__(domain={name: jft.ShapeWithDtype(shape)})

  def __call__(self, x):
    # NOTE, think of ‘Model‘ as being just a plain function that takes some
    # input and performs all the necessary computation for your model.
    # Note, ‘scaling‘ here is completely degenarate with ‘offset_std‘ in the
    # likelihood but the priors for them are very different.
    return jnp.exp(self.log_mean + self.log_std * x[self.name])

scl = CustomLogNormalPrior(0., 1., name="scaling", shape=(1, ))
```

Example: Priors

```
class Signal(jft.Model):
  def __init__(self, correlated_field, scaling):
    self.cf = correlated_field
    self.scaling = scaling
    super().__init__(init=self.cf.init | self.scaling.init)

  def __call__(self, x):
    # NOTE, think of ‘Model‘ as being just a plain function that takes some
    # input and performs all the necessary computation for your model.
    # Note, ‘scaling‘ here is completely degenarate with ‘offset_std‘ in the
    # likelihood but the priors for them are very different.
    return self.scaling(x) * jnp.exp(self.cf(x))

signal = Signal(correlated_field, scaling)
signal.domain  # Parameters of the model
signal.target  # Output of the model
```

Example: Likelihood

```
signal_response = signal
noise_cov = lambda x: 0.1**2 * x
noise_cov_inv = lambda x: 0.1**-2 * x

nll = jft.Gaussian(data, noise_cov_inv) @ signal_response
```

Example: Inference

```
key, subkey = random.split(key)
pos_init = jft.Vector(jft.random_like(subkey, signal_response.domain))
n_samples = 4
n_vi_iterations = 20
samples, state = jft.optimize_kl(
  nll,
  pos_init,
  n_vi_iterations,
  n_samples,
  key,
  point_estimates=(),
  [...]
  verbosity=0
)
```

Hands-On

```
$ pip install --user --upgrade "jax[cpu]"
$ pip install --user git+https://gitlab.mpcdf.mpg.de/ift/nifty.git@NIFTY_8
```

```
>> import nifty8.re as jft
```

Hands-On

iPython

Appendix

Appendix

Pure functions

- Function depends exclusively and deterministically on its arguments[3]
  - No mutable `global` variables
  - No mutable references as argument
- Function has not side-effects
  - No mutations
  - No input- or output-streams

[3] Google, *Common Gotchas in JAX*.

Appendix

In-Place Updates

are impure and thus not permitted

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

a_np = np.array([1., 2., 3.])
a_jnp = jnp.array(a_np)
a_np[2:] = 10.

np.testing.assert_allclose(
  a_np,
  a_jnp.at[2:].set(10.)
)
```

Appendix

Control Flow with JIT

- Imposes much tighter constraints than required by e.g. JAX's `grad`
- Computation must be transparent to JAX
  - No branching in python (i.e. no `if`)
  - Use `jax.lax.cond` in favor of `if`
  - Use `jax.lax.scan`, `lax.fori_loop` ... in favor of `while`, `range`, ...
- Output shape and data-type may **only** depend on input shape and data-type

Appendix

Goodies

- Jacobian-Vector-Product (`jax.jvp`, `jax.linearize`)
- Vector-Jacobian-Product (`jax.vjp`)
- Vectorize (`jax.vmap`)
- Parallelize (`jax.pmap`)
- Transpose (`jax.linear_transpose`)
- Evaluate shape of function output (`jax.eval_shape`)