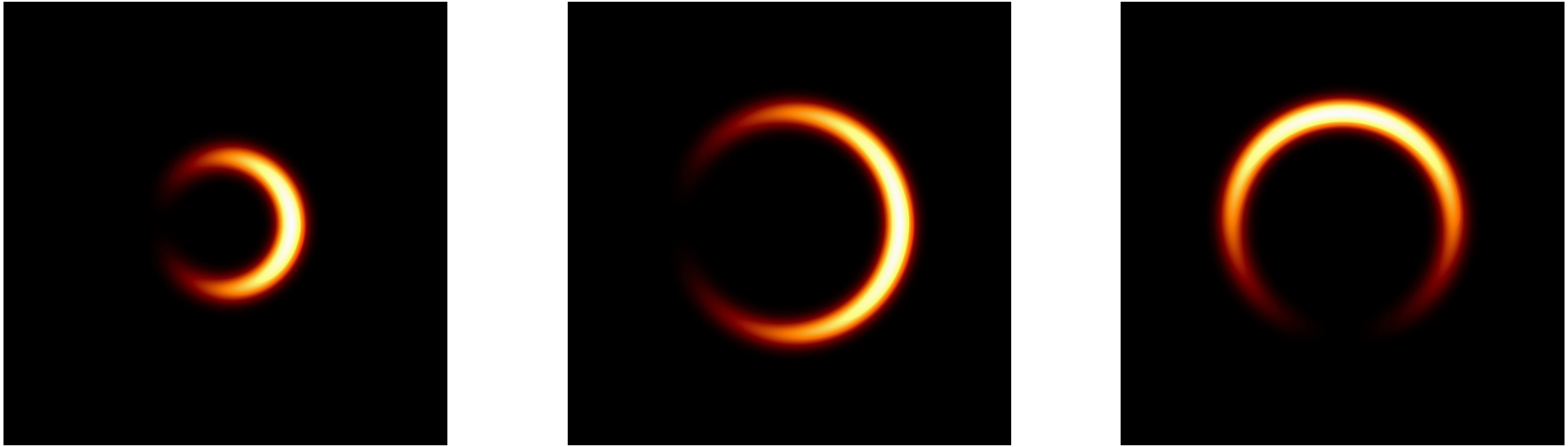
Accelerated inference for VLBI analysis

using JAX and NumPyro

Peter Brin
pbrin@cfa.harvard.edu

Geometric Model Fitting



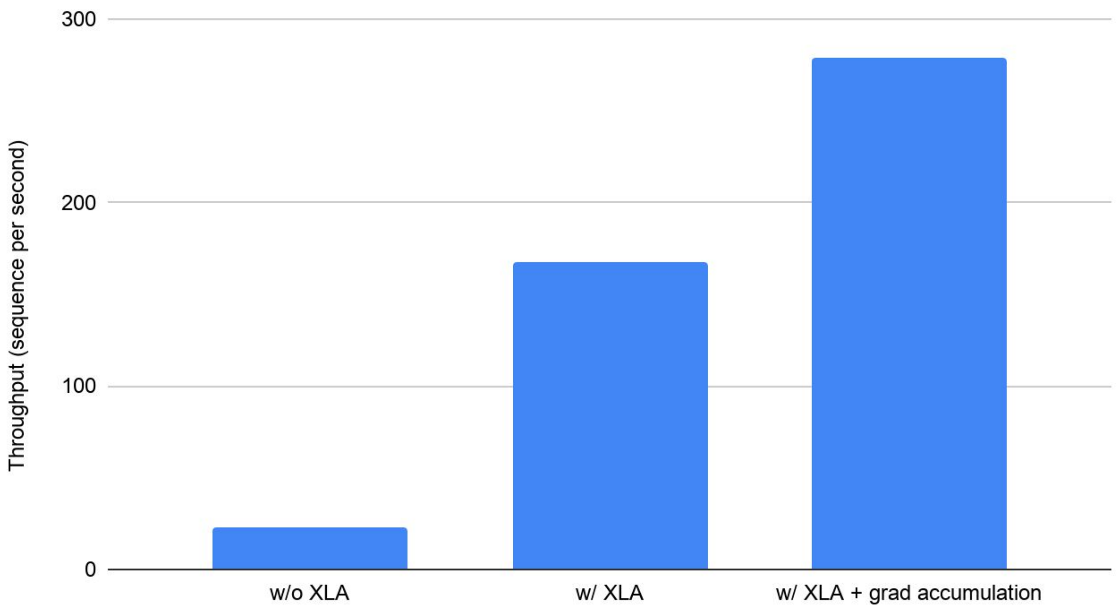
Geometric Model Fitting

```
|-------------------| 77.92% [6234/8000 26:15<07:26 Sampling 4 chains, 1,520 divergences]
```

What is JAX?

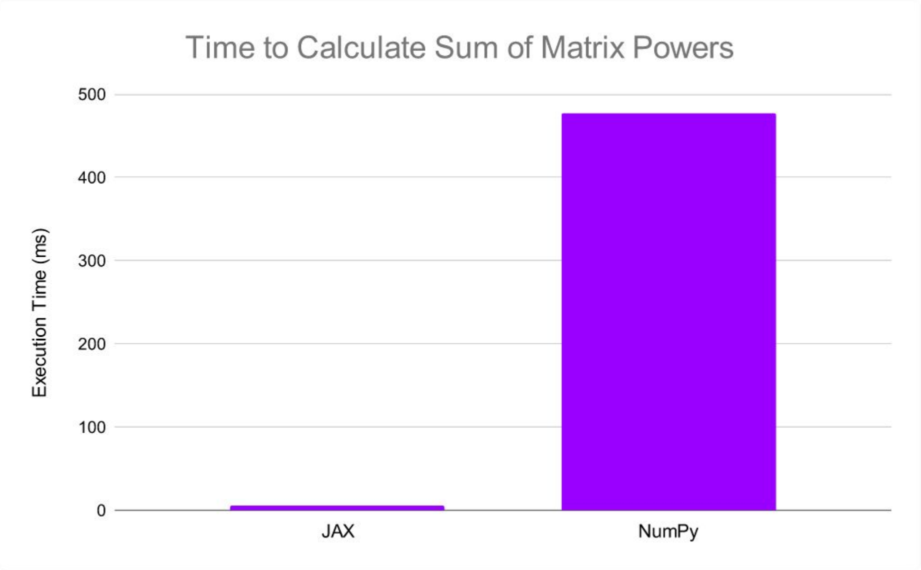
JAX is Autograd and XLA, brought together for high-performance machine learning research.

XLA: Accelerated Linear Algebra



credit: Google, sourced from https://www.tensorflow.org/xla

JAX: speed



JAX has the potential to be orders of magnitude faster than NumPy (n.b. JAX is using TPU and NumPy is using CPU in order to highlight that JAX's speed ceiling is much higher than NumPy's)

credit: Assembly AI, source https://www.assemblyai.com/blog/why-you-should-or-shouldnt-be-using-jax-in-2022/

What is JAX?

JAX is Autograd and XLA, brought together for high-performance machine learning research.

It provides composable transformations of Python+NumPy programs: differentiate, vectorize, parallelize, Just-In-Time compile to GPU/TPU, and more.

What is JAX?

JAX is Autograd and XLA, brought together for high-performance machine learning research.

It provides composable transformations of Python+NumPy programs: differentiate, vectorize, parallelize, Just-In-Time compile to GPU/TPU, and more.

Same familiar numpy API

np

```
def circ_gauss_sample_uv(u, v):
    val = (params['F0']
            * np.exp(-np.pi**2/(4.*np.log(2.)) * (u**2 + v**2) * params['FWHM']**2)
            * np.exp(1j * 2.0 * np.pi * (u * params['x0'] + v * params['y0'])))

    return val
```

np -> jnp

```
def circ_gauss_sample_uv(u, v):
    val = (params['F0']
            * jnp.exp(-jnp.pi**2/(4.*jnp.log(2.)) * (u**2 + v**2) * params['FWHM']**2)
            * jnp.exp(1j * 2.0 * jnp.pi * (u * params['x0'] + v * params['y0'])))

    return val
```

JAX code for CPUs

```python
def circ_gauss_sample_uv(u, v):
    val = (params['F0']
           * jnp.exp(-jnp.pi**2/(4.*jnp.log(2.)) * (u**2 + v**2) * params['FWHM']**2)
           * jnp.exp(1j * 2.0 * jnp.pi * (u * params['x0'] + v * params['y0'])))

    return val
```

JAX code for GPUs

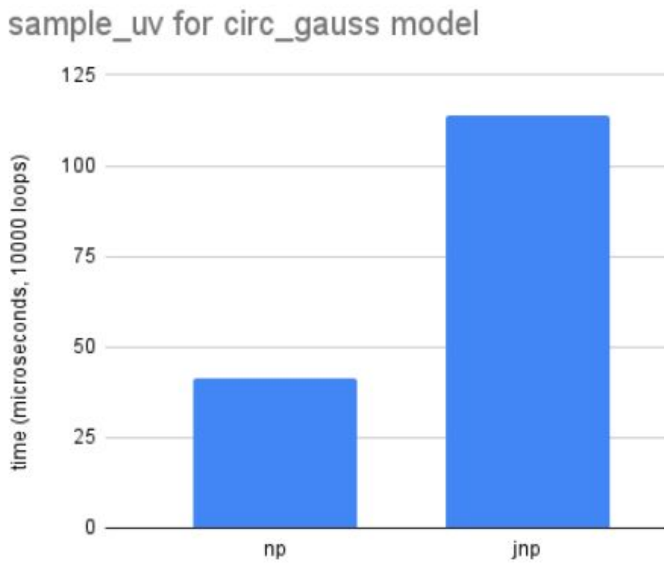
```python
def circ_gauss_sample_uv(u, v):
    val = (params['F0']
            * jnp.exp(-jnp.pi**2/(4.*jnp.log(2.)) * (u**2 + v**2) * params['FWHM']**2)
            * jnp.exp(1j * 2.0 * jnp.pi * (u * params['x0'] + v * params['y0'])))

    return val
```

JAX - timings

```
params = {'F0':1.3, 'x0':0, 'y0':0, 'FWHM':50*eh.RADPERUAS}

eht = eh.array.load_txt('EHT2025.txt')
model = eh.model.Model()
model = model.add_circ_gauss(**params)
tint_sec = 5
tadv_sec = 3600
tstart_hr = 0
tstop_hr = 24
bw_hz = 1e9
obs = model.observe(eht, tint_sec, tadv_sec, tstart_hr, tstop_hr, bw_hz, ampcal=True, phasecal=True,seed=4)
u = obs.data['u']
v = obs.data['v']
```

```
%timeit np_circ_gauss_sample_uv(u,v)
%timeit jnp_circ_gauss_sample_uv(u,v).block_until_ready()
```

JAX - timings

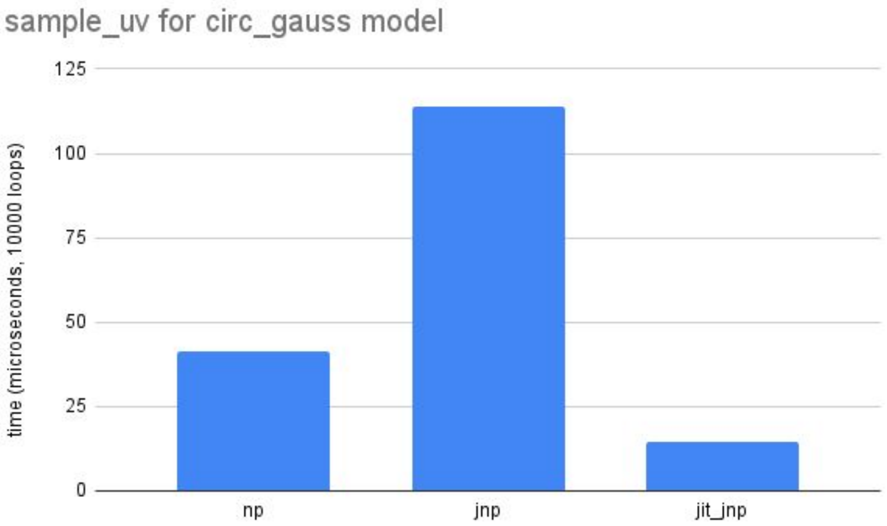

sample_uv for circ_gauss model
125
100
75
50
25
0
time (microseconds, 10000 loops)
np
jnp

JIT

```
params = {'F0':1.3, 'x0':0, 'y0':0, 'FWHM':50*eh.RADPERUAS}

eht = eh.array.load_txt('EHT2025.txt')
model = eh.model.Model()
model = model.add_circ_gauss(**params)
tint_sec = 5
tadv_sec = 3600
tstart_hr = 0
tstop_hr = 24
bw_hz = 1e9
obs = model.observe(eht, tint_sec, tadv_sec, tstart_hr, tstop_hr, bw_hz, ampcal=True, phasecal=True,seed=4)
u = obs.data['u']
v = obs.data['v']
```

```
jit_jnp_circ_gauss_sample_uv = jax.jit(jnp_circ_gauss_sample_uv)

#Running this once beforehand will ensure JIT compilation time for our function isn't added to our benchmarks:
jit_jnp_circ_gauss_sample_uv(u[0], v[0])

%timeit jit_jnp_circ_gauss_sample_uv(u,v).block_until_ready()
```

JIT


sample_uv for circ_gauss model
125
100
75
50
25
0
time (microseconds, 10000 loops)
np
jnp
jit_jnp

Autograd

```python
import jax
import jax.numpy as jnp
import matplotlib.pyplot as plt

def f(x):
    return jnp.exp( (-1)*(x**3 + x + jnp.sin(jnp.pi*x)) )
```

$$f(x) = e^{-x^3 - x - \sin(\pi x)}$$

Autograd

```
import jax
import jax.numpy as jnp
import matplotlib.pyplot as plt

def f(x):
    return jnp.exp( (-1)*(x**3 + x + jnp.sin(jnp.pi*x)) )

def manual_grad_f(x):
    return ((-1)*(3*x**2 + 1 + jnp.cos(jnp.pi*x)*jnp.pi)
            * jnp.exp( (-1)*(x**3 + x + jnp.sin(jnp.pi*x)) ))
```

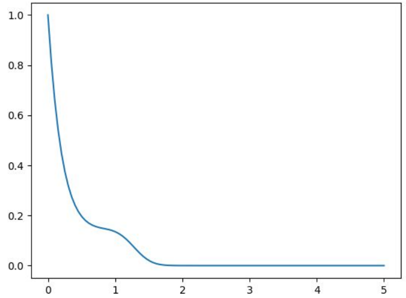
$$f(x) = e^{-x^3 - x - \sin(\pi x)}$$

Autograd

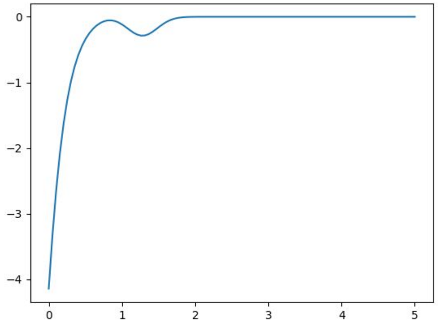
$$f(x) = e^{-x^3 - x - \sin(\pi x)}$$

```
import jax
import jax.numpy as jnp
import matplotlib.pyplot as plt

def f(x):
    return jnp.exp( (-1)*(x**3 + x + jnp.sin(jnp.pi*x)) )

def manual_grad_f(x):
    return ((-1)*(3*x**2 + 1 + jnp.cos(jnp.pi*x)*jnp.pi)
            * jnp.exp( (-1)*(x**3 + x + jnp.sin(jnp.pi*x)) ))

jax_grad_f = jax.grad(f)
```

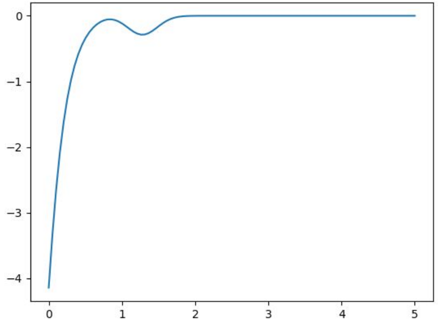
Autograd



f(x)



manual_grad_f



jax_grad_f

JAX (+scipy)

```
from jax.scipy.special import i0
```

JAX (+scipy?)

```python
from jax.scipy.special import i0
from jax.scipy.special import jv
```

```
IPython 8.2.0 -- An enhanced Interactive Python. Type '?' for help.
---------------------------------------------------------------------------
ImportError                               Traceback (most recent call last)
File ~/sao/presentation/scratch.py:8, in <module>
      4 import matplotlib.pyplot as plt
      7 from jax.scipy.special import i0
----> 8 from jax.scipy.special import jv
     10 # some computation that uses jv, ive, other Bessel functions
     12 def f(x):

ImportError: cannot import name 'jv' from 'jax.scipy.special' (/home/nova/.local/lib/python3.10/site-packages/jax/scipy/special.py)
```

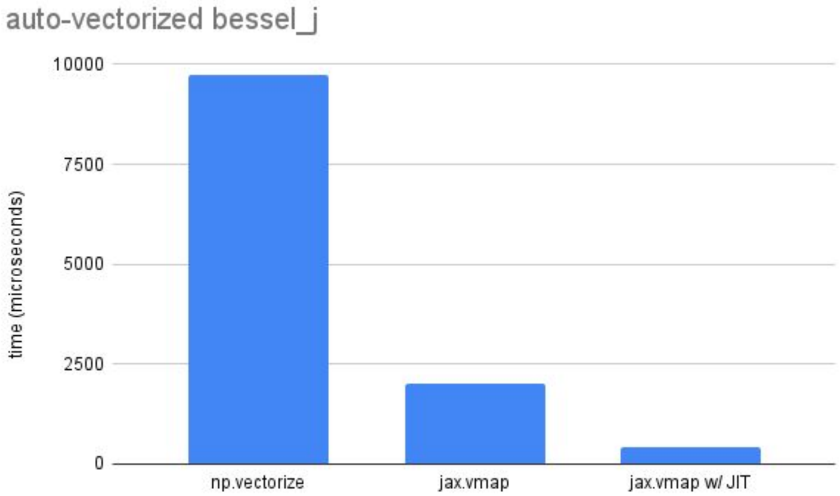
JAX (-scipy)

```python
# replacement for scipy.special.jv, which is not available in jax
# computed via trapz using the integral definition of J
def bessel_j(n, z, num_samples=100):
    z = jnp.asarray(z)
    scalar = z.ndim == 0
    if scalar:
        z = z[np.newaxis]
    z = z[:, np.newaxis]
    tau = np.linspace(0, jnp.pi, num_samples)
    integrands = jnp.trapz(jnp.cos(n*tau - z*jnp.sin(tau)), x=tau)
    if scalar:
        return (1./jnp.pi)*integrands.squeeze()
    return (1./jnp.pi)*integrands
```

vmap

```
def bessel_j_vtest(n, z, num_samples=100):
    tau = np.linspace(0, jnp.pi, num_samples)
    integrands = jnp.trapz(jnp.cos(n*tau - z*jnp.sin(tau)), x=tau)
    return (1./jnp.pi)*integrands

bessel_j_vtest = jax.vmap(bessel_j_vtest, in_axes=(0, 0), out_axes=0)
bessel_j_vtest(np.arange(100), np.arange(100))
```

vmap - running time



vmap: speed test

```
mat = random.normal(key, (150, 100))
batched_x = random.normal(key, (10, 100))

def apply_matrix(v):
  return jnp.dot(mat, v)
```

```
def naively_batched_apply_matrix(v_batched):
  return jnp.stack([apply_matrix(v) for v in v_batched])

print('Naively batched')
%timeit naively_batched_apply_matrix(batched_x).block_until_ready()
```

```
@jit
def vmap_batched_apply_matrix(v_batched):
  return vmap(apply_matrix)(v_batched)

print('Auto-vectorized with vmap')
%timeit vmap_batched_apply_matrix(batched_x).block_until_ready()
```

Example from JAX documentation: https://jax.readthedocs.io/en/latest/notebooks/quickstart.html

vmap: speed test

```
Naively batched
1.35 ms ± 3.78 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

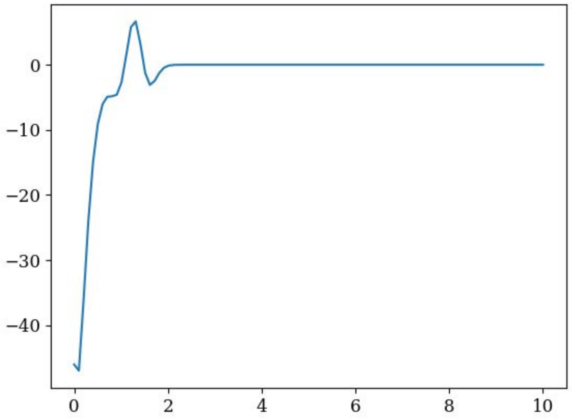
```
Manually batched
11.9 µs ± 81.4 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

```
Auto-vectorized with vmap
36.5 µs ± 73 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

Example from JAX documentation: https://jax.readthedocs.io/en/latest/notebooks/quickstart.html

JAX: composability

```
x = jnp.linspace(0, 10, 100)
y = jax.jit(jax.vmap(jax.grad(jax.grad(jax.grad(f)))))(x)
plt.plot(x, y)
```



JAX underneath the hood

Intermediate language: Jaxpr

```
def f(x):
    return jnp.exp( (-1)*(x**3 + x + jnp.sin(jnp.pi*x)) )

print(jax.make_jaxpr(f)(1.))
```

[6] ✓ 0.5s

```
{ lambda ; a:f32[]. let
    b:f32[] = integer_pow[y=3] a
    c:f32[] = add b a
    d:f32[] = mul a 3.141592653589793
    e:f32[] = sin d
    f:f32[] = add c e
    g:f32[] = mul f -1.0
    h:f32[] = exp g
  in (h,) }
```

JAX underneath the hood

Tracers: abstract, concrete

```python
@jit
def f(x, y):
  print("Running f():")
  print(f"  x = {x}")
  print(f"  y = {y}")
  result = jnp.dot(x + 1, y + 1)
  print(f"  result = {result}")
  return result

x = np.random.randn(3, 4)
y = np.random.randn(4)
f(x, y)
```

```
Running f():
  x = Traced<ShapedArray(float32[3,4])>with<DynamicJaxprTrace(level=0/1)>
  y = Traced<ShapedArray(float32[4])>with<DynamicJaxprTrace(level=0/1)>
  result = Traced<ShapedArray(float32[3])>with<DynamicJaxprTrace(level=0/1)>
```

From JAX documentation: https://jax.readthedocs.io/en/latest/notebooks/quickstart.html

JAX underneath the hood

High level API, jax.numpy, provides familiar np interface

Lower level API, jax.lax, maps directly to XLA

🔪 JAX: The Sharp Bits 🔪

Pure functions

🔪 JAX: The Sharp Bits 🔪

Pure functions

Immutable data types

🔪 JAX: The Sharp Bits 🔪

Pure functions

Immutable data types

No global state

🔪 JAX: The Sharp Bits 🔪

Pure functions

Immutable data types

No global state

Random number generation

🔪 JAX: The Sharp Bits 🔪

Pure functions

Immutable data types

No global state

Random number generation

Unfamiliar error messages

🔪 JAX: The Sharp Bits 🔪

```
---------------------------------------------------------------------------
ConcretizationTypeError                   Traceback (most recent call last)
/home/nova/sao/modeling-tutorial/jax_numpyro_model_fitting.ipynb Cell 8 in <cell line: 6>()
      3         return 0
      4     return x
----> 6 jax.jit(f)(2.)

    [... skipping hidden 14 frame]

 TracerArrayConversionError: The numpy.ndarray conversion method __array__() was called on th
 xprTrace(level=0/1)>
 While tracing the function f at /tmp/ipykernel_62806/90201900.py:3 for jit, this concrete va
 t 'n'.
 See https://jax.readthedocs.io/en/latest/errors.html#jax.errors.TracerArrayConversionError

 UnexpectedTracerError                    Traceback (most recent call last)
 /home/nova/sao/modeling-tutorial/jax_numpyro_model_fitting.ipynb Cell 10 in <cell line: 10>()
       8 f(3)
      10 for elt in y:
 ---> 11     print(elt + 3)
```

🔪 JAX: The Sharp Bits 🔪

Pure functions

Immutable data types

No global state

Random number generation

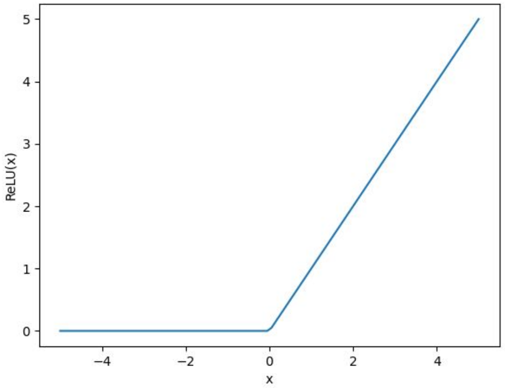
Unfamiliar error messages

Control flow (!)

🔪 JAX: The Sharp Bits 🔪

Control flow (!) (pt. 2)

```python
@jax.jit
def relu(x):
    if x<0:
        return 0
    return x

relu(-3)
```

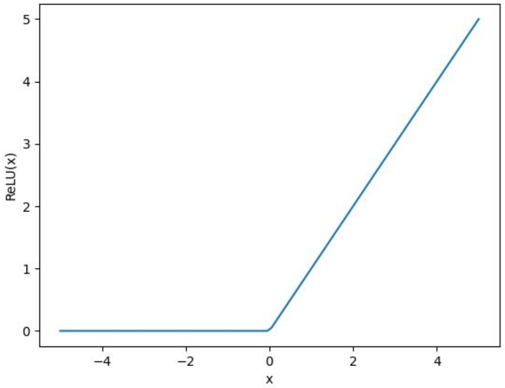


🔪 JAX: The Sharp Bits 🔪

Control flow (!) (pt. 2)

```python
@jax.jit
def relu(x):
    if x<0:
        return 0
    return x

relu(-3)
```

[21] 0.9s




```
---------------------------------------------------------------------------
ConcretizationTypeError                   Traceback (most recent call last)
/home/nova/sao/modeling-tutorial/jax_numpyro_model_fitting.ipynb Cell 11 in <cell line: 7>()
      4         return 0
      5     return x
----> 7 relu(-3)

    [... skipping hidden 14 frame]
```

🔪 JAX: The Sharp Bits 🔪

Control flow (!) (pt. 3)

construct	jit	grad
if	✗	✔
for	✔*	✔
while	✔*	✔
lax.cond	✔	✔
lax.while_loop	✔	fwd
lax.fori_loop	✔	fwd
lax.scan	✔	✔

* = argument-**value**-independent loop condition - unrolls the loop

numpyro

Probabilistic programming language

Bayesian inference tasks

HMC/NUTS sampling

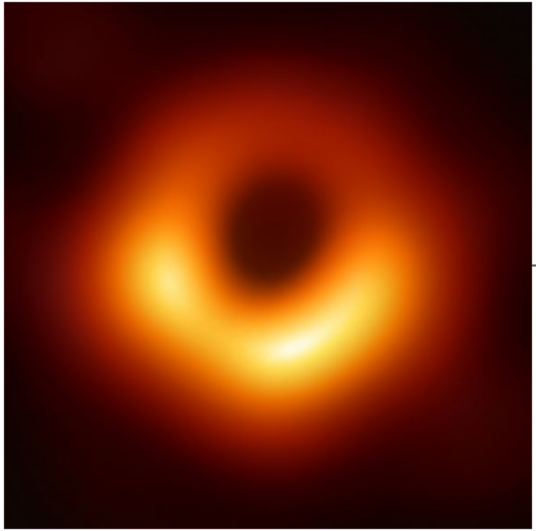
Accelerated VLBI modeling with JAX and numpyro

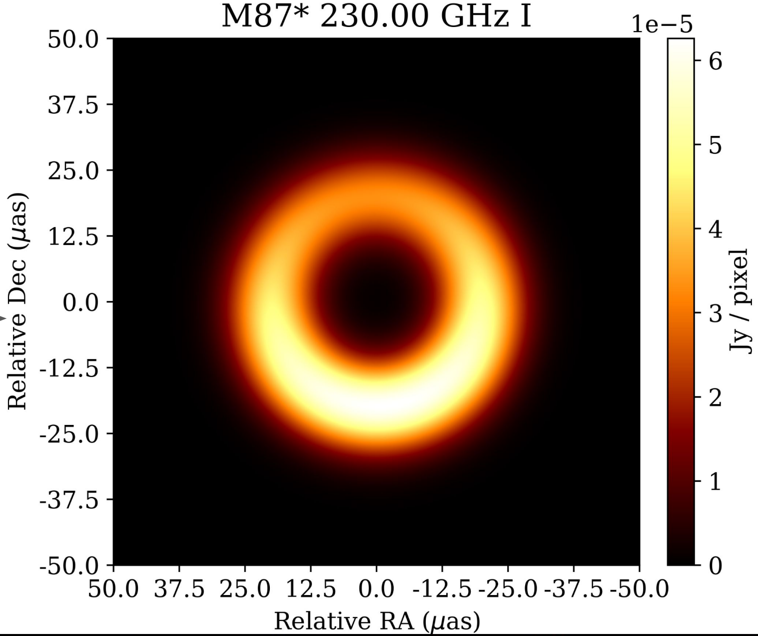
```
def gaussian_model():
    dx = numpyro.sample("dx", dist.Uniform(low=-40, high=40))
    dy = numpyro.sample("dy", dist.Uniform(low=-40, high=40))
    h = numpyro.sample("h", dist.Uniform(low=0, high=80))
    f = numpyro.sample("f", dist.Uniform(low=0, high=10))
    ft = generate_circ_gauss_model(dx, dy, h, f)

    u = obs.data['u']
    v = obs.data['v']
    vis = obs.data['vis']
    sigma = obs.data['sigma']

    # Fit real and imaginary parts separately.
    numpyro.sample("re(obs)", dist.Normal(ft(u,v).real, sigma), obs = vis.real)
    numpyro.sample("im(obs)", dist.Normal(ft(u,v).imag, sigma), obs = vis.imag)
```

Accelerated VLBI modeling with JAX and numpyro

```
# Start from this source of randomness. We will split keys for subsequent operations.
rng_key = random.PRNGKey(0)
rng_key, rng_key_ = random.split(rng_key)

# Run NUTS.
kernel = NUTS(PPL_model)
num_warmup = 1000
num_samples = 2000
mcmc = MCMC(kernel, num_warmup=num_warmup, num_samples=num_samples)
mcmc.run(
    rng_key_
)
mcmc.print_summary()
```

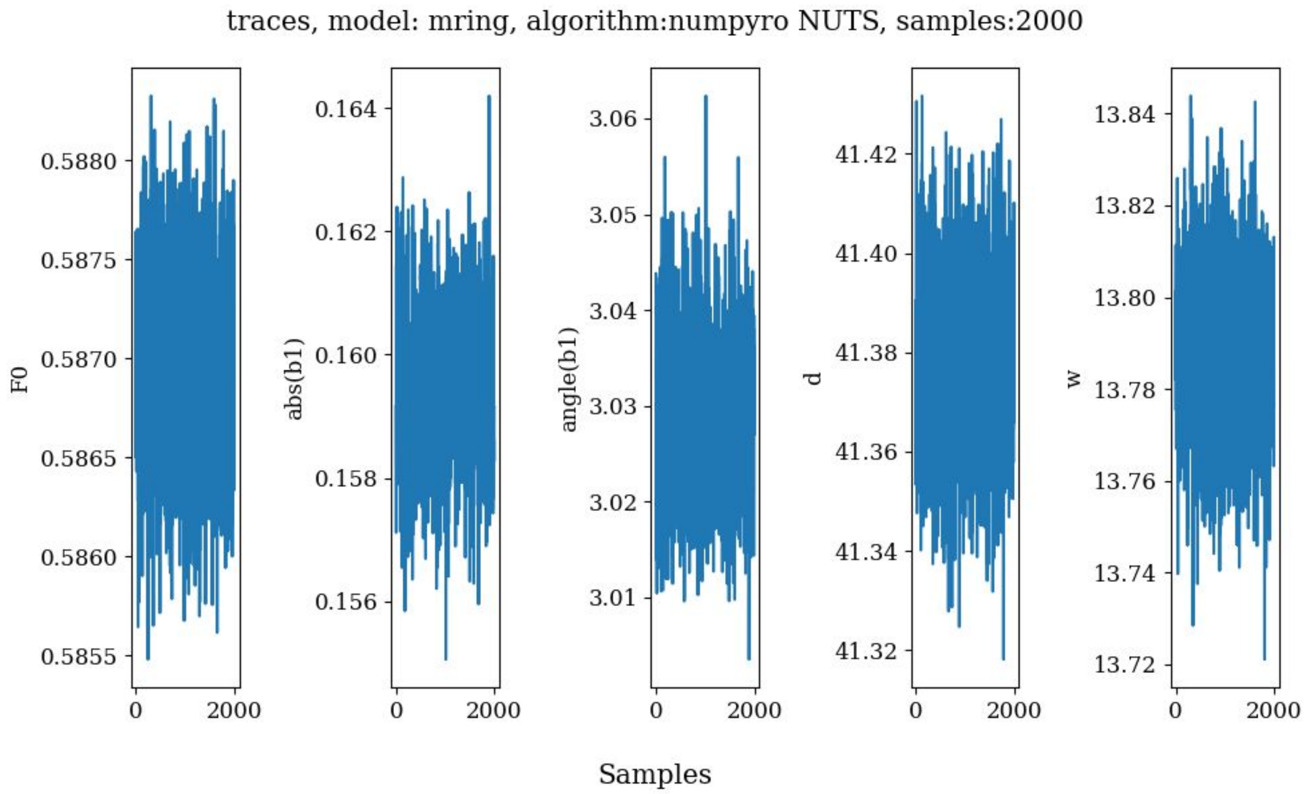
Model fitting M87* 2017 April 11 Stokes I visibility amplitudes and closure phases to an m-ring model (m=1)



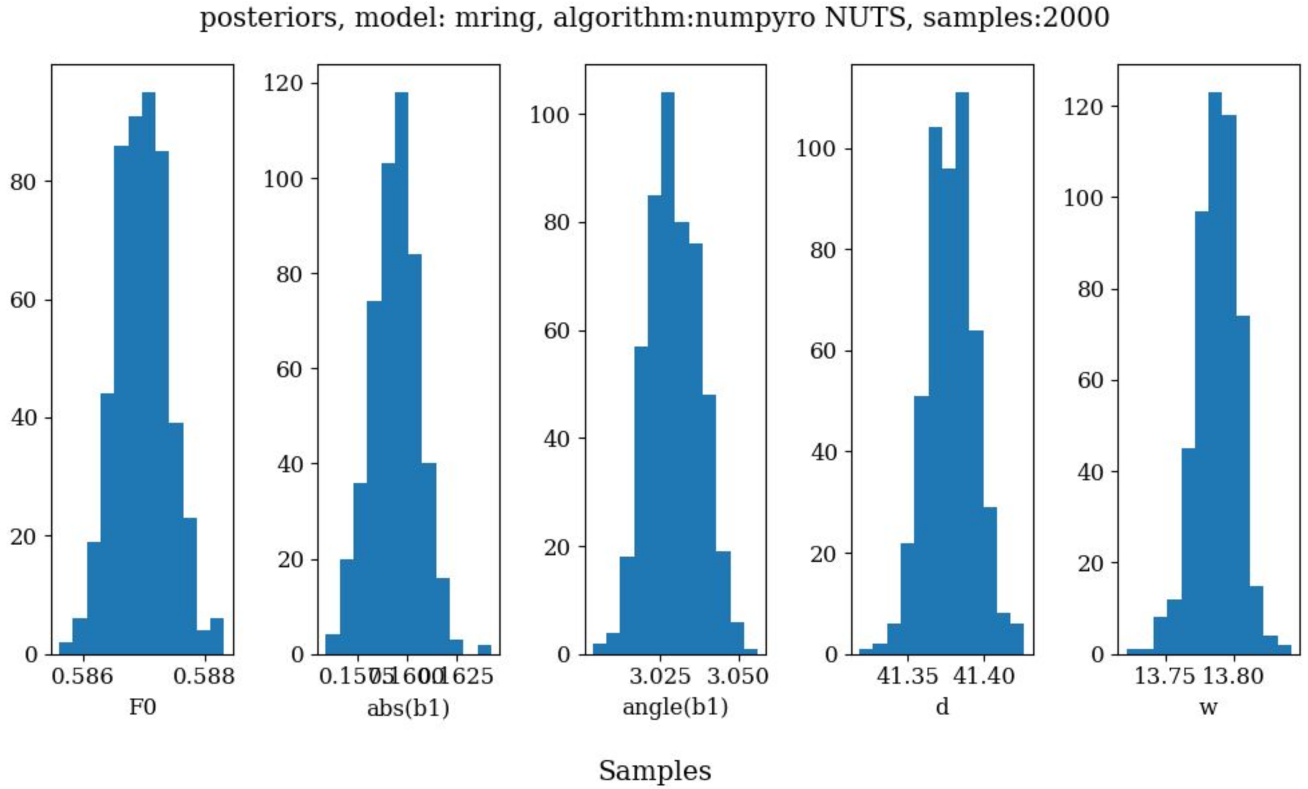


	mean	std	median	5.0%	95.0%	n_eff	r_hat
F0	0.59	0.00	0.59	0.59	0.59	998.75	1.00
abs(b1)	0.16	0.00	0.16	0.16	0.16	569.52	1.00
angle(b1)	3.03	0.01	3.03	3.02	3.04	587.46	1.00
d	41.38	0.02	41.38	41.35	41.40	2288.56	1.00
w	13.79	0.02	13.79	13.76	13.81	662.17	1.00

Model fitting M87* 2017 April 11 Stokes I visibility amplitudes and closure phases to an m-ring model (m=1)



Model fitting M87* 2017 April 11 Stokes I visibility amplitudes and closure phases to an m-ring model (m=1)



Running time comparison of CPU vs GPU model fitting

VM: eht-gpu-test

CPU: Intel Xeon @ 2.2 GHz

GPU: Nvidia Tesla P4

CPU model fitting time:

GPU model fitting time:

Running time comparison of CPU vs GPU model fitting

VM: eht-gpu-test

CPU: Intel Xeon @ 2.2 GHz

GPU: Nvidia Tesla P4

CPU model fitting time: 06:42:50

GPU model fitting time: 00:02:23

Running time comparison of CPU vs GPU model fitting

VM: eht-gpu-test

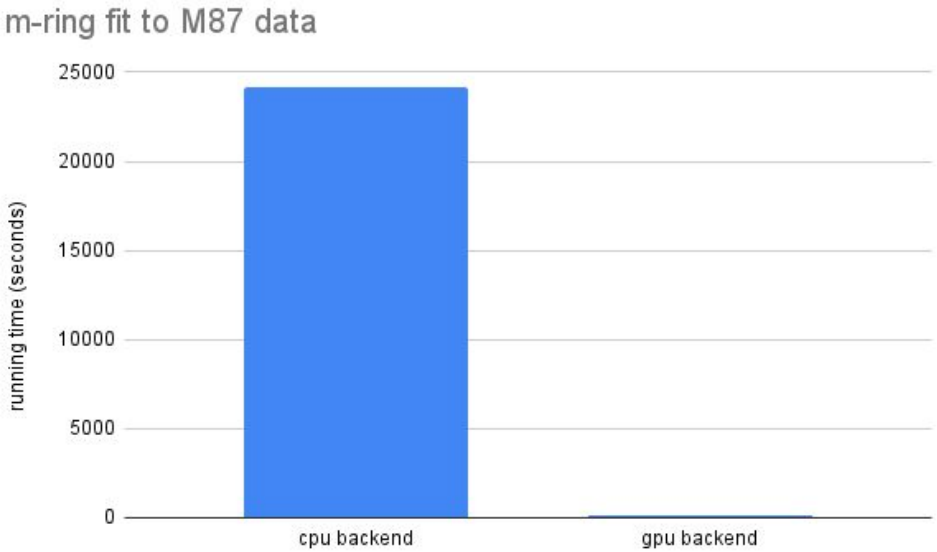
CPU: Intel Xeon @ 2.2 GHz

GPU: Nvidia Tesla P4

CPU model fitting time: 06:42:50

GPU model fitting time: 00:02:23

~170x speedup?



Running time comparison of CPU vs GPU model fitting

VM: eht-gpu-test

CPU: Intel Xeon @ 2.2 GHz

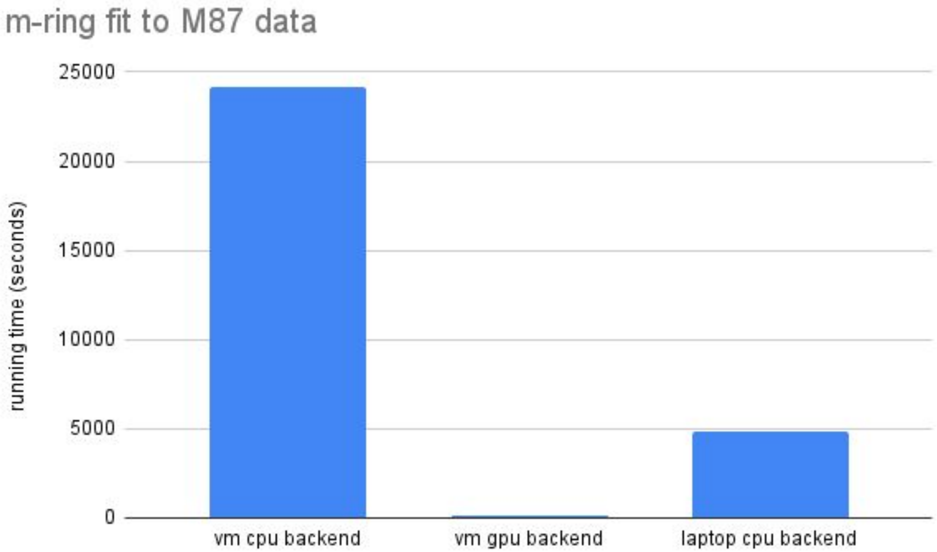
GPU: Nvidia Tesla P4

CPU model fitting time: 06:42:50

GPU model fitting time: 00:02:23

laptop model fitting time: 01:21:11

~34x speedup



Outlook

Is JAX appropriate for VLBI analysis? What problems does it solve, and where will we likely run into difficulties?

Complex numbers? More special functions? Third party library calls? Custom C/CUDA calls?

How well does this integrate into existing libraries such as eht-imaging?

JAX or Julia?

Code available at:

https://github.com/saopeter/bayesian_ring_inference

(Work in progress!)

Included jaxperiments.ipynb, open in Google Colab and play around with JAX without having to install on a local machine!