

The scientific ecosystem

'Autodifferentiable GPU-capable scipy'



Patrick Kidger



Cradle.bio

This ecosystem:

- ~4 years old.
- Now fairly mature.

This ecosystem:

- ~4 years old.
- Now fairly mature.

Goals for this talk:

- Give you a new tool.
- Geek out about scientific computing.

Background

Libraries

Case studies

Background

Libraries

Case studies

Numerical Python

Numerical Python

NumPy: array-based programming model

Numerical Python

NumPy: array-based programming model

SciPy: scientific computing operations

Numerical Python

[NumPy](#): array-based programming model

[SciPy](#): scientific computing operations

[CuPy](#): GPU-backed arrays

Numerical Python

[NumPy](#): array-based programming model

[SciPy](#): scientific computing operations

[CuPy](#): GPU-backed arrays

[Numba](#): JIT compilation

Numerical Python

[NumPy](#): array-based programming model

[SciPy](#): scientific computing operations

[CuPy](#): GPU-backed arrays

[Numba](#): JIT compilation

[Autograd](#): autodifferentiation

Numerical Python

[NumPy](#): array-based programming model

[SciPy](#): scientific computing operations

[CuPy](#): GPU-backed arrays

[Numba](#): JIT compilation

[Autograd](#): autodifferentiation

[TensorFlow](#): machine learning operations

Numerical Python

[NumPy](#): array-based programming model

[SciPy](#): scientific computing operations

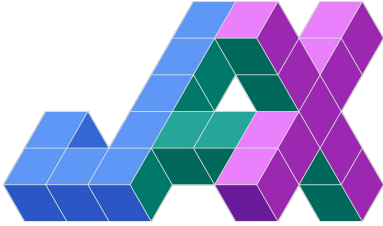
[CuPy](#): GPU-backed arrays

[Numba](#): JIT compilation

[Autograd](#): autodifferentiation

[TensorFlow](#): machine learning operations

[PyTorch](#): eager execution; model-building syntax



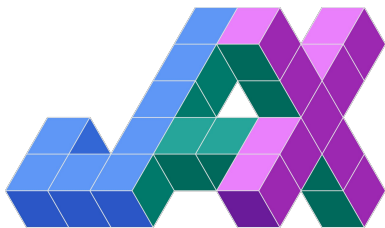
Familiar API

JAX provides a familiar NumPy-style API.

```
import jax.numpy as jnp

def forward(params, inputs):
    for W, b in params:
        outputs = inputs @ W + b
        inputs = jnp.maximum(outputs, 0)
    return outputs

def loss(params, x, y):
    predicted_y = forward(params, x)
    return jnp.sum((predicted_y - y) ** 2)
```



Familiar API

JAX provides a familiar NumPy-style API.

Transformations

JAX includes composable function transformations for compilation, batching, autodiff, and autoparallel.

```
import jax.numpy as jnp
from jax import grad

def forward(params, inputs):
    ...

def loss(params, x, y):
    ...

grad_fn = grad(loss)
```

```
import jax.numpy as jnp
from jax import jit, grad

def forward(params, inputs):
    ...

def loss(params, x, y):
    ...

grad_fn = jit(grad(loss))
```

```
import jax.numpy as jnp
from jax import jit, grad

def forward(params, inputs):
    ...

def loss(params, x, y):
    ...

grad_fn = jit(grad(loss))
```

```
import jax.numpy as jnp
from jax import jit, grad

def forward(params, inputs):
    ...

def loss(params, x, y):
    ...

grad_fn = jit(grad(loss))
```

`jit`: compilation + autoparallelism

```
import jax.numpy as jnp
from jax import jit, grad

def forward(params, inputs):
    ...

def loss(params, x, y):
    ...

grad_fn = jit(grad(loss))
```

`jit`: compilation + autoparallelism

`grad`: reverse-mode autodifferentiation

```
import jax.numpy as jnp
from jax import jit, grad

def forward(params, inputs):
    ...

def loss(params, x, y):
    ...

grad_fn = jit(grad(loss))
```

`jit`: compilation + autoparallelism

`grad`: reverse-mode autodifferentiation

`jvp`: forward-mode autodifferentiation

```
import jax.numpy as jnp
from jax import jit, grad

def forward(params, inputs):
    ...

def loss(params, x, y):
    ...

grad_fn = jit(grad(loss))
```

`jit`: compilation + autoparallelism

`grad`: reverse-mode autodifferentiation

`jvp`: forward-mode autodifferentiation

`vmap`: autobatching

```
import jax.numpy as jnp
from jax import jit, grad

def forward(params, inputs):
    ...

def loss(params, x, y):
    ...

grad_fn = jit(grad(loss))
```

`jit`: compilation + autoparallelism

`grad`: reverse-mode autodifferentiation

`jvp`: forward-mode autodifferentiation

`vmap`: autobatching

`jacfwd (=vmap+jvp)`: jacobians

```
import jax.numpy as jnp
from jax import jit, grad

def forward(params, inputs):
    ...

def loss(params, x, y):
    ...

grad_fn = jit(grad(loss))
```

`jit`: compilation + autoparallelism

`grad`: reverse-mode autodifferentiation

`jvp`: forward-mode autodifferentiation

`vmap`: autobatching

`jacfwd (=vmap+jvp)`: jacobians

`jacrev (=vmap+grad)`: jacobians again!

```
import jax.numpy as jnp
from jax import jit, grad

def forward(params, inputs):
    ...

def loss(params, x, y):
    ...

grad_fn = jit(grad(loss))
```

`jit`: compilation + autoparallelism

`grad`: reverse-mode autodifferentiation

`jvp`: forward-mode autodifferentiation

`vmap`: autobatching

`jacfwd (=vmap+jvp)`: jacobians

`jacrev (=vmap+grad)`: jacobians again!

+more advanced stuff:

```
import jax.numpy as jnp
from jax import jit, grad

def forward(params, inputs):
    ...

def loss(params, x, y):
    ...

grad_fn = jit(grad(loss))
```

`jit`: compilation + autoparallelism

`grad`: reverse-mode autodifferentiation

`jvp`: forward-mode autodifferentiation

`vmap`: autobatching

`jacfwd (=vmap+jvp)`: jacobians

`jacrev (=vmap+grad)`: jacobians again!

+more advanced stuff:

`linearize`,

```
import jax.numpy as jnp
from jax import jit, grad

def forward(params, inputs):
    ...

def loss(params, x, y):
    ...

grad_fn = jit(grad(loss))
```

`jit`: compilation + autoparallelism

`grad`: reverse-mode autodifferentiation

`jvp`: forward-mode autodifferentiation

`vmap`: autobatching

`jacfwd (=vmap+jvp)`: jacobians

`jacrev (=vmap+grad)`: jacobians again!

+more advanced stuff:

`linearize`,

`linear_transpose`,

```
import jax.numpy as jnp
from jax import jit, grad

def forward(params, inputs):
    ...

def loss(params, x, y):
    ...

grad_fn = jit(grad(loss))
```

`jit`: compilation + autoparallelism

`grad`: reverse-mode autodifferentiation

`jvp`: forward-mode autodifferentiation

`vmap`: autobatching

`jacfwd (=vmap+jvp)`: jacobians

`jacrev (=vmap+grad)`: jacobians again!

+more advanced stuff:

`linearize`,

`linear_transpose`,

...



Familiar API

JAX provides a familiar NumPy-style API.

Transformations

JAX includes composable function transformations for compilation, batching, autodiff, and autoparallel.

Run Anywhere

The same code executes on multiple backends, including CPU, GPU, & TPU

```
import jax.numpy as jnp
from jax import jit, grad

def forward(params, inputs):
    ...

def loss(params, x, y):
    ...

grad_fn = jit(grad(loss))
```

```
import jax.numpy as jnp
from jax import jit, grad

def forward(params, inputs):
    ...

def loss(params, x, y):
    ...

grad_fn = jit(grad(loss))

grads = grad_fn(params, x, y)
```

```
import jax.numpy as jnp
from jax import jit, grad
```

```
def forward(params, inputs):
```

```
    ...
```

```
def loss(params, x, y):
```

```
    ...
```

```
grad_fn = jit(grad(loss))
```

```
grads = grad_fn(params, x, y)
```



Runs on a single CPU

```
import jax.numpy as jnp
from jax import jit, grad

def forward(params, inputs):
    ...

def loss(params, x, y):
    ...

grad_fn = jit(grad(loss))
sharding = jax.sharding.NamedSharding(device_mesh)
x, y = jax.device_put((x, y), sharding)
grads = grad_fn(params, x, y) ← Runs on a single CPU
```

```
import jax.numpy as jnp
from jax import jit, grad
```

```
def forward(params, inputs):
```

```
    ...
```

```
def loss(params, x, y):
```

```
    ...
```

```
grad_fn = jit(grad(loss))
```

```
sharding = jax.sharding.NamedSharding(device_mesh)
```

```
x, y = jax.device_put((x, y), sharding)
```

```
grads = grad_fn(params, x, y)
```

Runs on a cluster of 32 GPUs

Runs on a single CPU



```
import jax.numpy as jnp
from jax import jit, grad
```

```
def forward(params, inputs):
```

```
    ...
```

```
def loss(params, x, y):
```

```
    ...
```

```
grad_fn = jit(grad(loss))
```

```
sharding = jax.sharding.NamedSharding(device_mesh)
```

```
x, y = jax.device_put((x, y), sharding)
```

```
grads = grad_fn(params, x, y)
```

Runs on a cluster of ~~32~~ GPUs

Runs on a single CPU

...or more!



Why

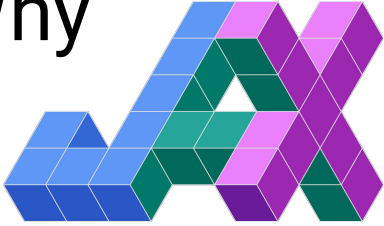


Why



How does it compare to...

Why

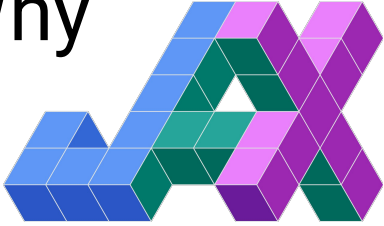


?

How does it compare to...

PyTorch?

Why



?

How does it compare to...

PyTorch?

JAX is:

Why



How does it compare to...

PyTorch?

JAX is:

- **Much** faster for scientific computing.

Why



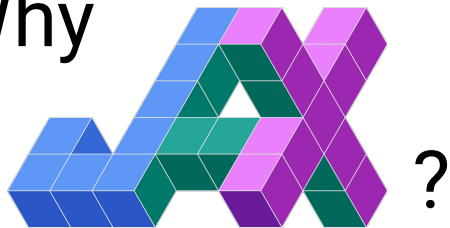
How does it compare to...

PyTorch?

JAX is:

- *Much* faster for scientific computing.
- Advanced features:

Why



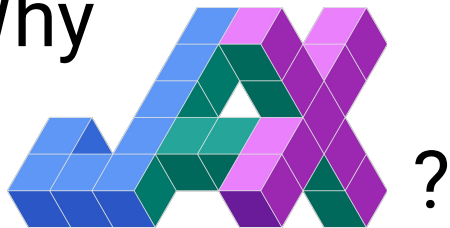
How does it compare to...

PyTorch?

JAX is:

- *Much* faster for scientific computing.
- Advanced features:
 - forward-mode autodiff

Why



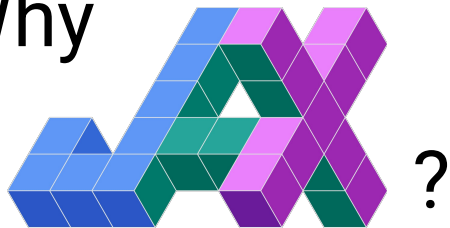
How does it compare to...

PyTorch?

JAX is:

- *Much* faster for scientific computing.
- Advanced features:
 - forward-mode autodiff
 - vmap

Why



How does it compare to...

PyTorch?

JAX is:

- *Much* faster for scientific computing.
- Advanced features:
 - forward-mode autodiff
 - vmap
 - scientific ecosystem

Why



How does it compare to...

PyTorch?

JAX is:

- *Much* faster for scientific computing.
- Advanced features:
 - forward-mode autodiff
 - vmap
 - scientific ecosystem
- Trade-off: trickier to use.

Why



How does it compare to...

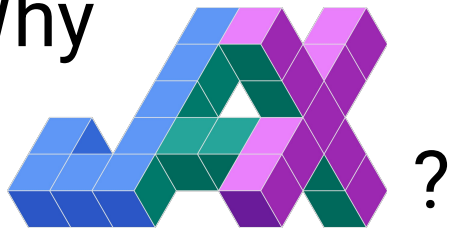
PyTorch?

JAX is:

- *Much* faster for scientific computing.
- Advanced features:
 - forward-mode autodiff
 - vmap
 - scientific ecosystem
- Trade-off: trickier to use.

Julia?

Why



How does it compare to...

PyTorch?

JAX is:

- *Much* faster for scientific computing.
- Advanced features:
 - forward-mode autodiff
 - vmap
 - scientific ecosystem
- Trade-off: trickier to use.

Julia?

JAX has:

Why



?

How does it compare to...

PyTorch?

JAX is:

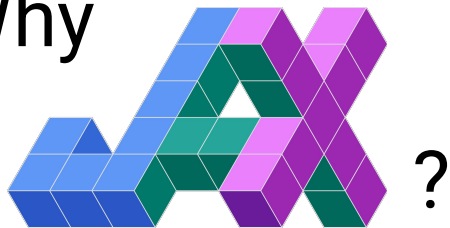
- *Much* faster for scientific computing.
- Advanced features:
 - forward-mode autodiff
 - vmap
 - scientific ecosystem
- Trade-off: trickier to use.

Julia?

JAX has:

- Fewer correctness issues.

Why



How does it compare to...

PyTorch?

JAX is:

- *Much* faster for scientific computing.
- Advanced features:
 - forward-mode autodiff
 - vmap
 - scientific ecosystem
- Trade-off: trickier to use.

Julia?

JAX has:

- Fewer correctness issues.
- More mature autodiff.

Why



How does it compare to...

PyTorch?

JAX is:

- *Much* faster for scientific computing.
- Advanced features:
 - forward-mode autodiff
 - vmap
 - scientific ecosystem
- Trade-off: trickier to use.

Julia?

JAX has:

- Fewer correctness issues.
- More mature autodiff.
- Large-scale autoparallel.

Why



How does it compare to...

PyTorch?

JAX is:

- *Much* faster for scientific computing.
- Advanced features:
 - forward-mode autodiff
 - vmap
 - scientific ecosystem
- Trade-off: trickier to use.

Julia?

JAX has:

- Fewer correctness issues.
- More mature autodiff.
- Large-scale autoparallel.
- Trade-off: fewer niche features.

Why



How does it compare to...

PyTorch?

JAX is:

- *Much* faster for scientific computing.
- Advanced features:
 - forward-mode autodiff
 - vmap
 - scientific ecosystem
- Trade-off: trickier to use.

Julia?

JAX has:

- Fewer correctness issues.
- More mature autodiff.
- Large-scale autoparallel.
- Trade-off: fewer niche features.

Notably used by:

MIT.

Why



How does it compare to...

PyTorch?

JAX is:

- *Much* faster for scientific computing.
- Advanced features:
 - forward-mode autodiff
 - vmap
 - scientific ecosystem
- Trade-off: trickier to use.

Notably used by:

OpenAI
About 90% of academia.

Julia?

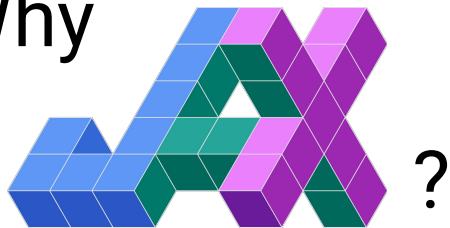
JAX has:

- Fewer correctness issues.
- More mature autodiff.
- Large-scale autoparallel.
- Trade-off: fewer niche features.

Notably used by:

MIT.

Why



Notably used by:

Google/DeepMind;
Many (most?) frontier labs;
Astrophysicists.

How does it compare to...

PyTorch?

JAX is:

- *Much* faster for scientific computing.
- Advanced features:
 - forward-mode autodiff
 - vmap
 - scientific ecosystem
- Trade-off: trickier to use.

Notably used by:

OpenAI
About 90% of academia.

Julia?

JAX has:

- Fewer correctness issues.
- More mature autodiff.
- Large-scale autoparallel.
- Trade-off: fewer niche features.

Notably used by:

MIT.

Background

Libraries

Case studies

Diffrax

Difffrax

ODE + SDE (+ some PDE) solvers.

Difffrax

ODE + SDE (+ some PDE) solvers.

Let's take advantage of first-class autodiff!

Difffrax

ODE + SDE (+ some PDE) solvers.

Let's take advantage of first-class autodiff!

```
from difffrax import diffeqsolve, ODETerm, Kvaerno5
```

Difffrax

ODE + SDE (+ some PDE) solvers.

Let's take advantage of first-class autodiff!

```
from difffrax import diffeqsolve, ODETerm, Kvaerno5
import jax.numpy as jnp
```

Difffrax

ODE + SDE (+ some PDE) solvers.

Let's take advantage of first-class autodiff!

```
from difffrax import diffeqsolve, ODETerm, Kvaerno5
import jax.numpy as jnp

term = ODETerm(...)
```

Difffrax

ODE + SDE (+ some PDE) solvers.

Let's take advantage of first-class autodiff!

```
from difffrax import diffeqsolve, ODETerm, Kvaerno5
import jax.numpy as jnp

term = ODETerm(...)
solver = Kvaerno5() # implicit solver; Jacobian found using autodiff!
```

Difffrax

ODE + SDE (+ some PDE) solvers.

Let's take advantage of first-class autodiff!

```
from difffrax import diffeqsolve, ODETerm, Kvaerno5
import jax.numpy as jnp

term = ODETerm(...)
solver = Kvaerno5() # implicit solver; Jacobian found using autodiff!
y0 = jnp.array(...)
```

Difffrax

ODE + SDE (+ some PDE) solvers.

Let's take advantage of first-class autodiff!

```
from difffrax import diffeqsolve, ODETerm, Kvaerno5
import jax.numpy as jnp

term = ODETerm(...)
solver = Kvaerno5() # implicit solver; Jacobian found using autodiff!
y0 = jnp.array(...)

solution = diffeqsolve(term, solver, t0=0, t1=1, dt0=0.1, y0=y0)
```

Diffrax vs ...

Diffrax vs ...

...SciPy?

Diffrax vs ...

...SciPy?

- Much faster (40x)

Diffrax vs ...

...SciPy?

- Much faster (40x)
- Autodifferentiation / GPU support

Diffrax vs ...

...SciPy?

- Much faster (40x)
- Autodifferentiation / GPU support
- Loads more features

Diffrax vs ...

...SciPy?

- Much faster (40x)
- Autodifferentiation / GPU support
- Loads more features

...torchdiffeq?

Difffrax vs ...

...SciPy?

- Much faster (40x)
- Autodifferentiation / GPU support
- Loads more features

...torchdiffeq?

- Much faster (200x – b/c PyTorch vs JAX)

Diffrax vs ...

...SciPy?

- Much faster (40x)
- Autodifferentiation / GPU support
- Loads more features

...torchdiffeq?

- Much faster (200x – b/c PyTorch vs JAX)
- Loads more features

Diffrax vs ...

...SciPy?

- Much faster (40x)
- Autodifferentiation / GPU support
- Loads more features

...torchdiffeq?

- Much faster (200x – b/c PyTorch vs JAX)
- Loads more features

...DifferentialEquations.jl?

Diffrax vs ...

...SciPy?

- Much faster (40x)
- Autodifferentiation / GPU support
- Loads more features

...torchdiffeq?

- Much faster (200x – b/c PyTorch vs JAX)
- Loads more features

...DifferentialEquations.jl?

- Python, NumPy

Diffrax vs ...

...SciPy?

- Much faster (40x)
- Autodifferentiation / GPU support
- Loads more features

...torchdiffeq?

- Much faster (200x – b/c PyTorch vs JAX)
- Loads more features

...DifferentialEquations.jl?

- Python, NumPy
- Reliable autodiff, autoparallel

Diffrax vs ...

...SciPy?

- Much faster (40x)
- Autodifferentiation / GPU support
- Loads more features

...torchdiffeq?

- Much faster (200x – b/c PyTorch vs JAX)
- Loads more features

...DifferentialEquations.jl?

- Python, NumPy
- Reliable autodiff, autoparallel
- Same speed

Diffrax vs ...

...SciPy?

- Much faster (40x)
- Autodifferentiation / GPU support
- Loads more features

...torchdiffeq?

- Much faster (200x – b/c PyTorch vs JAX)
- Loads more features

...DifferentialEquations.jl?

- Python, NumPy
- Reliable autodiff, autoparallel
- Same speed
- (But a few less solvers!)

Diffrax vs ...

...SciPy?

- Much faster (40x)
- Autodifferentiation / GPU support
- Loads more features

...torchdiffeq?

- Much faster (200x – b/c PyTorch vs JAX)
- Loads more features

...DifferentialEquations.jl?

- Python, NumPy
- Reliable autodiff, autoparallel
- Same speed
- (But a few less solvers!)

1. ODE + SDE (+ some PDE) solvers;
 - DDE and DAE solvers?
2. High order, implicit, IMEX, symplectic;
3. Save at time points; dense output;
4. Multiple methods for backpropagation; e.g. Griewank-style recursive checkpointing.
5. Adaptive step size controllers (PID etc.)
6. Event handling; discontinuities;
7. Distributed computing; GPU/TPU support;
8. ...
9. etc!

Diffrax vs ...

...SciPy?

- Much faster (40x)
- Autodifferentiation / GPU support
- Loads more features

...torchdiffeq?

- Much faster (200x – b/c PyTorch vs JAX)
- Loads more features

...DifferentialEquations.jl?

- Python, NumPy
- Reliable autodiff, autoparallel
- Same speed
- (But a few less solvers!)

1. ODE + SDE (+ some PDE) solvers;
 - DDE and DAE solvers?
2. High order, implicit, IMEX, symplectic;
3. Save at time points; dense output;
4. Multiple methods for backpropagation; e.g. Griewank-style recursive checkpointing.
5. Adaptive step size controllers (PID etc.)
6. Event handling; discontinuities;
7. Distributed computing; GPU/TPU support;
8. ...
9. etc!

Diffrax also introduces an approach to unifying diffeq numerics, via rough path theory. “SDDAEs” anyone?!

jaxtyping

jaxtyping

JIT-time shape-checks. Uses a mini-DSL to describe shapes (anonymous, variadic, symbolic, fixed, etc.)

jaxtyping

JIT-time shape-checks. Uses a mini-DSL to describe shapes (anonymous, variadic, symbolic, fixed, etc.)

```
def fn(x: np.ndarray, y: np.ndarray):
```

jaxtyping

JIT-time shape-checks. Uses a mini-DSL to describe shapes (anonymous, variadic, symbolic, fixed, etc.)

```
def fn(x: np.ndarray, y: np.ndarray):  
    # x and y must have the same shape  
    ...
```

jaxtyping

JIT-time shape-checks. Uses a mini-DSL to describe shapes (anonymous, variadic, symbolic, fixed, etc.)

```
def fn(x: np.ndarray, y: np.ndarray):  
    # x and y must have the same shape  
    ...  
  
from jaxtyping import Array, Float
```

jaxtyping

JIT-time shape-checks. Uses a mini-DSL to describe shapes (anonymous, variadic, symbolic, fixed, etc.)

```
def fn(x: np.ndarray, y: np.ndarray):  
    # x and y must have the same shape  
    ...  
  
from jaxtyping import Array, Float  
  
def batch_outer_product(  

```

jaxtyping

JIT-time shape-checks. Uses a mini-DSL to describe shapes (anonymous, variadic, symbolic, fixed, etc.)

```
def fn(x: np.ndarray, y: np.ndarray):  
    # x and y must have the same shape  
    ...  
  
from jaxtyping import Array, Float  
  
def batch_outer_product(x: Float[Array, "*batch dim_x"],
```

jaxtyping

JIT-time shape-checks. Uses a mini-DSL to describe shapes (anonymous, variadic, symbolic, fixed, etc.)

```
def fn(x: np.ndarray, y: np.ndarray):  
    # x and y must have the same shape  
    ...  
  
from jaxtyping import Array, Float  
  
def batch_outer_product(x: Float[Array, "*batch dim_x"],  
                        y: Float[Array, "*batch dim_y"]
```

jaxtyping

JIT-time shape-checks. Uses a mini-DSL to describe shapes (anonymous, variadic, symbolic, fixed, etc.)

```
def fn(x: np.ndarray, y: np.ndarray):  
    # x and y must have the same shape  
    ...  
  
from jaxtyping import Array, Float  
  
def batch_outer_product(x: Float[Array, "*batch dim_x"],  
                        y: Float[Array, "*batch dim_y"]  
                        ) -> Float[Array, "*batch dim_x dim_y"]:  
    ...
```

jaxtyping

JIT-time shape-checks. Uses a mini-DSL to describe shapes (anonymous, variadic, symbolic, fixed, etc.)

```
def fn(x: np.ndarray, y: np.ndarray):  
    # x and y must have the same shape  
    ...  
  
from jaxtyping import Array, Float  
  
def batch_outer_product(x: Float[Array, "*batch dim_x"],  
                        y: Float[Array, "*batch dim_y"]  
                        ) -> Float[Array, "*batch dim_x dim_y"]:  
    ...
```

Lineax

```
import lineax as lx
op = lx.JacobianLinearOperator(...)
vector = ...
solution = linear_solve(op, vector, lx.QR())
```

Lineax

```
import lineax as lx
op = lx.JacobianLinearOperator(...)
vector = ...
solution = linear_solve(op, vector, lx.QR())
```

Optimistix

```
import optimistix as optx

class HybridMinimiser(optx.AbstractBFGS):
    line_search = optx.ClassicalTrustRegion(optx.DoglegDescent(...))

optx.minimise(f, HybridMinimiser(...), adjoint=optx.ImplicitAdjoint())
```

Lineax

```
import lineax as lx
op = lx.JacobianLinearOperator(...)
vector = ...
solution = linear_solve(op, vector, lx.QR())
```

Equinox

```
import equinox as eqx

class Model(eqx.Module):
    weight: jax.Array
    bias: jax.Array

    def __call__(self, x):
        return self.weight @ x
        + self.bias
```

```
import optimistix as optx

class HybridMinimiser(optx.AbstractBFGS):
    line_search = optx.ClassicalTrustRegion(optx.DoglegDescent(...))

optx.minimise(f, HybridMinimiser(...), adjoint=optx.ImplicitAdjoint())
```

Optimistix

Lineax

```
import lineax as lx
op = lx.JacobianLinearOperator(...)
vector = ...
solution = linear_solve(op, vector, lx.QR())
```

Optax

```
import optax
optim = optax.adam(3e-4)
```

Optimistix

Equinox

```
import equinox as eqx

class Model(eqx.Module):
    weight: jax.Array
    bias: jax.Array

    def __call__(self, x):
        return self.weight @ x
        + self.bias
```

```
import optimistix as optx

class HybridMinimiser(optx.AbstractBFGS):
    line_search = optx.ClassicalTrustRegion(optx.DoglegDescent(...))

optx.minimise(f, HybridMinimiser(...), adjoint=optx.ImplicitAdjoint())
```

Lineax

```
import lineax as lx
op = lx.JacobianLinearOperator(...)
vector = ...
solution = linear_solve(op, vector, lx.QR())
```

Optax

```
import optax
optim = optax.adam(3e-4)
```

Optimistix

Equinox

```
import equinox as eqx

class Model(eqx.Module):
    weight: jax.Array
    bias: jax.Array

    def __call__(self, x):
        return self.weight @ x
        + self.bias
```

```
import optimistix as optx

class HybridMinimiser(optx.AbstractBFGS):
    line_search = optx.ClassicalTrustRegion(optx.DoglegDescent(...))

optx.minimise(f, HybridMinimiser(...), adjoint=optx.ImplicitAdjoint())
```

Quax

```
import quax
import quax.examples.lora as lora
arr = lora.LoraArray(...)
vec = jnp.array(...)
quax.quaxify(lambda a, b: a @ b)(arr, vec)
```

Lineax

```
import lineax as lx
op = lx.JacobianLinearOperator(...)
vector = ...
solution = linear_solve(op, vector, lx.QR())
```

Optax

```
import optax
optim = optax.adam(3e-4)
```

Optimistix

Equinox

```
import equinox as eqx

class Model(eqx.Module):
    weight: jax.Array
    bias: jax.Array

    def __call__(self, x):
        return self.weight @ x
        + self.bias
```

```
import optimistix as optx

class HybridMinimiser(optx.AbstractBFGS):
    line_search = optx.ClassicalTrustRegion(optx.DoglegDescent(...))

optx.minimise(f, HybridMinimiser(...), adjoint=optx.ImplicitAdjoint())
```

Quax

```
import quax
import quax.examples.lora as lora
arr = lora.LoraArray(...)
vec = jnp.array(...)
quax.quaxify(lambda a, b: a @ b)(arr, vec)
```

Haliax & Levanter

```
import haliax as hax
Key = hax.Axis("position", 1024)

def attention(query, key, value):
    scores = hax.dot(Key, query, key) /
        jnp.sqrt(Key.size)
    scores = hax.nn.normalization.softmax(scores, Key)
    return hax.dot(Key, scores, value)
```

Lineax

```
import lineax as lx
op = lx.JacobianLinearOperator(...)
vector = ...
solution = linear_solve(op, vector, lx.QR())
```

Optax

```
import optax
optim = optax.adam(3e-4)
```

... !

Optimistix

Equinox

```
import equinox as eqx

class Model(eqx.Module):
    weight: jax.Array
    bias: jax.Array

    def __call__(self, x):
        return self.weight @ x
        + self.bias
```

```
import optimistix as optx

class HybridMinimiser(optx.AbstractBFGS):
    line_search = optx.ClassicalTrustRegion(optx.DoglegDescent(...))

optx.minimise(f, HybridMinimiser(...), adjoint=optx.ImplicitAdjoint())
```

Haliax & Levanter

```
import haliax as hax
Key = hax.Axis("position", 1024)

def attention(query, key, value):
    scores = hax.dot(Key, query, key) /
        jnp.sqrt(Key.size)
    scores = hax.nn.normalization.softmax(scores, Key)
    return hax.dot(Key, scores, value)
```

Quax

```
import quax
import quax.examples.lora as lora
arr = lora.LoraArray(...)
vec = jnp.array(...)
quax.quaxify(lambda a, b: a @ b)(arr, vec)
```

Background

Libraries

Case studies

Case study: Equinox + Diffrax + jaxtyping for neural ODEs!

Case study: Equinox + Diffrax + jaxtyping for neural ODEs!

```
import diffrax as dfx
import equinox as eqx
import jax
from jaxtyping import Array, Float
```

Case study: Equinox + Diffrax + jaxtyping for neural ODEs!

```
import diffrax as dfr
import equinox as eqx
import jax
from jaxtyping import Array, Float

mlp = eqx.nn.MLP(in_size=4, out_size=4, width_size=32, depth=1,
                 key=jax.random.key(0))
```

Case study: Equinox + Diffrax + jaxtyping for neural ODEs!

```
import diffrax as dfr
import equinox as eqx
import jax
from jaxtyping import Array, Float

mlp = eqx.nn.MLP(in_size=4, out_size=4, width_size=32, depth=1,
                 key=jax.random.key(0))

def solve_neural_ode(y0: Float[Array, "4"])
```

Case study: Equinox + Diffrax + jaxtyping for neural ODEs!

```
import diffrax as dfx
import equinox as eqx
import jax
from jaxtyping import Array, Float

mlp = eqx.nn.MLP(in_size=4, out_size=4, width_size=32, depth=1,
                 key=jax.random.key(0))

def solve_neural_ode(y0: Float[Array, "4"])
    sol = dfx.diffeqsolve(dfx.ODETerm(lambda t, y, args: mlp(y)),
```

Case study: Equinox + Diffrax + jaxtyping for neural ODEs!

```
import diffrax as dfx
import equinox as eqx
import jax
from jaxtyping import Array, Float

mlp = eqx.nn.MLP(in_size=4, out_size=4, width_size=32, depth=1,
                 key=jax.random.key(0))

def solve_neural_ode(y0: Float[Array, "4"])
    sol = dfx.diffeqsolve(dfx.ODETerm(lambda t, y, args: mlp(y)),
                          dfx.Tsit5(), t0=0, t1=1, dt0=0.1, y0=y0)
```

Case study: Equinox + Diffrax + jaxtyping for neural ODEs!

```
import diffrax as dfx
import equinox as eqx
import jax
from jaxtyping import Array, Float

mlp = eqx.nn.MLP(in_size=4, out_size=4, width_size=32, depth=1,
                key=jax.random.key(0))

def solve_neural_ode(y0: Float[Array, "4"])
    sol = dfx.diffeqsolve(dfx.ODETerm(lambda t, y, args: mlp(y)),
                        dfx.Tsit5(), t0=0, t1=1, dt0=0.1, y0=y0
                        saveat=dfx.SaveAt(ts=jax.numpy.linspace(0, 1, 10)))
```

Case study: Equinox + Diffrax + jaxtyping for neural ODEs!

```
import diffrax as dfx
import equinox as eqx
import jax
from jaxtyping import Array, Float

mlp = eqx.nn.MLP(in_size=4, out_size=4, width_size=32, depth=1,
                key=jax.random.key(0))

def solve_neural_ode(y0: Float[Array, "4"])
    sol = dfx.diffeqsolve(dfx.ODETerm(lambda t, y, args: mlp(y)),
                        dfx.Tsit5(), t0=0, t1=1, dt0=0.1, y0=y0
                        saveat=dfx.SaveAt(ts=jax.numpy.linspace(0, 1, 10)))

    return sol.ys
```

Case study: Equinox + Diffrax + jaxtyping for neural ODEs!

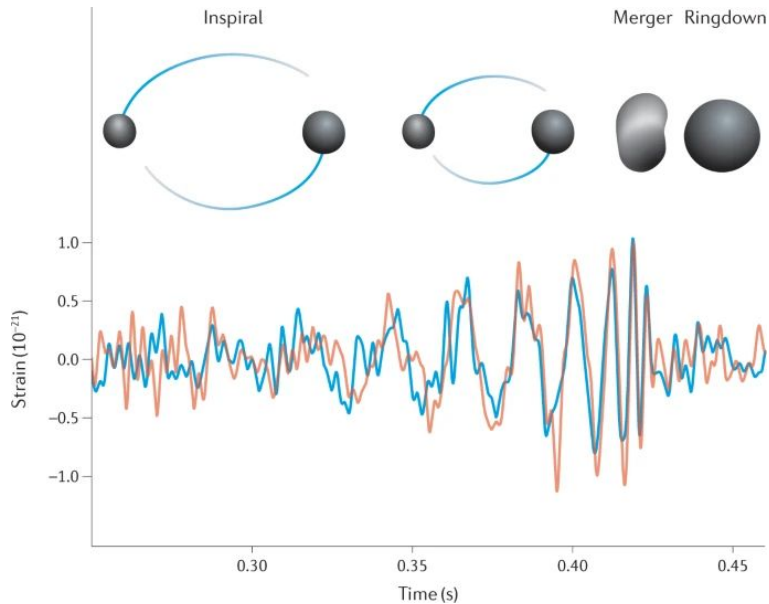
```
import diffrax as dfx
import equinox as eqx
import jax
from jaxtyping import Array, Float

mlp = eqx.nn.MLP(in_size=4, out_size=4, width_size=32, depth=1,
                key=jax.random.key(0))

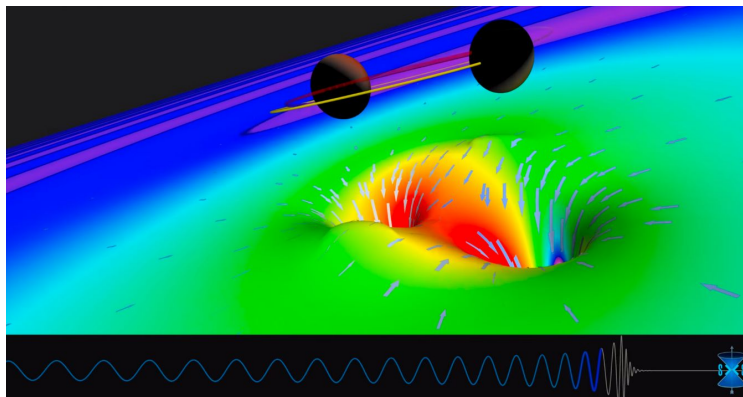
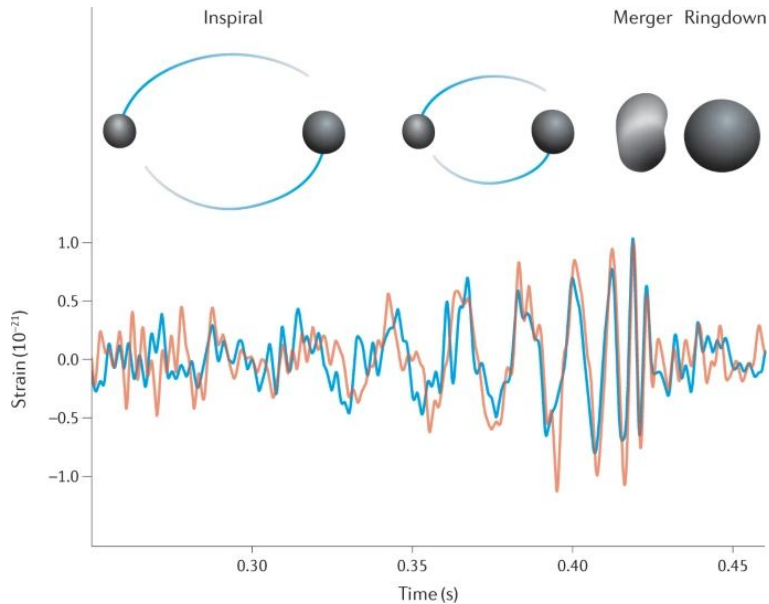
def solve_neural_ode(y0: Float[Array, "4"]) -> Float[Array, "10 4"]:
    sol = dfx.diffeqsolve(dfx.ODETerm(lambda t, y, args: mlp(y)),
                        dfx.Tsit5(), t0=0, t1=1, dt0=0.1, y0=y0,
                        saveat=dfx.SaveAt(ts=jax.numpy.linspace(0, 1, 10)))
    return sol.ys
```

Case study: Gravitational waveform modelling

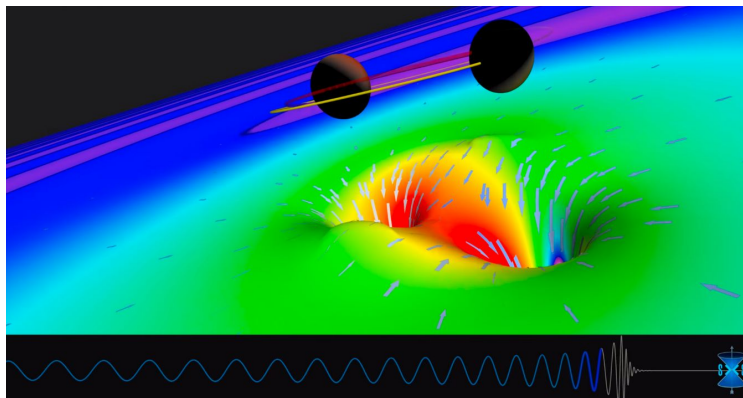
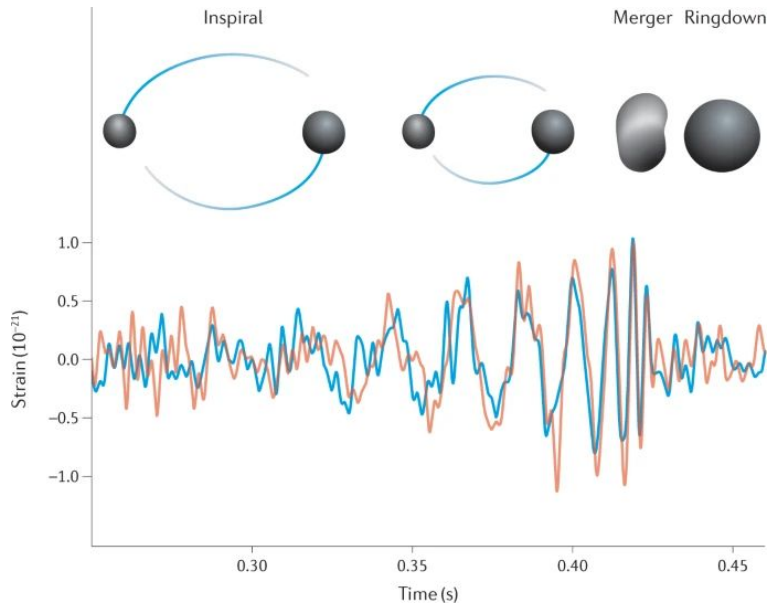
Case study: Gravitational waveform modelling



Case study: Gravitational waveform modelling

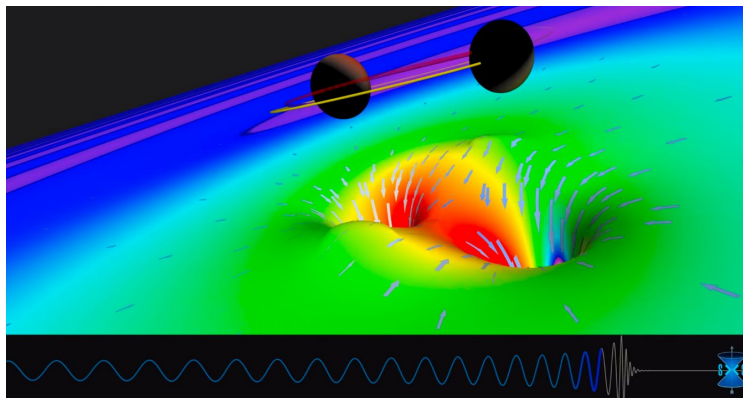
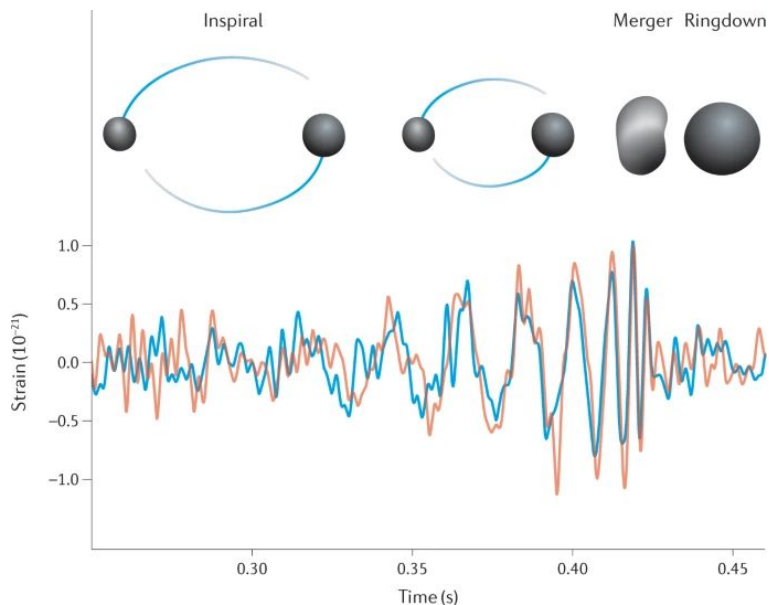


Case study: Gravitational waveform modelling



How they use JAX+Equinox+...:

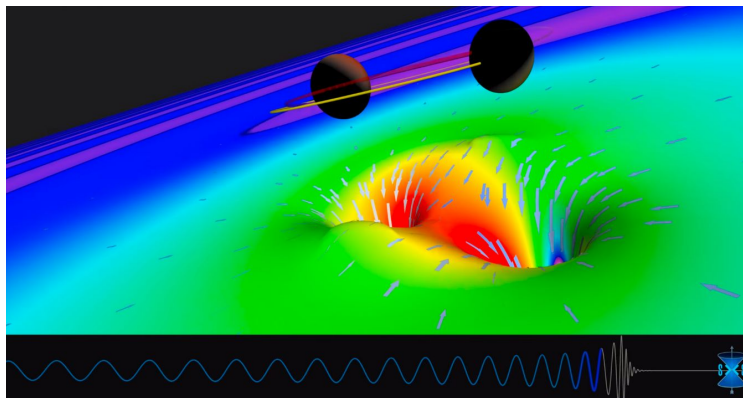
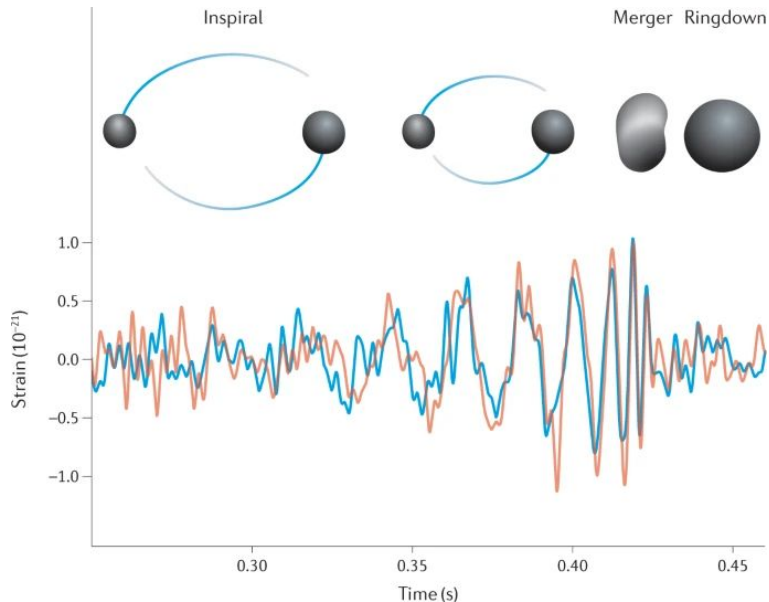
Case study: Gravitational waveform modelling



How they use JAX+Equinox+...:

- Uses ensembling to optimally leverage the GPUs.

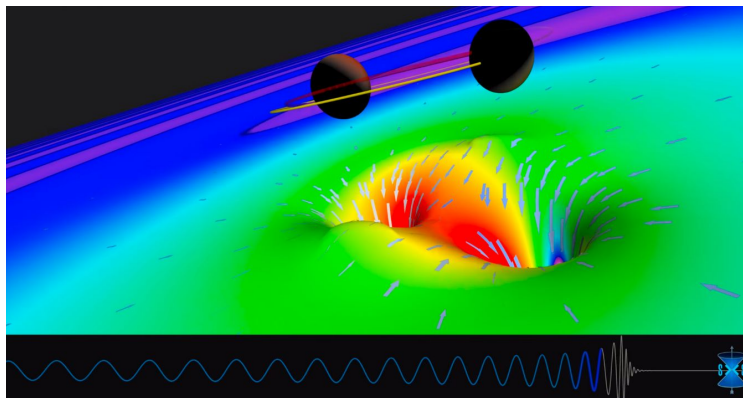
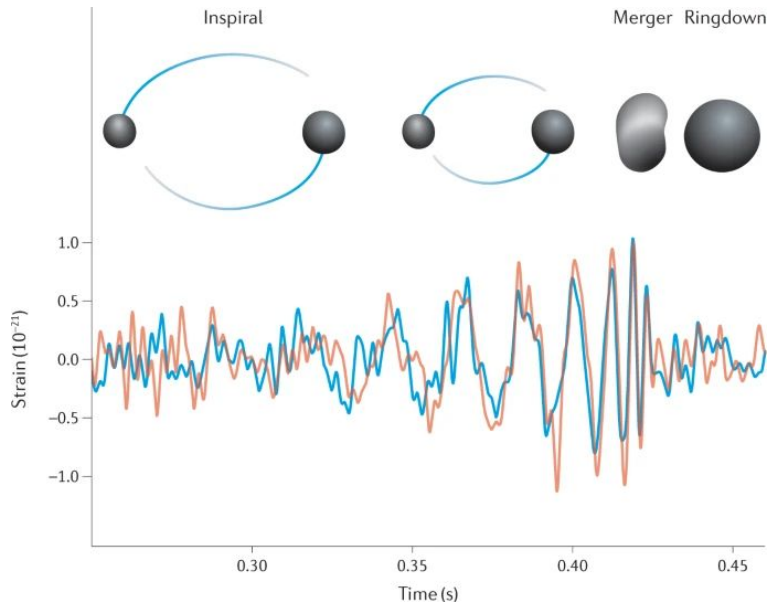
Case study: Gravitational waveform modelling



How they use JAX+Equinox+...:

- Uses ensembling to optimally leverage the GPUs.
 - 100-500x speedup on an A100; 200-2000x on a H100.

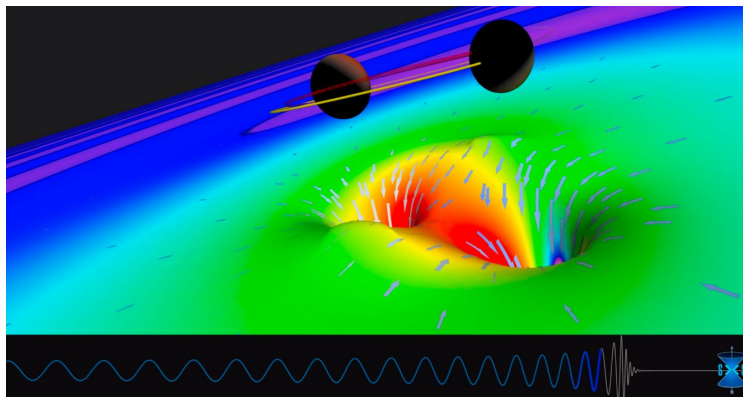
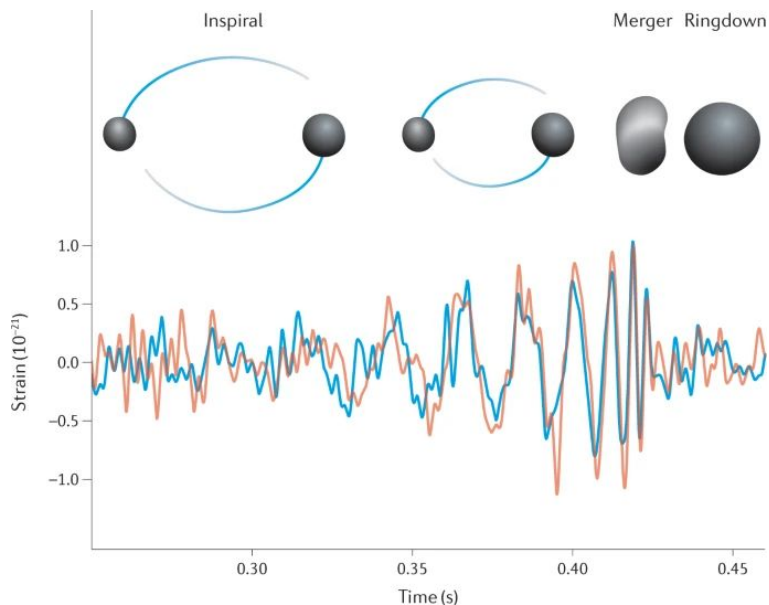
Case study: Gravitational waveform modelling



How they use JAX+Equinox+...:

- Uses ensembling to optimally leverage the GPUs.
 - 100-500x speedup on an A100; 200-2000x on a H100.
- Has a high performance spline using Lineax.

Case study: Gravitational waveform modelling



How they use JAX+Equinox+...:

- Uses ensembling to optimally leverage the GPUs.
 - 100-500x speedup on an A100; 200-2000x on a H100.
- Has a high performance spline using Lineax.

*“The non-JAX version takes ... anywhere from **2 hours to 10 days**.
Now with the JAX code, we are expecting to **deliver results in minutes**”*

If we have time: on software quality!

If we have time: on software quality!

For those starting out...



If we have time: on software quality!

For those starting out...

- Lint (=check for common errors) with `ruff`.

If we have time: on software quality!

For those starting out...

- Lint (=check for common errors) with `ruff`.
- Format with `ruff-format`.

If we have time: on software quality!

For those starting out...

- Lint (=check for common errors) with `ruff`.
- Format with `ruff-format`.
- Type-check with `pyright`.

If we have time: on software quality!

For those starting out...

- Lint (=check for common errors) with `ruff`.
- Format with `ruff-format`.
- Type-check with `pyright`.

and run them automatically with `pre-commit`.

If we have time: on software quality!

For those starting out...

- Lint (=check for common errors) with `ruff`.
- Format with `ruff-format`.
- Type-check with `pyright`.

and run them automatically with `pre-commit`.

(Feel free to steal my config from

<https://github.com/patrick-kidger/equinox/blob/main/.pre-commit-config.yaml>

)

If we have time: on software quality!

For those starting out...

- Lint (=check for common errors) with `ruff`.
- Format with `ruff-format`.
- Type-check with `pyright`.

and run them automatically with `pre-commit`.

(Feel free to steal my config from

<https://github.com/patrick-kidger/equinox/blob/main/.pre-commit-config.yaml>

)

Learn collaborative coding:

If we have time: on software quality!

For those starting out...

- Lint (=check for common errors) with `ruff`.
- Format with `ruff-format`.
- Type-check with `pyright`.

and run them automatically with `pre-commit`.

(Feel free to steal my config from

<https://github.com/patrick-kidger/equinox/blob/main/.pre-commit-config.yaml>

)

Learn collaborative coding:

- Pull requests + code review on GitHub.

If we have time: on software quality!

For those starting out...

- Lint (=check for common errors) with ruff.
- Format with ruff-format.
- Type-check with pyright.

and run them automatically with pre-commit.

(Feel free to steal my config from

<https://github.com/patrick-kidger/equinnox/blob/main/.pre-commit-config.yaml>

)

Learn collaborative coding:

- Pull requests + code review on GitHub.

For those already familiar:



If we have time: on software quality!

For those starting out...

- Lint (=check for common errors) with ruff.
- Format with ruff-format.
- Type-check with pyright.

and run them automatically with pre-commit.

(Feel free to steal my config from

<https://github.com/patrick-kidger/equinox/blob/main/.pre-commit-config.yaml>

)

Learn collaborative coding:

- Pull requests + code review on GitHub.

For those already familiar:

```
class AbstractSolver(eqx.Module):  
    @abstractmethod  
    def step(...): ...
```

If we have time: on software quality!

For those starting out...

- Lint (=check for common errors) with ruff.
- Format with ruff-format.
- Type-check with pyright.

and run them automatically with pre-commit.

(Feel free to steal my config from

<https://github.com/patrick-kidger/equinox/blob/main/.pre-commit-config.yaml>

)

Learn collaborative coding:

- Pull requests + code review on GitHub.

For those already familiar:

```
class AbstractSolver(equ.Module):
    @abstractmethod
    def step(...): ...

class AbstractRungeKutta(AbstractSolver):
    tableau: AbstractClassVar[ButcherTableau]
    def step(...): ...
```

If we have time: on software quality!

For those starting out...

- Lint (=check for common errors) with ruff.
- Format with ruff-format.
- Type-check with pyright.

and run them automatically with pre-commit.

(Feel free to steal my config from

<https://github.com/patrick-kidger/equinox/blob/main/.pre-commit-config.yaml>

)

Learn collaborative coding:

- Pull requests + code review on GitHub.

For those already familiar:

```
class AbstractSolver(eqx.Module):
    @abstractmethod
    def step(...): ...

class AbstractRungeKutta(AbstractSolver):
    tableau: AbstractClassVar[ButcherTableau]
    def step(...): ...
```

- Type safety (~traits) of Rust;

If we have time: on software quality!

For those starting out...

- Lint (=check for common errors) with ruff.
- Format with ruff-format.
- Type-check with pyright.

and run them automatically with pre-commit.

(Feel free to steal my config from

<https://github.com/patrick-kidger/equinox/blob/main/.pre-commit-config.yaml>

)

Learn collaborative coding:

- Pull requests + code review on GitHub.

For those already familiar:

```
class AbstractSolver(eqx.Module):
    @abstractmethod
    def step(...): ...

class AbstractRungeKutta(AbstractSolver):
    tableau: AbstractClassVar[ButcherTableau]
    def step(...): ...
```

- Type safety (~traits) of Rust;
- Abstract/final pattern of Julia;

If we have time: on software quality!

For those starting out...

- Lint (=check for common errors) with ruff.
- Format with ruff-format.
- Type-check with pyright.

and run them automatically with pre-commit.

(Feel free to steal my config from

<https://github.com/patrick-kidger/equinox/blob/main/.pre-commit-config.yaml>

)

Learn collaborative coding:

- Pull requests + code review on GitHub.

For those already familiar:

```
class AbstractSolver(equ.Module):
    @abstractmethod
    def step(...): ...

class AbstractRungeKutta(AbstractSolver):
    tableau: AbstractClassVar[ButcherTableau]
    def step(...): ...
```

- Type safety (~traits) of Rust;
- Abstract/final pattern of Julia;
- Lots and lots of functional programming.

JAX ecosystem: <https://github.com/patrick-kidger/diffrax> (+links within)

JAX ecosystem: <https://github.com/patrick-kidger/diffraction> (+links within)



contact@kidger.site



twitter.com/PatrickKidger



[PatrickKidger.bsky.social](https://patrickkidger.bsky.social)



github.com/patrick-kidger

JAX ecosystem: <https://github.com/patrick-kidger/diffrax> (+links within)



contact@kidger.site



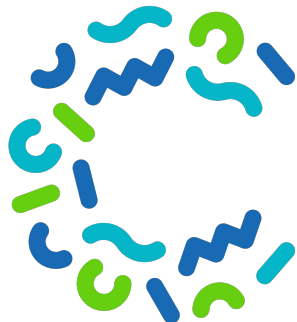
twitter.com/PatrickKidger



PatrickKidger.bsky.social



github.com/patrick-kidger



Cradle.bio

Thank you for your attention!

JAX ecosystem: <https://github.com/patrick-kidger/diffrax> (+links within)



contact@kidger.site



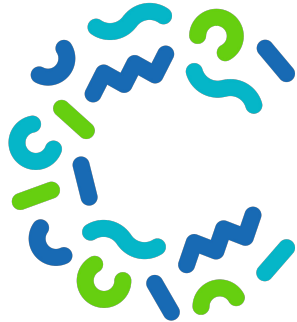
twitter.com/PatrickKidger



PatrickKidger.bsky.social



github.com/patrick-kidger



Cradle.bio