A Theoretical Guide on Differentiation and a Practical Guide to JAX

(or, how never to compute a derivative by hand again)

May 19, 2022

The Basics

JAX is a lot like numpy

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

x = np.array([1., 2., 3.])
y = np.array([0., 1., 5.])
z = np.array([2., 3., 4.])
print(f"with numpy: {x * y + z}")


x_ = jnp.array([1., 2., 3.])
y_ = jnp.array([0., 1., 5.])
z_ = jnp.array([2., 3., 4.])
print(f"with jax: {x_ * y_ + z_}")
```

✓ 0.5s Python

JAX is a lot like numpy

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

x = np.array([1., 2., 3.])
y = np.array([0., 1., 5.])
z = np.array([2., 3., 4.])
print(f"with numpy: {x * y + z}")


x_ = jnp.array([1., 2., 3.])
y_ = jnp.array([0., 1., 5.])
z_ = jnp.array([2., 3., 4.])
print(f"with jax: {x_ * y_ + z_}")
```

✓ 0.5s Python

```
with numpy: [ 2.  5. 19.]
with jax: [ 2.  5. 19.]
```

Most np functions have a jnp counterpart

```python
np_linspace = np.linspace(0, 1, 3)
jnp_linspace = jnp.linspace(0, 1, 3)

print(f"np_linspace: {np_linspace}")
print(f"jnp_linspace: {jnp_linspace}\n")

A = np.random.normal(size = (10, 10))
np_2norm = np.linalg.norm(A, ord = 2)
jnp_2norm = jnp.linalg.norm(A, ord = 2) # notice that we can feed np array into jnp function!

print(f"np_2norm: {np_2norm:.4f}")
print(f"jnp_2norm: {jnp_2norm:.4f}\n")
```

✓ 0.6s Python

Most np functions have a jnp counterpart

```python
np_linspace = np.linspace(0, 1, 3)
jnp_linspace = jnp.linspace(0, 1, 3)

print(f"np_linspace: {np_linspace}")
print(f"jnp_linspace: {jnp_linspace}\n")

A = np.random.normal(size = (10, 10))
np_2norm = np.linalg.norm(A, ord = 2)
jnp_2norm = jnp.linalg.norm(A, ord = 2) # notice that we can feed np array into jnp function!

print(f"np_2norm: {np_2norm:.4f}")
print(f"jnp_2norm: {jnp_2norm:.4f}\n")
```

✓ 0.6s Python

```
np_linspace: [0.  0.5 1. ]
jnp_linspace: [0.  0.5 1. ]


np_2norm: 5.2232
jnp_2norm: 5.2232
```

matplotlib can accept jnp arrays

```python
import matplotlib.pyplot as plt

x = jnp.linspace(0, 2 * jnp.pi)
y = jnp.sin(x)
plt.plot(x, y)
```

✓ 0.9s Python

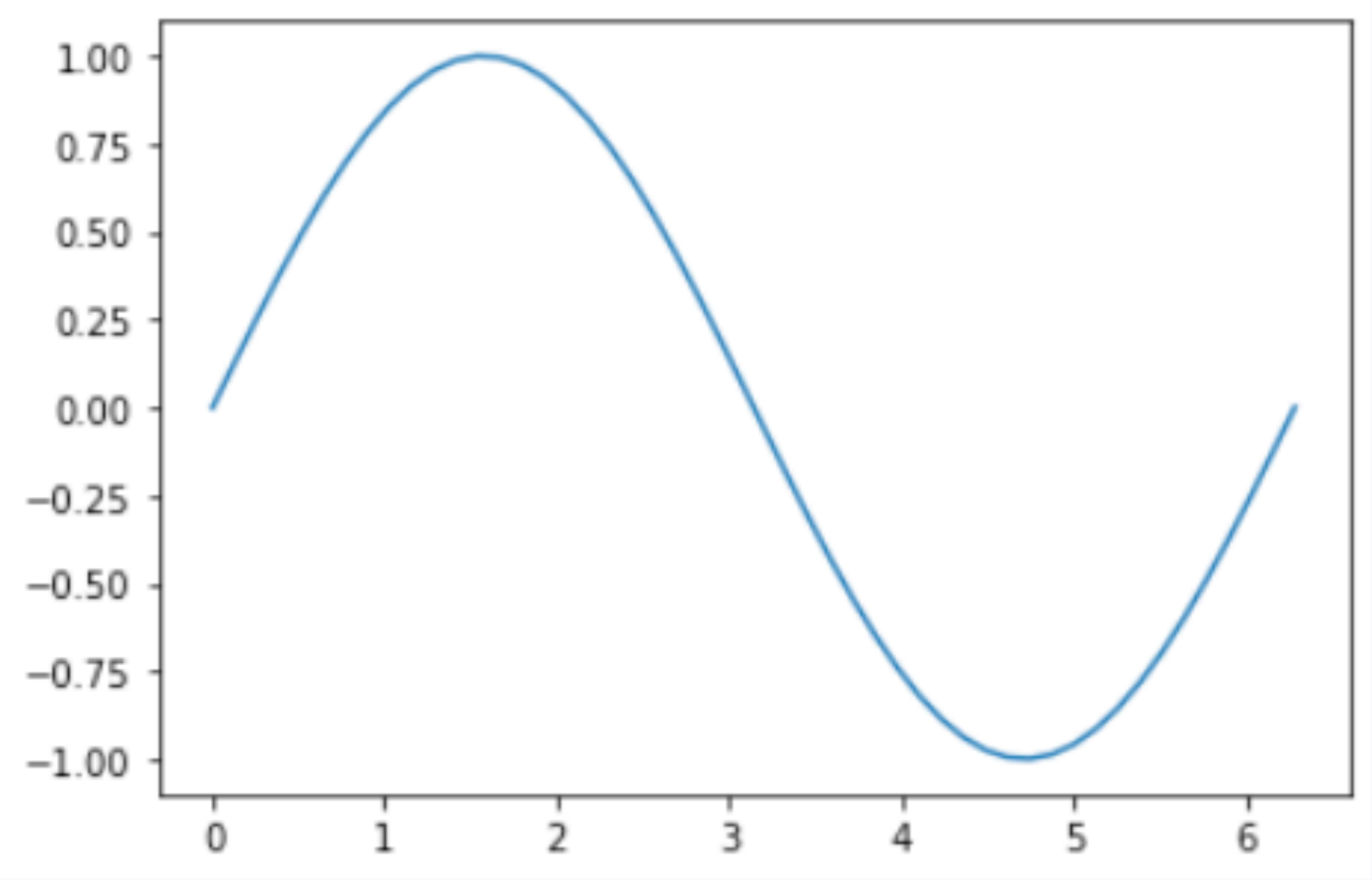
matplotlib can accept jnp arrays

```python
import matplotlib.pyplot as plt

x = jnp.linspace(0, 2 * jnp.pi)
y = jnp.sin(x)
plt.plot(x, y)
```

✓ 0.9s Python

```
[<matplotlib.lines.Line2D at 0x7f5d24561dc0>]
```



RNG is different

```python
from jax import random

key = random.PRNGKey(0) # initialize key from your favorite integer

A = random.normal(key, shape = (2, 2))
print(f"A: {A}")
B = random.normal(key, shape = (2, 2)) # will get the same thing as A!
print(f"B: {B}")

key, subkey = random.split(key) # new key and subkey are different from original key
C = random.normal(key, shape = (2, 2))
print(f"C: {C}")
D = random.normal(subkey, shape = (2, 2))
print(f"D: {D}")
```

✓ 0.2s Python

RNG is different

```python
from jax import random

key = random.PRNGKey(0) # initialize key from your favorite integer

A = random.normal(key, shape = (2, 2))
print(f"A: {A}")
B = random.normal(key, shape = (2, 2)) # will get the same thing as A!
print(f"B: {B}")

key, subkey = random.split(key) # new key and subkey are different from original key
C = random.normal(key, shape = (2, 2))
print(f"C: {C}")
D = random.normal(subkey, shape = (2, 2))
print(f"D: {D}")
```

✓ 0.2s Python

```
A: [[ 1.8160863  -0.75488514]
 [ 0.33988908 -0.53483534]]
B: [[ 1.8160863  -0.75488514]
 [ 0.33988908 -0.53483534]]
C: [[ 0.13893168  1.370668  ]
 [-0.53116107  0.02033782]]
D: [[ 1.1378784  -0.14331433]
 [-0.59153634  0.79466224]]
```

Updating arrays is different

```python
# in numpy, we can do in-place array updates:
A = np.array([1, 2, 3])
print(f"this is A: {A}")

A[0] = 0
print(f"this is the new A in numpy: {A}")
```

✓ 0.3s Python

```python
# in jax, this doesn't work:
A = jnp.array([1, 2, 3])
print(f"this is A: {A}")

A[0] = 0 # this will throw an error
```

⊗ 1.2s Python

```python
# instead, we have to do this:

A = A.at[0].set(0)
print(f"this is the new A: {A}")
```

✓ 0.1s Python

Updating arrays is different

```python
# in numpy, we can do in-place array updates:
A = np.array([1, 2, 3])
print(f"this is A: {A}")

A[0] = 0
print(f"this is the new A in numpy: {A}")
```

✓ 0.3s Python

```
this is A: [1 2 3]
this is the new A in numpy: [0 2 3]
```

```python
# in jax, this doesn't work:
A = jnp.array([1, 2, 3])
print(f"this is A: {A}")

A[0] = 0 # this will throw an error
```

⊗ 1.2s Python

```python
# instead, we have to do this:

A = A.at[0].set(0)
print(f"this is the new A: {A}")
```

✓ 0.1s Python

Updating arrays is different

```python
# in numpy, we can do in-place array updates:
A = np.array([1, 2, 3])
print(f"this is A: {A}")

A[0] = 0
print(f"this is the new A in numpy: {A}")
```

✓ 0.3s Python

```
this is A: [1 2 3]
this is the new A in numpy: [0 2 3]
```

```python
# in jax, this doesn't work:
A = jnp.array([1, 2, 3])
print(f"this is A: {A}")

A[0] = 0 # this will throw an error
```

⊗ 1.2s Python

```
this is A: [1 2 3]

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/data/philliplo125/jax-tutorial/tutorial.ipynb Cell 15' in <cell line: 5>()
      2 A = jnp.array([1, 2, 3])
      3 print(f"this is A: {A}")
----> 5 A[0] = 0

File ~/anaconda3/envs/jax-tutorial/lib/python3.9/site-packages/jax/_src/numpy/lax_numpy.py:4512, in _unimplemented_setitem(self, i, x)
   4507 def _unimplemented_setitem(self, i, x):
   4508   msg = ("'{}' object does not support item assignment. JAX arrays are "
   4509          "immutable. Instead of ``x[idx] = y``, use ``x = x.at[idx].set(y)`` "
   4510          "or another .at[] method: "
   4511          "https://jax.readthedocs.io/en/latest/_autosummary/jax.numpy.ndarray.at.html")
-> 4512   raise TypeError(msg.format(type(self)))

TypeError: '<class 'jaxlib.xla_extension.DeviceArray'>' object does not support item assignment. JAX arrays are immutable. Instead of ``x[id
x] = y``, use ``x = x.at[idx].set(y)`` or another .at[] method: https://jax.readthedocs.io/en/latest/_autosummary/jax.numpy.ndarray.at.html
```

```python
# instead, we have to do this:

A = A.at[0].set(0)
print(f"this is the new A: {A}")
```

✓ 0.1s Python

Updating arrays is different

```python
# in numpy, we can do in-place array updates:
A = np.array([1, 2, 3])
print(f"this is A: {A}")

A[0] = 0
print(f"this is the new A in numpy: {A}")
```

✓ 0.3s Python

```
this is A: [1 2 3]
this is the new A in numpy: [0 2 3]
```

```python
# in jax, this doesn't work:
A = jnp.array([1, 2, 3])
print(f"this is A: {A}")

A[0] = 0 # this will throw an error
```

1.2s Python

```
this is A: [1 2 3]

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/data/philliplo125/jax-tutorial/tutorial.ipynb Cell 15' in <cell line: 5>()
      2 A = jnp.array([1, 2, 3])
      3 print(f"this is A: {A}")
----> 5 A[0] = 0

File ~/anaconda3/envs/jax-tutorial/lib/python3.9/site-packages/jax/_src/numpy/lax_numpy.py:4512, in _unimplemented_setitem(self, i, x)
   4507 def _unimplemented_setitem(self, i, x):
   4508   msg = ("'{}' object does not support item assignment. JAX arrays are "
   4509          "immutable. Instead of ``x[idx] = y``, use ``x = x.at[idx].set(y)`` "
   4510          "or another .at[] method: "
   4511          "https://jax.readthedocs.io/en/latest/_autosummary/jax.numpy.ndarray.at.html")
-> 4512   raise TypeError(msg.format(type(self)))

TypeError: '<class 'jaxlib.xla_extension.DeviceArray'>' object does not support item assignment. JAX arrays are immutable. Instead of ``x[id
x] = y``, use ``x = x.at[idx].set(y)`` or another .at[] method: https://jax.readthedocs.io/en/latest/_autosummary/jax.numpy.ndarray.at.html
```

```python
# instead, we have to do this:

A = A.at[0].set(0)
print(f"this is the new A: {A}")
```

✓ 0.1s Python

```
this is the new A: [0 2 3]
```

Computing Gradients

JAX is highly functional

Use `grad()` to map a scalar-valued function to its derivative

```python
from jax import grad

sin_grad = grad(jnp.sin)
print(f"sin'(1): {sin_grad(1.0):.4f}")
print(f"cos(1): {jnp.cos(1.0):.4f}")
```

✓ 0.4s Python

JAX is highly functional

Use `grad()` to map a scalar-valued function to its derivative

```python
from jax import grad

sin_grad = grad(jnp.sin)
print(f"sin'(1): {sin_grad(1.0):.4f}")
print(f"cos(1): {jnp.cos(1.0):.4f}")
```

✓ 0.4s Python

```
sin'(1): 0.5403
cos(1): 0.5403
```

Get fancy with function composition

Let $f(x) = \sin(x^2) + 3x^2$, then $f'(x) = 2x\cos(x^2) + 6x$

```python
def f(x):
    return jnp.sin(x**2) + 3 * x**2

def grad_f_manual(x):
    return 2 * x * jnp.cos(x**2) + 6 * x

grad_f_auto = grad(f)

x = 2.0
print(f"manual f'(1): {grad_f_manual(x):.4f}")
print(f"auto f'(1): {grad_f_auto(x):.4f}")
```

✓ 0.1s Python

Get fancy with function composition

Let $f(x) = \sin(x^2) + 3x^2$, then $f'(x) = 2x\cos(x^2) + 6x$

```python
def f(x):
    return jnp.sin(x**2) + 3 * x**2

def grad_f_manual(x):
    return 2 * x * jnp.cos(x**2) + 6 * x

grad_f_auto = grad(f)

x = 2.0
print(f"manual f'(1): {grad_f_manual(x):.4f}")
print(f"auto f'(1): {grad_f_auto(x):.4f}")
```

✓ 0.1s Python

```
manual f'(1): 9.3854
auto f'(1): 9.3854
```

Multivariate Functions

$$f(x, y, z) = x^2 + 3\sin(y) + \cos(z)$$

$$\nabla f(x, y, z) = (2x, 3\cos(y), -\sin(z))$$

```python
# denote X = (x, y, z)

def f(X):
    return X[0]**2 + 3 * jnp.sin(X[1]) + jnp.cos(X[2])

def grad_f_manual(X):
    return jnp.array([2 * X[0], 3 * jnp.cos(X[1]), -jnp.sin(X[2])])

grad_f_auto = grad(f)

X = jnp.array([2.0, 3.0, 1.0])
print(f"manual f'(1): {grad_f_manual(X)}")
print(f"auto f'(1): {grad_f_auto(X)}")
```

✓ 0.5s Python

Multivariate Functions

$$f(x, y, z) = x^2 + 3\sin(y) + \cos(z)$$

$$\nabla f(x, y, z) = (2x, 3\cos(y), -\sin(z))$$

```python
# denote X = (x, y, z)

def f(X):
    return X[0]**2 + 3 * jnp.sin(X[1]) + jnp.cos(X[2])

def grad_f_manual(X):
    return jnp.array([2 * X[0], 3 * jnp.cos(X[1]), -jnp.sin(X[2])])

grad_f_auto = grad(f)

X = jnp.array([2.0, 3.0, 1.0])
print(f"manual f'(1): {grad_f_manual(X)}")
print(f"auto f'(1): {grad_f_auto(X)}")
```

✓ 0.5s Python

```
manual f'(1): [ 4.          -2.9699774  -0.84147096]
auto f'(1): [ 4.          -2.9699774  -0.84147096]
```

Differentiate with respect to dictionaries

$$f(x, y, z) = x^2 + 3\sin(y) + \cos(z)$$

$$\nabla f(x, y, z) = (2x, 3\cos(y), -\sin(z))$$

```python
def f(X):
    return X['x']**2 + 3 * jnp.sin(X['y']) + jnp.cos(X['z'])

def grad_f_manual(X):
    return jnp.array([2 * X['x'], 3 * jnp.cos(X['y']), -jnp.sin(X['z'])])

grad_f_auto = grad(f)

X = {'x' : 2.0, 'y' : 3.0, 'z' : 1.0}
print(f"manual f'(1): {grad_f_manual(X)}")
print(f"auto f'(1): {grad_f_auto(X)}")
```

✓ 0.5s Python

Differentiate with respect to dictionaries

$$f(x, y, z) = x^2 + 3\sin(y) + \cos(z)$$

$$\nabla f(x, y, z) = (2x, 3\cos(y), -\sin(z))$$

```python
def f(X):
    return X['x']**2 + 3 * jnp.sin(X['y']) + jnp.cos(X['z'])

def grad_f_manual(X):
    return jnp.array([2 * X['x'], 3 * jnp.cos(X['y']), -jnp.sin(X['z'])])

grad_f_auto = grad(f)

X = {'x' : 2.0, 'y' : 3.0, 'z' : 1.0}
print(f"manual f'(1): {grad_f_manual(X)}")
print(f"auto f'(1): {grad_f_auto(X)}")
```

✓ 0.5s Python

```
manual f'(1): [ 4.         -2.9699774  -0.84147096]
auto f'(1): {'x': DeviceArray(4., dtype=float32, weak_type=True), 'y':
DeviceArray(-2.9699774, dtype=float32, weak_type=True), 'z':
DeviceArray(-0.84147096, dtype=float32, weak_type=True)}
```

Multiple multivariate arguments

$$f(x, y) = \|x\|^2 + \|y\| + \langle x, y \rangle$$

$$D_x f(x, y) = 2x + y$$

$$D_y f(x, y) = \frac{y}{\|y\|} + x$$

$$x, y \in \mathbb{R}^3$$

```python
def f(x, y):
    return jnp.linalg.norm(x)**2 + jnp.linalg.norm(y) + jnp.inner(x, y)

Dxf = grad(f, 0) # take the derivative wrt the zeroth argument
Dyf = grad(f, 1) # take the derivative wrt the first argument

x = jnp.array([1., 2., 3.])
y = jnp.array([0, -1., 1.])

print(f"derivative wrt x: {Dxf(x, y)}")
print(f"manually computed: {2 * x + y}\n")
print(f"derivative wrt y: {Dyf(x, y)}")
print(f"manually computed: {y / jnp.linalg.norm(y) + x}")
```

✓ 0.8s Python

Multiple multivariate arguments

$$f(x, y) = \|x\|^2 + \|y\| + \langle x, y \rangle$$

$$D_x f(x, y) = 2x + y$$

$$D_y f(x, y) = \frac{y}{\|y\|} + x$$

$$x, y \in \mathbb{R}^3$$

```python
def f(x, y):
    return jnp.linalg.norm(x)**2 + jnp.linalg.norm(y) + jnp.inner(x, y)

Dxf = grad(f, 0) # take the derivative wrt the zeroth argument
Dyf = grad(f, 1) # take the derivative wrt the first argument

x = jnp.array([1., 2., 3.])
y = jnp.array([0, -1., 1.])

print(f"derivative wrt x: {Dxf(x, y)}")
print(f"manually computed: {2 * x + y}\n")
print(f"derivative wrt y: {Dyf(x, y)}")
print(f"manually computed: {y / jnp.linalg.norm(y) + x}")
```

✓ 0.8s Python

```
derivative wrt x: [2. 3. 7.]
manually computed: [2. 3. 7.]

derivative wrt y: [1.        1.2928932 3.7071068]
manually computed: [1.        1.2928932 3.7071068]
```

value_and_grad()

Useful for first-order optimization methods

```python
from jax import value_and_grad

def f(x):
    return x ** 2

value, grad = value_and_grad(f)(3.0)
print(f"value: {value}")
print(f"grad: {grad}")
```

✓ 0.4s Python

value_and_grad()

Useful for first-order optimization methods

```python
from jax import value_and_grad

def f(x):
    return x ** 2

value, grad = value_and_grad(f)(3.0)
print(f"value: {value}")
print(f"grad: {grad}")
```

✓ 0.4s Python

```
value: 9.0
grad: 6.0
```

What is Automatic Differentiation?

Not finite differences or symbolic differentiation

Not finite differences or symbolic differentiation

- Finite differences: $\frac{d}{dx}f(x) \approx \frac{f(x+h) - f(x)}{h}$

Not finite differences or symbolic differentiation

- Finite differences: $\dfrac{d}{dx}f(x) \approx \dfrac{f(x+h) - f(x)}{h}$
    - Numerically unstable

Not finite differences or symbolic differentiation

- Finite differences: $\frac{d}{dx}f(x) \approx \frac{f(x+h) - f(x)}{h}$
    - Numerically unstable
    - Expensive

Not finite differences or symbolic differentiation

- Finite differences: $\frac{d}{dx}f(x) \approx \frac{f(x+h) - f(x)}{h}$
    - Numerically unstable
    - Expensive
- Symbolic differentiation:

Not finite differences or symbolic differentiation

- Finite differences: $\frac{d}{dx}f(x) \approx \frac{f(x+h) - f(x)}{h}$
  - Numerically unstable
  - Expensive
- Symbolic differentiation:

```
In[1]:= D[Log[1 + Exp[w2 * Log[1 + Exp[w1 * x + b1]] + b2]], w1]
```

$$\text{Out[1]=} \frac{e^{b1+b2+w1\,x+w2\,\mathrm{Log}\left[1+e^{b1+w1\,x}\right]}\, w2\, x}{\left(1+e^{b1+w1\,x}\right) \times \left(1+e^{b2+w2\,\mathrm{Log}\left[1+e^{b1+w1\,x}\right]}\right)}$$

Chain rule

Chain rule

$$h(x) = f(g(x))$$

$$\frac{dh}{dx} = \frac{df}{dg}\frac{dg}{dx}$$

Chain rule

$$h(x) = f(g(x))$$

$$\frac{dh}{dx} = \frac{df}{dg}\frac{dg}{dx}$$

$$\frac{d}{dx}f(g_1(x), \ldots, g_K(x)) = \sum_{k=1}^{K} \left(\frac{d}{dx}g_k(x)\right) D_k f(g_1(x), \ldots, g_K(x))$$

Chain rule

$$h(x) = f(g(x))$$

$$\frac{dh}{dx} = \frac{df}{dg}\frac{dg}{dx}$$

$$\frac{d}{dx}f(g_1(x), \ldots, g_K(x)) = \sum_{k=1}^{K} \left(\frac{d}{dx}g_k(x)\right) D_k f(g_1(x), \ldots, g_K(x))$$

Idea: the chain rule is <u>modular</u>; as long as we know the derivatives of elementary functions, we can compute the derivatives of arbitrary compositions of these functions

Example: regularized univariate linear predictor

$$z = mx + b$$

$$\hat{y} = \sigma(z)$$

$$L^{m,b}(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$$

$$R(m) = \frac{1}{2}m^2$$

$$L_R^{m,b} = L^{m,b}(\hat{y}, y) + \lambda R(m)$$

Example: regularized univariate linear predictor

$$z = mx + b$$

$$\hat{y} = \sigma(z)$$

$$L^{m,b}(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$$

$$R(m) = \frac{1}{2}m^2$$

$$L_R^{m,b} = L^{m,b}(\hat{y}, y) + \lambda R(m)$$

$$\begin{tikzcd}
x \arrow[dr] & & y \arrow[dr] & & \\
b \arrow[r] & z \arrow[r] & \hat{y} \arrow[r] & L \arrow[r] & L_R \\
m \arrow[ur] \arrow[rrr] & & & R \arrow[ur] &
\end{tikzcd}$$

Example: regularized univariate linear predictor

$$z = mx + b$$

$$\hat{y} = \sigma(z)$$

$$L^{m,b}(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$$

$$R(m) = \frac{1}{2}m^2$$

$$L_R^{m,b} = L^{m,b}(\hat{y}, y) + \lambda R(m)$$

$$\begin{tikzcd}
x \arrow[dr] & & y \arrow[dr] & & \\
b \arrow[r] & z \arrow[r] & \hat{y} \arrow[r] & L \arrow[r] & L_R \\
m \arrow[ur] \arrow[rrr] & & & R \arrow[ur] &
\end{tikzcd}$$

parent → child, <u>derivative w.r.t. parent requires derivatives w.r.t children</u>

Example: regularized univariate linear predictor

$$z = mx + b$$
$$\hat{y} = \sigma(z)$$
$$L^{m,b}(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$$
$$R(m) = \frac{1}{2}m^2$$
$$L_R^{m,b} = L^{m,b}(\hat{y}, y) + \lambda R(m)$$

$$\begin{tikzcd}
x \arrow[dr] & & y \arrow[dr] & & \\
b \arrow[r] & z \arrow[r] & \hat{y} \arrow[r] & L \arrow[r] & L_R \\
m \arrow[ur] \arrow[rrr] & & & R \arrow[ur] &
\end{tikzcd}$$

parent → child, <u>derivative w.r.t. parent requires derivatives w.r.t children</u>

Automatic differentiation algorithm: work backwards through the computation tree, computing derivatives of parents w.r.t children

Example: regularized univariate linear predictor

$$z = mx + b$$
$$\hat{y} = \sigma(z)$$
$$L^{m,b}(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$$
$$R(m) = \frac{1}{2}m^2$$
$$L_R^{m,b} = L^{m,b}(\hat{y}, y) + \lambda R(m)$$

$$\begin{tikzcd}
x \arrow[dr] & & y \arrow[dr] & & \\
b \arrow[r] & z \arrow[r] & \hat{y} \arrow[r] & L \arrow[r] & L_R \\
m \arrow[ur] \arrow[rrr] & & & R \arrow[ur] &
\end{tikzcd}$$

$$D_{L_R} = 1$$

parent → child, <u>derivative w.r.t. parent requires derivatives w.r.t children</u>

Automatic differentiation algorithm: work backwards through the computation tree, computing derivatives of parents w.r.t children

Example: regularized univariate linear predictor

$$z = mx + b$$
$$\hat{y} = \sigma(z)$$
$$L^{m,b}(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$$
$$R(m) = \frac{1}{2}m^2$$
$$L_R^{m,b} = L^{m,b}(\hat{y}, y) + \lambda R(m)$$

$$\begin{tikzcd}
x \arrow[dr] & & y \arrow[dr] & & \\
b \arrow[r] & z \arrow[r] & \hat{y} \arrow[r] & L \arrow[r] & L_R \\
m \arrow[ur] \arrow[rrr] & & & R \arrow[ur] &
\end{tikzcd}$$

parent → child, <u>derivative w.r.t. parent requires derivatives w.r.t children</u>

Automatic differentiation algorithm: work backwards through the computation tree, computing derivatives of parents w.r.t children

$$D_{L_R} = 1$$

$$D_L = D_{L_R}\frac{dL_R}{dL} = D_{L_R}$$

Example: regularized univariate linear predictor

$$z = mx + b$$
$$\hat{y} = \sigma(z)$$
$$L^{m,b}(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$$
$$R(m) = \frac{1}{2}m^2$$
$$L_R^{m,b} = L^{m,b}(\hat{y}, y) + \lambda R(m)$$

$$\begin{tikzcd}
x \arrow[dr] & & y \arrow[dr] & & \\
b \arrow[r] & z \arrow[r] & \hat{y} \arrow[r] & L \arrow[r] & L_R \\
m \arrow[ur] \arrow[rrr] & & & R \arrow[ur] &
\end{tikzcd}$$

parent → child, <u>derivative w.r.t. parent requires derivatives w.r.t children</u>

Automatic differentiation algorithm: work backwards through the computation tree, computing derivatives of parents w.r.t children

$$D_{L_R} = 1$$

$$D_L = D_{L_R}\frac{dL_R}{dL} = D_{L_R}$$

$$D_R = D_{L_R}\frac{dL_R}{dR} = \lambda D_{L_R}$$

Example: regularized univariate linear predictor

$$z = mx + b$$
$$\hat{y} = \sigma(z)$$
$$L^{m,b}(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$$
$$R(m) = \frac{1}{2}m^2$$
$$L_R^{m,b} = L^{m,b}(\hat{y}, y) + \lambda R(m)$$

$$\begin{tikzcd}
x \arrow[dr] & & y \arrow[dr] & & \\
b \arrow[r] & z \arrow[r] & \hat{y} \arrow[r] & L \arrow[r] & L_R \\
m \arrow[ur] \arrow[rrr] & & & R \arrow[ur] &
\end{tikzcd}$$

parent → child, <u>derivative w.r.t. parent requires derivatives w.r.t children</u>

Automatic differentiation algorithm: work backwards through the computation tree, computing derivatives of parents w.r.t children

$$D_{L_R} = 1$$

$$D_L = D_{L_R}\frac{dL_R}{dL} = D_{L_R}$$

$$D_R = D_{L_R}\frac{dL_R}{dR} = \lambda D_{L_R}$$

$$D_{\hat{y}} = D_L\frac{dL}{dy} = (\hat{y} - y)D_L$$

Example: regularized univariate linear predictor

$$
\begin{aligned}
z &= mx + b \\
\hat{y} &= \sigma(z) \\
L^{m,b}(\hat{y}, y) &= \frac{1}{2}(\hat{y} - y)^2 \\
R(m) &= \frac{1}{2}m^2 \\
L_R^{m,b} &= L^{m,b}(\hat{y}, y) + \lambda R(m)
\end{aligned}
$$

$$
\begin{tikzcd}
x \arrow[dr] & & y \arrow[dr] & & \\
b \arrow[r] & z \arrow[r] & \hat{y} \arrow[r] & L \arrow[r] & L_R \\
m \arrow[ur] \arrow[rrr] & & & R \arrow[ur] &
\end{tikzcd}
$$

parent → child, <u>derivative w.r.t. parent requires derivatives w.r.t children</u>

Automatic differentiation algorithm: work backwards through the computation tree, computing derivatives of parents w.r.t children

$$D_{L_R} = 1$$

$$D_L = D_{L_R}\frac{dL_R}{dL} = D_{L_R}$$

$$D_R = D_{L_R}\frac{dL_R}{dR} = \lambda D_{L_R}$$

$$D_{\hat{y}} = D_L\frac{dL}{dy} = (\hat{y} - y)D_L$$

$$D_z = D_{\hat{y}}\frac{d\hat{y}}{dz} = D_{\hat{y}}\sigma'(z)$$

Example: regularized univariate linear predictor

$$z = mx + b$$
$$\hat{y} = \sigma(z)$$
$$L^{m,b}(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$$
$$R(m) = \frac{1}{2}m^2$$
$$L_R^{m,b} = L^{m,b}(\hat{y}, y) + \lambda R(m)$$

$$\begin{tikzcd}
x \arrow[dr] & & y \arrow[dr] & & \\
b \arrow[r] & z \arrow[r] & \hat{y} \arrow[r] & L \arrow[r] & L_R \\
m \arrow[ur] \arrow[rrr] & & & R \arrow[ur] &
\end{tikzcd}$$

parent → child, <u>derivative w.r.t. parent requires derivatives w.r.t children</u>

Automatic differentiation algorithm: work backwards through the computation tree, computing derivatives of parents w.r.t children

$$D_{L_R} = 1$$

$$D_L = D_{L_R}\frac{dL_R}{dL} = D_{L_R}$$

$$D_R = D_{L_R}\frac{dL_R}{dR} = \lambda D_{L_R}$$

$$D_{\hat{y}} = D_L\frac{dL}{dy} = (\hat{y} - y)D_L$$

$$D_z = D_{\hat{y}}\frac{d\hat{y}}{dz} = D_{\hat{y}}\sigma'(z)$$

$$D_m = D_z\frac{dz}{dm} + D_R\frac{dR}{dm}$$
$$= D_z x + D_R m$$

Example: regularized univariate linear predictor

$$
\begin{aligned}
z &= mx + b \\
\hat{y} &= \sigma(z) \\
L^{m,b}(\hat{y}, y) &= \frac{1}{2}(\hat{y} - y)^2 \\
R(m) &= \frac{1}{2}m^2 \\
L_R^{m,b} &= L^{m,b}(\hat{y}, y) + \lambda R(m)
\end{aligned}
$$

$$
\begin{tikzcd}
x \arrow[dr] & & y \arrow[dr] & & \\
b \arrow[r] & z \arrow[r] & \hat{y} \arrow[r] & L \arrow[r] & L_R \\
m \arrow[ur] \arrow[rrr] & & & R \arrow[ur] &
\end{tikzcd}
$$

parent → child, <u>derivative w.r.t. parent requires derivatives w.r.t children</u>

Automatic differentiation algorithm: work backwards through the computation tree, computing derivatives of parents w.r.t children

$$D_{L_R} = 1$$

$$D_L = D_{L_R}\frac{dL_R}{dL} = D_{L_R}$$

$$D_R = D_{L_R}\frac{dL_R}{dR} = \lambda D_{L_R}$$

$$D_{\hat{y}} = D_L\frac{dL}{dy} = (\hat{y} - y)D_L$$

$$D_z = D_{\hat{y}}\frac{d\hat{y}}{dz} = D_{\hat{y}}\sigma'(z)$$

$$
\begin{aligned}
D_m &= D_z\frac{dz}{dm} + D_R\frac{dR}{dm} \\
&= D_z x + D_R m
\end{aligned}
$$

etc…

Example: regularized univariate linear predictor

$$z = mx + b$$
$$\hat{y} = \sigma(z)$$
$$L^{m,b}(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$$
$$R(m) = \frac{1}{2}m^2$$
$$L_R^{m,b} = L^{m,b}(\hat{y}, y) + \lambda R(m)$$

$$\begin{tikzcd}
x \arrow[dr] & & y \arrow[dr] & & \\
b \arrow[r] & z \arrow[r] & \hat{y} \arrow[r] & L \arrow[r] & L_R \\
m \arrow[ur] \arrow[rrr] & & & R \arrow[ur] &
\end{tikzcd}$$

parent → child, <u>derivative w.r.t. parent requires derivatives w.r.t children</u>

Automatic differentiation algorithm: work backwards through the computation tree, computing derivatives of parents w.r.t children

$$D_{L_R} = 1$$

$$D_L = D_{L_R}\frac{dL_R}{dL} = D_{L_R}$$

$$D_R = D_{L_R}\frac{dL_R}{dR} = \lambda D_{L_R}$$

$$D_{\hat{y}} = D_L\frac{dL}{dy} = (\hat{y} - y)D_L$$

$$D_z = D_{\hat{y}}\frac{d\hat{y}}{dz} = D_{\hat{y}}\sigma'(z)$$

$$D_m = D_z\frac{dz}{dm} + D_R\frac{dR}{dm}$$
$$= D_z x + D_R m$$

etc…

Key insight: modular! No need to recompute things!

Jacobians

What is a Jacobian?

What is a Jacobian?

- Let $f : \mathbb{R}^n \to \mathbb{R}^m$

What is a Jacobian?

- Let $f : \mathbb{R}^n \to \mathbb{R}^m$
- What is the Jacobian of f?

What is a Jacobian?

- Let $f : \mathbb{R}^n \to \mathbb{R}^m$
- What is the Jacobian of f ?
- Calculus student: matrix of partial derivatives

What is a Jacobian?

- Let $f : \mathbb{R}^n \to \mathbb{R}^m$
- What is the Jacobian of f ?
- Calculus student: matrix of partial derivatives

$$Jf(x) = \left[\frac{\partial f}{\partial x_{ij}}\right]_{ij} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_j} \\ & \vdots & \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_j} \end{bmatrix}$$

What is a Jacobian?

- Let $f : \mathbb{R}^n \to \mathbb{R}^m$
- What is the Jacobian of f ?
- Calculus student: matrix of partial derivatives

$$Jf(x) = \left[\frac{\partial f}{\partial x_{ij}}\right]_{ij} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_j} \\ & \vdots & \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_j} \end{bmatrix}$$

- Functional analyst: linear operator from $\mathbb{R}^n \to \mathbb{R}^m$

What is a Jacobian?

- Let $f : \mathbb{R}^n \to \mathbb{R}^m$
- What is the Jacobian of f?
- Calculus student: matrix of partial derivatives

$$Jf(x) = \left[\frac{\partial f}{\partial x_{ij}}\right]_{ij} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_j} \\ & \vdots & \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_j} \end{bmatrix}$$

- Functional analyst: linear operator from $\mathbb{R}^n \to \mathbb{R}^m$
- Differential geometer: a map from the tangent bundle of $\mathbb{R}^n$ to $\mathbb{R}^m$

Thinking about Jacobians like a Functional Analyst

Thinking about Jacobians like a Functional Analyst

- Linear operator vs. the matrix *representing* that operator!

Thinking about Jacobians like a Functional Analyst

- Linear operator vs. the matrix *representing* that operator!
- Jacobian as a linear operator:

Thinking about Jacobians like a Functional Analyst

- Linear operator vs. the matrix *representing* that operator!
- Jacobian as a linear operator:

$$Jf(x) : \mathbb{R}^n \to \mathbb{R}^m$$
$$v \mapsto Jf(x)v$$

Thinking about Jacobians like a Functional Analyst

- Linear operator vs. the matrix *representing* that operator!
- Jacobian as a linear operator:

$$Jf(x) : \mathbb{R}^n \to \mathbb{R}^m$$
$$v \mapsto Jf(x)v$$

- The domain of $Jf(x)$ is the tangent space to the domain of $f : \mathbb{R}^n \to \mathbb{R}^m$, denoted $T_x\mathbb{R}^n$; this just happens to be a *copy* of $\mathbb{R}^n$

Thinking about Jacobians like a Functional Analyst

- Linear operator vs. the matrix *representing* that operator!
- Jacobian as a linear operator:

$$Jf(x) : \mathbb{R}^n \to \mathbb{R}^m$$

$$v \mapsto Jf(x)v$$

- The domain of $Jf(x)$ is the tangent space to the domain of $f : \mathbb{R}^n \to \mathbb{R}^m$, denoted $T_x\mathbb{R}^n$; this just happens to be a *copy* of $\mathbb{R}^n$
  - Tangent space to a manifold is the vector space best approximating that manifold

Thinking about Jacobians like a Functional Analyst

- Linear operator vs. the matrix *representing* that operator!
- Jacobian as a linear operator:

$$Jf(x) : \mathbb{R}^n \to \mathbb{R}^m$$
$$v \mapsto Jf(x)v$$

- The domain of $Jf(x)$ is the tangent space to the domain of $f : \mathbb{R}^n \to \mathbb{R}^m$, denoted $T_x\mathbb{R}^n$; this just happens to be a *copy* of $\mathbb{R}^n$
  - Tangent space to a manifold is the vector space best approximating that manifold
  - Tangent vectors v and domain vectors x are different types of objects

Thinking about Jacobians like a Functional Analyst

- Linear operator vs. the matrix *representing* that operator!
- Jacobian as a linear operator:

$$Jf(x) : \mathbb{R}^n \to \mathbb{R}^m$$
$$v \mapsto Jf(x)v$$

- The domain of $Jf(x)$ is the tangent space to the domain of $f : \mathbb{R}^n \to \mathbb{R}^m$, denoted $T_x\mathbb{R}^n$; this just happens to be a *copy* of $\mathbb{R}^n$
  - Tangent space to a manifold is the vector space best approximating that manifold
  - Tangent vectors v and domain vectors x are different types of objects
- Similarly, the range of $Jf(x)$ is the tangent space to the range of f at $f(x)$, denoted $T_{f(x)}\mathbb{R}^m$

Thinking about Jacobians like a Differential Geometer

Thinking about Jacobians like a Differential Geometer

- If we don't fix x, then Jf is a function that first takes a domain vector $x \in \mathbb{R}^n$, which then defines a linear operator at that point:

Thinking about Jacobians like a Differential Geometer

- If we don't fix x, then Jf is a function that first takes a domain vector $x \in \mathbb{R}^n$, which then defines a linear operator at that point:

$$Jf : \mathbb{R}^n \to (\underbrace{\mathbb{R}^n \to \mathbb{R}^m}_{\text{an operator}})$$

Thinking about Jacobians like a Differential Geometer

- If we don't fix x, then Jf is a function that first takes a domain vector $x \in \mathbb{R}^n$, which then defines a linear operator at that point:

$$Jf : \mathbb{R}^n \to (\underbrace{\mathbb{R}^n \to \mathbb{R}^m}_{\text{an operator}})$$

- Better yet, we can think of Jf as taking in two inputs; domain is *tangent bundle* of $\mathbb{R}^n$:

Thinking about Jacobians like a Differential Geometer

- If we don't fix x, then Jf is a function that first takes a domain vector $x \in \mathbb{R}^n$, which then defines a linear operator at that point:

$$Jf : \mathbb{R}^n \to (\underbrace{\mathbb{R}^n \to \mathbb{R}^m}_{\text{an operator}})$$

- Better yet, we can think of Jf as taking in two inputs; domain is *tangent bundle* of $\mathbb{R}^n$:

$$Jf : (x, v) \mapsto Jf(x)v$$

Thinking about Jacobians like a Differential Geometer

- If we don't fix x, then Jf is a function that first takes a domain vector $x \in \mathbb{R}^n$, which then defines a linear operator at that point:

$$Jf : \mathbb{R}^n \to (\underbrace{\mathbb{R}^n \to \mathbb{R}^m}_{\text{an operator}})$$

- Better yet, we can think of Jf as taking in two inputs; domain is *tangent bundle* of $\mathbb{R}^n$:

$$Jf : (x, v) \mapsto Jf(x)v$$

- When thinking of Jacobians in this way: we call this the <u>Jacobian-vector product</u> (JVP)

Thinking about Jacobians like a Differential Geometer

- If we don't fix x, then Jf is a function that first takes a domain vector $x \in \mathbb{R}^n$, which then defines a linear operator at that point:

$$Jf : \mathbb{R}^n \to (\underbrace{\mathbb{R}^n \to \mathbb{R}^m}_{\text{an operator}})$$

- Better yet, we can think of Jf as taking in two inputs; domain is *tangent bundle* of $\mathbb{R}^n$:

$$Jf : (x, v) \mapsto Jf(x)v$$

- When thinking of Jacobians in this way: we call this the <u>Jacobian-vector product</u> (JVP)

JAX's `jvp()`

$$f(x, y) = (x + 2y, \sin(x)e^y, y^3),\ f : \mathbb{R}^2 \to \mathbb{R}^3$$

$$Jf(x, y) = \begin{bmatrix} 1 & 2 \\ \cos(x)e^y & \sin(x)e^y \\ 0 & 3y^2 \end{bmatrix} \in \mathbb{R}^{3\times 2}$$

```python
from jax import jvp

def f(x):
    return jnp.array([x[0] + 2 * x[1], jnp.sin(x[0]) * jnp.exp(x[1]), x[1]**3])

def Jf_manual(x):
    return jnp.array([
        [1, 2],
        [jnp.cos(x[0]) * jnp.exp(x[1]), jnp.sin(x[0]) * jnp.exp(x[1])],
        [0, 3 * x[1] **2]
    ])

# compute the jvp manually
def jvp_manual(x, v):
    return Jf_manual(x) @ v

x = jnp.array([1., 2.])
v = jnp.array([2., 3.])

print(f"manually computed jvp: {jvp_manual(x, v)}")

# compute the jvp using jax.jvp():

func_value, jvp_JAX = jvp(f, (x,), (v,)) # jvp returns both the function value and the jvp
print(f"JAX computed jvp: {jvp_JAX}")
```

✓ 0.3s Python

JAX's `jvp()`

$$f(x, y) = (x + 2y, \sin(x)e^y, y^3),\ f : \mathbb{R}^2 \to \mathbb{R}^3$$

$$Jf(x, y) = \begin{bmatrix} 1 & 2 \\ \cos(x)e^y & \sin(x)e^y \\ 0 & 3y^2 \end{bmatrix} \in \mathbb{R}^{3\times 2}$$

```python
from jax import jvp

def f(x):
    return jnp.array([x[0] + 2 * x[1], jnp.sin(x[0]) * jnp.exp(x[1]), x[1]**3])

def Jf_manual(x):
    return jnp.array([
        [1, 2],
        [jnp.cos(x[0]) * jnp.exp(x[1]), jnp.sin(x[0]) * jnp.exp(x[1])],
        [0, 3 * x[1] **2]
    ])

# compute the jvp manually
def jvp_manual(x, v):
    return Jf_manual(x) @ v

x = jnp.array([1., 2.])
v = jnp.array([2., 3.])

print(f"manually computed jvp: {jvp_manual(x, v)}")

# compute the jvp using jax.jvp():

func_value, jvp_JAX = jvp(f, (x,), (v,)) # jvp returns both the function value and the jvp
print(f"JAX computed jvp: {jvp_JAX}")
```

✓ 0.3s Python

```
manually computed jvp: [ 8.        26.637676 36.       ]
JAX computed jvp: [ 8.        26.637676 36.       ]
```

But what if we want the full Jacobian matrix?

Post-multiplying a matrix with the jth standard basis vector reveals the jth column of the matrix, so can construct the full $m \times n$ Jacobian matrix with n JVPs:

```python
def Jf_JVPs(x): # compute the full Jacobian at x using JVPs
    Jf = np.zeros(shape = (3, 2))
    for j in range(2):
        # compute the jth basis vector
        ej = np.zeros(shape = 2)
        ej[j] = 1

        # set the jth column equal to JVP with v = ej
        Jf[:, j] = jvp(f, (x,), (ej,))[1] # remember jvp returns function value and jvp

    return Jf

x = jnp.array([2., 3.])

print(f"Jf with JVPs: \n {Jf_JVPs(x)}")
print(f"Jf manual: \n {Jf_manual(x)}")
```

✓ 0.7s Python

But what if we want the full Jacobian matrix?

Post-multiplying a matrix with the jth standard basis vector reveals the jth column of the matrix, so can construct the full $m \times n$ Jacobian matrix with n JVPs:

```python
def Jf_JVPs(x): # compute the full Jacobian at x using JVPs
    Jf = np.zeros(shape = (3, 2))
    for j in range(2):
        # compute the jth basis vector
        ej = np.zeros(shape = 2)
        ej[j] = 1

        # set the jth column equal to JVP with v = ej
        Jf[:, j] = jvp(f, (x,), (ej,))[1] # remember jvp returns function value and jvp

    return Jf

x = jnp.array([2., 3.])

print(f"Jf with JVPs: \n {Jf_JVPs(x)}")
print(f"Jf manual: \n {Jf_manual(x)}")
```

✓ 0.7s Python

```
Jf with JVPs:
 [[ 1.          2.        ]
 [-8.35853291 18.26372719]
 [ 0.         27.        ]]
Jf manual:
 [[ 1.        2.      ]
 [-8.358533 18.263727]
 [ 0.       27.      ]]
```

Forward-mode differentiation: `jacfwd()`

```python
from jax import jacfwd

print(f"Jf with jacfwd(): \n {jacfwd(f)(x)}")
```

✓ 0.3s Python

```
Jf with jacfwd():
 [[ 1.         2.        ]
 [-8.358533 18.263727]
 [ 0.        27.        ]]
```

JVPs good for tall and skinny Jacobians

JVPs good for tall and skinny Jacobians

- JVPs compute $Jf(x)v$

JVPs good for tall and skinny Jacobians

- JVPs compute $Jf(x)v$

- By repeating this for $v = e_1, \ldots, e_n$, we can reconstruct the full Jacobian matrix one column at a time; `jacfwd()` does this for us under the hood

JVPs good for tall and skinny Jacobians

- JVPs compute $Jf(x)v$

- By repeating this for $v = e_1, \ldots, e_n$, we can reconstruct the full Jacobian matrix one column at a time; `jacfwd()` does this for us under the hood

- Better for tall and skinny Jacobians (a few variables mapping to a lot of variables)

JVPs good for tall and skinny Jacobians

- JVPs compute $Jf(x)v$
- By repeating this for $v = e_1, \ldots, e_n$, we can reconstruct the full Jacobian matrix one column at a time; `jacfwd()` does this for us under the hood
- Better for tall and skinny Jacobians (a few variables mapping to a lot of variables)
- But often more interested in short and fat Jacobians (a lot of variables mapping to a few variables), e.g., loss functions

JVPs good for tall and skinny Jacobians

- JVPs compute $Jf(x)v$
- By repeating this for $v = e_1, \ldots, e_n$, we can reconstruct the full Jacobian matrix one column at a time; `jacfwd()` does this for us under the hood
- Better for tall and skinny Jacobians (a few variables mapping to a lot of variables)
- But often more interested in short and fat Jacobians (a lot of variables mapping to a few variables), e.g., loss functions
- If only we could reconstruct a Jacobian matrix one row at a time…

Vector-Jacobian products (VJPs)

Vector-Jacobian products (VJPs)

- If $Jf(x) \in \mathbb{R}^{m \times n}$ is a Jacobian matrix, then premultiplying with the basis vector $e_i^T \in \mathbb{R}^m$ will reveal the ith row of the Jacobian matrix

Vector-Jacobian products (VJPs)

- If $Jf(x) \in \mathbb{R}^{m \times n}$ is a Jacobian matrix, then premultiplying with the basis vector $e_i^T \in \mathbb{R}^m$ will reveal the ith row of the Jacobian matrix

- Vector Jacobian products: $(x, v^T) \mapsto v^T Jf(x)$

Vector-Jacobian products (VJPs)

- If $Jf(x) \in \mathbb{R}^{m \times n}$ is a Jacobian matrix, then premultiplying with the basis vector $e_i^T \in \mathbb{R}^m$ will reveal the ith row of the Jacobian matrix

- Vector Jacobian products: $(x, v^T) \mapsto v^T Jf(x)$

- v^T lives in the *cotangent space* (dual of the tangent space) of the range of f(x)

Vector-Jacobian products (VJPs)

- If $Jf(x) \in \mathbb{R}^{m \times n}$ is a Jacobian matrix, then premultiplying with the basis vector $e_i^T \in \mathbb{R}^m$ will reveal the ith row of the Jacobian matrix

- Vector Jacobian products: $(x, v^T) \mapsto v^T Jf(x)$

- v^T lives in the *cotangent space* (dual of the tangent space) of the range of f(x)

    - Remember $Jf(x)$ maps from the tangent space of $\mathbb{R}^n$ at a point to the tangent space of $\mathbb{R}^m$ at a point

Vector-Jacobian products (VJPs)

- If $Jf(x) \in \mathbb{R}^{m \times n}$ is a Jacobian matrix, then premultiplying with the basis vector $e_i^T \in \mathbb{R}^m$ will reveal the ith row of the Jacobian matrix

- Vector Jacobian products: $(x, v^T) \mapsto v^T Jf(x)$

- v^T lives in the *cotangent space* (dual of the tangent space) of the range of f(x)

  - Remember $Jf(x)$ maps from the tangent space of $\mathbb{R}^n$ at a point to the tangent space of $\mathbb{R}^m$ at a point

  - So v^T is a *linear functional* on the tangent space of $\mathbb{R}^m$: $v^T \in (T_{f(x)}\mathbb{R}^m)^* \cong \mathbb{R}^m$

Vector-Jacobian products (VJPs)

- If $Jf(x) \in \mathbb{R}^{m \times n}$ is a Jacobian matrix, then premultiplying with the basis vector $e_i^T \in \mathbb{R}^m$ will reveal the ith row of the Jacobian matrix

- Vector Jacobian products: $(x, v^T) \mapsto v^T Jf(x)$

- v^T lives in the *cotangent space* (dual of the tangent space) of the range of f(x)

    - Remember $Jf(x)$ maps from the tangent space of $\mathbb{R}^n$ at a point to the tangent space of $\mathbb{R}^m$ at a point

    - So v^T is a *linear functional* on the tangent space of $\mathbb{R}^m$: $v^T \in (T_{f(x)}\mathbb{R}^m)^* \cong \mathbb{R}^m$

- The n-dimensional vector $v^T Jf(x)$ is a row vector, making it a linear opeartor on $\mathbb{R}^n$, so it lives in the cotangent space of $\mathbb{R}^n$: $v^T Jf(x) \in (T_x\mathbb{R}^n)^* \cong \mathbb{R}^n$

$$
\begin{tikzcd}
(T_x\mathbb{R}^n)^* & (T_{f(x)}\mathbb{R}^m)^* \arrow[l, "\mathrm{vjp}"'] \\
T_x\mathbb{R}^n \arrow[r, "\mathrm{jvp}"] & T_{f(x)}\mathbb{R}^m \\
\mathbb{R}^n \arrow[r, "f"] & \mathbb{R}^m
\end{tikzcd}
$$

vjp() example

$$f(x, y) = (x + 2y, \sin(x)e^y, y^3),\ f : \mathbb{R}^2 \to \mathbb{R}^3$$

$$Jf(x, y) = \begin{bmatrix} 1 & 2 \\ \cos(x)e^y & \sin(x)e^y \\ 0 & 3y^2 \end{bmatrix} \in \mathbb{R}^{3\times 2}$$

```python
from jax import vjp

def f(x):
    return jnp.array([x[0] + 2 * x[1], jnp.sin(x[0]) * jnp.exp(x[1]), x[1]**3])

def Jf_manual(x):
    return jnp.array([
        [1, 2],
        [jnp.cos(x[0]) * jnp.exp(x[1]), jnp.sin(x[0]) * jnp.exp(x[1])],
        [0, 3 * x[1] **2]
    ])

# compute the vjp manually
def vjp_manual(x, v):
    return v @ Jf_manual(x)

x = jnp.array([1., 2.])
v = jnp.array([2., 3., 4.])

print(f"manually computed jvp: {vjp_manual(x, v)}")

# compute the jvp using jax.jvp():

func_value, vjp_JAX = vjp(f, x) # vjp signature is a bit different from jvp
print(f"JAX computed jvp: {vjp_JAX(v)}")
```

✓ 0.2s Python

`vjp()` example

$$f(x, y) = (x + 2y, \sin(x)e^y, y^3), \; f : \mathbb{R}^2 \to \mathbb{R}^3$$

$$Jf(x, y) = \begin{bmatrix} 1 & 2 \\ \cos(x)e^y & \sin(x)e^y \\ 0 & 3y^2 \end{bmatrix} \in \mathbb{R}^{3\times 2}$$

```python
from jax import vjp

def f(x):
    return jnp.array([x[0] + 2 * x[1], jnp.sin(x[0]) * jnp.exp(x[1]), x[1]**3])

def Jf_manual(x):
    return jnp.array([
        [1, 2],
        [jnp.cos(x[0]) * jnp.exp(x[1]), jnp.sin(x[0]) * jnp.exp(x[1])],
        [0, 3 * x[1] **2]
    ])

# compute the vjp manually
def vjp_manual(x, v):
    return v @ Jf_manual(x)

x = jnp.array([1., 2.])
v = jnp.array([2., 3., 4.])

print(f"manually computed jvp: {vjp_manual(x, v)}")

# compute the jvp using jax.jvp():

func_value, vjp_JAX = vjp(f, x) # vjp signature is a bit different from jvp
print(f"JAX computed jvp: {vjp_JAX(v)}")
```

✓ 0.2s Python

```
manually computed jvp: [13.976972 70.65303 ]
JAX computed jvp: (DeviceArray([13.976972, 70.65303 ], dtype=float32),)
```

Computing a full Jacobian matrix using `vjp()`

Post-multiplying a matrix with the ith standard basis vector reveals the ith column of the matrix, so can construct the full $m \times n$ Jacobian matrix with m VJPs:

```python
def Jf_VJPs(x): # compute the full Jacobian at x using VJPs
    Jf = np.zeros(shape = (3, 2))
    for i in range(3):
        # compute the ith basis vector
        ei = np.zeros(shape = 3)
        ei[i] = 1

        # set the ith column equal to JVP with v = ej
        Jf[i] = vjp(f, x)[1](ei)[0]
    return Jf

x = jnp.array([2., 3.])

print(f"Jf with VJPs: \n {Jf_VJPs(x)}")
print(f"Jf manual: \n {Jf_manual(x)}")
```

✓ 0.1s Python

Computing a full Jacobian matrix using `vjp()`

Post-multiplying a matrix with the ith standard basis vector reveals the ith column of the matrix, so can construct the full $m \times n$ Jacobian matrix with m VJPs:

```python
def Jf_VJPs(x): # compute the full Jacobian at x using VJPs
    Jf = np.zeros(shape = (3, 2))
    for i in range(3):
        # compute the ith basis vector
        ei = np.zeros(shape = 3)
        ei[i] = 1

        # set the ith column equal to JVP with v = ej
        Jf[i] = vjp(f, x)[1](ei)[0]
    return Jf

x = jnp.array([2., 3.])

print(f"Jf with VJPs: \n {Jf_VJPs(x)}")
print(f"Jf manual: \n {Jf_manual(x)}")
```

✓ 0.1s Python

```
Jf with VJPs:
 [[ 1.          2.        ]
 [-8.35853291 18.26372719]
 [ 0.         27.        ]]
Jf manual:
 [[ 1.        2.      ]
 [-8.358533 18.263727]
 [ 0.       27.      ]]
```

Reverse-mode differentiation: `jacrev()`

```python
from jax import jacrev

print(f"Jf with jacrev(): \n {jacrev(f)(x)}")
```

✓ 0.3s Python

```
Jf with jacrev():
 [[ 1.         2.       ]
 [-8.358533 18.263727]
 [ 0.        27.       ]]
```

Reverse-mode differentiation: `jacrev()`

```python
from jax import jacrev

print(f"Jf with jacrev(): \n {jacrev(f)(x)}")
```

✓ 0.3s Python

```
Jf with jacrev():
 [[ 1.          2.        ]
 [-8.358533 18.263727]
 [ 0.         27.        ]]
```

…better for short and fat Jacobians

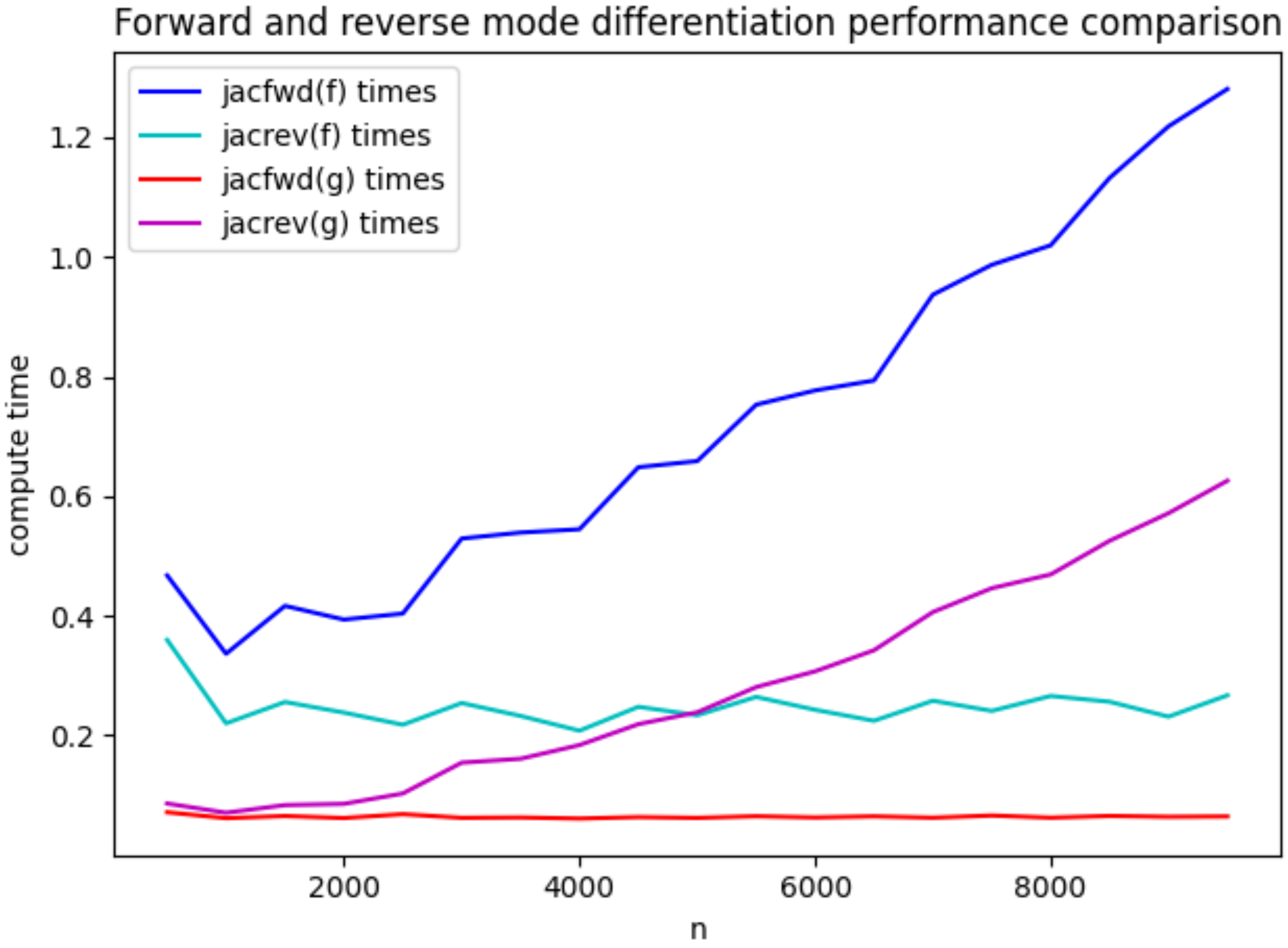
Forward vs. Reverse Mode Performance Comparison

$$f(x) = \left\| \frac{e^x - x^2}{\tan x} \right\|, x \in \mathbb{R}^n$$

$$g(t) = t^{\sin(1, t, t^2, \ldots, t^{n-1})}, t \in \mathbb{R}$$

Forward vs. Reverse Mode Performance Comparison

$$f(x) = \left\| \frac{e^x - x^2}{\tan x} \right\|, x \in \mathbb{R}^n \qquad g(t) = t^{\sin(1,t,t^2,\ldots,t^{n-1})}, t \in \mathbb{R}$$



Performance

JAX comes with JIT compilation

JAX comes with JIT compilation

- Automatic differentiation involves *tracing* a sequence of arithmetic computations to build up the computation graph

JAX comes with JIT compilation

- Automatic differentiation involves *tracing* a sequence of arithmetic computations to build up the computation graph

- Jitting (just-in-time compilation) compiles the computation graph and makes things run much faster

JAX comes with JIT compilation

- Automatic differentiation involves *tracing* a sequence of arithmetic computations to build up the computation graph

- Jitting (just-in-time compilation) compiles the computation graph and makes things run much faster

    - Jitting traces through the entire computation (through function calls), so only need to jit outermost function

JAX comes with JIT compilation

- Automatic differentiation involves *tracing* a sequence of arithmetic computations to build up the computation graph
- Jitting (just-in-time compilation) compiles the computation graph and makes things run much faster
    - Jitting traces through the entire computation (through function calls), so only need to jit outermost function
- Two limitations

JAX comes with JIT compilation

- Automatic differentiation involves *tracing* a sequence of arithmetic computations to build up the computation graph
- Jitting (just-in-time compilation) compiles the computation graph and makes things run much faster
  - Jitting traces through the entire computation (through function calls), so only need to jit outermost function
- Two limitations
  - Control flow must be agnostic to valued of traced variables

JAX comes with JIT compilation

- Automatic differentiation involves *tracing* a sequence of arithmetic computations to build up the computation graph
- Jitting (just-in-time compilation) compiles the computation graph and makes things run much faster
  - Jitting traces through the entire computation (through function calls), so only need to jit outermost function
- Two limitations
  - Control flow must be agnostic to valued of traced variables
  - Requires array shapes of functions to be known ahead of time

JIT example, simple function

$$f(x) = \sigma\left(\| \hat{I} - I \|_2^2 \right), I \in \mathbb{R}^{128\times128}$$

JIT example, simple function

$$f(x) = \sigma\left(\| \hat{I} - I \|_2^2 \right), I \in \mathbb{R}^{128\times128}$$

```python
def f_np(Ihat, I):
  diff = np.linalg.norm(Ihat - I) **2 / 2
  return 1 / (1 + np.exp(-diff))


def f_jnp(Ihat, I):
  diff = jnp.linalg.norm(Ihat - I) **2 / 2
  return 1 / (1 + jnp.exp(-diff))

f_jnp_jitted = jit(f_jnp)
```

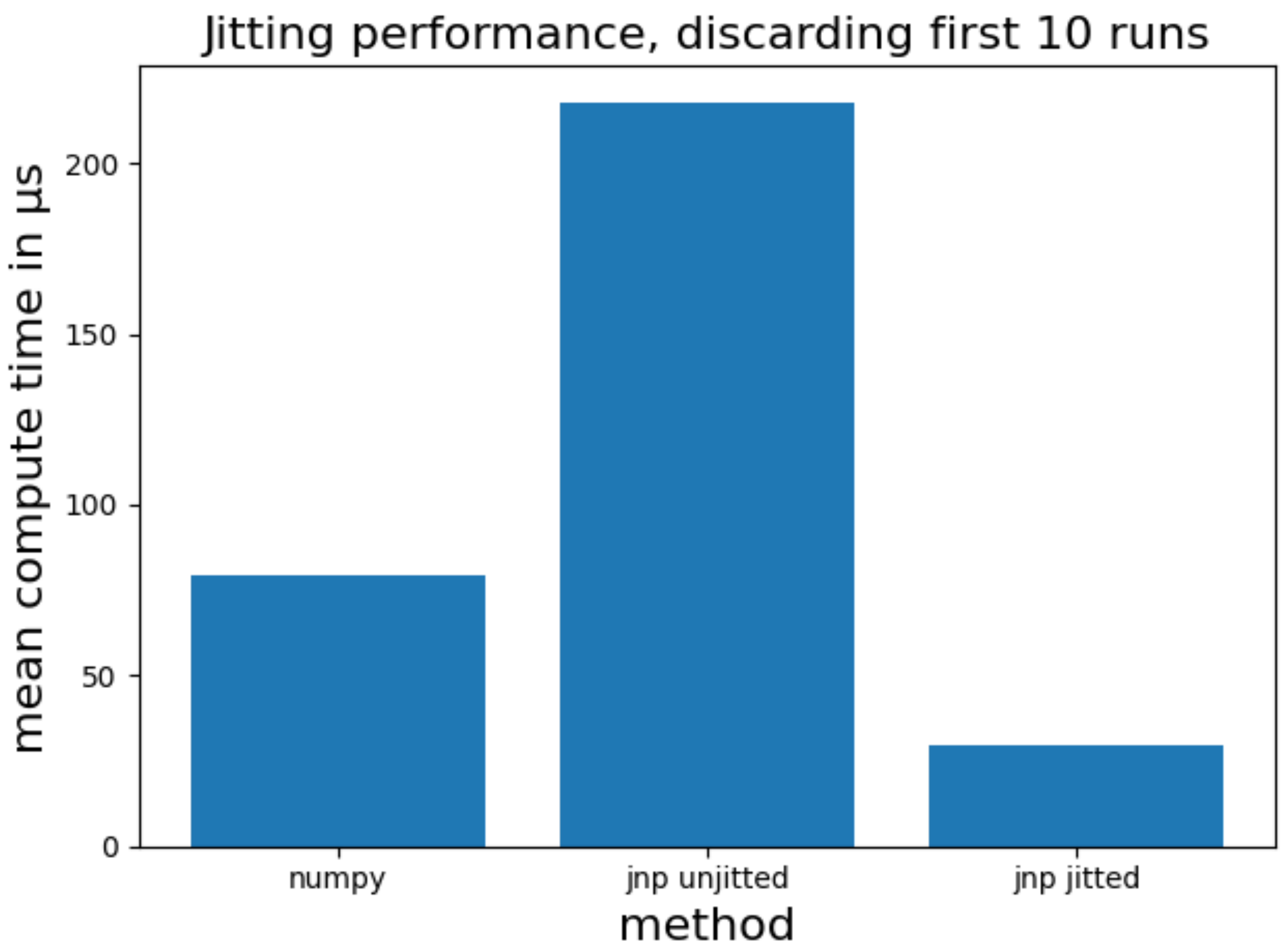
JIT example, simple function

$$f(x) = \sigma\left(\| \hat{I} - I \|_2^2 \right), I \in \mathbb{R}^{128\times128}$$

```python
def f_np(Ihat, I):
  diff = np.linalg.norm(Ihat - I) **2 / 2
  return 1 / (1 + np.exp(-diff))


def f_jnp(Ihat, I):
  diff = jnp.linalg.norm(Ihat - I) **2 / 2
  return 1 / (1 + jnp.exp(-diff))

f_jnp_jitted = jit(f_jnp)
```



JIT example, Bezier rendering function

$$\mathbf{B}(t) = (1-t)^3\mathbf{P_0} + 3(1-t)^2t\mathbf{P_1} + 3(1-t)t^2\mathbf{P_2} + t^3\mathbf{P_3},$$

$t \in [0,1]$

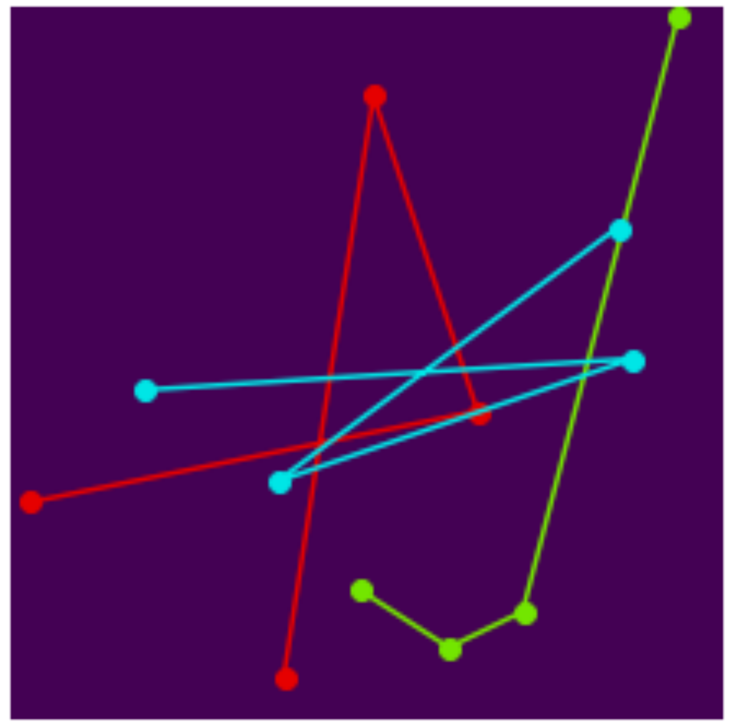
$\mathbf{P_i} \in \mathbb{R}^2$, control points

JIT example, Bezier rendering function

$$\mathbf{B}(t) = (1-t)^3\mathbf{P_0} + 3(1-t)^2t\mathbf{P_1} + 3(1-t)t^2\mathbf{P_2} + t^3\mathbf{P_3},$$

$t \in [0,1]$

$\mathbf{P_i} \in \mathbb{R}^2$, control points



Control Points

$\xmapsto{f(P)}$

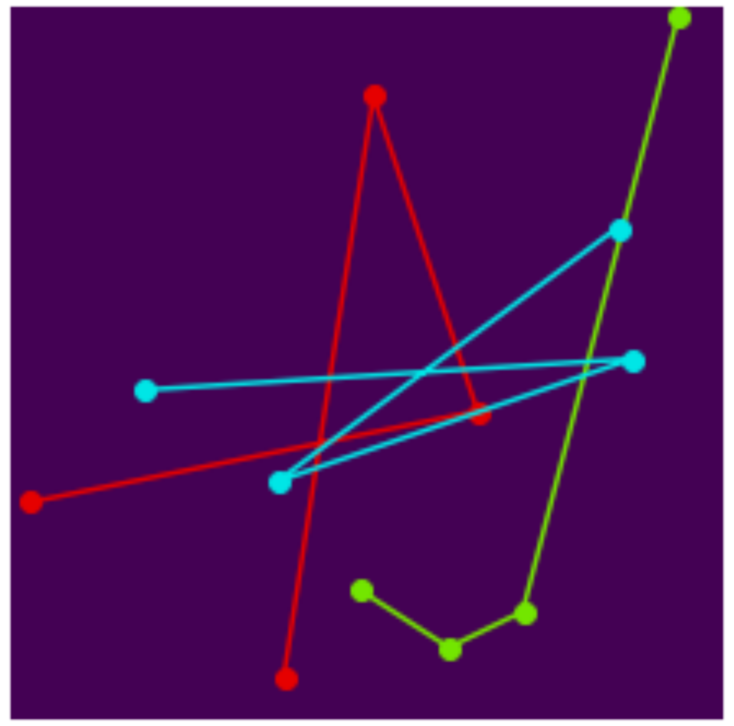
Rendered Image

JIT example, Bezier rendering function

$$\mathbf{B}(t) = (1-t)^3\mathbf{P_0} + 3(1-t)^2 t\mathbf{P_1} + 3(1-t)t^2\mathbf{P_2} + t^3\mathbf{P_3},$$

$t \in [0,1]$

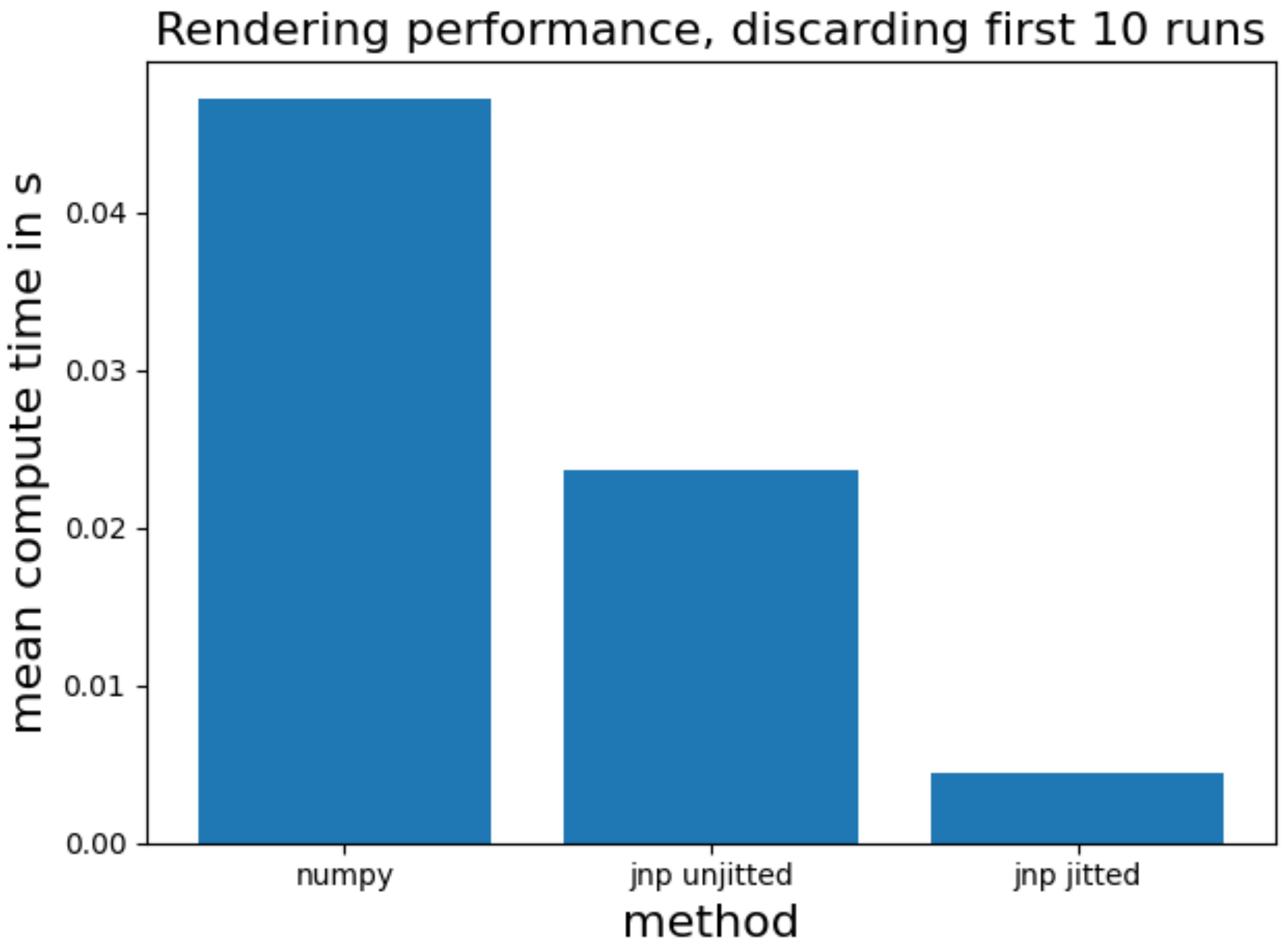
$\mathbf{P_i} \in \mathbb{R}^2$, control points



Control Points

$\xrightarrow{f(P)}$

Rendered Image



Control flow can’t depend on values of inputs

```python
from jax import jit

def add_one_or_two(x, add_one):
  if add_one: # control flow depends on the argument add_one
    return x + 1
  else:
    return x + 2

add_one_or_two_jitted = jit(add_one_or_two)
add_one_or_two_jitted(1.0, True)
```

1.5s

Python

Control flow can’t depend on values of inputs

```python
from jax import jit

def add_one_or_two(x, add_one):
    if add_one: # control flow depends on the argument add_one
        return x + 1
    else:
        return x + 2

add_one_or_two_jitted = jit(add_one_or_two)
add_one_or_two_jitted(1.0, True)
```

1.5s Python

```
---------------------------------------------------------------------------
ConcretizationTypeError                   Traceback (most recent call last)
/data/philliplo125/jax-tutorial/tutorial.ipynb Cell 41' in <cell line: 10>()
      7     return x + 2
      9 add_one_or_two_jitted = jit(add_one_or_two)
---> 10 add_one_or_two_jitted(1.0, True)

    [... skipping hidden 14 frame]

/data/philliplo125/jax-tutorial/tutorial.ipynb Cell 41' in add_one_or_two(x, add_one)
      3 def add_one_or_two(x, add_one):
----> 4   if add_one: # control flow depends on the argument add_one
      5     return x + 1
      6   else:

    [... skipping hidden 1 frame]

File ~/anaconda3/envs/jax-tutorial/lib/python3.9/site-packages/jax/core.py:1123, in concretization_function_error.<locals>.error(self, arg)
   1122 def error(self, arg):
-> 1123   raise ConcretizationTypeError(arg, fname_context)

ConcretizationTypeError: Abstract tracer value encountered where concrete value is expected: Traced<ShapedArray(bool[], weak_type=True)>with<DynamicJaxprTrace(level=0/1)>
The problem arose with the `bool` function.
While tracing the function add_one_or_two at /tmp/ipykernel_1263005/1343836691.py:3 for jit, this concrete value was not available in Python because it depends on the value of the arg
ument 'add_one'.

See https://jax.readthedocs.io/en/latest/errors.html#jax.errors.ConcretizationTypeError
```

Use `static_argnums`

```python
def add_one_or_two(x, add_one):
    if add_one: # control flow depends on the argument add_one
        return x + 1
    else:
        return x + 2

add_one_or_two_jitted = jit(add_one_or_two, static_argnums = 1)
add_one_or_two_jitted(1.0, True)
```

✓ 0.1s Python

```
DeviceArray(2., dtype=float32, weak_type=True)
```

Array shapes can't depend on values of inputs

```python
def stupid_mult(a, b): # multiply two integers a and b in a very dumb way
    bs = b * np.ones(shape = (a,)) # the shape of bs depends on a!
    ab = np.sum(bs)

    return ab

stupid_mult_jitted = jit(stupid_mult)
stupid_mult_jitted(3, 4)
```

0.9s

Python

Array shapes can’t depend on values of inputs

```python
def stupid_mult(a, b): # multiply two integers a and b in a very dumb way
    bs = b * np.ones(shape = (a,)) # the shape of bs depends on a!
    ab = np.sum(bs)

    return ab

stupid_mult_jitted = jit(stupid_mult)
stupid_mult_jitted(3, 4)
```

0.9s Python

```
---------------------------------------------------------------------------
TracerIntegerConversionError              Traceback (most recent call last)
/data/philliplo125/jax-tutorial/tutorial.ipynb Cell 45' in <cell line: 8>()
      5     return ab
      7 stupid_mult_jitted = jit(stupid_mult)
----> 8 stupid_mult_jitted(3, 4)

    [... skipping hidden 14 frame]

/data/philliplo125/jax-tutorial/tutorial.ipynb Cell 45' in stupid_mult(a, b)
      1 def stupid_mult(a, b): # multiply two integers a and b in a very dumb way
----> 2     bs = b * np.ones(shape = (a,)) # the shape of bs depends on a!
      3     ab = np.sum(bs)
      5     return ab

File ~/anaconda3/envs/jax-tutorial/lib/python3.9/site-packages/numpy/core/numeric.py:204, in ones(shape, dtype, order, like)
    201 if like is not None:
    202     return _ones_with_like(shape, dtype=dtype, order=order, like=like)
--> 204 a = empty(shape, dtype, order)
    205 multiarray.copyto(a, 1, casting='unsafe')
    206 return a

File ~/anaconda3/envs/jax-tutorial/lib/python3.9/site-packages/jax/core.py:519, in Tracer.__index__(self)
    518 def __index__(self):
--> 519     raise TracerIntegerConversionError(self)

TracerIntegerConversionError: The __index__() method was called on the JAX Tracer object Traced<ShapedArray(int32[], weak_type=True)>with<DynamicJaxprTrace(level=0/1)>
See https://jax.readthedocs.io/en/latest/errors.html#jax.errors.TracerIntegerConversionError
```

Array shapes can't depend on values of inputs

```python
def stupid_mult(a, b): # multiply two integers a and b in a very dumb way
    bs = b * np.ones(shape = (a,)) # the shape of bs depends on a!
    ab = np.sum(bs)

    return ab

stupid_mult_jitted = jit(stupid_mult)
stupid_mult_jitted(3, 4)
```

⊗ 0.9s Python

```
---------------------------------------------------------------------------
TracerIntegerConversionError              Traceback (most recent call last)
/data/phillilo125/jax-tutorial/tutorial.ipynb Cell 45' in <cell line: 8>()
      5     return ab
      7 stupid_mult_jitted = jit(stupid_mult)
----> 8 stupid_mult_jitted(3, 4)

    [... skipping hidden 14 frame]

/data/phillilo125/jax-tutorial/tutorial.ipynb Cell 45' in stupid_mult(a, b)
      1 def stupid_mult(a, b): # multiply two integers a and b in a very dumb way
----> 2     bs = b * np.ones(shape = (a,)) # the shape of bs depends on a!
      3     ab = np.sum(bs)
      5     return ab

File ~/anaconda3/envs/jax-tutorial/lib/python3.9/site-packages/numpy/core/numeric.py:204, in ones(shape, dtype, order, like)
    201 if like is not None:
    202     return _ones_with_like(shape, dtype=dtype, order=order, like=like)
--> 204 a = empty(shape, dtype, order)
    205 multiarray.copyto(a, 1, casting='unsafe')
    206 return a

File ~/anaconda3/envs/jax-tutorial/lib/python3.9/site-packages/jax/core.py:519, in Tracer.__index__(self)
    518 def __index__(self):
--> 519   raise TracerIntegerConversionError(self)

TracerIntegerConversionError: The __index__() method was called on the JAX Tracer object Traced<ShapedArray(int32[], weak_type=True)>with<DynamicJaxprTrace(level=0/1)>
See https://jax.readthedocs.io/en/latest/errors.html#jax.errors.TracerIntegerConversionError
```

```python
stupid_mult_jitted = jit(stupid_mult, static_argnums = 0)
stupid_mult_jitted(3, 4)
```

✓ 0.1s Python

Array shapes can't depend on values of inputs

```python
def stupid_mult(a, b): # multiply two integers a and b in a very dumb way
  bs = b * np.ones(shape = (a,)) # the shape of bs depends on a!
  ab = np.sum(bs)

  return ab

stupid_mult_jitted = jit(stupid_mult)
stupid_mult_jitted(3, 4)
```

⊗ 0.9s Python

```
---------------------------------------------------------------------------
TracerIntegerConversionError              Traceback (most recent call last)
/data/phillipl0125/jax-tutorial/tutorial.ipynb Cell 45' in <cell line: 8>()
      5   return ab
      7 stupid_mult_jitted = jit(stupid_mult)
----> 8 stupid_mult_jitted(3, 4)

    [... skipping hidden 14 frame]

/data/phillipl0125/jax-tutorial/tutorial.ipynb Cell 45' in stupid_mult(a, b)
      1 def stupid_mult(a, b): # multiply two integers a and b in a very dumb way
----> 2   bs = b * np.ones(shape = (a,)) # the shape of bs depends on a!
      3   ab = np.sum(bs)
      5   return ab

File ~/anaconda3/envs/jax-tutorial/lib/python3.9/site-packages/numpy/core/numeric.py:204, in ones(shape, dtype, order, like)
    201 if like is not None:
    202     return _ones_with_like(shape, dtype=dtype, order=order, like=like)
--> 204 a = empty(shape, dtype, order)
    205 multiarray.copyto(a, 1, casting='unsafe')
    206 return a

File ~/anaconda3/envs/jax-tutorial/lib/python3.9/site-packages/jax/core.py:519, in Tracer.__index__(self)
    518 def __index__(self):
--> 519   raise TracerIntegerConversionError(self)

TracerIntegerConversionError: The __index__() method was called on the JAX Tracer object Traced<ShapedArray(int32[], weak_type=True)>with<DynamicJaxprTrace(level=0/1)>
See https://jax.readthedocs.io/en/latest/errors.html#jax.errors.TracerIntegerConversionError
```

```python
stupid_mult_jitted = jit(stupid_mult, static_argnums = 0)
stupid_mult_jitted(3, 4)
```

✓ 0.1s Python

```
DeviceArray(12., dtype=float32)
```

Automatic vectorization with `vmap()`

Compute Frobenius norm of one hundred 1000 x 1000 matrices

```python
def frob_norm(A):
  '''
  Compute the Frobenus norm of a matrix A
  '''
  return jnp.linalg.norm(A)

def manual_batched_frob_norm(As):
  n = As.shape[0]
  norms = jnp.zeros(shape = As.shape[0])
  for i in range(n):
    norms = norms.at[i].set(frob_norm(As[i]))
  return norms

vmap_batched_frob_norm = vmap(frob_norm)

jitted_vmap_batched_frob_norm = jit(vmap_batched_frob_norm)
```

Automatic vectorization with `vmap()`

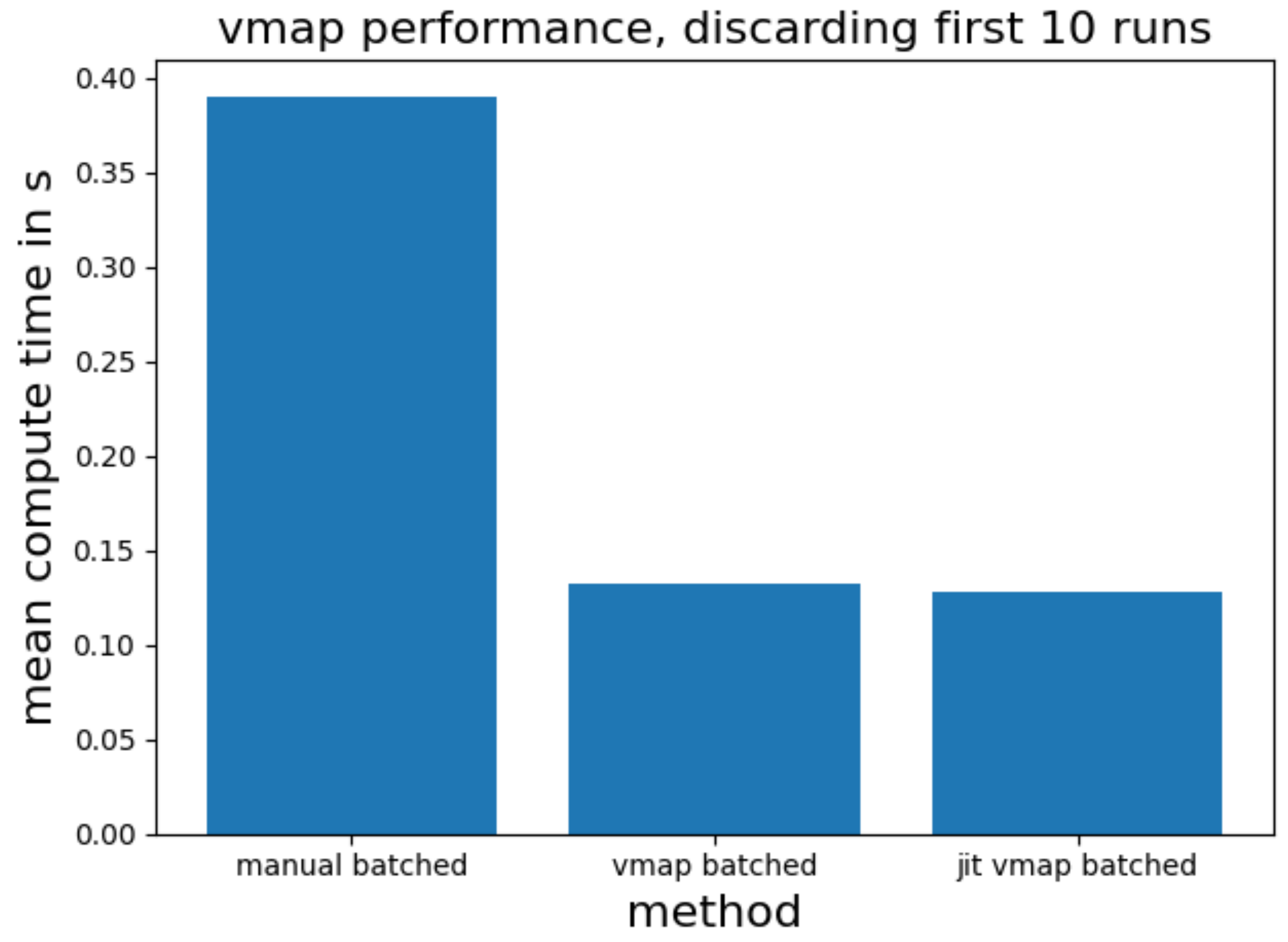
Compute Frobenius norm of one hundred 1000 x 1000 matrices

```python
def frob_norm(A):
  '''
  Compute the Frobenus norm of a matrix A
  '''
  return jnp.linalg.norm(A)

def manual_batched_frob_norm(As):
  n = As.shape[0]
  norms = jnp.zeros(shape = As.shape[0])
  for i in range(n):
    norms = norms.at[i].set(frob_norm(As[i]))
  return norms

vmap_batched_frob_norm = vmap(frob_norm)

jitted_vmap_batched_frob_norm = jit(vmap_batched_frob_norm)
```



Other things you can do with JAX

Other things you can do with JAX

- Can arbitrarily compose forward and reverse mode differentiation

Other things you can do with JAX

- Can arbitrarily compose forward and reverse mode differentiation
- Can compute derivatives for functions between tensors of arbitrary shape

Other things you can do with JAX

- Can arbitrarily compose forward and reverse mode differentiation
- Can compute derivatives for functions between tensors of arbitrary shape
- Deep learning library: flax

Other things you can do with JAX

- Can arbitrarily compose forward and reverse mode differentiation
- Can compute derivatives for functions between tensors of arbitrary shape
- Deep learning library: flax
- Can run transparently on GPUs and TPUs, compiled with XLA

Other things you can do with JAX

- Can arbitrarily compose forward and reverse mode differentiation
- Can compute derivatives for functions between tensors of arbitrary shape
- Deep learning library: flax
- Can run transparently on GPUs and TPUs, compiled with XLA
- Repo: https://github.com/PhillipLo/jax-tutorial