Google JAX

John Stachurski

2025

Code

In this lecture series we will code in Python

Favorite libraries

- Google JAX
- Google JAX
- Google JAX...

Code

In this lecture series we will code in Python

Favorite libraries

- Google JAX
- Google JAX
- Google JAX...

Topics

- Scientific computing: history and background
- JIT compilation
- Autodiff
- Array operations
- Functional programming

History: Setting the stage

Before we can understand JAX, we need to know a bit about the history of scientific computing

Let's recall some of the major paradigms and ideas:

- Languages and compilers
- Dynamic and static types
- Background on vectorization / JIT compilers

Fortran / C — static types and AOT compilers

Example. Suppose we want to compute the sequence

$$k_{t+1} = s k_t^{\alpha} + (1 - \delta) k_t$$

from some given k_0

Let's write a function in C that

1. implements the loop
2. returns the last k_t

```c
int main() {
    double k = 0.2;
    double alpha = 0.4;
    double s = 0.3;
    double delta = 0.1;
    int i;
    int n = 1000;
    for (i = 0; i < n; i++) {
        k = s * pow(k, alpha) + (1 - delta) * k;
    }
    printf("k = %f\n", k);
}
```

First we compile the whole program (ahead-of-time compilation):

```
❯❯ gcc solow.c -o out -lm
```

Now we execute:

```
❯❯ ./out
x = 6.240251
```

Pros

- fast arithmetic
- fast loops

Cons

- slow to write
- lack of portability
- hard to debug
- hard to parallelize
- low interactivity

For comparison, the same operation in Python:

```python
α = 0.4
s = 0.3
δ = 0.1
n = 1_000
k = 0.2

for i in range(n-1):
    k = s * k**α + (1 - δ) * k

print(k)
```

Python is **interpreted** rather than compiled

- code is executed statement by statement
- data types are queried on the fly
- arithmetic operations require method resolution

Pros

- easy to write
- high portability
- immediate feedback — high interactivity
- easy to debug

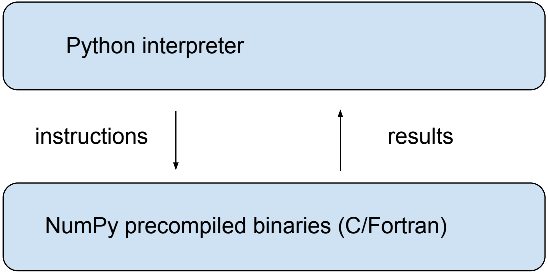
Cons

- slow

So how can we get

good execution speeds **and** high productivity / interactivity?

Python + NumPy

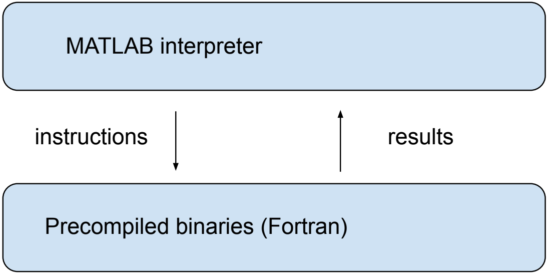


```python
import numpy

A = ((2.0, -1.0),
     (5.0, -0.5))

b = (0.5, 1.0)

A, b = np.array(A), np.array(b)

x = np.inv(A) @ b
```

1. Arrays defined with high-level commands
   - (Python / NumPy API)
2. Execution takes place in an efficient low-level environment
   - Efficient machine code (compiled C / Fortran)
3. Results are returned to the high-level interface

MATLAB

NumPy is similar to and borrows from the older MATLAB programming environment



```
A = [2.0, -1.0
     5.0, -0.5];

b = [0.5, 1.0]';

x = inv(A) * b
```

Advantages of NumPy / MATLAB

- Operations are passed to specialized machine code
- Type-checking is paid per array, not per array element

Disadvantages

- Can be highly memory intensive (intermediate arrays)
- Fails to specialize on array **shapes**
- Limited — how would you accelerate the Solow code using NumPy?

Julia — rise of the JIT compilers

Can do MATLAB / NumPy style vectorized operations

```
A = [2.0  -1.0
     5.0  -0.5]

b = [0.5  1.0]'

x = inv(A) * b
```

But also has fast loops via an efficient JIT compiler

Example. Suppose, again, that we want to compute

$$k_{t+1} = sk_t^{\alpha} + (1 - \delta)k_t$$

from some given k_0

- Iterative, not easily vectorized

```julia
function solow(k0, α=0.4, δ=0.1, n=1_000)
    k = k0
    for i in 1:(n-1)
        k = s * k^α + (1 - δ) * k
    end
    return k
end

solow(0.2)  # JIT-compiled at first call
```

Julia accelerates solow at runtime via a JIT compiler

Pros:

- fast execution — assuming correct type inference
- dynamically typed...(but compiler wants type stability)
- close to the maths

Cons:

- Everything compiled might not be optimal
  - debugging is more challenging
  - slow first runs
- Package instability
- Repeated breaking changes

Python + Numba — same architecture, same speed

```python
from numba import jit

@jit(nopython=True)
def solow(k0, α=0.4, δ=0.1, n=1_000):
    k = k0
    for i in range(n-1):
        k = s * k**α + (1 - δ) * k
    return k

solow(0.2)
```

Runs at same speed as Julia / C / Fortran

OK, let's talk about the next generation...

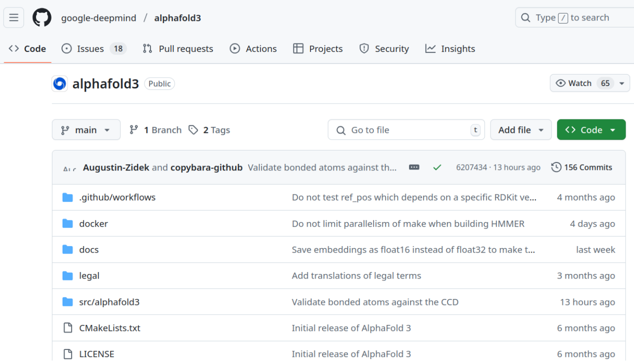
https://jax.readthedocs.io/en/latest/

A high-performance numerical computing library

- Developed by Google Research (prev. Google Brain)
- NumPy-style API for array operations
- GPU/TPU acceleration
- Automatic differentiation
- Math-centric library semantics
- Rising popularity among ML researchers

"The JAX compiler aims to enable researchers to write Python programs...that are automatically compiled and scaled to leverage accelerators and supercomputers"

Example. AlphaFold3 is built with Google JAX



Highly accurate protein structure prediction with AlphaFold

John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool,...

Nature Vol. 596 (2021)

- Citation count = 35K
- Nobel Prize in Chemistry 2024

"The acronym JAX stands for Just After eXecution"

- monitor function execution once and then compile

Another acronym:

- Just-in-time compilation
- Automatic differentiation
- XLA (accelerated linear algebra)

Familiar NumPy-style array API

```python
import jax.numpy as jnp

A = ((2.0, -1.0),
     (5.0, -0.5))

b = (0.5, 1.0)

A, b = jnp.array(A), jnp.array(b)

x = jnp.inv(A) @ b
```

Implicit JIT via the XLA pipeline

The sequence of actions for performing `jnp.inv(A)` are as follows:

1. JAX identifies that it needs to invert a matrix `A` of specific data type and shape
2. JAX passes this information to XLA in an intermediate representation
3. XLA generates compiled code specialized to your hardware, the data type and shape of the array
4. The code is executed on the device and the result is returned to the user
5. The code is cached in memory for future use (when called again with the same specific dtype and shape)

Explicit just-in-time compilation

We can also explicitly JIT compile JAX functions

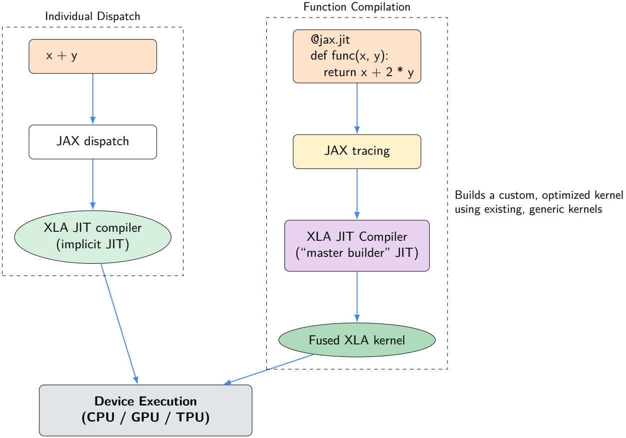
```
@jax.jit
def f(x):
    term1 = 2 * jnp.sin(3 * x) * jnp.cos(x/2)
    term2 = 0.5 * x**2 * jnp.cos(5*x) / (1 + 0.1 * x**2)
    term3 = 3 * jnp.exp(-0.2 * (x - 4)**2) * jnp.sin(10*x)
    return term1 + term2 + term3
```

- Compiles at first call (e.g., `result = f(x)`)
- Compiler specializes on both shape and data type

Compiler tools for optimizing function operations:

- Operations combined into fused kernels for GPU/TPU
- Eliminate intermediate buffers / memory writes and reads
- Loop unrolling
- Specialized algorithms
- Memory layout optimization for multi-dimensional arrays

Implicit and explicit JIT



Automatic differentiation

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

def f(θ, x):
  for W, b in θ:
    w = x @ W + b
    x = jnp.tanh(w)
  return x

def loss(θ, x, y):
  return jnp.sum((y - f(θ, x))**2)

grad_loss = jit(grad(loss))  # Now use gradient descent
```

More features of JAX

Let's review some other features

- Functional programming
- PyTrees

Functional Programming

JAX adopts a functional programming style

$\Longrightarrow$ Functions are **pure**

```
def f(θ, x):
  for W, b in θ:
    w = W @ x + b
    x = jnp.tanh(w)
  return x

def loss(θ, x, y):
  return jnp.sum((y - f(θ, x))**2)
```

Pure functions:

1. **Deterministic**
2. **No side effects**

Deterministic means

- Same input $\implies$ same output
- Outputs do not depend on global state

No side effects

- Won't change global state
- Won't modify data passed to the function
- Typically use immutable data

A non-pure function

```python
tax_rate = 0.1
prices = [10.0, 20.0]

def add_tax(prices):
    for i, price in enumerate(prices):
        prices[i] = price * (1 + tax_rate)
    print('Modified prices: ', prices)
    return prices
```

Why is this not pure?

A pure function

```
tax_rate = 0.1
prices = (10.0, 20.0)

def add_tax_pure(prices, tax_rate):
    return [price * (1 + tax_rate) for price in prices]
```

General advantages:

- Helps testing: each function can operate in isolation
- Promotes deterministic behavior and hence reproducibility
- Prevents bugs that arise from mutating shared state

Advantages for JAX:

- Data dependencies are explicit, which helps with optimizing complex computations
- Pure functions are easier to differentiate (autodiff)
- Pure functions are easier to parallelize and optimize (don't depend on shared mutable state)
- Transformations can be composed cleanly

In summary, functional programming is good for

- JIT, autodiff, & parallelization

JAX PyTrees

Consider a function of the form

$$f_\theta = G_m \circ G_{m-1} \circ \cdots \circ G_2 \circ G_1$$

where

- $G_\ell x = \sigma_\ell(x W_\ell + b_\ell)$ for $\ell = 1, \ldots, m$
- θ represents the "vector" of all parameters
- σ_ℓ is a given function

The idea that the vector θ contains all parameters is conceptually useful but awkward within code...

To handle these kinds of situations we can use PyTrees

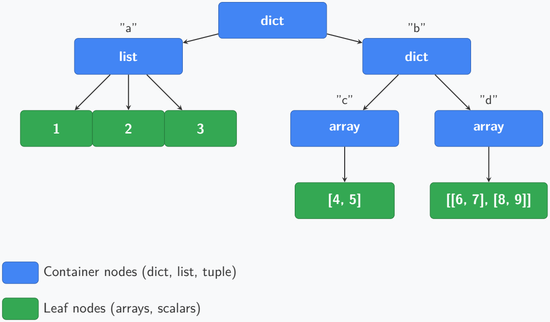
- A tree-like data structure built from Python containers
- A concept, not a data type
- Used to store parameters

Examples.

- A list of dictionaries, each dictionary contains parameters
- A dictionary of lists
- A dictionary of lists of dictionaries
- etc.

JAX PyTree Structure

```
pytree = {
    "a": [1, 2, 3],
    "b": {"c": jnp.array([4, 5]), "d": jnp.array([[6, 7], [8, 9]])}
}
```



JAX can

- apply functions to all leaves in a PyTree structure
- differentiate functions with respect to the leaves of PyTrees
- etc.

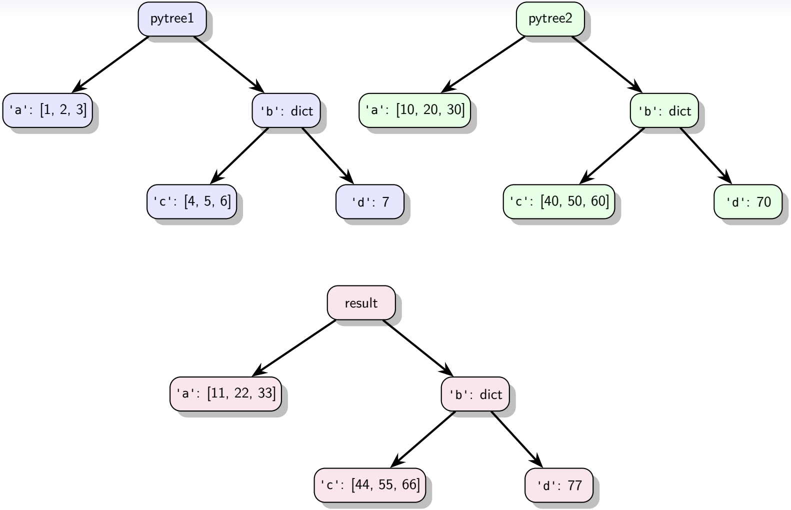


Figure: `jax.tree.map(lambda x, y: x + y, pytree1, pytree2)`

```python
# Apply gradient updates to all parameters
def sgd_update(params, grads, learning_rate):
    return jax.tree.map(
        lambda p, g: p - learning_rate * g,
        params,
        grads
    )

# Calculate gradients (PyTree with same structure as params)
loss_grad = jax.grad(loss_fn)
grads = loss_grad(params, x, y)

# Update all parameters at once
updated_params = sgd_update(params, grads, 0.01)
```

Summary

Advantages over NumPy / MATLAB

- Machine code specialized to data types, shapes and devices!
- Automatically matches tasks with accelerators
- Same code, multiple backends (CPUs, GPUs, TPUs)
- Can fuse array operations for speed and memory efficiency
- Elegant functional style
- Integrated efficient autodiff

Advantages of JAX (vs PyTorch / Tensorflow / etc.) for economists:

- elegant functional programming style – close to maths
- elegant autodiff tools
- array operations follow standard NumPy API

Exposes low level functions