Google JAX

John Stachurski

2025

Code

In this lecture series we will code in Python

Favorite libraries

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

Code

In this lecture series we will code in Python

Favorite libraries

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

Topics

- 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:

- Some history of scientific computing
- 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:

```
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)
```

Pros

- easy to write
- high portability
- easy to debug
- high interactivity

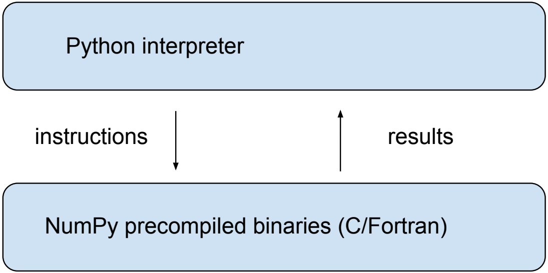
Cons

- slow

So how can we get

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

Python + NumPy

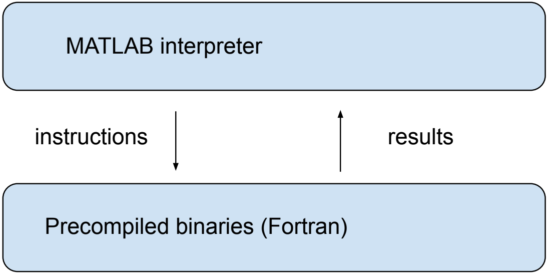


```python
import numpy

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 are defined in a high-level interface
2. Execution takes place in an efficient low-level environment
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 to 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**
- What if we can't convert problems to array-processing operations?

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

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

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)
```

Julia accelerates `solow` at runtime via a JIT compiler

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

What's JAX?

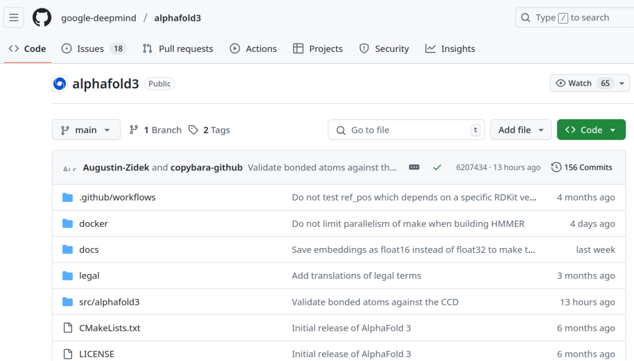
https://jax.readthedocs.io/en/latest/

A high-performance numerical computing library

- Developed by Google Research (prev. Google Brain)
- Easy-to-use NumPy-style API for array operations
- Simple 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 = 30K
- Nobel Prize in Chemistry 2024

"The acronym JAX stands for Just After eXecution, since to compile a function we first monitor its execution once in Python."

- Just-in-time compilation
- Automatic differentiation
- Accelerated linear algebra

- significant advantages over C / Fortran / NumPy / Julia / Numba
- But is not uniformly "better" — discuss later

"The acronym JAX stands for Just After eXecution, since to compile a function we first monitor its execution once in Python."

- Just-in-time compilation
- Automatic differentiation
- Accelerated linear algebra

- significant advantages over C / Fortran / NumPy / Julia / Numba
- But is not uniformly "better" — discuss later

Familiar NumPy-style array API

```
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
```

Just-in-time compilation

```
@jax.jit
def f(x):
    """
    A function that transforms an array 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 runtime based on specified shape & data type

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
```

Accelerated linear algebra (XLA)

Array operations are

- JIT-compiled
- automatically parallelized
- automatically optimized for and deployed to available hardware

Advantages over NumPy / MATLAB

- Can specialize machine code to data types **and** shapes!
- Automatically matches tasks with accelerators (GPU / TPU)
- Fuses array operations for speed and memory efficiency

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

- exposes low level functions
- elegant functional programming style – close to maths
- elegant autodiff tools
- array operations follow standard NumPy API
- automatic parallelization
- same code, multiple backends (CPUs, GPUs, TPUs)

Features of JAX

Let's look at some useful features

Functional Programming

JAX adopts a **functional programming style**

Key feature: Functions are pure

- Deterministic: same input $\implies$ same output
- Have no side effects (don't modify state outside their scope)

A non-pure function

```python
tax_rate = 0.1  # Global
price = 10.0    # Global

def add_tax_non_pure():
    global price
    # The next line both accesses and modifies global state
    price = price * (1 + tax_rate)
    return price
```

A pure function

```python
def add_tax_non_pure(price, tax_rate=0.1):
    price = price * (1 + tax_rate)
    return price
```

General advantages:

- Helps testing: each function can operate in isolation
- Data dependencies are explicit, which helps with understanding and optimizing complex computations
- Promotes deterministic behavior and hence reproducibility
- Prevents subtle bugs that arise from mutating shared state

Advantages for JAX:

- Pure functions are easier to differentiate (don't modify external state)
- Pure functions are easier to parallelize and optimize (don't depend on shared mutable state)
- Transformations can be composed cleanly (predictable results)

JAX PyTrees

A PyTree is a concept in the JAX library that refers to a tree-like data structure built from Python containers.

Examples.

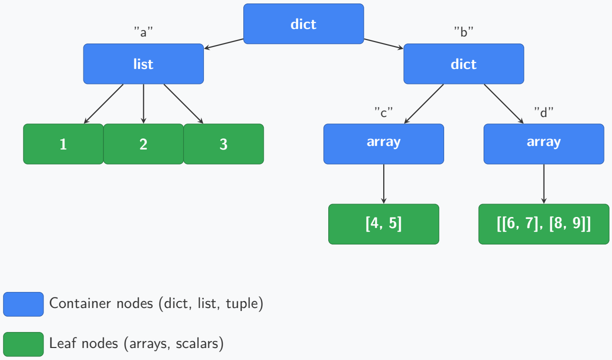
- A dictionary of lists of parameters
- A list of dictionaries of parameters, etc.

JAX can

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

JAX PyTree Structure

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



```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)
grads = jax.grad(loss_fn)(params, inputs, targets)

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