

# Dynamic Programming with Google JAX

CEF 2024

John Stachurski

June 2024

# Topics

- Types of programming languages
- Vectorization vs JIT compilers
- Motivation for JAX
- Intro to JAX
- JAX for DP

Target audience: people who are new to / curious about JAX

Please feel free to question / debate / share your experiences

Slides, code:

[https://github.com/QuantEcon/cef\\_2024\\_singapore](https://github.com/QuantEcon/cef_2024_singapore)

Quick poll:

- Python programmers?
  - NumPy? Numba? PyTorch? JAX?
- Julia?
- MATLAB?
- C?
- Fortran?

Regular GPU users?

# Old school: static types & AOT compilers

**Example.** Consider the Solow model

$$k_{t+1} = sk_t^\alpha + (1 - \delta)k_t \quad \text{with } k_0 \text{ given}$$

Fortran code:

---

```
program main
  implicit none
  integer, parameter :: dp=kind(0.d0)
  integer :: n=1000
  real(dp) :: s=0.3_dp
  real(dp) :: a=1.0_dp
  real(dp) :: delta=0.1_dp
  real(dp) :: alpha=0.4_dp
  real(dp) :: k=0.2_dp
  integer :: i
  do i = 1, n - 1
    k = a * s * k**alpha + (1 - delta) * k
  end do
  print *, 'k = ', k
end program main
```

---

Relative merits of Fortran / C / other static type AOT compiled languages?

## Pros

- fast loops / arithmetic

## Cons

- low interactivity
- time consuming to write / read / debug
- hard to parallelize





## Pros

- high interactivity
- easy to write / read / debug

## Cons

- slow loops / arithmetic

Why is pure Python slow?

## Pros

- high interactivity
- easy to write / read / debug

## Cons

- slow loops / arithmetic

Why is pure Python slow?

## Problem 1: Type checking

---

```
x, y = 1, 2  
z = x + y
```

```
x, y = 1.0, 2.0  
z = x + y
```

```
x, y = 'foo', 'bar'  
z = x + y
```

---

How does Python know which operation to perform?

Answer: Python checks the type of the objects first

---

```
>> x = 1
>> type(x)
int
```

---

```
>> x = 'foo'
>> type(x)
str
```

---

In a large loop, this type checking generates massive overhead

## Problem 2: Memory management

---

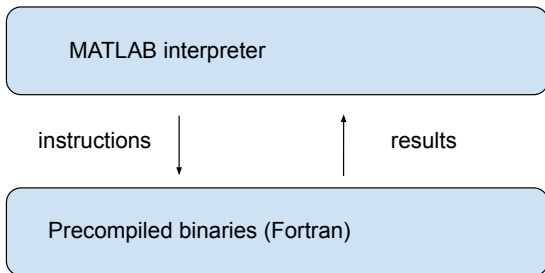
```
>>> import sys
>>> x = [1.0, 2.0]
>>> sys.getsizeof(x) * 8      # number of bits
576                           # whaaaat???
```

---

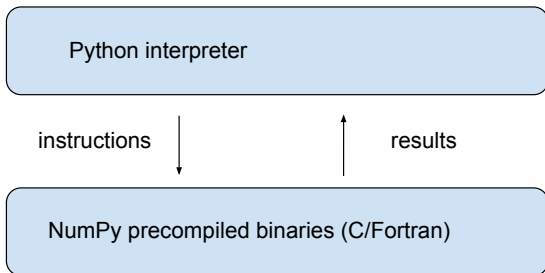
So how can we get

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

## MATLAB's vectorization trick



## Python + NumPy – stealing MATLAB's idea





# Vectorization: pros and cons

## Pros

- high interactivity

## Cons

- some tasks cannot be efficiently vectorized
- cannot adapt flexibly to function arguments / hardware

## Julia — rise of the JIT compilers

---

```
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)
```

---

Function `solow` is efficiently JIT compiled after the first call

## Python + Numba copy Julia

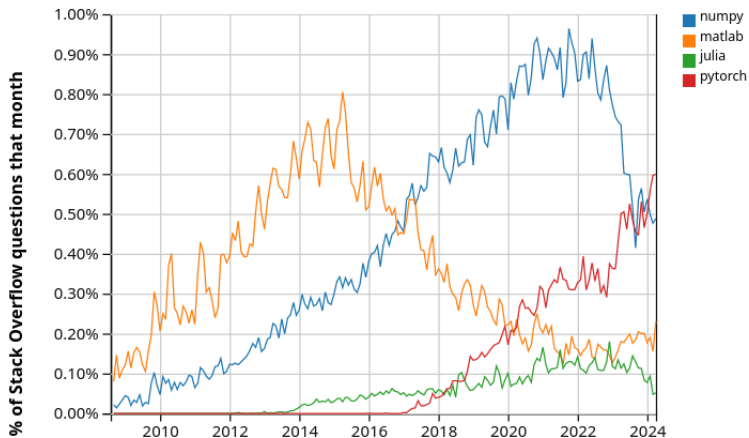
```
from numba import jit

@jit
def solow(k0,  $\alpha=0.4$ ,  $\delta=0.1$ , n=1_000):
    k = k0
    for i in range(n-1):
        k = s * k** $\alpha$  + (1 -  $\delta$ ) * k
    return k

solow(0.2)
```

Runs at same speed as Julia / C / Fortran

Some trends:



So where does JAX fit in?

Let's start with some motivation and background

# AI-driven scientific computing

AI is changing the world

- image processing / computer vision
- speech recognition, translation
- scientific knowledge discovery
- forecasting and prediction
- generative AI

Plus killer drones, skynet, etc....

# AI-driven scientific computing

AI is changing the world

- image processing / computer vision
- speech recognition, translation
- scientific knowledge discovery
- forecasting and prediction
- generative AI

Plus killer drones, skynet, etc....

## Projected spending on AI in 2024:

- Google: \$48 billion
- Microsoft: \$60 billion
- Meta: \$40 billion
- etc.



# Deep learning in two slides

Supervised deep learning: find a good approximation to an unknown functional relationship

$$y = f(x) \quad (x \in \mathbb{R}^d, y \in \mathbb{R})$$

## Examples.

- $x$  = sequence of words,  $y$  = next word
- $x$  = weather sensor data,  $y$  = max temp tomorrow

Problem:

- observe  $(x_i, y_i)_{i=1}^n$  and seek  $f$  such that  $y_{n+1} \approx f(x_{n+1})$

Training: minimize the empirical loss

$$\ell(\theta) := \sum_{i=1}^n (y_i - f_{\theta}(x_i))^2 \quad \text{s.t.} \quad \theta \in \Theta$$

But what is  $\{f_{\theta}\}_{\theta \in \Theta}$ ?

In the case of ANNs, we consider all  $f_{\theta}$  having the form

$$f_{\theta} = \sigma \circ A_1 \circ \cdots \circ \sigma \circ A_{k-1} \circ \sigma \circ A_k$$

where

- $A_i x = W_i x + b_i$  is an affine map
- $\sigma$  is a nonlinear “activation” function

Training: minimize the empirical loss

$$\ell(\theta) := \sum_{i=1}^n (y_i - f_{\theta}(x_i))^2 \quad \text{s.t.} \quad \theta \in \Theta$$

But what is  $\{f_{\theta}\}_{\theta \in \Theta}$ ?

In the case of ANNs, we consider all  $f_{\theta}$  having the form

$$f_{\theta} = \sigma \circ A_1 \circ \cdots \circ \sigma \circ A_{k-1} \circ \sigma \circ A_k$$

where

- $A_i x = W_i x + b_i$  is an affine map
- $\sigma$  is a nonlinear “activation” function

Training: minimize the empirical loss

$$\ell(\theta) := \sum_{i=1}^n (y_i - f_{\theta}(x_i))^2 \quad \text{s.t.} \quad \theta \in \Theta$$

But what is  $\{f_{\theta}\}_{\theta \in \Theta}$ ?

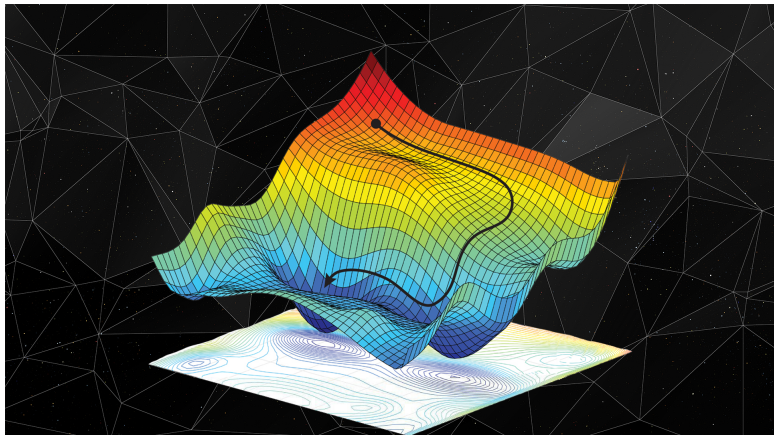
In the case of ANNs, we consider all  $f_{\theta}$  having the form

$$f_{\theta} = \sigma \circ A_1 \circ \dots \circ \sigma \circ A_{k-1} \circ \sigma \circ A_k$$

where

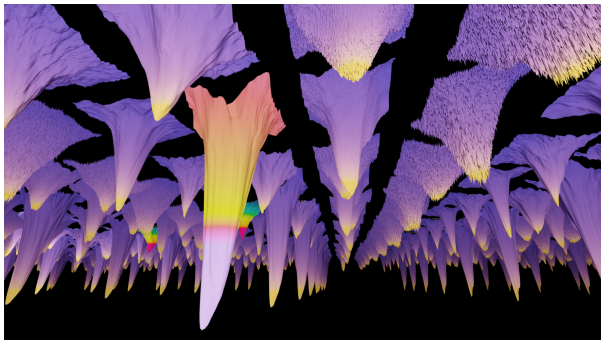
- $A_i x = W_i x + b_i$  is an affine map
- $\sigma$  is a nonlinear “activation” function

Minimizing a smooth loss functions – what algorithm?



Source: <https://danielkhv.com/>

Deep learning:  $\theta \in \mathbb{R}^d$  where  $d = ?$



Source: <https://losslandscape.com/gallery/>

# Hardware



“NVIDIA supercomputers are the factories of the AI industrial revolution.” – Jensen Huang



# Software

## Core elements

- automatic differentiation (for gradient descent)
- parallelization (GPUs! — how many?)
- Compilers / JIT-compilers

Crucially, these components must be well integrated



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

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

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

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

Source: JAX readthedocs

# JAX for economists

I do mathematical modeling / optimization / simulation

My wishlist:

- automated parallelization
- JIT compiler
- integrated autodiff
- automatically / transparently supports CPUs / GPUs / TPUs

JAX ticks these boxes