

# Modern Computational Economics and Policy Applications

A workshop for the IMF's Institute for Capacity Development

Chase Coleman and John Stachurski

March 2024

Slides, code, personnel, course outline:

[https://github.com/QuantEcon/imf\\_2024](https://github.com/QuantEcon/imf_2024)

Quick poll:

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

This morning:

1. Bird's eye view of scientific computing
2. The AI revolution and its impact on scientific computing
3. The Python language and its scientific ecosystem
4. Working with Jupyter

# Bird's eye view of scientific computing

Topics covered in these slides

1. traditional ahead-of-time (AOT) compiled languages
2. interpreted languages and the “vectorization” trick
3. beyond vectorization: modern just-in-time (JIT) compilers
4. parallelization

# Traditional paradigm: static types and AOT compilers

Typical languages: Fortran / C / C++

**Example.** Suppose we want to compute the sequence

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

from some given  $k_0$

Let's write a function in C that

1. implements the loop
2. returns the last  $k_t$

---

```
#include <stdio.h>
```

```
#include <math.h>
```

```
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);  
}
```

---

```
ϕ john on gz-precision .../imf_2024 on β main  
>> gcc solow.c -o out -lm
```

```
ϕ john on gz-precision .../imf_2024 on β main  
>> ./out
```

```
k = 6.240251
```

---



## Pros

- fast loops / arithmetic

## Cons

- low interactivity!
- time consuming to write large programs
- relatively hard to read / debug
- low portability
- hard to parallelize!!

For comparison, the same operation in Python:

```
alpha = 0.4
s = 0.3
delta = 0.1
n = 1_000
k = 0.2

for i in range(n):
    k = s * k**alpha + (1 - delta) * k

print(k)
```

## Pros

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

## Cons

- slow loops / arithmetic

Why is pure Python slow?

## Pros

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

## Cons

- slow loops / arithmetic

Why is pure Python slow?

## Problem 1: Type checking

Consider the Python code snippets

# Ints

$$x, y = 1, 2$$
$$z = x + y \quad \# \quad z = 3$$

## # Floats

```
x, y = 1.0, 2.0
```

$$z = x + y \quad \# \quad z = 3.0$$

## # Strings

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

```
z = x + y      # z = 'foobar'
```

## 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 = [2.56, 3.21]
>>> sys.getsizeof(x) * 8      # number of bits
576                            # whaaaat???
>>> sys.getsizeof(x[0]) * 8   # number of bits
192                            # whaaaat???
```

---

Also, lists of numbers are pointers to dispersed int/float objects — not contiguous data

So how can we get

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



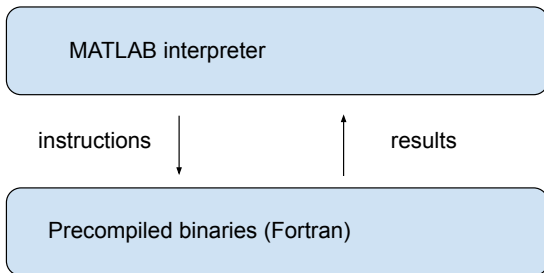
## MATLAB

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

```
b = [0.5, 1.0]';
```

$$x = \text{inv}(A) * b$$

# The vectorization trick



# Python + NumPy

---

```
import numpy
```

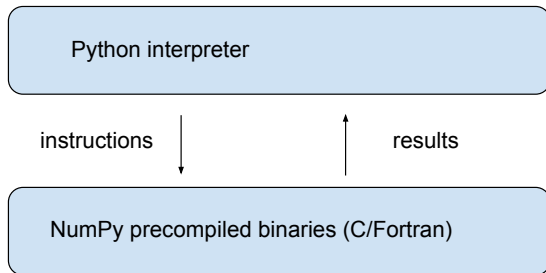
```
A = ((2.0, -1.0),  
      (5.0, -0.5))
```

```
b = (0.5, 1.0)
```

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

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

---



# Vectorization: the good, the bad and the ugly

## Pros

- high interactivity / portability
- many scientific calculations can be framed as operations on arrays

## Cons

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

These last two issues turn out to be extremely important



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

---

```
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 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

# Parallelization

For tasks that can be divided across multiple “workers,”

$$\text{execution time} = \text{time per worker} / \text{number of workers}$$

So far we have been discussing time per worker

- running code fast along a single thread

The other option for speed gains is

- divide up the execution task
- spread across multiple threads / processes

# Parallelization is the big game changer powering the AI revolution

Market Summary > NVIDIA Corp

879.44 USD

✓ Following

+440.44 (100.33%) ↑ past 6 months

Closed: 14 Mar, 7:59 pm GMT-4 • Disclaimer

After hours 872.98 -6.46 (0.73%)

1D | 5D | 1M | **6M** | YTD | 1Y | 5Y | Max

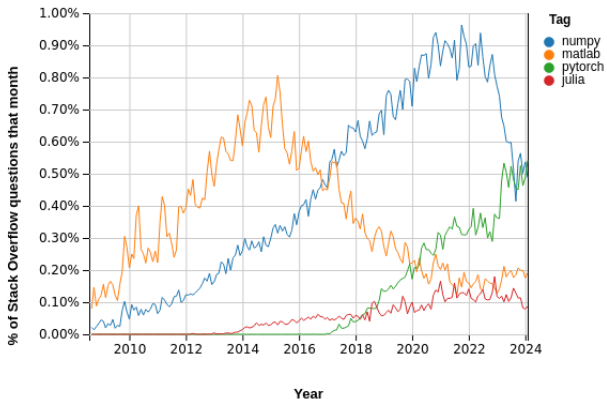


What economists need: software that will parallelize **for us**

- automated intelligent parallelization
- JIT compiled — flexible
- portable
- seamlessly supports most CPUs / GPUs / hardware accelerators

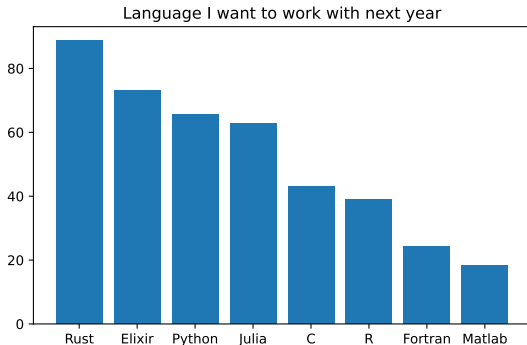
Last topic: Trends and future directions

Some trends:



Source: Stackoverflow Trends

## Stack Overflow 2023 Developer Survey (50 languages)



— <https://survey.stackoverflow.co/2023/>