# Numerical Computing: Background

## Prepared for the Bank of Portugal Computational Economics Course

John Stachurski

October 2025

# Topics

- Low level languages
- Interpreted code
- Array processing
- JIT compilation

# History: Setting the stage

Let's briefly discuss the evolution 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:

```
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

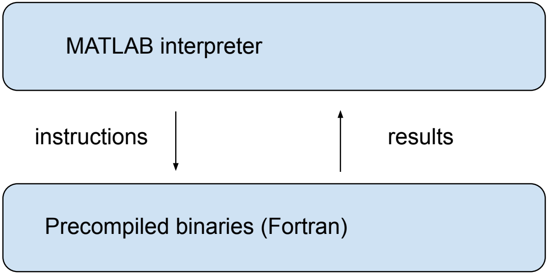
- slow

So how can we get

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

# MATLAB

"MATLAB is Fortran for the 1990s!"



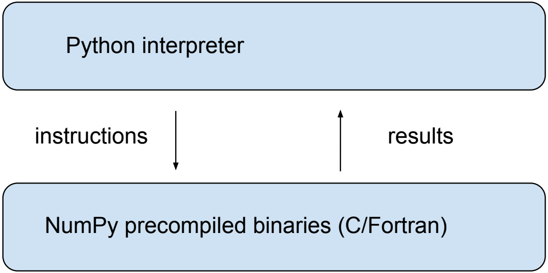
```
A = [2.0, -1.0
     5.0, -0.5];

b = [0.5, 1.0]';

x = inv(A) * b
```

# Python + NumPy

Open source MATLAB-like array operations within Python



```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

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
- Repeated breaking changes and package instability
- Parallelization not well automated

# 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