Modern Computational Economics and Policy Applications

A workshop for the IMF's Institute for Capacity Development

Chase Coleman and John Stachurski

March 2024

Sides, code, personnel, course outline:

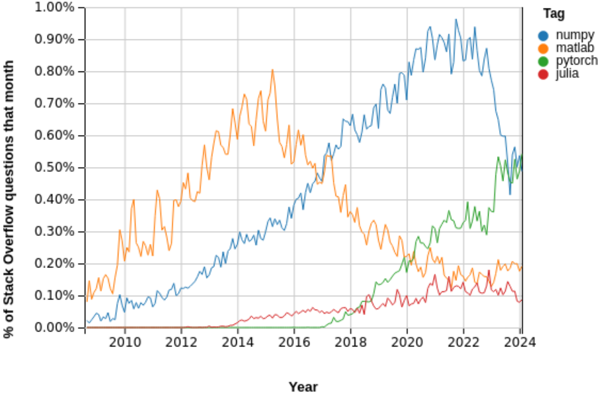
https://github.com/QuantEcon/imf_2024

Background

We start with some background on modern scientific computing.

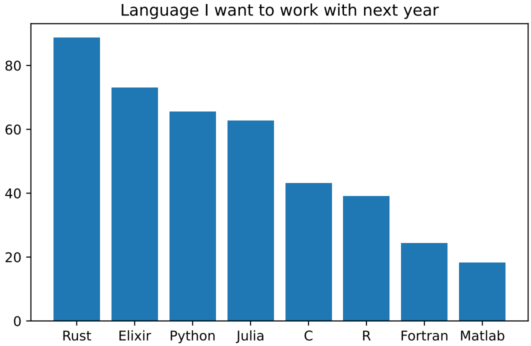
- Traditional compiled languages
- Interpreted languages and “vectorization”
- Modern JIT compilers
- Parallelization

Some trends:



Source: Stackoverflow Trends

Stack Overflow 2023 Developer Survey (50 languages)



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

A review of some scientific computing environments

General purpose scientific computing environments:

1. Fortran / C / C++
2. MATLAB $\approx$ Python + NumPy
3. Julia $\approx$ Python + Numba
4. Python + Google JAX $\approx$ Python + PyTorch

Fortran / C / 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
#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

Cons

- time consuming to write
- hard to debug
- hard to parallelize
- low interactivity
- low portability

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)
```

Pros

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

Cons

- slow

Why is pure Python slow?

Pros

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

Cons

- slow

Why is pure Python slow?

Problem 1: Type checking

Consider the Python code snippets

```python
x, y = 1, 2
z = x + y        # z = 3
```

```python
x, y = 1.0, 2.0
z = x + y        # z = 3.0
```

```python
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

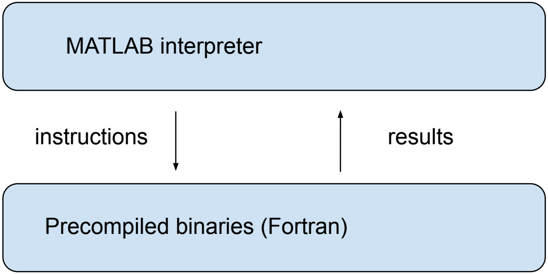
good execution speeds **and** high productivity / interactivity?

MATLAB

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

b = [0.5, 1.0]';

x = inv(A) * b
```


MATLAB interpreter
instructions
results
Precompiled binaries (Fortran)

Python + NumPy

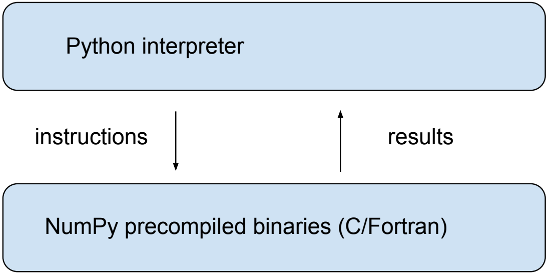
```python
import numpy

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
```


Python interpreter
instructions
results
NumPy precompiled binaries (C/Fortran)

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 copy Julia

```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

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

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

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