Recent Trends in Scientific Computing

John Stachurski

November 2023

Topics

- Traditional compiled languages
- Modern JIT compilers
- AI-driven scientific computing
- Where are we heading?
- Economic applications

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 — static types and AOT compilers

Example. Suppose we want to compute the sequence

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

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 delta = 0.1;
    int i;
    int n = 1000;
    for (i = 0; i < n; i++) {
        k = pow(k, alpha) + (1 - delta) * k;
    }
    printf("k = %f\n", k);
}
```

```
ϕ john on gz-precision …/treasury_2023 on β main
❯❯ gcc solow.c -o out -lm

ϕ john on gz-precision …/treasury_2023 on β main
❯❯ ./out

x = 46.415888
```

Pros

- fast

Cons

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

For comparison, the same operation in Python:

```
α = 0.4
δ = 0.1
n = 1_000
k = 0.2

for i in range(n-1):
    k = k**α + (1 - δ) * k

print(k)
```

Pros

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

Cons

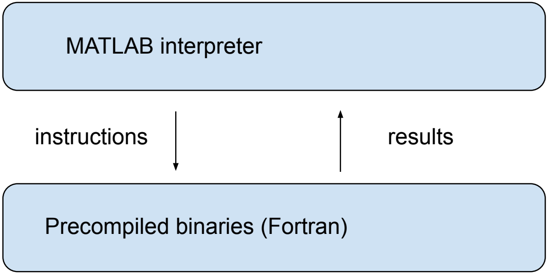
- slow

So how can we get

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

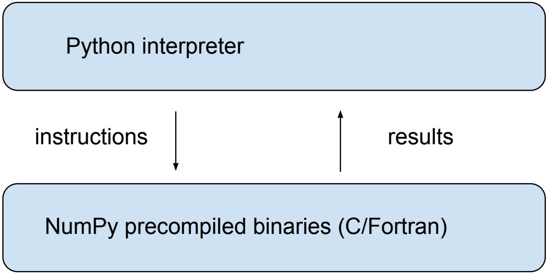
MATLAB

```
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)

Phase 2A: 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
```


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

Phase 3: 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

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

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 = k^α + (1 - δ) * k
    end
    return k
end

solow(0.2)
```

Julia accelerates `solow` at runtime via a JIT compiler

Phase 3 continued: 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 = k**α + (1 - δ) * k
    return k

solow(0.2)
```

Phase 4: AI-driven scientific computing

Key players

- TensorFlow, PyTorch
- Google JAX
- Mojo?

Examples.

- OpenAI uses PyTorch
- Google Bard uses Google JAX
- Apple Ajax uses Google JAX

Lightning introduction to deep learning

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

$$y = f(x)$$

- x is the input and y is the output

Examples.

- $x =$ weather sensor data, $y =$ max temp tomorrow
- $x =$ income distribution, $y =$ tax revenue
- $x =$ unfinished sentence, $y =$ next word

Training

Nonlinear regression: Take data set $(x_i, y_i)_{i=1}^n$ and solve

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

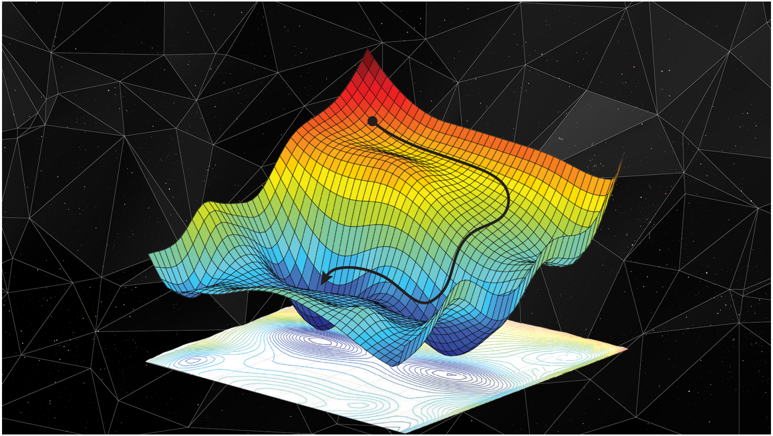
In the case of ANNs, we consider all ψ_θ having the form

$$\psi_\theta = A_1 \circ \sigma \circ \cdots \circ A_k \circ \sigma$$

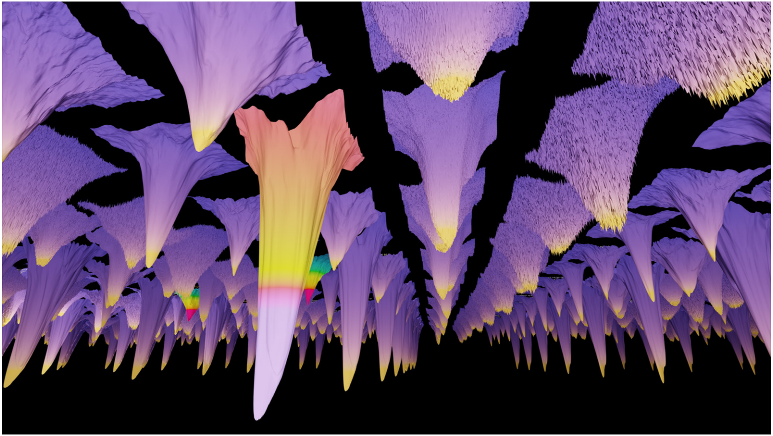
where

- $A_i x = W_i x + b_i$ is an affine map
- σ is a nonlinear "activation" function

Minimizing a smooth loss functions



Source: https://danielkhv.com/



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

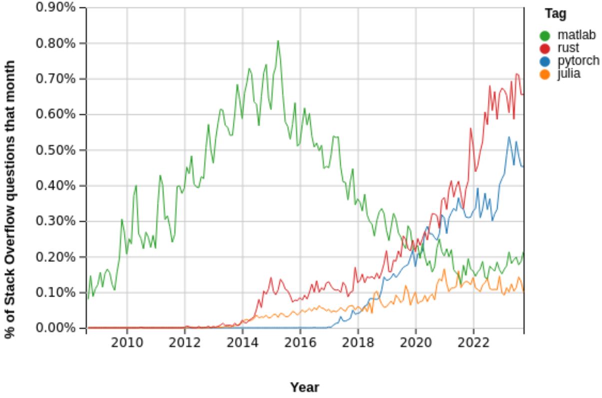
Core elements

- automatic differentiation
- parallelization (CPUs / GPUs / TPUs)
- Compilers / JIT-compilers

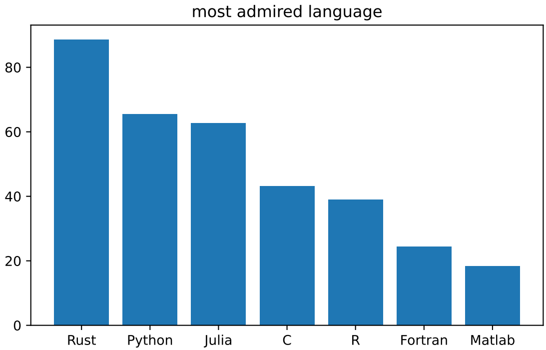
```python
import jax.numpy as jnp
from jax import grad, jit

def predict(params, x):
  for W, b in params:
    y = W @ x + b
    x = jnp.tanh(y)
  return y

def loss(params, x, targets):
  preds = predict(params, x)
  return jnp.sum((preds - targets)**2)

grad_loss = jit(grad(loss))
```

Some trends

Stackoverflow Trends



Stack Overflow 2023 Developer Survey (50 languages)



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

Sample code

https://github.com/QuantEcon/treasury_2023