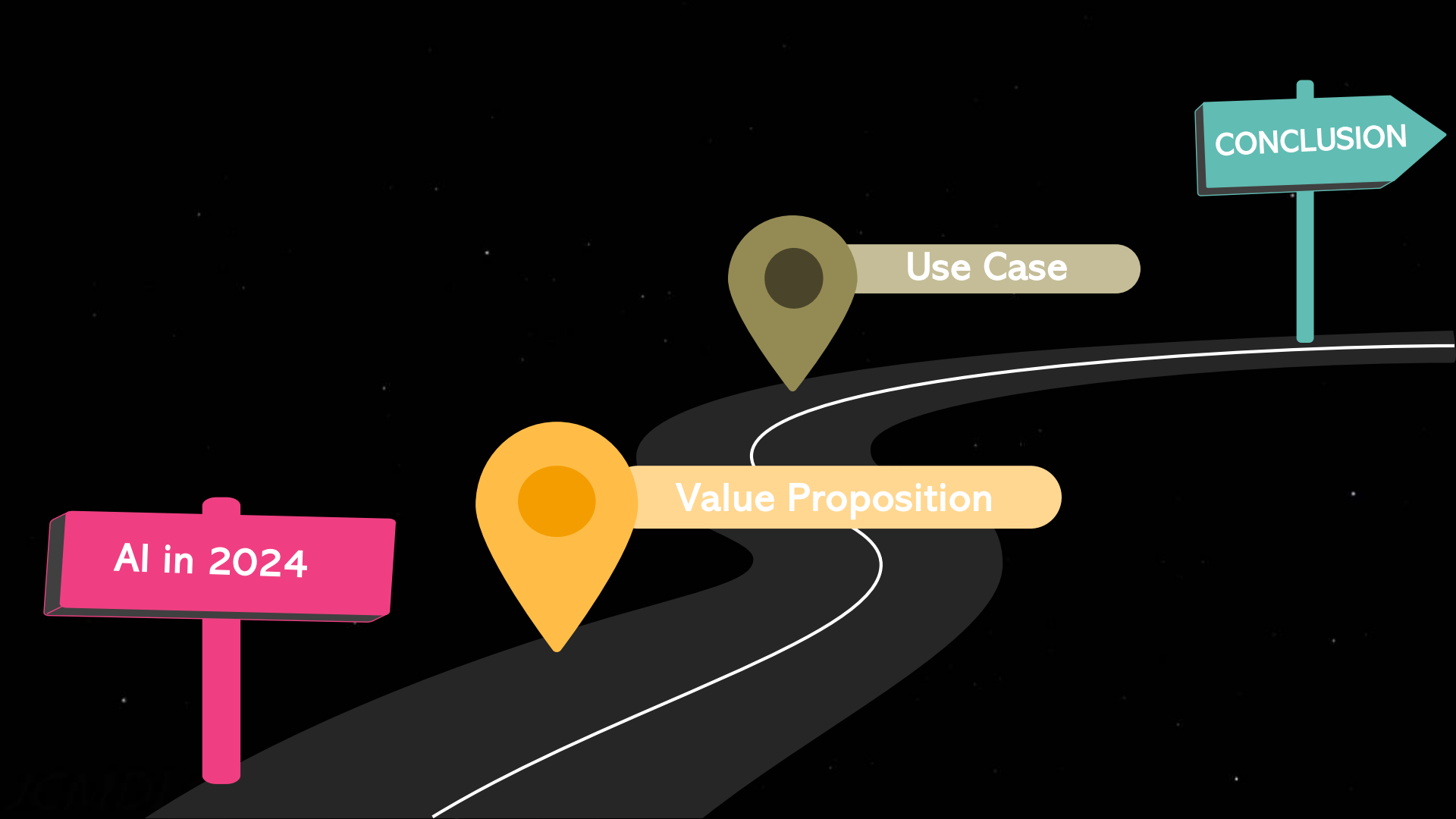
MOJO 🔥
A NEW HOPE FOR AI ?

MOJO 🔥
THE NEXT AI LANGUAGE ?


python
4.0


AI in 2024
Value Proposition
Use Case
CONCLUSION

What is
The current language of
AI ?

C3-GPTo Advisor

L'IA, ou intelligence artificielle, peut être développée et programmée dans différents langages de programmation. Certains des langages les plus couramment utilisés pour créer des systèmes d'IA incluent Python, Java, C++, et R, entre autres. Python est particulièrement populaire dans le domaine de l'IA en raison de sa simplicité, de sa flexibilité, de sa large gamme de bibliothèques et de frameworks dédiés à l'IA (comme TensorFlow, PyTorch, scikit-learn, etc.), et de sa communauté active de développeurs.

StackOverflow Advisor

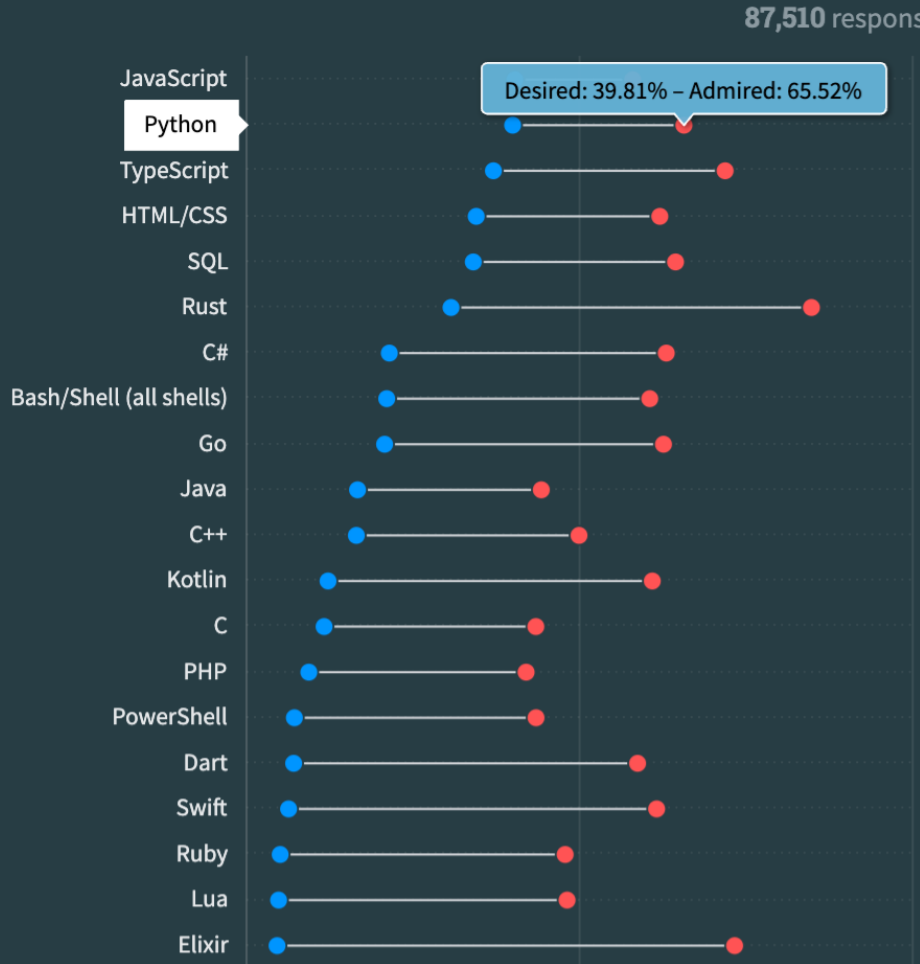
2023
Developer
Survey

In May 2023 over 90,000 developers responded to our annual survey about how they learn and level up, which tools they're using, and which ones they want.

Read the overview →

Methodology →

StackOverflow Advisor



Overview

Developer Profile

Technology

Most popular technologies

Admired and Desired

Worked with vs. want to work with

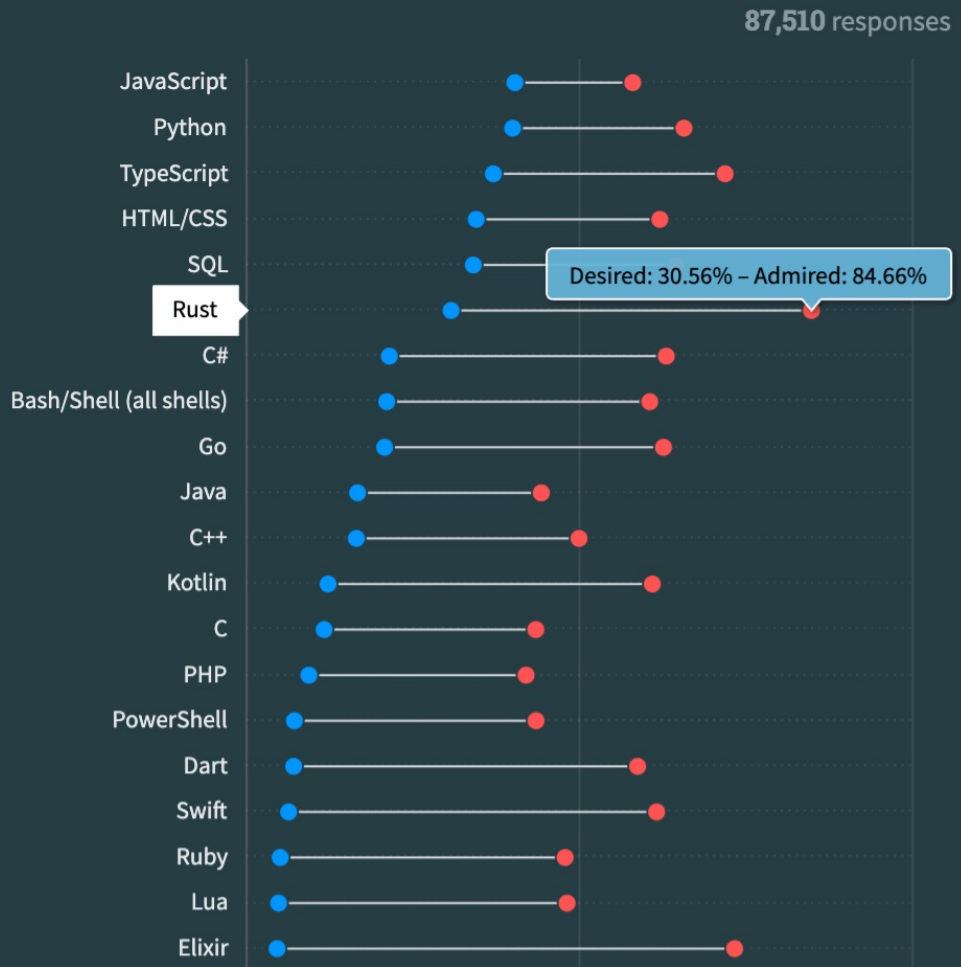
Top paying technologies

AI

Work

Community

StackOverflow Advisor



Overview

Developer Profile

Technology

Most popular technologies

Admired and Desired

Worked with vs. want to work with

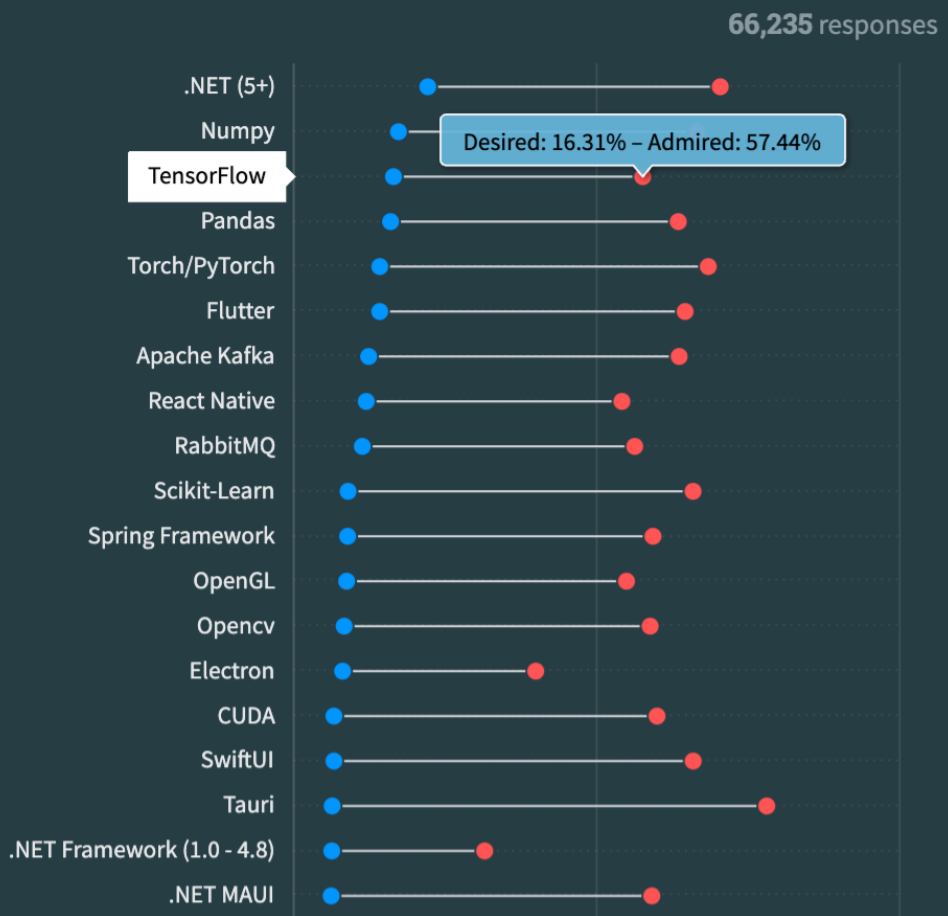
Top paying technologies

AI

Work

Community

StackOverflow Advisor



Overview

Developer Profile

Technology

Most popular technologies

Admired and Desired

Worked with vs. want to work with

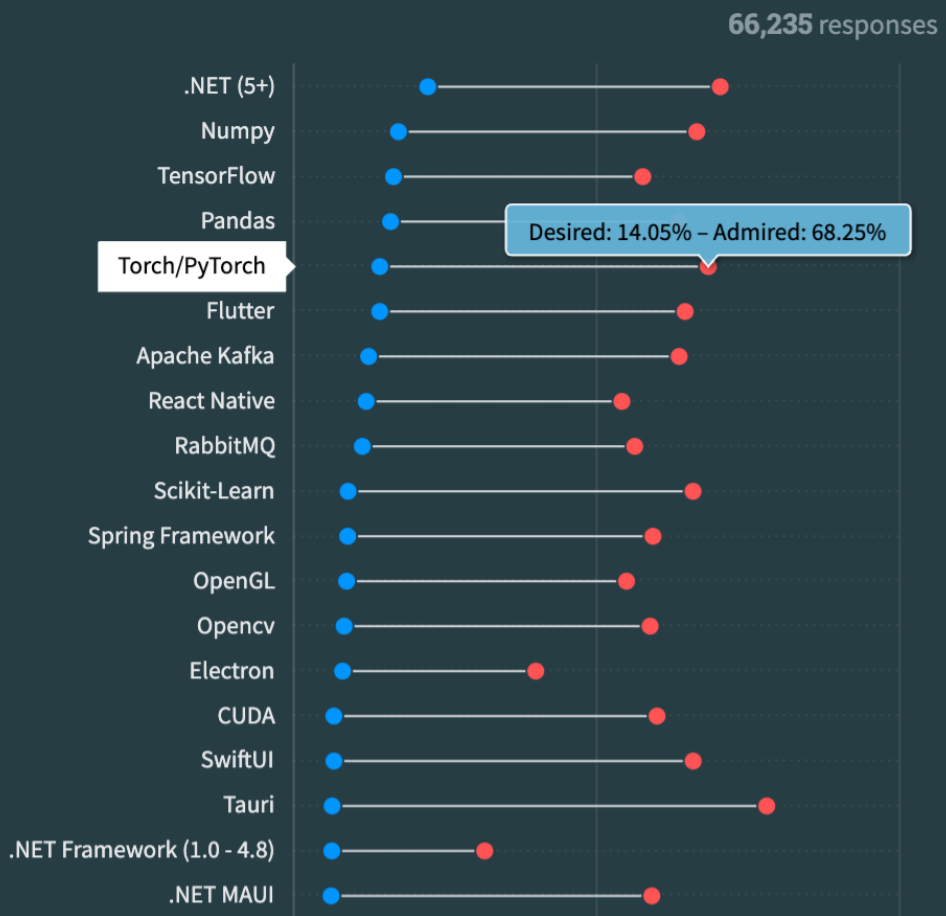
Top paying technologies

AI

Work

Community

StackOverflow Advisor



Overview

Developer Profile

Technology

Most popular technologies

Admired and Desired

Worked with vs. want to work with

Top paying technologies

AI

Work

Community

How To Choose a Framework ?

https://www.youtube.com/watch?v=k4Tfg6-7cyQ

AI Programming Landscape

Model 

System  

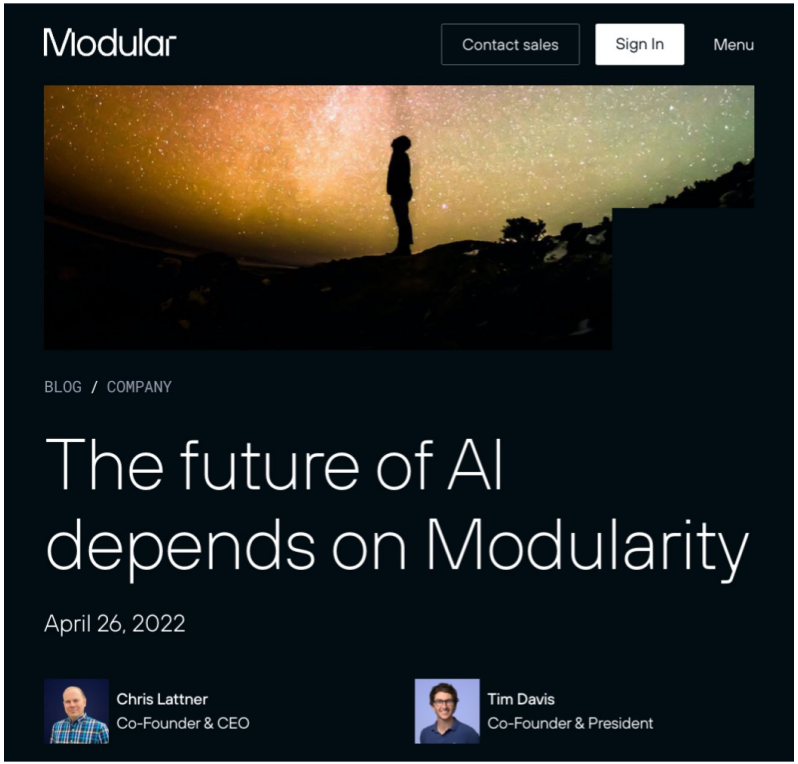
Hardware CUDA, OpenCL, ROCm

NEW KID IN TOWN!

Mojo 🔥

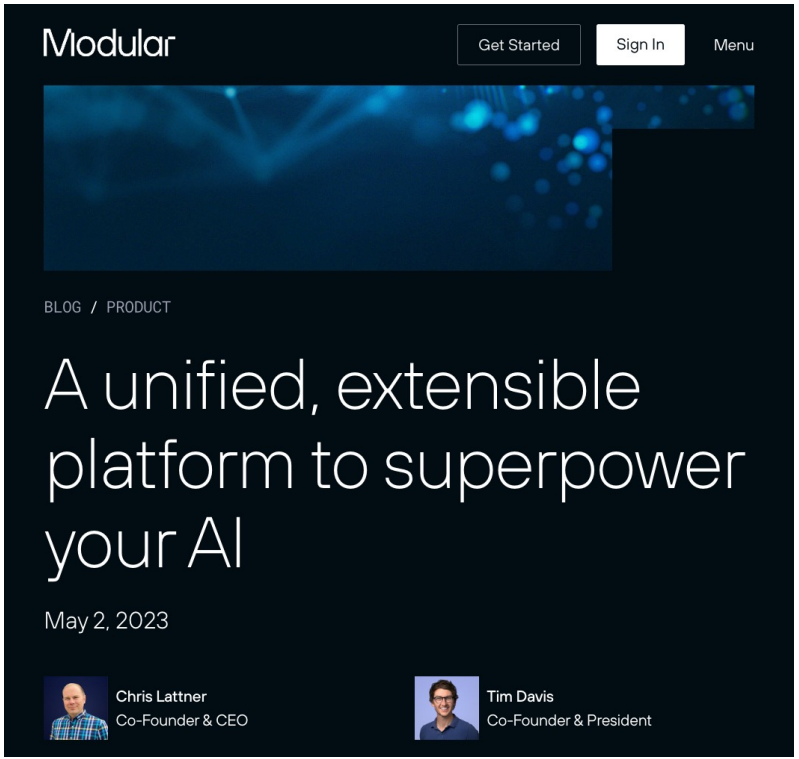
02/05/2023

Mojo is born!



https://www.modular.com/blog/the-future-of-ai-depends-on-modularity

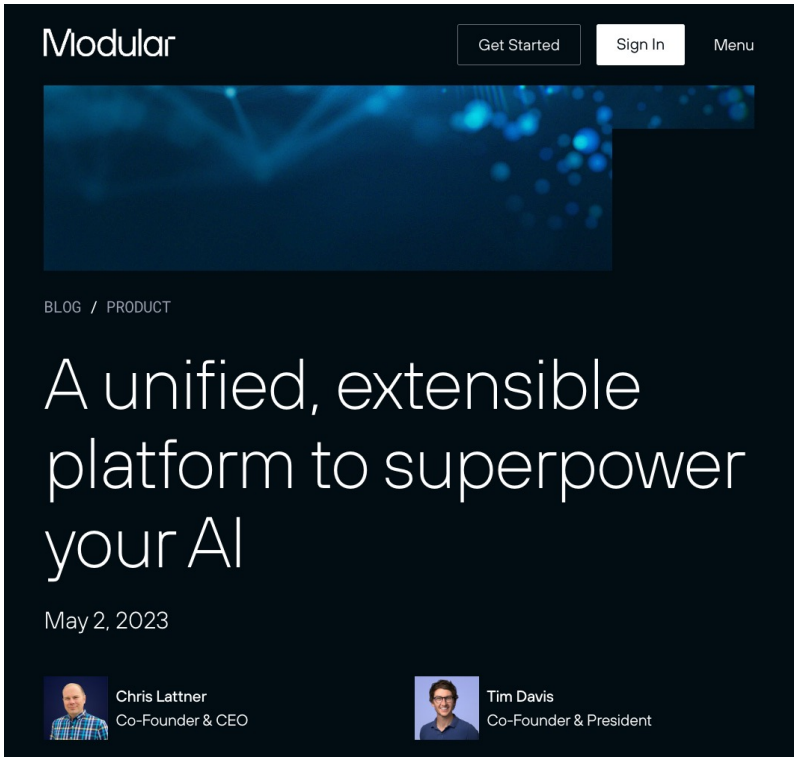
Mojo is born!



https://www.modular.com/blog/a-unified-extensible-platform-to-superpower-your-ai

Modular
Accelerated
e**X**ecution
platform

Mojo is born!



- Member of the python family (superset of python)
- Support modern chip architectures (thanks to MLIR)
- Predictable low level performance

https://www.modular.com/blog/a-unified-extensible-platform-to-superpower-your-ai

Mojo is born!

Chris Lattner

2000 beginning of the project LLVM

2003 release of LLVM 1.0

2007 release of CLang 1.0

2008 XCode 3.1

2011 Clang replace gcc on macos

2014 release of Swift 1.0

2018 beginning of the MLIR

2022 creation of Modular cie

2023 🔥

https://www.nondot.org/sabre/

Mojo is blazing fast!

https://www.modular.com/blog/how-mojo-gets-a-35-000x-speedup-over-python-part-1

Mojo is blazing fast!

https://www.modular.com/blog/how-mojo-gets-a-35-000x-speedup-over-python-part-1

Changelog

2022/01 incorporation

2022/07 seed round (30 M$)

2023/05 announce MAX & Mojo

2023/08 serie B (100 M$)

2023/09 release mojo 0.2.1

2023/10 release mojo 0.4.0

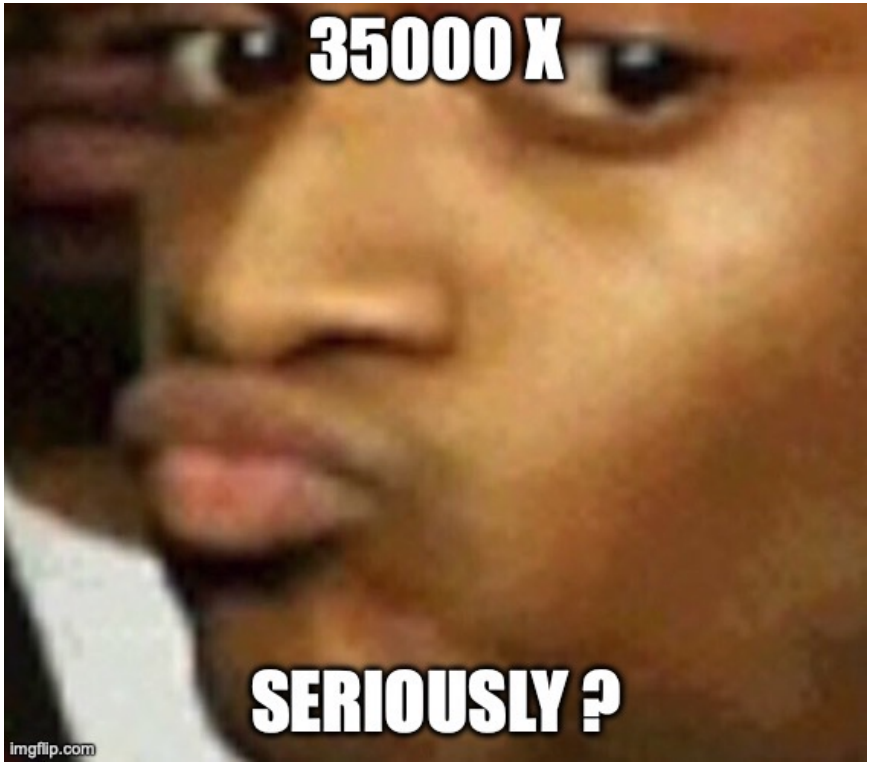
..

2024/01 release mojo 0.7.0

2024/02 release MAX & mojo 24.1

2024/06 release MAX & mojo 24.4

Mojo is blazing fast!



https://www.modular.com/blog/how-mojo-gets-a-35-000x-speedup-over-python-part-1

Performance matters!

Performance matters :

- for our users

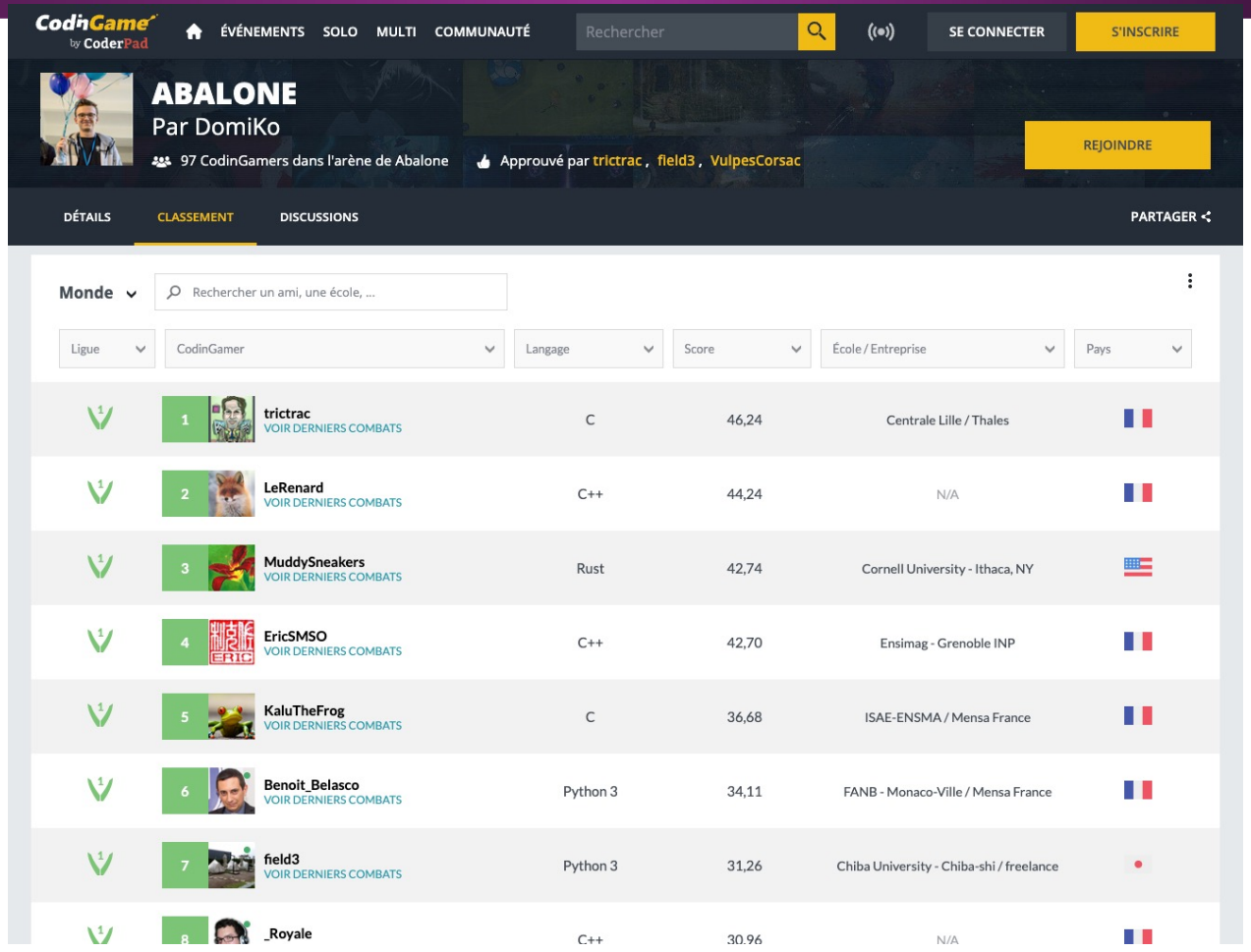
Your resume is being processed

Performance matters!

Performance matters :

- for our users
- for (artificial) intelligence

PERFORMANCE MATTERS!



Performance matters!

Performance matters :

- for our users
- for (artificial) intelligence
- for the planet

PERFORMANCE MATTERS!

Ranking Programming Languages by Energy Efficiency

Rui Pereira[a,b], Marco Couto[c,b], Francisco Ribeiro[c,b], Rui Rua[c,b], Jácome Cunha[c,b], João Paulo Fernandes[d], João Saraiva[c,b]

[a] *C4 — Centro de Competências em Cloud Computing (C4-UBI), Universidade da Beira Interior, Rua Marquês d'Ávila e Bolama, 6201-001, Covilhã, Portugal*
[b] *HASLab/INESC Tec, Portugal*
[c] *Universidade do Minho, Portugal*
[d] *Departamento de Engenharia Informática, Faculdade de Engenharia da Universidade do Porto & CISUC, Portugal*

Abstract

This paper compares a large set of programming languages regarding their efficiency, including from an energetic point-of-view. Indeed, we seek to establish and analyze different rankings for programming languages based on their energy efficiency. The goal of being able to rank programming languages based on their energy efficiency is both recent, and certainly deserves further studies. We have taken rigorous and strict solutions to 10 well defined programming problems, expressed in (up to) 27 programming languages, from the well known Computer Language Benchmark Game repository. This repository aims to compare programming languages based on a strict set of implementation rules and configurations for each benchmarking problem. We have also built a framework to automatically, and systematically, run, measure and compare the energy, time, and memory efficiency of such solutions. Ultimately, it is based on such comparisons that we propose a series of efficiency rankings, based on single and multiple criteria.

Our results show interesting findings, such as how slower/faster languages can consume less/more energy, and how memory usage influences energy consumption. We also present a simple way to use our results to provide software engi-

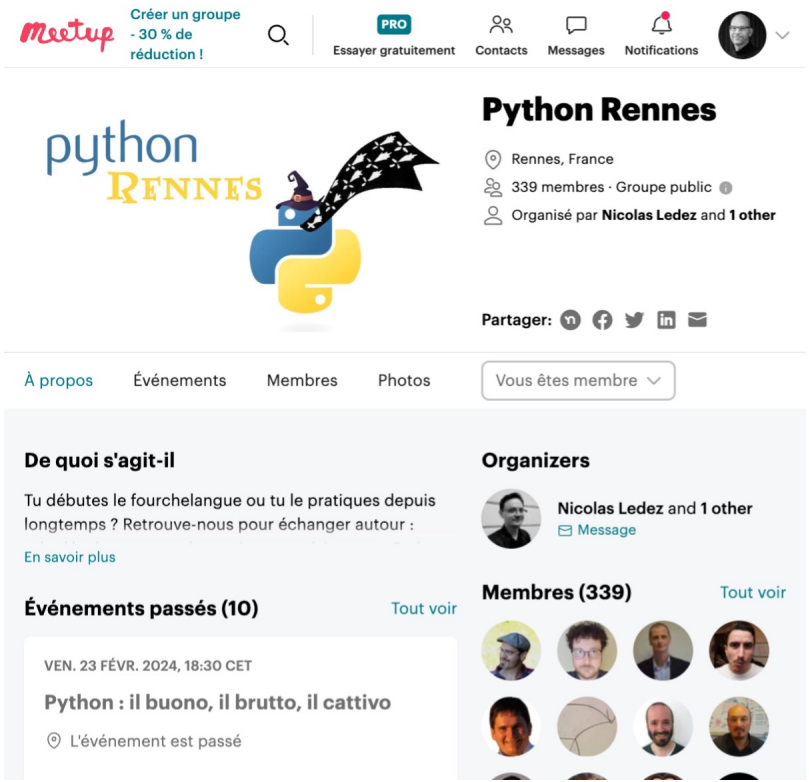
Email addresses: rui.a.pereira@inesctec.pt (Rui Pereira), marco.l.couto@inesctec.pt (Marco Couto), francisco.j.ribeiro@inesctec.pt (Francisco Ribeiro), rui.a.rua@inesctec.pt (Rui Rua), jacome@di.uminho.pt (Jácome Cunha), jpf@dei.uc.pt (João Paulo Fernandes), jas@di.uminho.pt (João Saraiva)

Preprint submitted to Elsevier *January 4, 2021*

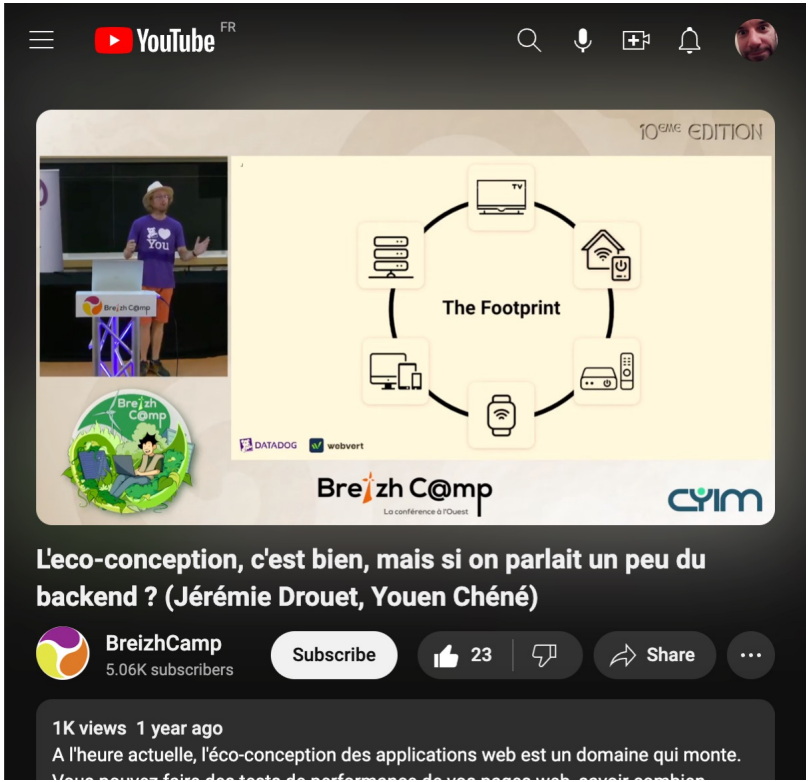
https://haslab.github.io/SAFER/scp21.pdf

	Energy
(c) C	1.00
(c) Rust	1.03
(c) C++	1.34
(c) Ada	1.70
(v) Java	1.98
(c) Pascal	2.14
(c) Chapel	2.18
(v) Lisp	2.27
(c) Ocaml	2.40
(c) Fortran	2.52
(c) Swift	2.79
(c) Haskell	3.10
(v) C#	3.14
(c) Go	3.23
(i) Dart	3.83
(v) F#	4.13
(i) JavaScript	4.45
(v) Racket	7.91
(i) TypeScript	21.50
(i) Hack	24.02
(i) PHP	29.30
(v) Erlang	42.23
(i) Lua	45.98
(i) Jruby	46.54
(i) Ruby	69.91
(i) Python	75.88
(i) Perl	79.58

Meetup python-rennes



https://www.meetup.com/fr-FR/python-rennes/



https://www.youtube.com/watch?v=gE6HUsmh554

Performance matters!

Performance matters :

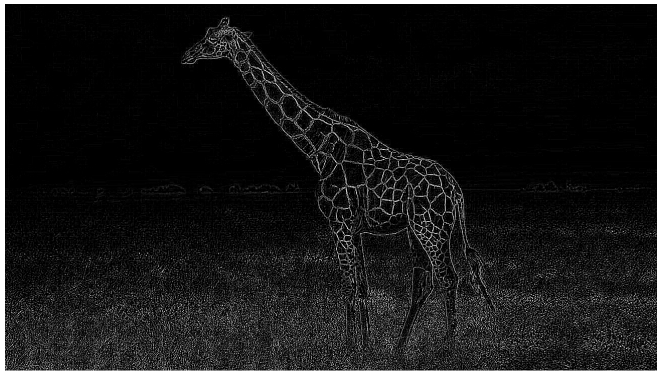
- for our users
- for (artificial) intelligence
- for the planet

IT'S
DEMO TIME!

Laplacian filter
(edge detection)

Edge Detection

$$\Delta = \nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$$

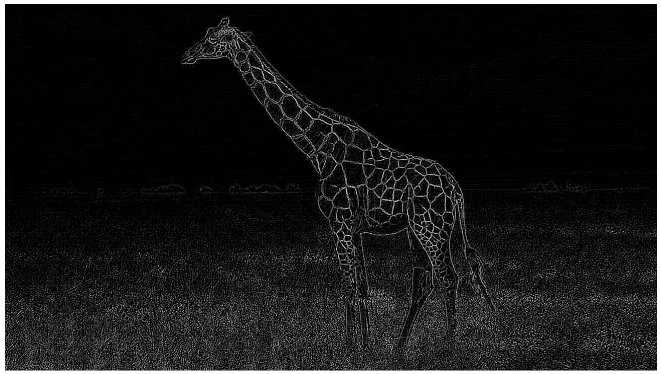


Edge Detection

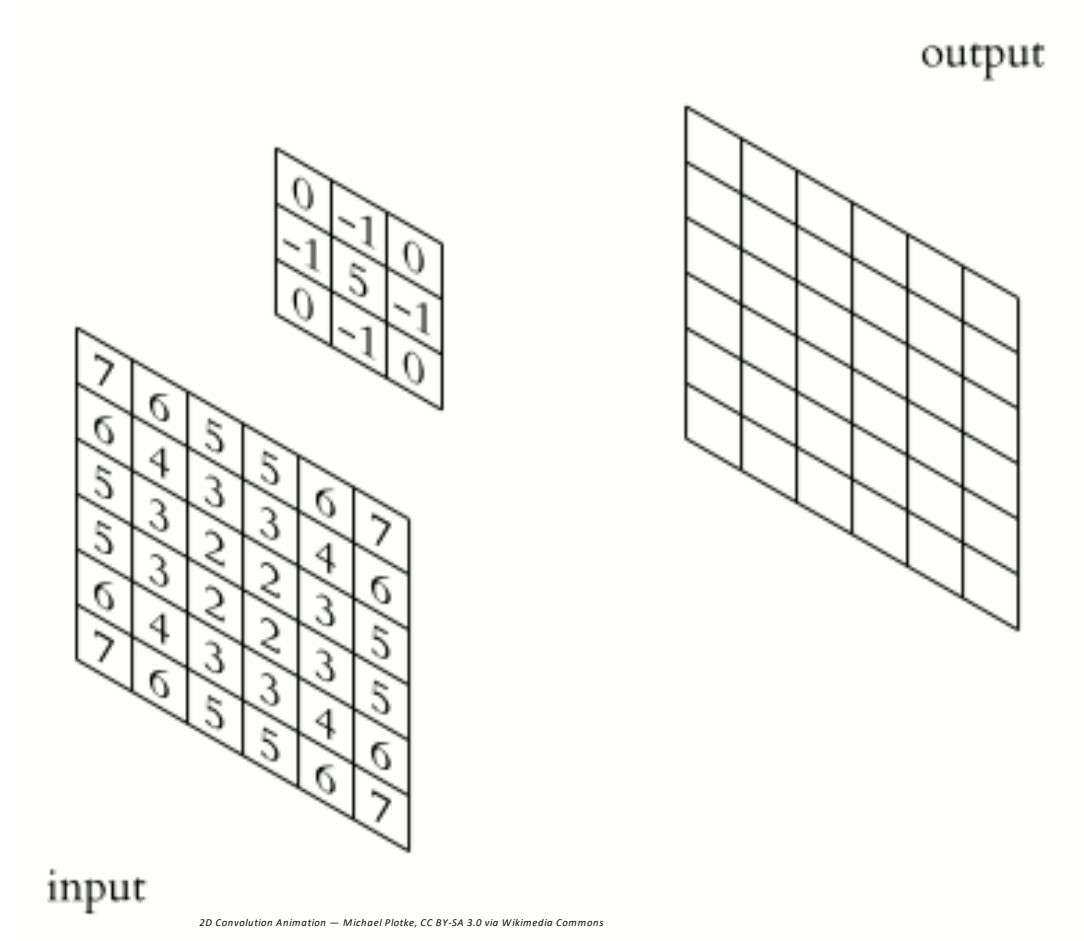
Convolve

kernel

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

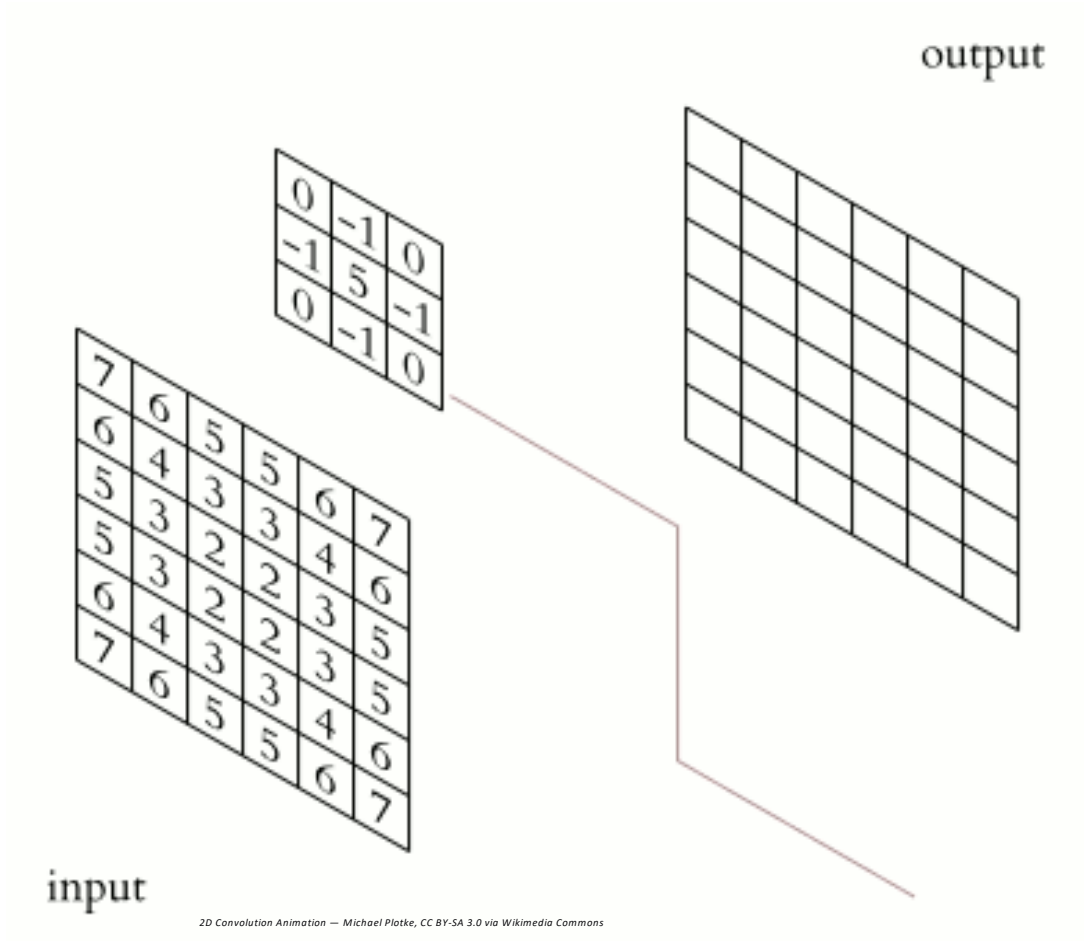


Hop hop hop



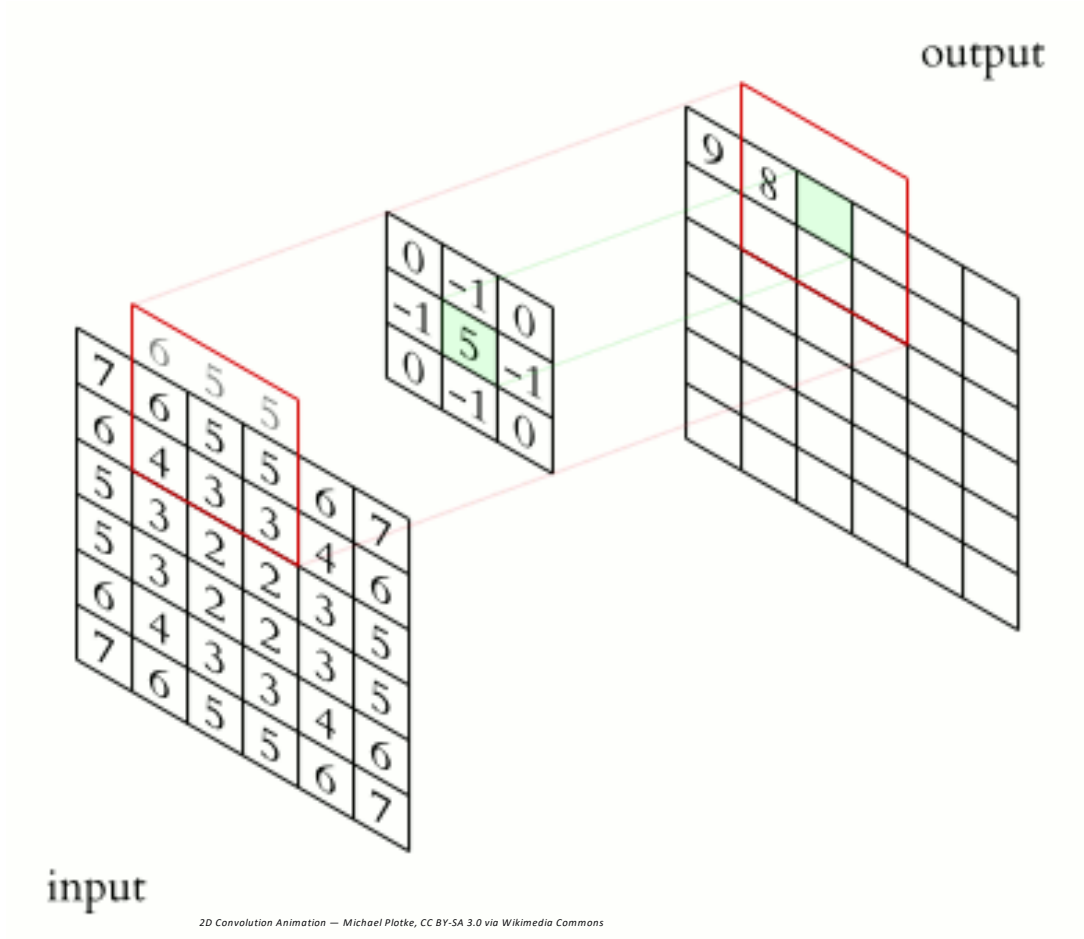
2D Convolution Animation — Michael Plotke, CC BY-SA 3.0 via Wikimedia Commons

Edge Detection



2D Convolution Animation — Michael Plotke, CC BY-SA 3.0 via Wikimedia Commons

Edge Detection



2D Convolution Animation — Michael Plotke, CC BY-SA 3.0 via Wikimedia Commons

Recap

naïve version : 500 ms

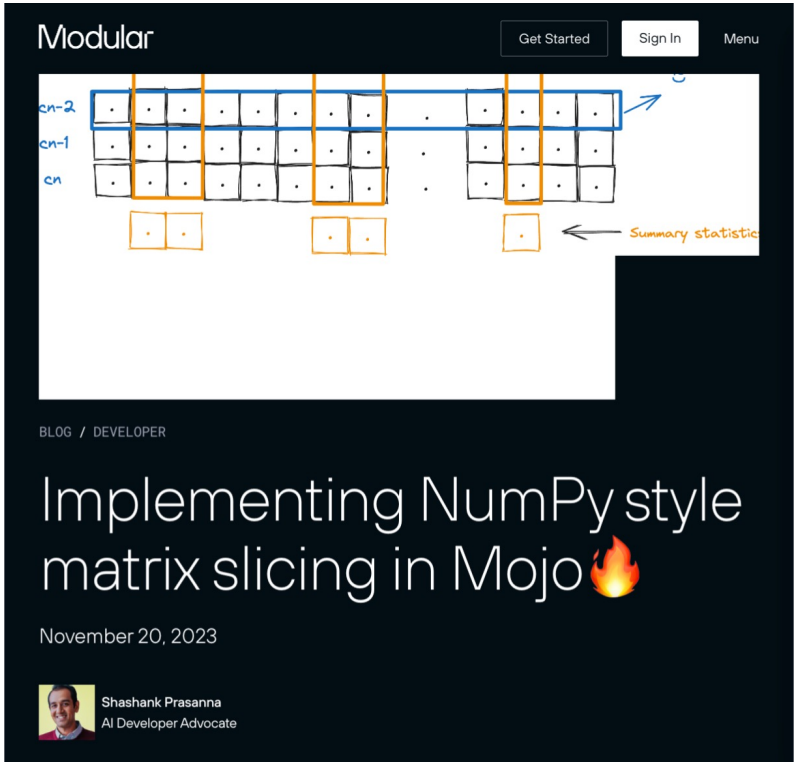
numpy mul : 250 ms x 2

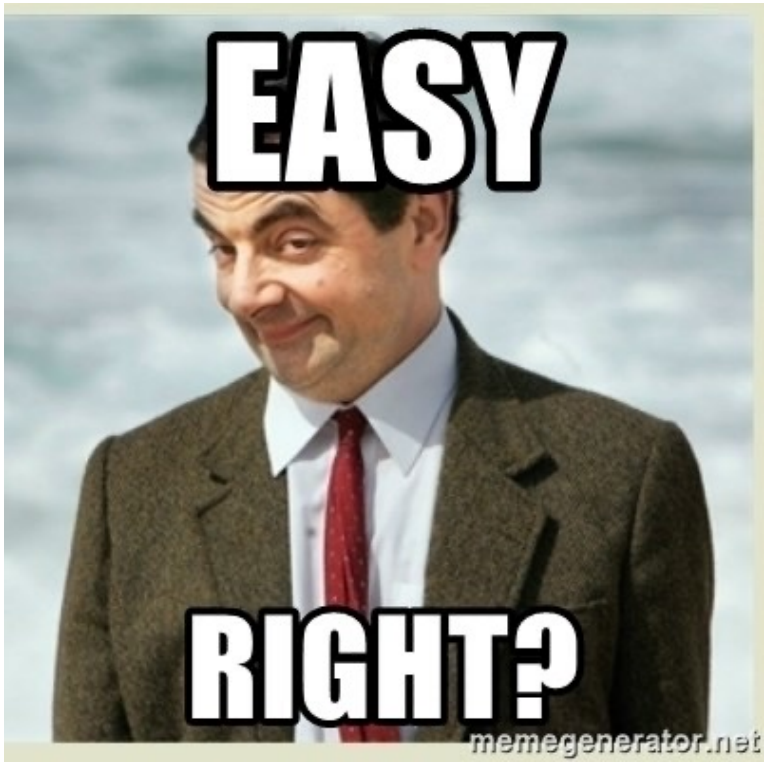
numpy+numba : 50 ms x 10

opencv : 0.5 ms x 1000

And now in mojo ?

And now in Mojo!





https://www.modular.com/blog/implementing-numpy-style-matrix-slicing-in-mojo

Power ! Unlimited Power !

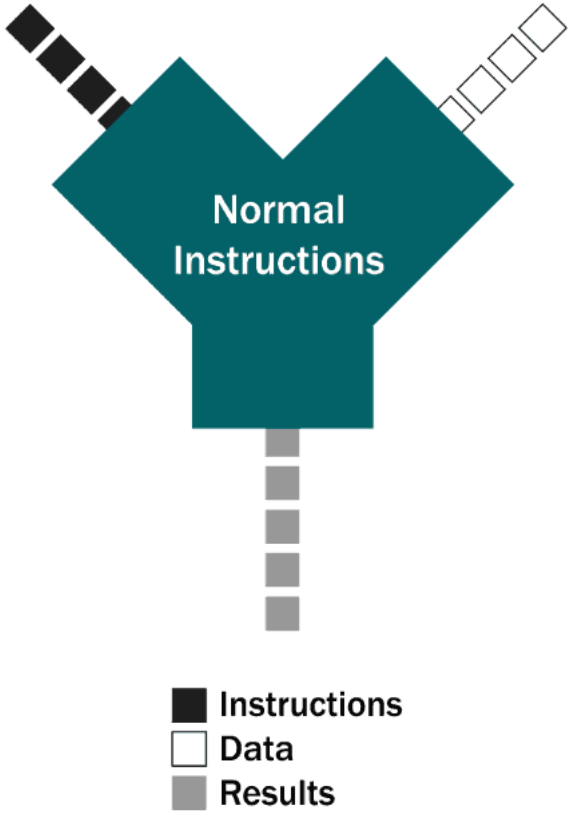
Credits to Georges L. and Corentin R.

“The greatest teacher, failure is”

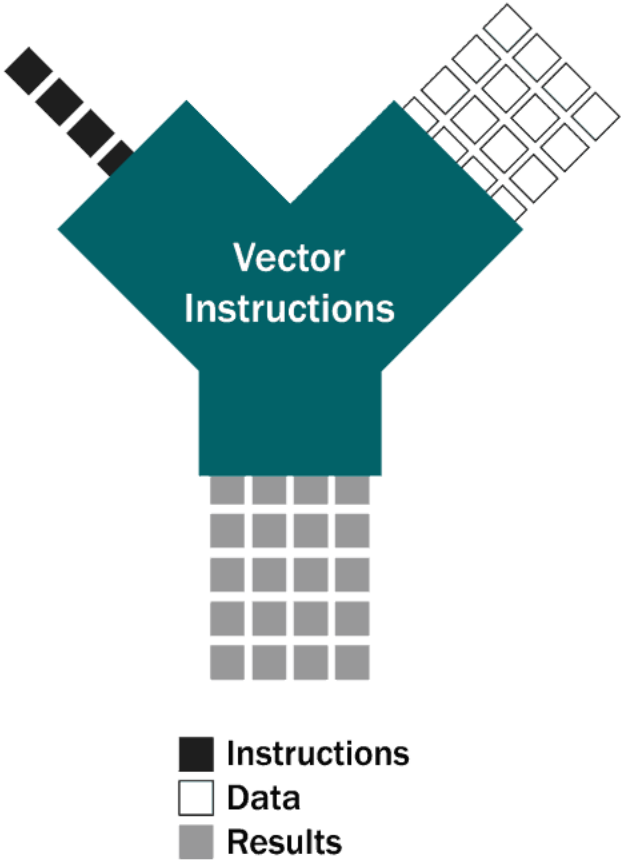
IT'S DEMO TIME !

Let's optimize !

SISD Architecture



SIMD Architecture



Algorithm vectorization

```
fn naive(img: Matrix[DType.float32], kernel: Matrix[DType.float32]) -> Matrix[DType.float32]:
  var result = Matrix[DType.float32](img.height, img.width)
  # Loop through each pixel in the image
  # But skip the outer edges of the image
  for y in range(1, img.height-1):
    for x in range(1, img.width-1):
      # For each pixel, compute the product elements wise
      var acc: Float32 = 0
      for k in range(3) :
        for l in range(3):
          acc += img[y-1+k, x-1+l] * kernel[k, l]
      # Normalize the result
      result[y, x] = min(255, max(0, acc))
  return result
```

Algorithm vectorization

```
alias nelts = simdwidthof[DType.float32]()
```

```
fn naive(img: Matrix[DType.float32], kernel: Matrix[DType.float32]) -> Matrix[DType.float32]:
  var result = Matrix[DType.float32](img.height, img.width)
  # Loop through each pixel in the image
  # But skip the outer edges of the image
  for y in range(1, img.height-1):
    for x in range(1, img.width-1):
      # For each pixel, compute the product elements wise
      var acc: Float32 = 0
      for k in range(3) :
        for l in range(3):
          acc += img[y-1+k, x-1+l] * kernel[k, l]
      # Normalize the result
      result[y, x] = min(255, max(0, acc))
  return result
```

Algorithm vectorization

```
alias nelts = simdwidthof[DType.float32]()


fn naive(img: Matrix[DType.float32], kernel: Matrix[DType.float32]) -> Matrix[DType.float32]:
  var result = Matrix[DType.float32](img.height, img.width)
  # Loop through each pixel in the image
  # But skip the outer edges of the image
  for y in range(1, img.height-1)
    for x in range(1, img.width-1  nelts):
      # For each pixel, compute the product elements wise
      var acc: Float32 = 0
      for k in range(3) :
        for l in range(3):
          acc += img[y-1+k, x-1+l] * kernel[k, l]
      # Normalize the result
      result[y, x] = min(255, max(0, acc))
  return result
```

Algorithm vectorization

```
alias nelts = simdwidthof[DType.float32]()


fn naive(img: Matrix[DType.float32], kernel: Matrix[DType.float32]) -> Matrix[DType.float32]:
  var result = Matrix[DType.float32](img.height, img.width)
  # Loop through each pixel in the image
  # But skip the outer edges of the image
  for y in range(1, img.height-1):
    for x in range(1, img.width-1, nelts):
      # For each pixel, compute the product elements wise
      # var acc: Float32 = 0
      var acc: SIMD[DType.float32,nelts] = 0
      # Loop over the kernel
      for k in range(3):
        for l in range(3):
          acc += img[y-1+k, x-1+l] * kernel[k, l]
      # Normalize the result
      result[y, x] = min(255, max(0, acc))
  return result
```

Algorithm vectorization

```
fn naive(img: Matrix[DType.float32], kernel: Matrix[DType.float32]) -> Matrix[DType.float32]:
  var result = Matrix[DType.float32](img.height, img.width)
  # Loop through each pixel in the image
  # But skip the outer edges of the image
  for y in range(1, img.height-1):
    for x in range(1, img.width-1, nelts):
      # For each pixel, compute the product elements wise
      # var acc: Float32 = 0
      var acc: SIMD[DType.float32,nelts] = 0
      for k in range(3) :
        for l in range(3):
          # acc += img[y-1+k, x-1+l] * kernel[k, l]
          acc += img.simd_load[nelts](y-1+k, x-1+l) * kernel[k, l]
      # result[y, x] = min(255, max(0, acc))
      result.simd_store[nelts](y, x, min(255, max(0, acc)))
return result
```

Algorithm vectorization

```
fn naive(img: Matrix[DType.float32], kernel: Matrix[DType.float32]) -> Matrix[DType.float32]:
  var result = Matrix[DType.float32](img.height, img.width)
  # Loop through each pixel in the image
  # But skip the outer edges of the image
  for y in range(1, img.height-1):
    for x in range(1, img.width-1, nelts):
      # For each pixel, compute the product elements wise
      # var acc: Float32 = 0
      var acc: SIMD[DType.float32,nelts] = 0
      for k in range(3) :
        for l in range(3):
          # acc += img[y-1+k, x-1+l] * kernel[k, l]
          acc += img.simd_load[nelts](y-1+k, x-1+l) * kernel[k, l]
      # Normalize the result
      # result[y, x] = min(255, max(0, acc))
      result.simd_store[nelts](y, x, min(255, max(0, acc)))
  # Handle remaining elements with scalars.
  for n in range(nelts * (img.width-1 // nelts), img.width-1) :
    var acc: Float32 = 0
    for k in range(3) :
      for l in range(3):
        acc += img[y-1+k, n-1+l] * kernel[k, l]
    result[y, n] = min(255, max(0, acc))
  return result
```

Algorithm vectorization

```
alias nelts = simdwidthof[DType.float32]()

fn naive(img: Matrix[DType.float32], kernel: Matrix[DType.float32]) -> Matrix[DType.float32]:
  var result = Matrix[DType.float32](img.height, img.width)
  # Loop through each pixel in the image
  # But skip the outer edges of the image
  for y in range(1, img.height-1):
    @parameter
    fn dot[nelts: Int](x: Int):
      # For each pixel, compute the product elements wise
      var acc: SIMD[DType.float32,nelts] = 0
      for k in range(3) :
        for l in range(3):
          acc += img.simd_load[nelts](y-1+k, x-1+l) * kernel[k, l]
      # Normalize the result
      result.simd_store[nelts](y, x, max(0, acc))
    vectorize[dot, nelts](size=img.width-1)

  return result
```

Algorithm vectorization

```
alias nelts = simdwidthof[DType.float32]()

fn naive(img: Matrix[DType.float32], kernel: Matrix[DType.float32]) -> Matrix[DType.float32]:
  var result = Matrix[DType.float32](img.height, img.width)
  # Loop through each pixel in the image
  # But skip the outer edges of the image
  for y in range(1, img.height-1):
    @parameter
    fn dot[nelts: Int](x: Int):
      # For each pixel, compute the product elements wise
      var acc: SIMD[DType.float32,nelts] = 0
      for k in range(3) :
        for l in range(3):
          acc += img.simd_load[nelts](y-1+k, x-1+l) * kernel[k, l]
      # Normalize the result
      result.simd_store[nelts](y, x, min(255, max(0, acc)))
    vectorize[dot, nelts](size=img.width-1)

  return result
```

Algorithm vectorization

```
alias nelts = simdwidthof[DType.float32]()

fn naive(img: Matrix[DType.float32], kernel: Matrix[DType.float32]) -> Matrix[DType.float32]:
  var result = Matrix[DType.float32](img.height, img.width)
  # Loop through each pixel in the image
  # But skip the outer edges of the image
  for y in range(1, img.height-1):
    @parameter
    fn dot[nelts: Int](x: Int):
      # For each pixel, compute the product elements wise
      var acc: SIMD[DType.float32,nelts] = 0
      for k in range(3) :
        for l in range(3):
          acc += img.simd_load[nelts](y-1+k, x+l) * kernel[k, l]
      # Normalize the result
      result.simd_store[nelts](y, x+1, min(255, max(0, acc)))
    vectorize[dot, nelts](size=img.width-2)

  return result
```

Recap

- Far from stable
- Compilation AOT or JIT
- Python friendly but not Python
- Dynamic Python vs Static Mojo
- Python interoperability
- Predictable behavior with semantic ownership
- Low level optimization
- Blazingly fast

Mojo 🔥

The next AI Language?

Conclusion

- Python is not yet dead !
  But he moves slowly
- This is a great team !
  Will they be able to deploy their platform strategy ?
- Will they be able to unite a community?
  To be open-source or not to be

Merci!

Jean-Luc Tromparent

Principal Engineer @ hellowork group learn, work, move.

https://linkedin.com/in/jltromparent

https://github.com/jiel/laplacian_filters_benchmark

https://noti.st/jlt/qheM0m

👉 I need your feedback 👈