Performance Tuning of Scientific Software

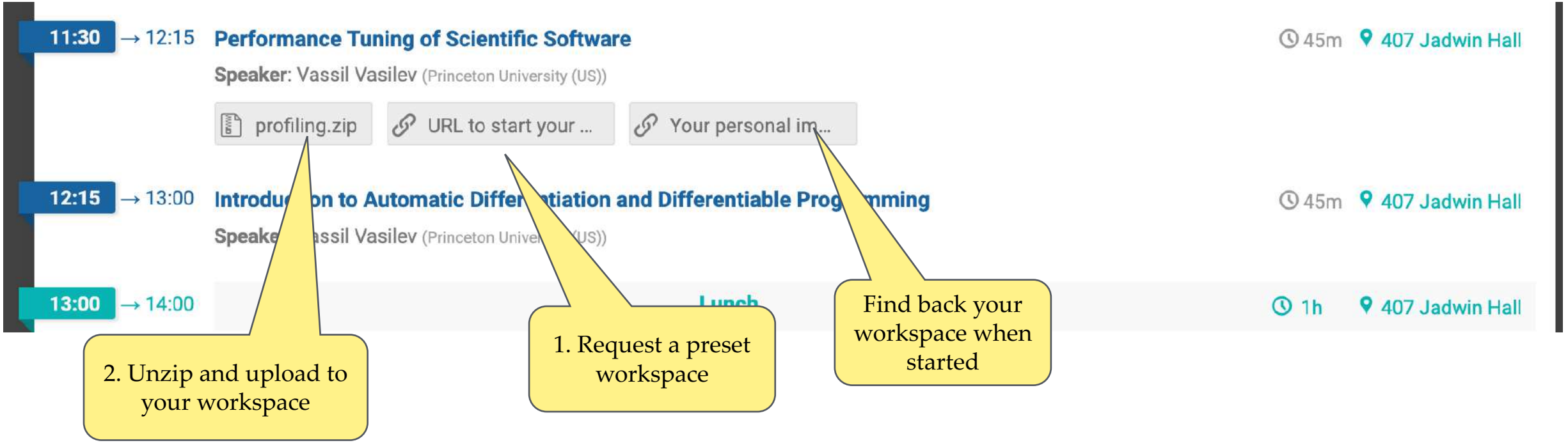
Dr Vassil Vassilev, Princeton/CERN

compiler-research.org

Prerequisites

- Be interactive, ask questions!
- Please share your own experience!
- Have a laptop
- Install the software required:
  - **g++** (GNU, not Apple clang) with **OpenMP** support — the build uses -O2 -pg -fno-inline -fopenmp **gprof** (ships with **binutils**)
  - **make** (for Makefile.repro / make profile)
  - **coreutils** — cksum for the Q4 bit-identical proof a POSIX shell; optionally any PPM-capable image viewer (or ImageMagick to convert) so you can *see* the render
  - sudo apt install build-essential binutils

Prerequisites



Warmup

- How many of you have used Python before?
  - How many of you have a project > 1000 lines of code?
  - > 5000 lines?
  - > 15000 lines?
  - How many of you worked in a team of 2 on such project?
    - > 5
    - > 10
- How many of you coded in a different language than Python?
  - In a low-level language such as C++, C, or assembly?
- What's your average data set? MB, GB, TB, PB, EB?

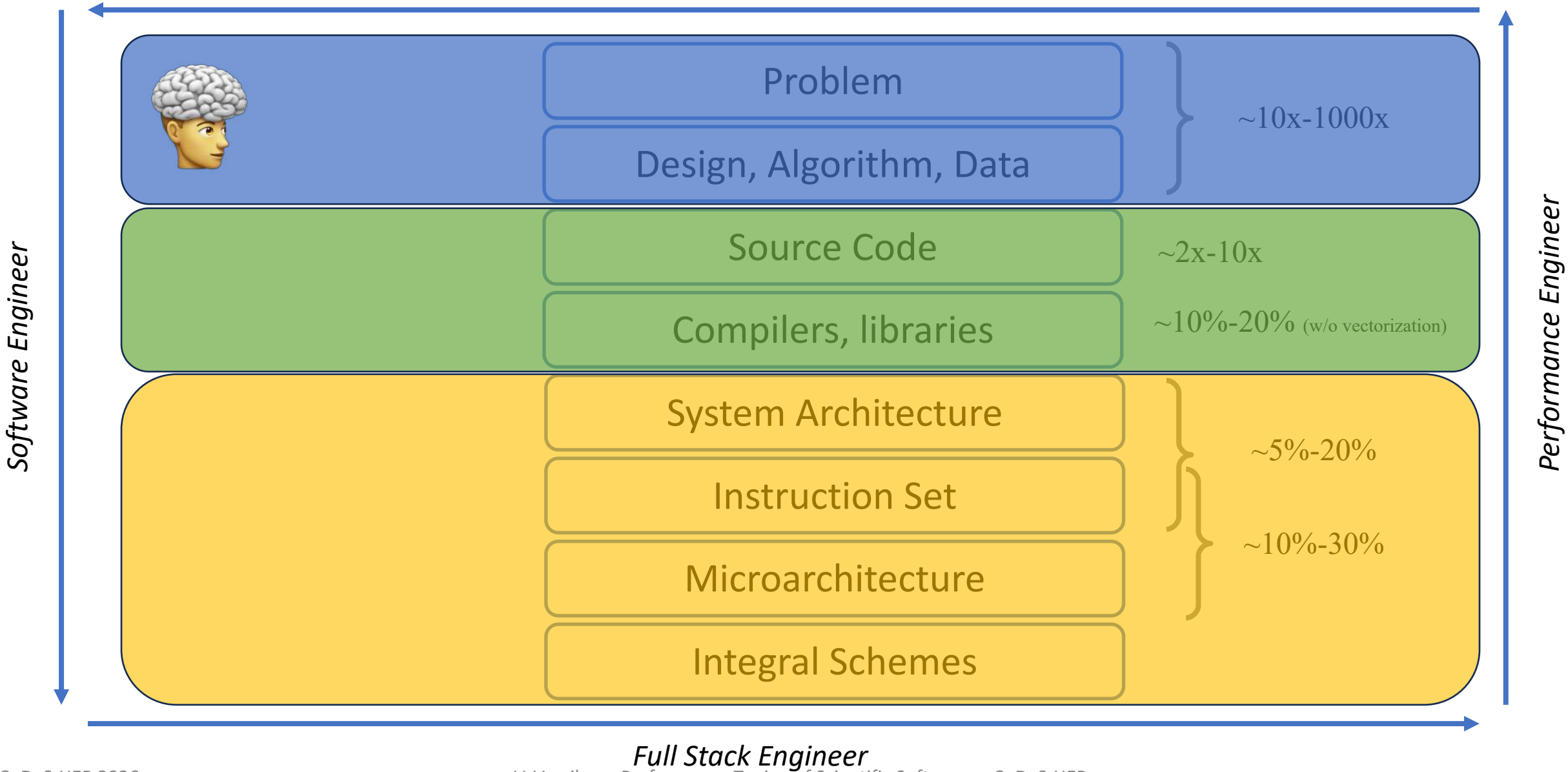
Goals

- Understand the general principles behind high-performance programming
- Recognize and explore performance optimization opportunities
- Build intuition about computer program execution
- Practice

Just Enough Performance

```
def f(N = 100, M = 1000, L = 10000):
    for i in range(N):
        for j in range(M):
            for k in range(L):
                g(i, j, k)
```

Outline


Problem
Design, Algorithm, Data
~10x-1000x
Source Code
Compilers, libraries
~2x-10x
~10%-20% (w/o vectorization)
System Architecture
Instruction Set
Microarchitecture
Integral Schemes
~5%-20%
~10%-30%
Software Engineer
Performance Engineer
Full Stack Engineer

If I had an hour to solve a problem I'd spend 55 minutes thinking about the problem and 5 minutes thinking about the solutions

- *We have no proof Albert Einstein ever said*

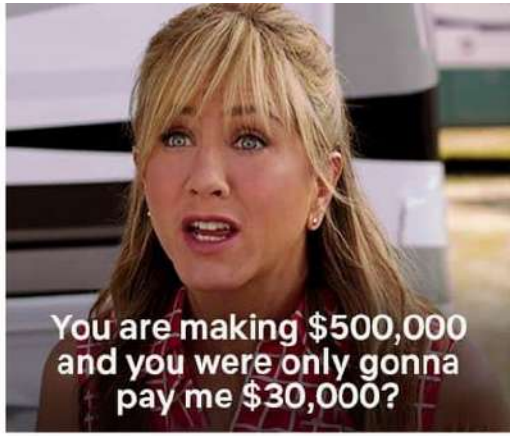
Problem Defining

- Is a problem worth solving?
- What would be the impact of having a solution?
- Is the problem new to the domain?
- Does the problem exist anywhere else? How it is solved in other domains?
- What domain knowledge it requires and can it define away some issues with existing solutions?
- Can I decompose it
  - Pontryagin’s suboptimality principle
  - Morphology analysis

Solution Designing

- What's the mathematical foundation of my solution?
- What are the implementation requirements?
  - Do I need ~~3 years~~ 1B tokens to implement it?
  - What skills?
  - How many people?









Solution Designing

- How do I translate the solution into code?
  - What's my input data?
  - What algorithms I should use? What's their complexity?
  - What are my data structures?
  - How to scale?

Debugging is 3 times more difficult than coding the algorithm.
What happens if coded a solution at the limit of our skills?

Algorithm

What makes one program better than another?

- ***Correctness***
- Speed
- Resources it takes
- What else?



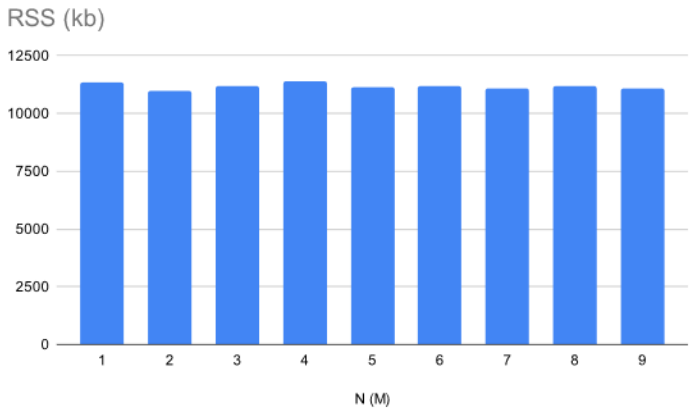
Algorithm Analysis. Example

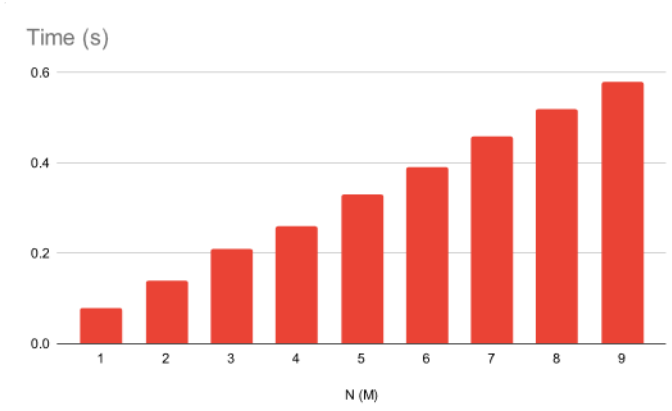
Find the sum of the first N numbers.

$$\sum_{i=0}^{n} i$$

```
def sum_to_n(n):
    total = 0
    for i in range(n + 1):
        total += i
    return total
```

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$$

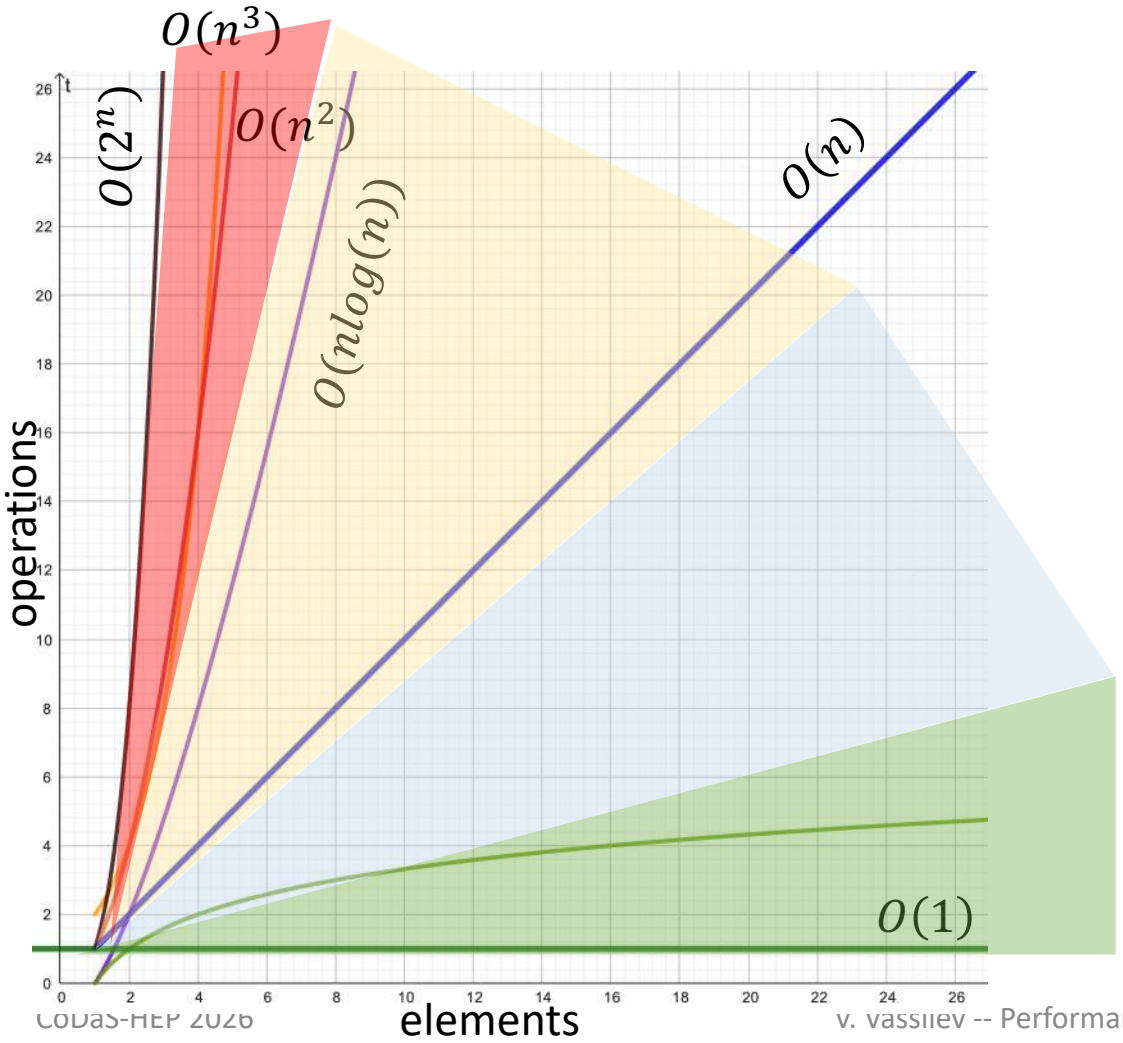




N	Time (s)	RSS (kb)
1M	0.08	11320
2M	0.14	10952
3M	0.21	11172
4M	0.26	11416
5M	0.33	11156
6M	0.39	11180
7M	0.46	11056
8M	0.52	11188
9M	0.58	11088

The Big Picture With Big O

Worst-case algorithm complexity analysis.



Big O is an asymptotic notation which is used to describe the limiting behavior of a function.

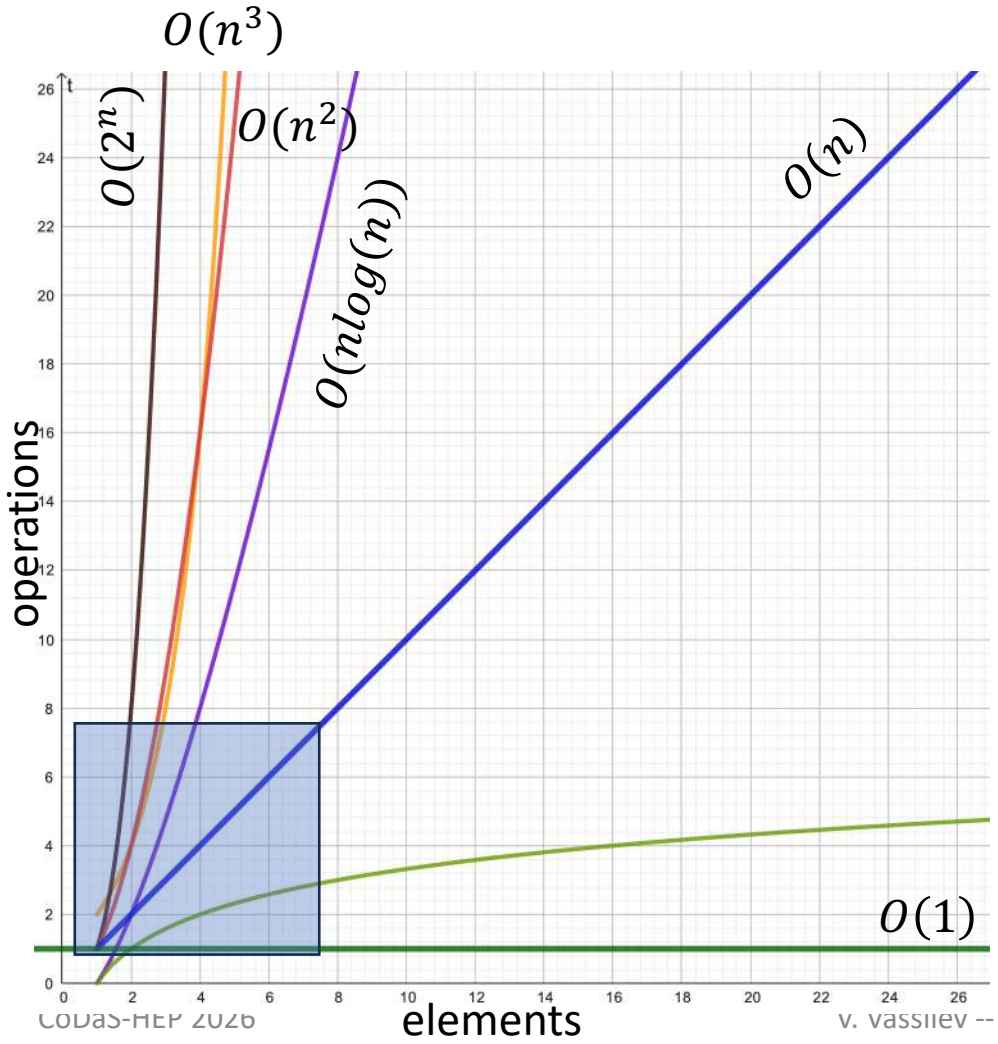
In CS it's used to classify algorithms according to how their run time or space requirements grow as the input size grows.

A very important tool in the programmer's toolbox.

Handout time

The Big Picture With Big O

Worst-case algorithm complexity analysis. O



f(n)	Name	Example
1	Constant	dict/set lookup; array index (Task 6's x in set)
logn	Logarithmic	binary search in sorted data (Task 2 ex2)
n	Linear	1 pass over an array; linear search (Task 2 ex1)
nlogn	Log Linear	any comparison sort; merge sort (Task 3 ex3)
n^2	Quadratic	bubble sort (Task 3 ex1);
n^3	Cubic	naive matrix multiply; triple-nested loop
2^n	Exponential	Halting problem/Generalized Chess, Go
n!	Factorial	Brute forcing travel salesman

Big O. Limitations.

```
list = [1, ..., 63]
list.append(64) # Would that be O(1)?
```



- $O(M*N^2)$ can matter
- Worst-case scenarios happen rarely
- Depending on the properties of the input data the algorithmic complexity can vary

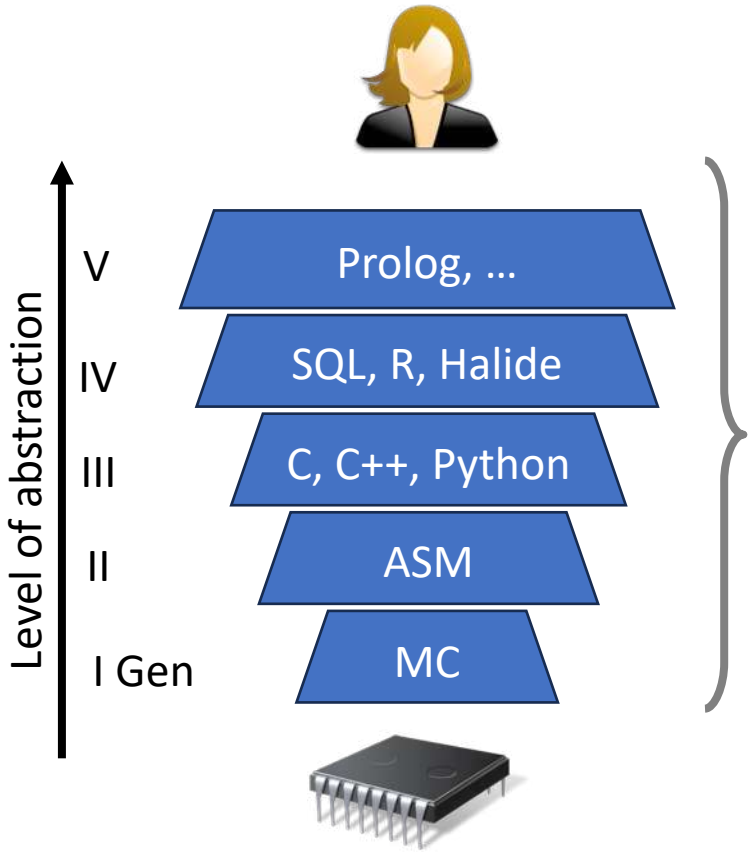
Programming is the art of replacing old bugs with new ones.

- My personal experience

Brief History of Programming Languages

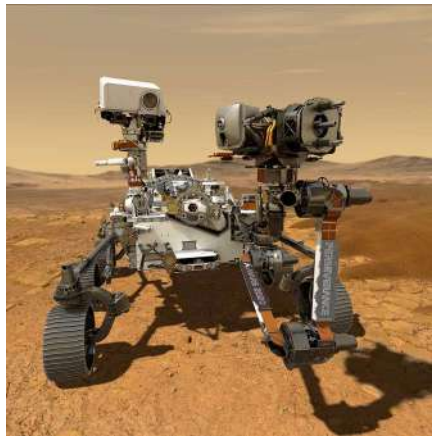
- **1957** Fortran — first compilers (arithmetic expressions, statements, procedures)
- **1960** Algol — first formal definition of PL (BNF grammars, block structure, recursion)
- **1970** Pascal — user-defined types, virtual machines
- **1972** C — structured programming, static type system
- **1985** C++ — object-oriented, exceptions, templates
- **1991** Python — duck typing, ease of use

- ***Important steps in imperative PL**



Know Your Toolchain









C++
- Efficiency
- Stability
- Backward compatibility

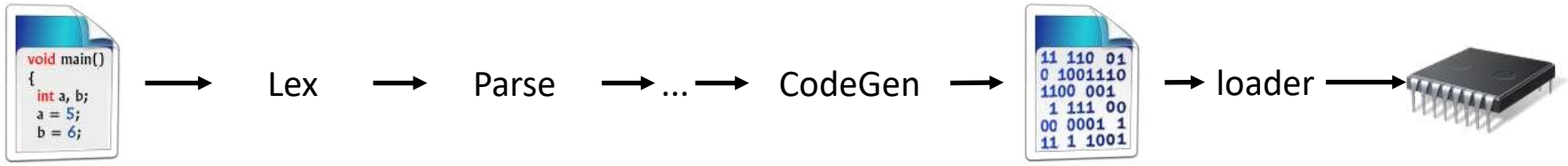
“Prioritizes Performance over Surprise which is sometimes surprising“ T. Winters Link

Python
- Readability
- Simplicity
- Flexibility

“Special cases aren't special enough to break the rules” Zen of Python Link

Translator Classification

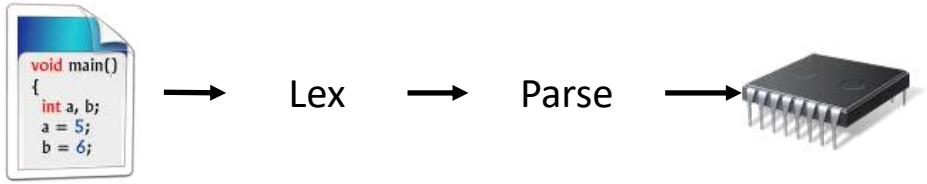
Compiler translates to machine code



Source

Machine code

Interpreter executes the source "directly"

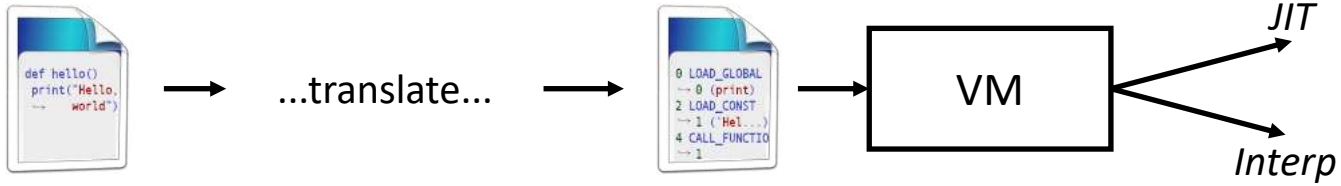


Source

interpretation

❖ The operators in loops are analyzed again and again

Hybrid compiler interprets intermediate representation



Source

Intermediate Representation (eg. bytecode)

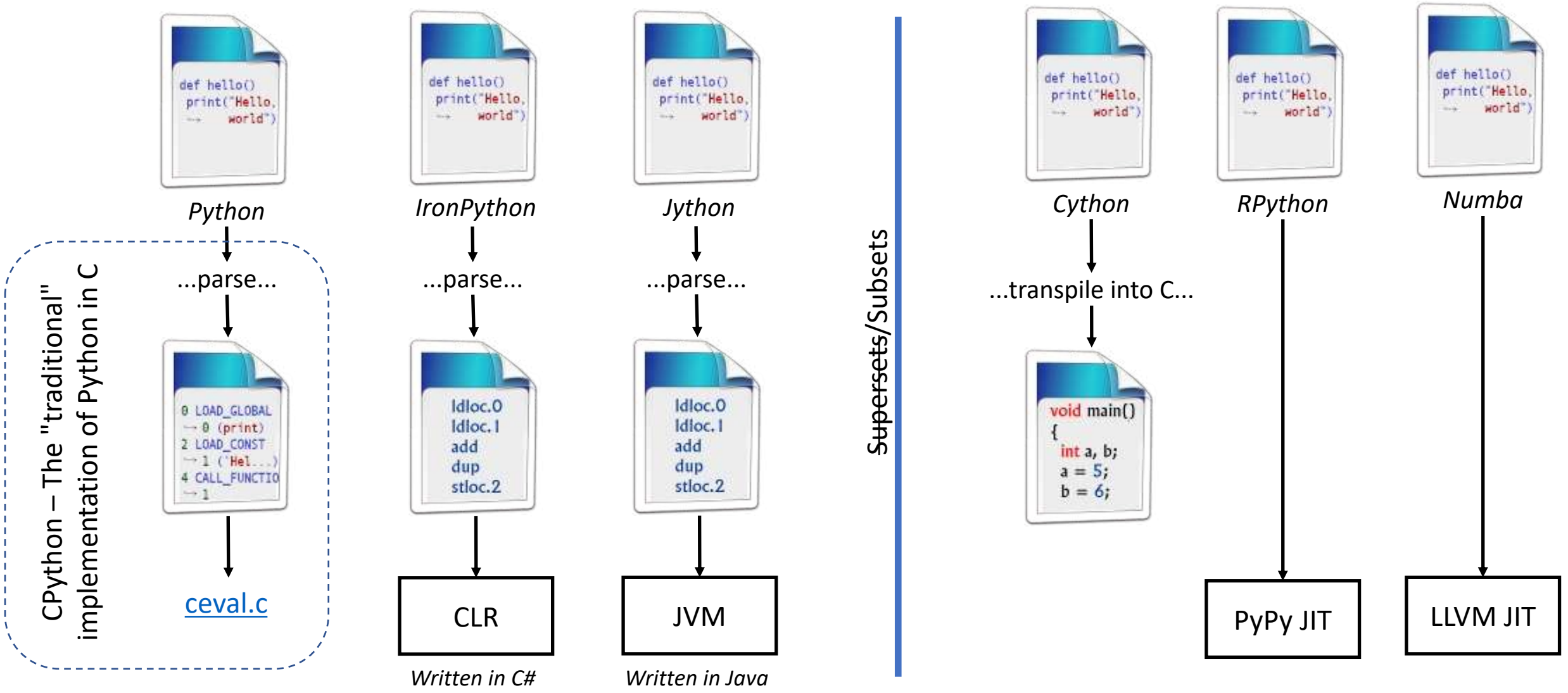
❖ The source translates to code for virtual machine (VM)

❖ The VM runtime interprets the code simulating real machine

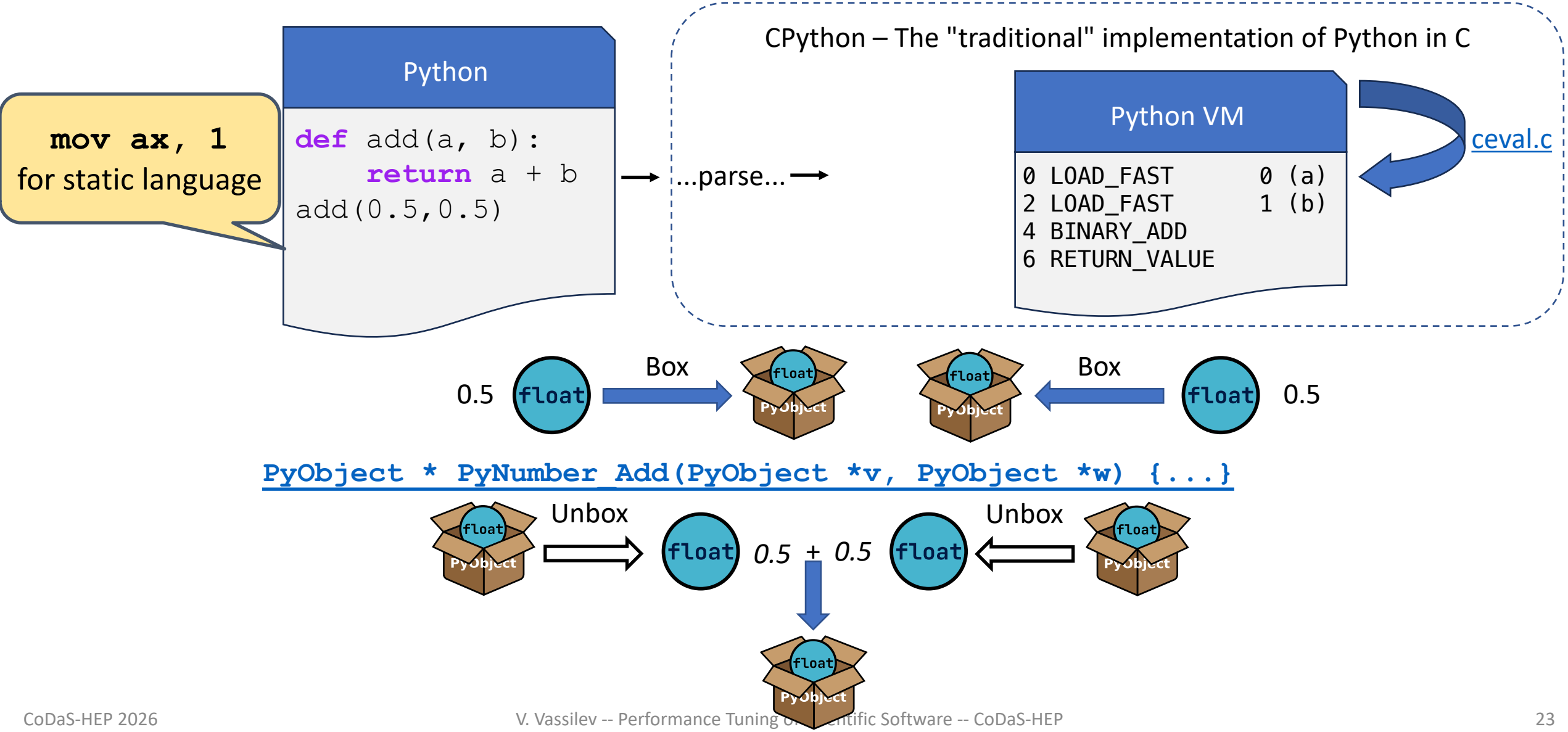
❖ The Just-In-Time compiler translates the representation at startup time.

Python Jungle

Trading Flexibility for Performance



Trading Performance For Flexibility



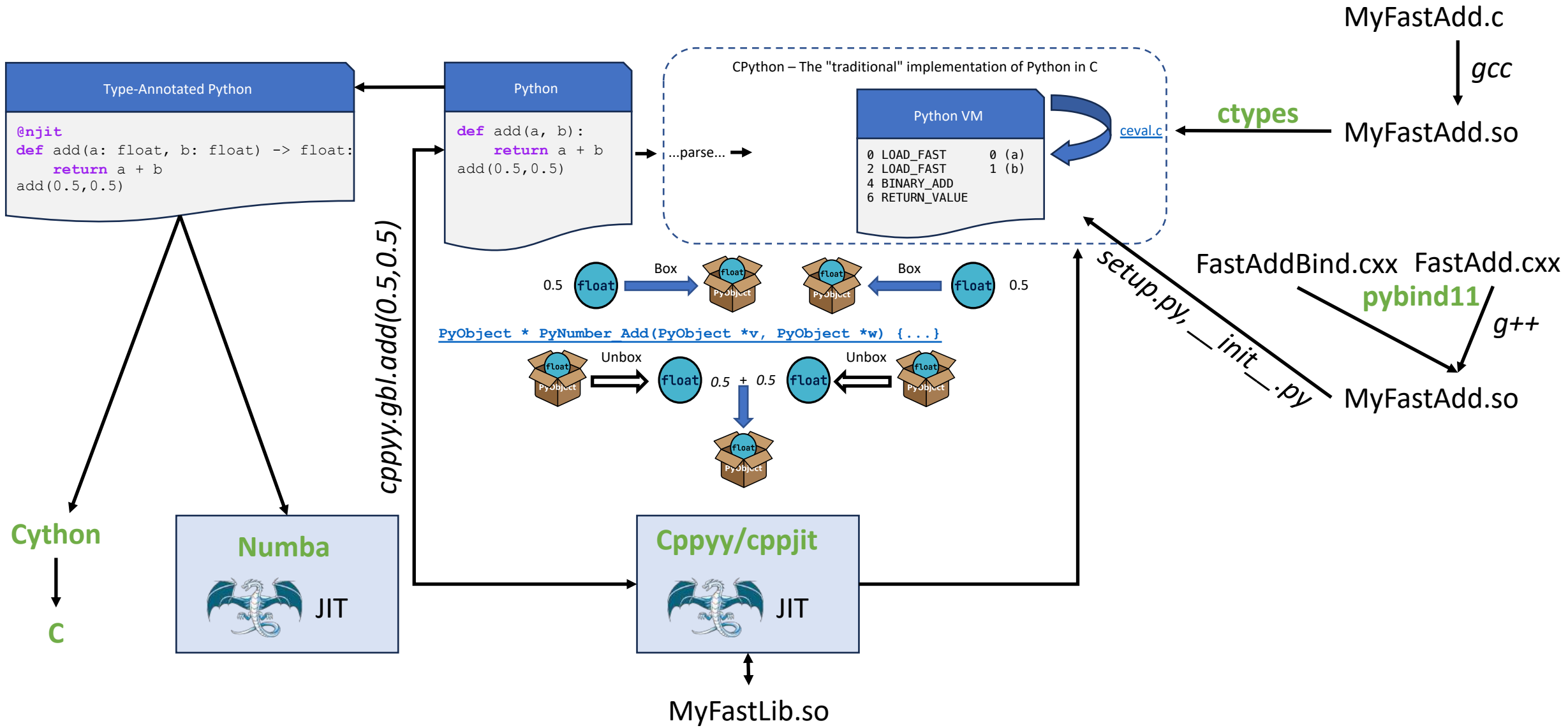
Expressing Optimization Assumptions

- Use built-in functions and libraries
  They are heavily optimized and implemented in C. Eg. use sum() instead of manually iterating
- Avoid global state
  Global variables are slower to access and hinder optimizations
- Minimize the use of loops
  Use map, filter, reduce from libraries such as NumPy
- Use proper data structures
  Choose the correct data structure for the task. Prefer immutable types such as tuple and frozenset.
  Prefer pre-allocated, pre-resized types to avoid amortization effects
- Be mindful with string operations such as concatenation

Expressing Optimization Assumptions

- Avoid excessive object creation
  Think about the created objects especially in tight loops
- Avoid memory leaks in resource release
  Use the `with` construct to manage resources such as files or network connections
- Avoid using try/except constructs for control flow
  Exceptions are designed for error handling and less so for other things
- Avoid unnecessary abstractions
  Abstractions introduce often indirection which can be inefficient
- Use declarative style
  List comprehensions and generator expressions allow sometimes better performance.
- Use memoisation techniques to avoid recomputation
  Cache results with `functools.lru_cache`

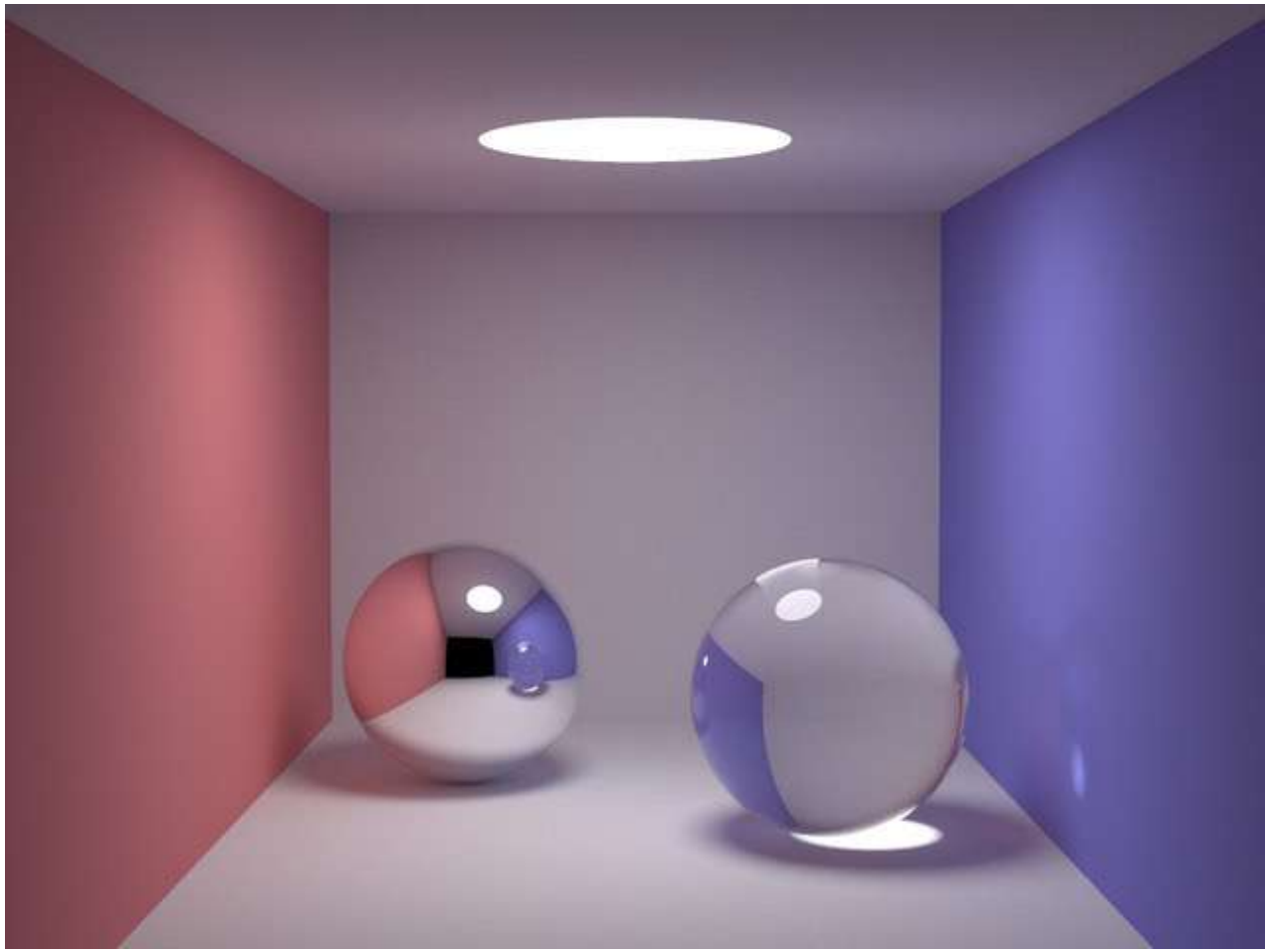
From Bindings to Full Language InterOp



“The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; *premature optimization is the root of all evil (or at least most of it) in programming*.”
- Donald Knuth

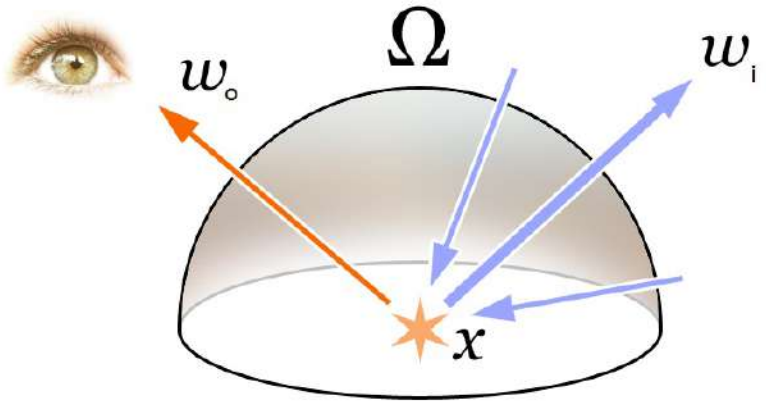
Virtual Photography with SmallPt

A global illumination renderer. It is 99 lines of C++, is open source, and renders the above scene using unbiased **Monte Carlo** path tracing.



Back to handouts

What did we compute?



$$L_{\mathrm{r}}(\mathbf{x}, \omega_{\mathrm{o}}, \lambda, t) = \int_{\Omega} f_{\mathrm{r}}(\mathbf{x}, \omega_{\mathrm{i}}, \omega_{\mathrm{o}}, \lambda, t) L_{\mathrm{i}}(\mathbf{x}, \omega_{\mathrm{i}}, \lambda, t)(\omega_{\mathrm{i}} \cdot \mathbf{n}) \, \mathrm{d}\, \omega_{\mathrm{i}}$$

We estimated an integral.

- **the integrand is a physical process** — scatter, absorb, emit
- **the domain is too big to enumerate** — so sample it
- **the estimator is Monte Carlo** — error ~ 1/√N

That is why more samples means less noise. That is why it parallelized so well: the samples are independent.

This Shape Is Everywhere

$$\text{output} = \int \text{kernel}(\text{state}, \text{state}') \cdot \text{density}(\text{state}') \; d(\text{state}')$$

domain	**kernel**	**output**
rendering	BRDF	radiance
neutron transport	scattering cross-section	flux
detector simulation	interaction model	hits
HEP likelihood	detector response	expected events

Fredholm equation of the second kind. Same estimator. Same 1/√N. Same parallelism.

GEANT4 is a path tracer that happens to shoot pions.