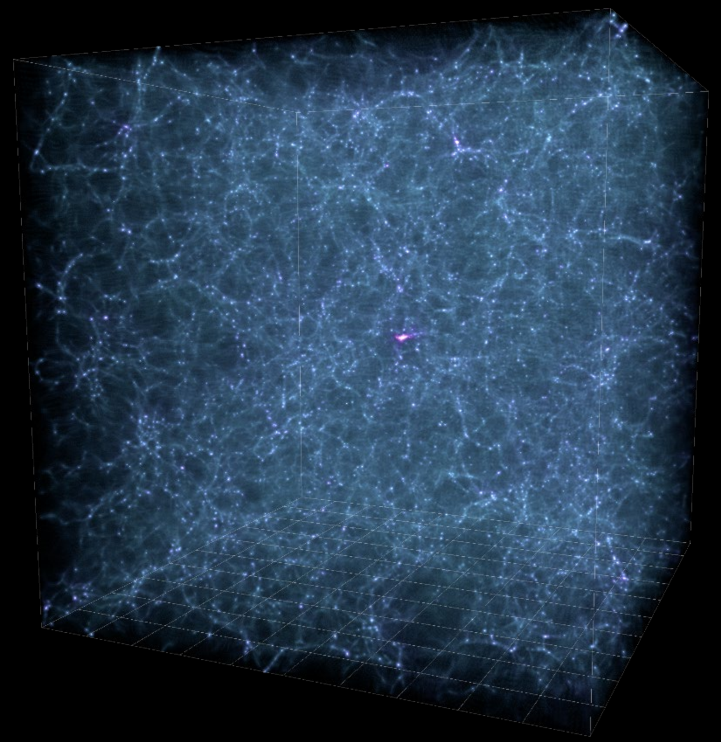
The upcoming wall of software complexity in computational sciences

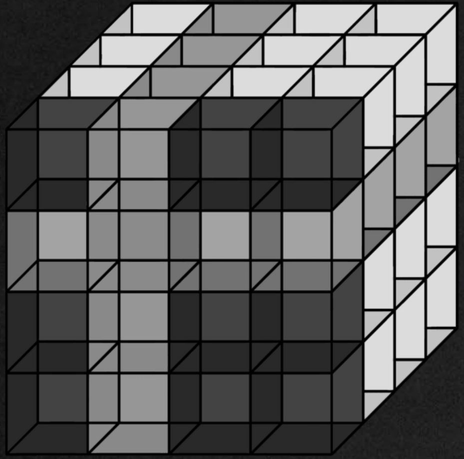
Journées Nationales du GDR GPL – June 2022 – Vannes, France

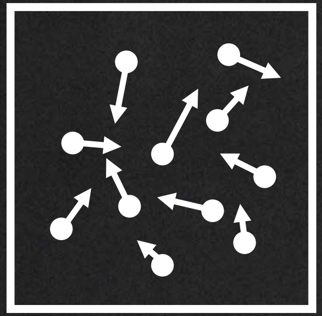
Vincent Reverdy, CNRS IN2P3/INS2I, Laboratoire d'Annecy de Physique des Particules



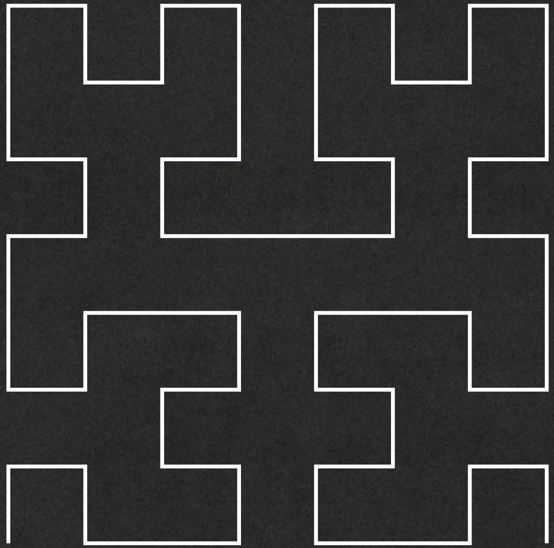
LAPP l'Observatoire de Paris LUTH | PSL ILLINOIS NCSA

From nice drawings on a blackboard…

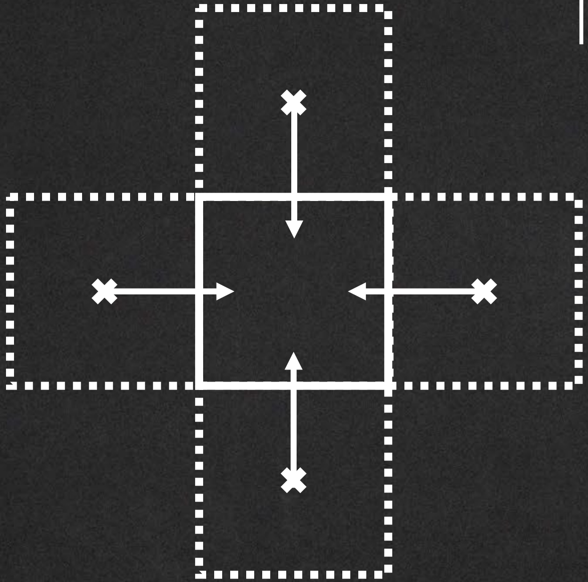




$$G_{\mu\nu} = \frac{8\pi G}{c^4} T_{\mu\nu}$$



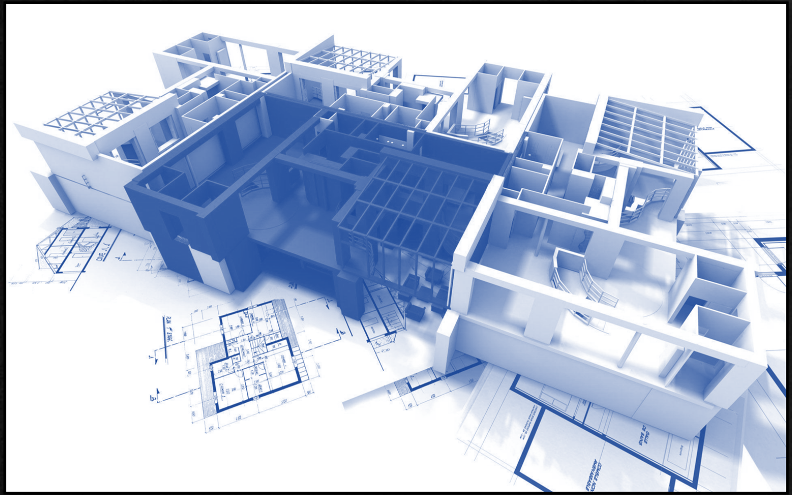
$$\frac{d^2\eta}{d\lambda^2} \approx -\frac{2a'}{a}\frac{d\eta}{d\lambda}\frac{d\eta}{d\lambda} - \frac{2}{c^2}\frac{d\Phi}{d\lambda}\frac{d\eta}{d\lambda} + 2\frac{\partial\Phi}{\partial\eta}\left(\frac{d\eta}{d\lambda}\right)^2$$



… to unmaintainable monsters

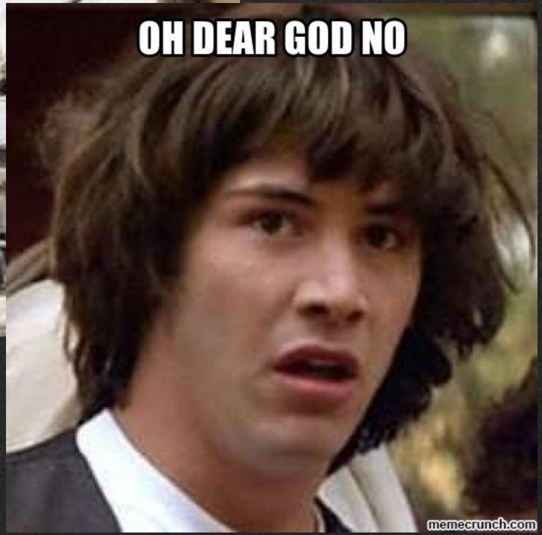
```
type, Kind>::value) && (std::tuple_size<typename std::remove_cv<typename std::remove_reference<Tuple>::type>::type>::value >= 1)>::type> static constexpr Kind accumulate(Tuple&& tuple);
template <typename Kind, class Operation = std::plus<Kind>, typename... Kinds, class = typename std::enable_if<(std::is_convertible<Kind, int>::value) && (std::is_convertible<typename std::
result_of<Operation(Kind, Kind)>::type, Kind>::value)>::type> static constexpr Kind accumulate(const Kind value, const Kinds... values);
template <typename Kind, class Operation = std::plus<Kind>, class = typename std::enable_if<(std::is_convertible<Kind, int>::value) && (std::is_convertible<typename std::result_of<Operation(
Kind, Kind)>::type, Kind>::value)>::type> static constexpr Kind accumulate(const Kind value);
template <typename Kind, int Exponent = 1, int One = 1, bool Greater = (Exponent > 1), bool Less = (Exponent < 0), bool Equal = (Exponent == 1), class = typename std::enable_if<std::
is_convertible<Kind, int>::value>::type> static constexpr Kind pow(const Kind value);
template <typename Integer, Integer Zero = Integer(), Integer One = Integer(1), Integer Ones = ~Zero, Integer Size = sizeof(Integer)*std::numeric_limits<unsigned char>::digits, class =
typename std::enable_if<((std::is_integral<Integer>::value) ? (std::is_unsigned<Integer>::value) : (std::is_convertible<Integer, int>::value)) && (!std::is_floating_point<Integer>::value)>::
type> static constexpr Integer block(const Integer location = Zero, const Integer length = Size);
template <typename Integer, Integer Index = Integer(), Integer Zero = Integer(), Integer One = Integer(1), Integer Condition = (Index+One <= sizeof(Integer)*std::numeric_limits<unsigned char
>::digits), class = typename std::enable_if<((std::is_integral<Integer>::value) ? (std::is_unsigned<Integer>::value) : (std::is_convertible<Integer, int>::value)) && (!std::is_floating_point<
Integer>::value)>::type> static constexpr Integer periodic(const Integer period = One, const Integer offset = Zero);
template <typename Integer, Integer Index = Integer(), Integer Zero = Integer(), Integer One = Integer(1), Integer Size = sizeof(Integer)*std::numeric_limits<unsigned char>::digits, Integer
Condition = (Index+One <= Size), class = typename std::enable_if<((std::is_integral<Integer>::value) ? (std::is_unsigned<Integer>::value) : (std::is_convertible<Integer, int>::value)) && (!
std::is_floating_point<Integer>::value)>::type> static constexpr Integer comb(const Integer period = One, const Integer offset = Zero);
template <typename Integer, Integer Mask = ~Integer(), Integer Step = Integer(1), Integer One = Integer(1), Integer Ones = ~Integer(), Integer Condition = (Step+One <= sizeof(Integer)*std::
numeric_limits<unsigned char>::digits), class = typename std::enable_if<((std::is_integral<Integer>::value) ? (std::is_unsigned<Integer>::value) : (std::is_convertible<Integer, int>::value))
&& (!std::is_floating_point<Integer>::value)>::type> static constexpr Integer nhp(const Integer value);
template <typename Integer, Integer Mask = ~Integer(), Integer Step = Integer(1), Integer One = Integer(1), Integer Ones = ~Integer(), Integer Condition = (Step+One <= nhp<Integer>(sizeof(
Integer)*std::numeric_limits<unsigned char>::digits)), class = typename std::enable_if<((std::is_integral<Integer>::value) ? (std::is_unsigned<Integer>::value) : (std::is_convertible<Integer,
 int>::value)) && (!std::is_floating_point<Integer>::value)>::type> static constexpr Integer bhsmask(const Integer value);
template <typename Integer, Integer Mask = ~Integer(), Integer Step = nhp<Integer>(sizeof(Integer)*std::numeric_limits<unsigned char>::digits), Integer Zero = Integer(), Integer One = Integer
(1), Integer Ones = ~Zero, Integer Size = nhp<Integer>(sizeof(Integer)*std::numeric_limits<unsigned char>::digits), Integer Temporary = block<Integer>(Step, Step), class = typename std::
enable_if<((std::is_integral<Integer>::value) ? (std::is_unsigned<Integer>::value) : (std::is_convertible<Integer, int>::value)) && (!std::is_floating_point<Integer>::value)>::type> static
constexpr Integer lzcnt(const Integer value);
template <typename Integer, Integer Mask = ~Integer(), Integer Step = nhp<Integer>(sizeof(Integer)*std::numeric_limits<unsigned char>::digits), Integer Zero = Integer(), Integer One = Integer
(1), Integer Ones = ~Zero, Integer Size = nhp<Integer>(sizeof(Integer)*std::numeric_limits<unsigned char>::digits), Integer Temporary = periodic<Integer>(Step*(One+One)+(Step == Zero)), class
 = typename std::enable_if<((std::is_integral<Integer>::value) ? (std::is_unsigned<Integer>::value) : (std::is_convertible<Integer, int>::value)) && (!std::is_floating_point<Integer>::value
)>::type> static constexpr Integer tzcnt(const Integer value);
template <typename Integer, Integer Mask = ~Integer(), Integer Step = Integer(), Integer One = Integer(1), Integer Condition = (Step+One <= sizeof(Integer)*std::numeric_limits<unsigned char
>::digits), Integer Temporary = ((Condition) ? ((Mask >> Step) & (Condition)) : (Condition)), class = typename std::enable_if<((std::is_integral<Integer>::value) ? (std::is_unsigned<Integer
>::value) : (std::is_convertible<Integer, int>::value)) && (!std::is_floating_point<Integer>::value)>::type> static constexpr Integer popcnt(const Integer value);
template <typename Integer, Integer Mask = ~Integer(), Integer Step = Integer(), Integer Shift = Integer(), Integer One = Integer(1), Integer Condition = (Step+One <= sizeof(Integer)*std::
numeric_limits<unsigned char>::digits), Integer Temporary = ((Condition) ? ((Mask >> Step) & (Condition)) : (Condition)), class = typename std::enable_if<((std::is_integral<Integer>::value) ?
 (std::is_unsigned<Integer>::value) : (std::is_convertible<Integer, int>::value)) && (!std::is_floating_point<Integer>::value)>::type> static constexpr Integer pext(const Integer value);
template <typename Integer, Integer Mask = ~Integer(), Integer Step = Integer(), Integer Shift = Integer(), Integer One = Integer(1), Integer Condition = (Step+One <= sizeof(Integer)*std::
numeric_limits<unsigned char>::digits), Integer Temporary = ((Condition) ? ((Mask >> Step) & (Condition)) : (Condition)), class = typename std::enable_if<((std::is_integral<Integer>::value) ?
 (std::is_unsigned<Integer>::value) : (std::is_convertible<Integer, int>::value)) && (!std::is_floating_point<Integer>::value)>::type> static constexpr Integer pdep(const Integer value);
template <typename Integer, Integer Mask = ~Integer(), Integer Period = sizeof(Integer)*std::numeric_limits<unsigned char>::digits, Integer Step = Integer(), Integer Count = Integer(),
Integer Zero = Integer(), Integer One = Integer(1), Integer Size = sizeof(Integer)*std::numeric_limits<unsigned char>::digits, Integer Condition = (Step+One <= Size), Integer Population =
popcnt<Integer>(Mask)+((Period-(popcnt<Integer>(Mask)%Period))*(popcnt<Integer>(Mask)%Period != Zero)), Integer Destination = (((((Condition) ? ((Mask >> Step) & (Condition)) : (Condition)) ?
 (Count) : (Zero))%((Population+(Period*(Population+One <= Period)))/Period))+((((Condition) ? ((Mask >> Step) & (Condition)) : (Condition)) ? (Count) : (Zero))/((Population+(Period*(
Population+One <= Period)))/Period)), Integer Temporary = ((Condition) ? ((Mask >> Step) & (Condition)) : (Condition)) && (Destination < Size), class = typename std::enable_if<((std::
is_integral<Integer>::value) ? (std::is_unsigned<Integer>::value) : (std::is_convertible<Integer, int>::value)) && (!std::is_floating_point<Integer>::value)>::type> static constexpr Integer
itlc(const Integer value);
template <typename Integer, Integer Mask = ~Integer(), Integer Period = sizeof(Integer)*std::numeric_limits<unsigned char>::digits, Integer Step = Integer(), Integer Count = Integer(),
Integer Zero = Integer(), Integer One = Integer(1), Integer Size = sizeof(Integer)*std::numeric_limits<unsigned char>::digits, Integer Condition = (Step+One <= Size), Integer Population =
popcnt<Integer>(Mask)+((Period-(popcnt<Integer>(Mask)%Period))*(popcnt<Integer>(Mask)%Period != Zero)), Integer Destination = (((((Condition) ? ((Mask >> Step) & (Condition)) : (Condition)) ?
 (Count) : (Zero))%Period)*(Population/Period))+((((Condition) ? ((Mask >> Step) & (Condition)) : (Condition)) ? (Count) : (Zero))/Period), Integer Temporary = ((Condition) ? ((Mask >> Step)
& (Condition)) : (Condition)) && (Destination < Size), class = typename std::enable_if<((std::is_integral<Integer>::value) ? (std::is_unsigned<Integer>::value) : (std::is_convertible<Integer,
 int>::value)) && (!std::is_floating_point<Integer>::value)>::type> static constexpr Integer dtlc(const Integer value);
template <typename Integer, Integer Mask = ~Integer(), Integer Length = sizeof(Integer)*std::numeric_limits<unsigned char>::digits, bool Msb = false, std::size_t Step = Integer(), Integer
Zero = Integer(), Integer Direction = ((!Msb) || ((Length*(Step+1)) <= (sizeof(Integer)*std::numeric_limits<unsigned char>::digits))), Integer Left = ((Msb) ? (sizeof(Integer)*std::
numeric_limits<unsigned char>::digits-(Length*(Step+1))) : (Length*Step))*(Direction), Integer Right = ((Msb) ? ((Length*(Step+1))-sizeof(Integer)*std::numeric_limits<unsigned char>::digits)
 : (Length*Step))*(!Direction), Integer Condition = ((Left+1 <= sizeof(Integer)*std::numeric_limits<unsigned char>::digits) && (Right+1 <= sizeof(Integer)*std::numeric_limits<unsigned char>::
digits) && (Right+1 <= Length)), class Tuple, Integer Count = ((std::tuple_size<typename std::remove_cv<typename std::remove_reference<Tuple>::type>::type>::value)-(Step+1)), class = typename
 std::enable_if<((std::is_integral<Integer>::value) ? (std::is_unsigned<Integer>::value) : (std::is_convertible<Integer, int>::value)) && (!std::is_floating_point<Integer>::value) && (std::
is_convertible<typename std::tuple_element<Step, typename std::remove_cv<typename std::remove_reference<Tuple>::type>::type>::type, Integer>::value)>::type> static constexpr Integer glue(
```

Expectation vs reality



Expectation

?



Reality

An introductory tale

Let's start with a story

Once upon a time...

...in a galaxy far far away...

Let's start with a story

...on a small piece of rock...

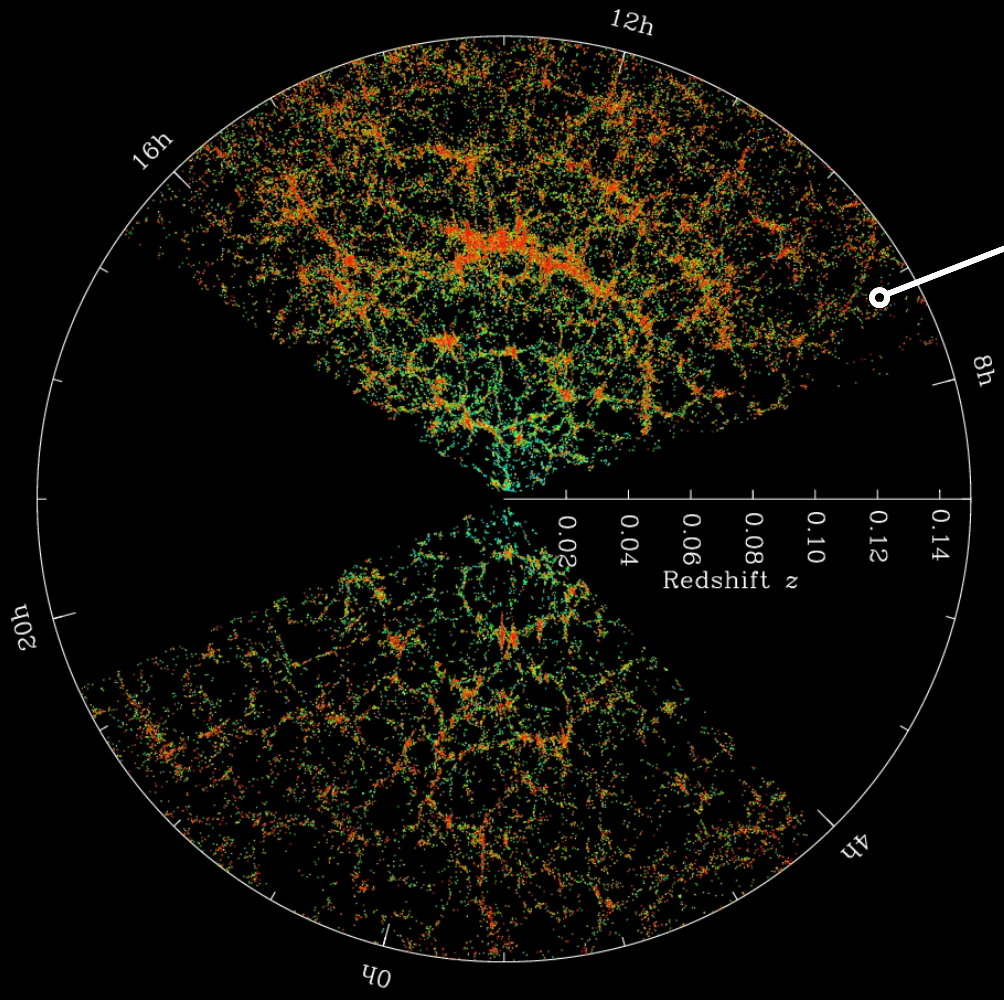
...wandering aimlessly in a vast Universe...

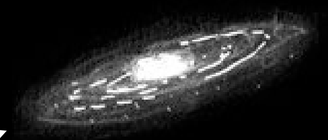
Let's start with a story

...a team of astrophysicists was wondering about the nature of life, the Universe, and everything.

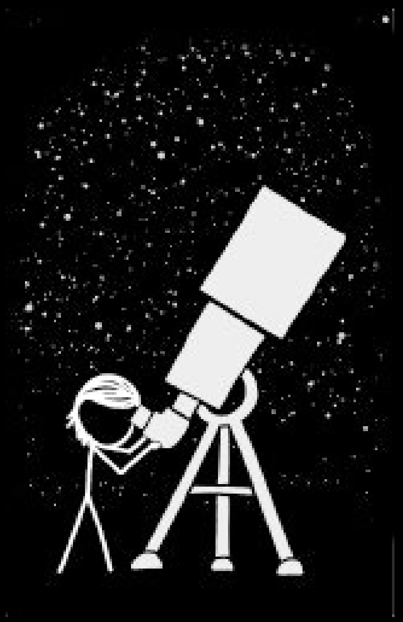
Let's start with a story

Why do galaxies form a cosmic web?



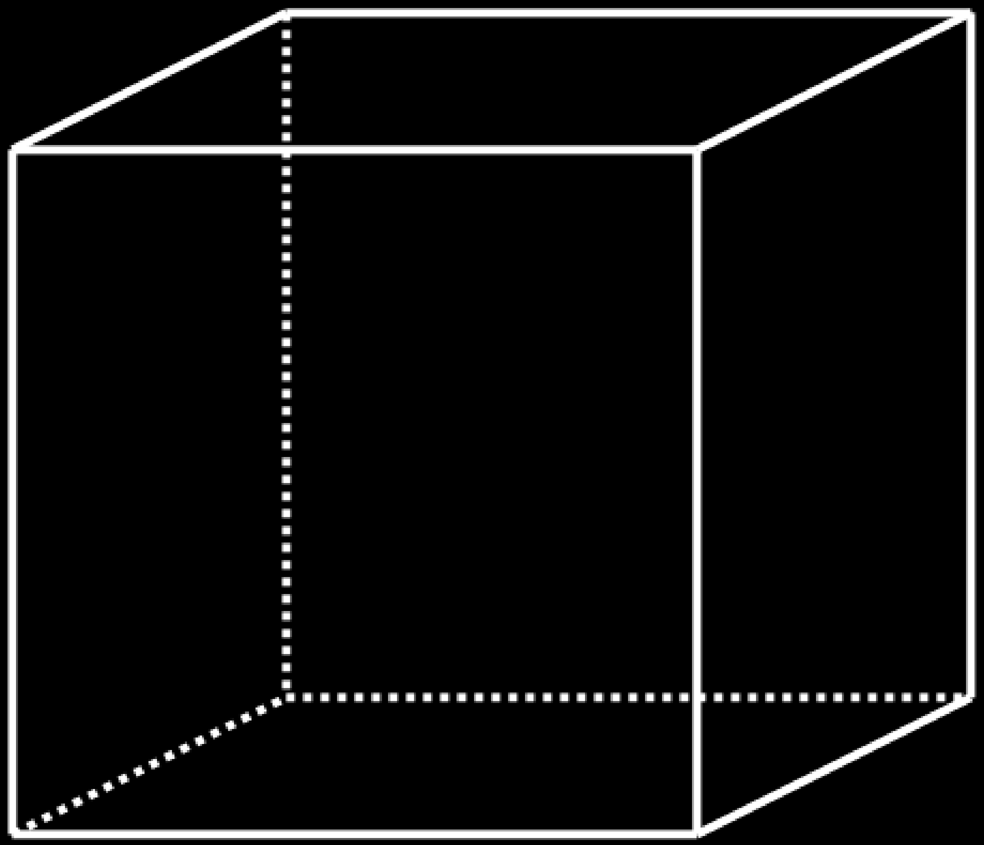


(every point is a galaxy containing billions of stars)



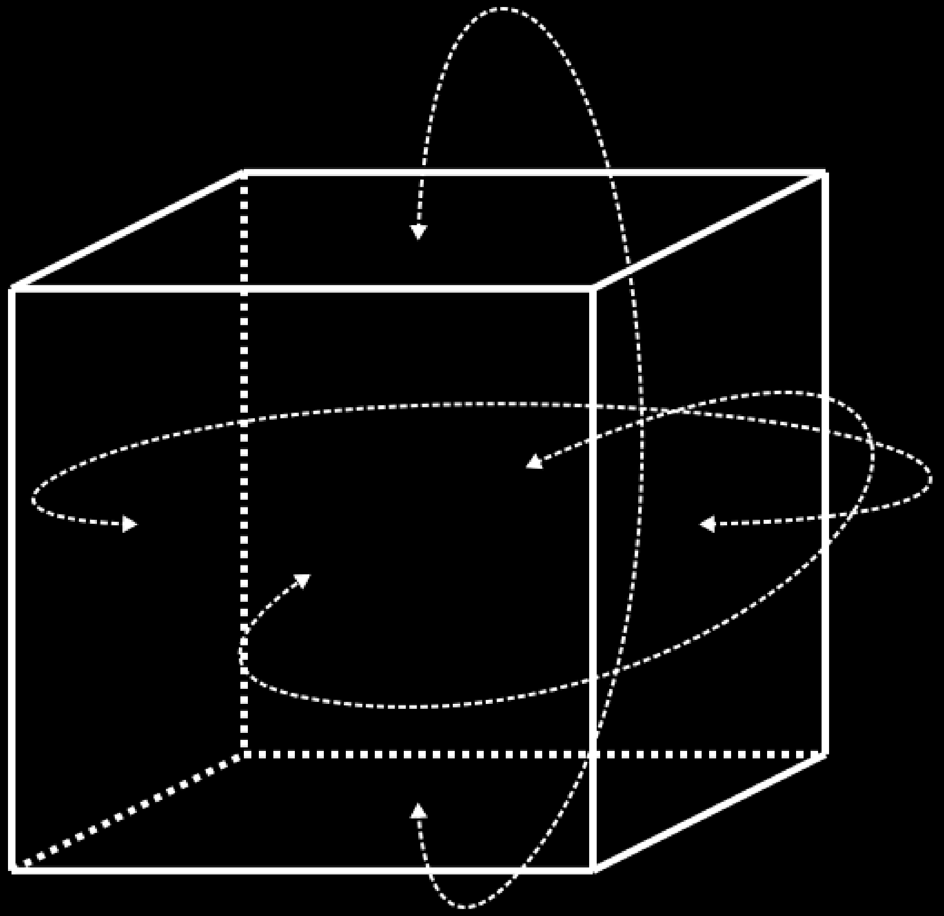
Let's run simulations to better understand where it comes from!

A crash course in astrophysics simulations



They said: "Let's take an enormous box...

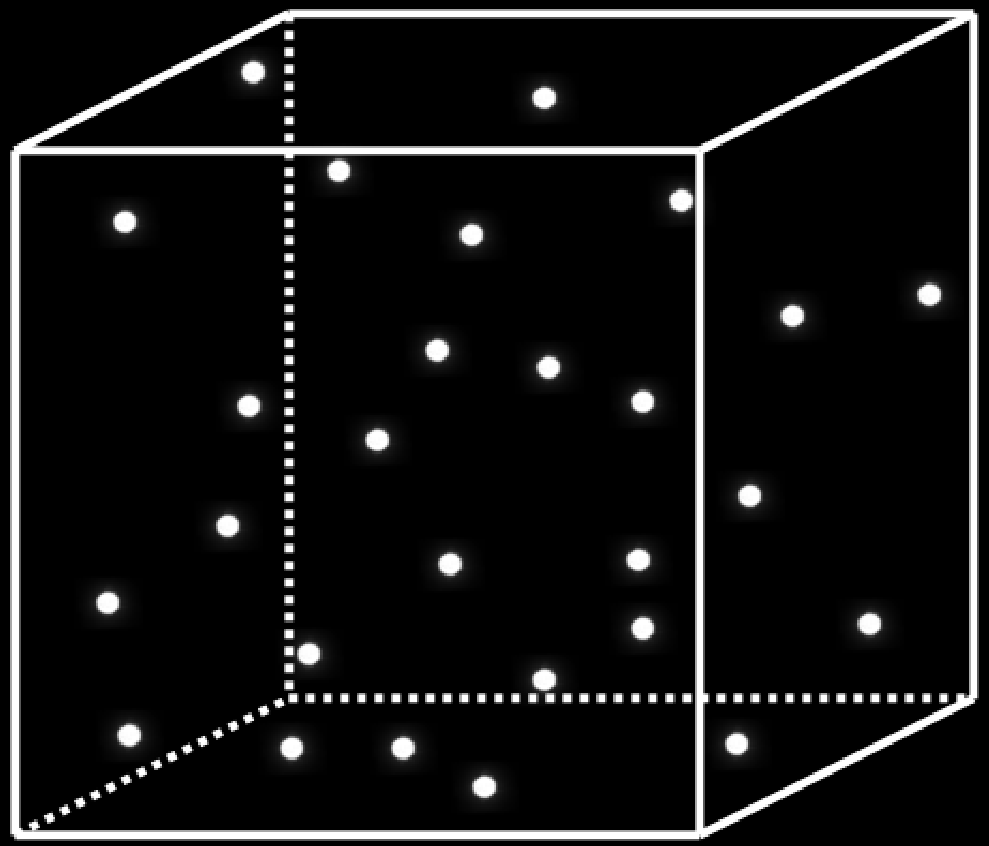
A crash course in astrophysics simulations



...with periodic boundary conditions...

(3D torus)

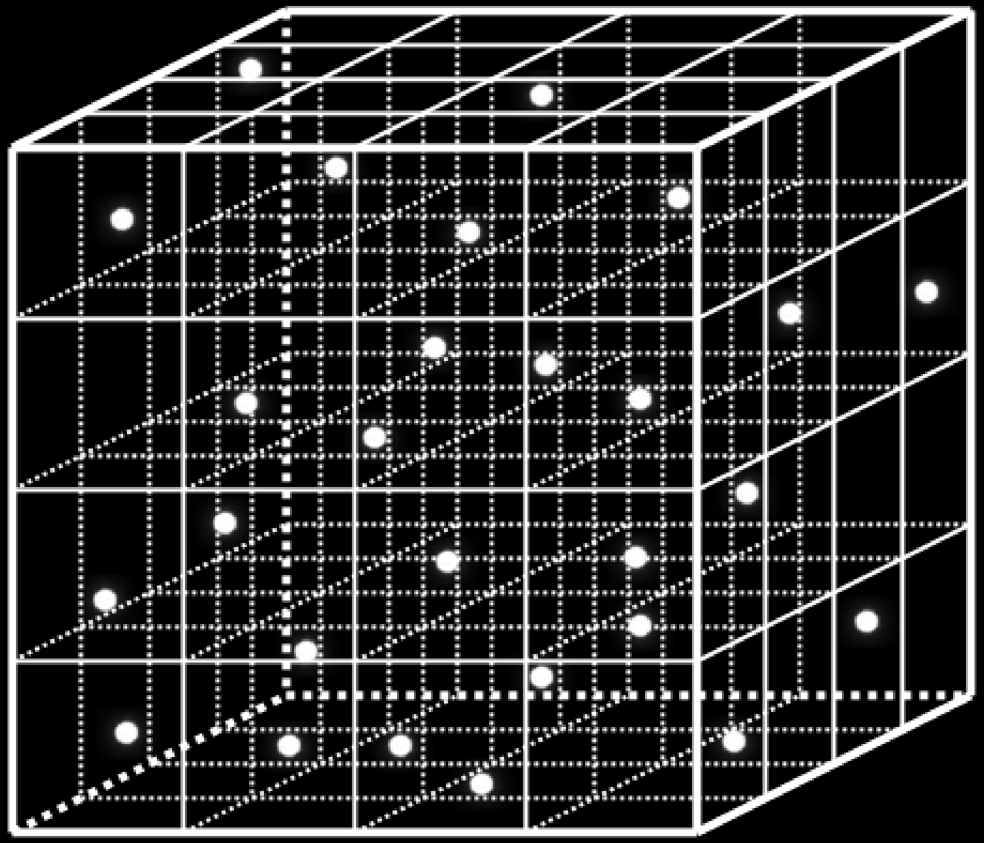
A crash course in astrophysics simulations



...and let's fill that enormous box with particles weighing the mass of millions of suns...

(note: yes that's kind of huge)

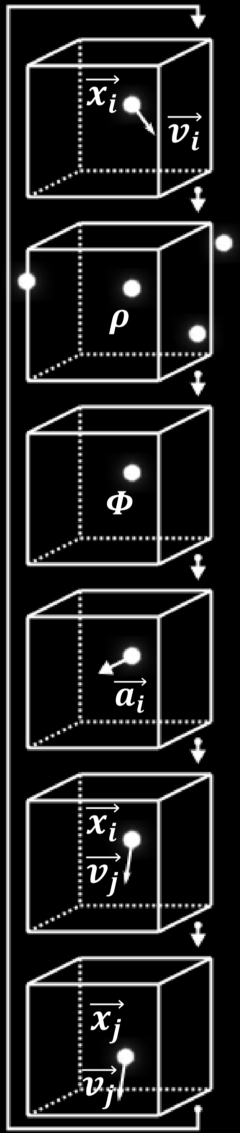
A crash course in astrophysics simulations



Now, divide the box in cells using a regular grid and apply the following recipe:

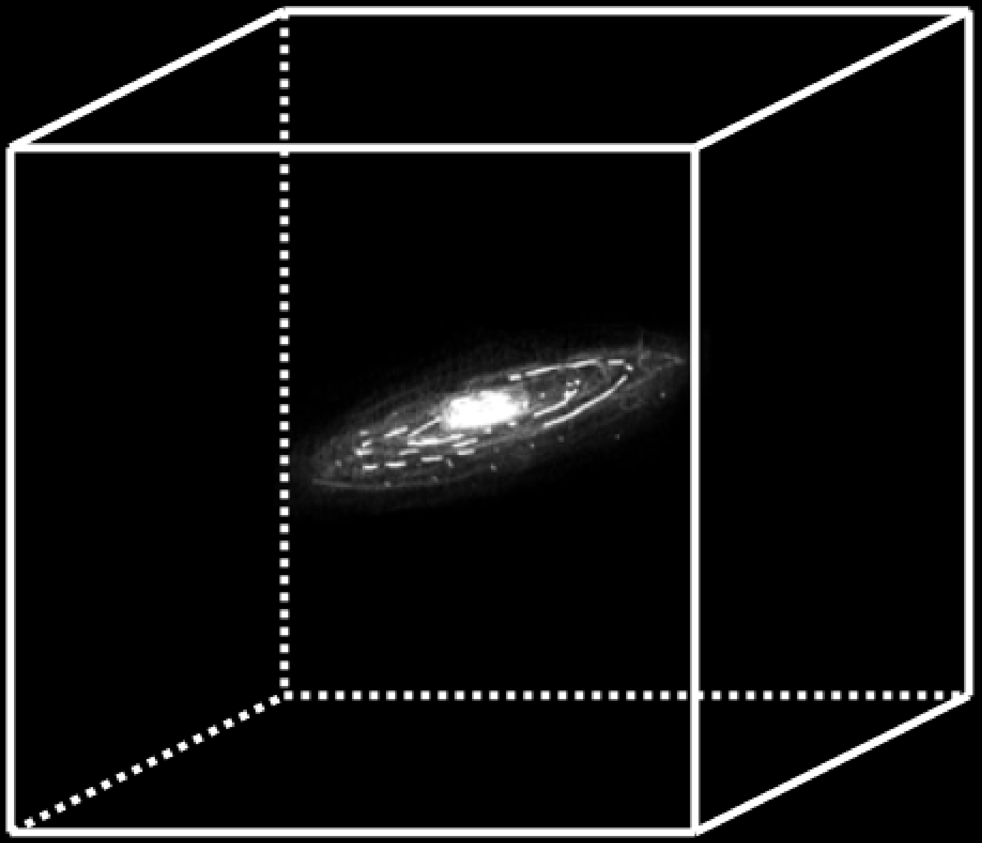
A crash course in astrophysics simulations

- 7) Restart at 1) with uploaded position $\vec{x}$ and speed $\vec{v}$



- 1) For each cell c containing particles with position $\vec{x_i}$ and velocity $\vec{v_i}$
- 2) Interpolate density ρ in cell c depending on surrounding particles
- 3) From ρ compute the gravitational potential Φ
- 4) From Φ interpolate back the acceleration $\vec{a}$ at position $\vec{x_i}$
- 5) From $\vec{a}$ compute the new speed $\vec{v_j}$ of each particle
- 6) From $\vec{v_j}$ compute the new position $\vec{x_j}$ of each particle

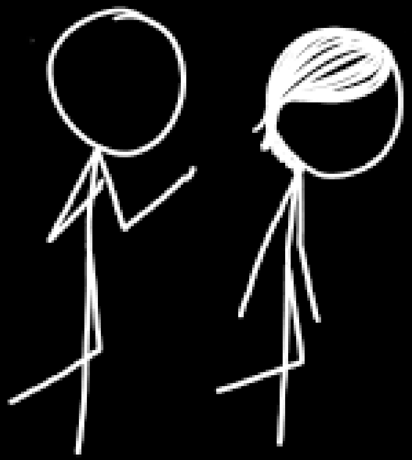
A crash course in astrophysics simulations



Using this recipe with millions of particles
we can simulate galaxy formation!

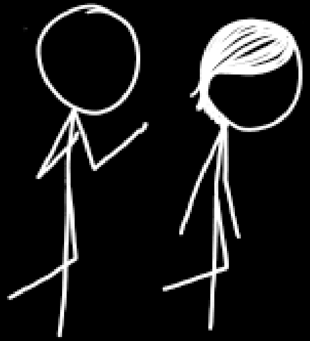
From galaxies to expanding the Universe

Simulating galaxies is nice...



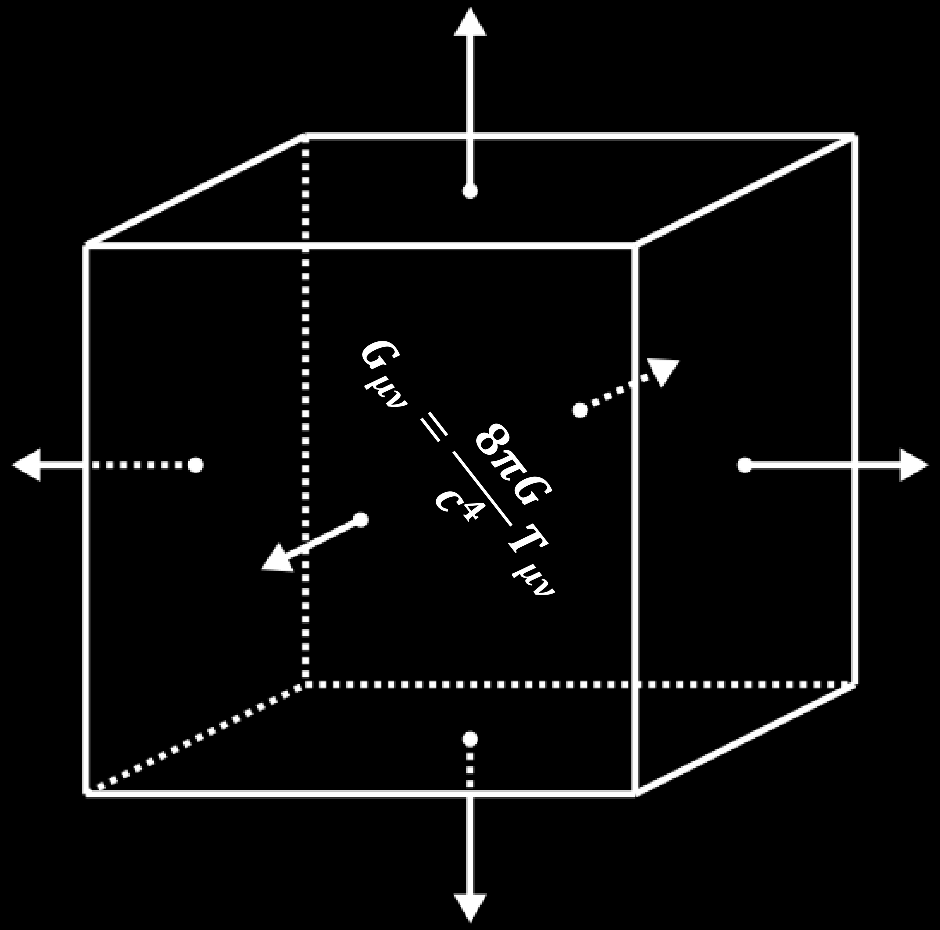
...but simulating the expansion of the Universe
requires to take the approach to a whole new level...

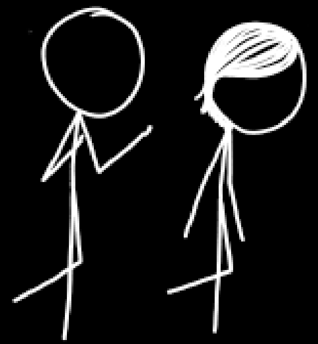
From galaxies to expanding the Universe



First they took a supercomputer.

From galaxies to expanding the Universe

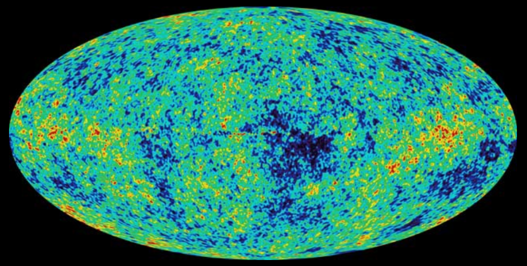




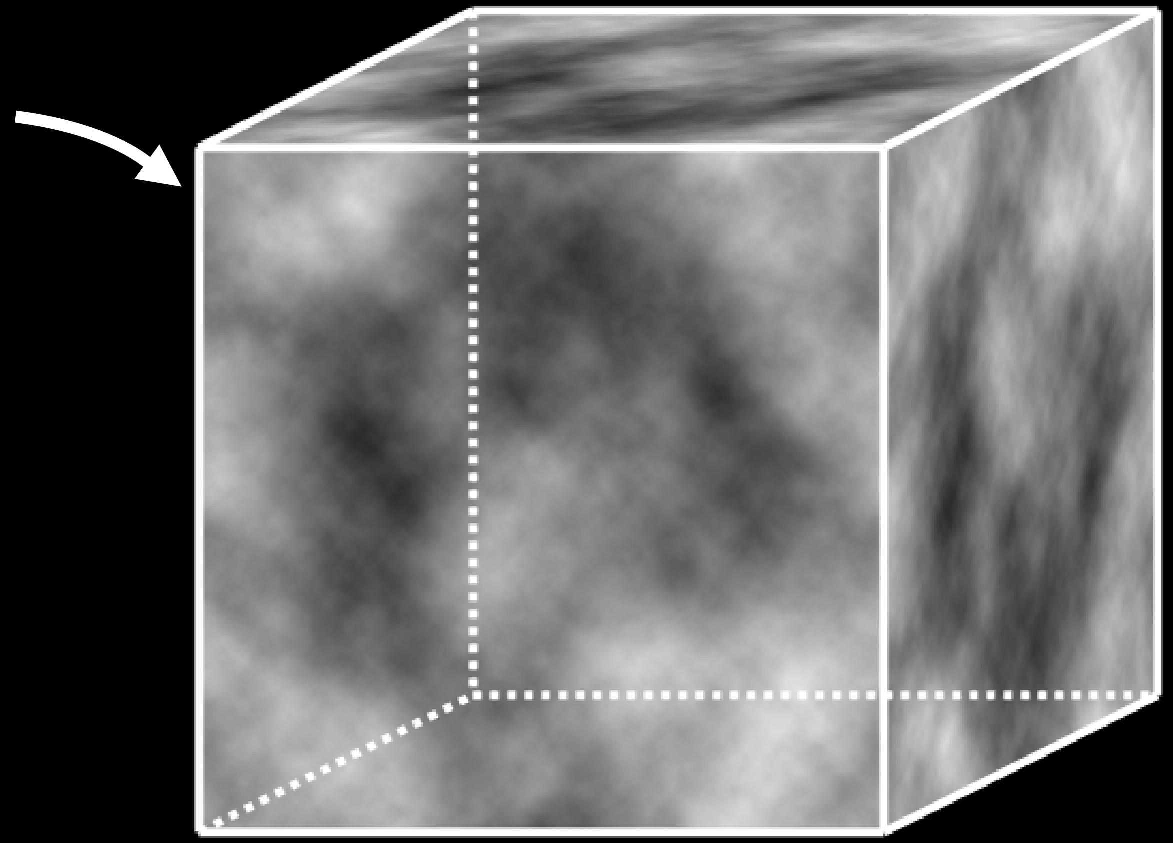
Second they made the box expand as the Universe does according to General Relativity

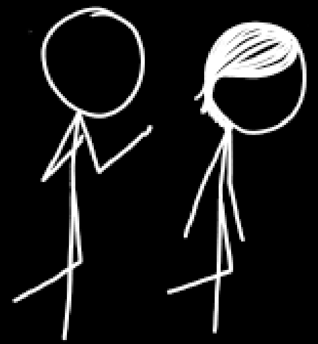
(considering a homogeneous FLRW metric)

From galaxies to expanding the Universe



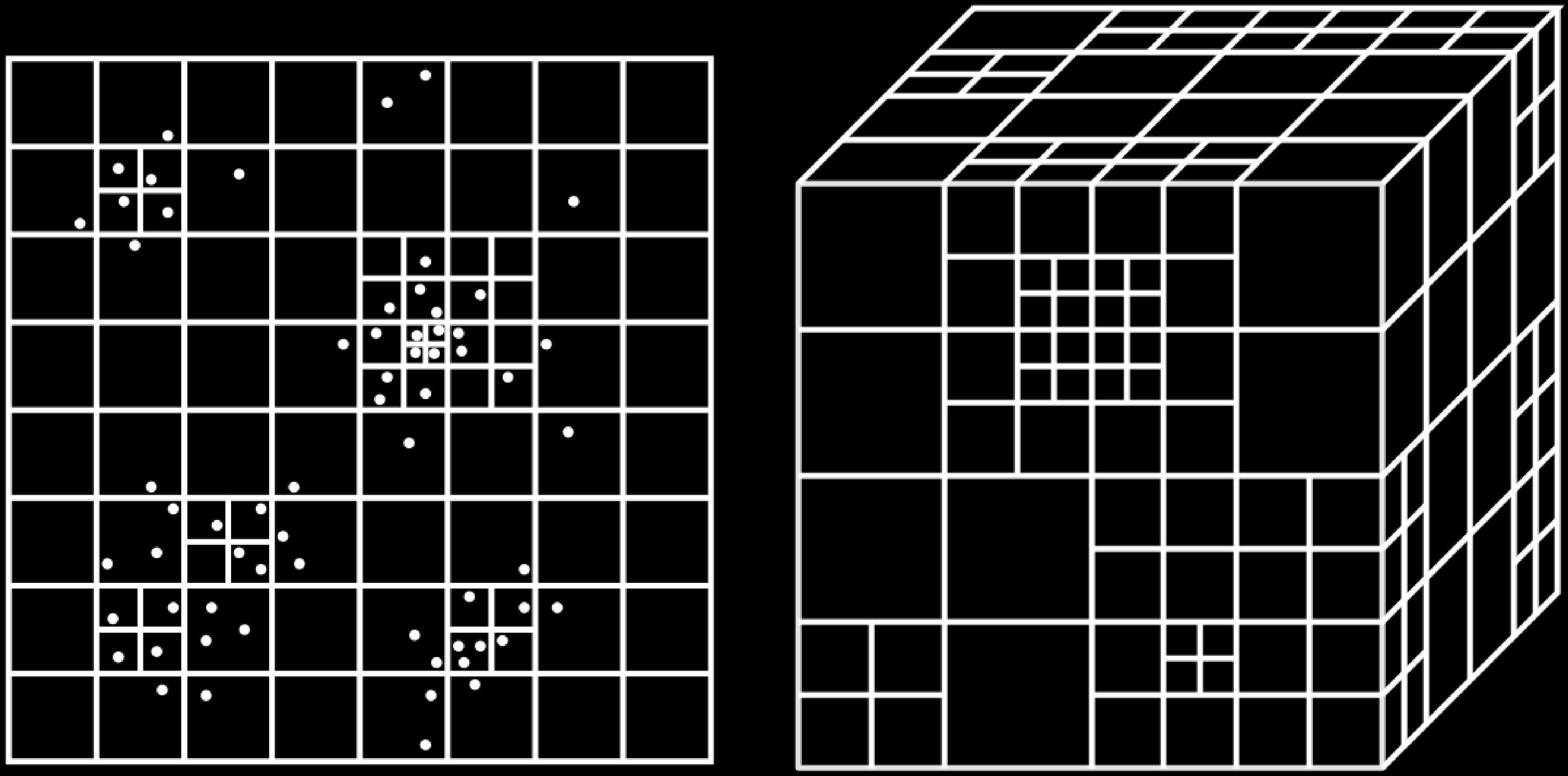
Cosmological Microwave Background

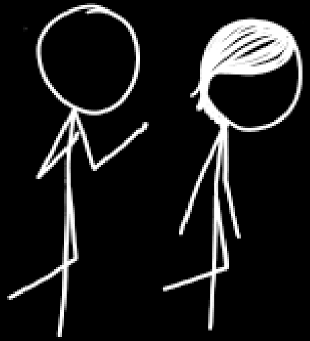




Third, they filled the box with billions of particles with the same statistical distribution as the matter in the primordial Universe.

From galaxies to expanding the Universe

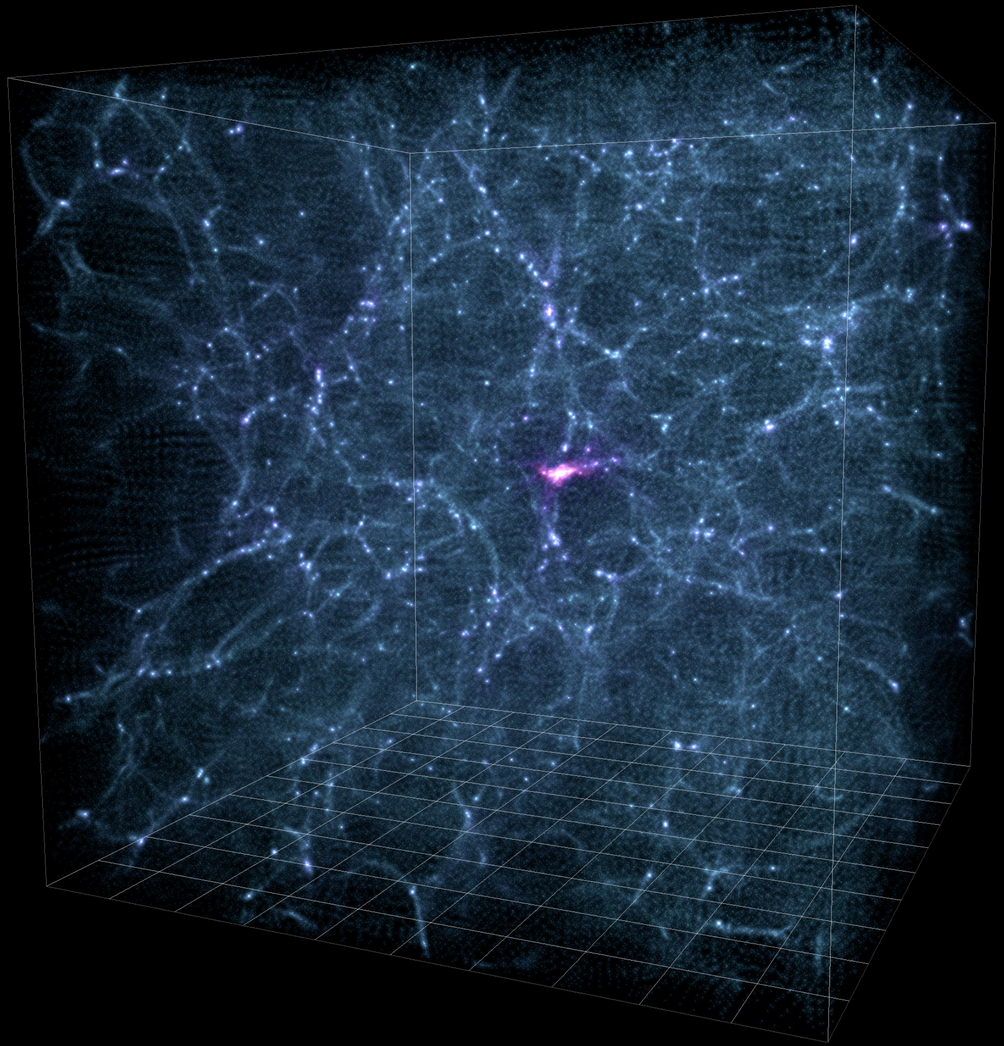




Fourth, they updated their algorithm using an Adaptive Mesh Refinement (AMR) strategy to increase resolution in regions of interest.

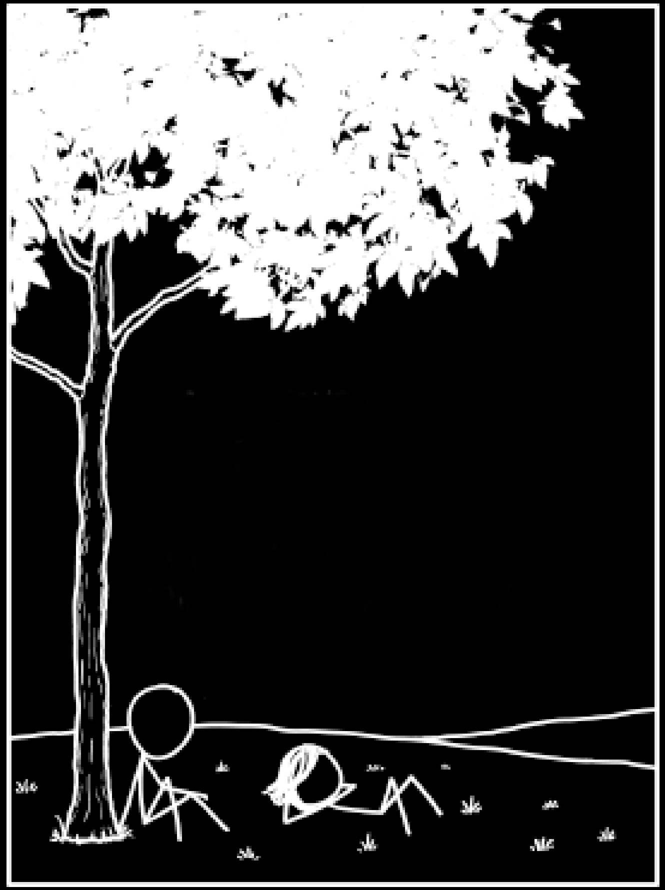
From galaxies to expanding the Universe

And after all this work this is what they obtained:



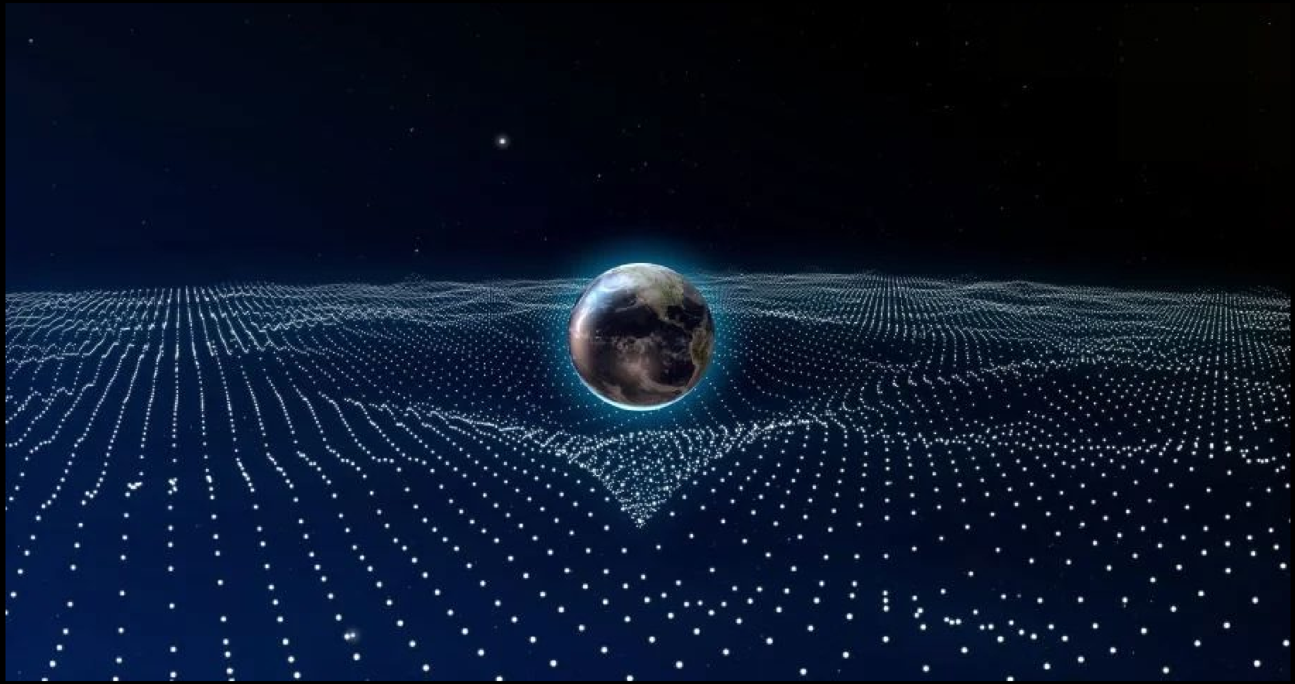
From galaxies to expanding the Universe

...and they lived happily ever after...



...except for one tiny annoying detail...

A tiny annoying detail about General Relativity



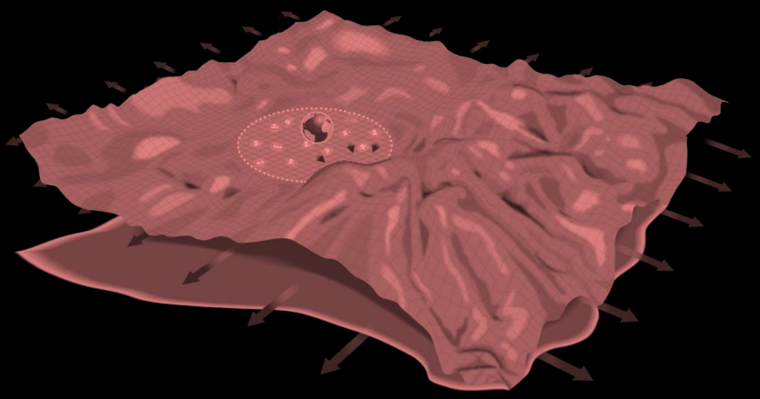
Wait, what about General Relativity?

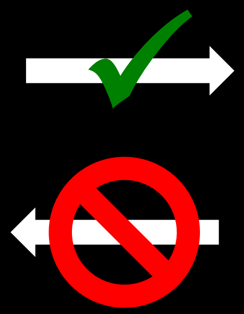
$$G_{\mu\nu} = \frac{8\pi G}{c^4} T_{\mu\nu}$$

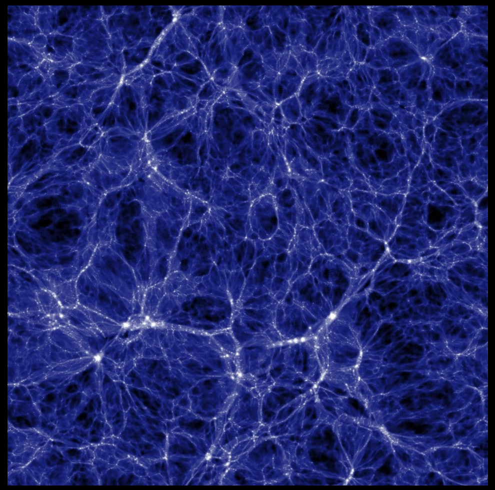
space-time geometry

energy-matter contents

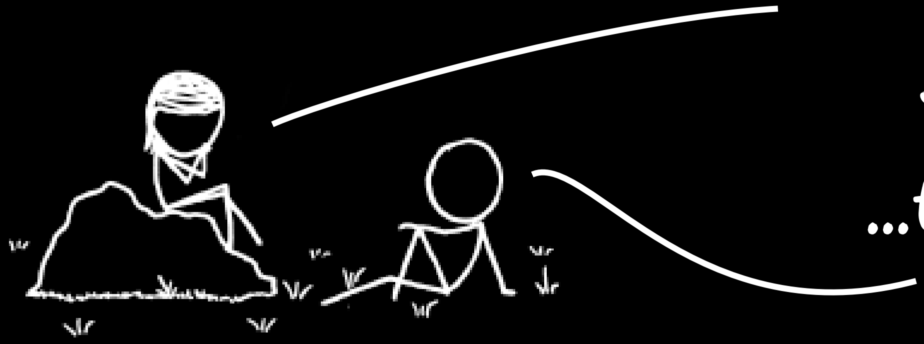
A tiny annoying detail about General Relativity







space-time geometry → $$G_{\mu\nu} = \frac{8\pi G}{c^4} T_{\mu\nu}$$ ← energy-matter contents

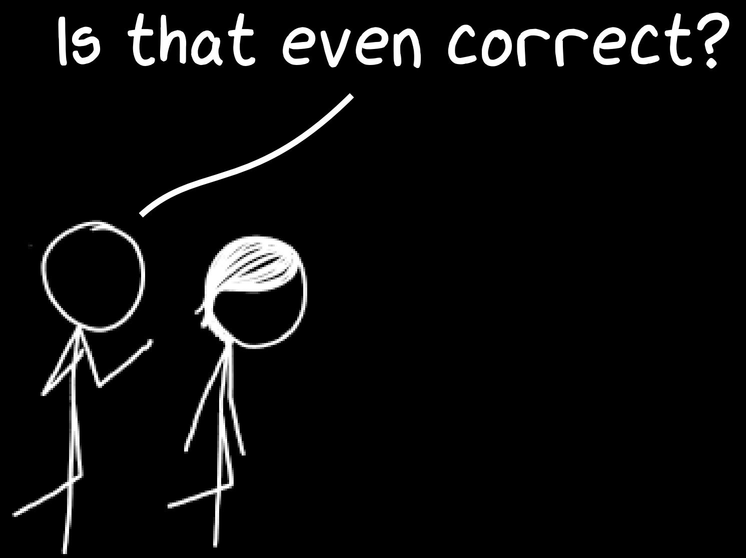


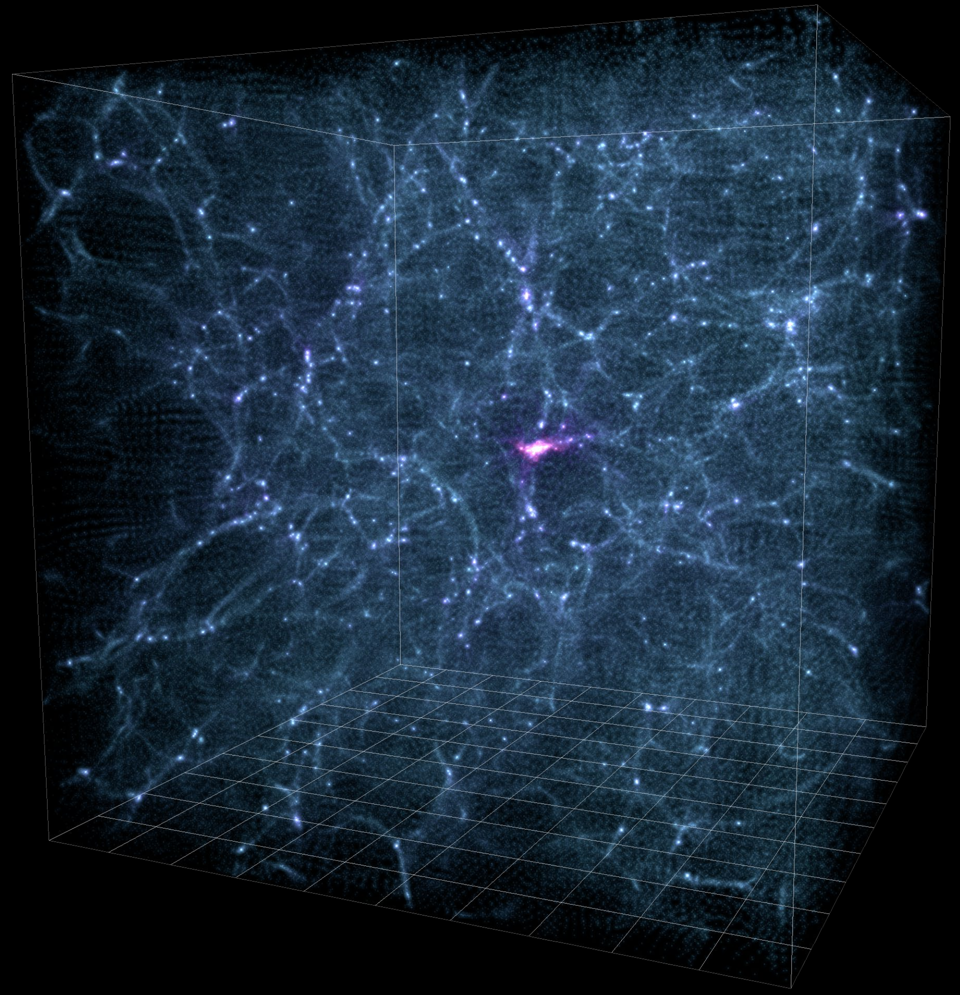
In cosmological simulations the space-time geometry evolution is precomputed…

…that means no dynamic backreaction of the contents on the geometry

A tiny annoying detail about General Relativity

It's classical physics in a pre-computed expanding background



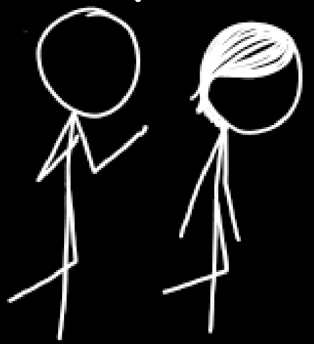


A tiny annoying detail about General Relativity

Why no fully relativistic simulations ?

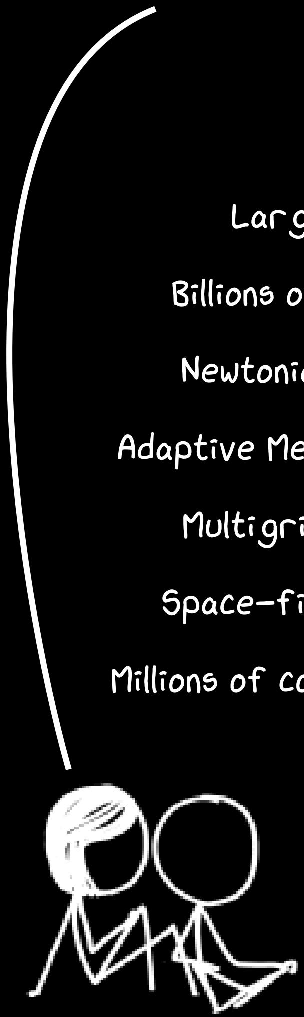
Because...

1. There is not enough computing power
2. Even if there is, it's not possible algorithmically
3. Ok, maybe... but in any case it's not interesting



The untold truth

Because no-one really knows how to write such a code...



Numerical cosmology

Large scale

Billions of particles

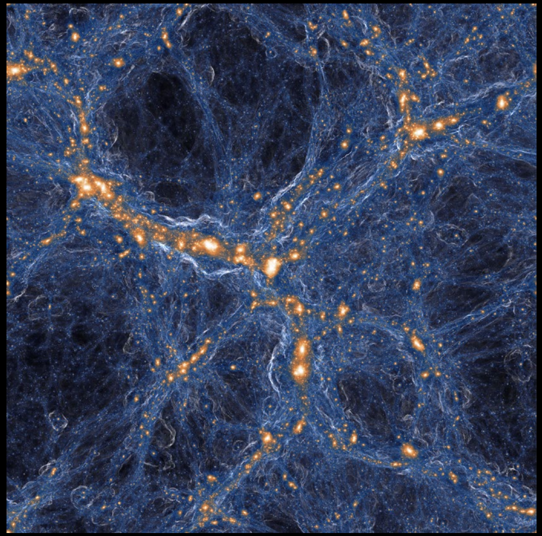
Newtonian gravity

Adaptive Mesh Refinement

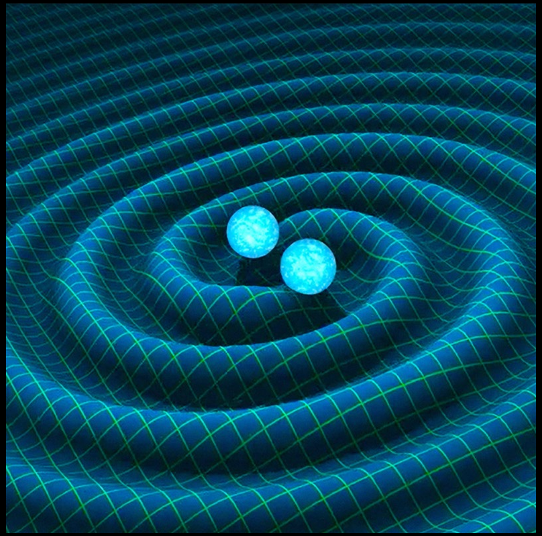
Multigrid methods

Space-filling curves

Millions of computing hours



Numerical relativity



"Small" scale

Few bodies

General relativity

Fixed-grids

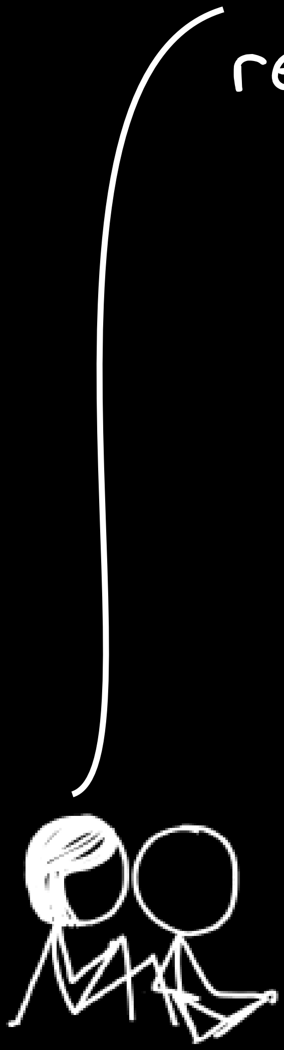
Spectral methods

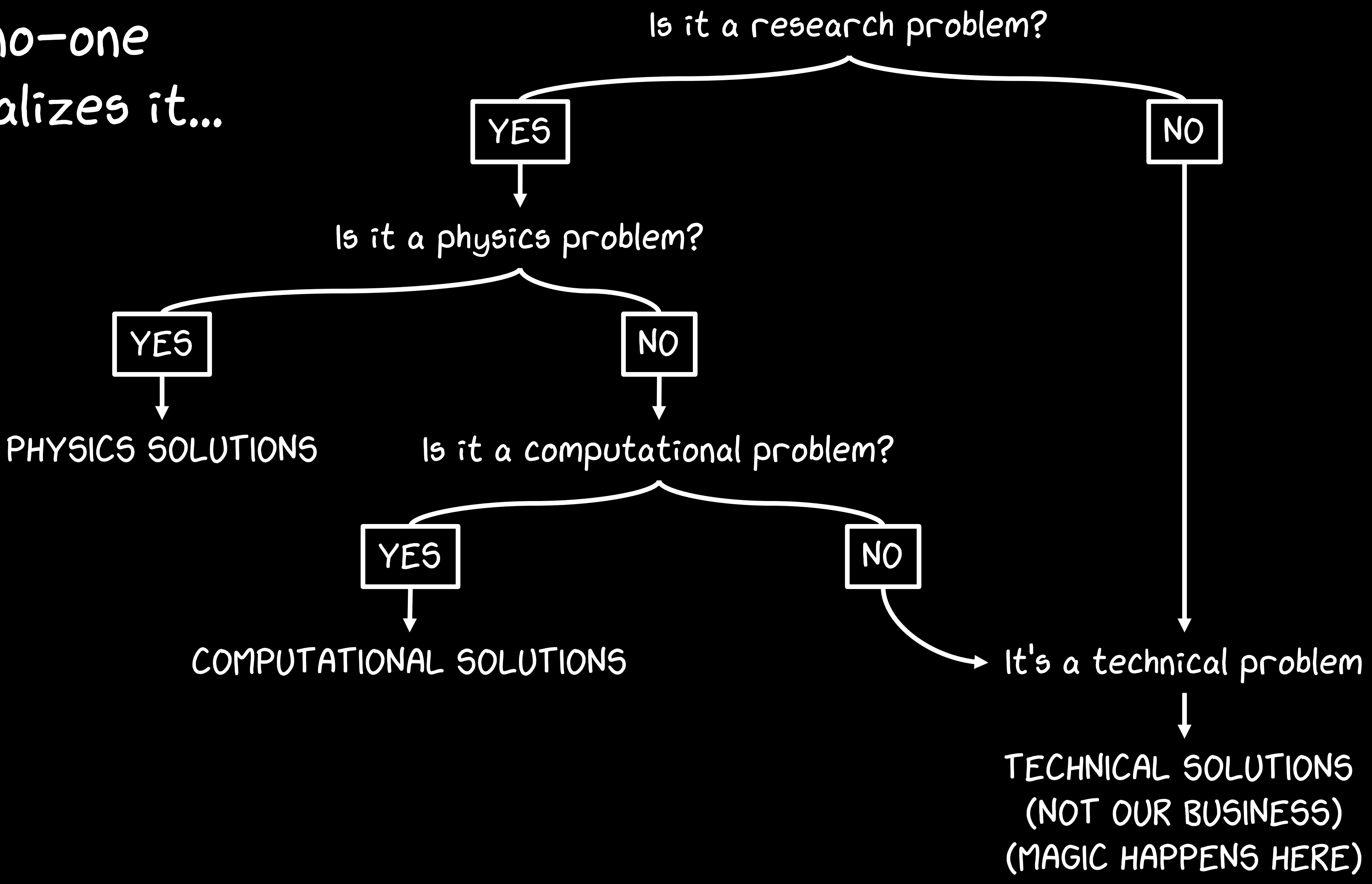
Non-trivial initial conditions

Two domains with uncomposable complex codes!

The untold truth

...and no-one really realizes it...



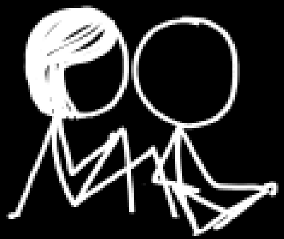


The untold truth

Programs = Code = Technical artifacts

For the most part,
in computational sciences,
the structural complexity of programs
is an unthought

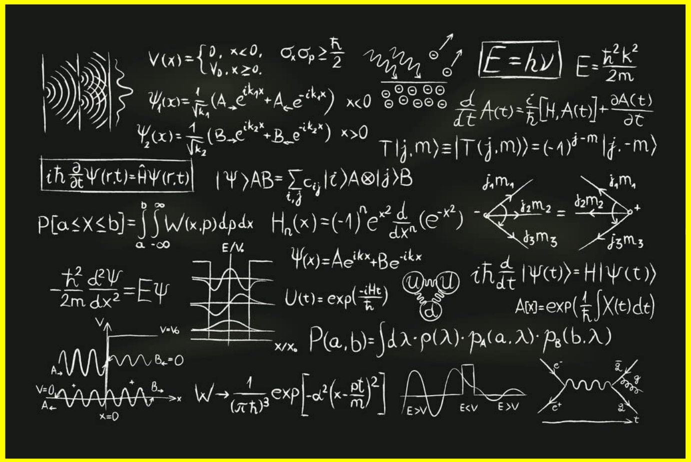
There is no solution to be found
to a problem that does not exist



Framing the problem of software complexity

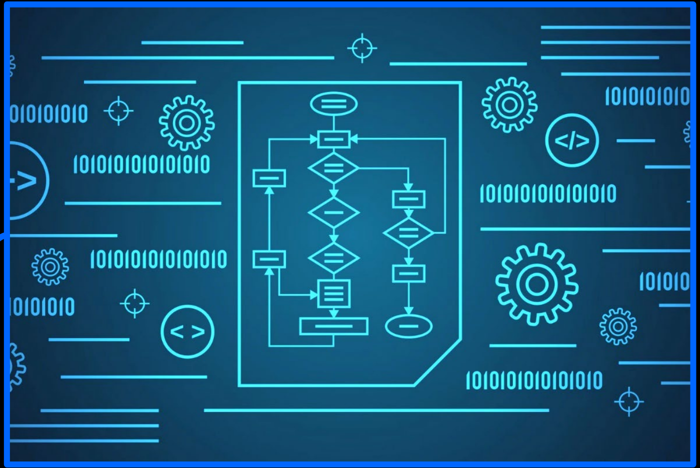
Most physics codes are built from the same categories of components

Physics

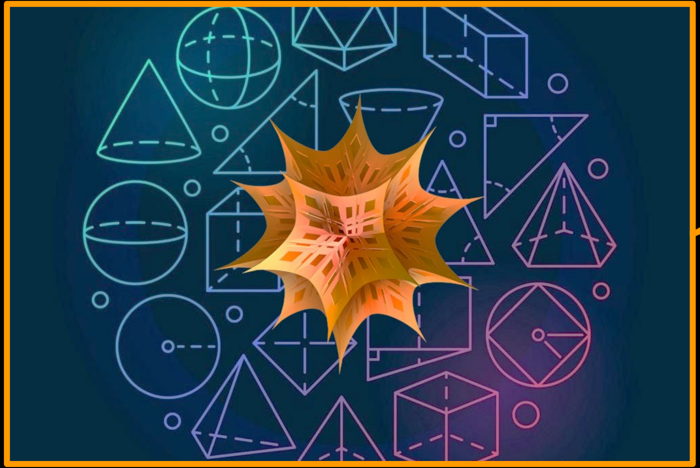


Hardware architectures

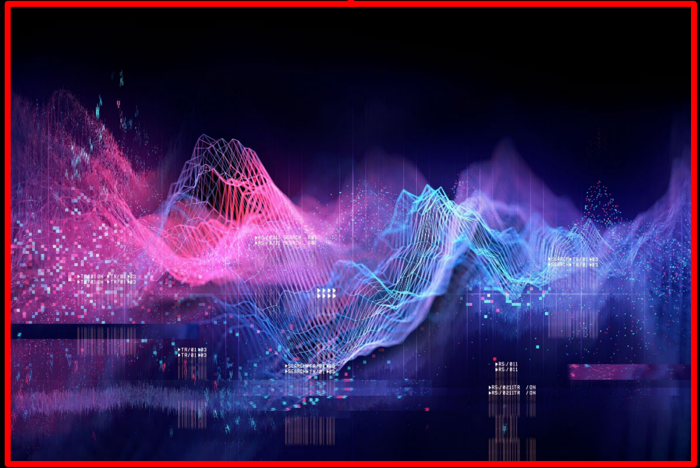
Algorithms & Numerical methods



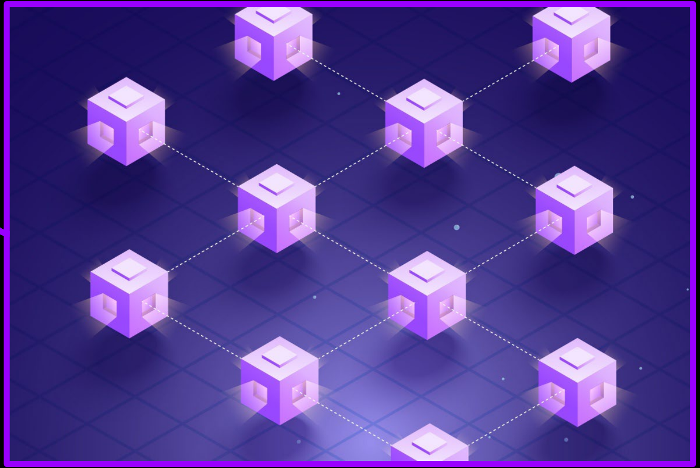
Numerical Physics Code



Topology & Geometry

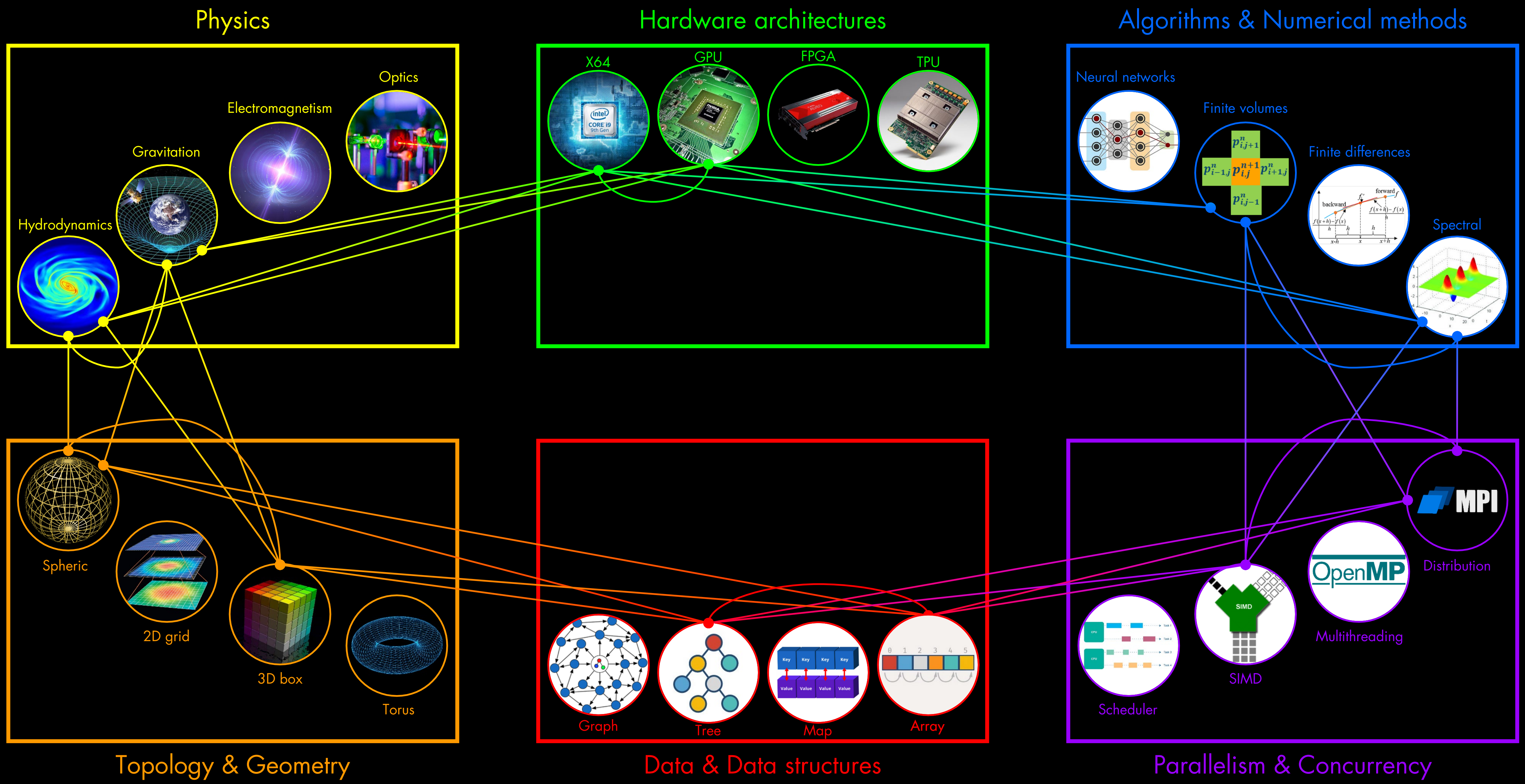


Data & Data structures

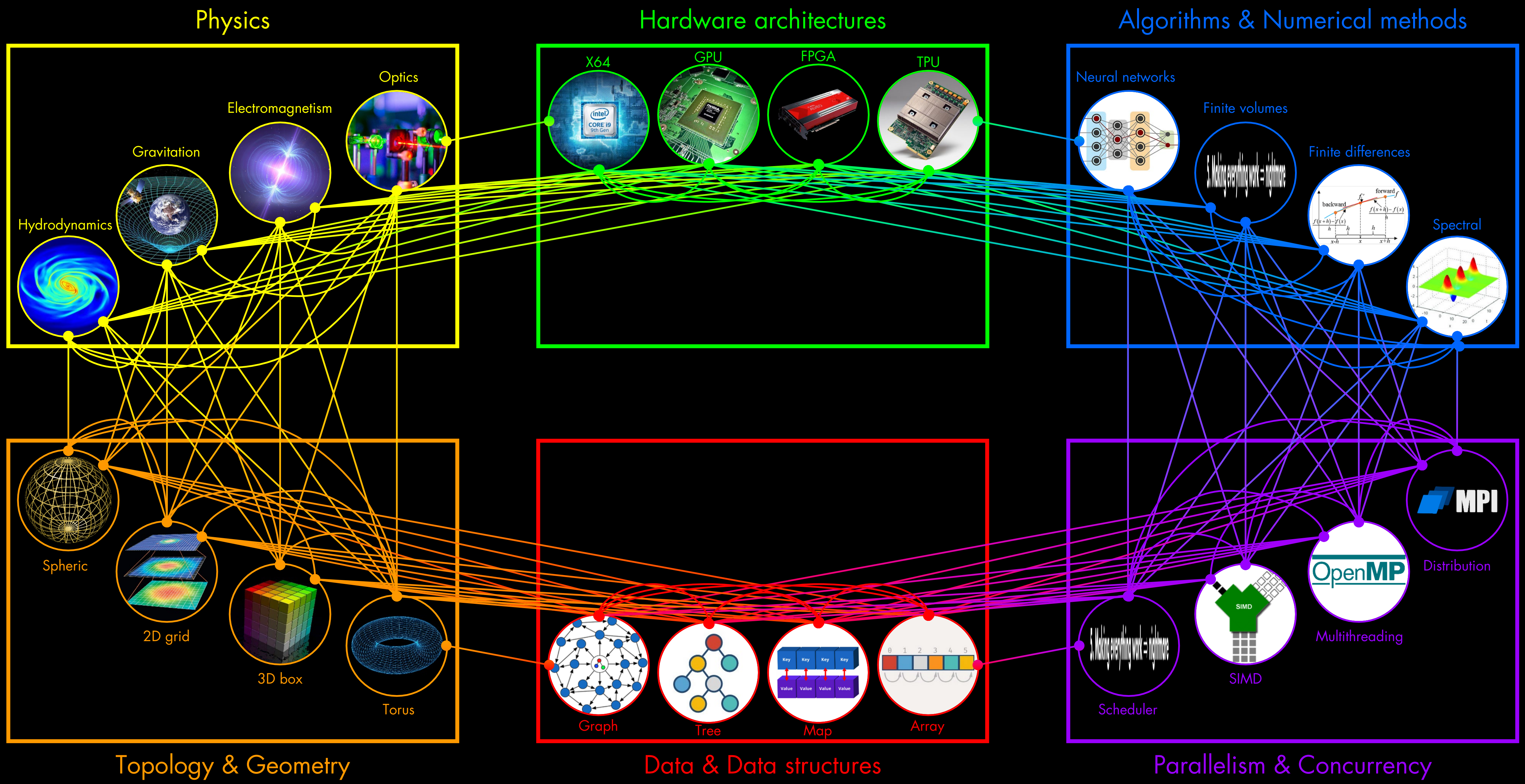


Parallelism & Concurrency

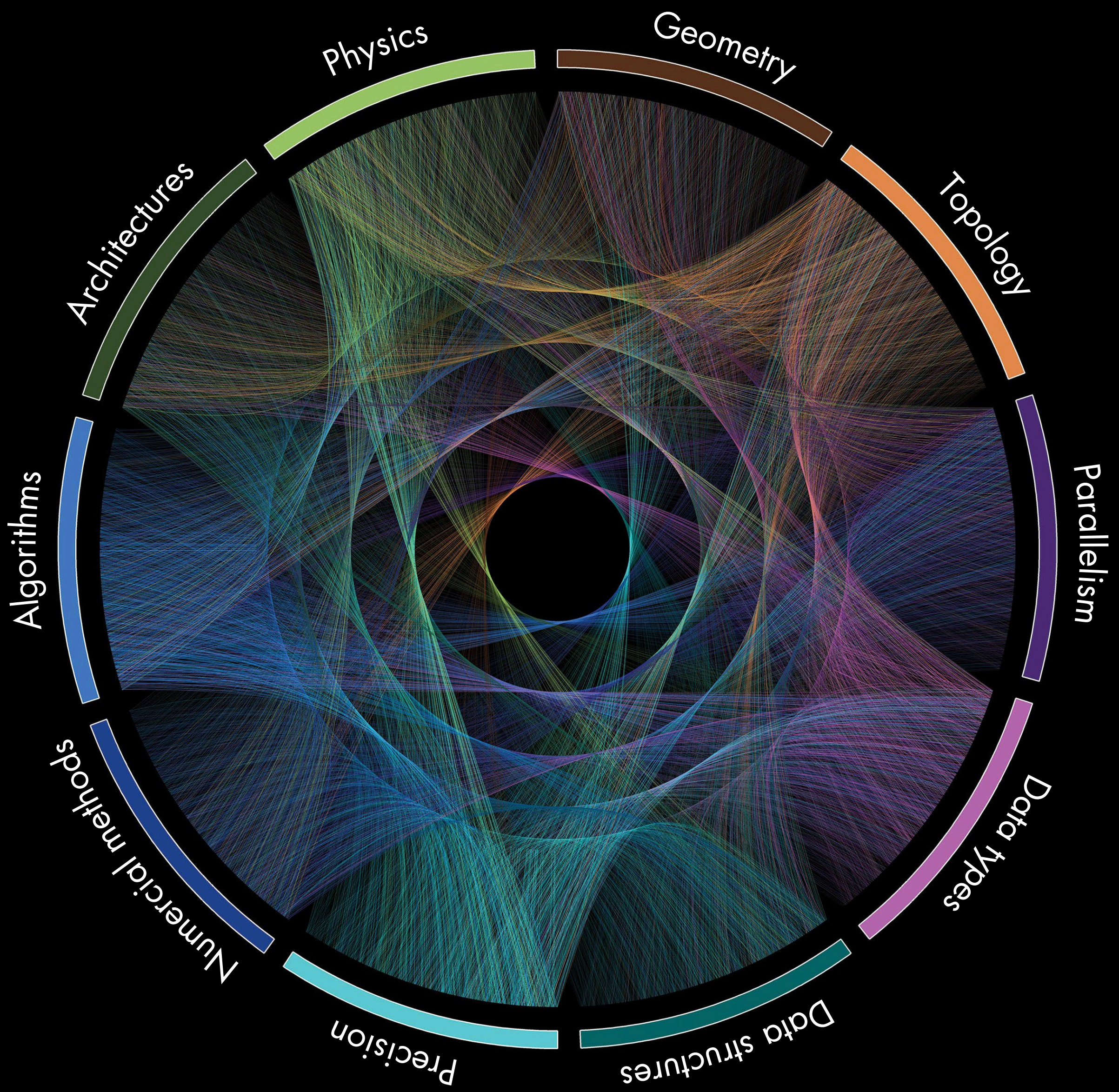
Combining individual components



Combining individual components

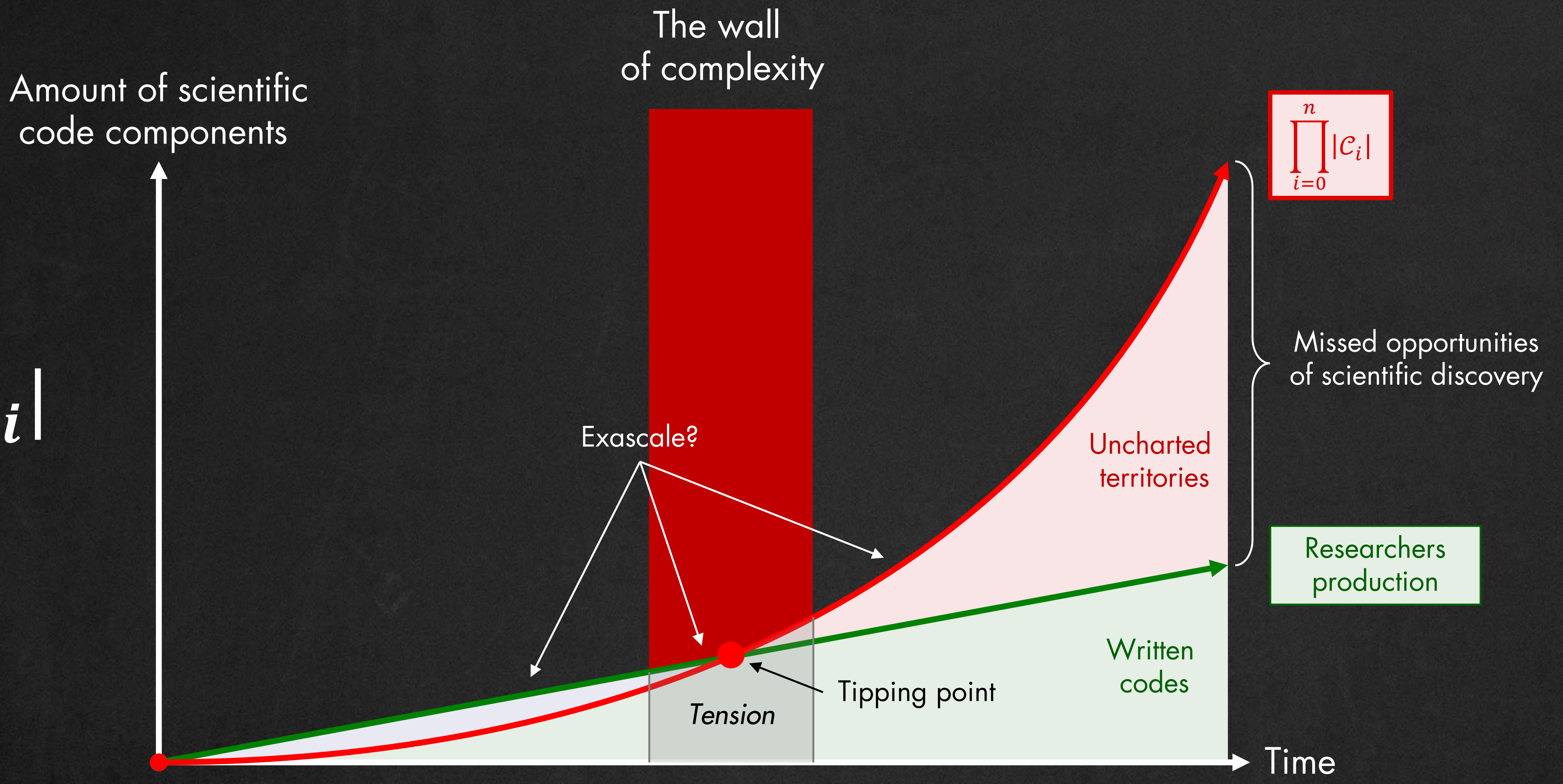


A combinatorial explosion of complexity



Hitting the wall of complexity

$$\prod_{i=0}^{n} |\mathcal{C}_i|$$



Monodisciplinar approaches are not enough

Theoretical physics

Cosmology

High enegy physics

Galaxy dynamics

Astrophysics

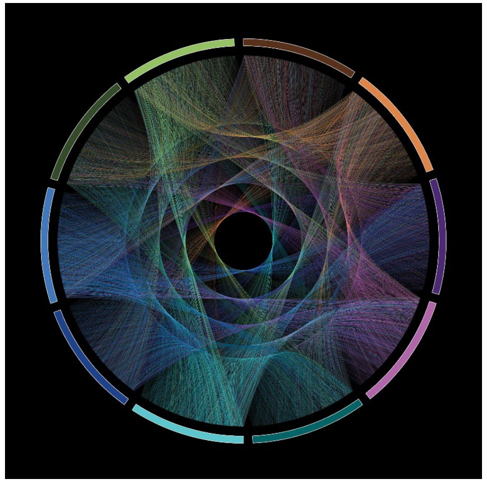
Stellar physics

Planetology

Optics

Astronomy

Instrument-ation



Proof theory

Logic

Lambda calculus

Formal methods

Programming languages

Computer science

Compilers

Graph theory

Algorithmic complexity

Type theory

The astrophysics approach

- Computer science = given
- Simplifying physics/models
- More physics for same level of complexity

The computer science approach

- Astrophysics = given
- Better algorithms/data structures/performances
- More computing for same level of complexity

Attacking combinatorial explosion as a proper scientific problem

Astrophysics ⟷ Computer science

Technical problem → Technical solutions

Software engineering problem → Software engineering solutions

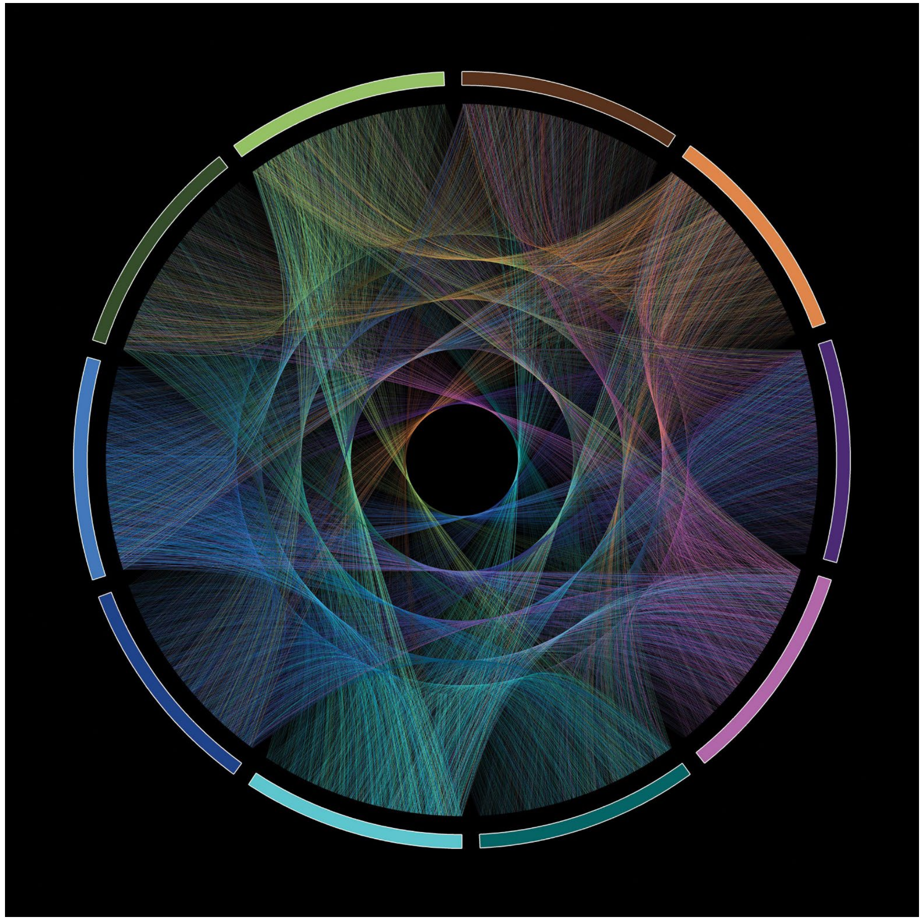
Treating the root cause instead of the symptom

- Technical symptom ⟹ code
- Root cause ⟹ compositionality / combinatorial explosion

Proper scientific problem → Proper scientific solutions

Interdisciplinary by nature

- Emergent complexity
- Composition of physics and computer science components



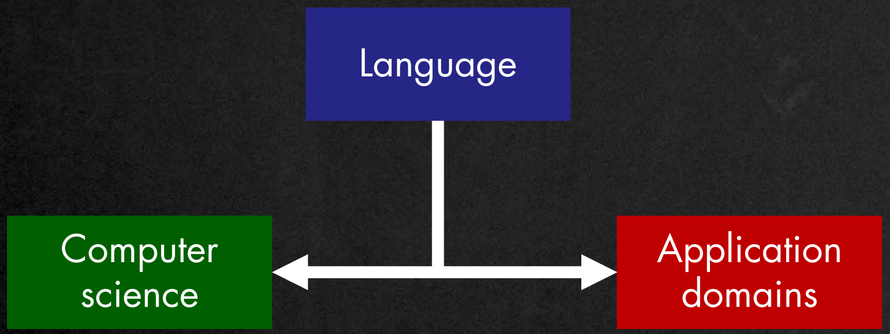
Framing the problem

$$\prod_{i=0}^{n} |c_i| \overset{?}{\Longrightarrow} \sum_{i=0}^{n} |c_i|$$

(in first approximation)

Structural complexity of programs

- Analogy with algorithmic complexity but on the structure of programs itself



A possible angle of attack

- Software architecture
- Programming languages
- Compilers
- …

A practical guiding framework

Practical design principles

Handling software complexity

- Generally guided by practical development principles
- Not coming from theoretical proofs

Design patterns

- Creational patterns
- Structural patterns
- Behavioral patterns
- Concurrency patterns
- Functional patterns

Coding principles

- Liskov substitution principle
- Law of Demeter
- Composition over inheritance
- Rule of three

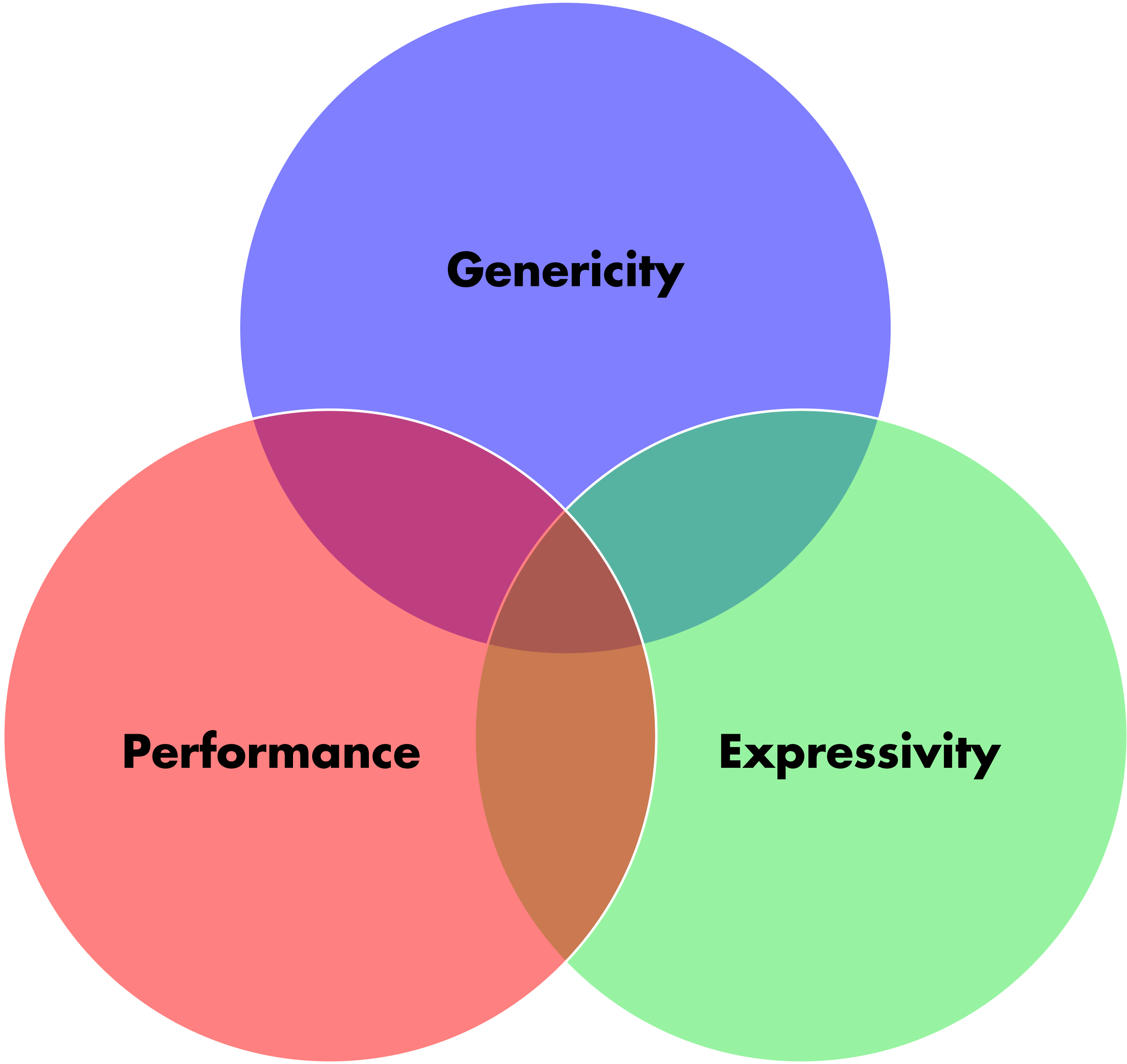
Tools

- Unit tests
- Autocompletion
- Static analysis

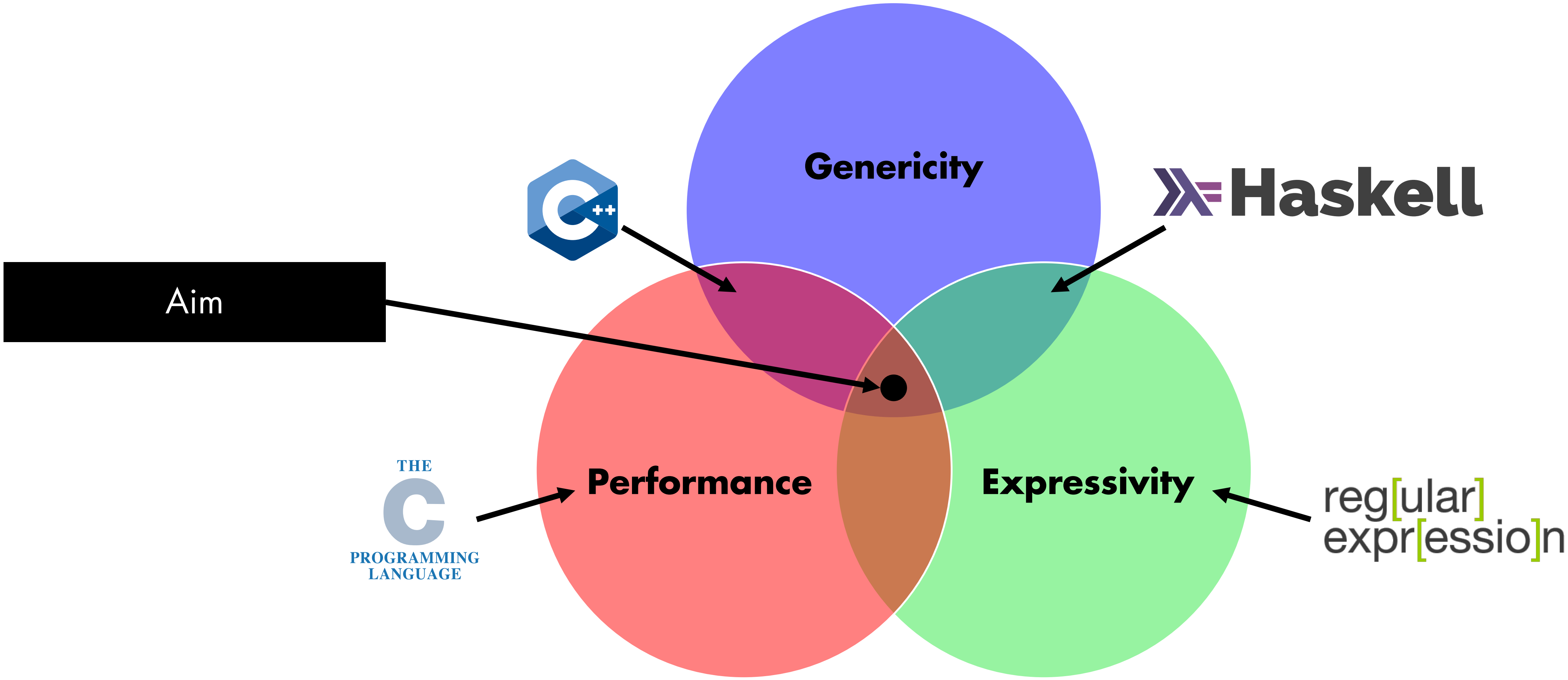
Development strategies

- Lean development
- DevOps
- Agile
- SCRUM

A guiding metric of good vs bad for abstraction: the GPE triad

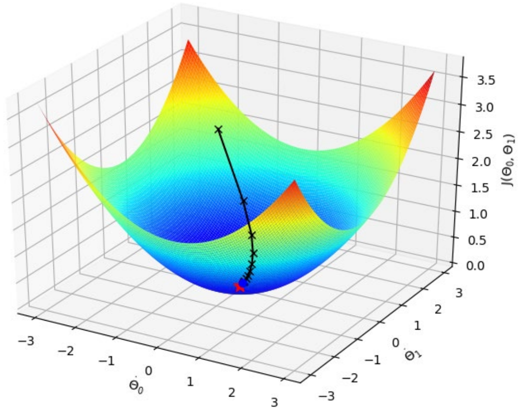


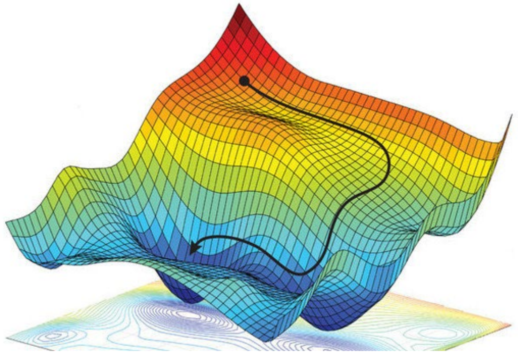
Approaching abstraction through genericity, performance, and expressivity



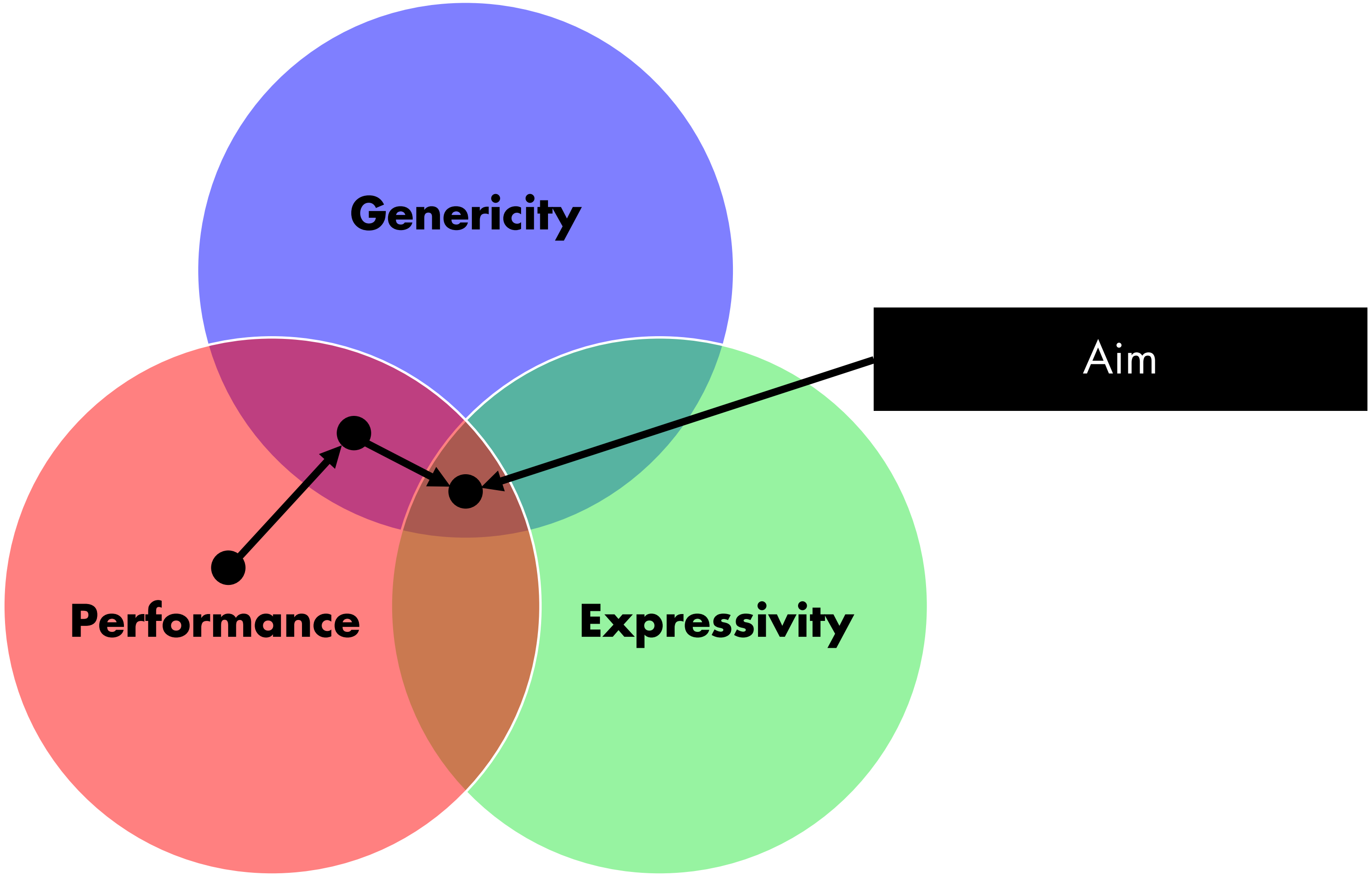
A converging iterative process

Expectation





Reality



Multidimensional arrays as a canonical example: why still no nd-arrays in C++23?

In Python

- Numpy arrays

In C++: performance concerns

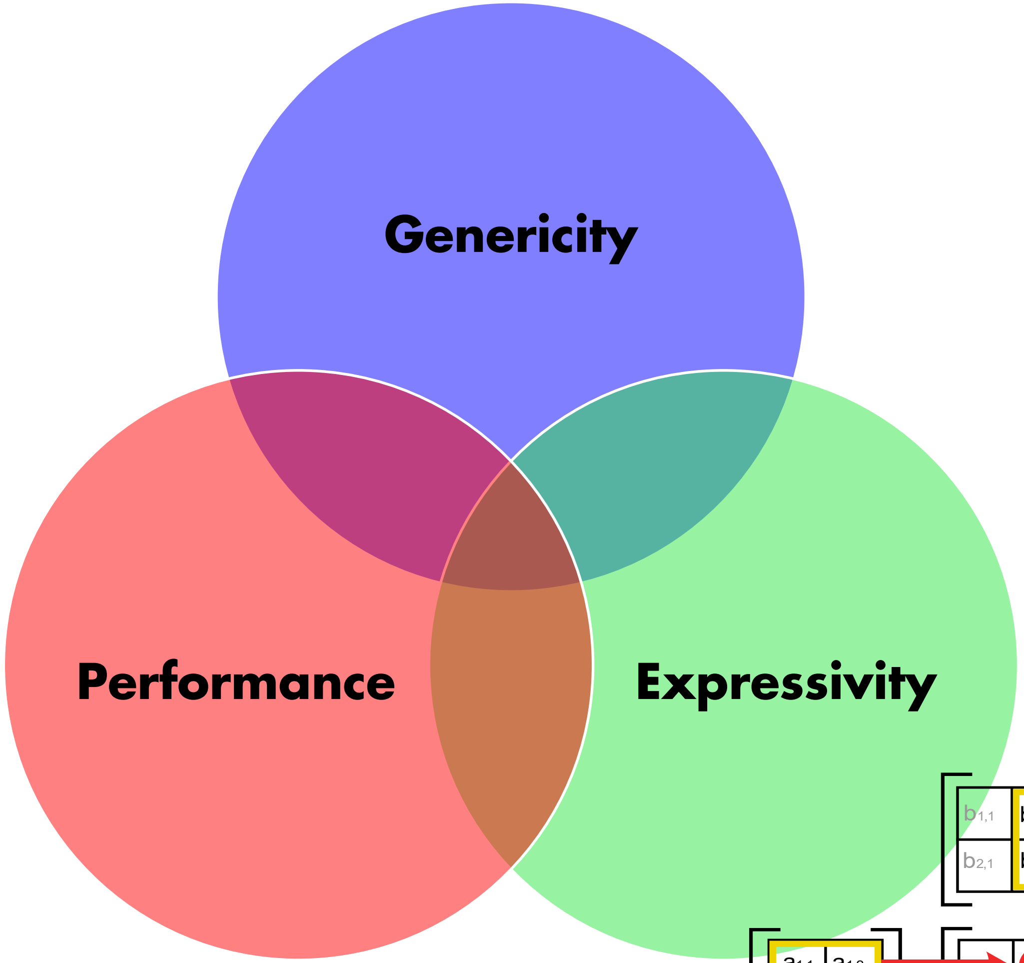
- At least BLAS performance
- Exploiting SIMD
- Memory footprint
- Parallelizability

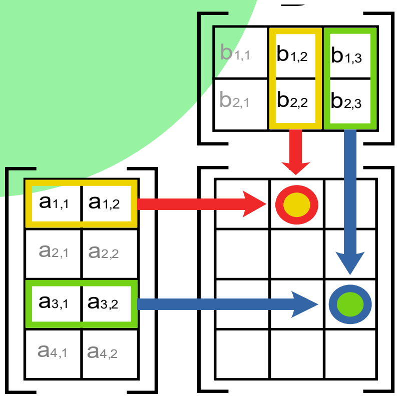
In C++: genericity concerns

- Iterator and data types
- Access patterns
- Symmetries
- Allocator

In C++: expressivity concerns

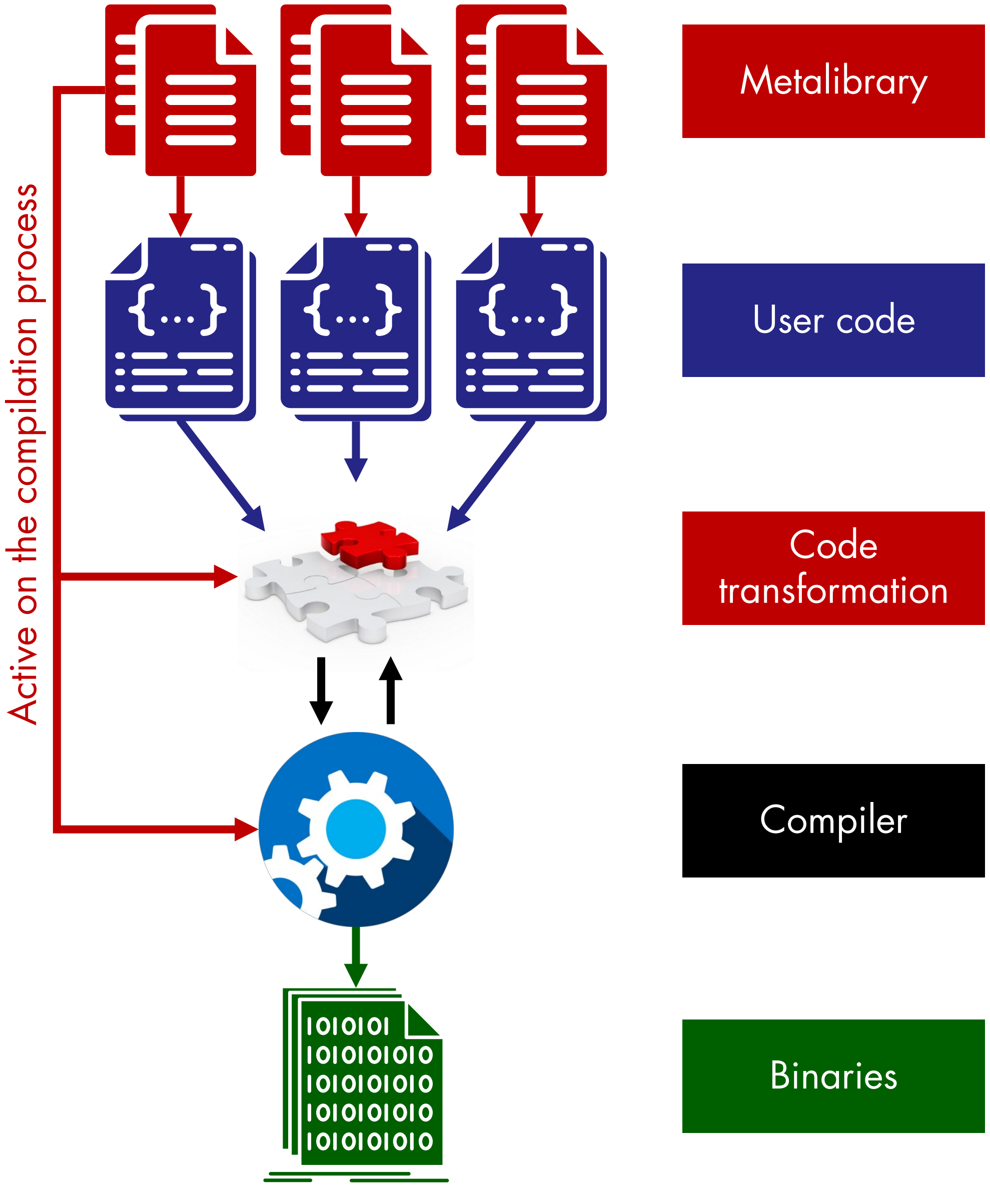
- Terse syntax for most cases
- Full syntax for expert users
- Options passing





Exploring performance concerns

Ensuring best possible performances



Several possible approaches

- New languages
- New compilers
- Preprocessors and code generators
- Metaprogramming and active libraries

Active libraries

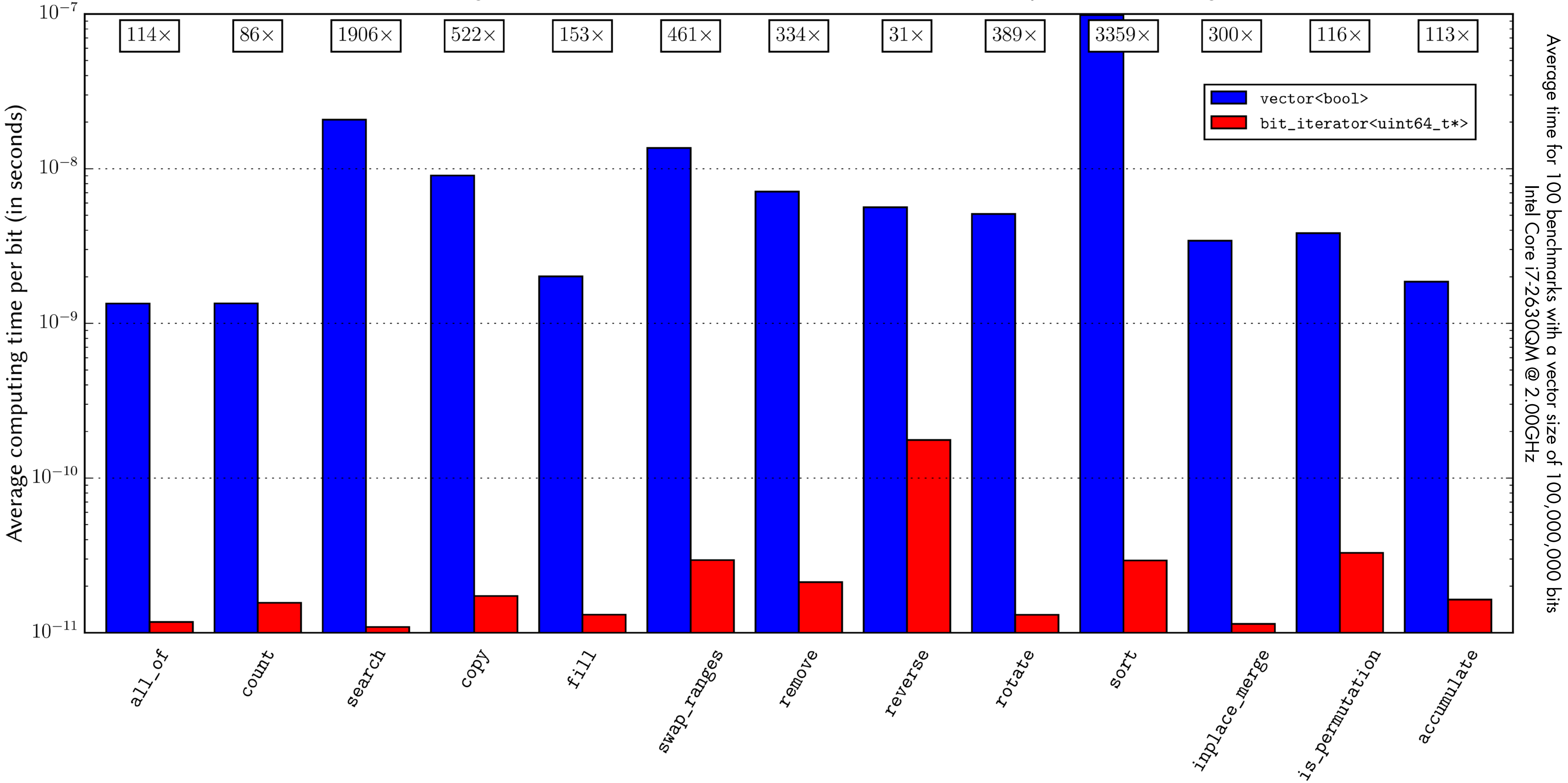
- Feedback of the user code on the library
- C++ template metaprogramming
- Generative programming
- Compile-time execution

Embedded Domain Specific Languages

- DSL within a host language (C++)

Improved architecture, improved performances

Benchmark of standard algorithms on `vector<bool>` vs their `bit_iterator` specialization (logarithmic scale)



Standardizing bit manipulation utilities

ISO International Organization for Standardization

ISO/IEC

JTC1 – Information Technology

SC22 – Programming Languages

National Representatives***

WG21 – The C++ Standards Committee

FDIS Approval

CD & PDTS Approval

Internal Approval

Advisory Groups

- Admin Group
- Direction Group
- ABI Review Group
- Security Review Group

Working Groups

- **Core** Core Language Wording
- **LWG*** Library Wording
- **EWG** Language Evolution
- **LEWG**** Library Evolution

Wording & Consistency

Design & Target

Study Groups (SG)

SG1* Concurrency	**SG2** Modules	**SG3** Filesystem	**SG4** Networking	**SG5** Transactional Memory
SG6** Numerics	**SG7*** Reflection	**SG8*** Concepts	**SG9** Ranges	**SG10** Feature Test
SG11 Databases	**SG12** Undefined Behavior	**SG13** HMI, I/O	**SG14**** Low Latency, HPC, Embedded	**SG15** Tooling
SG16 Text	**SG17** EWG Incubator	**SG18**** LEWG Incubator	**SG19***** Machine Learning	**SG20** Education
SG21 Contracts	**SG22** C/C++ Liaison			

Research
Incubation
High-level design
Domain specific review

- **Core language track**
- **Library track**
- **Domain specific investigations**

Bit manipulation standardization timeline

2016	First proposal
2017	LEWG approval
C++17	`std::byte` release
2019	International approval
C++20	First `<bit>` release
(C++23)	`<bit>` ranges
(C++26)	`<bit>` algorithms

High-performance computational sciences when software complexity is the bottleneck

Software

- Combinatorial explosion of complexity
- Low-level optimization opportunities

Hardware

- Pure performance still grows exponentially
- Explosion of optimization opportunities

New bottlenecks

- Development time
- Human resources

Not bottlenecks anymore

- Hardware capabilities
- Pure performance

Consequence

- Software always lags far behind hardware

In first-order approximation

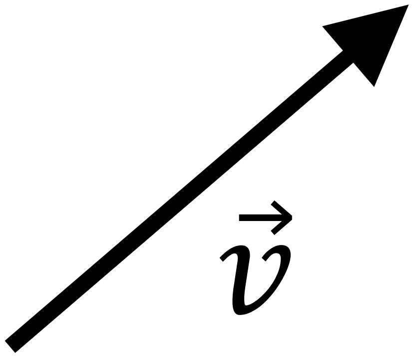
- Computational power can be considered as infinite at time of development

Exploring genericity and abstraction strategies

The root of all evil

$$\begin{pmatrix} 3.5 \\ -1.2 \\ 0.9 \end{pmatrix}$$

In numerical physics



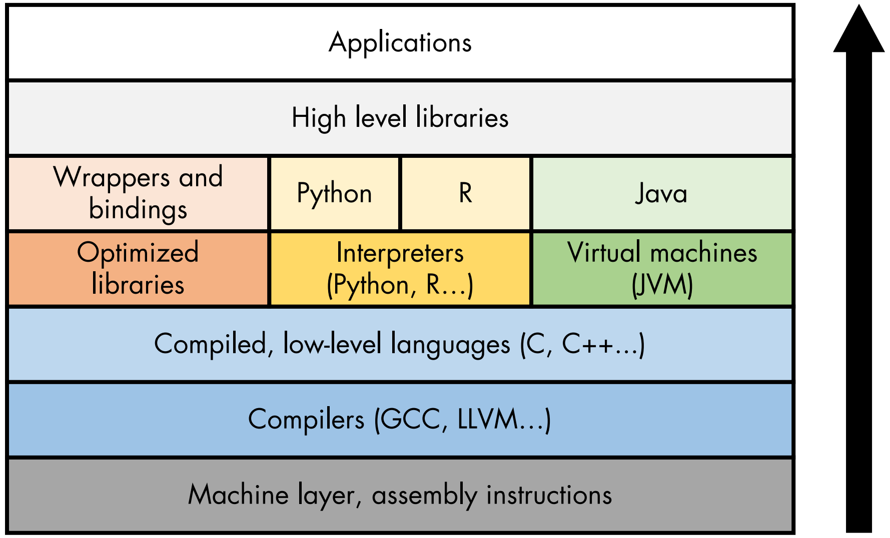
In maths

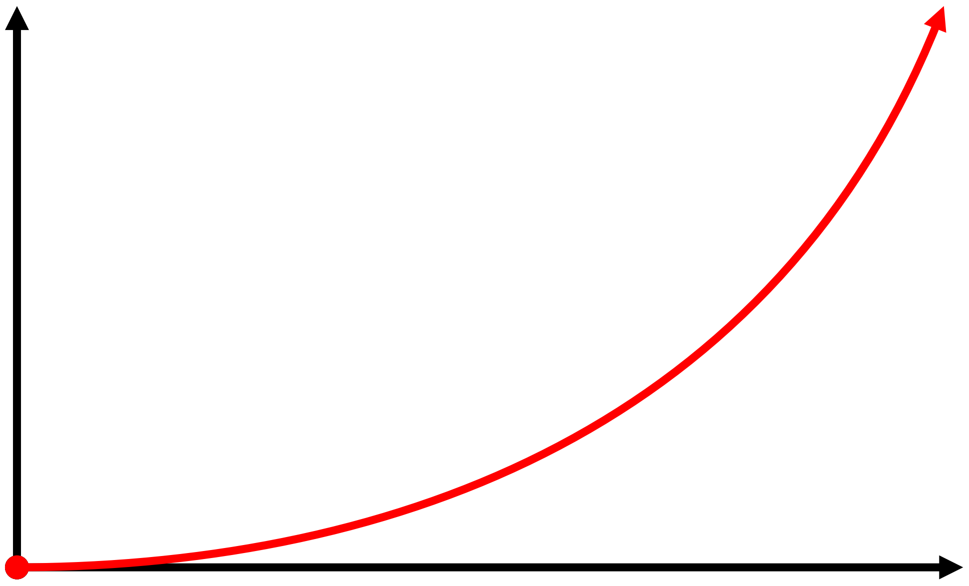
Components ≠ Vector

$\vec{v}$: independence from change of basis

Amplification of conceptual approximations

Illustrative Software Stack

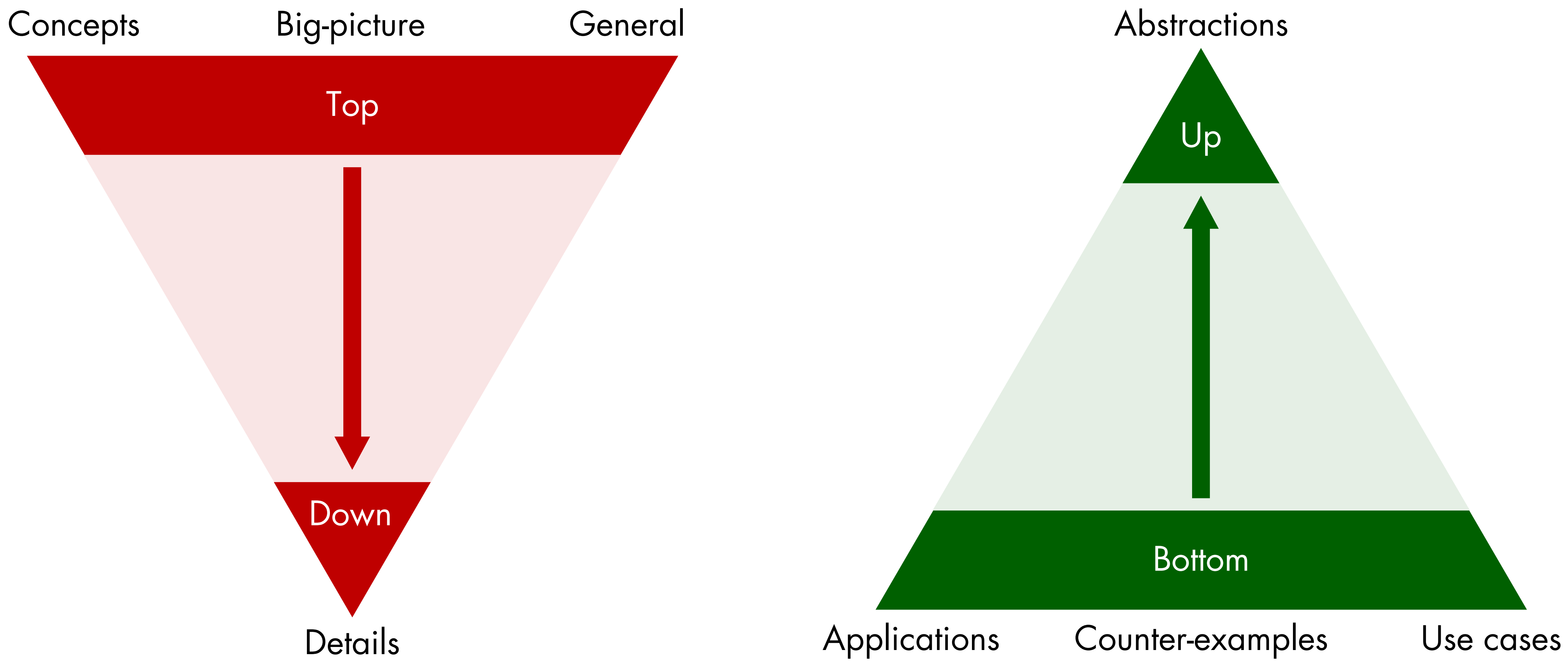




Conceptual approximations get amplified through higher layers of abstractions

- "Almost right" can quickly transform into "Totally wrong"

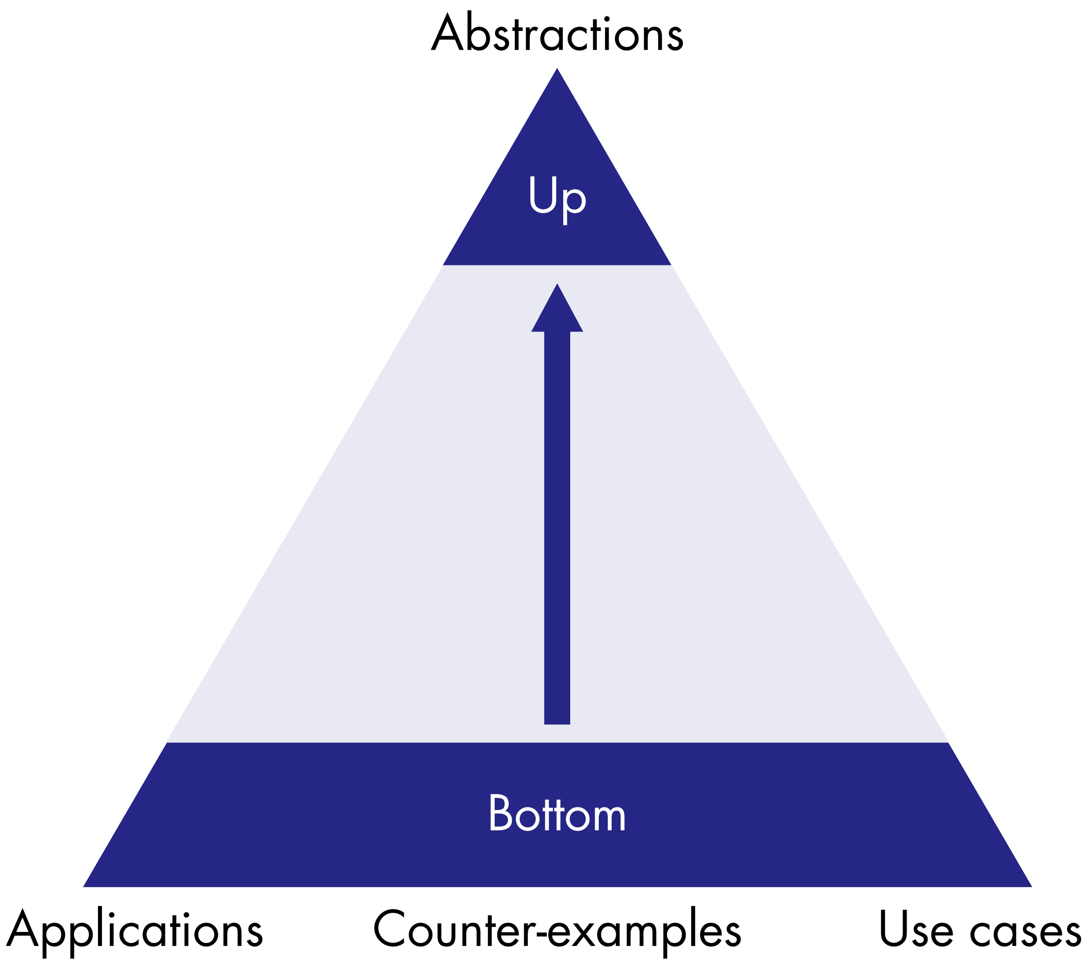
Top-down vs bottom-up approaches



Better for software implementation

Better for software architecture

Bottom-up approaches tend to work better to find the right abstractions



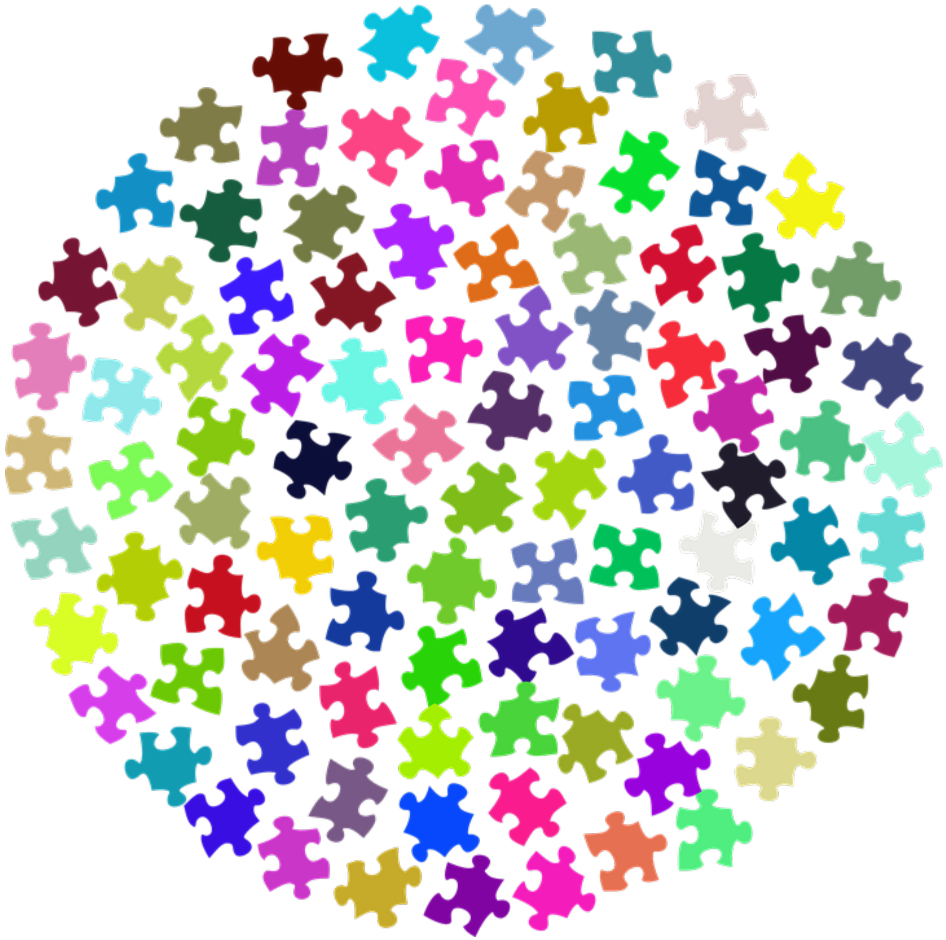
Bottom-up approach

- Accumulate concrete examples first
- Let abstractions emerge from details

Programming languages vs human languages

- Human concepts ≠ Computer concepts
- Human languages are fuzzy by nature
- Programming languages need rigorous definitions

Constraining abstractions from use cases: mapping the design space



Looking for all possible constraints

- More use cases $\Longrightarrow$ More constraints on abstractions
- Starting with everything one may want
- Looking for the weirdest applications
- Finding boundaries

Remove constraints one by one

- Some use cases add more constraints than others
- Start by removing corner cases that add strong constraints

Software architecture is not about what one **can** have it's about **deciding** what one **cannot** have

Concept-based programming

Concept-based programming

- Allow to define mathematical classes of types

Object Oriented Programming

- Monolithic type hierarchies
- Context-independent hierarchies
- Top-down approach

Concept-based Programming

- Named sets of constraints
- Context-dependent constraints
- Bottom-up approach

$x : T \qquad x \to \sqrt{x}$

T should be a number

$v : T, i : U \qquad (v, i) \to v[i]$

T should be a container
U should be an integer

Exploring expressivity and DSLs

Defining expressivity

Practical definition

- Ease-of-use
- Clear
- Concise
- Precise
- Accurate

Formal definition

See "*On the expressive power of programming languages, Science of Computer Programming, M. Felleisen, Science of Computer Programming, 1991*"

Symbolic calculus in C++

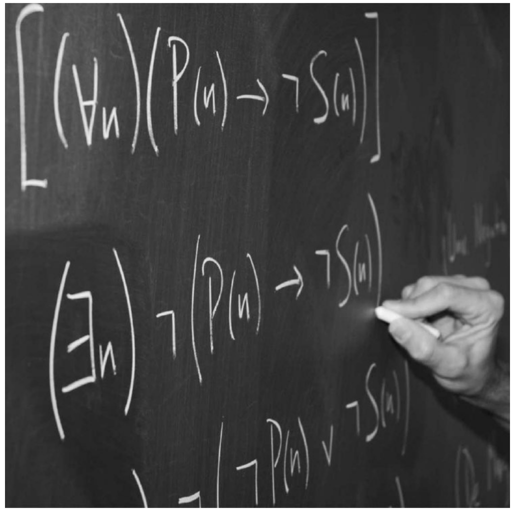
Symbolic calculus with matrices

```cpp
int main(int argc, char* argv[]) {
    // Defining 2D dynamic array type
    using matrix = decltype(ndarray<double, shape()()>);

    // Defining symbols
    symbolic a;
    symbolic X;
    symbolic Y;
    symbolic Z;

    // Loading data
    matrix x = read_data("xdata.csv");
    matrix y = read_data("ydata.csv");
    matrix z = read_data("zdata.csv");

    // Symbolic formula
    formula f = a * X * transpose(Y) * Z;

    // Computation
    return f(a = 0.5, X = x, Y = y, Z = z);
}
```

Designing Domain Specific Languages

Most important principle

- Start with what users should be able to write



Interdisciplinarity

- Start from application domain
- Reverse engineer grammar rules from application domain

AST manipulation

- DSL: Domain-Specific Languages: Create new languages with new compilers
- EDSL: Embedded Domain-Specific Languages: Use metaprogramming for AST manipulation

Passing as much information as possible to compilers

Current state of affairs

- Compilers generally have no idea what the end user has in mind
- Information is lost in between the user and the compiler
- Compilers try to guess the information that has been lost

Code transformation

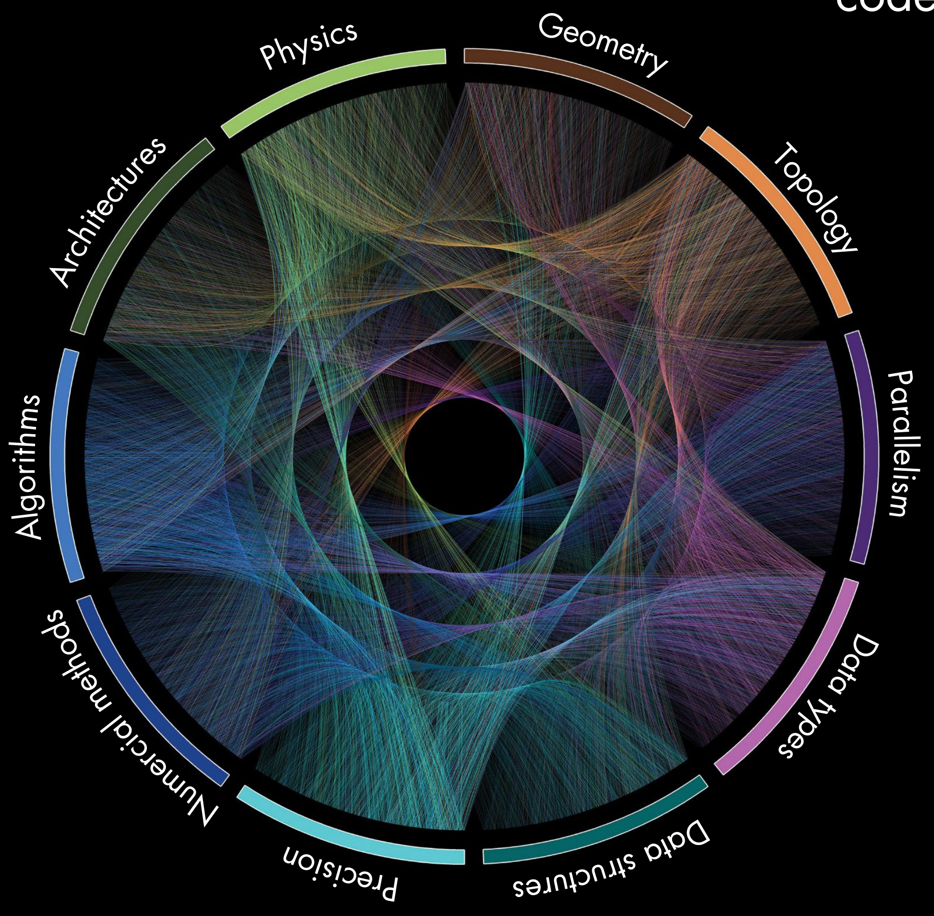
- High-level information is useful information to be exploited for code transformation

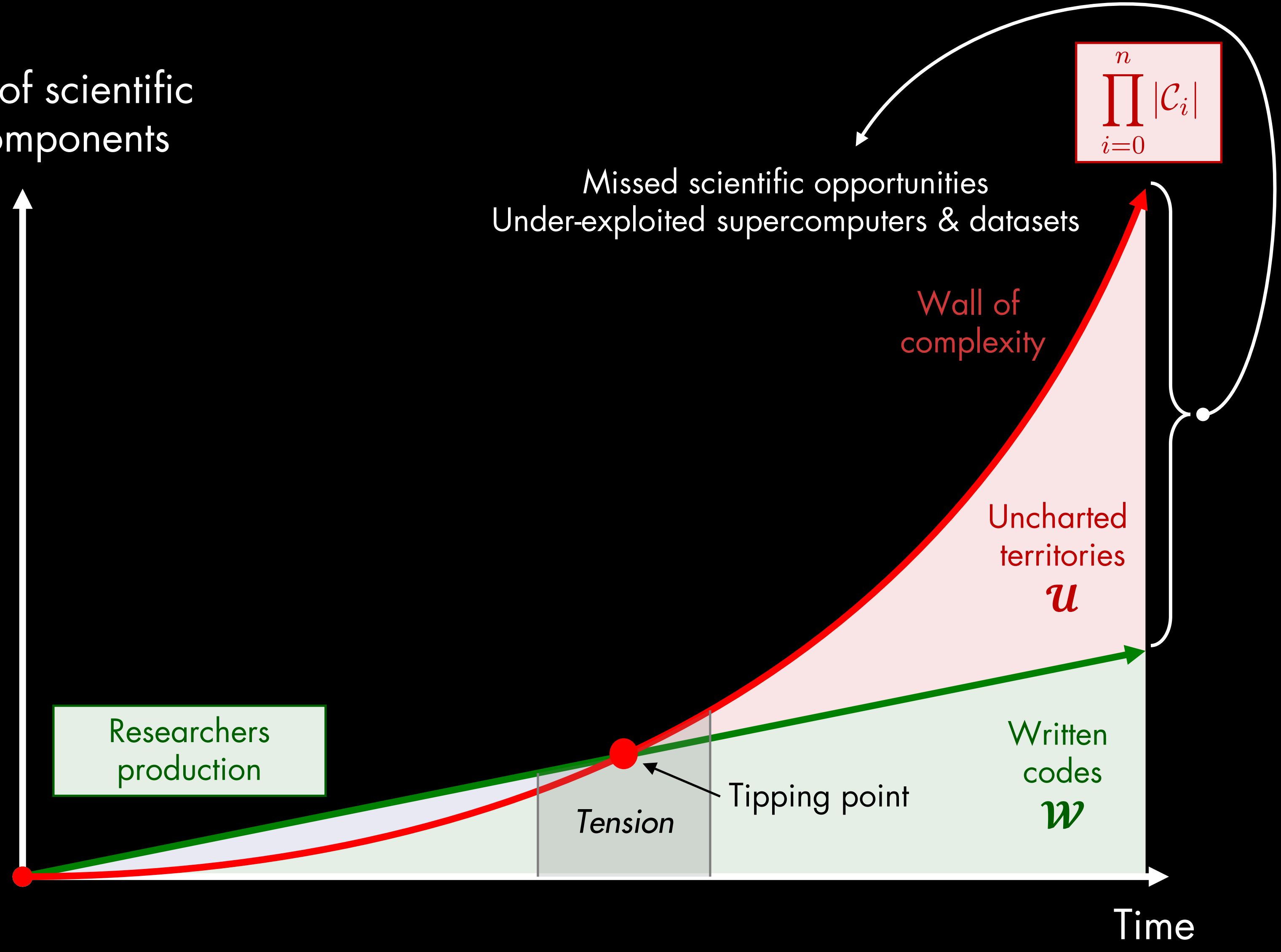
Keeping the structure

- Reflecting the structure of application domain abstractions in the structure of programs

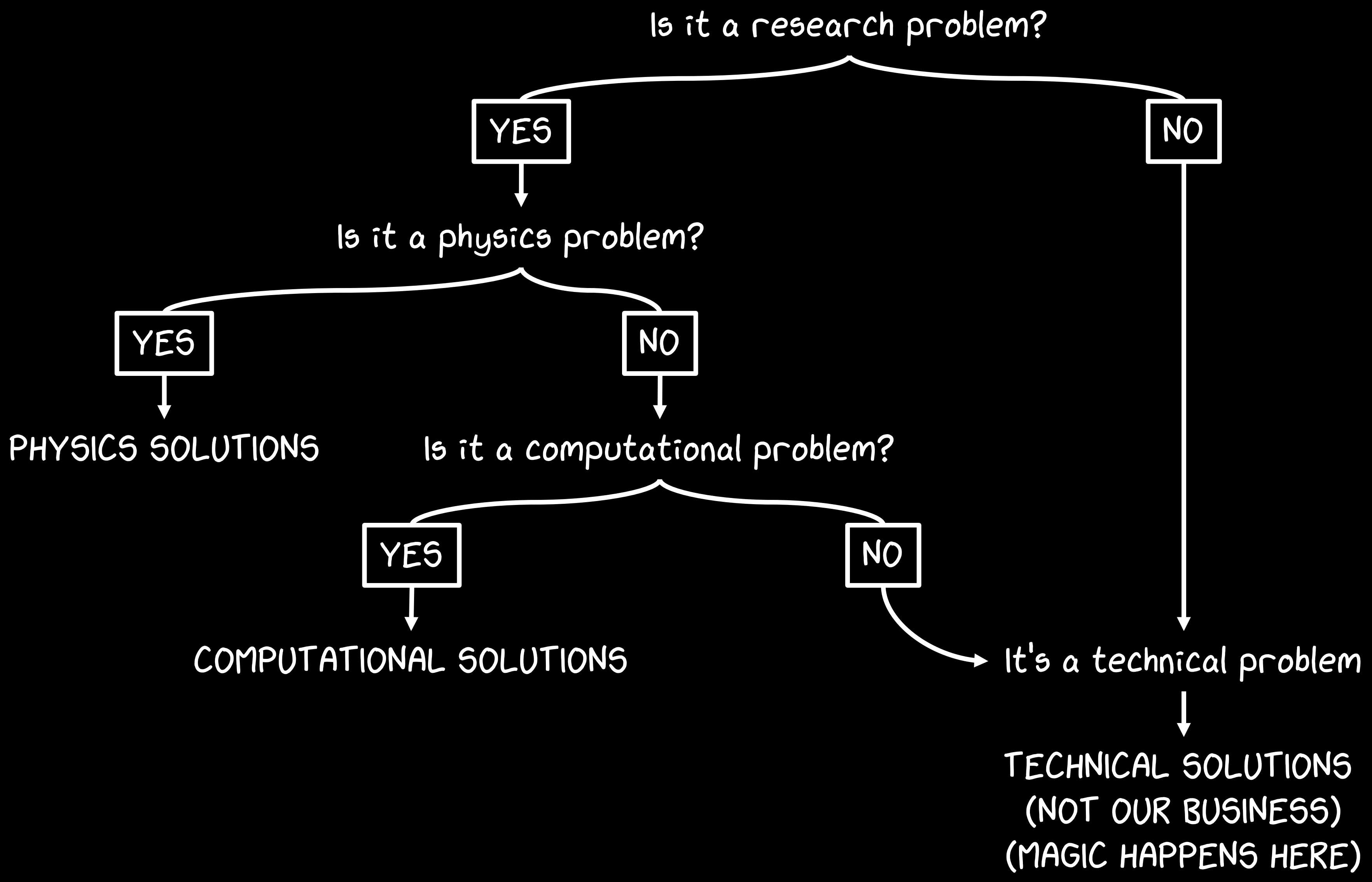
Facing the wall of software complexity

Summary: 1) There is a combinatorial explosion of complexity is scientific codes





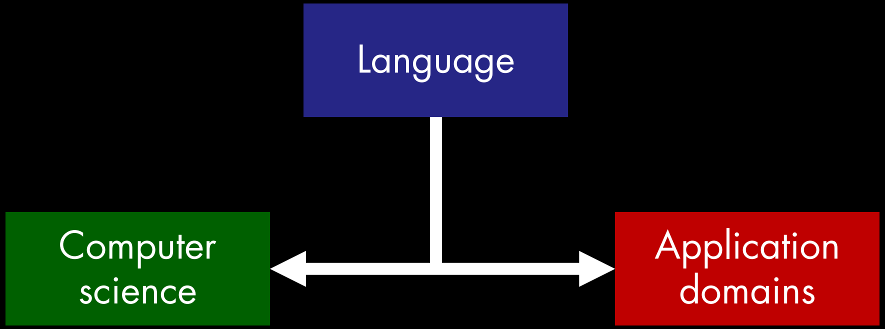
Summary: 2) The problem is often a blind spot of computational sciences



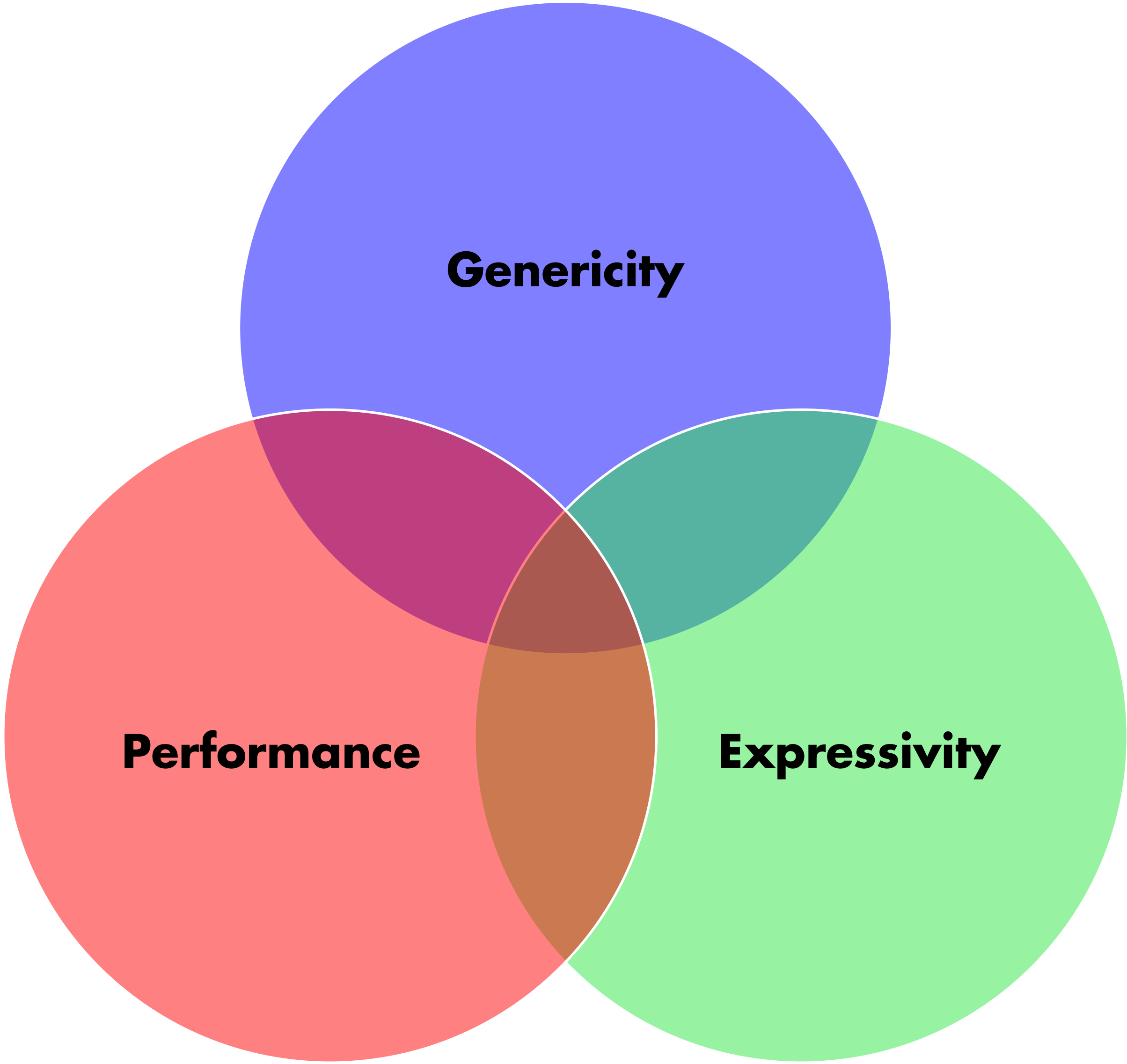
Summary: 3) Working on programming languages can allow to reduce this complexity

$$\prod_{i=0}^{n} |c_i| \overset{?}{\Longrightarrow} \sum_{i=0}^{n} |c_i|$$

(in first approximation)



Summary: 4) Evaluating solutions in terms of GPE can serve as a guide



Conclusions

Performance

- In first-order approximation, computational power can be considered as infinite at time of development

Genericity

- Conceptual approximations get amplified through higher layers of abstractions
- Concept-based design using bottom-up approach can help

Expressivity

- Starting with what users should be able to write
- Pass as much high-level information as possible to compilers
- Reflecting the structure of the application domain into the structure of programs

The wall of software complexity

- Many application domains are facing or will soon face a problem of structural code complexity
- It's anything but a technical problem and will require computer science approaches
- Research in programming languages and compilers can help

Thank you for your attention

Any question?