The Missing Step

Making Data-Oriented Design One Million Times Faster

ANDREW DRAKEFORD

Meeting C++
2025

Leave One Out Regression

- One million elements, One million leave one out regressions:

Reference implementation

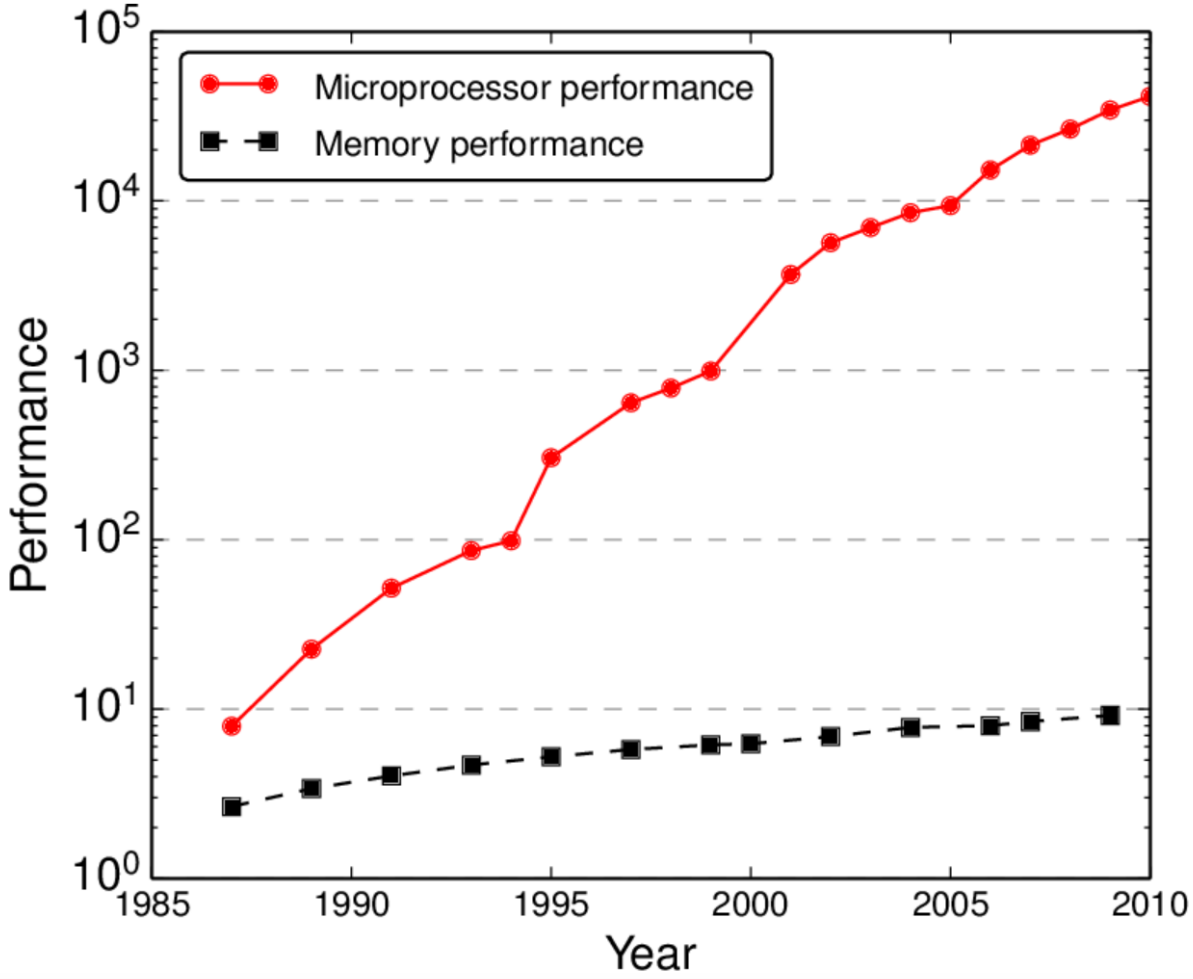
```
generating data set size 1000000
setting data
fitting data
1.46681e+07 milli seconds per fit
D:\online25\DR3\x64\ICC2023\LeaveOneOutRegression.exe (process 79504) exited with code 0.
Press any key to close this window . . .
```

New implementation

```
18.0961 milli seconds per fit
D:\online25\DR3\x64\ICC2023\LeaveOneOutRegression.exe (process 58192) exited with code 0.
Press any key to close this window . . .
```

The Memory Wall.

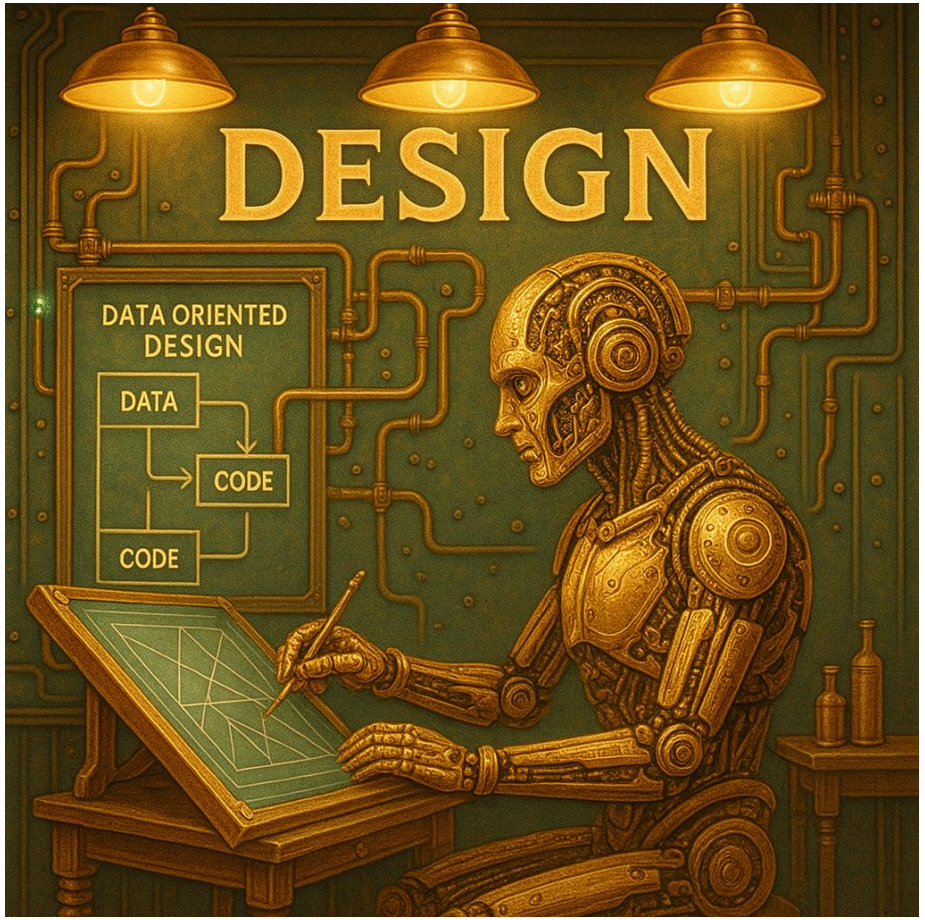
Understand the data => understand the problem.



ROAD MAP

- Data Oriented Design - overview
- Polya Problem Solving
- Examples
  - Matrix example
  - Regression Example
  - Implementation
  - Design for accuracy
- Getting started with Polya

Data Oriented Design



Data-Oriented Design Principles:

- Programs just transform data from one form to another
- Understand the problem by understanding the data
- Understand the cost to understand the problem
- Understand the hardware => understand cost
- Code is data

Data-Oriented Design Principles:

- **CONTEXT** the more you have the better you can make your solution
- **Software** is a real thing running on real hardware
- **Reason** must prevail (evidence)
- Don’t solve problems you don’t have.

Data-Oriented Design Book Richard Fabian

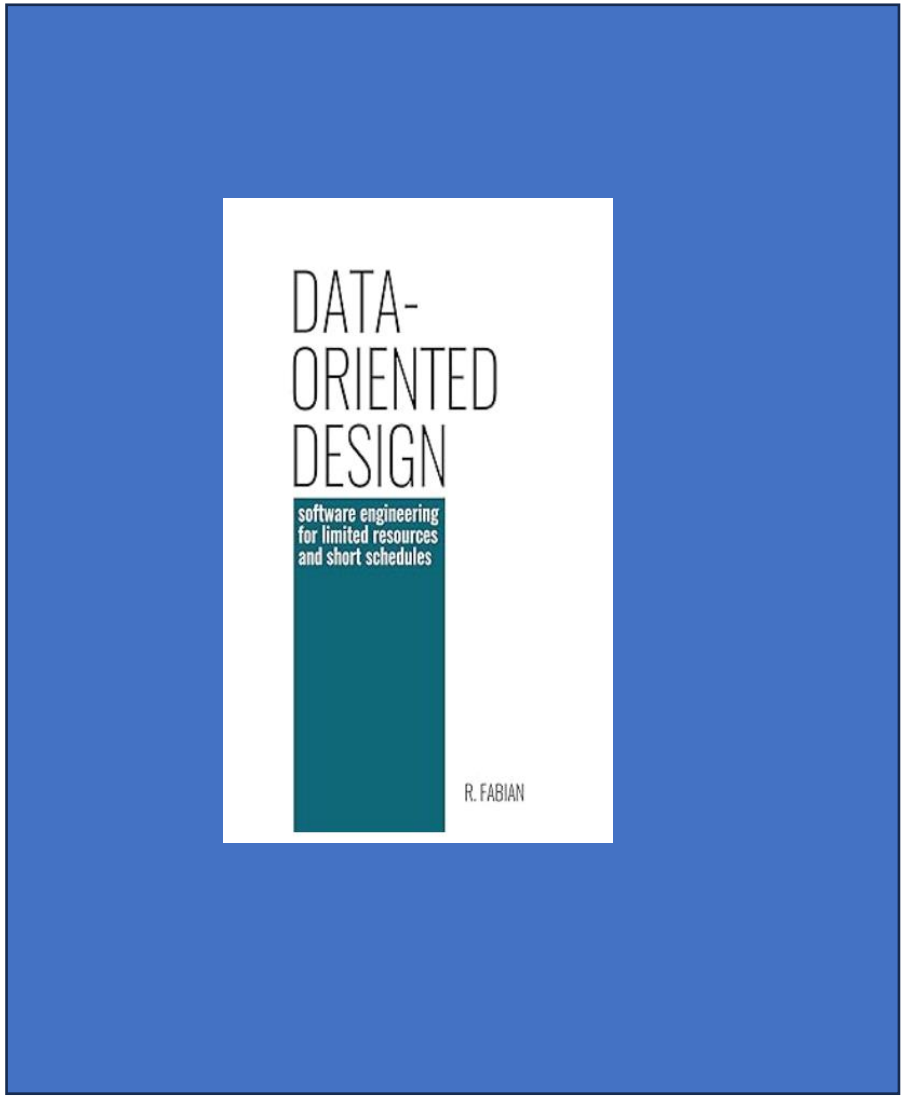
- Also mentions the power of ordering

- https://www.dataorienteddesign.com/dodbook/

- **Revisiting Data-Oriented Design** WEB PDF
  Lucian Radu Teodorescu.
  Overload, 30(167):4-8, February 2022.

- **Vittorio Romeo cppCon2025**

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



- DOD encourages us to focus on data layout and access so we don’t hit the memory wall

- High Art is transforming critical inner loops to SIMD operations applied to contiguous vectors of data.

THE REAL DESIGN PROBLEM

Real Life Problem/Solution Is Highly Dimensional

- Decomposition and algorithm choices
- Compile time off-loading with constexpr
- Spatial layout
- Execution ordering
- Vectorization
- Caching and execution ordering
- Handling Randomness and look-ups

- WE MUST CONSIDER THESE FACTORS IN CONTEXT OF PROBLEM DOMAIN

Sub Problem Grouping

- Logical .. Algorithmic
- Physical spatiotemporal ordering, layout and sequencing
- Problem Domain
  - Idiosyncratic utility function
  - Idiosyncratic side information/ constraints
  - Idiosyncratic invariants or data patterns to exploit

The Missing Step

- The “missing step” in classic Data-oriented Design is a *method* for *working the design problem*—a disciplined, heuristic-driven way to move from “understand the problem” to a concrete, better formulation.

- Considering the whole design problem is hard. It is highly dimensional and extends beyond just code and hardware aspects.

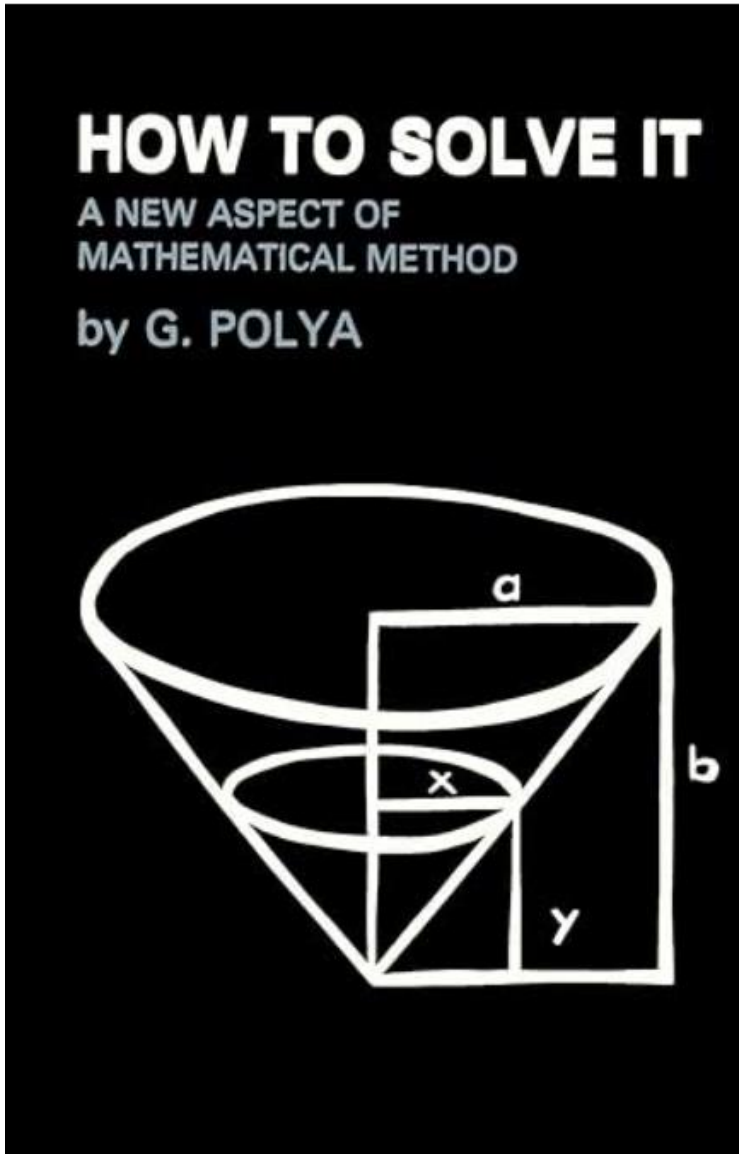
- Domain-specific factors can dominate the design process to such a degree that HFT and games developers' concerns are wildly different.

Solving the Design Problem

A Little Help from George Polya

George Pólya was a Hungarian-American mathematician. He was a professor of mathematics from 1914 to 1940 at ETH Zürich and from 1940 to 1953 at Stanford University. He made fundamental contributions to combinatorics, number theory, numerical analysis and probability theory.

He is also noted for his work in heuristics and mathematics education.[2] He has been described as one of The Martians,[3] an informal category which included one of his most famous students at ETH Zurich, John von Neumann.



"Everyone should know the work of George Polya on how to solve problems"

Marvin Minsky

https://www.hlevkin.com/hlevkin/90MathPhysBioBooks/Math/Polya/George_Polya_How_To_Solve_It_.pdf

Polya In A NutShell

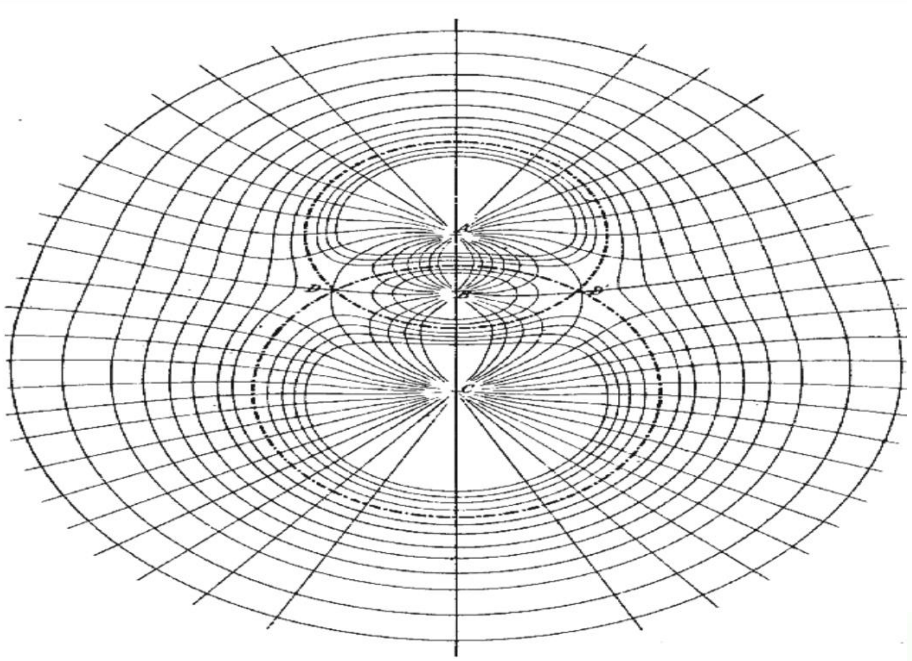
- George Pólya’s method encourages you to:

1. **Understand**
2. **Plan**
3. **Execute**
4. **Reflect**

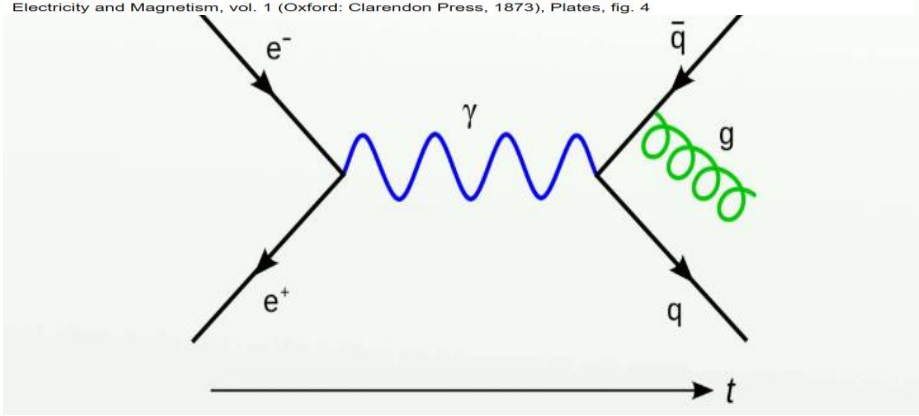
Understand The Problem

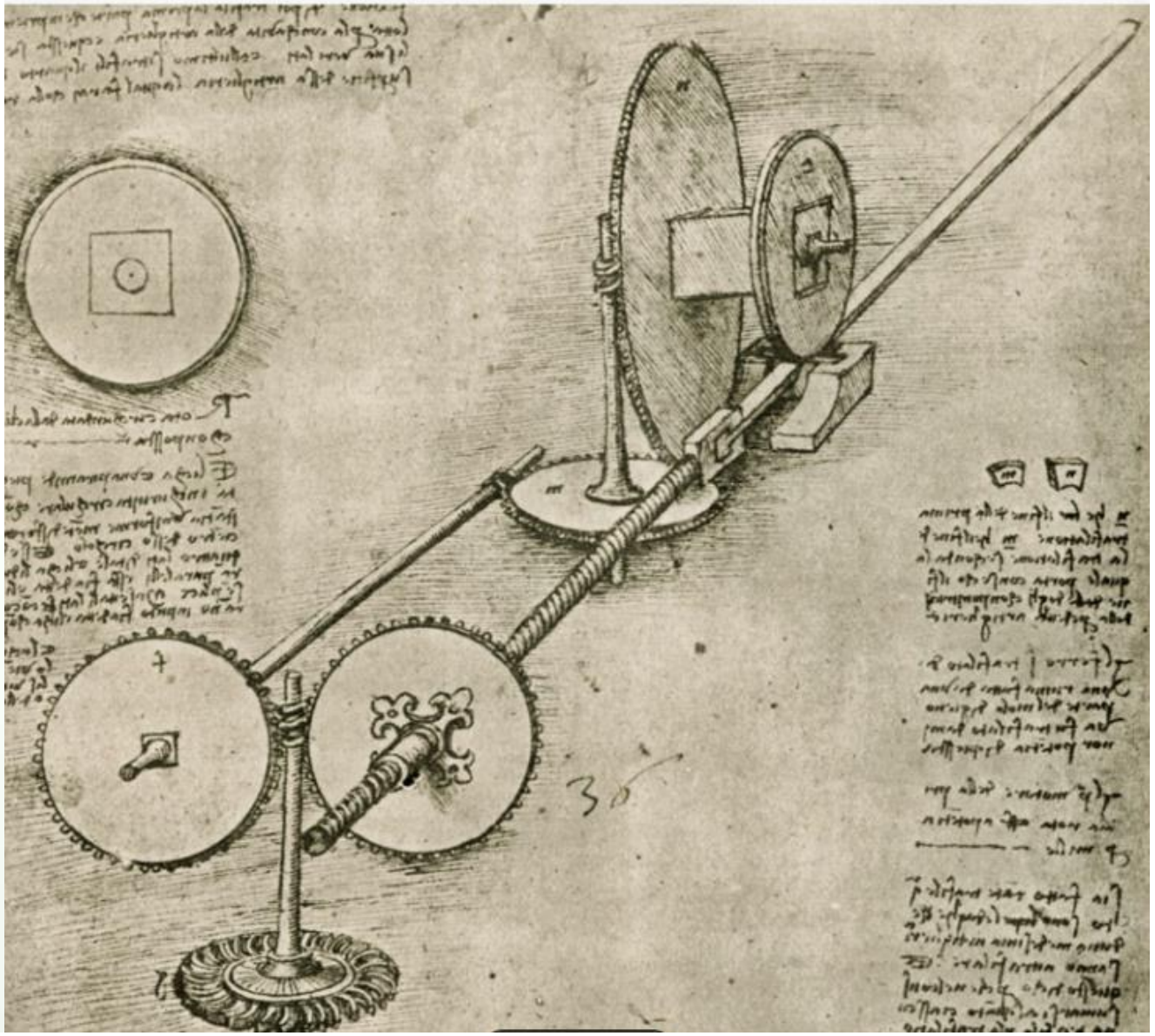
- What is the unknown? What are the data? What is the condition?
- Is it possible to satisfy the condition? Is it sufficient? Or redundant? Or contradictory?
- Separate various parts of the condition. Can I write them down?
- Draw a picture.

Draw a Diagram

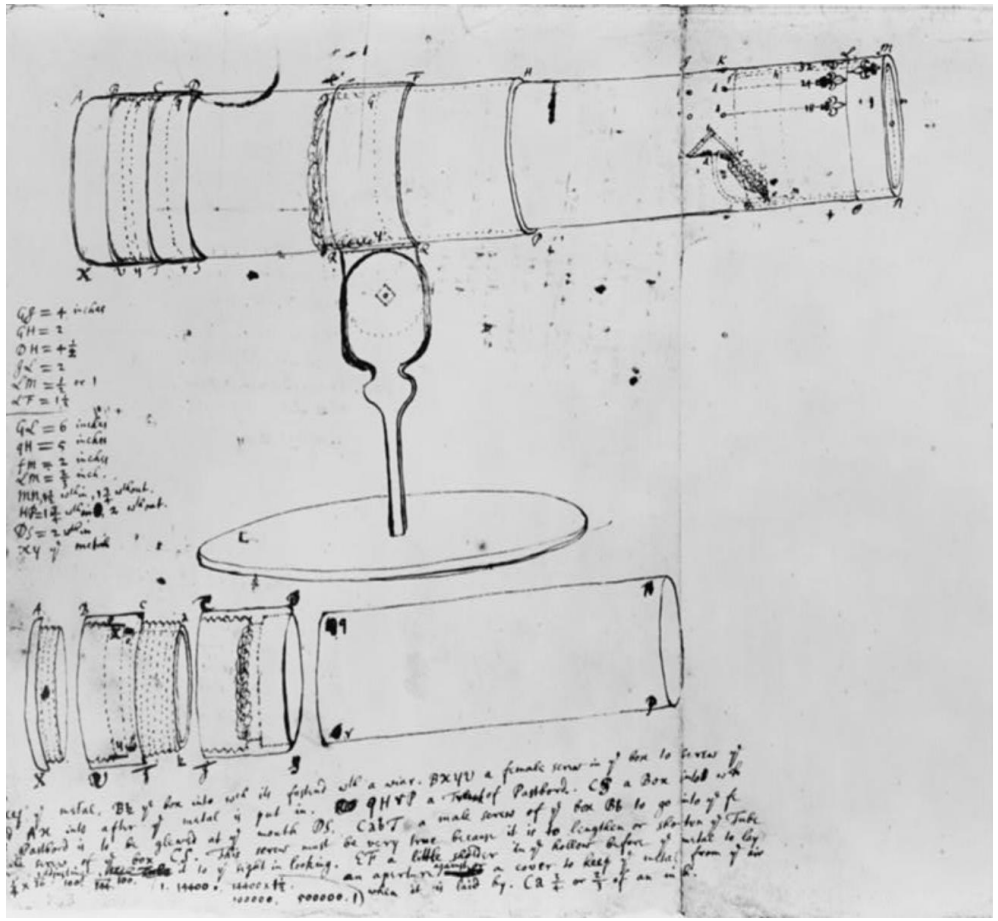


One of James Clerk Maxwell's examples for drawings of lines of force. From James Clerk Maxwell, A Treatise on Electricity and Magnetism, vol. 1 (Oxford: Clarendon Press, 1873), Plates, fig. 4

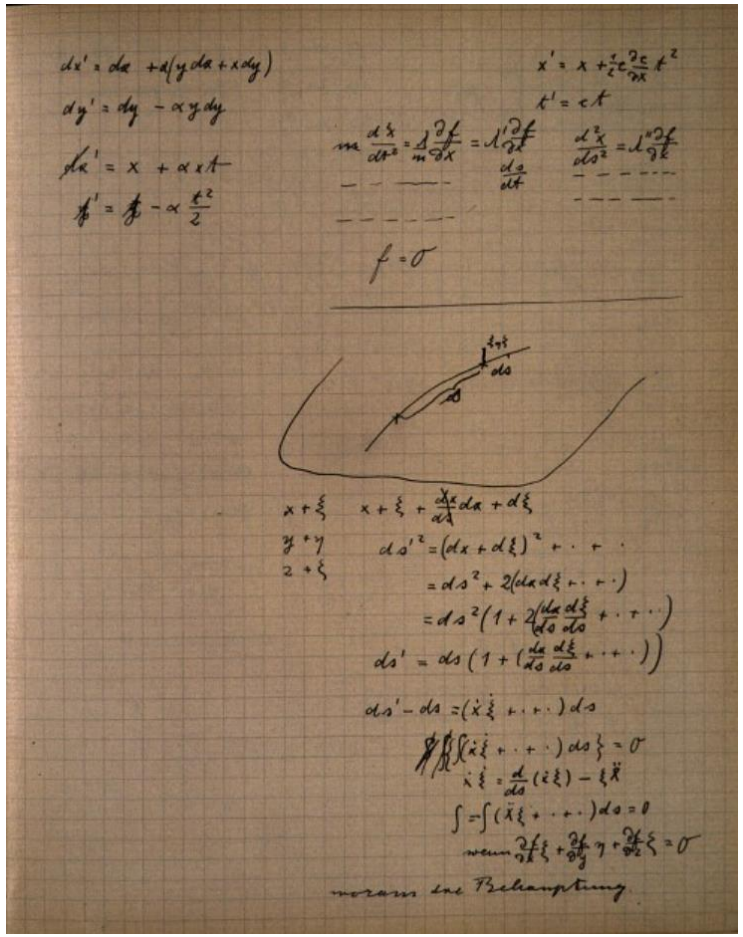




Newton



Einstein



Devise a Plan – Recognition

- Find the connection between the unknown and the data.
- Have you seen and solved the problem before?
- Do you know a related problem?
- Can you restate the problem differently?
- Look at the unknown. Do you know problems giving the same or similar unknowns? Could the same solution approach be used?
- Possibly solve auxiliary problems if no immediate connection between the unknown and the data

Devise a Plan -2 – Auxiliary Problems

- Try to solve a related or easier (auxiliary problem)
- Relax constraints, consider a more general or specific similar problem?
- What happens if you change the data, the unknown? Does this bring you closer to a solution?
- Did you use all the data? Using all the information in our problem space might give us conditions to exploit.
- Have you taken into account all essential notions involved in the problem?

Devise a Plan 3 – Heuristics

- **Key Heuristic Strategies**:
- Decomposition and Recombination
- Analogy
- Generalization and Specialization
- Working Backwards
- Auxiliary Elements (constructions, diagrams, notation, intermediate goals)
- Reduction: mapping the problem to a known problem

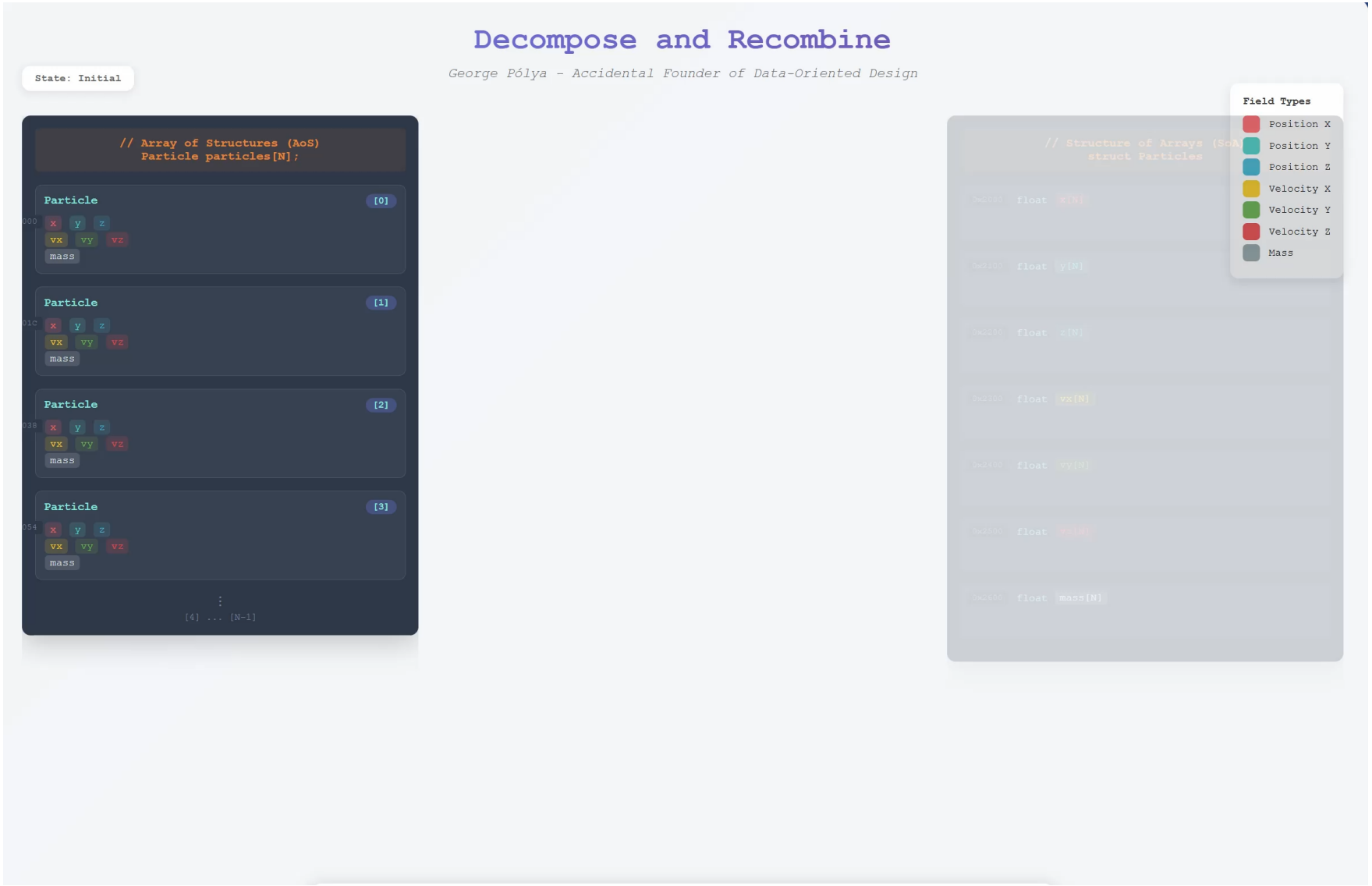
Heuristics/Decomposition?

- Break problems into smaller, independent sub-problems
- We all know this, so what's the big deal?

Heuristics/Decompose and Recombine

- **Problem:** Optimal Data layout

Heuristics/Decompose and Recombine



Heuristics/Analogy

- **Problem:** A simulation accesses data scattered across many small objects.

Heuristics/Analogy

- **Problem:** A simulation accesses data scattered across many small objects.
- **Analogy:**
  Assembling a car engine in a workshop, but every bolt and screw is in a different warehouse on the other side of town.

Heuristics/Analogy

- **Problem: Poor Data Cohesion**
  A simulation accesses data scattered across many small objects
- **Analogy:**
  Assembling a car engine in a workshop, but every bolt and screw is in a different warehouse on the other side of town.
- **What the analogy suggests: ->Use a Workbench**
  Bring what's needed for the current task close together- organise tools and parts by usage, not type

Heuristics/Analogy

- **Problem: Poor Data Cohesion**
  A simulation accesses data scattered across many small objects
- **Analogy:**
  Assembling a car engine in a workshop, but every bolt and screw is in a different warehouse on the other side of town.
- **What the analogy suggests: ->Use a Workbench**
  Bring what's needed for the current task close together- organise tools and parts by usage, not type
- **Solution insight:**
  Group frequently-used data together in memory, and apply spatial locality for access patterns

Heuristics/**Generalisation**

- “Can I solve a *more general* version of this problem first?”
  Paradoxically, generalising often makes the problem *simpler to think about* —
  because you remove distracting details and see the structure more clearly.

Heuristics/**Generalisation**

Problem:

You have a growing set of **if/else** conditions that decide pricing, validation, or access control. Adding new cases keeps breaking old ones.

Generalisation:
Step back: you're not dealing with *pricing rules*, but rather with applying rules to *inputs*. Abstract it into a **data-driven rules engine** or table of conditions.

Outcome:
The generalised model supports all current and future rules.
You didn't patch the code — you elevated the problem from code to *data*.

Heuristics/**Specialisation**

- **Pólya Heuristic: Specialisation**
- “Can I make the problem easier by *fixing some parameters* or looking at a special case?”
- This is often the key to insight — solving an easier case first gives you a foothold.

Heuristics/**Specialisation**

- **Problem:**
  A recursive parser crashes on large input.
  The trace is unreadable.
- **Specialisation:**
  Feed it a tiny example — maybe one line of input.
  Watch the recursion by hand, or in a debugger, step by step.
- **Outcome:**
  The small case exposes the logic flaw or incorrect base case.
  The insight generalises back to the full version.

Heuristics/**Working Backwards**

Pólya Heuristic: Working Backwards:
“Start from the desired outcome and trace backwards the steps or conditions that must be true to get there.”

It’s the same principle mathematicians use in proofs, and engineers use in reverse-engineering or backpropagation.

Heuristics/**Working Backwards – performance**

- **Start** from the *data that must be hot* for performance.
  Ask: “What must have happened earlier to make that true?”
  Then structure your cache key or data layout around that pattern of access.
  This reverse reasoning — from hot data back to the producer — often exposes reuse or reordering opportunities.
- **Example:**
  You discover the *final* computation reuses intermediate values.
  so you work backwards to restructure data flow to preserve them.

3 Carry Out The Plan

- Work through the tasks in the plan. Checking/testing the results of intermediate findings.

- Prototyping, measuring performance , checking that the faster code generates the correct results.

4 Reflection

- Check the results
- Could I get to the same results via a different route?
- Can I see the answer/ solution at a glance?
- Can this effort/result solve other problems?

Diagnosis – Main Reasons For Failure

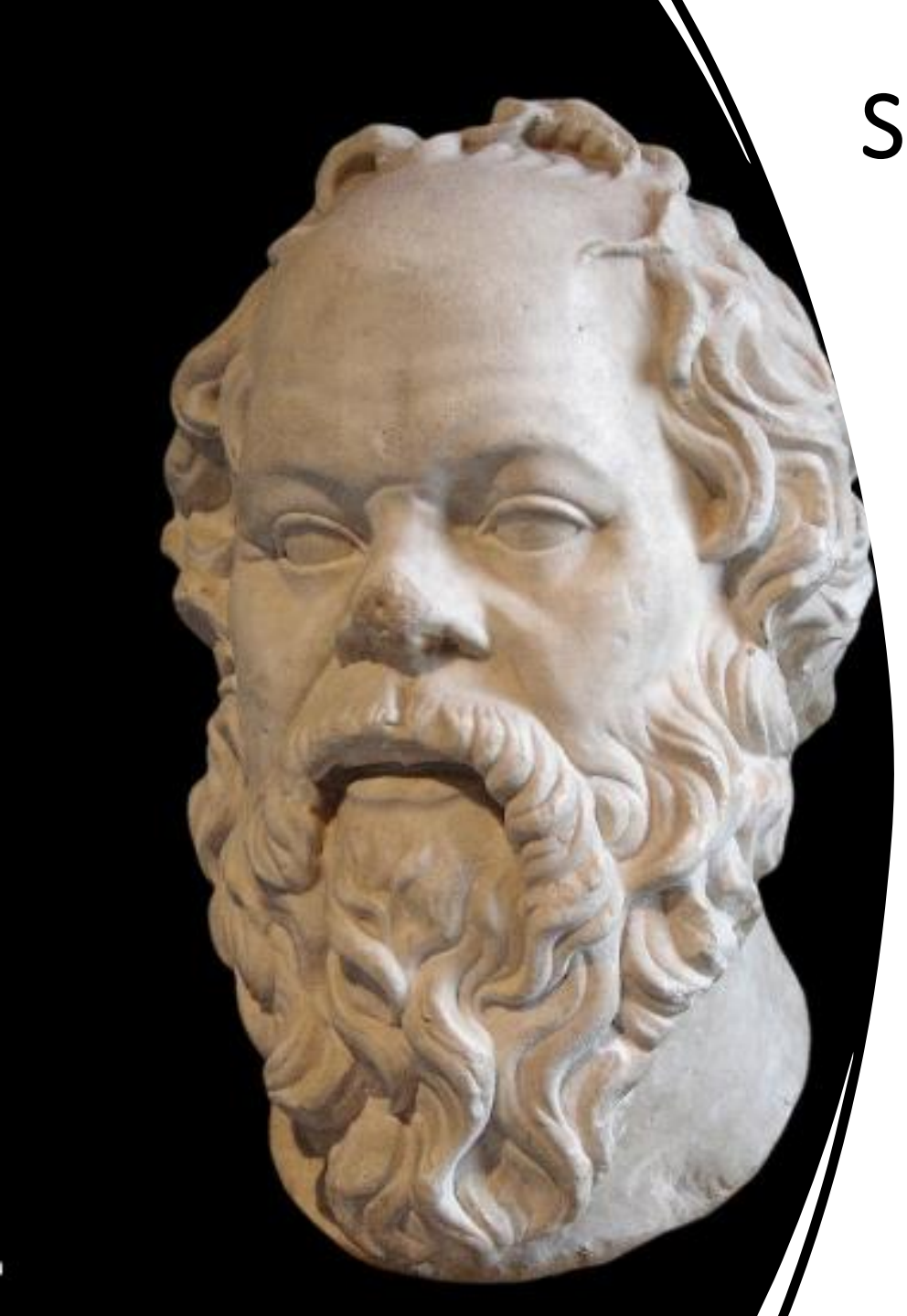
- **Incomplete understanding of the problem:** Lack of concentration on understanding the problem in the initial phase.

- **Planning Failure:** Two opposite faults
  - Rushing in without a plan or general idea
  - Waiting for an idea to come.

- **Execution Failure:** Carelessness

Essential Elements for Success

- Positive Mindset: curiosity, persistence and a growth mindset.
- Notes: Keep notes on thoughts, attempts, and processes.
- Develop discovery through thoughtful questions: Socratic Questioning?



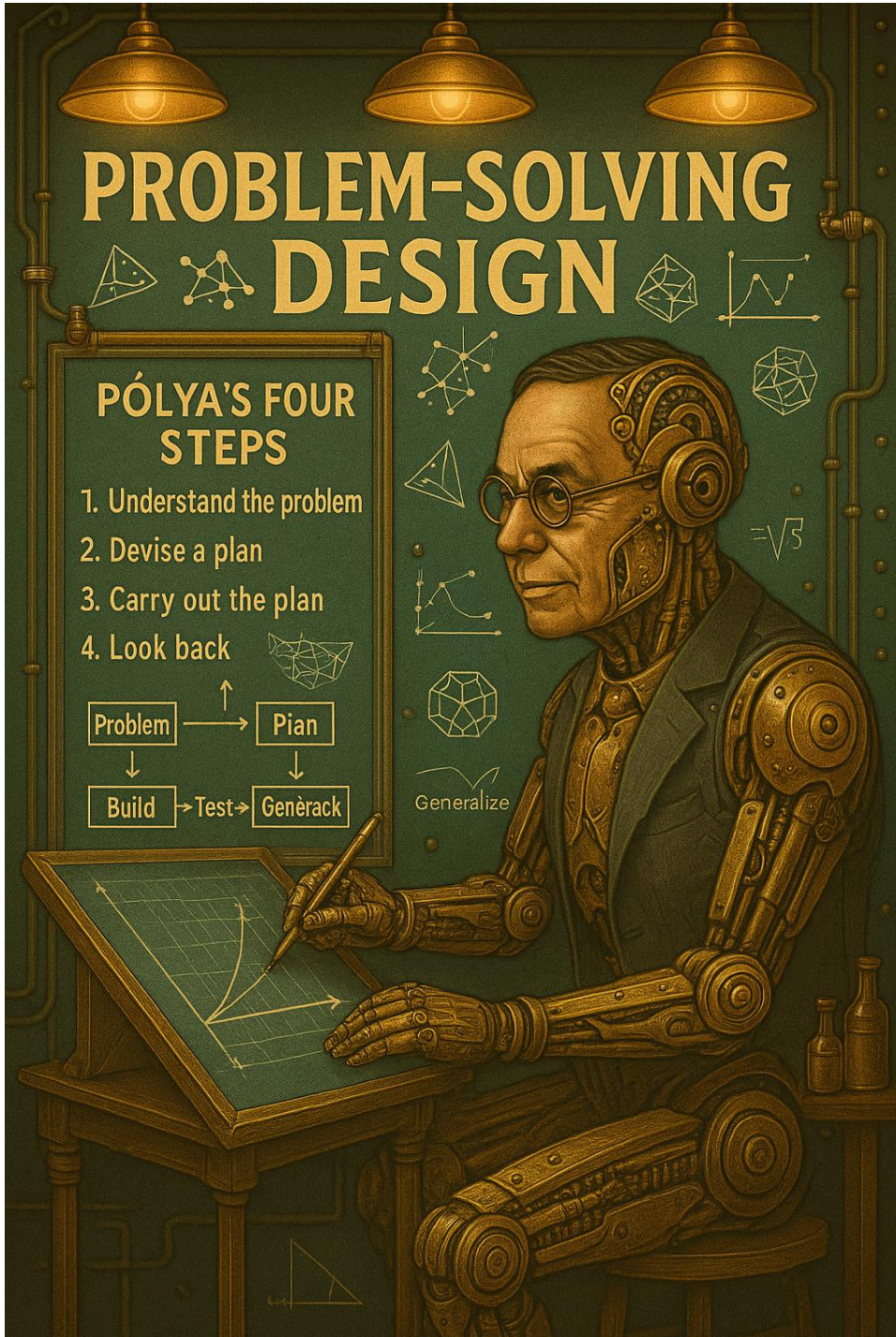
Socratic Questions on DOD

Type	Purpose	Example
Clarification	Define key concepts	"What do we mean by 'cache efficiency'?"
Probing Assumptions	Challenge beliefs	"Why do we assume OOP is the best approach?"
Probing Evidence	Check reasoning	"What benchmarks prove this design is faster?"
Alternative Views	Consider other perspectives	"What would an OOP advocate say about DOD?"
Implications	Explore consequences	"How will DOD affect maintainability?"
Questioning the Question	Reflect on our inquiry	"Are we focusing on the right problem?"

Polya Structure + Heuristics

- **Polya:** Structured problem solving => the how to
- **Process**: Understand =>plan => execute=>reflect
- **Heuristics**: (Activities) to work on the problem: e.g. decomposition
- **Diagnostics:** common failure modes
- **Signs of progress:** simplifications, invariant constraints, beneficial results on the way to solving. Whole condition.
- **Your takeaway:** more places to attack performance and concrete tools/checklists to do it

EXAMPLES


PROBLEM-SOLVING DESIGN
PÓLYA'S FOUR STEPS
1. Understand the problem
2. Devise a plan
3. Carry out the plan
4. Look back
Problem
Pian
Build
Test
Genèrack
Generalize

Example

- A troublesome function in a moment matching pricing algorithm:
- Sum over all elements of a square matrix, where the axis has sets of equal values and monotonic increasing time indices

$$S = \sum_{i=1}^{N} \sum_{j=1}^{N} F(t, T_i, T_j)$$

Initial Code

```
    for (int i = 1; i <= N; ++i)
    {
        for (int j = 1; j <= N; ++j)
        {
            double v = f_impl(t, i, j);
            acc += v;
        }
    }
```

Example

- A troublesome function in a moment matching pricing algorithm:
- Sum over all elements of a square matrix, where the axis has sets of equal values and monotonic increasing time indices

S = $\sum_{i=1}^{N} \sum_{j=1}^{N} F(t, T_i, T_j)$

However inside of expensive function F, F_Impl just takes a single argument, of value $\min(t, T_i, T_j)$

```cpp
inline double f_impl_math(int m)
{
    //the expensive function
    return std::exp(std::sin(static_cast<double>(m)));
}

// f(t,i,j) = exp(sin(min(t,i,j))) ;
inline double f_impl(int t, int i, int j)
{
    int m = std::min(t, std::min(i, j));
    return f_impl_math(m);
}
```

What we calculate

$$S = \sum_{i=1}^{N} \sum_{j=1}^{N} F_{impl}(\min(t, T_i, T_j))$$

Plan

- Simplify, create an auxiliary problem and investigate
- Draw a picture

Simplify => auxiliary problem

- Drop the first argument t:

$$S = \sum_{i=1}^{N} \sum_{j=1}^{N} F_{impl}(\min(t, T_i, T_j))$$

=>

$$S = \sum_{i=1}^{N} \sum_{j=1}^{N} F_{impl}(\min(T_i, T_j))$$

Simplify => auxiliary problem

- Drop the first argument t:

$$S = \sum_{i=1}^{N} \sum_{j=1}^{N} F_{impl}(\min(T_i, T_j))$$

Simplify => auxiliary problem

- Drop the first argument t:

$$S = \sum_{i=1}^{N} \sum_{j=1}^{N} F_{impl}(\min(T_i, T_j))$$

- Change date T_i, T_j to integers

$$S = \sum_{i=1}^{N} \sum_{j=1}^{N} F_{impl}\ (\min(i, j))$$

Simplify => auxiliary problem

- Drop the first argument t:

$$S = \sum_{i=1}^{N} \sum_{j=1}^{N} F_{impl}(\min(T_i, T_j))$$

- Change date T_i, T_j to integers

$$S = \sum_{i=1}^{N} \sum_{j=1}^{N} F_{impl}\ (\min(i, j))$$

- Simplify F_{impl}:

F_{impl}(x) { return x ;}

Simplify => auxiliary problem

- Drop the first argument t:

  S = $\sum_{i=1}^{N} \sum_{j=1}^{N} F_{impl}(\min(T_i, T_j))$

- Change date T_i, T_j to integers

  S= $\sum_{i=1}^{N} \sum_{j=1}^{N} F_{impl}\ (\min(\mathrm{i}, \mathrm{j}))$

- Simplify F_{impl}:

  F_{impl}(x) { return x ;}

Plot the matrix

Draw Example Matrix (7 x7)

	1	2	3	4	5	6	7
1							
2							
3							
4							
5							
6							
7							

set values for min(i,j)

	1	2	3	4	5	6	7
1	1	1	1	1	1	1	1
2	1	2	2	2	2	2	2
3	1	2	3	3	3	3	3
4	1	2	3	4	4	4	4
5	1	2	3	4	5	5	5
6	1	2	3	4	5	6	6
7	1	2	3	4	5	6	7

min(i,j): Highlight the pattern

	1	2	3	4	5	6	7
1	1	1	1	1	1	1	1
2	1	2	2	2	2	2	2
3	1	2	3	3	3	3	3
4	1	2	3	4	4	4	4
5	1	2	3	4	5	5	5
6	1	2	3	4	5	6	6
7	1	2	3	4	5	6	7

min(i,j): Highlight the pattern

	1	2	3	4	5	6	7
1	1	1	1	1	1	1	1
2	1	2	2	2	2	2	2
3	1	2	3	3	3	3	3
4	1	2	3	4	4	4	4
5	1	2	3	4	5	5	5
6	1	2	3	4	5	6	6
7	1	2	3	4	5	6	7

We get N distinct equivalent regions

Size of region $S_i = 2(N - i)+1$

Reduces to a single summation
O(N^2) -> O(N)

$$S = \sum_{i=1}^{N} F(T_i\ \ ) * (2(N-i)+1)$$

- Walk along the diagonal elements call the function and multiply by scale factor (number of elements) and add to running sum

Auxiliary collapsed version

```
double acc = 0.0;
for (int i = 1; i <= N; ++i)
{
    int w = 2*(N - i) + 1;          // multiplicity for level
    double v =  f_impl_math(i);
    acc += static_cast<double>(w) * v;
}
```

Adding the extra variable t, to the min condition

Three variants

- $\mathbf{t < T_1}$
- $\mathbf{t > T_N}$
- $\boldsymbol{T_1 < t < T_N}$

min(t,i,j) where $\mathbf{t < T_1}$ $\boldsymbol{EG}$ $\boldsymbol{t < 1.0}$

	1	2	3	4	5	6	7
1	t	t	t	t	t	t	t
2	t	t	t	t	t	t	t
3	t	t	t	t	t	t	t
4	t	t	t	t	t	t	t
5	t	t	t	t	t	t	t
6	t	t	t	t	t	t	t
7	t	t	t	t	t	t	t

$$S = F(t) * N^2$$

min(t,i,j) where $\mathbf{t > T_N}$ $EG\ t > 7.0$

	1	2	3	4	5	6	7
1	1	1	1	1	1	1	1
2	1	2	2	2	2	2	2
3	1	2	3	3	3	3	3
4	1	2	3	4	4	4	4
5	1	2	3	4	5	5	5
6	1	2	3	4	5	6	6
7	1	2	3	4	5	6	7

$$S = \sum_{i=1} F(Ti) * (2(N - i) + 1)$$

min(t,i,j) where $T_1 < t < T_N$

	1	2	3	4	5	6	7
1	1	1	1	1	1	1	1
2	1	2	2	2	2	2	2
3	1	2	3	3	3	3	3
4	1	2	3	**t**	**t**	**t**	**t**
5	1	2	3	**t**	**t**	**t**	**t**
6	1	2	3	**t**	**t**	**t**	**t**
7	1	2	3	**t**	**t**	**t**	**t**

$$S = \sum_{i=1}^{K} F(Ti) * (2(N-i)+1)$$

$$+F(t) * (N-K)^2$$

Result

- Our new approach gives us between 1 and N calls to the expensive function, as opposed to N^2

- Strategic win, particularly when we move to assets which have matrix elements on an hourly basis instead of monthly

Collapsed with special case added

```cpp
// Collapsed O(N) sum for S = sum_{i=1..N} sum_{j=1..N} exp(sin(min(t,i,j)))
// Works for all t > 0 (integer or non-integer), indices 1..N.
static double sum_collapsed_all_t(int N, double t) {
    const bool  t_is_int = std::fabs(t - std::round(t)) <= 1e-12 * (std::fabs(t) + 1.0);
    const int   L = static_cast<int>(std::floor(t));
    const int   U = static_cast<int>(std::ceil(t));
    double acc = 0.0;

    if (t >= N) {
        for (int k = 1; k <= N; ++k)
            acc += (2.0 * (N - k) + 1.0) * F(static_cast<double>(k));
    } else {
        int up_to = t_is_int ? (L - 1) : L;
        if (up_to > N) up_to = N;
        for (int k = 1; k <= up_to; ++k)
            acc += (2.0 * (N - k) + 1.0) * F(static_cast<double>(k));
        long long side = t_is_int ? (static_cast<long long>(N) - L + 1LL)
                                  : (static_cast<long long>(N) - U + 1LL);
        if (side > 0) acc += static_cast<double>(side * side) * F(t);
    }
    return acc;
}
```

Try It Yourself

- Godbolt here : https://godbolt.org/z/5dqx1Ysd7

Speed up as factor variation of t

```
N=40  reps=1000  (times in milliseconds)
  t small  (t=4.250000):
    collapsed: 0.000 ms    naive: 0.037 ms    speedup: 263.036x
  t medium (t=24.370000):
    collapsed: 0.001 ms    naive: 0.037 ms    speedup: 61.855x
  t large  (t=40.420000):
    collapsed: 0.001 ms    naive: 0.038 ms    speedup: 39.522x
```

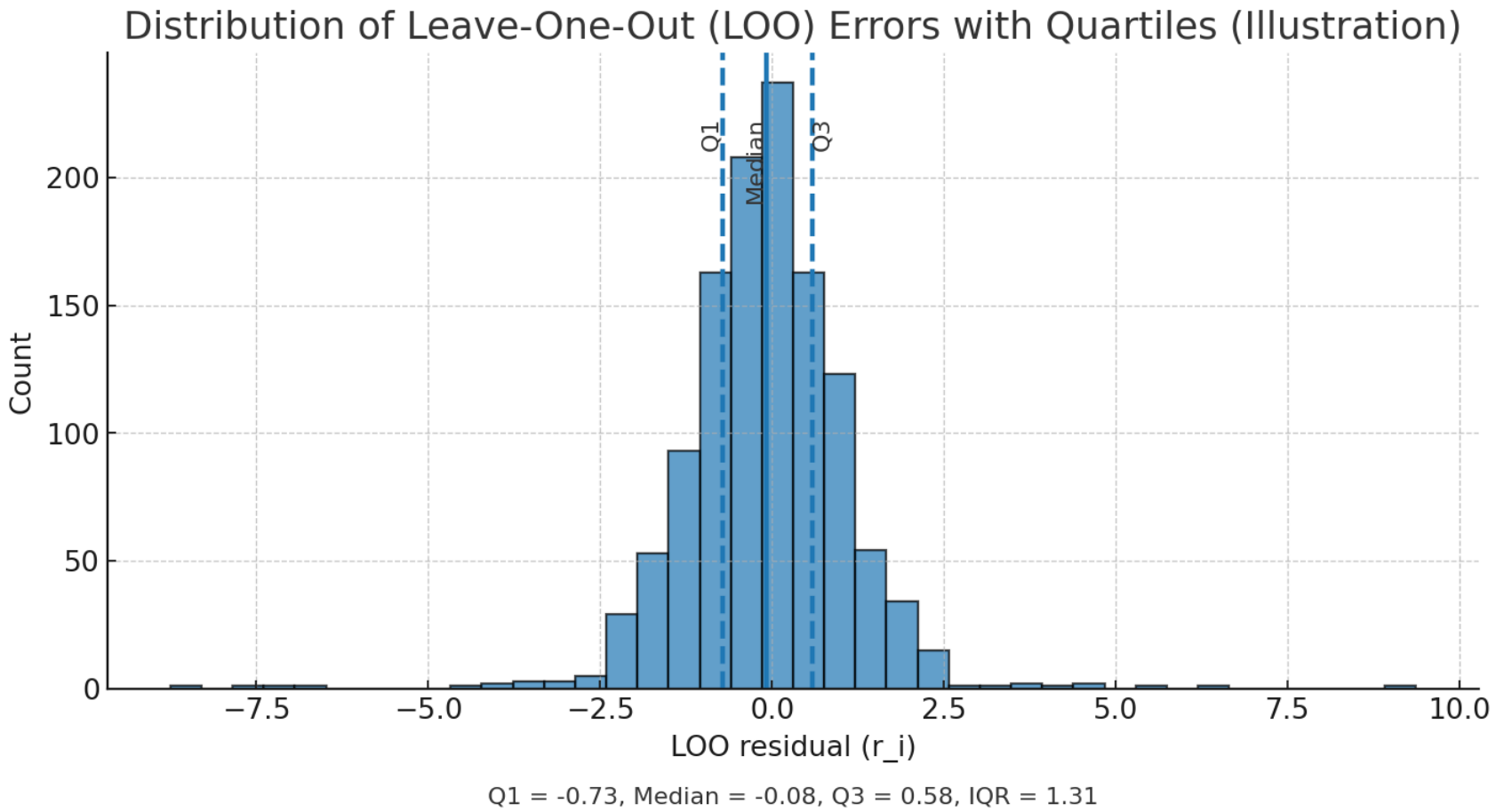
- That’s our first example from real world experience
- A tricky performance problem solved by the insight generated from a simplifed auxillary problem and drawing a picture
- I do have other examples
- Now for the million times faster example

Example Leave One Out Regression

Example Leave One Out Regression

- Machine Learning
- Trying to find predictive factors for simple linear relationships.
- Assessing feature vector/model quality.
- For each data point we forecast the value at that point using a model built from the whole data set minus that point. The difference between forecast and observed gives the error at each point.
- A final step of estimating quantiles or inter-quantile range as a metric for the quality of the fit, and suitability of the relationship as a predictor.

Example loo residuals



The Application

- Use simple linear low (single dimension)
- Generally, will be an $O(N^2)$
- For each (N) data points
  - Fit (N -1) data points and compute residual

Regularised Linear Regression

- Makes more stable by adding a penalty for larger slopes

Understanding The problem

- Questions about data/ size
- Performance requirements
- Accuracy requirements
- Have I solved it before?

- Known solutions, call least squares fit many times with permuted leave one out data

Understanding the problem

- What is the algorithm used
- Why is it so expensive / slow

Model and Objective function

Model $\hat{y}_i = \beta_0 + \beta_1 x_i, \quad i = 1, \ldots$

Ridge Objective $\min_{\beta_0, \beta_1} \sum_{i=1} [y_i - (\beta_0 + \beta_1 x_i)]^2 + \lambda \beta_1^2$

- λ: regularization parameter
- β_0: intercept (not penalized)
- β_1: slope (penalized)

The Design Matrix

$$X = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ 1 & x_3 \\ 1 & x_4 \\ 1 & x_5 \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix}, \quad \boldsymbol{\beta} = \begin{bmatrix} \beta_0 \\ \beta_1 \end{bmatrix}$$

We include an intercept by adding a column of 1's:

Regularisation

We penalize only β_1. Hence, our penalty matrix is:

$$\Lambda = \begin{bmatrix} 0 & 0 \\ 0 & \lambda \end{bmatrix}.$$

Interpretation:

- Top-left entry = 0 $\Longrightarrow$ do **not** penalize β_0.
- Bottom-right entry = λ $\Longrightarrow$ penalize β_1 with strength λ.

The Normal Equations

The **Normal Equation** for ridge with intercept (unpenalized) is:

$$\left(X^\top X + \Lambda\right)\beta = X^\top \mathbf{y}$$

$$\beta = \left(X^\top X + \Lambda\right)^{-1} X^\top \mathbf{y}$$

The Essence of the Regression Calculation

$$X^\top X = \begin{bmatrix} \mathsf{N} & \sum x_i \\ \sum x_i & \sum x_i^2 \end{bmatrix} = \begin{bmatrix} \mathsf{N} & S_x \\ S_x & S_{xx} \end{bmatrix}$$

$$\Lambda = \begin{bmatrix} 0 & 0 \\ 0 & \lambda \end{bmatrix}$$

$$X^\top X + \Lambda = \begin{bmatrix} \mathsf{N} & S_x \\ S_x & S_{xx} \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & \lambda \end{bmatrix} = \begin{bmatrix} \mathsf{N} & S_x \\ S_x & S_{xx} + \lambda \end{bmatrix}$$

$$X^\top \mathbf{y} = \begin{bmatrix} \sum y_i \\ \sum x_i y_i \end{bmatrix} = \begin{bmatrix} S_y \\ S_{xy} \end{bmatrix}$$

Explicit β_0 and β_1

$$\beta_0 = \frac{(S_{xx} + \lambda)\, S_y - S_x\, S_{xy}}{\mathsf{N}\,(S_{xx} + \lambda) - (S_x)^2}$$

$$\beta_1 = \frac{\mathsf{N}\; S_{xy} - S_x\, S_y}{\mathsf{N}\,(S_{xx} + \lambda) - (S_x)^2}$$

Approach

Understand the problem

- Identify slow /expensive parts

Planning

- Explore auxiliary problems that reflect the slow parts
- Draw some pictures
- Generalise solutions to auxiliary problems
- Construct a new algorithm

The slow bit (repeated N times)

$$X^\top X = \begin{bmatrix} \mathsf{N} & \sum x_i \\ \sum x_i & \sum x_i^2 \end{bmatrix} = \begin{bmatrix} \mathsf{N} & S_x \\ S_x & S_{xx} \end{bmatrix}$$

- But we are repeating this N times with slightly different data sets
- It's the first X^TX term that introduces the O(N) dependence into the fitting
- Also, the last term in X^TY

The slow bit (repeated N times)

$$X^\top X = \begin{bmatrix} \mathsf{N} & \sum x_i \\ \sum x_i & \sum x_i^2 \end{bmatrix} = \begin{bmatrix} \mathsf{N} & S_x \\ S_x & S_{xx} \end{bmatrix}$$

How are we going to make this go faster?

The slow bit (repeated N times)

$$X^\top X = \begin{bmatrix} \mathsf{N} & \sum x_i \\ \sum x_i & \sum x_i^2 \end{bmatrix} = \begin{bmatrix} \mathsf{N} & S_x \\ S_x & S_{xx} \end{bmatrix}$$

How are we going to make this go faster?

Un-sequenced reduction and transform reduce!
Scale with threads and SIMD (if we are lucky)

If only this was dereferencing a nullptr!

STOP

If only this was dereferencing a nullptr!

- We have applied an answer we know, to a problem we recognise.
- We have not fully considered the **context.** We have only considered what we might do on an existing inner loop.
- Huge restrictions in the scope of solutions we might consider

Expand scope

- Consider speeding up the whole set of X^TX not just each perturbed version.

- This is a key element

Consider a simpler auxiliary problem

$$X^\top X = \begin{bmatrix} \mathrm{N} & \sum x_i \\ \sum x_i & \sum x_i^2 \end{bmatrix} = \begin{bmatrix} \mathrm{N} & S_x \\ S_x & S_{xx} \end{bmatrix}$$

How does this single element compute, vary over all the different leave-one-out summations we will do?

Consider a simpler auxiliary problem

- One of the summations in the X^TX matrix, for all the leave-one-out perturbations
- Pick leave one out sum of X_i
- Draw a picture, or work an example by hand.

Summation rows leaving out an element

1.1	2.2	3.3	4.4	5.5	6.6	7.7	8.8	9.9	10.1	11.11	
1.1	2.2	3.3	4.4	5.5	6.6	7.7	8.8	9.9	10.1		12.12
1.1	2.2	3.3	4.4	5.5	6.6	7.7	8.8	9.9		11.11	12.12
1.1	2.2	3.3	4.4	5.5	6.6	7.7	8.8		10.1	11.11	12.12
1.1	2.2	3.3	4.4	5.5	6.6	7.7		9.9	10.1	11.11	12.12
1.1	2.2	3.3	4.4	5.5	6.6		8.8	9.9	10.1	11.11	12.12
1.1	2.2	3.3	4.4	5.5		7.7	8.8	9.9	10.1	11.11	12.12
1.1	2.2	3.3	4.4		6.6	7.7	8.8	9.9	10.1	11.11	12.12
1.1	2.2	3.3		5.5	6.6	7.7	8.8	9.9	10.1	11.11	12.12
1.1	2.2		4.4	5.5	6.6	7.7	8.8	9.9	10.1	11.11	12.12
1.1		3.3	4.4	5.5	6.6	7.7	8.8	9.9	10.1	11.11	12.12

Lots of repetition

Signs of progress

- We are looking at the problem differently. We have broadened the context
- We understand the problem better.
- A huge amount of repetition, we must surely be able to find a way to exploit this.

Summation rows leaving out an element

1.1	2.2	3.3	4.4	5.5	6.6	7.7	8.8	9.9	10.1	11.11	
1.1	2.2	3.3	4.4	5.5	6.6	7.7	8.8	9.9	10.1		12.12

Can we reuse the sum for the first row to calculate the second row ?

Summation rows leaving out an element

1.1	2.2	3.3	4.4	5.5	6.6	7.7	8.8	9.9	10.1	11.11	
1.1	2.2	3.3	4.4	5.5	6.6	7.7	8.8	9.9	10.1		12.12

Can we re-use the sum for the first row to calculate the second row ?

$$Sum\_loo\_1 = Sum\_loo\_0 + x\_0 - x\_1$$
$$Sum\_loo\_2 = Sum\_loo\_0 + x\_0 - x\_2$$

Lots of repetition

Summation rows leaving out an element

1.1	2.2	3.3	4.4	5.5	6.6	7.7	8.8	9.9	10.1	11.11	
1.1	2.2	3.3	4.4	5.5	6.6	7.7	8.8	9.9	10.1		12.12

But Sum_loo_0 + x_0 is just the sum of all the elements
So each leave one out sum(i) = Sum_all – x_i

So, compute the sum over the whole row:
Then compute leave-one-out sums by subtracting the left-out element

Generalisation ?

- Does this generalise to all our leave one out sums ?

Summation rows leaving out an element

1.1	2.2	3.3	4.4	5.5	6.6	7.7	8.8	9.9	10.1	11.11	
1.1	2.2	3.3	4.4	5.5	6.6	7.7	8.8	9.9	10.1		12.12
1.1	2.2	3.3	4.4	5.5	6.6	7.7	8.8	9.9		11.11	12.12
1.1	2.2	3.3	4.4	5.5	6.6	7.7	8.8		10.1	11.11	12.12
1.1	2.2	3.3	4.4	5.5	6.6	7.7		9.9	10.1	11.11	12.12
1.1	2.2	3.3	4.4	5.5	6.6		8.8	9.9	10.1	11.11	12.12
1.1	2.2	3.3	4.4	5.5		7.7	8.8	9.9	10.1	11.11	12.12
1.1	2.2	3.3	4.4		6.6	7.7	8.8	9.9	10.1	11.11	12.12
1.1	2.2	3.3		5.5	6.6	7.7	8.8	9.9	10.1	11.11	12.12
1.1	2.2		4.4	5.5	6.6	7.7	8.8	9.9	10.1	11.11	12.12
1.1		3.3	4.4	5.5	6.6	7.7	8.8	9.9	10.1	11.11	12.12

Lots of repetition

12.12	-12.12	1.1	2.2	3.3	4.4	5.5	6.6	7.7	8.8	9.9	10.1	11.11	
11.11	-11.11	1.1	2.2	3.3	4.4	5.5	6.6	7.7	8.8	9.9	10.1		12.12
10.1	-10.1	1.1	2.2	3.3	4.4	5.5	6.6	7.7	8.8	9.9		11.11	12.12
9.9	-9.9	1.1	2.2	3.3	4.4	5.5	6.6	7.7	8.8		10.1	11.11	12.12
8.8	-8.8	1.1	2.2	3.3	4.4	5.5	6.6	7.7		9.9	10.1	11.11	12.12
7.7	-7.7	1.1	2.2	3.3	4.4	5.5	6.6		8.8	9.9	10.1	11.11	12.12
6.6	-6.6	1.1	2.2	3.3	4.4	5.5		7.7	8.8	9.9	10.1	11.11	12.12
5.5	-5.5	1.1	2.2	3.3	4.4		6.6	7.7	8.8	9.9	10.1	11.11	12.12
4.4	-4.4	1.1	2.2	3.3		5.5	6.6	7.7	8.8	9.9	10.1	11.11	12.12
3.3	-3.3	1.1	2.2		4.4	5.5	6.6	7.7	8.8	9.9	10.1	11.11	12.12
2.2	-2.2	1.1		3.3	4.4	5.5	6.6	7.7	8.8	9.9	10.1	11.11	12.12

Add two columns containing + and - the omitted values: these don’t change the horizontal sum

Summation rows leaving out an element

12.12	1.1	2.2	3.3	4.4	5.5	6.6	7.7	8.8	9.9	10.1	11.11		-12.12
11.11	1.1	2.2	3.3	4.4	5.5	6.6	7.7	8.8	9.9	10.1		12.12	-11.11
10.1	1.1	2.2	3.3	4.4	5.5	6.6	7.7	8.8	9.9		11.11	12.12	-10.1
9.9	1.1	2.2	3.3	4.4	5.5	6.6	7.7	8.8		10.1	11.11	12.12	-9.9
8.8	1.1	2.2	3.3	4.4	5.5	6.6	7.7		9.9	10.1	11.11	12.12	-8.8
7.7	1.1	2.2	3.3	4.4	5.5	6.6		8.8	9.9	10.1	11.11	12.12	-7.7
6.6	1.1	2.2	3.3	4.4	5.5		7.7	8.8	9.9	10.1	11.11	12.12	-6.6
5.5	1.1	2.2	3.3	4.4		6.6	7.7	8.8	9.9	10.1	11.11	12.12	-5.5
4.4	1.1	2.2	3.3		5.5	6.6	7.7	8.8	9.9	10.1	11.11	12.12	-4.4
3.3	1.1	2.2		4.4	5.5	6.6	7.7	8.8	9.9	10.1	11.11	12.12	-3.3
2.2	1.1		3.3	4.4	5.5	6.6	7.7	8.8	9.9	10.1	11.11	12.12	-2.2

Move negative column to right hand side

Summation rows leaving out an element

12.12	1.1	2.2	3.3	4.4	5.5	6.6	7.7	8.8	9.9	10.1	11.11		-12.12
11.11	1.1	2.2	3.3	4.4	5.5	6.6	7.7	8.8	9.9	10.1		12.12	-11.11
10.1	1.1	2.2	3.3	4.4	5.5	6.6	7.7	8.8	9.9		11.11	12.12	-10.1
9.9	1.1	2.2	3.3	4.4	5.5	6.6	7.7	8.8		10.1	11.11	12.12	-9.9
8.8	1.1	2.2	3.3	4.4	5.5	6.6	7.7		9.9	10.1	11.11	12.12	-8.8
7.7	1.1	2.2	3.3	4.4	5.5	6.6		8.8	9.9	10.1	11.11	12.12	-7.7
6.6	1.1	2.2	3.3	4.4	5.5		7.7	8.8	9.9	10.1	11.11	12.12	-6.6
5.5	1.1	2.2	3.3	4.4		6.6	7.7	8.8	9.9	10.1	11.11	12.12	-5.5
4.4	1.1	2.2	3.3		5.5	6.6	7.7	8.8	9.9	10.1	11.11	12.12	-4.4
3.3	1.1	2.2		4.4	5.5	6.6	7.7	8.8	9.9	10.1	11.11	12.12	-3.3
2.2	1.1		3.3	4.4	5.5	6.6	7.7	8.8	9.9	10.1	11.11	12.12	-2.2

Use LHS column values to fill leave one out spaces

Summation rows leaving out an element

0	1.1	2.2	3.3	4.4	5.5	6.6	7.7	8.8	9.9	10.1	11.11	12.12	-12.12
0	1.1	2.2	3.3	4.4	5.5	6.6	7.7	8.8	9.9	10.1	11.11	12.12	-11.11
0	1.1	2.2	3.3	4.4	5.5	6.6	7.7	8.8	9.9	10.1	11.11	12.12	-10.1
0	1.1	2.2	3.3	4.4	5.5	6.6	7.7	8.8	9.9	10.1	11.11	12.12	-9.9
0	1.1	2.2	3.3	4.4	5.5	6.6	7.7	8.8	9.9	10.1	11.11	12.12	-8.8
0	1.1	2.2	3.3	4.4	5.5	6.6	7.7	8.8	9.9	10.1	11.11	12.12	-7.7
0	1.1	2.2	3.3	4.4	5.5	6.6	7.7	8.8	9.9	10.1	11.11	12.12	-6.6
0	1.1	2.2	3.3	4.4	5.5	6.6	7.7	8.8	9.9	10.1	11.11	12.12	-5.5
0	1.1	2.2	3.3	4.4	5.5	6.6	7.7	8.8	9.9	10.1	11.11	12.12	-4.4
0	1.1	2.2	3.3	4.4	5.5	6.6	7.7	8.8	9.9	10.1	11.11	12.12	-3.3
0	1.1	2.2	3.3	4.4	5.5	6.6	7.7	8.8	9.9	10.1	11.11	12.12	-2.2

Use LHS column values to fill leave one out spaces

Summation across all rows in matrix and write sum in LH column

82.83	1.1	2.2	3.3	4.4	5.5	6.6	7.7	8.8	9.9	10.1	11.11	12.12	-12.12
82.83	1.1	2.2	3.3	4.4	5.5	6.6	7.7	8.8	9.9	10.1	11.11	12.12	-11.11
82.83	1.1	2.2	3.3	4.4	5.5	6.6	7.7	8.8	9.9	10.1	11.11	12.12	-10.1
82.83	1.1	2.2	3.3	4.4	5.5	6.6	7.7	8.8	9.9	10.1	11.11	12.12	-9.9
82.83	1.1	2.2	3.3	4.4	5.5	6.6	7.7	8.8	9.9	10.1	11.11	12.12	-8.8
82.83	1.1	2.2	3.3	4.4	5.5	6.6	7.7	8.8	9.9	10.1	11.11	12.12	-7.7
82.83	1.1	2.2	3.3	4.4	5.5	6.6	7.7	8.8	9.9	10.1	11.11	12.12	-6.6
82.83	1.1	2.2	3.3	4.4	5.5	6.6	7.7	8.8	9.9	10.1	11.11	12.12	-5.5
82.83	1.1	2.2	3.3	4.4	5.5	6.6	7.7	8.8	9.9	10.1	11.11	12.12	-4.4
82.83	1.1	2.2	3.3	4.4	5.5	6.6	7.7	8.8	9.9	10.1	11.11	12.12	-3.3
82.83	1.1	2.2	3.3	4.4	5.5	6.6	7.7	8.8	9.9	10.1	11.11	12.12	-2.2

Use LHS column values to fill leave one out spaces

Summation across all rows in matrix and write sum in LH column

82.83	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	-12.12
82.83	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	-11.11
82.83	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	-10.1
82.83	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	-9.9
82.83	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	-8.8
82.83	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	-7.7
82.83	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	-6.6
82.83	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	-5.5
82.83	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	-4.4
82.83	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	-3.3
82.83	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	-2.2

Use LHS contains sum over all elements

Summation across all rows in matrix and write sum in LH column

82.83	-12.12
82.83	-11.11
82.83	-10.1
82.83	-9.9
82.83	-8.8
82.83	-7.7
82.83	-6.6
82.83	-5.5
82.83	-4.4
82.83	-3.3
82.83	-2.2

Use LHS contains sum over all elements
RH column contains left out values

What are the costs for this solution?

- First sum over whole row to get Sn cost N
- Then create each leave one out sum Sk SN –xi cost N operations
- We can compute all our leave one out sums in 2 N operations

The New Algorithm : Summations over set

- For all leave one out sums, sum over the complete set S_N
- $S_N(x)$
- $S_N(1)$
- $S_N(x^2)$
- $S_N(xy)$
- $S_N(y)$

- Generate Leave one out at index i, by subtracting $f(x_i)$ vector from the sum
- For each set of Loo sums
  - N ops to reduce
  - N ops to create Loo sums

- 2N operations

Leave one out vector sums

- For all leave-one-out sums, sum over the complete set S_N
- Leave one out $Sx\_i = S_N(x) - x\_i$ vector of $(x\_i * -1) + Sx\_i$
- Leave one out $Sn\_i = S_N(1) = N-1$
- Leave one out $Sxx\_i = S_N(x^2) - (x\_i * x\_i)$ element-wise multiply $x\_i * x\_i$
- Leave one out $Sxy\_i = S_N(xy) - (x\_i * y\_i)$ element-wise multiply $x\_i\ y\_i$
- Leave one out $S_{y\_i} = S_N(y) - y\_i$ vector of $(y\_i * -1) + Sy\_i$

New Algorithm

- Generate the new fits using the closed-form expressions for Betas
- 1) Generate sums over the whole data set Sx, Sxx, Sy, Sxy
- 2) Generate leave one out sums by subtracting vectors of x, xx, y, xy from corresponding sum
- 3) Evaluate the form solution using vector operations

Explicit β_0 and β_1

After computing the inverse, we get:

$$\beta_0 = \frac{(S_{xx} + \lambda)\, S_y - S_x\, S_{xy}}{\mathsf{N}(S_{xx} + \lambda) - (S_x)^2}, \qquad \beta_1 = \frac{\mathsf{N} S_{xy} - S_x\, S_y}{\mathsf{N}(S_{xx} + \lambda) - (S_x)^2}.$$

Note the denominators are the same !

Reflection

Reflection

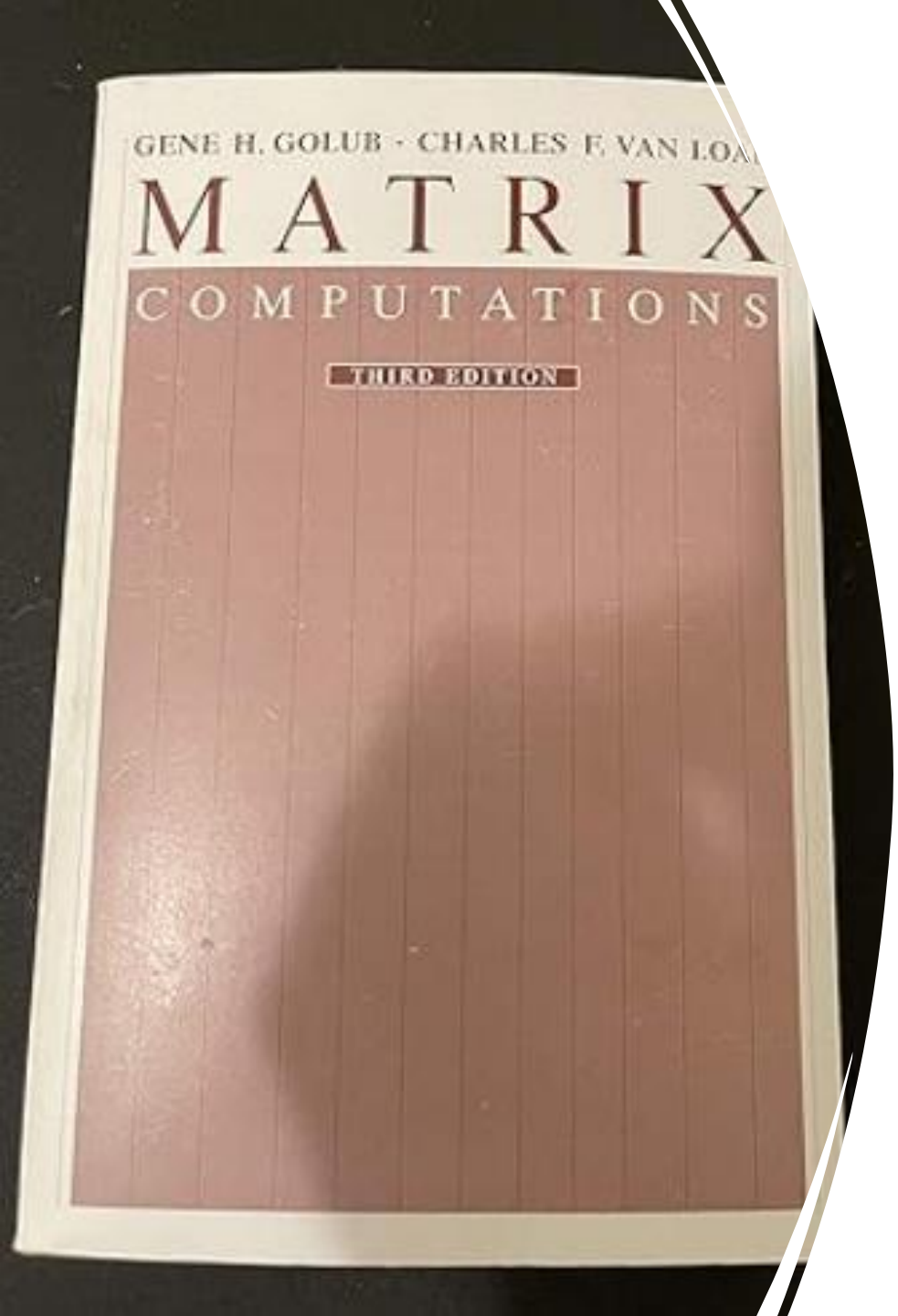
- This was an important illustration of the benefit of not just trying to optimise the inner loop.
- The benefits of expanding context
- Looking at a simpler, auxiliary sub-problem
- We might also look at calculating an example by hand

Generalisation Reflection

- We can generalise our Trick.
- If we are using reduction to calculate N values on perturbations of a set of data of size N. And there is an inverse operation to the reduction operation

- Generate O(N) perturbed sets
  - Generate Reduction value for each O(N)
  - -> $O(N^2)$

- Generate reduction on whole set O(N)
- Apply inverse operation on whole set result for each permutation O(1)
  - -> O(N)



Reflection

- Had we studied some post-grad course in computational matrix methods
- We might have seen it as some matrix update problem

Technical

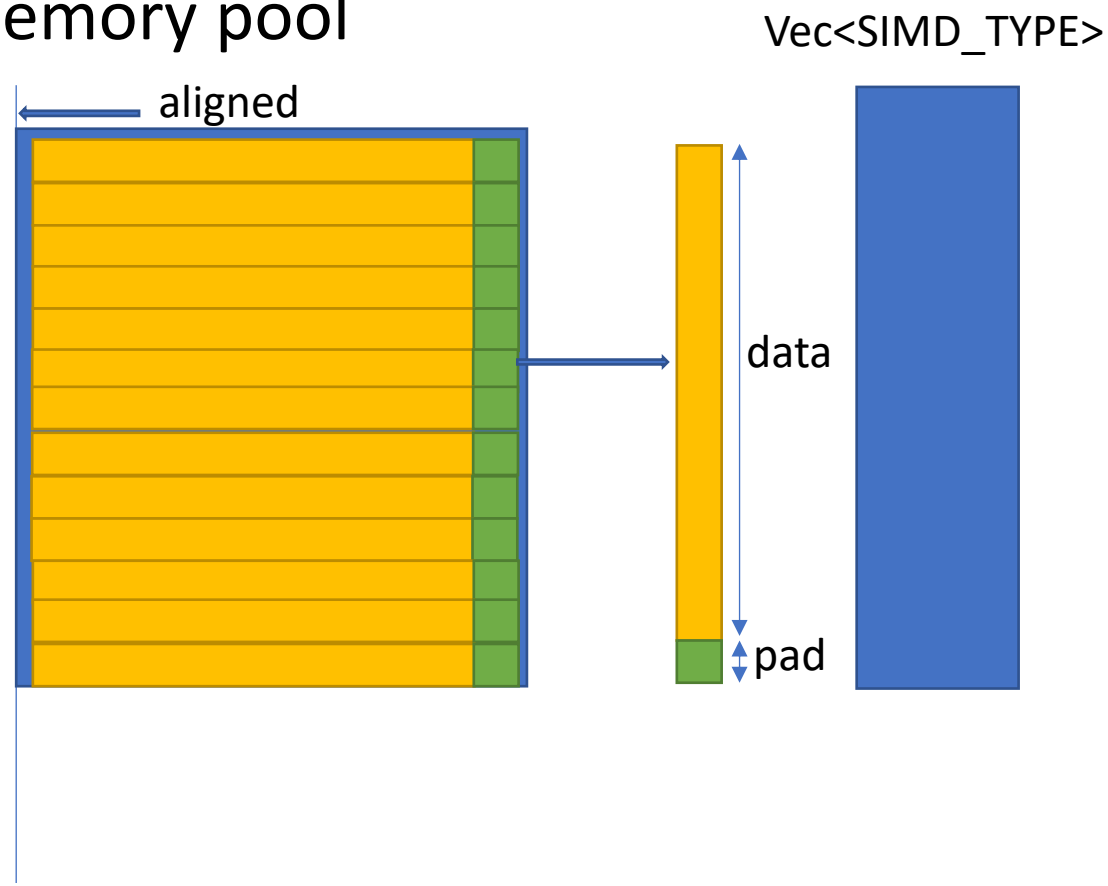
Hardware Memory and Vectorisation

- Use DR3

- Contiguous memory layout
- Memory pool so efficient memory allocation
- Vectorised math operations

VecXX Utility

Memory pool



Math Operators and Functions

vec_A = vec_B+ vec_C

vec_A >= vec_B

vec_A = sin(vec_C)

VecXX

- Memory managed vector type
- Supports math functions and operations
- Contiguous, aligned and padded
- Can change the scalar type and instruction set
- Substitutable for scalar type so we drop into existing code to make it vectorised

Create a vectorized version using DR3

- Code available on https://github.com/andyD123/DR3
- Implementation using vectorised library.
- Using contiguous memory layout and vectorised instructions
- We can transform scalar code into vector code.
- Using auto to avoid explicitly indicating vector or scalar typing.

Explicit β_0 and β_1

After computing the inverse, we get:

$$\beta_0 = \frac{(S_{xx} + \lambda)\, S_y - S_x\, S_{xy}}{\mathsf{N}(S_{xx} + \lambda) - (S_x)^2}, \qquad \beta_1 = \frac{\mathsf{N} S_{xy} - S_x\, S_y}{\mathsf{N}(S_{xx} + \lambda) - (S_x)^2}.$$

Note the denominators are the same !

```
auto MULT = [](auto x, auto y) { return x * y; };
auto SUM = [](auto x, auto y) { return x + y; };
auto SQR = [](auto x) {return x * x; };

//compute reductions for Sx,Sy,Sxx,Sxy
auto S_x = reduce(data_X, SUM);
auto S_y = reduce(data_Y, SUM);
auto S_xx = transformReduce(data_X, SQR, SUM);
auto S_xy = transformReduce(data_X, data_Y, MULT, SUM);

// compute leave one out vectors

auto SX_loo = S_x - data_X;   //leave one out SX
auto SY_loo = S_y - data_Y;   //leave one out SY

auto data_X_squared = data_X * data_X;
auto SXX_loo = S_xx - data_X_squared; //leave one out SXX

auto data_X_Y = data_X * data_Y;
auto SXY_loo = S_xy - data_X_Y;  //leave one out SXY

double lambda = 0.0;// 0.1;  //regularisation parameter
double Sz = data_X.size() - 1.0;

// Compute the fit parameters
auto denominator = (Sz * (SXX_loo + lambda)) - (SX_loo * SX_loo);

auto Beta_0_numerator = (SXX_loo + lambda) * SY_loo - SX_loo * SXY_loo;
auto Beta_0 = Beta_0_numerator / denominator; //vector of fits for Beta 0 offsets

auto Beta_1_numerator = Sz * SXY_loo - (SX_loo * SY_loo);
auto Beta_1 = Beta_1_numerator / denominator;  //vector of Beta_1   slopes
```

$$\beta_0 = \frac{(S_{xx} + \lambda)\, S_y - S_x\, S_{xy}}{\mathsf{N}(S_{xx} + \lambda) - (S_x)^2}$$

$$\beta_1 = \frac{\mathsf{N} S_{xy} - S_x\, S_y}{\mathsf{N}(S_{xx} + \lambda) - (S_x)^2}$$

Leave One Out Regression- Performance Run

- 1 million elements LOO

```
18.0961 milli seconds per fit
D:\online25\DR3\x64\ICC2023\LeaveOneOutRegression.exe (process 58192) exited with code 0.
Press any key to close this window . . .
```

```
generating data set size 1000000
setting data
fitting data
1.46681e+07 milli seconds per fit
D:\online25\DR3\x64\ICC2023\LeaveOneOutRegression.exe (process 79504) exited with code 0.
Press any key to close this window . . .
```

How can we make these summations more accurate?

Heuristics Mini: Work-Backwards

- How do we make the floating point addition more accurate?

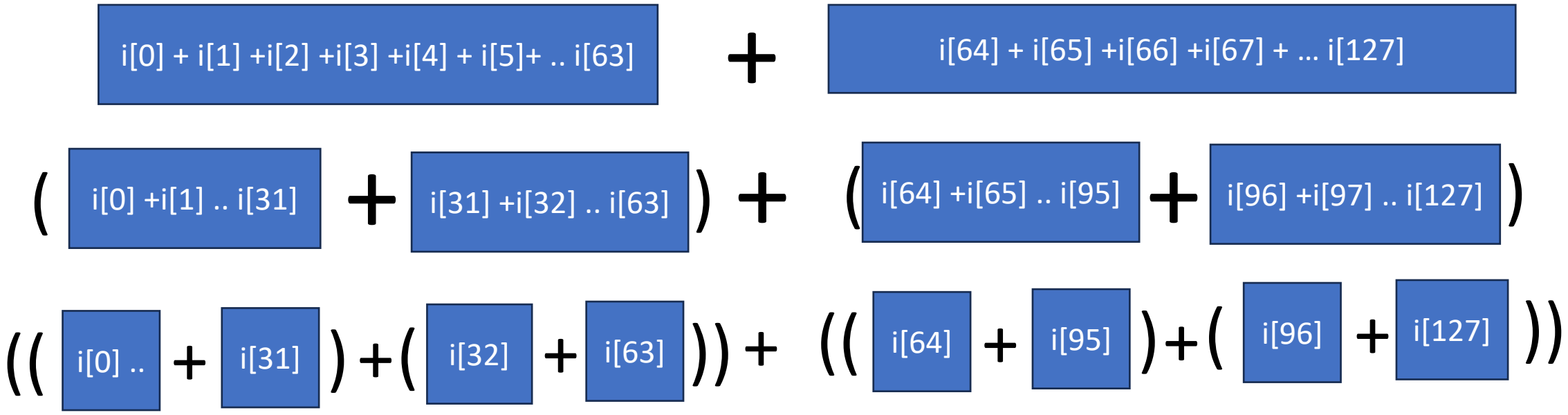
Heuristics Mini: Work-Backwards

- How do we make the floating point addition more accurate?
- Most accurate when combining **similar-magnitude** partial sums

Heuristics Mini: Work-Backwards

- How do we make the sums of floating point more accurate?
- Most accurate when combining **similar-magnitude** partial sums
- Recurse halves -> quarters-> pairs => pairwise (balanced) summation

Pairwise Summation



Lets Experiment

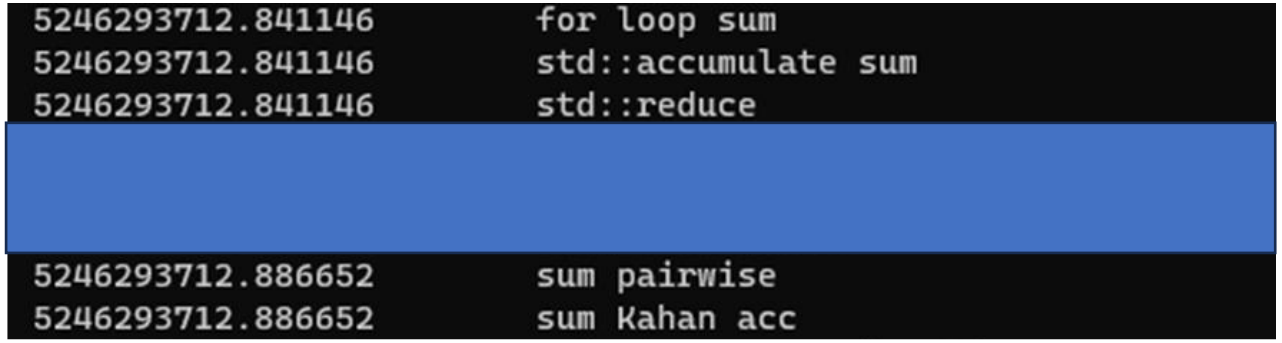
- 10 Billion 0.0-1.0 numbers added up using
- For loop
- Std::accumulate
- Std::reduce

- Pairwise reduce
- Kahan summation

- What happens if we permute the input data

Results

Sums of the same set of numbers using Kahan and pairwise



The other methods all agree

They say the sum is

5246293712.886652

Results
Sums of the same set of numbers using Kahan and pairwise

```
5246293712.841456  5246293712.841735  5246293712.841146        for loop sum
5246293712.841456  5246293712.841735  5246293712.841146        std::accumulate sum
5246293712.841456  5246293712.841735  5246293712.841146        std::reduce

5246293712.886652  5246293712.886652  5246293712.886652        sum pairwise
5246293712.886652  5246293712.886652  5246293712.886652        sum Kahan acc
```

The other methods all agree

They say the sum is

5246293712.886652

Results
Sums of the same set of numbers using Kahan and pairwise

5246293712.840066	5246293712.841456	5246293712.841735	5246293712.841146	for loop sum
5246293712.840066	5246293712.841456	5246293712.841735	5246293712.841146	std::accumulate sum
5246293712.840066	5246293712.841456	5246293712.841735	5246293712.841146	std::reduce
5246293712.886653	5246293712.886652	5246293712.886652	5246293712.886652	sum pairwise
5246293712.886652	5246293712.886652	5246293712.886652	5246293712.886652	sum Kahan acc

The other methods all agree
They say the sum is
5246293712.886652

```
auto MULT = [](auto x, auto y) { return x * y; };
auto SUM = [](auto x, auto y) { return x + y; };
auto SQR = [](auto x) {return x * x; };

//compute reductions for Sx,Sy,Sxx,Sxy
auto S_x = reduce(data_X, SUM);
auto S_y = reduce(data_Y, SUM);
auto S_xx = transformReduce(data_X, SQR, SUM);
auto S_xy = transformReduce(data_X, data_Y, MULT, SUM);

// compute leave one out vectors

auto SX_loo = S_x - data_X;   //leave one out SX
auto SY_loo = S_y - data_Y;   //leave one out SY

auto data_X_squared = data_X * data_X;
auto SXX_loo = S_xx - data_X_squared; //leave one out SXX

auto data_X_Y = data_X * data_Y;
auto SXY_loo = S_xy - data_X_Y;  //leave one out SXY

double lambda = 0.0;// 0.1;  //regularisation parameter
double Sz = data_X.size() - 1.0;

// Compute the fit parameters
auto denominator = (Sz * (SXX_loo + lambda)) - (SX_loo * SX_loo);

auto Beta_0_numerator = (SXX_loo + lambda) * SY_loo - SX_loo * SXY_loo;
auto Beta_0 = Beta_0_numerator / denominator; //vector of fits for Beta 0 offsets

auto Beta_1_numerator = Sz * SXY_loo - (SX_loo * SY_loo);
auto Beta_1 = Beta_1_numerator / denominator;  //vector of Beta_1   slopes
```

$$\beta_0 = \frac{(S_{xx} + \lambda)\, S_y - S_x\, S_{xy}}{\mathsf{N}(S_{xx} + \lambda) - (S_x)^2}$$

$$\beta_1 = \frac{\mathsf{N} S_{xy} - S_x\, S_y}{\mathsf{N}(S_{xx} + \lambda) - (S_x)^2}$$

```
auto MULT = [](auto a, auto b) { return a * b; };
auto SUM = [](auto a, auto b) { return a + b; };
auto SQR = [](auto a) { return a * a; };

// Pairwise (accurate) reductions
auto S_x  = pairwise_reduce(x, SUM);
auto S_y  = pairwise_reduce(y, SUM);
auto S_xx = pairwise_transformReduce(x, SQR, SUM);
auto S_xy = pairwise_transformReduce(x, y, MULT, SUM);

auto SX_loo = S_x - x;
auto SY_loo = S_y - y;
auto data_X_squared = x * x;
auto SXX_loo = S_xx - data_X_squared;
auto data_X_Y = x * y;
auto SXY_loo = S_xy - data_X_Y;

double Sz = x.size() - 1.0;
auto SXX_loo_plus_lambda = SXX_loo + lambda;

auto denominator = (Sz * SXX_loo_plus_lambda) - (SX_loo * SX_loo);
auto inv_denominator = 1.0 / denominator;

auto Beta_0_numerator = SXX_loo_plus_lambda * SY_loo - SX_loo * SXY_loo;
auto Beta_0 = Beta_0_numerator * inv_denominator;

auto Beta_1_numerator = Sz * SXY_loo - (SX_loo * SY_loo);
auto Beta_1 = Beta_1_numerator * inv_denominator;
```

Take Aways

- Design addresses a highly dimensional problem.
  - **Logical** algorithm design
  - **Physical** optimising spatiotemporal memory use patterns
  - **Idiosyncratic** aspects of the actual problem itself.

- Working through the problem space in a structured way by using approaches such as Polya's can help

- Always look to expand the context of your thinking around problem areas. This can be very useful in keeping brilliant solutions in play.

- Drawing pictures and solving easier ancillary problems is unlikely to be wasted time

If you cannot solve the proposed problem

- Try to solve first some related problem …
- Human superiority lies in …going around the obstacle that cannot be overcome directly, in devising some suitable auxiliary problem when the original one appears insoluble.
- Example of a Polya conducting a problem-solving session with students https://www.youtube.com/watch?v=h0gbw-Ur_do

Next Step Getting Started

- Get a copy of “How to Solve It book”

  Read the foreword by Professor Ian Stewart

  Read the section “Practical Problems”

- “The Algorithm Design Manual”

- Notes on heuristics https://github.com/andyD123/cppCon25
- code available on https://github.com/andyD123/DR3
- contact e mail - andreedrakeford@hotmail.com