Computational Research 101

Experimental Design - Data Analysis

Software Engineering

Domenico Salvagnin

DEI, University of Padova

IPCO 2023
Madison, WI

Motivation

Computational research is key ingredient in many areas

Deceptively simple:

1. Implement algorithm
2. Compare against existing methods
3. Publish

Not so fast…

Need proper practices for:

Correctness

Reproducibility of Results

Sharing

Outline

Experimental design

Data analysis

Writing code

Conclusions

Experimental Design

Why experiments?

Increasing gap between theory and practice:

- Worst-case analysis not always very telling
- Average-case analysis often relies on questionable assumptions
- Complex algorithms and data structures too hard/impossible to analyze theoretically
- Many complex algorithms never implemented at all :-(

Why experiments?

Inherent value in efficient implementations:

- Algorithm engineering can improve runtime by orders of magnitude
- Theories are confirmed *and* suggested by experimentation
- Empirical science no easier than theoretical one (just different)

Which type of paper?

Different types of papers:

- Application
- Horse Race
- Experimental Analysis

The Scientific Method

1. exploration
2. formulate hypothesis or question
3. design experiment to test its validity
4. data analysis
5. draw conclusions

and reiterate!

Basic Principles

Ask interesting questions

Use appropriate testsets

Use appropriate experimental design

Use reasonable efficient implementation

Ensure reproducibility

Test significance and draw justified conclusions

Interesting Questions

Exploratory experimentation to find good questions

Think before you compute

- Which interactions are you planning to investigate?
- Are you collecting the right data?
- What are the potential outcomes? How would they affect the hyphotesis?

Don't spend too much time (on the wrong experiments)

Testsets

Cannot clearly test on all possible instances…

Need the (finite) subset to be representative of the actual instances of interest

Easier said than done…

Testsets

Large and heterogeneous enough

Right level of difficulty (avoid ceiling/floor effects)

Avoid over/underrepresentation of problem classes

Real-world vs randomly-generated models

Types of Experiments

Manipulation

Observation

Factorial

Pitfalls

Missing control experiment (related: confirmation bias)

Data collection biases (e.g., survival bias)

Overtuning

- Split into training / testing / validation
- Avoid data leakage

Pitfalls

Do *not* compare algorithms that play a different game:

- Exact methods vs. heuristics
- Exact methods with different stopping criteria (e.g. optimality tolerances!)

You never compare algorithms, but rather their implementation!

Reasonable implementation

Naïve implementation can hide fundamental weaknesses of the proposed approach

Unnecessarily inefficient implementation is a limit factor in how much testing you are able to perform

Do not overdo it! (see *profiling* tomorrow)

Reproducibility

Keep track of all necessary metadata about the experiments

Provide enough information for another researcher to be able to replicate the experiment

- No need to replicate the same runtime and/or path
- But an equivalent experiment should be consistent and allowing to draw the same conclusions

Reproducibility

Code and paper must match

Horse race papers: instances and/or the code publicly available

Tell the full story:

- Do not overly aggregate data (put detailed results online, in a proper format!)
- Do not hide/omit anomalous results
- Always report running times (even if not the main focus)

Speaking of time…

If measuring time (*almost always*), need to enforce reliable measures:

- *Identical* machines
- *Exclusive* usage
- No background processes
- No Turboboost and Hyperthreading (or similar)
- Avoid unnecessary I/O (disk and network) while benchmarking

What about the timelimit?

Introduces a bias *against* methods that solve more models

Inherently nondeterministic and nonreproducible

…but it is a practical necessity: use the highest your resources can afford!

[No, PAR-x is not the answer]

What performance measures?

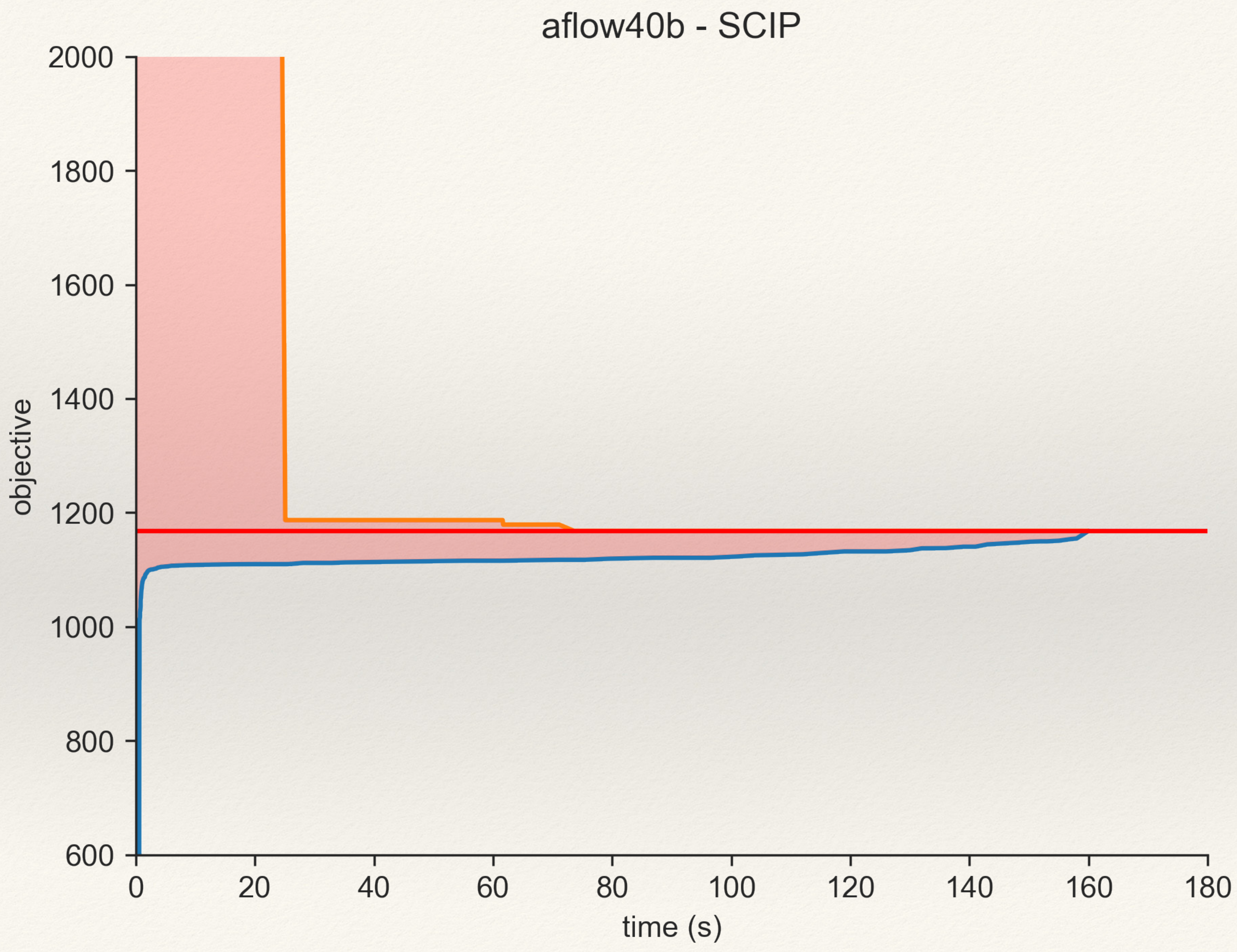
time to optimality (but remember time limit bias!)

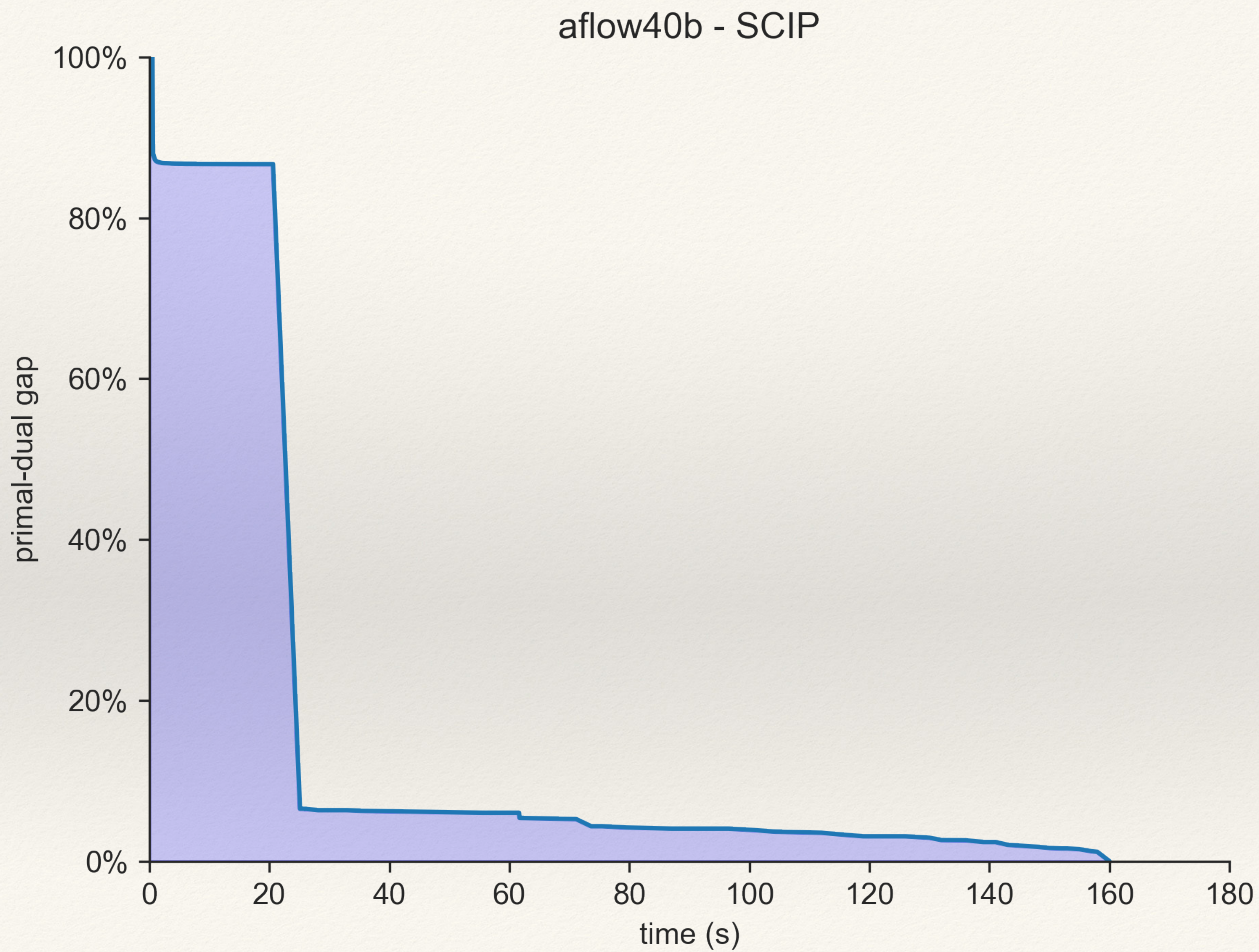
- variant: time to x% gap

number of instances solved (within resource limits)

B&B nodes (only meaningful for instances solved to optimality by all methods under comparison!)

Primal-dual integral (gives more global view on solution process)


aflow40b - SCIP
objective
time (s)
2000
1800
1600
1400
1200
1000
800
600
0
20
40
60
80
100
120
140
160
180


aflow40b - SCIP
primal-dual gap
100%
80%
60%
40%
20%
0%
0
20
40
60
80
100
120
140
160
180
time (s)

What performance measures?

What about heuristics?

- Number of solutions found (success rate)
- Time to first solution
- Time to optimal solution
- Time to solution within X% gap
- Primal integral

Justified Conclusions

Use appropriate data analysis tools

Interpret the data: look for patterns/explanations

Avoid statements not supported by data

Data Analysis

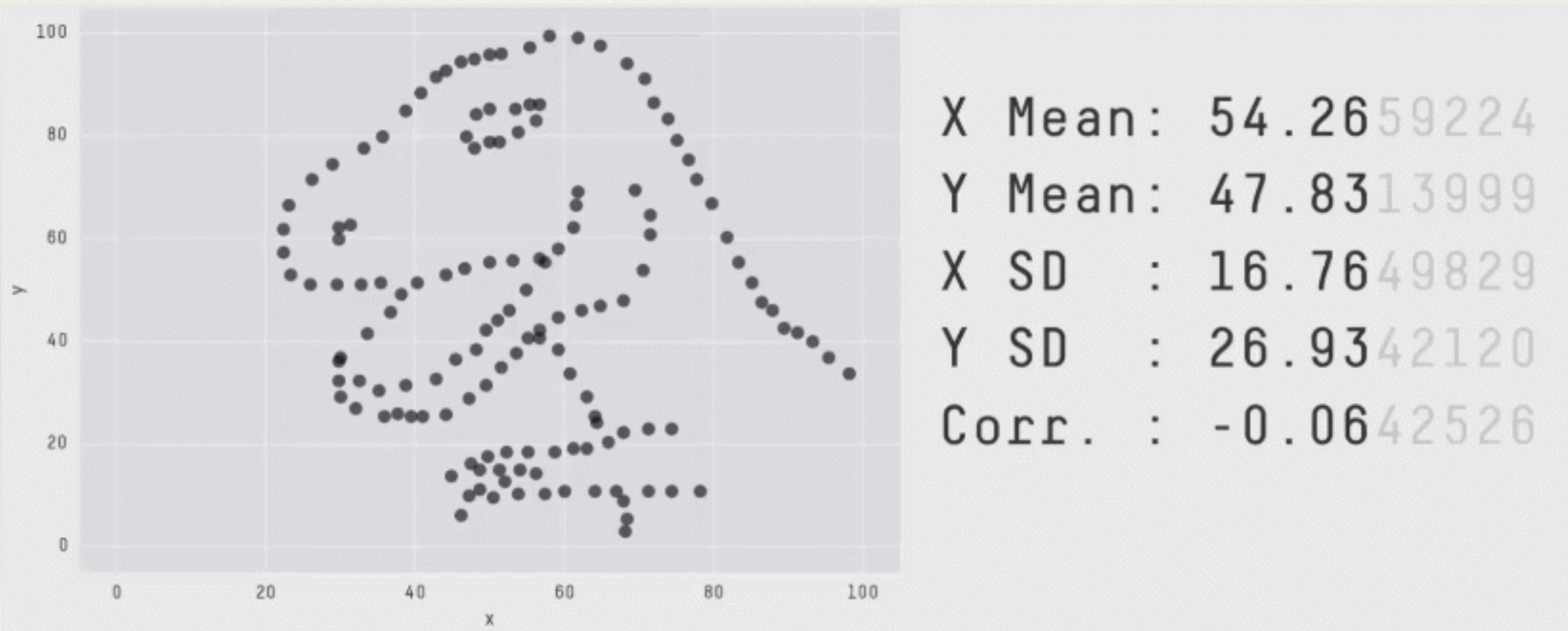
How to aggregate results?

Performance variability

Statistical significance of results

How to aggregate numbers?

First of all: aggregation always leads to information loss



Always look at the whole dataset before aggregation!

Means are mean

Obvious choice: arithmetic mean

- Proportional to total runtime in the real world
- Unsuited for normalized numbers
- Too sensitive to big numbers

Geometric mean: too sensitive to small numbers

Quartiles / medians: too insensitive to measure progress

Common empirical tradeoff: shifted geometric mean

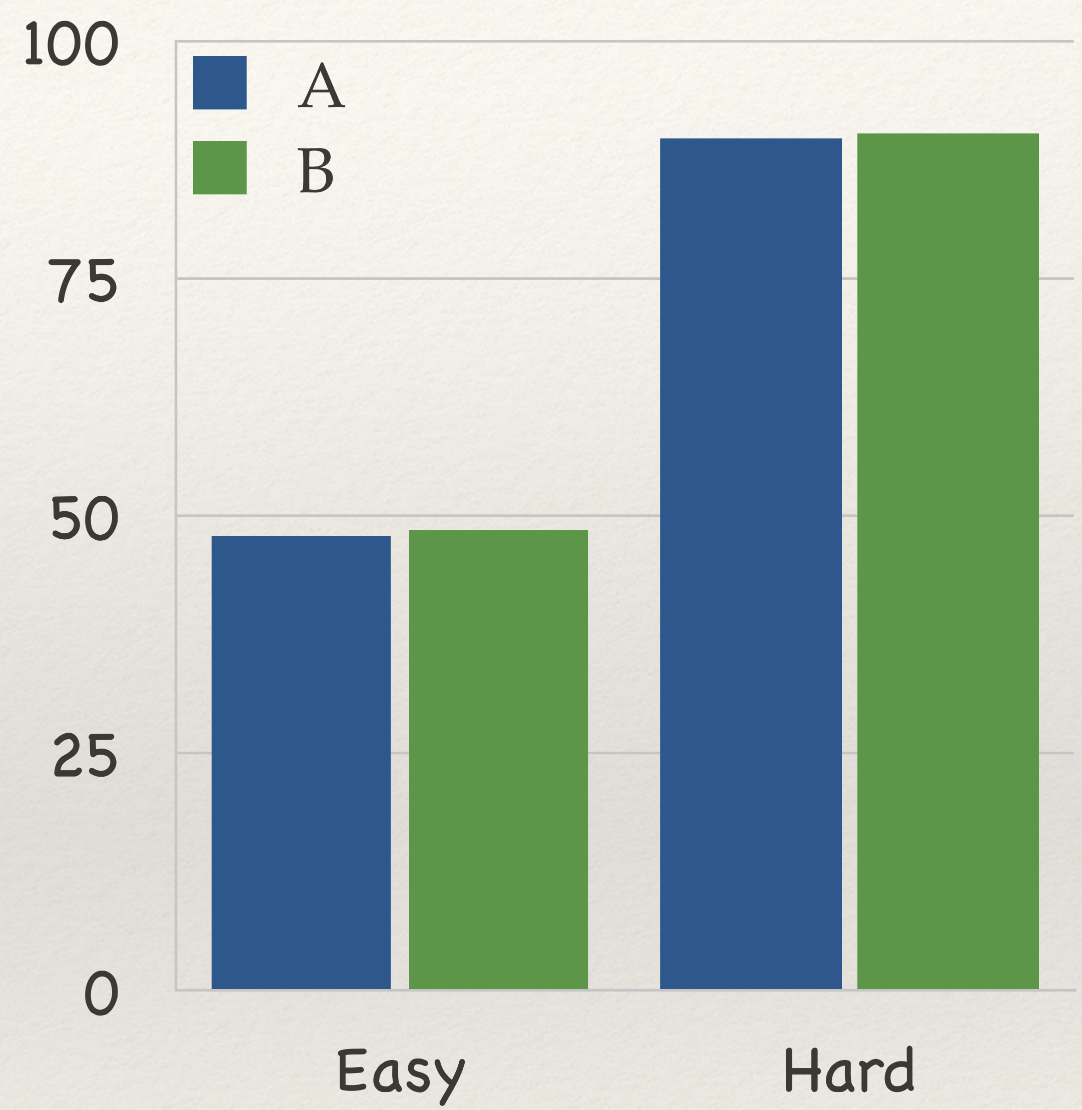
How to split results?

Aggregation on the whole testset too harsh: what about splitting into groups first?

Very natural criterion: split by difficulty of instances

Beware of biased selections :-(

```
N = 10000
names = list('AB')
df = pd.DataFrame(100*np.random.rand(N, 2), columns=names)
```




```
easy = df[df['A'] < 80]
hard = df[df['A'] >= 80]
```

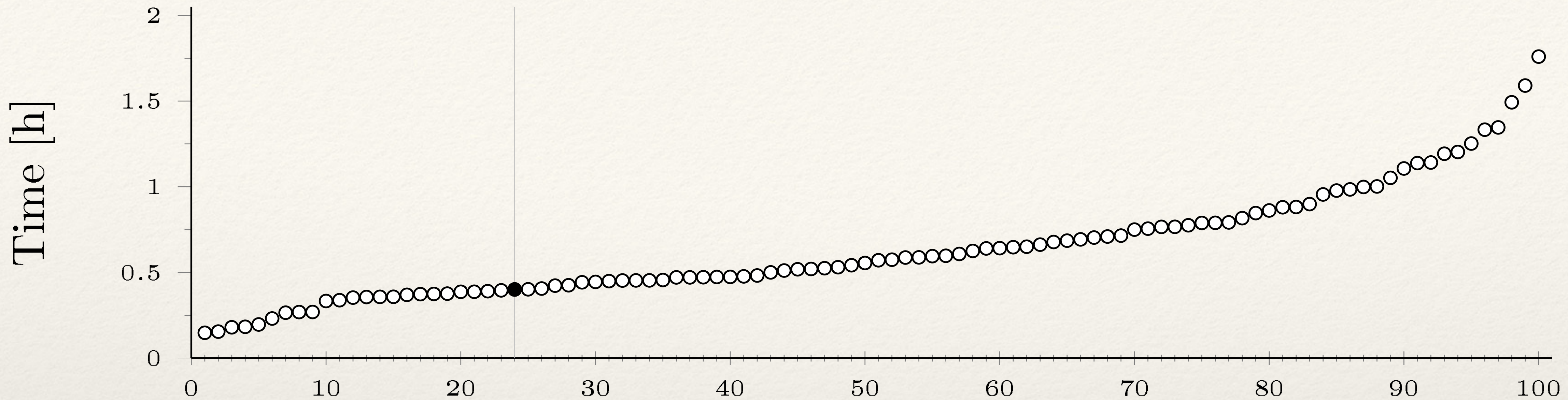
```
easy = df[df.min(axis=1) < 80]
hard = df[df.min(axis=1) >= 80]
```

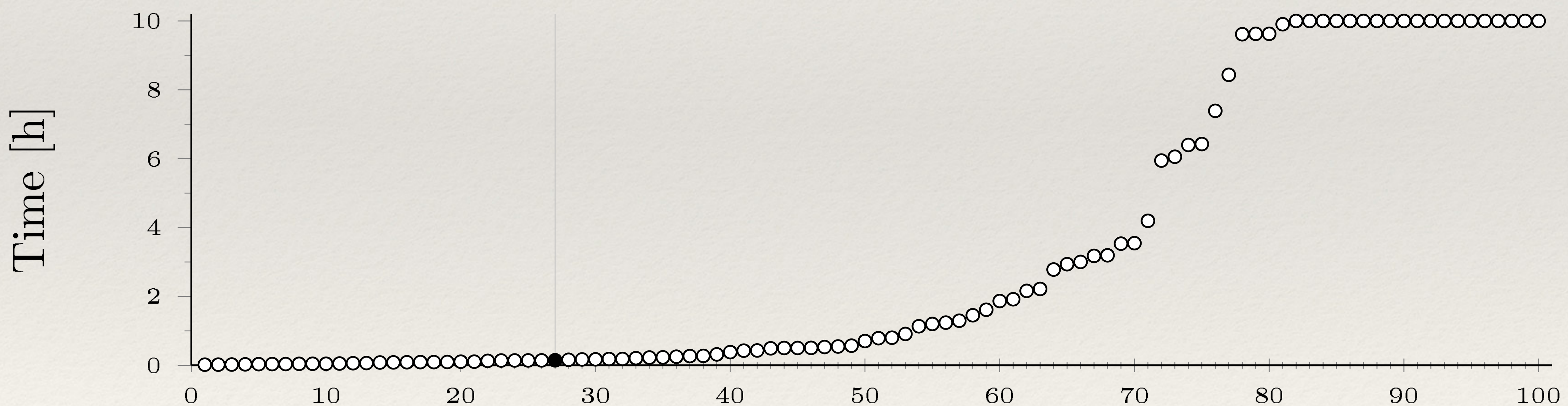
Performance Variability

The behaviour of the solver can be significantly influenced by seemingly neutral changes in the environment/input data

- random seed initialization
- order of constraints and variables in the problem

Runtime of the a given algorithm is basically a random variable even on what is mathematically the same instance!


Time [h]
2
1.5
1
0.5
0
0
10
20
30
40
50
60
70
80
90
100


Time [h]
10
8
6
4
2
0
0
10
20
30
40
50
60
70
80
90
100

Performance Variability

Where does performance variability comes from?

In a nutshell: imperfect tie-breaking

- Solvers take many decisions with limited knowledge

Can we fix it?   No

This is the price to pay for trying to be smart and efficient!

Statistical Tests to the Rescue

Mathematically sound approach to computationally evaluate the probability of two sequences of numbers coming from the same distribution (*null hypothesis*)

- numbers are the performance measures on the selected testset by the methods under comparison
- null hypothesis is that the methods are equivalent, and the difference we measured is just noise

Statistical Tests II

Many different statistical tests that differs on:

- Assumptions
- Power
- Paired vs unpaired samples

Statistical Tests III

McNemar

binary outcome (solved,success)

Wilcoxon

quantitative (runtime,nodes)

Either way: need a sufficiently large testset!!!

Remember that significant ≠ meaningful

Conclusions (so far)

1. Apply Scientific method
2. Avoid biases in experiment setup/data analysis
3. Deal with Performance Variability
4. Use Statistical tests

Writing (Good) Code

Why good code?

Correctness

Productivity

Sharing

"I like my code to be elegant and efficient. The logic should be straightforward to make it hard for bugs to hide, the dependencies minimal to ease maintenance, error handling complete according to an articulated strategy, and performance close to optimal so as not to tempt people to make the code messy with unprincipled optimizations. Clean code does one thing well."

–Bjarne Stroustrup (inventor of C++)

How to write good code?

Writing (good) code is *inherently* hard:

- Complex problems require complex code
- Requires (almost inhuman) attention to every details…
- …over (many) different levels of abstractions

How to write good code?

It does not come natural, but can be learnt!

- Best practices from software engineering
- Use the right tool for the job

This is very relevant even for *academic* code developed by a *single* person

Use right tool for the job

Pick the right language for job:

- Compiled language where performance is critical
- Scripting language for the rest

Use sufficiently powerful editor/IDE

Don't debug with print statements: learn to use a debugger

Don't do version control by hand

Invest time in learning your tools

Did you know that `vim` supports compiler assisted code completion?

Did you know that `gdb` can be scripted with Python?

Are you proficient with templates and the STL in C++?

Are you proficient with `numpy`, `pandas` and `matplotlib` in Python?

Don't reinvent the wheel!!!

Git Git Git

There is **no** real excuse for not using version control

- Even for academic code
- Even if developing alone

At a bare minimum:

- Need to sync code back and forth between laptop and workstation
- Ability to revert back bad changes
- And no, sending code by email or `scp` is not an alternative

Git: a game changer

Changes your approach to coding completely:

- Time machine for code
- Multiverse for code *(branches)*
- Key to reproducibility *(tags/commit hashes)*

Fundamental to share code with others!

No need to become a wizard: can go long way with just the basics!

Good code: correctness

Correctness trumps everything else…

…but how to make sure your code is correct?

Readability

Testing

Reviews

Good code: readability

Code is read way more often than it is written

Unreadable code is hard to:

- Understand (and thus argue correct)
- Maintain (modify, fix, improve, extend)

Please be gentle to the next developer (it could be you!)

"Debugging is twice as hard as writing the code in the first place.
Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."

–Brian Kernighan

Good code: readability

Proper naming (classes, functions, variables)

Use language features whenever appropriate

Split code into relatively short chunks

Single responsibility (*one thing well*) at a single abstraction level

Explicit preconditions, postconditions, invariants (`asserts`)

Proper naming

Intention revealing

```
int t; // elapsed time in hours
```

bad

```
int elapsedHours;
```

good

Unambiguous

```
void copy(char* a1, char* a2);
```

bad

```
void copy(char* source, char* destination);
```

good

Searchable

Use language from problem domain

Language features

Const / access modifiers

```
void copy(char* source, char* destination);
```

bad

```
void copy(const char* source, char* destination);
```

good

Named constants (instead of defines)

```
if (context == 128) {...}
```

really bad

```
#define RELAXATION_CONTEXT 128

if (context == RELAXATION_CONTEXT) {...}
```

bad

```
const int RELAXATION_CONTEXT = 128;

if (context == RELAXATION_CONTEXT) {... }
```

good

Functions

Small

Single responsibility (*one thing well*) at a single abstraction level

Avoid side effects

Explicit preconditions, postconditions, invariants (`asserts`)

How many times the same code needs to be written before we turn it into a function?

Once, just once!

Good code: comments

```
int colIndex; //< column index
```

useless

```
// indices 1-based
for (int i = 0; i < n; i++) {
   a[i] = compute_value(i);
}
```

wrong

```
// Autogenerated, do not edit. All changes will be undone
```

```
// http://tools.ietf.org/html/rfc4180 suggests that CSV lines
// should be terminated by CRLF, hence the \r\n.
csvStringBuilder.append("\r\n");
```

useful

“If the comment and code disagree, both are probably wrong.”

–Bjarne Stroustrup

“Don’t comment bad code—rewrite it!”

–Brian Kernighan & P.J. Plaugher

Good code: testing

There is no substitute for actual testing

Whenever the program misbehaves, write a test

But how do you know the program has a bug?

Well, you should write the tests **first**!

Testing must not be an afterthought!

Time consuming? Yes, but usually well worth it!

Good code: reviews

One of the most effective practices

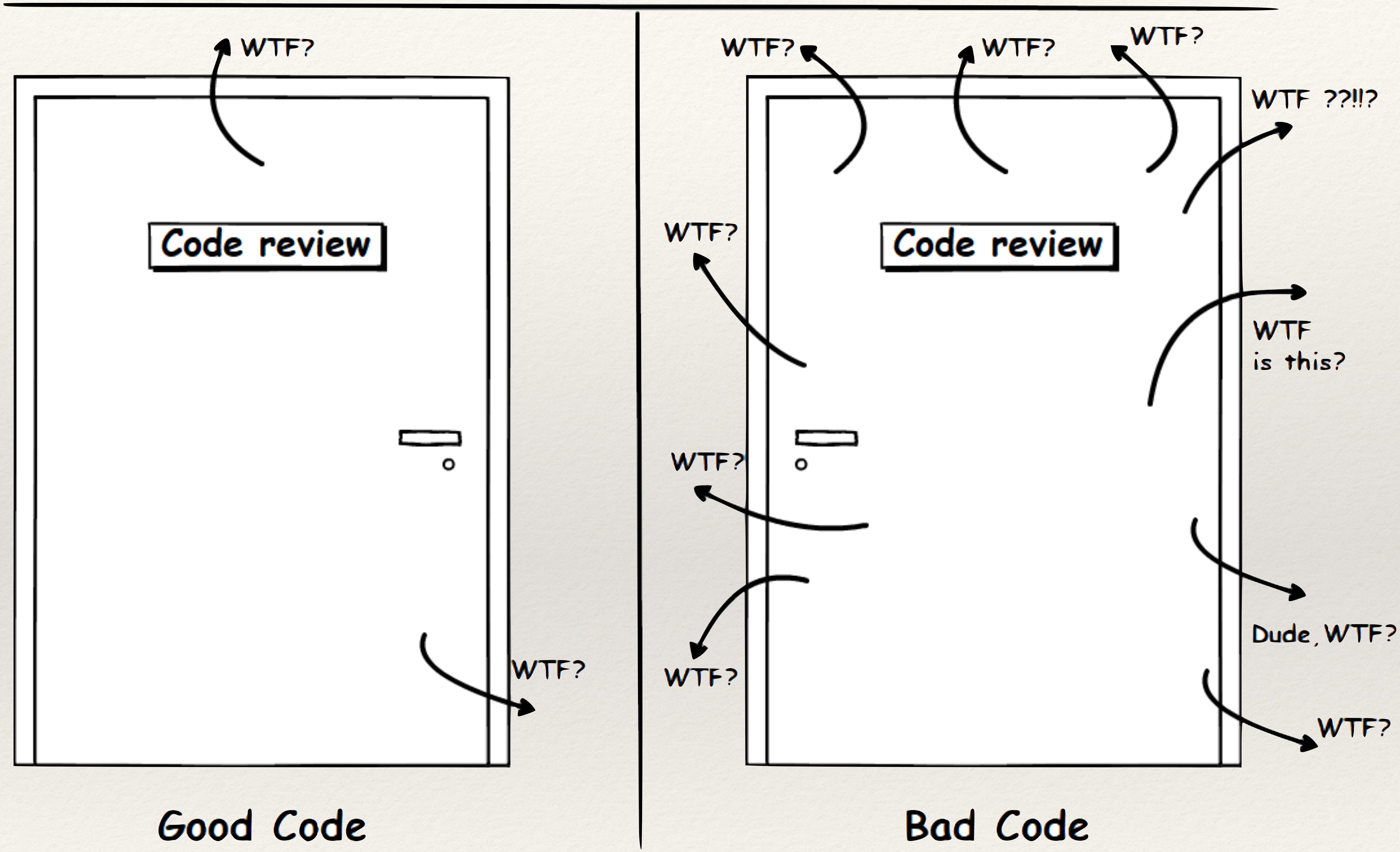
4 eyes are better than 2, but not just that

As authors of the code, we are the most biased in evaluating it

If developing alone, please ask your supervisor to review your code

After all, you would never put your name on a paper without checking its proofs…this is no different!

Code quality measurement:
WTFs per minute



Good code: performance

"Premature optimization is the root of all evil."

–Sir Tony Hoare

"The full version of the quote is "*We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.*" and I agree with this. Its usually not worth spending a lot of time micro-optimizing code before it's obvious where the performance bottlenecks are. But, conversely, when designing software at a system level, performance issues should always be considered from the beginning."

–Charles Cook

Good code: profiling

Worry about other issues (good algorithm design and good implementations of those algorithms) before worrying about counting cycles.

How do you spot inefficiencies?

- Do *not* trust your judgement
- Use a profiler (valgrind, perf, VTune)

Conclusions

Inherently hard (no easier than theoretical research)

Many challenges:

- general lack of formal training
- increasingly resource hungry
- some fields are more mature

Still great opportunities :-)

One more thing…

Presentation

Don’t spoil your hard work with a mediocre presentation!

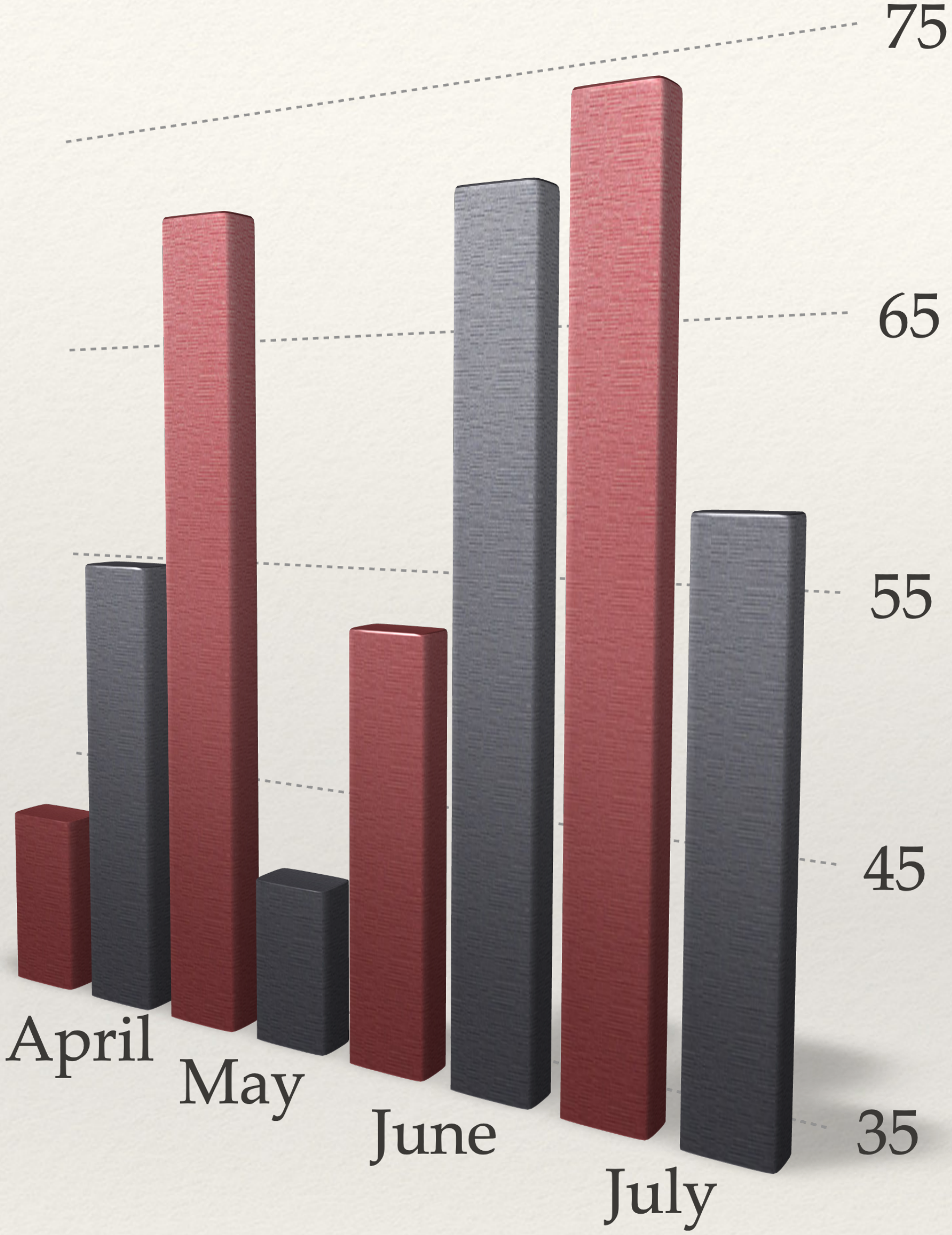
Invest time to learn properly display of information in:

- Tables
- Plots
- Slides

Remember: you are a professional communicator!

Instance	Time	Nodes
A	1.34	1
B	7.1	123
C	100.2	40000
D	3600	1453322
E	15.22	430
F	93.333	7023
G	211.3	50311

Instance	Time	Nodes
A	1.34	1
B	7.10	123
C	100.20	40,000
D	3,600.00	1,453,322
E	15.22	430
F	93.33	7,023
G	211.30	50,311


75
65
55
45
35
April
May
June
July

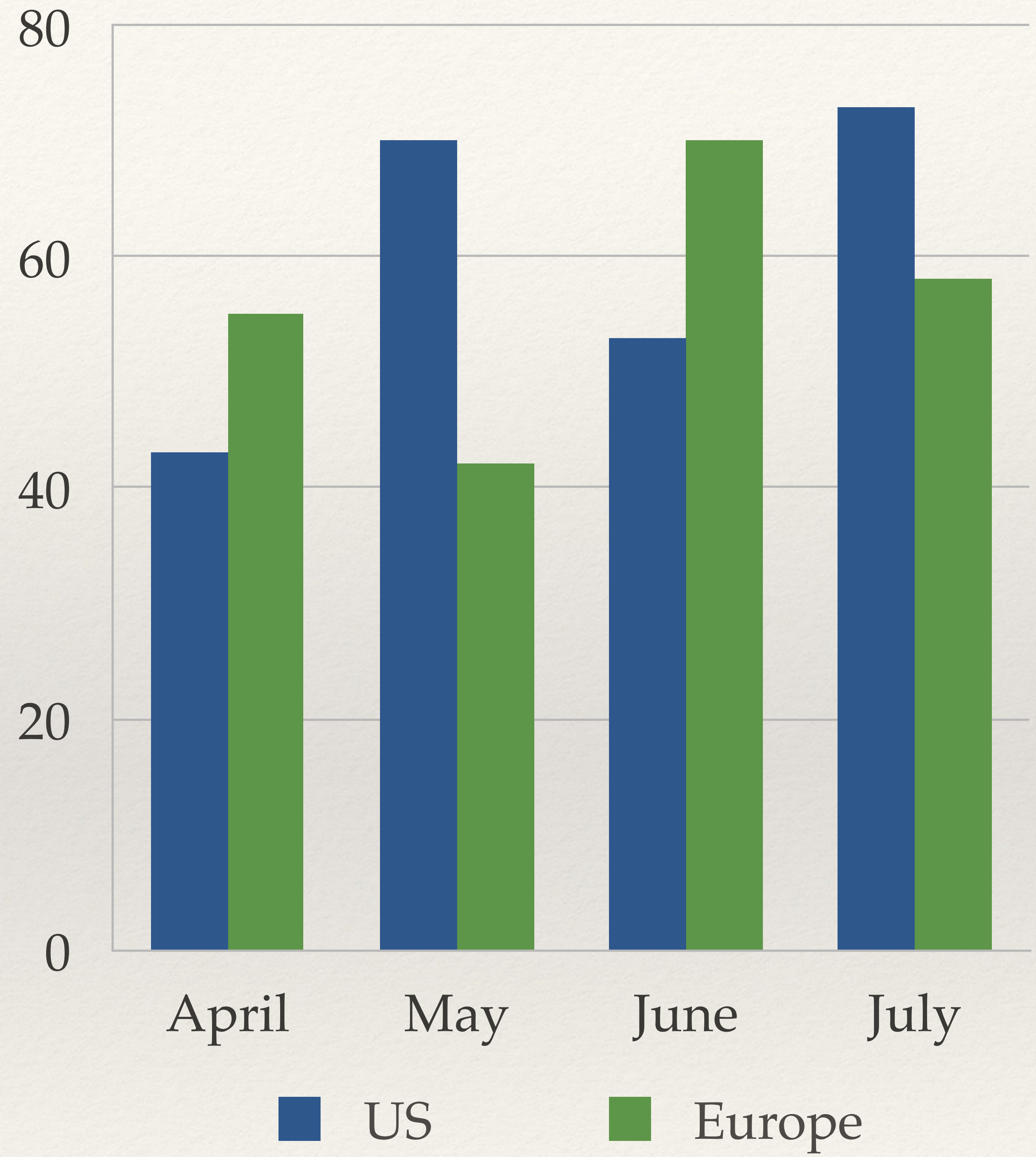

80
60
40
20
0
April
May
June
July
US
Europe

Table 2 **Comparison of Different Training Methods with Respect to Computing Time, Percentage WAD, and Validation Function (Cumulative Delay in Minutes), for Different Lines and Trade-Off α**

		Fat			Slim1			Slim2			LR		
α (%)	Line	Delay	WAD (%)	Time (s)	Delay	WAD (%)	Time (s)	Delay	WAD (%)	Time (s)	Delay	WAD (%)	Time (s)
0	BZVR	16,149	—	9,667	16,316	—	532	16,294	—	994	16,286	—	2.27
0	BrBO	12,156	—	384	12,238	—	128	12,214	—	173	12,216	—	0.49
0	MUVR	18,182	—	377	18,879	—	88	18,240	—	117	18,707	—	0.43
0	PDBO	3,141	—	257	3,144	—	52	3,139	—	63	3,137	—	0.25
Tot:		49,628	—	10,685	50,577	—	800	49,887	—	1,347	50,346	—	3.44
1	BZVR	14,399	16.4	10,265	15,325	45	549	14,787	17	1,087	14,662	18	2.13
1	BrBO	11,423	21.6	351	11,646	42	134	11,472	21	156	11,499	23	0.48
1	MUVR	17,808	12.9	391	18,721	37	96	17,903	12	120	18,386	8	0.48
1	PDBO	2,907	15.6	250	3,026	51	57	2,954	11	60	2,954	13	0.27
Tot:		46,537	66.5	11,257	48,718	175	836	47,116	61	1,423	47,501	62	3.36
5	BZVR	11,345	15.9	9,003	12,663	48	601	11,588	19	982	12,220	22	1.99
5	BrBO	9,782	18.9	357	11,000	50	146	9,842	22	164	10,021	23	0.51
5	MUVR	16,502	14.5	385	18,106	41	86	16,574	13	107	17,003	11	0.45
5	PDBO	2,412	14.7	223	2,610	44	49	2,508	20	57	2,521	19	0.28
Tot:		40,041	64	9,968	44,379	183	882	40,512	74	1,310	41,765	75	3.23
10	BZVR	9,142	21.4	9,650	10,862	50	596	9,469	24	979	10,532	33	2.01
10	BrBO	8,496	19.1	387	10,179	51	132	8,552	20	157	8,842	23	0.51
10	MUVR	15,153	14.7	343	17,163	49	84	15,315	15	114	15,710	13	0.43
10	PDBO	1,971	19.9	229	2,244	49	50	2,062	27	55	2,314	37	0.25
Tot:		34,762	75.1	10,609	40,448	199	862	35,398	86	1,305	37,398	106	3.2
20	BZVR	6,210	28.5	9,072	7,986	50	538	6,643	31	1,019	8,707	52	2.04
20	BrBO	6,664	22.1	375	8,672	53	127	6,763	23	153	7,410	30	0.52
20	MUVR	13,004	17.1	384	15,708	52	91	13,180	18	116	13,576	19	0.42
20	PDBO	1,357	28.4	230	1,653	49	55	1,486	34	60	1,736	53	0.28
Tot:		27,235	96.1	10,061	34,019	204	811	28,072	106	1,348	31,429	154	3.26
40	BZVR	3,389	35.4	10,486	4,707	50	578	3,931	37	998	5,241	51	2.31
40	BrBO	4,491	27.7	410	6,212	52	130	4,544	29	166	6,221	52	0.53
40	MUVR	10,289	21.8	376	13,613	52	95	10,592	25	108	11,479	34	0.45
40	PDBO	676	37.1	262	879	49	55	776	41	57	1,010	52	0.28
Tot:		18,845	122	11,534	25,411	203	858	19,843	132	1,329	23,951	189	3.57

References

- T. Achterberg and R. Wunderling. "*Mixed Integer Programming: Analyzing 12 Years of Progress*" In Facets of Combinatorial Optimization, 449–81, 2013.
- F. Brooks. "*The Mythical Man-Month*", 1975
- B. W. Kernighan and P. J. Plauger. "*The Elements of Programming Style*", 1978
- E. S. Raymond. "*The Art of UNIX Programming*", 2003
- M. Fowler et al. "*Refactoring: Improving the Design of Existing Code*", 1999
- R. C. Martin. "*Clean Code*", 2008
- K. Beck. "*Test-Driven Development by Example*", 2002
- R. Hyde. "The Fallacy of Premature Optimization", ACMUbiquity, 2009
- T. Preston-Werner. "*The Git Parable*", 2009 (on the web)
- G. Law. "*Give me 15 minutes & I'll change your view of GDB*", CppCon 2015 (on youtube)
- S. Few. "*Show Me the Numbers*", 2012
- C. O. Wilke. "*Fundamentals of Data Visualization*", 2019