Engineering the simplex method

0. Context

Linear programming

$$\begin{array}{lrcl} \min & c^T x & & \\ \text{s.t.} & A\,x & = & b \\ & x & \geq & 0 \end{array} \tag{LP}$$

Linear programming (with tolerances)

$$\begin{array}{llcr} \min & c^T x & & \\ \text{s.t.} & A\,x & = & b \\ & x & \geq & -\varepsilon \end{array} \qquad \text{(LP)}$$

How do we solve (LP)?

Simplex methods

combinatorial algorithm
(active set / basis)

exponentially many iterations
(worst case, as far as we know)

Interior-point methods

converging algorithm
(point on central path)

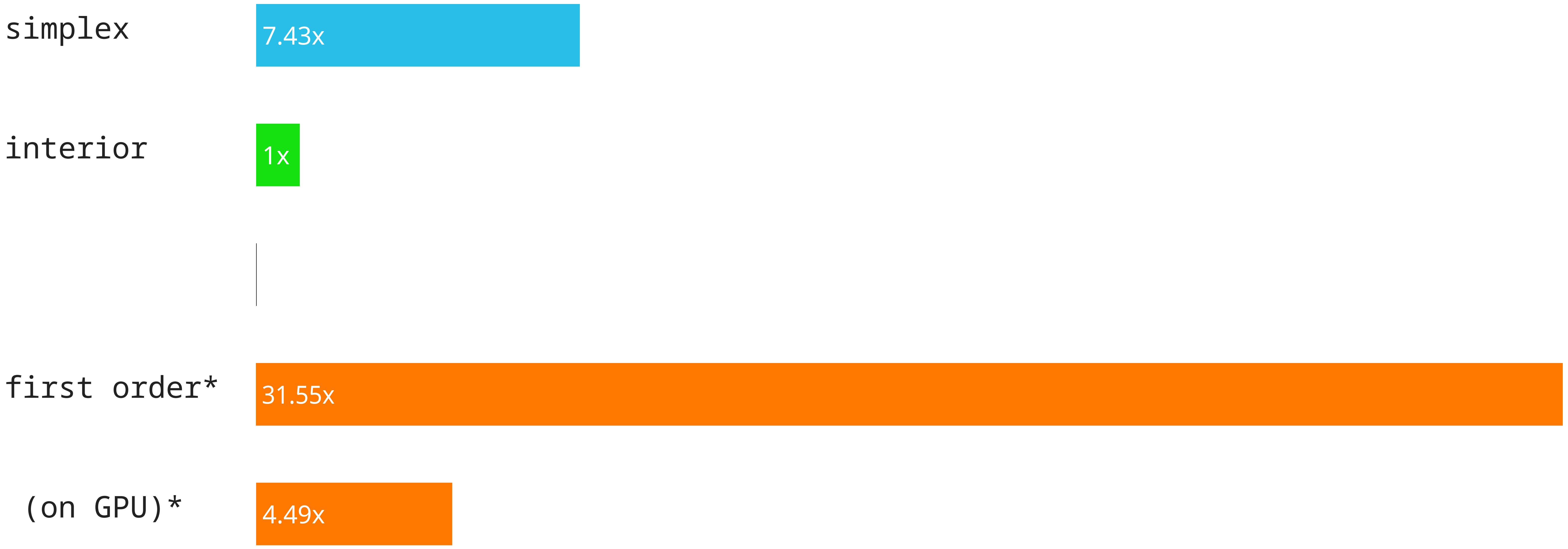
superlinear convergence
weakly polynomial

First-order methods

converging algorithm
(primal-dual iterate)

linear convergence
exponentially many iters

In practice



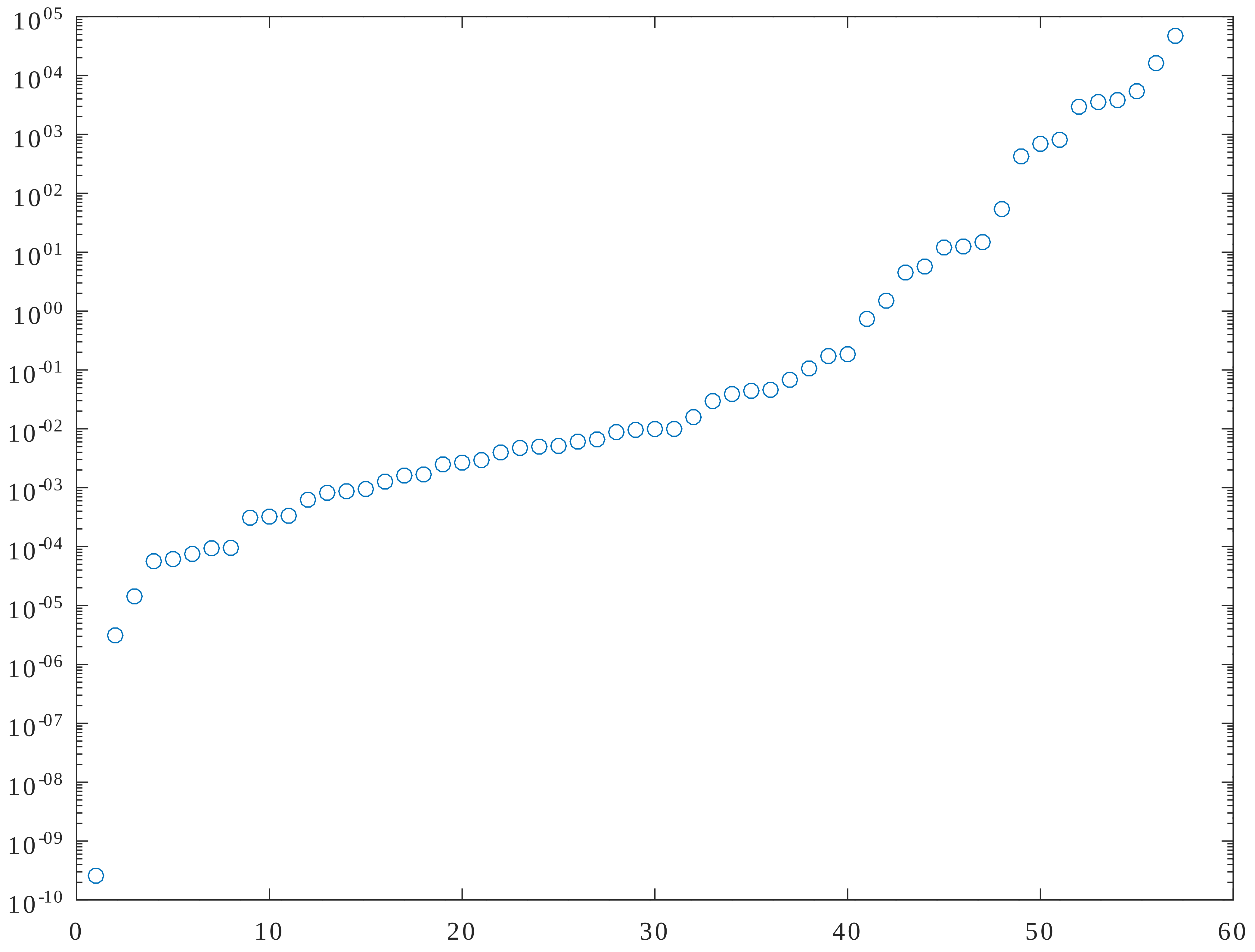
About first-order methods

FOM are more useful than numbers suggest:

- they are factorization-free
- when factorization exhausts memory, other methods will
  - crash, or
  - page to storage and become >100x slower

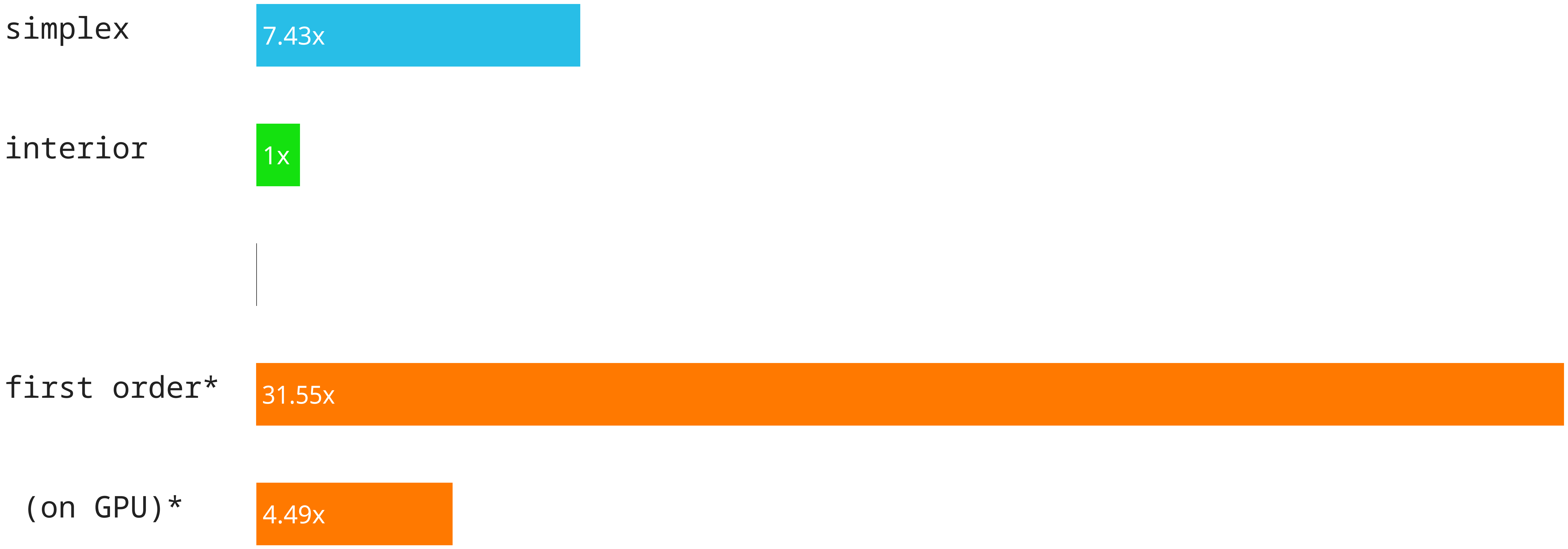
FOM are less useful than those numbers suggest:

- convergence is **much** slower than interior-point methods
- to compensate, they use different notions for “feasibility” and “optimality”



Absolute primal violation at termination with cuOPT H100
Mittelmann “lpfeas” benchmarks 2025-06-22

In practice (2)



Why the simplex method then?

- Accuracy
- Warm-start

Accuracy: 64-bit floating-point arithmetic

$$\pm\ 1.\mathtt{mmm...} \times 2^{\pm\mathtt{xxx..}}$$

±	1 sign bit: + or -
`mmm...`	52 “mantissa” bits
`±xxx..`	11 “exponent” bits (-1022..1023)
total	64 bits

- Hardware implements (8-, 16-, 32- and) 64-bit arithmetic natively.
- Software 128-bit arithmetic is ~60x slower than 64-bit arithmetic

Accuracy: how the simplex method helps

The simplex method provides a combinatorial data structure: **a basis**.

Even if the whole algorithm runs with inaccurate arithmetic,

at the end we can use the output basis to carefully recompute

- a basic solution, and
- reduced costs.

In practice, the output basis is optimal in almost all cases [Koch, 2003]

Warm-start

- interior-point methods cannot warm-start (they need a solution on the central path)

- warm-start enables
  - branch and bound (SCIP devs report 6 avg. iter per node)
  - column generation
  - cutting planes (incl. exponential formulations like TSP)

Note: best of both worlds

Given the output of an interior-point method

- we can identify a corresponding (but not necessarily optimal) basis
- and perform simplex-like pivots to get an optimal basis

This is “**crossover**” (strongly poly-time [Megiddo, 1991])

In practice (3)

1. The implementation

How much work is it?

component	lines of C code
sys, memory, etc.	23k
file format	8k
presolve	18k
linear algebra	14k
simplex logic	21k
total	**84k**

For reference,

solver	lines of code	remarks
HiGHS	163k, C++	(incl. MIP & IPM)
coin-clp	359k, C++	
SoPlex	54k, C++	(no presolve)
GLPK	122k, C	(incl. MIP)

Simplex performance

*Numbers extracted from (4 year old) Mittelmann simplex benchmarks, 2021-12-15

Ingredients

- Few iterations
- Fast iterations ← today
- Numerical stability
- Strong presolve

2. The simplex method is WEIRD

Experiment

Take an instance (`pds-40`) for which **computing the tableau pivot row** is the main bottleneck.

$$\bar{A}_i \quad = \quad e_i^T B^{-1} A \quad = \quad v^T A$$

$$\begin{array}{|l}
\bar{A}_i := 0 \\
\text{for } i : v_i \neq 0 \\
\qquad \text{for } j : A_{ij} \neq 0 \\
\qquad\qquad {\color{red} z} := v_i A_{ij} \\
\qquad\qquad \bar{A}_{ij} := \bar{A}_{ij} + {\color{red} \text{sign}(z)\sqrt{z^2}}
\end{array}$$

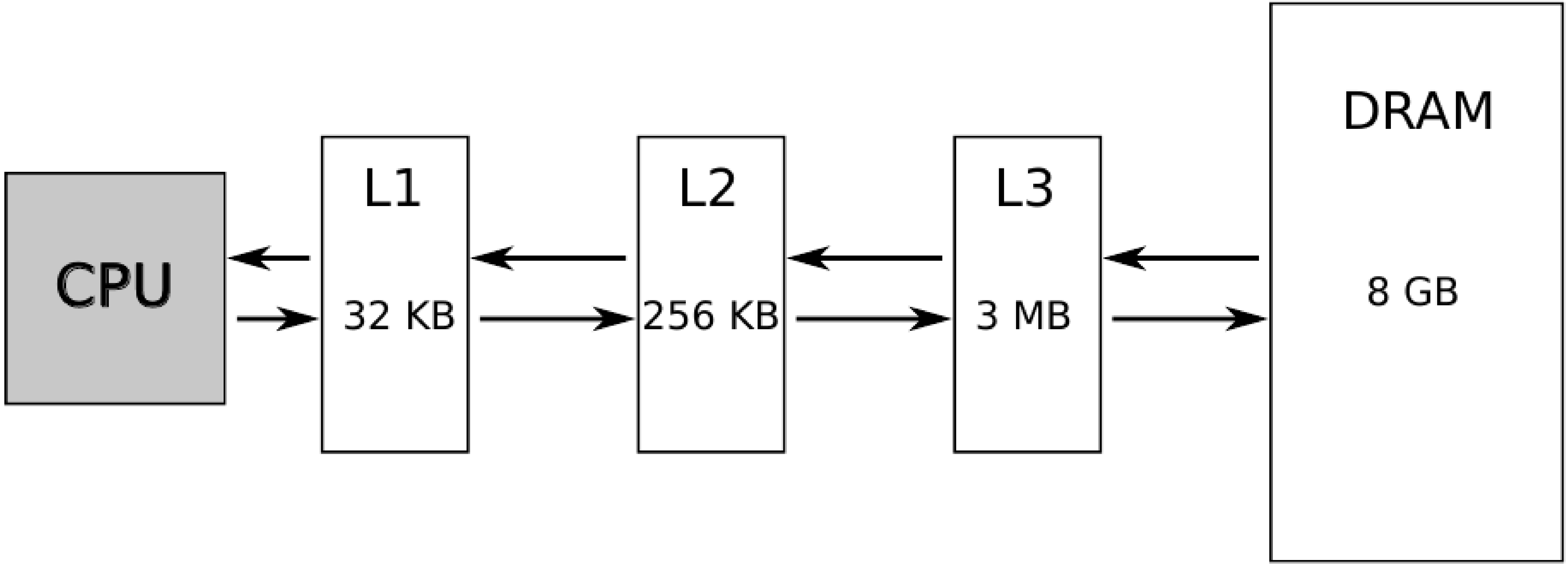
```
perf stat -d -d ./solve pds-40.mps.gz
```

```
 Performance counter stats for './solve pds-40.mps.gz':

          7,705.83 msec task-clock                #    1.012 CPUs utilized
               780      context-switches          #  101.222 /sec
                 8      cpu-migrations            #    1.038 /sec
            98,148      page-faults               #   12.737 K/sec
    23,550,072,100      cycles                    #    3.056 GHz
    27,369,781,800      instructions              #    1.16  insn per cycle
     4,639,369,202      branches                  #  602.060 M/sec
       197,304,728      branch-misses             #    4.25% of all branches
     6,203,991,788      L1-dcache-loads           #  805.104 M/sec
     1,244,920,759      L1-dcache-load-misses     #   20.07% of all L1-dcache accesses
       370,725,987      LLC-loads                 #   48.110 M/sec
       109,968,335      LLC-load-misses           #   29.66% of all LL-cache accesses
   <not supported>      L1-icache-loads
        30,602,848      L1-icache-load-misses
     6,094,141,741      dTLB-loads                #  790.848 M/sec
        13,956,784      dTLB-load-misses          #    0.23% of all dTLB cache accesses
            70,242      iTLB-loads                #    9.115 K/sec
           334,678      iTLB-load-misses          #  476.46% of all iTLB cache accesses
```

```
   23,550,072,100      cycles                    #    3.056 GHz
   27,369,781,800      instructions              #    1.16  insn per cycle
```

- 1.16 insn per cycle!!
- theoretical peak is 4
- CPU backend idle 71% of the time!



DRAM cache latency: >80 cycles

Compare an LP solve:

```
   23,550,072,100      cycles                    #    3.056 GHz
   27,369,781,800      instructions              #    1.16  insn per cycle
    4,639,369,202      branches                  #  602.060 M/sec
      197,304,728      branch-misses             #    4.25% of all branches
    6,203,991,788      L1-dcache-loads           #  805.104 M/sec
    1,244,920,759      L1-dcache-load-misses     #   20.07% of all L1-dcache accesses
      370,725,987      LLC-loads                 #   48.110 M/sec
      109,968,335      LLC-load-misses           #   29.66% of all LL-cache accesses
```

- 1.16 insn per cycle
- Memory accesses:

→	80 %	L1 cache
→	14 %	L2 cache
→	4 %	L3 cache
→	2 %	DRAM

1 / 50

With a heap sort:

```
    8,243,723,656      cycles                    #    3.077 GHz
   27,308,190,832      instructions              #    3.31  insn per cycle
    4,188,096,890      branches                  #    1.563 G/sec
       73,279,441      branch-misses             #    1.75% of all branches
    7,743,516,824      L1-dcache-loads           #    2.891 G/sec
          416,347      L1-dcache-load-misses     #    0.01% of all L1-dcache accesses
           22,855      LLC-loads                 #    8.532 K/sec
              812      LLC-load-misses           #    3.55% of all LL-cache accesses
```

- 3.31 insn per cycle
- Memory accesses:

→	99.99462 %	L1 cache
→	0.00508 %	L2 cache
→	0.00028 %	L3 cache
→	0.00001 %	DRAM

1 / 10 M

The simplex method is heavily bottlenecked on memory **latency**

(bandwidth is fine, <1GB/s in the above example, out of around 20 GB/s)

3. Implementation choices

Standard form

$$\begin{array}{lrcccc} \min & & c^T & x & & \\ \text{s.t.} & & A & x & = & 0 \\ & l & \leq & x & \leq & u \end{array} \qquad \text{(LP)}$$

where

$$c^T = [\ c_0^T \mid 0^T\ ], \qquad A = [\ A_0 \mid I\ ]$$

Implications:

- A is always full row rank
- phase-1 artificial variables are always available
- we can always “repair” a singular “basis” by inserting identity columns

Primal superbasics

Assuming $A = [\ B \mid N\ ]$,

$$Bx_B + Nx_N = 0$$

so

$$x_B = B^{-1}(0 - Nx_N).$$

For $j \in N$, we generally assume $\quad x_j = l_j \quad \text{or} \quad x_j = u_j.$

But x_j can also take any constant value $\tilde{x}_j$ with $l_j < \tilde{x}_j < u_j$.

It is then said to be primal superbasic.

Dual superbasics

Similarly we generally assume reduced costs $\bar{c}$ to be such that $\bar{c}_B = 0$:

$$\bar{c}^T = c^T - c_B^T B^{-1} A \qquad \text{i.e.,} \qquad \begin{cases} \bar{c}_B^T = c_B^T - c_B^T B^{-1} B = 0 \\ \bar{c}_N^T = c_N^T - c_B^T B^{-1} N \end{cases}$$

but we can compute instead:

$$\bar{c}^T = c^T - (c_B^T - \tilde{c}_B^T) B^{-1} A \qquad \text{i.e.,} \qquad \begin{cases} \bar{c}_B^T = \tilde{c}_B^T \\ \bar{c}_N^T = c_N^T - (c_B^T - \tilde{c}_B^T) B^{-1} N \end{cases}$$

Whenever $\tilde{c}_j \neq 0$ for some $j \in B$, we say that x_j is dual superbasic.

Why superbasics?

- Allowing superbasics generalizes (essentially for free) the primal and dual simplex methods.
- With superbasics, any basis can represent any feasible point.
- With a few additional types of pivot operations, the simplex method can remove those superbasics.

Useful for:

- repairing singular “bases” without losing (primal or dual) feasibility
- numerical difficulties in crossover
- postsolve