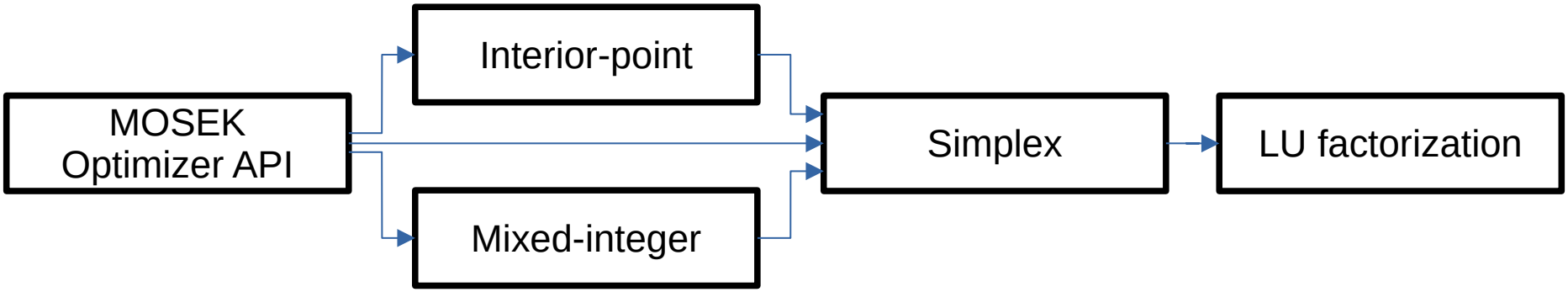
On reimplementing the simplex optimizers in MOSEK

SIAM Conference on Optimization (OP26)

Henrik A. Friberg

www.mosek.com

An essential piece



Why rewrite from scratch?

Legacy simplex code:

- >100 K lines of code (excluding presolve and LU modules).
- No documentation.
- No active development in 15 years.
- Maintenance is a burden.
- Hard to evolve.

Why rewrite from scratch?

Legacy simplex code:

- >100 K lines of code (excluding presolve and LU modules).
- No documentation.
- No active development in 15 years.
- Maintenance is a burden.
- Hard to evolve.
- **Performance is quite good.**
  - Back then we had no support of MIP.
  - Test set was significantly smaller.
  - Closer to the golden age of implementation (1990 - 2000).

A long undertaking

- 2017. Journal reading + experimentation in Julia.
- 2020. Started simplex module.
- 2021. Started dual simplex algorithm.
- 2023. Started LU module.
- 2024. Started primal simplex algorithm.

Credits:
Erling D. Andersen, Gianluca Frison and Henrik A. Friberg.

A long undertaking

- 2017. Journal reading + experimentation in Julia.
- 2020. Started simplex module.
- 2021. Started dual simplex algorithm.
- 2023. Started LU module.
- 2024. Started primal simplex algorithm.
- MOSEK 11 (2024).
  Default node solver = new dual simplex.
- MOSEK 12 (2026?).
  Default simplex solvers = new primal+dual simplex.

Credits:
Erling D. Andersen, Gianluca Frison and Henrik A. Friberg.

Main ingredients

- Sparse LU factorization (Suhl and Suhl 1990).
- Sparse LU update (Suhl and Suhl 1993).
- Sparse LU solves (Gilbert and Peierls 1988).
- Piecewise-linear functions in the objective (Fourer 1992).
- Steepest edge pricing weights (Forrest and Goldfarb 1992); exact dual and approximate primal (Świętanowski 1998).
- Boundflipping ratio test with Harris tolerance (Koberstein 2005).
- Perturbation (Koberstein 2005?).
- Hybrid full/partial pricing.
- A good foundation in computer science:
  - Floating-point numerics.
  - Sparse linear algebra.
  - Priority queue (to avoid full scan of pricing candidates).
  - Selection algorithm (to avoid full sort of breakpoint candidates).

Main ingredients

- Sparse LU factorization (Suhl and Suhl 1990).
- Sparse LU update (Suhl and Suhl 1993).
- Sparse LU solves (Gilbert and Peierls 1988).
- Piecewise-linear functions in the objective (Fourer 1992).
- Steepest edge pricing weights (Forrest and Goldfarb 1992); exact dual and approximate primal (Świętanowski 1998).
- Boundflipping ratio test with Harris tolerance (Koberstein 2005).
- Perturbation (Koberstein 2005?).
- Hybrid full/partial pricing.
- A good foundation in computer science:
  - Floating-point numerics.
  - Sparse linear algebra.
  - Priority queue (to avoid full scan of pricing candidates).
  - Selection algorithm (to avoid full sort of breakpoint candidates).

Main ingredients

- Sparse LU factorization (Suhl and Suhl 1990).
- Sparse LU update (Suhl and Suhl 1993).
- Sparse LU solves (Gilbert and Peierls 1988).
- Piecewise-linear functions in the objective (Fourer 1992).
- Steepest edge pricing weights (Forrest and Goldfarb 1992); exact dual and approximate primal (Świętanowski 1998).
- Boundflipping ratio test with Harris tolerance (Koberstein 2005).
- Perturbation (Koberstein 2005?).
- Hybrid full/partial pricing.
- A good foundation in computer science:
  - Floating-point numerics.
  - Sparse linear algebra.
  - Priority queue (to avoid full scan of pricing candidates).
  - Selection algorithm (to avoid full sort of breakpoint candidates).

Main ingredients

- Sparse LU factorization (Suhl and Suhl 1990).
- Sparse LU update (Suhl and Suhl 1993).
- Sparse LU solves (Gilbert and Peierls 1988).
- Piecewise-linear functions in the objective (Fourer 1992).
- Steepest edge pricing weights (Forrest and Goldfarb 1992); exact dual and approximate primal (Świętanowski 1998).
- Boundflipping ratio test with Harris tolerance (Koberstein 2005).
- Perturbation (Koberstein 2005?).
- Hybrid full/partial pricing.
- A good foundation in computer science:
  - Floating-point numerics.
  - Sparse linear algebra.
  - Priority queue (to avoid full scan of pricing candidates).
  - Selection algorithm (to avoid full sort of breakpoint candidates).

Main ingredients

- Sparse LU factorization (Suhl and Suhl 1990).
- Sparse LU update (Suhl and Suhl 1993).
- Sparse LU solves (Gilbert and Peierls 1988).
- Piecewise-linear functions in the objective (Fourer 1992).
- Steepest edge pricing weights (Forrest and Goldfarb 1992); exact dual and approximate primal (Świętanowski 1998).
- Boundflipping ratio test with Harris tolerance (Koberstein 2005).
- Perturbation (Koberstein 2005?).
- Hybrid full/partial pricing.
- A good foundation in computer science:
  - Floating-point numerics.
  - Sparse linear algebra.
  - Priority queue (to avoid full scan of pricing candidates).
  - Selection algorithm (to avoid full sort of breakpoint candidates).

Main ingredients

- Sparse LU factorization (Suhl and Suhl 1990).
- Sparse LU update (Suhl and Suhl 1993).
- Sparse LU solves (Gilbert and Peierls 1988).
- Piecewise-linear functions in the objective (Fourer 1992).
- Steepest edge pricing weights (Forrest and Goldfarb 1992); exact dual and approximate primal (Świętanowski 1998).
- Boundflipping ratio test with Harris tolerance (Koberstein 2005).
- Perturbation (Koberstein 2005?).
- Hybrid full/partial pricing.
- A good foundation in computer science:
  - Floating-point numerics.
  - Sparse linear algebra.
  - Priority queue (to avoid full scan of pricing candidates).
  - Selection algorithm (to avoid full sort of breakpoint candidates).

Main ingredients

- Sparse LU factorization (Suhl and Suhl 1990).
- Sparse LU update (Suhl and Suhl 1993).
- Sparse LU solves (Gilbert and Peierls 1988).
- Piecewise-linear functions in the objective (Fourer 1992).
- Steepest edge pricing weights (Forrest and Goldfarb 1992); exact dual and approximate primal (Świętanowski 1998).
- Boundflipping ratio test with Harris tolerance (Koberstein 2005).
- Perturbation (Koberstein 2005?).
- Hybrid full/partial pricing.
- A good foundation in computer science:
  - Floating-point numerics.
  - Sparse linear algebra.
  - Priority queue (to avoid full scan of pricing candidates).
  - Selection algorithm (to avoid full sort of breakpoint candidates).

Deterministic time

Simplex is mostly memory bound $\Longrightarrow$ strong correlation between **compute time** and **memory access counter**:

$$\text{time} \propto \text{ticks}.$$

Deterministic time

Simplex is mostly memory bound $\Longrightarrow$ strong correlation between **compute time** and **memory access counter**:

$$\text{time} \propto \text{ticks}.$$

```
res = 0.0
for k in 1:nz
    j = sub[k]
    res += x[j] * y[j]
end

ticks += nz * sizeof(Int)
ticks += 2 * nz * cachelinesize()
```

Deterministic time

A deterministic replacement of time:

- Log line frequency.
- Work limits (way better than max. iterations).

Enabling concurrent optimization:

- Instead of heuristic choise (primal or dual)
- Run both concurrently.
- Fastest method impose work limit on the other.
- MOSEK 12: If numthreads ≥ 2, free simplex will do this.

Predicting fastest execution path:

- LU refactorization frequency.
- Hybrid full/partial pricing strategy.

Deterministic time

A deterministic replacement of time:

- Log line frequency.
- Work limits (way better than max. iterations).

Enabling concurrent optimization:

- Instead of heuristic choise (primal or dual)
- Run both concurrently.
- Fastest method impose work limit on the other.
- MOSEK 12: If numthreads ≥ 2, free simplex will do this.

Predicting fastest execution path:

- LU refactorization frequency.
- Hybrid full/partial pricing strategy.

Deterministic time

A deterministic replacement of time:

- Log line frequency.
- Work limits (way better than max. iterations).

Enabling concurrent optimization:

- Instead of heuristic choise (primal or dual)
- Run both concurrently.
- Fastest method impose work limit on the other.
- MOSEK 12: If numthreads ≥ 2, free simplex will do this.

Predicting fastest execution path:

- LU refactorization frequency.
- Hybrid full/partial pricing strategy.

Polymorphic floating-point type

Simplex may encounter nasty numerics. Rather than fixing with duct tape and chewing gum, we use precision boosting.

$$\text{normal precision} \longrightarrow \text{extended precision.}$$

Polymorphic floating-point type

Simplex may encounter nasty numerics. Rather than fixing with duct tape and chewing gum, we use precision boosting.

normal precision $\longrightarrow$ extended precision.
(Zhang 2025)

```
using MultiFloats

x = Float64x1(2.0); sqrt(x)
# 1.4142135623730951

x = Float64x2(2.0); sqrt(x)
# 1.41421356237309504880168872420968
```

Polymorphic floating-point type

Simplex may encounter nasty numerics. Rather than fixing with duct tape and chewing gum, we use precision boosting.

$$\text{normal precision} \longrightarrow \text{extended precision.}$$
(Zhang 2025)

Characterization:

- Two 64-bit floats $\approx$ 128-bit float precision.
- Works on all relevant platforms.
- Only 2–4x slower.
- Same range as one 64-bit float ($\approx 10^{-324}$ to 10^{308}).

Polymorphic floating-point type

Examples of use:

- Simple and robust approach to solve numerically nasty problems (slowdown only affect nasty problems).
- Finding numerical bugs (same code, different precision).
- Work in progress:
  - Support for tiny tolerances $\Rightarrow$ high-quality solutions.

Emphasis on sparsity

Simplex needs many iterations. If you enter dense or numerically unstable regions you quickly loose to other methods such as the interior-point method.

As example, legacy starting point:

- Close to primal feasible.
- Close to dual feasible.
- Very sparse (basis matrix, search directions, etc.).
- Numerically stable.

Emphasis on sparsity

Simplex needs many iterations. If you enter dense or numerically unstable regions you quickly loose to other methods such as the interior-point method.

As example, new starting point:

- ~~Close to primal feasible~~.
- ~~Close to dual feasible~~.
- Very sparse (basis matrix, search directions, etc.).
- Numerically stable.

Emphasis on sparsity

Emphasis on sparsity

What you will see

- Starts far from target.
- Quickly catch up and take the lead.
- Returns more sparse and numerically stable solutions (if there are multiple optimal points).

Benchmarks

Primal simplex

		GMST ratio		Wins (5%)	
	#	legacy	new	legacy	new
All	2023	1.0	0.57	121	1425
Easy	962	1.0	0.89	63	440
Challenge	1061	1.0	0.38	58	985

Easy max time ≤ 0.1 seconds.

GMST geometric mean of shifted times ($+0.01\,\mathrm{s}$)

Benchmarks

Primal simplex

		GMST ratio		Wins (5%)	
	#	legacy	new	legacy	new
All	2023	1.0	0.57	121	1425
Easy	962	1.0	0.89	63	440
Challenge	1061	1.0	0.38	58	985

Easy max time ≤ 0.1 seconds.

GMST geometric mean of shifted times ($+0.01$ s)

Benchmarks

Dual simplex

		GMST ratio		Wins (5%)	
	#	legacy	new	legacy	new
All	2105	1.0	0.84	366	1189
Easy	1077	1.0	0.93	105	473
Challenge	1028	1.0	0.75	261	716

Easy max time ≤ 0.1 seconds.

GMST geometric mean of shifted times ($+0.01\,\mathrm{s}$)

Unfortunately not ready at time of benchmarking, but speed-up potential is seen by Virtual Best Solver,

$$\text{vbs} = \min(\text{primal}, \text{dual}).$$

		GMST ratio			Loss (5%)		
	#	primal	dual	vbs	primal	dual	vbs
All	2157	1.0	0.97	0.81	890	693	0
Easy	1201	1.0	0.99	0.92	377	274	0
Challenge	956	1.0	0.94	0.47	513	419	0

Unfortunately not ready at time of benchmarking, but speed-up potential is seen by Virtual Best Solver,

$$\text{vbs} = \min(\text{primal}, \text{dual}).$$

		GMST ratio			Loss (5%)		
	#	primal	dual	vbs	primal	dual	vbs
All	2157	1.0	0.97	0.81	890	693	0
Easy	1201	1.0	0.99	0.92	377	274	0
Challenge	956	1.0	0.94	0.47	513	419	0

Unfortunately not ready at time of benchmarking, but speed-up potential is seen by Virtual Best Solver,

$$\text{vbs} = \min(\text{primal}, \text{dual}).$$

		GMST ratio			Loss (5%)		
	#	primal	dual	vbs	primal	dual	vbs
All	2157	1.0	0.97	0.81	890	693	0
Easy	1201	1.0	0.99	0.92	377	274	0
Challenge	956	1.0	0.94	0.47	513	419	0

Summary

In MOSEK 12 we switch to using the new primal and dual simplex solvers as default.

- Support for deterministic times and work limits.
- Support for concurrent optimization.
- Improved robustness against numerically difficult problems.
- Improved performance.

References I

- Forrest, J. J. and D. Goldfarb (1992). "Steepest-edge simplex algorithms for linear programming". In: *Mathematical Programming* 57.1-3, pp. 341–374.
- Fourer, R. (1992). "A simplex algorithm for piecewise-linear programming III: Computational analysis and applications". In: *Mathematical Programming* 53.1-3, pp. 213–235.
- Gilbert, J. R. and T. Peierls (1988). "Sparse partial pivoting in time proportional to arithmetic operations". In: *SIAM Journal on Scientific and Statistical Computing* 9, pp. 862–874.
- Koberstein, A. (2005). "The Dual Simplex Method, Techniques for a fast and stable implementation". PhD thesis. Paderborn University.
- Suhl, L. M. and U. H. Suhl (1993). "A fast LU update for linear programming". In: *Annals of Operations Research* 43.1, pp. 33–47.
- Suhl, U. H. and L. M. Suhl (1990). "Computing Sparse LU Factorizations for Large-Scale Linear Programming Bases". In: *INFORMS Journal on Computing* 2.4, pp. 325–335.

References II

- Świętanowski, A. (1998). "A New Steepest Edge Approximation for the Simplex Method for Linear Programming". In: *Computational Optimization and Applications* 10, pp. 271–281.
- Zhang, D. K. (2025). "Fast Algorithms for High-Precision Floating-Point Arithmetic". PhD thesis. Stanford University.