Master in Statistics and Operative Research

Large Scale Optimization

Primal Affine Scaling Algorithm

Víctor Diví i Cuesta

Universitat Politècnica de Catalunya

Facultat de Matemàtiques i Estadística

Contents

1 Introduction

Interior-point methods are a family of algorithms used to solve linear and nonlinear problems, exploring the interior of the feasible region, instead of traversing its boundary (as Simplex does). The first of these methods, the Primal Affine Scaling Algorithm, was introduced by the Russian mathematician I. I. Dikin in 1967, although it remained largely unknown in western countries until Karmarkar's work in 1984. In this report, a Julia implementation of the Primal Affine Scaling Algorithm is presented (Section 2), and it is tested with several Netlib problems against other open-source solvers[1] (Section 3).

2 Implementation

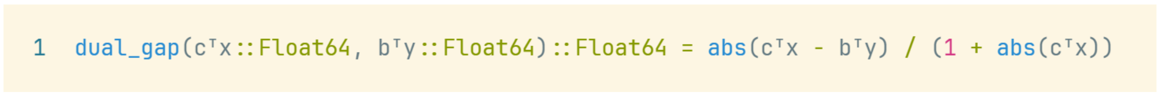
The Primal Affine Scaling (PAS) has been implemented in Julia[2], an open-source programming language designed to be very good at numerical and scientific computing. The complete code (split into several files) is delivered along this report, and the actual functions may differ from the captures[3] shown here only in formatting and logging to improve readability, but not in actual functionality.

2.1 Main Algorithm

The main body of the `primal_affine_scaling` function shown in Code 1. The function takes an optimization problem in standard equality form, an initial point (that must be feasible), and optionally a tolerance ϵ (which defaults to $1e^{-6}$) and a step reduction factor ρ (which defaults to 0.995) (lines 2-3).

First, it stores some constants for ease of use (lines 5-7) and makes sure that the coefficient matrix is full-rank (lines 10-18). If it is not, the problem is transformed so the coefficient matrix is full-rank and the function is called on the new problem. If the transformation fails, the function returns an error. How the problem is transformed and why it may fail is explained later in Subsection 2.4

Then, history arrays are created to keep the points, directions, and gaps of every iteration (lines 15-18) and variables are initialized. Line 27 solves the first system $ADA^T y = ADc$ by performing a Cholesky factorization of ADA^T and using Julia's $\backslash$ operator (which is used to solve linear systems of equations). The dual gap is computed using the `dual_gap` function, defined as follows:



```julia
dual_gap(cᵀx::Float64, bᵀy::Float64)::Float64 = abs(cᵀx - bᵀy) / (1 + abs(cᵀx))
```

[1]Spoiler Alert: it loses.

[2]https://julialang.org/

[3]Code is shown using screenshots rather than listings due to the presence of Unicode characters in the code that Latex complains about.

```julia
 1  function primal_affine_scaling(
 2      P::StandardProblem, x⁰::VF,
 3      ϵ::Float64=1e-6, ρ::Float64=0.995
 4  )::Result
 5      # Constants
 6      (; A, b, c) = P
 7      Aᵀ = transpose(A)
 8
 9      # Checks
10      if rank(A) ≠ size(A, 1)
11          @info "Problem is not full rank. Fixing."
12          frp = make_full_rank(P)
13          if (rank(frp.A) ≠ size(frp.A, 1))
14              @warn "Couldn't make problem full rank. Aborting."
15              return Result([x⁰], [], [], 0, stop_rank, false)
16          end
17          return primal_affine_scaling(frp, x⁰, ϵ, ρ)
18      end
19      @assert A*x⁰ ≈ b "Initial x⁰ point is not feasible."
20
21      @info "Starting Primal-Affine Scaling."
22
23      # History
24      x::VVF = [x⁰]
25      Δ::VVF = []
26      gap::VF = []
27
28      # Initializations
29      previous_cost = Inf
30      k = 1
31      D = Diagonal(x[k])^2
32      AD = A*D
33      ADAᵀ = AD*Aᵀ
34      ADc = AD*c
35      y = cholesky!(Symmetric(collect(ADAᵀ))) \ ADc
36      push!(gap, dual_gap(c'x[k], b'y))
37
38      reason = stop_gap_reached
39
```

(a) Initialization phase

```julia
39
40      while gap[k] > ϵ
41          if A*x[k] ≉ b || c'x[k] > previous_cost
42              @info "Feasibility lost or fluctuation detected at $k."
43              pop!(Δ)
44              pop!(x)
45              pop!(gap)
46              k -= 1
47
48              if ρ < 0.1
49                  @info "ρ too small ($ρ), stopping at last point."
50                  reason = stop_small_ρ
51                  break
52              end
53              ρ̂ = ρ > 0.6 ? 0.6 : 0.9ρ
54              @info "Repeating iteration with reduced ρ ($ρ → $ρ̂)."
55              ρ = ρ̂
56          end
57
58          previous_cost = c'x[k]
59
60          z = c - Aᵀ * y
61          push!(Δ, -D * z) # Δₖ = -Dz
62
63          @assert !all(Δ[k] .⩾ 0) "Unbounded Problem."
64
65          α = ρ * minimum((-x[k] ./ Δ[k])[Δ[k] .< 0])
66          push!(x, x[k] + α * Δ[k])
67
68          k += 1
69
70          D = Diagonal(x[k])^2
71          mul!(AD, A, D)
72          mul!(ADAᵀ, AD, Aᵀ)
73          mul!(ADc, AD, c)
74          try
75              y = cholesky!(Symmetric(collect(ADAᵀ))) \ ADc
76          catch
77              @info "Cholesky failed. Trying normal solve."
78              try
79                  y = solve(LinearProblem(collect(ADAᵀ), ADc)).u
80              catch
81                  @info "Normal solve failed. Stopping algorithm."
82                  reason = stop_cholesky_fail
83                  break
84              end
85          end
86
87          push!(gap, dual_gap(c'x[k], b'y))
88      end
89
90      return Result(x, Δ, gap, reason, isapprox(A*x[end], b, rtol=ϵ))
91  end
```

(b) Iterative phase

Code 1: Primal Affine Scaling function

Once initialized, we move on to the iterative phase of the algorithm. Until the dual gap is smaller than the provided tolerance ϵ, we perform the following steps:

1. Ensure that feasibility is not lost and the objective function has not increased (lines 40-58)

2. Compute new direction Δ, ensuring the problem is not unbounded (lines 60-63)

3. Compute step size α and new point x (lines 65-66)

4. Recompute y and the dual gap (lines 70-87)

In case that the feasibility is lost or that the objective function has increased from the previous iteration, it is assumed that the reason is a too-large step in the previous iteration. To try and fix it, we remove the last iteration and adjust the step size reduction factor ρ as follows:

- If ρ is smaller than 0.1, we stop the algorithm and return the previous point (last feasible point).
- If ρ is larger than 0.6 (first time), we set it to 0.6. This value is an arbitrarily chosen value lower than 2/3, to try to ensure the problem converges.
- Otherwise (subsequent times), we reduce it 10% ($\rho = 0.9\rho$).

To obtain the vector of dual variables y, we first try to use the same method as in the initialization, performing a Cholesky factorization of ADA^T. However, as the algorithm progresses, the factorization may fail due to the matrix not being positive-definite (due to numerical issues, since we make sure A is full-rank). When this happens, the $ADA^Ty = ADc$ is solved using the LinearSolve.jl library, which provides a unified interface for many linear solvers in Julia.

2.2 Initial Point: Big M

The `primal_affine_scaling` function introduced in the previous section requires an initial feasible point. However, such is not usually known or trivial to compute, so in case one is not provided, we run another version of the `primal_affine_scaling` function (Code 2), where we first apply the Big M method to expand the problem and obtain an initial feasible point, and then run the previous `primal_affine_scaling`, update the result feasibility if the additional variable is higher than the tolerance ϵ, and return the result without the additional variable:

```julia
function primal_affine_scaling(P::StandardProblem, ϵ::Float64=1e-6, ρ::Float64=0.995)::Result
    @info "No initial point provided. Expanding problem to obtain feasible initial point."
    PM, x̄⁰ = expand_problem(P)

    (; o, x, Δ, gap, reason, feasible) = primal_affine_scaling(PM, x̄⁰, ϵ, ρ)

    if o[end] > ϵ
        @warn "Problem is unfeasible: xₙ₊₁ > ϵ ⟹ $(o[end]) > $ϵ"
        feasible = false
    else
        @info "Problem is feasible: xₙ₊₁ < ϵ ⟹ $(o[end]) < $ϵ"
    end

    # Remove xₙ₊₁
    return Result([xᵢ[1:end-1] for xᵢ in x], [Δᵢ[1:end-1] for Δᵢ in Δ], gap, reason, feasible)
end
```

Code 2: Primal Affine Scaling without initial point

The actual expansion is done with the `expand_problem` function (Code 3), which takes a problem in standard equality form and returns the same problem expanded with the Big M method and the feasible point (vector of 1s). The M chosen is 100 times the highest absolute value in the objective cost vector c.

```julia
function expand_problem((; A, b, c)::StandardProblem)::Tuple{StandardProblem,VF}
    n = size(c, 1)

    r = b - A * ones(n)
    Ā = hcat(A, r)

    M = maximum(abs.(c)) * 100
    c̄ = push!(copy(c), M)

    return StandardProblem(Ā, b, c̄), ones(n+1)
end
```

Code 3: Big M Expansion

2.3 Standardization

All previous functions take as parameter a problem in standard equality form. However, many Netlib problems are not, and instead have lower and upper bounds for the variable vector x: $lo < x < hi$. To be able to handle these cases, a `standardization` function has been implemented to transform these problems into standard equality form (Code 4).

```julia
function standardize(
    (; A, b, c, lo, hi)::ExtendedProblem
)::Tuple{StandardProblem, Int, Int}
    original_constraints = size(A, 1)
    original_variables = size(A, 2)

    surplus_indices = findall(≠(0), lo)
    slack_indices = findall(≠(Inf), hi)

    if size(surplus_indices, 1) == 0 && size(slack_indices, 1) == 0
        @info "No standardization required"
        return StandardProblem(A, b, c),
                original_constraints,
                original_variables
    end

    num_constraints = original_constraints
    num_variables = original_variables

    (I, J, V) = findnz(A)
    b̂ = copy(b)

    # Add surplus variable and constraint for
    # every variable with a lower bound
    for var_index in surplus_indices
        num_constraints += 1
        num_variables += 1

        # Add constraint x - s = lo

        push!(I, num_constraints)
        push!(J, num_variables)
        push!(V, -1)

        push!(I, num_constraints)
        push!(J, var_index)
        push!(V, 1)

        push!(b̂, lo[var_index] ≠ Inf ? lo[var_index] : -10^100)
    end

    # Add slack variable and constraint
    # for every variable with an upper bound
    for var_index in slack_indices
        num_constraints += 1
        num_variables += 1

        # Add constraint x + s = hi

        push!(I, num_constraints)
        push!(J, num_variables)
        push!(V, 1)

        push!(I, num_constraints)
        push!(J, var_index)
        push!(V, 1)

        push!(b̂, hi[var_index])
    end

    return StandardProblem(
                sparse(I, J, V), b̂,
                vcat(c, zeros(num_variables - original_variables))
            ),
            original_constraints,
            original_variables
end
```

Code 4: Standardize function

This function adds a slack variable for every variable with a higher bound different than ∞, and a surplus variable for every one with a lower bound different than 0:

$$x > lo \rightarrow x - s^- = lo, x \geq 0, s^- \geq 0$$
$$x < hi \rightarrow x + s^+ = hi, x \geq 0, s^+ \geq 0$$

In case the lower bound is $-\infty$, -10^{20} is used instead.

2.4 Rankfulness

As mentioned in Subsection 2.1, the implemented `primal_affine_scaling` function requires the coefficient matrix A to be full-rank. To transform a problem without a full-rank A matrix into one, the `make_full_rank` function has been implemented (Code 5), which tries two things:

- Remove any constraint where all coefficients are 0[4].
- Remove any linearly dependent rows (constraints) from the matrix.

```julia
function make_full_rank(P::StandardProblem)::StandardProblem
    if rank(P.A) == size(P.A, 1)
        @debug "Problem is already full rank"
        return P
    end
    @debug "Removing empty constraints"
    P1 = remove_empty_constraints(P)
    if rank(P1.A) == size(P1.A, 1)
        return P1
    end
    @debug "Removing linearly dependent constraints using QR decomposition"
    P2 = remove_linear_dependencies(P1)

    if rank(P2.A) ≠ size(P2.A, 1)
        @warn "Couldn't make A full-rank"
    end

    return P2
end

function remove_empty_constraints((; A, b, c)::StandardProblem)::StandardProblem
    empty_rows = [i for i = 1:size(A, 1) if all(A[i, :] .== 0)]

    A_reduced = A[Not(empty_rows), :]
    b_reduced = b[Not(empty_rows)]

    return StandardProblem(A_reduced, b_reduced, c)
end

function remove_linear_dependencies((; A, b, c)::StandardProblem)::StandardProblem
    _, _, p = qr(Matrix(A'), ColumnNorm())
    independent_rows = sort(p[1:rank(A)])

    A_reduced = A[independent_rows, :]
    b_reduced = b[independent_rows]

    return StandardProblem(A_reduced, b_reduced, c)
end
```

Code 5: Full Ranker function

[4]This may seem absurd, but it happens in at least one Netlib problem.

To remove linearly dependent constraints, we apply a QR factorization with column pivoting on A^T:

$$A^T P = QR$$

Where P is the permutation matrix, Q is an orthogonal matrix, and R is an upper triangular matrix. Let $r = rank(A)$. Since QR decomposition with column pivoting is a Rank-Revealing QR factorization [Gu and Eisenstat(1996)], we get that the first r columns of the permutation matrix P indicate a set of linearly independent columns in A^T, which correspond to a set of linearly independent rows in A.

3 Results

3.1 Scaling Effect

The scaling applied when computing the direction of movement in each step is essential for the proper functioning of the implemented algorithm. Without scaling the problem to compute the movement, we may get too close to a boundary and the movement at each iteration become extremely small.

To showcase this behavior, Table 1 shows the evolution of the objective function and the dual gap for the **lp__afiro** problem with and without scaling. Performing the scaling, it only takes 19 iterations to reach a small enough dual gap, while when no scaling is performed (i.e. $D = I$), the movement rapidly becomes extremely small. The Diff column in the No Scaling section shows the difference in the objective value between consecutive iterations. We can see how the difference gets progressively smaller and smaller until being numerically equal to 0.0, meaning we are essentially not moving any more. In this case, the algorithm keeps iterating forever, stuck in that point.

3.2 Netlib Problems

The implemented algorithm has been tested with a total of 81 of Netlib's LP problems. In particular the first 81 problems after sorting them ascendingly by the size of their coefficient matrices after applying standardization. All problems have been run with a tolerance $\epsilon = 1e^{-6}$ and a step size reduction factor $\rho = 0.95$.

The same problems have been solved using two of the many solvers available in Julia: Tulip [Tanneau et al.(2021)Tanneau, Anjos and Lodi] and HiGHS [Huangfu and Hall(2018)]. Tulip is a pure Julia solver that implements the homogeneous primal-dual interior-point algorithm with multiple centrality corrections, while HiGHS is a Julia wrapper of the HiGHS solver written in C++ based on the dual revised simplex solver presented in [Huangfu and Hall(2018)].

3.2.1 Successfully solved problems

Of the 81 problems tested, the implementation presented in this work was able to successfully solve 59 of them, while Tulip and HiGHS were able to solve 81 (all) and

Iteration	**Scaling**		**No Scaling**		
	Objective	**Gap**	**Objective**	**Diff**	**Gap**
1	8.2	6.626188e-03	8.2	-	6.62618e-03
2	2.56809	1.249836e-02	2.56809	-5.63190e-00	1.33398e-02
3	-5.43410	5.447611e-02	2.28650	-2.81595e-01	1.36778e-02
4	-15.91889	2.859262e-01	2.27242	-1.40797e-02	1.36947e-02
5	-19.43239	5.932920e-01	2.27171	-7.03988e-04	1.36956e-02
6	-19.34493	9.720273e-01	2.27168	-3.51994e-05	1.36956e-02
7	-19.61344	9.221972e-01	2.27167	-1.75997e-06	1.36956e-02
8	-25.22546	9.522876e-01	2.27167	-8.79985e-08	1.36956e-02
9	-104.45138	5.528041e-01	2.27167	-4.39992e-09	1.36956e-02
10	-433.27614	7.830551e-02	2.27167	-2.19996e-10	1.36956e-02
11	-450.18206	3.280598e-02	2.27167	-1.09992e-11	1.36956e-02
12	-459.76638	1.117858e-02	2.27167	-5.50670e-13	1.36956e-02
13	-463.62882	2.520168e-03	2.27167	-2.75335e-14	1.36956e-02
14	-464.46115	6.482929e-04	2.27167	-8.88178e-16	1.36956e-02
15	-464.66422	1.987324e-04	2.27167	0.0	1.36956e-02
16	-464.73736	3.479627e-05	2.27167	0.0	1.36956e-02
17	-464.74716	1.339203e-05	2.27167	0.0	1.36956e-02
18	-464.75183	2.863069e-06	2.27167	0.0	1.36956e-02
19	-464.75278	7.904968e-07	2.27167	0.0	1.36956e-02

Table 1: Behavior of the implemented algorithm with and without affine scaling for the lp_afiro problem

64 respectively. Table 2 show a subset of 11 successfully solved problems showing the dimensions of the original problem (number of variables N, number of constraints M, total size and number of non-zero coefficients of A) and of the standardization problem (SEF indicates whether the original problem was in Standard Equality Form) and, for each of the three solvers, the iterations performed (Iters), the cost of the given solution (Cost), the execution time (Time), and the reason the algorithm stopped (Gap meaning that the duality gap became smaller than the tolerance ϵ, and ρ meaning that the step size reduction factor ρ became less than 0.1). The gaps between the implemented algorithm and the solvers' solutions is also shown (Gap). The selected problems are those with the smallest gaps, to try to make the presented implementation look as good as possible.

A csv file containing the full results of the performed experiments is delivered with the code.

As can be seen, the implemented algorithm obtains results very close to the other two algorithms, in general with a gap in the objective function of less than 0.001. For the smallest problems, our implementation requires more iterations than both solvers, for example, for problem **lp_afiro**, our implementation required 19 iterations, while Tulip and HiGHS required 8 and 6 respectively. However, as problems become larger the number of iterations required doesn't increase much for our implementation (and even less for Tulip), but it does significantly for HiGHS. For example, for the problem **lp__sctap1**, our implementation and Tulip requires a low number of iterations, 31 and 16 respectively, while HiGHS performs a total of 369 iterations. This is to be expected, since both HiGHS solves the problems using a simplex-based algorithm, while we and Tulip use an interior-point method.

Regarding execution time, our implementation gets completely destroyed by both Tulip and HiGHS. The results shown in Table 2 already show an important difference in execution times, but this only gets worse and worse as problems become larger. For example, for one of the largest problem solved, **lp__d2q06c**, which has 5831 variables and 2171 constraints, our algorithm takes more than 3 hours to solve, while both Tulip and HiGHS take less than a second. The root of this long execution times is the use of the Big M method for obtaining an initial feasible point, since it adds a completely dense column to the coefficient matrix A, which turns ADA^T into a completely dense matrix, making both its computation and the subsequent computation of $y = (ADA^T)^{-1}ADc$ very expensive and time-consuming.

3.2.2 Failed problems

Of the 22 problems unsolved, there have been multiple reasons for the failure of the presented implementations, such as:

- Algorithm finalizes properly, but the solution is later reported as unfeasible because the additional variable added with the Big M method is larger than the tolerance.
- The problem couldn't be made full-rank.

The first reason happens because in some cases, the chosen M ($100 * max(abs(c))$) is simply not large enough. This can be solved by simply picking a larger one. For example, when running the **lp__agg** problem, using the chosen M yields a result where the additional variable has value of 0.97438, which is obviously above the tolerance. However, if using instead $M = 1000000000 * max(abs(c))$, returns a feasible result with the additional variable having a value of $4.57626e^{-8}$, now below the accepted tolerance.

For some problems, the Julia function to compute the rank returns a value of 1 when it is not. In these cases, our algorithm tries to make the problem full-rank, but it fails, because the new problem is also reported to be rank-deficient by Julia. Initially, extremely low singular values were blamed for this misreporting. For example, the problem **lp__tuff** has several singular values well below the tolerance (see here). However, other problems suffered from the same issues while having no abnormally small singular value, such as **lp__vtp__base** (see here). After some testing, changing the tolerance of the rank function provides the proper rank of the matrix.

Problem	**Dimensions**				**Standardized**					**Primal Affine Scaling**			
	N	**M**	**Size**	**NZ**	**SEF**	**SN**	**SM**	**SSize**	**SNZ**	**Iters**	**Cost**	**Time (s)**	**Stop**
lp__afiro	51	27	1377	102	True	51	27	1377	102	19	-464.75279	2.12570e-03	Gap
lp__sc50	78	50	3900	160	True	78	50	3900	160	38	-64.57505	1.06776e-02	Gap
lp__sc50b	78	50	3900	148	True	78	50	3900	148	39	-69.99997	1.01279e-02	Gap
lp__blend	114	74	8436	522	True	114	74	8436	522	31	-30.81214	2.07330e-02	Gap
lp__lotfi	366	153	55998	1136	True	366	153	55998	1136	90	-25.26396	7.72051e-01	ρ
lp__scsd1	760	77	58520	2388	True	760	77	58520	2388	22	8.66668	8.00923e-02	Gap
lp__recipe	204	91	18564	687	False	320	207	66240	919	28	-266.61596	3.12957e-01	Gap
lp__e226	472	223	105256	2768	True	472	223	105256	2768	50	-18.75192	9.10986e-01	Gap
lp__scorpion	466	388	180808	1534	True	466	388	180808	1534	21	1.87813e+03	9.05481e-01	Gap
lp__sctap1	660	300	198000	1872	True	660	300	198000	1872	31	1.41225e+03	1.31558e-00	Gap
lp__scsd6	1350	147	198450	4316	True	1350	147	198450	4316	22	50.50002	4.77150e-01	Gap

Problem	**Tulip**				**HiGHS**			
	Iters	**Cost**	**Gap**	**Time (s)**	**Iters**	**Cost**	**Gap**	**Time (s)**
lp__afiro	8	-464.75314	3.52470e-04	6.68900e-04	6	-464.75314	3.52901e-04	5.24044e-04
lp__sc50	10	-64.57508	2.93217e-05	1.90890e-02	27	-64.57508	2.94672e-05	7.80106e-04
lp__sc50b	10	-70.00000	3.32987e-05	8.64100e-04	14	-70.00000	3.45132e-05	6.24895e-04
lp__blend	9	-30.81215	1.05980e-05	2.25904e-02	93	-30.81215	1.18564e-05	1.61791e-03
lp__lotfi	26	-25.26471	7.50580e-04	7.59140e-03	140	-25.26471	7.50682e-04	2.50983e-03
lp__scsd1	9	8.66667	8.54941e-06	4.15660e-03	86	8.66667	8.54879e-06	2.26521e-03
lp__recipe	9	-266.61600	3.52374e-05	9.48540e-03	19	-266.61600	3.52579e-05	1.28102e-03
lp__e226	18	-18.75193	7.58628e-06	1.12160e-02	364	-18.75193	8.44733e-06	7.12824e-03
lp__scorpion	13	1.87812e+03	3.86238e-04	5.33860e-03	89	1.87812e+03	3.56583e-04	3.14307e-03
lp__sctap1	16	1.41225e+03	7.99257e-04	9.98530e-03	369	1.41225e+03	7.69837e-04	6.05702e-03
lp__scsd6	11	50.50000	2.39481e-05	1.33236e-02	270	50.50000	2.32042e-05	6.85692e-03

Table 2: Characteristics and Experimental Results of 11 Netlib Problems

4 Conclusions

In this assignment we have provided a Julia implementation of the Primal Affine Scaling algorithm, analyzed the importance of the affine scaling in the computation of a good movement, and identified some shortcomings. Although solutions for some of these issues have been implemented, some have been left (such as the wrong reporting of rank-deficient matrices).

Moreover we have presented experimental results of our implementation using several problems from the Netlib LP collection, and compared the performance against two open-source solvers implementing different algorithms (both simplex-based and interior-point). Our implementation is capable of solving the majority of the used problems, and can keep up with the other solvers (regarding number of iterations). However, the method to obtain a first feasible initial point greatly slowed down the algorithm during the experiments.

References

[Gu and Eisenstat(1996)] Gu, M., Eisenstat, S.C., 1996. Efficient algorithms for computing a strong rank-revealing qr factorization. SIAM Journal on Scientific Computing 17, 848–869. URL: `https://doi.org/10.1137/0917055`, doi:`10.1137/0917055`, `arXiv:https://doi.org/10.1137/0917055`.

[Huangfu and Hall(2018)] Huangfu, Q., Hall, J.A.J., 2018. Parallelizing the dual revised simplex method. Mathematical Programming Computation 10, 119–142. URL: `https://doi.org/10.1007/s12532-017-0130-5`, doi:`10.1007/s12532-017-0130-5`.

[Tanneau et al.(2021)Tanneau, Anjos and Lodi] Tanneau, M., Anjos, M.F., Lodi, A., 2021. Design and implementation of a modular interior-point solver for linear optimization. Mathematical Programming Computation URL: `https://doi.org/10.1007/s12532-020-00200-8`, doi:`10.1007/s12532-020-00200-8`.