Master in Statistics and Operative Research

Large Scale Optimization

Primal-Dual Path-Following Interior Point Method

Víctor Diví i Cuesta

Universitat Politècnica de Catalunya

Facultat de Matemàtiques i Estadística

Contents

List of Tables

List of Codes

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). One of the most efficient and commonly used is the Primal-Dual Path-Following Algorithm. In this report, a Julia implementation is presented with several alternative steps (Section 2). These steps are then evaluated with several Netlib problems against each other and other open-source solvers[1] (Section 3).

2. Implementation

The Primal-Dual Path-Following (PDPF) Algorithm has been implemented in Julia[2]. The complete code (split into several files) is delivered along this report. The actual functions may be slightly different from the codes[3] shown here, but only in formatting or minor changes to improve readability, not in functionality.

2.1. Main Algorithm

The main `primal_dual_path_following` function is shown in Code 1. It takes an optimization in standard equality form, an initial point for the algorithm (divided into the three components x^0, λ^0, s^0), and, optionally, a `Step` object indicating the kind of step that will be performed (a Newton step by default), the feasibility and optimality tolerances ε^f and ε^o (both of which default to $1e-8$) and the step reduction factor ρ (which defaults to 0.99).

The function starts making sure the given coefficient matrix is full-rank. If it is not, we try to transform the problem so it is[4], and call again the function with the now-full-rank problem. If the transformation fails, we abort.

We then create the variables that in which we will store the points, directions, residuals, and gaps of every iteration, and initialize them. Once inside the loop, we perform the following steps:

- Compute the direction (lines 32-33): this computation is not made here, but rather delegated to the `compute_direction` function, which, depending on the value of `step` will perform different movements. This method and all the step possibilities are explained in detail in Section 2.2 below.
- Compute the step sizes and new point (lines 35-40): the primal and dual step sizes (α^p and α^d) are computed and used to compute the next point.
- Calculate and check gaps (lines 44-53): the residuals and gaps are computed and checked against the given tolerances. If all the gaps are lower than their respective tolerances, we stop. Otherwise, we repeat the steps again. The primal and dual gaps are computed using the `gap` function defined right after the main function.

```
1 function primal_dual_path_following(
2   P::StandardProblem, x⁰::VF, λ⁰::VF, s⁰::VF,
3   step::Step=NewtonStep(), ϵᶠ::Float64=1e-8, ϵᵒ::Float64=1e-8, ρ::Float64=0.99
4 )::Result
5     # Constants
6     (; A, b, c) = P
7
8     # Checks
```

[1] Spoiler Alert: it doesn't lose :).

[2] https://julialang.org

[3] This time it's actual code rather than screenshots because I ditched Latex to try Typst.

[4] The transformation is exactly the same as in the Primal Affine Scaling report, and is therefore not explained in more detail here.

```
    if rank(A, tol=ϵᶠ) != size(A, 1)
        @info "Problem is not full rank. Fixing."
        frp = make_full_rank(P)
        if (rank(frp.A, tol=ϵᶠ) != size(frp.A, 1))
            @warn "Couldn't make problem full rank. Aborting."
            return Result([x⁰], [λ⁰], [s⁰], [], [], [], [], [], [], [], [])
        end
        @info "Problem successfully full-ranked. Discarding initial point."
        return primal_dual_path_following(frp, step, ϵᶠ, ϵᵒ, ρ)
    end

    # History
    x::VVF = [x⁰]; λ::VVF = [λ⁰]; s::VVF = [s⁰]
    Δx::VVF = []; Δλ::VVF = []; Δs::VVF = []

    rᶜ::VVF = [[]], rᵇ::VVF = [[]], μ::VF = [0]
    primal_gap::VF = [0]; dual_gap::VF = [0]

    # Initializations

    k = 1

    while true
        Δxᵏ, Δλᵏ, Δsᵏ, μᵏ = compute_direction(step, P, x[k], λ[k], s[k])
        push!(Δx, Δxᵏ); push!(Δλ, Δλᵏ); push!(Δs, Δsᵏ)

        αᵖ = min(1, ρ * minimum((-x[k]./Δxᵏ)[Δxᵏ .< 0]; init=Inf))
        αˢ = min(1, ρ * minimum((-s[k]./Δsᵏ)[Δsᵏ .< 0]; init=Inf))

        push!(x, x[k] + αᵖ * Δxᵏ);
        push!(λ, λ[k] + αˢ * Δλᵏ);
        push!(s, s[k] + αˢ * Δsᵏ)

        k += 1

        push!(rᶜ, A' * λ[k] + s[k] - c)
        push!(rᵇ, A * x[k] - b)

        push!(dual_gap, gap(c, rᶜ[k]))
        push!(primal_gap, gap(b, rᵇ[k]))
        push!(μ, μᵏ)

        if dual_gap[k] <= ϵᶠ && primal_gap[k] <= ϵᶠ && μ[k] <= ϵᵒ
            break
        end
    end

    return Result(x, λ, s, Δx, Δλ, Δs, rᶜ, rᵇ, μ, primal_gap, dual_gap)
end

gap(v::VF, r::VF) = norm(r) / (1 + norm(v))
```

Code 1: Implementation of the `primal_dual_path_following` function in Julia.

2.2. Step computation

As mentioned above, the computation of the step at each iteration is performed by the `compute_step` function. The `step` parameter indicates exactly which step will be computed, and how: both the Newton Step and the Mehrothra (Corrector-Predictor) Step are implemented with the three methods to solve the linear systems of equations (base system, augmented system and normal equations).

This approach of moving the computation to dedicated functions has its advantages, namely the ability of seamlessly swap between step types and computations without changing the base code, but it also has its drawbacks. In this case, the methods recreate many matrices at every

iterations that could be either completely reused or at least overwritten to avoid reallocating memory. Since the main goal of this implementations is not pure performance,

Code 2 shows the definition of the `Step` type. Only two kind exist: `NewtonStep` and `MehrotraStep`. Both hold a `system` member indicating which system of equations will be used to compute it, which can be any of the three mentioned before.

A comparison between the performances is presented in Section 3.1.

```julia
abstract type Step end

@enum System begin
    base
    augmented
    normal
end

struct NewtonStep <: Step
    σ::Float64
    system::System
end
NewtonStep(σ::Float64) = NewtonStep(σ, normal)
NewtonStep() = NewtonStep(0.1, normal)

struct MehrotraStep <: Step
    system::System
end
MehrotraStep() = MehrotraStep(normal)
```

Code 2: Definition and subtypes of `Step` in Julia.

2.2.1. Newton Step

Codes 3, 4, and 5 show the implementations of the Newton Step solving the Base, Augmented and Normal systems respectively. The Base and Augmented systems have been solved using the LinearSolve.jl package, which provides a unified interface for many solvers and can choose the best depending on the system characteristics.

```julia
function newton_base_system(
  step::NewtonStep, (; A, b, c)::StandardProblem, x::VF, λ::VF, s::VF
)::Tuple{VF,VF,VF,Float64}
    # Vars
    (m, n) = size(A); (; σ) = step
    X = spdiagm(x); S = spdiagm(s)
    e = ones(n); I = spdiagm(e)
    Zᵐ = spzeros(m, m); Zⁿ = spzeros(n, n)
    Zᵐⁿ = spzeros(m, n); Zⁿᵐ = spzeros(n, m)
    μ = x's / n

    # System
    F = dropzeros!([
        Zⁿ A'  I;
        A  Zᵐ  Zᵐⁿ;
        S  Zⁿᵐ X
    ])

    rᶜ = A' * λ + s - c
    rᵇ = A * x - b
    rˣˢ = X * S * e - σ * μ * e
    R = [-rᶜ; -rᵇ; -rˣˢ]

    Δ = solve(LinearProblem(F, R)).u
    return Δ[1:n], Δ[n+1:n+m], Δ[n+m+1:end], μ
```

```
26 end
```

Code 3: Implementation of the Newton Step with Base System.

In the function solving the Augmented system (Code 4), note that Θ has been computed as $-(XS^{-1})^{-1}$ instead of the normal XS^{-1} to make the system matrix prettier, since it has no other use.

```
1  function newton_augmented_system(
2    step::NewtonStep, (; A, b, c)::StandardProblem, x::VF, λ::VF, s::VF)
3    ::Tuple{VF,VF,VF,Float64}
4      # Vars
5      (m, n) = size(A); (; σ) = step
6      X = spdiagm(x); X⁻¹ = spdiagm(1 ./ x); S = spdiagm(s);
7      Θ = -spdiagm(s ./ x) # -(X*S⁻¹)⁻¹,  Makes matrix below prettier :)
8      e = ones(n); Zᵐ = spzeros(m, m)
9      μ = x's / n
10
11     # System
12     F = dropzeros!([
13         Θ A';
14         A Zᵐ
15     ])
16
17     rᶜ = A' * λ + s - c
18     rᵇ = A * x - b
19     rˣˢ = X * S * e - σ * μ * e
20     R = [-rᶜ + X⁻¹ * rˣˢ; -rᵇ]
21
22     Δ = solve(LinearProblem(F, R)).u
23     Δx = Δ[1:n]
24     Δλ = Δ[n+1:n+m]
25     Δs = -X⁻¹ * (rˣˢ + S * Δx)
26
27     return Δx, Δλ, Δs, μ
28 end
```

Code 4: Implementation of the Newton Step with Augmented System.

The Normal system has been solved directly using a Cholesky factorization. Sometimes, due to numerical issues, the factorization fails. In these cases, we perform the factorization again but adding a small perturbation in the diagonal (lines 13-16 in Code 5). Note that here, unlike in the Augmented system, Θ is computed normally as XS^{-1}.

```
1  function newton_normal_system(
2    step::NewtonStep, (; A, b, c)::StandardProblem, x::VF, λ::VF, s::VF
3  )::Tuple{VF,VF,VF,Float64}
4      # Vars
5      (m, n) = size(A); (; σ) = step
6      X = spdiagm(x); X⁻¹ = spdiagm(1./x)
7      S = spdiagm(s); S⁻¹ = spdiagm(1./s)
8      Θ = X * S⁻¹; e = ones(n)
9      μ = x's / n
10
11     # System
12     AΘAᵀ = cholesky!(Symmetric(A * Θ * A'); check=false)
13     if !issuccess(AΘAᵀ)
14         @warn "Cholesky factorization failed. Adding perturbation"
15         AΘAᵀ = cholesky!(Symmetric(A * Θ * A' + 1e-6 * I); check=false)
16     end
17
18     rᶜ = A' * λ + s - c
```

```
    rᵇ = A * x - b
    rˣˢ = X * S * e - σ * μ * e
    R = -rᵇ + A * (-Θ*rᶜ + S⁻¹ * rˣˢ)

    Δλ = AΘAᵀ \ R
    Δs = -rᶜ - A'Δλ
    Δx = -S⁻¹ * (rˣˢ + X * Δs)

    return Δx, Δλ, Δs, μ
end
```

Code 5: Implementation of the Newton Step with Normal Equations.

2.2.2. Mehrotra Step

The Mehrotra (Corrector-Predictor) step with the base, normal and augmented systems are shown in Codes 6, 7, and 8 respectively. Now, all systems have been solved without using the LinearSolve.jl library, since it couldn't reuse the factorizations[5].

In all 3 systems, the suggested modification of the right-hand-side component of the complementarity equations has been applied (i.e. using $\sigma\mu e - \alpha_d^{\text{aff}}\Delta_X^{\text{aff}}\Delta_S^{\text{aff}}e$ instead of $\sigma\mu e - \Delta_X^{\text{aff}}\Delta_S^{\text{aff}}e$) has been applied in the first iterations ($\mu > 10$), since in some cases the procedure would get stuck at the beginning.

For the Base System (Code 6), an LU factorization has been used, since it's the one provided by Julia for generic square matrices.

```
function mehrotra_base_system(
  step::MehrotraStep, (; A, b, c)::StandardProblem, x::VF, λ::VF, s::VF
)::Tuple{VF,VF,VF,Float64}
    # Vars
    (m, n) = size(A)
    X = spdiagm(x); S = spdiagm(s)
    e = ones(n); I = spdiagm(e)
    Zᵐ = spzeros(m, m); Zⁿ = spzeros(n, n)
    Zᵐⁿ = spzeros(m, n); Zⁿᵐ = spzeros(n, m)
    μ = x's / n

    # System
    F = lu!(dropzeros!([
        Zⁿ A'  I;
        A  Zᵐ  Zᵐⁿ;
        S  Zⁿᵐ X
    ]))

    # Predictor
    rᶜ = A' * λ + s - c
    rᵇ = A * x - b
    rˣˢ = X * S * e
    P = [-rᶜ; -rᵇ; -rˣˢ]

    Δᵃ = F \ P
    Δxᵃ = Δᵃ[1:n]
    Δsᵃ = Δᵃ[n+m+1:end]

    αpᵃ = min(1, minimum((-x ./ Δxᵃ)[Δxᵃ .< 0]; init=Inf))
    αdᵃ = min(1, minimum((-s ./ Δsᵃ)[Δsᵃ .< 0]; init=Inf))
    μᵃ = ((x + αpᵃ * Δxᵃ)' * (s + αdᵃ * Δsᵃ)) / n
    σ = (μᵃ / μ)^3

```

[5] In theory, caching and reusing the factorizations should be possible, and it is stated in their documentation. However, when used in the code, it didn't yield correct results, so manual factorization and solving was the used approach.

```
    # Centering + Corrector
    if μ > 10
        c = μ * σ * e - αdᵃ * Δxᵃ .* Δsᵃ
    else
        c = μ * σ * e - Δxᵃ .* Δsᵃ
    end
    C = [zeros(n); zeros(m); c]

    Δᶜ = F \ P
    Δ = Δᵃ + Δᶜ

    return Δ[1:n], Δ[n+1:n+m], Δ[n+m+1:end], μ
end
```

Code 6: Implementation of the Mehrotra Step with Base System.

The Augmented System has been solved using also a LU factorization, since the Bunch-Kaufman factorization implementation is only available for dense matrices. According to the Julia documentation, the $\mathrm{LDL}^\intercal$ factorization supports arbitrary symmetric sparse matrices, but in practice, it fails due to the zeros in the diagonal (and suggests using the LU factorization in the error message). As in the Newton Step, here $\Theta = -\left(XS^{-1}\right)^{-1}$ instead of XS^{-1}.

```
function mehrotra_augmented_system(
  step::MehrotraStep, (; A, b, c)::StandardProblem, x::VF, λ::VF, s::VF
)::Tuple{VF,VF,VF,Float64}
    # Vars
    (m, n) = size(A)
    X = spdiagm(x); X⁻¹ = spdiagm(1 ./ x); S = spdiagm(s)
    Θ = -spdiagm(s ./ x) # -(X*S⁻¹)⁻¹,  Makes matrix below prettier :)
    e = ones(n); Zᵐ = spzeros(m, m)
    μ = x's / n

    # System
    F = lu!(dropzeros!([
        Θ A';
        A Zᵐ
    ]))

    # Predictor
    rᶜ = A' * λ + s - c
    rᵇ = A * x - b
    rˣˢ = X * S * e
    P = [-rᶜ + X⁻¹ * rˣˢ; -rᵇ]

    Δᵃ = Fᶠ \ P
    Δxᵃ = Δᵃ[1:n]
    Δλᵃ = Δᵃ[n+1:n+m]
    Δsᵃ = -X⁻¹ * (rˣˢ + S * Δxᵃ)

    αpᵃ = min(1, minimum((-x./Δxᵃ)[Δxᵃ.<0]; init=Inf))
    αdᵃ = min(1, minimum((-s./Δsᵃ)[Δsᵃ.<0]; init=Inf))
    μᵃ = ((x + αpᵃ * Δxᵃ)' * (s + αdᵃ * Δsᵃ)) / n
    σ = (μᵃ / μ)^3

    # Centering + Corrector
    if μ > 10
        c = -σ * μ * e + αdᵃ * Δxᵃ .* Δsᵃ
    else
        c = -σ * μ * e + Δxᵃ .* Δsᵃ
    end
    C = [X⁻¹ * c; zeros(m)]

    Δᶜ = Fᶠ \ C
    Δxᶜ = Δᶜ[1:n]
    Δλᶜ = Δᶜ[n+1:n+m]
```

```
    Δsᶜ = -A'Δλᶜ

    return Δxᵃ + Δxᶜ, Δλᵃ + Δλᶜ, Δsᵃ + Δsᶜ, μ
end
```

Code 7: Implementation of the Mehrotra Step with Augmented System.

For the Normal System, a Cholesky factorization is used. As in the Newton step implementation, when the factorization of the matrix fails, a small permutation is added to the diagonal.

```
function mehrotra_normal_system(
  step::MehrotraStep, (; A, b, c)::StandardProblem, x::VF, λ::VF, s::VF
)::Tuple{VF,VF,VF,Float64}
    # Vars
    (m, n) = size(A)
    X = spdiagm(x); S = spdiagm(s); S⁻¹ = spdiagm(1 ./ s)
    Θ = X * S⁻¹; e = ones(n)
    μ = x's / n

    # System
    AΘAᵀ = cholesky!(Symmetric(A * Θ * A'); check=false)
    if !issuccess(AΘAᵀ)
        @warn "Cholesky factorization failed. Adding perturbation"
        AΘAᵀ = cholesky!(Symmetric(A * Θ * A') + 1e-6 * I)
    end

    # Predictor
    rᶜ = A' * λ + s - c
    rᵇ = A * x - b
    rˣˢ = X * S * e
    R = -rᵇ + A * (-Θ*rᶜ + S⁻¹*rˣˢ)

    Δλᵃ = AΘAᵀ \ R
    Δsᵃ = -rᶜ - A'Δλᵃ
    Δxᵃ = -S⁻¹ * (rˣˢ + X * Δsᵃ)

    αpᵃ = min(1, minimum((-x ./ Δxᵃ)[Δxᵃ .< -1e-10]; init=Inf))
    αdᵃ = min(1, minimum((-s ./ Δsᵃ)[Δsᵃ .< -1e-10]; init=Inf))
    μᵃ = ((x + αpᵃ * Δxᵃ)' * (s + αdᵃ * Δsᵃ)) / n
    σ = (μᵃ / μ)^3

    # Centering + Corrector
    if μ > 10
        c = -σ * μ * e + αdᵃ * Δxᵃ .* Δsᵃ
    else
        c = -σ * μ * e + Δxᵃ .* Δsᵃ
    end
    C = A * S⁻¹ * c

    Δλᶜ = AΘAᵀ \ C
    Δsᶜ = -A'Δλᶜ
    Δxᶜ = -S⁻¹ * (c + X * Δsᶜ)

    return Δxᵃ + Δxᶜ, Δλᵃ + Δλᶜ, Δsᵃ + Δsᶜ, μ
end
```

Code 8: Implementation of the Mehrotra Step with Normal Equations.

3. Results

3.1. Step Computation

An initial comparison between the performance of the different step and computation methods has been carried out. A total of 77 Netlib problems have been used, and the full results are

provided in the `pdpf_step.csv` file. However, due to size and readability constraints, only a representative subset of the results are shown here.

The characteristics of the 6 representative problems are shown in Table 1, and as can be seen, include problems of varying size, both in standard and non-standard form.

Problem	Dimensions				Standardized Dimensions				
	N	M	Size	NZ	SEF	SN	SM	SSize	SNZ
lp_adlittle	138	56	7728	424	true	138	56	7728	424
lp_stocfor1	165	117	19305	501	true	165	117	19305	501
lp_israel	316	174	54984	2443	true	316	174	54984	2443
lp_etamacro	816	400	326400	2537	false	1105	689	761345	3115
lp_grow22	946	440	416240	8252	false	1826	1320	2410320	10012
lp_czprob	3562	929	3309098	10708	false	3791	1158	4389978	11166

Table 1: Characteristics of the Netlib problems used in the Step comparison.

The results of the comparison are shown in Tables 2 and 3, the first showing the number of iterations for each method/system, and the seconds showing the execution time. Experiments for which the execution failed have a dash in their values.

As can be seen, the Newton step with the Base system (and Augmented) are the most robust, and didn't fail for any of the shown problems. In practice all of them fail for some problems, but the Newton step with the Base system remains the most robust one. In many cases, the starting point is the factor that dictates whether a step fails or not. For example, the `lp_stocfor1` problem failed in the experiment, but when running it again several times with different starting points (chosen at random), it solved many of them.

In Table 2 we can see that normally, different computation methods don't modify the number of iterations (as is expected), but that's not always the case. For example, in many cases (such as `lp_etamacro`), the Augmented system requires more iterations (in both step types), while in others (`lp_grow22`), all computation methods require a different number of iterations. This is due to numerical errors in the computations yielding slightly different results, which end up being significant enough to make the algorithm run longer.

Problem	Newton			Mehrotra		
	Base	Augmented	Normal	Base	Augmented	Normal
lp_adlittle	43	43	43	24	-	-
lp_stocfor1	34	34	-	-	-	-
lp_israel	314	314	314	90	90	89
lp_etamacro	48	59	48	32	35	32
lp_grow22	941	934	936	231	239	238
lp_czprob	109	109	109	46	46	46

Table 2: Number of iterations for the different step types and systems

Regarding execution times, shown in Table 3, the behavior is as expected. Within the step type, the Normal system is significantly faster then the Augmented system, which, in turn, is significantly faster than the Base System. In general, the Mehrotra Step is faster than the

Newton Step, but the speedup is directly tied to the number of iterations required, and not because the step is inherently faster to compute (which shouldn't be since it requires more computations).

Problem	**Newton**			**Mehrotra**		
	Base	**Augmented**	**Normal**	**Base**	**Augmented**	**Normal**
lp_adlittle	2.97E-02	2.16E-02	5.22E-03	1.66E-02	-	-
lp_stocfor1	2.76E-02	2.11E-02	-	-	-	-
lp_israel	1.10E+00	8.95E-01	4.09E-01	3.04E-01	2.32E-01	1.20E-01
lp_etamacro	3.84E-01	3.74E-01	8.86E-02	2.50E-01	2.25E-01	6.00E-02
lp_grow22	1.30E+01	1.08E+01	3.11E+00	3.12E+00	2.77E+00	6.69E-01
lp_czprob	5.60E+00	3.94E+00	2.46E-01	2.40E+00	1.62E+00	9.83E-02

Table 3: Execution time in seconds for the different step types and systems

3.2. Solver comparison

The implemented procedure has been tested with a total of 77 of Netlib's LP problems. In particular the first 77 problems after sorting them ascendingly by the size of their coefficient matrices after applying standardization. All problems have been run with a feasibility tolerance $\varepsilon^f = 1e^{-8}$, an optimality tolerance $\varepsilon^o = 1e^{-8}$ and a step size reduction factor $\rho = 0.99$. The full results of the comparison are provided in the `pdpf_comparison.csv` file.

The same problems have been solved using two of the many solvers available in Julia: Tulip [1] and HiGHS [2]. 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 [2].

Of the 77 problems, our implementation of the Primal-Dual Path-Following Algorithm has been able to solve 74, although not with every step type. Table 4 shows the number of problems that have been solved by each step type. We can see that while we were able to solve almost all of them with the Newton step with Base system, we could solve less with the rest, the least being the Mehrotra step with the Normal system, with 32 problems solved. Tulip and HiGHS were able to solve 72 and 67 problems each.

	Newton			Mehrotra			Tulip	HiGHS
	Base	Augmented	Normal	Base	Augmented	Normal		
Num Solved	74	58	36	35	33	32	72	67

Table 4: Problems solved by each Step

We showcase here a handpicked selection of 6 problems in which the Mehrotra step with Normal system successfully solves the problems and compare its performance with that of the Tulip and HiGHS solvers. Table 5 show the dimensions and characteristics of the problems shown.

Problem	**Dimensions**				**Standardized Dimensions**				
	N	**M**	**Size**	**NZ**	**SEF**	**SN**	**SM**	**SSize**	**SNZ**
lp_afiro	51	27	1377	102	true	51	27	1377	102
lp_fit1p	1677	627	1051479	9868	false	2076	1026	2129976	10666

lp_fit1d	1049	24	25176	13427	false	2075	1050	2178750	15479
lp_sctap2	2500	1090	2725000	7334	true	2500	1090	2725000	7334
lp_czprob	3562	929	3309098	10708	false	3791	1158	4389978	11166
lp_sctap3	3340	1480	4943200	9734	true	3340	1480	4943200	9734

Table 5: Characteristics of the Netlib problems used in the Solver comparison.

Table 6 shows the number of iterations, cost of the solution and execution time for our implementation, the Tulip solver and the HiGHS solver.

We can see that in all 6 cases, our implementation of the PDPF obtains a similar result as both solvers. In number of iterations, it requires (generally) much fewer iterations than the HiGHS solver, which is to be expected since HiGHS is simplex-based. However, it still requires more than Tulip, which is interior-point-based.

As can be seen, our implementation manages to compete with the other solvers, even being faster in some cases (e.g. faster than Tulip for `lp_sctap3`).

Problem	**PDPF**			**Tulip**			**HiGHS**		
	Iter	**Cost**	**Time**	**Iter**	**Cost**	**Time**	**Iter**	**Cost**	**Time**
lp_afiro	14	-4.65E+02	2.34E-03	8	-4.65E+02	6.69E-04	6	-4.65E+02	5.24E-04
lp_fit1p	23	9.15E+03	7.79E-01	13	9.15E+03	2.91E-02	860	9.15E+03	3.77E-02
lp_fit1d	26	-9.15E+03	5.83E-02	25	-9.15E+03	4.47E-02	69	-9.15E+03	1.11E-02
lp_sctap2	21	1.72E+03	4.54E-02	12	1.72E+03	2.65E-02	912	1.72E+03	1.63E-02
lp_czprob	46	2.19E+06	9.83E-02	26	2.19E+06	5.96E-02	733	2.19E+06	2.46E-02
lp_sctap3	21	1.42E+03	5.91E-02	11	1.42E+03	8.47E-02	1237	1.42E+03	1.84E-02

Table 6: Experimental results between presented implementation, Tulip and HiGHS.

4. Conclusions

In this assignment we have provided a Julia implementation of the Primal-Dual Path-Following algorithm with two different step types (Newton and Mehrotra), with three different computations methods (Base, Augmented, and Normal).

Moreover we have presented a performance comparison between all the implemented steps, comparing both robustness and performance. The performance of our implementation has been also compared against two open-source solvers implementing different algorithms (both simplex-based and interior-point) using Netlib problems. Our implementation is capable of solving the majority of the used problems, and can keep up with the other solvers, not only regarding the solution obtained, but also in the time consumed.

Bibliography

[1] M. Tanneau, M. F. Anjos, and A. Lodi, "Design and implementation of a modular interior-point solver for linear optimization," *Mathematical Programming Computation*, Feb. 2021, doi: 10.1007/s12532-020-00200-8.

[2] Q. Huangfu and J. A. J. Hall, "Parallelizing the dual revised simplex method," *Mathematical Programming Computation*, vol. 10, no. 1, pp. 119–142, Mar. 2018, doi: 10.1007/s12532-017-0130-5.