École polytechnique fédérale de Lausanne

Semester project spring 2024

Master in Engineering Mathematics and Computational Science

Low-Rank Augmented Lagrangian Methods for Large-Scale SDP

An implementation of the HALLaR algorithm in Python

Author:
Edvin Dannäs

Supervisor:
Prof. Nicolas Boumal

Abstract

This report presents an implementation of the HALLaR algorithm (1) for solving large-scale semidefinite programming (SDP) problems in Python. The algorithm was assessed across multiple runs with varying initial points, demonstrating a consistent optimal value and robust performance. The impact of different choices of rank and β was analyzed with low rank solutions with β adjusted according to the graph size proved to be a good approach for fast and stable convergence.

In comparison with the original implementation by Monteiro et al. (1) this algorithm significantly lags behind in computational efficiency. Key areas for enhancing the performance of this implementation include refining the selection of input parameters, running optimization on a GPU and exploring other languages for improved efficiency in numerical computations. The algorithm should also be tested on other standard SDP problems and compared with other state-of-the-art SDP solvers.

A very important outcome of this project is the code repository `edannas/HALLaR` with the complete code for the implementation of the main algorithm as well as all supporting functions and files. This provides a solid framework for continued work.

Contents

1 Introduction

This section contains an introduction to the foundational concepts of this project: semidefinite programming, augmented lagrangian methods and low rank methods, as well as the project aim and an outline of the HALLaR algorithm which is the basis of this project.

1.1 Semidefinite Programming

Semidefinite programming (SDP) is a subcategory of mathematical optimization where a linear objective function is minimized (or maximized) over a domain given by the set of positive semidefinite (PSD) matrices. This domain is typically subject to additional linear constraints. A matrix X is PSD iff it is symmetric with only non-negative eigenvalues, commonly denoted as $X \succeq 0$. The set of PSD matrices forms a convex cone, and the feasible region of an SDP is the intersection of this cone with an affine space defined by the linear constraints (2).

SDP can be seen as a generalization of linear programming (LP), where the domain now is matrices instead of vectors, and the non-negativity constraints is substituted by the PSD constraint. In LP, the feasible region is a polyhedron defined by linear inequalities on vectors, while in the case of SDP, the feasible region is instead a convex set of matrices. This generalization allows SDP to handle a broader class of problems, involving matrix variables and more general constraints.

Mathematical Formulation
An SDP problem can be written as follows:

$$\begin{array}{rl} \min\limits_{X \in S^n} & \langle C, X \rangle \\ \text{subject to} & A(X) = b \\ & X \succeq 0 \end{array} \tag{1}$$

where $C \in \mathbb{S}^n$, $A : \mathbb{S}^n \to \mathbb{R}^m$ is a linear map, and $b \in \mathbb{R}^m$. $X \in \mathbb{S}^n$ is the optimization variable.
$\langle ., . \rangle$ is a representation of the Frobenius inner product of matrices, defined as:

$$\langle M, N \rangle = \mathrm{Tr}(M^T N) = \sum_{i,j} M_{ij} N_{ij}$$

The objective is to find the matrix X that minimizes the product $\langle C, X \rangle$ while satisfying the given constraints and ensuring the positive semi-definiteness of X.

Equivalently, the problem can be reformulated as follows:

$$\begin{array}{rl} \min\limits_{X \in S^n} & \mathrm{Tr}(CX) \\ \text{subject to} & A(X) = b \\ & X \succeq 0 \end{array} \tag{2}$$

The objective is to find the matrix X that minimizes the trace of CX while satisfying the given constraints and ensuring that X is PSD.

Applications
Semidefinite programming is a relatively new area within mathematical optimization that has gained attention due to its capability to model and approximate a wide range of practical problems. Recent increases in computational power and the development of efficient algorithms such as interior-point methods and first-order methods has made it possible to solve large-scale SDP problems.(3) Additionally, the integration of SDP with other optimization techniques has led to the development of hybrid methods that leverage the strengths of multiple approaches.(4; 3) Listed below are some key applications of SDP:

- **Control Theory**: SDP is used in control theory for solving problems related to system stability and performance, where linear matrix inequality constraints (LMIs) arise naturally from the differential equations that model the system.(5)
- **Combinatorial Optimization**: SDP provides relaxations for hard combinatorial problems like the max-cut problem and the graph coloring problem. The relaxation of these problems to SDPs often leads to approximation algorithms with provable guarantees.(6)
- **Signal Processing**: In signal processing, SDP is used for filter design and for solving inverse problems. It helps in designing filters that meet specific frequency response criteria and in estimating signals from noisy measurements.(7)
- **Quantum Information Science**: SDP is well suited for a range of problems in quantum information science, including predicting quantum states and for problems involving quantum entanglement and channel capacities.(8)

- **Finance**: In finance, SDP is used for portfolio optimization, risk management, and option pricing. For example, it helps in modeling the PSD covariance matrix of asset returns.(9)
- **Machine Learning**: SDP plays an important role in machine learning, particularly in support vector machines (SVMs) and kernel methods. It helps in deriving the optimal separating hyperplane and in learning the kernel matrix for non-linear SVMs.(10; 11) The applications within ML are subject to high efforts in efficiency improvements and scalability for successful implementations on the very large problem instances that are present within this field (12)

1.2 Augmented Lagrangian Methods

Augmented Lagrangian methods are a class of algorithms used for solving constrained optimization problems. The constrained optimization problem is converted into a series of unconstrained problems by incorporating the constraints into the objective function, using Lagrange multipliers and an additional quadratic penalty term.

Given a constrained optimization problem on the form:

$$\begin{aligned} \min_x \quad & f(x) \\ \text{subject to} \quad & c_i(x) = 0, \quad i = 1, \dots, m \end{aligned}$$

the augmented Lagrangian function $\mathcal{L}(x, \lambda, \rho)$ is defined as:

$$\mathcal{L}(x, \lambda, \rho) = f(x) - \sum_{i=1}^{m} \lambda_i c_i(x) + \frac{\rho}{2} \sum_{i=1}^{m} c_i(x)^2$$

where λ_i are the Lagrange multipliers for equality constraints, and ρ is a positive penalty parameter.

For the general SDP formulation, the augmented lagrangian function can be defined as:

$$\mathcal{L}(X, p, \beta) = \text{Tr}(CX) + p^T(A(X) - b) + \frac{\beta}{2}||A(X) - b||^2$$

where p is the vector of lagrangian multipliers and β is the penalty parameter.

Algorithm

The augmented Lagrangian method iteratively updates the primal variable X, the Lagrange multipliers p and the penalty parameter β. The basic steps of the algorithm are as follows:

1. Initialize $X^{(0)}$, $p^{(0)}$, and $\beta^{(0)}$.
2. At iteration k, solve the unconstrained optimization problem:

$$x^{(k+1)} = \arg\min_x \mathcal{L}_a(X, p^{(k)}, \beta^{(k)})$$

3. Update the Lagrange multipliers:

$$p^{(k+1)} = p^{(k)} + \beta^{(k)}(A(X_k) - b)$$

4. Update the penalty parameter β according to selected method.
5. Repeat steps 2-4 until convergence.

The method can be extended to handle inequality constraints but this is not of interest for this implementation.

Applications

Augmented Lagrangian methods are widely used in various fields due to their robustness and efficiency in handling complex constrained optimization problems. Some include:

- **Engineering Design**: Used for structural optimization, control system design, and other engineering problems where constraints play a critical role.(13; 14)
- **Economics and Finance**: Applied in portfolio optimization and other economic models involving constraints.(15)
- **Machine Learning**: Used in training models with regularization terms, constrained feature selection, and various other problems within machine learning.(16)
- **Operations Research**: Employed in solving large-scale linear and nonlinear programming problems, including scheduling, resource allocation, and logistics.(17; 18)

The augmented Lagrangian method provides a powerful framework for tackling constrained optimization problems by iteratively refining the solution and punishing violation of constraints.

1.3 Low Rank Methods

The Low-Rank (LR) approach in semidefinite programming (SDP) is motivated by the fact that SDPs often have optimal solutions with small ranks. Specifically, it has been established for both equality and inequality constrained SDP that $r^* \leq \sqrt{2m}$, where r^* is the smallest among the ranks of all optimal solutions of the SDP problem.(19; 20) This insight leads to the Burer-Monteiro approach, which consists of solving subproblems obtained by restricting the SDP to matrices of rank at most r, for some integer r. This is equivalently expressed as the non-convex smooth reformulation:

$$\begin{array}{cl} \min\limits_{U \in \mathbb{R}^{n\times r}} & \langle C, UU^T \rangle \\ \text{subject to} & A(UU^T) = b \\ & \|U\|_F \leq 1 \end{array} \tag{3}$$

The following is an equivalent and computationally preferred reformulation due to the lower memory requirements:

$$\begin{array}{cl} \min\limits_{U \in \mathbb{R}^{n\times r}} & Tr((CU)U^T) \\ \text{subject to} & A(UU^T) = b \\ & \|U\|_F \leq 1 \end{array} \tag{4}$$

Here, the Frobenius norm constraint $\|U\|_F \leq 1$ ensures the stability of the solution. This is equivalent to (2) when $r \geq r^*$, (U^* optimal for (4) $\Rightarrow X^* = U^*(U^*)^T$ optimal for (2).

The advantage of this approach lies in the fact that its matrix variable U has significantly fewer entries than X when $r \ll n$. However, since this form is non-convex, it may have stationary points that are not globally optimal, and the appropriate algorithm has to be chosen to handle the non-convex nature of the problem. The LR approach significantly reduces the computational burden associated with large-scale SDPs, making it a powerful tool in practical applications where other algorithms can be prohibited by the very large problem size.

Applications

Low-rank methods are utilized in various fields for their computational efficiency and ability to handle large-scale problems:

- **Signal Processing**: Low-rank approximations are used to recover signals from incomplete or noisy data, retrieving the underlying low-rank structure.(21)
- **Machine Learning**: Low-rank approximations are relied upon in ML for tasks like recommendation systems and dimensionality reduction.(22; 23)
- **Control Theory**: In robust control design, low-rank approximations are employed to simplify the complexity or reduce energy requirements of control systems while maintaining performance.(24)
- **Computer Vision**: Low-rank models are applied in image compression and other tasks where reducing the dimensionality of data is beneficial.(25)

1.4 HALLaR - algorithm outline

HALLaR is an inexact augmented Lagrangian (AL) method introduced by Monteiro et al. (1) that generates sequences $\{T_t\}$ and $\{p_t\}$ according to the following:

$$U_t \approx \arg\min_{U \in \mathbb{R}^{n\times r}} \{L_{LR}(U; p_{t-1})\} \tag{5}$$

$$p_t = p_{t-1} + \beta(A(U_t U_t^T) - b) \tag{6}$$

where

$$L_{LR}(U; p) := Tr((CU)U^T) + p^T(A(UU^T) - b) + \frac{\beta}{2}\|A(UU^T) - b\|^2 \tag{7}$$

The key part of HALLaR is the low rank approach to solving the augmented lagrangian function for large scale SDP. It includes a method called Hybrid Low-Rank (HLR), for finding an approximate global solution U_t of the AL subproblem (5), for some integer $r \geq 1$. Subproblem (5) is equivalent to restricting X in (4) to matrices with rank at most r. Since (5) is non-convex, it may have a *spurious* stationary point, depending on the choice of r.(1)

More specifically, HLR finds an approximate global solution by the following steps:

1. Finding a near stationary point $Y \in \mathbb{R}^{n\times r}$ using an adaptive accelerated inexact proximal point method (ADAP-AIPP).
2. Checking if $Y_k Y_k^T$ is nearly optimal for (2) through a minimum eigenvalue computation.
3. If not, a Frank-Wolfe step is used to move away from the current spurious near stationary point Y_k and the output is used as initial point for the next iteration.

Some notes: for obvious computational reasons HALLaR only stores the current iterate U and never computes the (implicit) iterate UU^T (in X-space). Also, under the some assumptions, it is shown that HALLaR obtains an approximate solution with provable computational complexity bounds expressed in terms of parameters associated with the SDP instance and specified tolerances.(1)

1.5 Aim

This project aims to study and understand the HALLaR algorithm as well as attempting and evaluating an implementation in Python based on Monteiro et al. (1).

2 Methods

The methodology consists of two parts: the implementation of the HALLaR algorithm in Python, and the evaluation of the implementation with regards to some pre-defined performance metrics.

2.1 Implementation

The implementation of the HALLaR algorithm in Python is structured as a function `hallar()` with several input parameters defining the initial points, tolerance, penalty parameter and maximum iterations. This function iteratively solves the Lagrangian minimization problem subject to a trace constraints until a stopping criterion is met. The following Python libraries are used in the implementation:

- `numpy`: For numerical operations and array handling.
- `scipy.optimize`: The `minimize` function is used for solving the Lagrangian minimization problem, and `NonlinearConstraint` used for defining the trace constraint on the optimization problem.

Below is an overview of the key components and structure of the implementation:

- **Initial Iterates**: The function initializes the iteration counter, current iterate Y_t, and Lagrangian multiplier p_t with the provided initial points.
- **Optimization Loop**: The main loop iterates until a stopping criterion is met or the maximum number of iterations is reached. Within each iteration, the Lagrangian minimization problem is solved using the `scipy.optimize.minimize()` function, subject to the trace constraint using the provided constraint function `con()`. In the low rank augmented lagrangian the PSD constraint is already satisfied (by the computation UU^T). There are three `scipy` optimizers that can handle constraints: `SLSQP`, `COBYLA` and `trust-constr`. The last option was chosen, since as the name suggest it seemed to be superior in satisfying the constraints. The gradient function was provided to the optimizer, something that as suspected greatly increased the efficiency of the algorithm.
- **Constraint Evaluation**: After obtaining the optimal solution, the function checks whether the trace constraint is satisfied for the solution Y_t. If the constraint is not satisfied, a message is printed indicating the violation.
- **Lagrangian Update**: The Lagrangian multiplier p_t is updated based on the violation of constraints multiplied with the penalty parameter β.
- **Stopping Criterion**: The function terminates when the norm of $A(UU^T) - b$ is less than the specified tolerance ϵ, indicating convergence.
- **Safety Break**: If the maximum number of iterations is reached, the function stops to prevent infinite looping.
- **Output**: The function returns the final iterate U_t, Lagrangian multiplier p_t, minimal eigenvalue θ_t, and the value of the Lagrangian at the optimal solution.

The implementation follows the algorithmic outline of HALLaR and utilizes efficient optimization techniques provided by the `scipy` library in Python to solve the low rank problem, instead of implementing HLR. It incorporates the necessary computations and checks to ensure convergence and satisfaction of constraints during the iterative process.

2.2 Performance Evaluation

This section includes a description of the sample problem, datasets, evaluation metrics, hardware, and software used for the performance evaluation of the HALLaR algorithm implementation.

2.2.1 Sample Problem - Maximum Stable Set

The maximum stable set problem for a graph $G = ([n], E)$ involves identifying the largest subset of vertices where no two vertices are adjacent. Lovász (26) introduced the θ-function, which serves as an upper bound for the maximum stable set. This θ-function can be formulated as an SDP (1):

$$\max\left\{ee^T \cdot X \mid X_{ij} = 0 \text{ for } (i,j) \in E,\ \operatorname{tr}(X) = 1,\ X \geq 0,\ X \in S^n(\mathbb{R})\right\} \tag{87}$$

where $e = (1, 1, \dots, 1) \in \mathbb{R}^n$. It has been demonstrated that the θ-function matches the stable set number exactly for perfect graphs (26).

2.2.2 Datasets

The `NetworkX` library was used to manage graphs, and specifically the function `gnm_random_graph()` was used to create random sample graphs with selected number of edges and nodes. Three sample graphs were created on which the algorithm is tested, with the following sizes (nodes, edges):

- *small1* : (100, 200)
- *small2* : (250, 500)
- *mid* : (500, 1000)

2.2.3 Evaluation Metrics

The following metrics were used to evaluate the performance of the algorithm:

- **Objective Function Value**: The final value of the objective function was compared to the value provided by the built in `maximal_independent_set()`-function of the `NetworkX` library. This function is based on the theory of subgraph-excluding algorithms presented in (27). The control algorithm is run 10 times and the average value is used as reference, it does not solve the maximum stable set problem.
- **Constraint Violation**: The norm of $A(UU^T) - b$, measuring how well the solution satisfies the constraints.
- **Number of Iterations**: The number of iterations required to converge to a solution.
- **Computation Time**: The total time to reach convergence.

These metrics provide a view of both the efficiency and the effectiveness (accuracy) of the algorithm. The performance of the algorithm was compared for different choices of rank and β.

2.2.4 Hardware

The performance evaluation was conducted on a MacBook Pro (14-inch, 2023) with the following specifications:

- **Processor**: Apple M2 Pro chip
- **Memory**: 16 GB
- **Operating System**: macOS Sonoma 14.2.1

2.2.5 Software

The software environment for the evaluation includes:

- **Python**: Version 3.11.0
- **NumPy**: For numerical computations
- **SciPy**: For optimization routines
- **NetworkX**: For handling graphs
- **Matplotlib**: For visualizing results

2.2.6 Experimental Setup

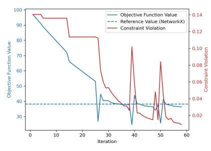
The experimental setup involves running the HALLaR algorithm on the datasets and recording the evaluation metrics for each run. The results were plotted and analyzed to determine the performance characteristics of the algorithm in terms of solution quality, computational efficiency, and scalability. The default convergence threshold is set to 1e−2. The initial point is chosen as a random matrix $U_0 \in \mathbb{R}^{nxr}$ normalized such that $||U||_F = 1$.

3 Results

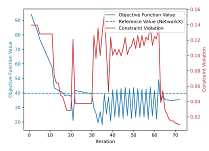
This section contains the results of the performance evaluation of the algorithm on the selected datasets.

3.1 β-selection

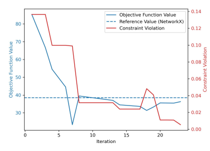
The algorithm was implemented on all datasets for different β-values and with rank = 1 (Figure 1). $\beta = 500$ seems to provide a stable and fast convergence for the small graphs, while smaller values are either very slow (see *Small2*, $\beta = 100$) or getting stuck for many iterations in suboptimal solutions (see *Small2*, $\beta = 200$). Larger values are fast but not always reliable (see *Small1*, $\beta = 5000$). $\beta = 500$ is used for further analysis on the small graphs. On the mid-size graph, due to computational time and the increased size of the graph, $\beta = 5000$ is chosen for further analysis. On this graph the algorithm with $\beta = 100$ or $\beta = 200$ did not terminate until the set limit of 1000 iterations; $\beta = 100$ being to slow, and $\beta = 200$ getting stuck in suboptimal solutions.



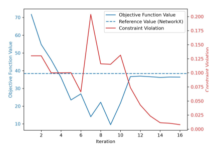
(a) *Small1*, $\beta = 100$



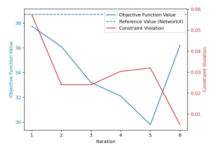
(b) *Small1*, $\beta = 200$



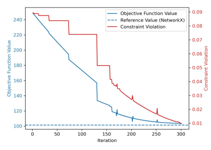
(c) *Small1*, $\beta = 500$



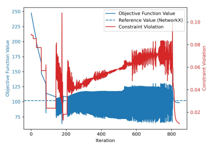
(d) *Small1*, $\beta = 1000$



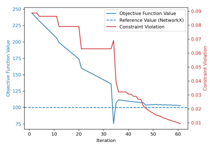
(e) *Small1*, $\beta = 5000$



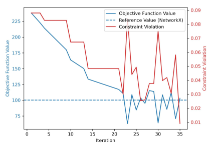
(f) *Small2*, $\beta = 100$



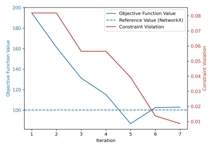
(g) *Small2*, $\beta = 200$



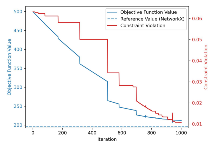
(h) *Small2*, $\beta = 500$



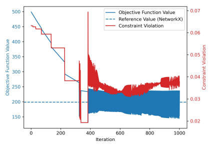
(i) *Small2*, $\beta = 1000$



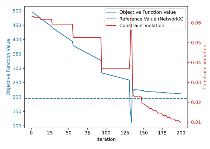
(j) *Small2*, $\beta = 5000$



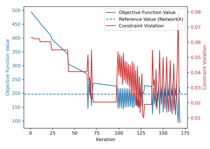
(k) *Mid*, $\beta = 100$



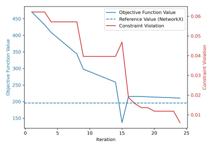
(l) *Mid*, $\beta = 200$



(m) *Mid*, $\beta = 500$



(n) *Mid*, $\beta = 1000$

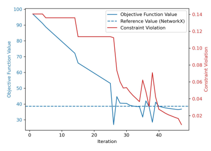


(o) *Mid*, $\beta = 5000$

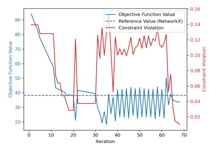
Figure 1: Implementation on all datasets for different β-values, rank = 1

3.2 Rank Selection

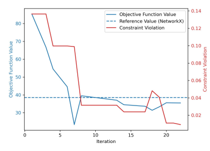
The algorithm was implemented on both small datasets for different β-values and with rank = 10 (Figure 2). The performance is similar to that with rank = 1, but with increased computational requirements, which is why the latter is chosen as default value for further analysis. E.g. *Small2*, $\beta = 100$ is very stable but also extremely slow in terms of computation time.



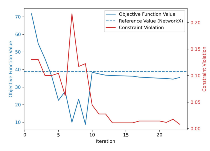
(a) *Small1*, $\beta = 100$



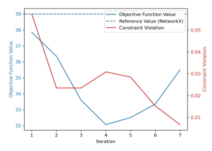
(b) *Small1*, $\beta = 200$



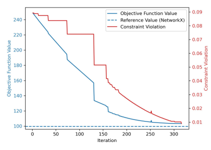
(c) *Small1*, $\beta = 500$



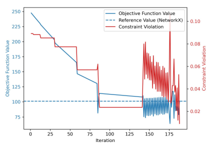
(d) *Small1*, $\beta = 1000$



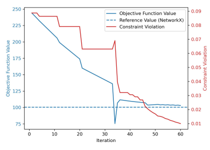
(e) *Small1*, $\beta = 5000$



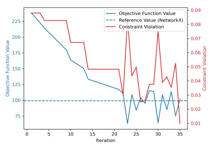
(f) *Small2*, $\beta = 100$



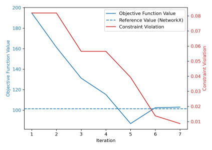
(g) *Small2*, $\beta = 200$



(h) *Small2*, $\beta = 500$



(i) *Small2*, $\beta = 1000$



(j) *Small2*, $\beta = 5000$

Figure 2: Implementation on small datasets for different β-values, rank = 10

3.3 Performance evaluation

Run No.	Optimal Value	$\|\|AX - b\|\|$	No. Iterations	Comp. time (s)	Nx-value (10 avg. & span)
1	35.8	0.004	19	2.0	37.8 (34-41)
2	36.5	0.009	12	1.2	39.1 (35-41)
3	35.7	0.008	27	2.9	38.6 (34-41)
4	36.3	0.008	35	3.4	38.7 (36-41)
5	35.6	0.007	35	3.7	38.8 (37-40)

Table 1: Performance metrics for dataset *Small1*. ($\beta = 500$)

Run No.	Optimal Value	$\|\|AX - b\|\|$	No. Iterations	Comp. time (s)	Nx-value (10 avg. & span)
1	103.6	0.010	60	39.9	100.5 (97-107)
2	103.0	0.009	61	34.6	100.3 (97-107)
3	103.0	0.010	60	30.7	100.7 (92-107)
4	103.1	0.010	60	34.6	102.3 (99-107)
5	103.1	0.010	60	31.7	100.2 (95-104)

Table 2: Performance metrics for dataset *Small2*. ($\beta = 500$)

Run No.	Optimal Value	$\|\|AX - b\|\|$	No. Iterations	Comp. time (s)	Nx-value (10 avg. & span)
1	211.2	0.008	23	32.0	197.1 (189-208)
2	210.3	0.006	24	33.8	194.3 (184-200)
3	211.2	0.008	23	31.3	197.8 (187-206)
4	211.2	0.008	23	31.4	196.9 (187-210)
5	211.2	0.008	23	31.3	198.0 (189-205)

Table 3: Performance metrics for dataset *Mid*. ($\beta = 5000$)

4 Discussion

The optimal value found by the algorithm is very consistent between different runs (with different initial points) and the value is correlating fairly well with the one suggested by the reference algorithm from the `NetworkX` library (tables 1-3). While the solution found by this built in algorithm, our algorithm seems fairly robust to changes in initial values, with similar solution, number of iterations and computation time across all runs. There seems to be a gap between the optimal value provided by our algorithm and the built in algorithm, that is also increasing from *small* to *mid*-implementations. This could be because of inaccuracies in our algorithm that increases with size of the graph, or with β, but it could also be because of the differences in algorithm architecture. As seen the size of the graph greatly effects the computation time, but so does the choice of β. For this reason the increased computational load of large graphs could at least partly be counteracted by the selection of a large β.

The choice of β greatly affected the behaviour of the algorithm, for example, higher values provide faster but also more unstable convergence. There were also some other interesting patterns, e.g. there seems to be a pattern that for $\beta = 200$ the algorithm ends up jumping between suboptimal solutions, the reason of which would be interesting to investigate further.

One can note that the computation time for this implementation is significantly higher than that in the implementation by Monteiro et al. (1) (e.g. a graph with 800 nodes and 1600 edges taking around 3 seconds). This could be for many reasons and suggest possible improvements for this algorithm design. A likely important factor is the HLR method that possibly would improve upon the scipy minimizer for this application.

Other possible improvements and suggestions for further work:

- Further investigate how the selection of rank and β effects the algorithm, and what would be an optimal set of parameters, e.g. an adaptive selection of β
- Test the algorithm on other standard sample problems such as phase retrieval or matrix completion.
- Try passing the optimization to a GPU and use parallel processing techniques of CUDA. This could be sensational for performance.
- The original implementation is in Julia, which compared to python could provide faster numerical computation.
- Compare performance with other state-of-the-art SDP solvers.

5 Code Repository

The complete code for the implementation of the main algorithm as well as all supporting functions and files is available at the GitHub repository `edannas/HALLaR`.

References

[1] R. D. C. Monteiro, A. Sujanani, and D. Cifuentes, “A low-rank augmented lagrangian method for large-scale semidefinite programming based on a hybrid convex-nonconvex approach,” 2024.

[2] B. Gärtner and J. Matoušek, *Semidefinite Programming*, pp. 15–25. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[3] B. Huang, S. Jiang, Z. Song, R. Tao, and R. Zhang, “Solving sdp faster: A robust ipm framework and efficient implementation,” 2021.

[4] W. J. van Hoeve, “A hybrid constraint programming and semidefinite programming approach for the stable set problem,” in *Principles and Practice of Constraint Programming – CP 2003* (F. Rossi, ed.), (Berlin, Heidelberg), pp. 407–421, Springer Berlin Heidelberg, 2003.

[5] V. Balakrishnan and F. Wang, *Semidefinite Programming in Systems and Control Theory*, pp. 421–441. Boston, MA: Springer US, 2000.

[6] L. Lovász, *Semidefinite Programs and Combinatorial Optimization*, pp. 137–194. New York, NY: Springer New York, 2003.

[7] S.-P. Wu, S. P. Boyd, and L. Vandenberghe, “Fir filter design via semidefinite programming and spectral factorization,” *Proceedings of 35th IEEE Conference on Decision and Control*, vol. 1, pp. 271–276 vol.1, 1996.

[8] V. Siddhu and S. Tayur, *Five Starter Pieces: Quantum Information Science via Semidefinite Programs*, p. 59–92. INFORMS, Oct. 2022.

[9] A. Gepp, G. Harris, and B. Vanstone, “Financial applications of semidefinite programming: a review and call for interdisciplinary research,” *Accounting and Finance*, vol. 60, pp. 3527–3555, December 2020.

[10] V. Piccialli, J. Schwiddessen, and A. M. Sudoso, “Optimization meets machine learning: An exact algorithm for semi-supervised support vector machines,” 2023.

[11] G. Lanckriet, N. Cristianini, P. Bartlett, L. Ghaoui, and M. Jordan, “Learning the kernel matrix with semi-definite programming.,” vol. 5, pp. 323–330, 01 2002.

[12] A. Majumdar, G. Hall, and A. A. Ahmadi, “A survey of recent scalability improvements for semidefinite programming with applications in machine learning, control, and robotics,” 2019.

[13] J. E. Coster and N. Stander, “Structural optimization using augmented lagrangian methods with secant hessian updating,” *Structural optimization*, vol. 12, no. 2, pp. 113–119, 1996.

[14] F. Lin, M. Fardad, and M. R. Jovanovic, “Augmented lagrangian approach to design of structured optimal state feedback gains,” *IEEE Transactions on Automatic Control*, vol. 56, no. 12, pp. 2923–2929, 2011.

[15] F. Xu, X. Li, Y.-H. Dai, and M. Wang, “New insights and augmented lagrangian algorithm for optimal portfolio liquidation with market impact,” *International Transactions in Operational Research*, vol. 30, no. 5, pp. 2640–2664, 2023.

[16] R. Gao, F. Tronarp, Z. Zhao, and S. Särkä, “Regularized state estimation and parameter learning via augmented lagrangian kalman smoother method,” in *2019 IEEE 29th International Workshop on Machine Learning for Signal Processing (MLSP)*, pp. 1–6, 2019.

[17] T. Nishi and M. Konishi, “An augmented lagrangian approach for scheduling problems,” *Jsme International Journal Series C-mechanical Systems Machine Elements and Manufacturing - JSME INT J C*, vol. 48, pp. 299–304, 12 2005.

[18] S. Yang, L. Ning, P. Shang, and C. Tong, “Augmented lagrangian relaxation approach for logistics vehicle routing problem with mixed backhauls and time windows,” *Transportation Research Part E Logistics and Transportation Review*, vol. 135, 02 2020.

[19] N. Boumal, V. Voroninski, and A. S. Bandeira, “The non-convex burer-monteiro approach works on smooth semidefinite programs,” 2018.

[20] D. Cifuentes, “On the burer-monteiro method for general semidefinite programs,” 2020.

[21] Z. Weng and X. Wang, “Low-rank matrix completion for array signal processing,” in *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 2697–2700, 2012.

[22] Z. Hu, F. Nie, R. Wang, and X. Li, “Low rank regularization: A review,” *Neural Networks*, vol. 136, pp. 218–232, 2021.

[23] C. Luo and Y. Xiang, “Dimensionality reduction based on low rank representation,” *The Open Automation and Control Systems Journal*, vol. 6, pp. 528–534, 2014.

[24] M. Cho, A. Abdallah, and M. Rasouli, “Low-rank lqr optimal control design over wireless communication networks,” 2023.

[25] R. Saha, V. Srivastava, and M. Pilanci, “Matrix compression via randomized low rank and low precision factorization,” 2023.

[26] L. M. Lovász, “On the shannon capacity of a graph,” *IEEE Trans. Inf. Theory*, vol. 25, pp. 1–7, 1979.

[27] R. Boppana and M. M. Halldórsson, “Approximating maximum independent sets by excluding subgraphs,” *BIT Numerical Mathematics*, vol. 32, no. 2, pp. 180–196, 1992.

Appendix A Code

```
# ----------------- IMPORTS AND SETUP -----------------
import numpy as np
from utils import *
from scipy.optimize import minimize, NonlinearConstraint

# -- Import problem definition --
# Predefined functions: A, A*, constraint_function, check_constraints, q
    , theta_tilde, compute_gradient
#                       import_graph_from_mtx, create_small_graph,
    create_large_graph, define_vars
# Predefined variables: nodes, edges, n, m, C, b

# ----------------- HALLaR algorithm -----------------
def hallar(
    # initial points
        Y_0, p_0,
    # tolerance pair
        epsilon_c, epsilon_p,
    # penalty parameter
        beta,
    # ADAP-AIPP parameters
        rho, lambda_0,
    # Maximum iterations
        max_iter
        ):

    # Define initial iterates
    t = 1
    Y_t = Y_0
    p_t = p_0

    # Solve Y_t (minimize Lagrangian) iteratively until stopping
        criterion is met
    # YY^T is automatically symmetric PSD.
    while True:
        # ---------------- scipy optimize -----------------
        # Define lagrangian for constant p_t
        def L_beta_scipy(Y_flat, *args):
            # Reshape Y_flat to Y
            n = len(nodes)
            s = int(len(Y_flat) / n)
            Y = Y_flat.reshape((n, s))

            AX_b = A(Y.dot(Y.T), m, nodes, edges) - b

            # Compute value of lagrangian function at YY^T
            return C.dot(Y).dot(Y.T).trace() + np.dot(p_t.T, AX_b) \
                + beta/2 * np.linalg.norm(AX_b) ** 2

        # Define and flatten the initial guess for optimization (warm
            start)
        Y_initial = Y_t
        Y_initial_flat = Y_initial.flatten()

        # Trace constraint
        con = lambda Y: np.sum(np.square(Y)) - 1
        nlc = NonlinearConstraint(con, -np.inf, 0)

        # Optimize the objective function subject to the trace
            constraint
```

```python
        result = minimize(L_beta_scipy, Y_initial_flat, constraints = nlc,
                          jac = compute_gradient, args=(A, A_adjoint, C, p_t, q, beta, nodes)) # , bounds=bounds

        # Retrieve the optimal solution
        optimal_solution_y_flat = result.x
        optimal_solution_y = optimal_solution_y_flat.reshape(Y_initial.shape)

        print("Trace of solution:", np.round(np.sum(np.square(optimal_solution_y)), 5))

        # Evaluate and check constraints for the optimal solution
        trace_constraint, eigenvalue_constraint = check_constraints(optimal_solution_y_flat, n)

        if not (trace_constraint and eigenvalue_constraint):
            print("Constraints are not satisfied (iteration {})".format(t))
        else:
            print("Constraints are satisfied.")
            pass

        # Define next iterate
        Y_t = optimal_solution_y
        # -----------------------------------------------------------------------

        AX_b = A(Y_t.dot(Y_t.T), m, nodes, edges) - b
        AX_b_norm = np.linalg.norm(AX_b)

        # Update Lagrangian multiplier (violation of constraints with penalty parameter beta)
        p_t = p_t + beta * AX_b

        # Stopping condition ||A(UU.T)-b|| < epsilon_p
        if AX_b_norm < epsilon_p: # ord='fro' ?
            print("Stopping criterion met (||A(UU.T)-b|| = {} < {} = epsilon_p)".format(
                np.round(AX_b_norm, 5),
                epsilon_p))
            break

        # Safety break
        if t > max_iter:
            print("Maximum iterations reached ({})".format(max_iter))
            break

        print("Iteration {}: ||A(UU.T)-b|| = {}, L(Y) = {}".format(t, \
            np.round(AX_b_norm, 5),
            np.round(L_beta_scipy(optimal_solution_y_flat), 5)))

        t = t + 1

    # Minimal eigenvalue computation
    theta_t = theta_tilde(Y_t.dot(Y_t.T), p_t, beta, A, A_adjoint, C, q)

    return Y_t, p_t, theta_t, L_beta_scipy(optimal_solution_y_flat)
```

Listing 1: HALLaR Algorithm Implementation in Python