MPAX: Mathematical Programming in JAX

Haihao Lu
Sloan School of Management
MIT
Cambridge, MA
haihao@mit.edu

Zedong Peng
Sloan School of Management
MIT
Cambridge, MA
zdpeng@mit.edu

Jinwen Yang
Department of Statistics
University of Chicago
Chicago, IL
jinweny@uchicago.edu

Abstract

This paper presents MPAX (Mathematical Programming in JAX), a versatile and efficient toolbox for integrating linear programming (LP) and quadratic programming (QP) into machine learning workflows. MPAX implements the state-of-the-art first-order methods, restarted average primal-dual hybrid gradient and reflected restarted Halpern primal-dual hybrid gradient, to solve LPs and QPs in JAX. This provides native support for hardware acceleration along with features such as batch solving, auto-differentiation, and device parallelism. Extensive numerical experiments demonstrate the advantages of MPAX over existing solvers.

1 Introduction

Linear programming and quadratic programming have long served as foundations across numerous fields, such as operations research, economics, and engineering, providing powerful robust tools for optimization and decision-making. Recently, these techniques have also found significant applications in machine learning. Notable examples include data-driven decision making [9, 20], learning with physical constraints [7, 10], learning to rank [6], end-to-end planning and control [2], among many others. The efficiency and effectiveness of these machine-learning approaches depend largely on the rapid processing of large-scale datasets, facilitated by parallel hardware accelerators such as graphics processing units (GPUs).

In contrast, traditional approaches to linear programming are not well suited for machine learning tasks. Broadly, there are two major paradigms for integrating mathematical programming with machine learning. The first approach involves migrating data from the GPU to the CPU, using commercial solvers like Gurobi on the CPU to solve the optimization problem, and then transferring the solution back to the GPU [9, 20, 23]. While these solvers are robust and reliable, frequent transfers of large-scale data between the CPU and GPU introduce significant overhead, often negating the efficiency gains achieved by using mature solvers. To address this limitation, recent efforts have focused on implementing optimization solvers natively on hardware accelerators. Examples include qpth [3] and qpax [24] that use interior-point methods (IPMs), and osqpth [22, 21], which employs the alternating direction method of multipliers (ADMM). While these methods reduce the need for multiple rounds of CPU-GPU communication, their primary bottleneck is the linear system solving step, which tends to be inefficient for large-scale problems and is not fully suitable for parallel accelerators.

To address the aforementioned limitations, we introduce MPAX (Mathematical Programming in JAX), a versatile and efficient toolbox for integrating linear programming into machine learning workflows. MPAX is built on two key pillars: recent advancements in first-order methods for solving classical linear programming problems, and the rapid progress of modern programming languages on contemporary computational platforms. From an algorithmic perspective, MPAX employs matrix-free first-order methods (FOMs) whose primary computational bottleneck is merely matrix-vector

39th Conference on Neural Information Processing Systems (NeurIPS 2025) Workshop: ScaleOPT: GPU-Accelerated and Scalable Optimization.

multiplication. Although FOMs have traditionally been considered less performant than IPM-based solvers, recent works such as cuPDLP [13, 16, 12] and PDQP [14] demonstrate that FOMs can achieve competitive numerical performance on GPUs, making them potentially suitable for general-purpose solvers. This efficiency is largely driven by the massive parallelization capabilities of GPUs for matrix-vector multiplications. Additionally, MPAX is implemented in JAX, offering a unified framework with several valuable features, including native support for CPUs, GPUs, and TPUs, batched solving of multiple instances, auto-differentiation, and efficient device parallelism. The use of JAX simplifies downstream applications and ensures seamless integration into machine learning pipelines, enhancing both flexibility and performance.

2 MPAX: Math Programming in JAX

MPAX is a hardware-accelerated, batchable, and differentiable solver built entirely in JAX, designed to address classic optimization problems encountered in data science and machine learning. MPAX supports general linear programs (LPs) and quadratic programs (QPs) formulated in the following form

$$\min_{l \le x \le u} \ c^\top x \quad \text{s.t. } Ax = b,\ Gx \ge h \qquad \text{(LP)} \qquad\qquad \min_{l \le x \le u} \ \tfrac{1}{2} x^\top Q x + c^\top x \quad \text{s.t. } Ax = b,\ Gx \ge h \qquad \text{(QP)}$$

where $G \in \mathbb{R}^{m_1 \times n}$, $A \in \mathbb{R}^{m_2 \times n}$, $Q \in \mathbb{R}^{n \times n}$, $c \in \mathbb{R}^n$, $h \in \mathbb{R}^{m_1}$, $b \in \mathbb{R}^{m_2}$, $l \in (\mathbb{R} \cup \{-\infty\})^n$, $u \in (\mathbb{R} \cup \{\infty\})^n$. As its core, MPAX has implemented the restarted average Primal-Dual Hybrid Gradient (raPDHG) algorithm for LPs and QPs, and the reflected restarted Halpern Primal-Dual Hybrid Gradient (r^{2}HPDHG) algorithm for LPs. Both algorithms build upon the vanilla PDHG framework and incorporate several enhancements to improve performance, including preconditioning, adaptive restart, adaptive step-size, primal weight update, infeasibility detection, and feasibility polishing. Full algorithm details of raPDHG and r^{2}HPDHG are provided in Appendix A. For further background on raPDHG and r^{2}HPDHG, we refer the readers to prior works [13, 15, 14].

The overall design of MPAX is illustrated in Figure 1. The process begins with preconditioning to improve the condition number of the input LP or QP instance. By default, MPAX applies two diagonal preconditioners: Ruiz scaling [19] and Pock and Chambolle's diagonal scaling method [17]. After preconditioning, raPDHG iterations or r^{2}HPDHG iterations begin on the scaled LP or QP, with a heuristic line search to determine an appropriate step size. During the iterations, three key conditions are periodically evaluated (by default, every 64 iterations): termination, restart, and infeasibility detection. For the restart condition, MPAX relies on the KKT error for raPDHG and the fixed-point residual for r^{2}HPDHG. When a restart occurs, the primal weight is updated to better harmonize the primal and dual spaces. Iterations terminate when the relative KKT error, including duality gap, primal feasibility, and dual feasibility, satisfies the specified termination tolerance. As a final step, feasibility polishing is applied after the PDHG iterations to further enhance the feasibility of the solution. This ensures the final output meets the feasibility requirements of real-world applications. The rest of this section introduces the advanced features of MPAX, most of which are not supported by previous Julia implementations [13, 15] and play an important role in machine learning applications.

2.1 Matrix Format

A key feature of MPAX is its support of both dense and sparse formats for the constraint matrix, providing flexibility to handle a wide range of LP and QP structures efficiently. The computational bottleneck of both raPDHG and r^{2}HPDHG algorithms lies in matrix-vector multiplication, which has been fully optimized in JAX for both sparse and dense matrices to maximize performance. However, mismatches between the matrix format and the actual data structure, such as storing a dense matrix in a sparse format or vice versa, can significantly impact computational efficiency. To address this, MPAX automatically follows the input matrix format to ensure that computations align with the structure of the given instance, thereby maintaining optimal efficiency and performance. In addition to natively accepting explicit vector and matrix inputs, MPAX also supports higher-level modeling via `CVXPY` [8] and integrates with the differentiable-optimization packages `cvxpylayers` [1] and `PyEPO` [23] for end-to-end training.

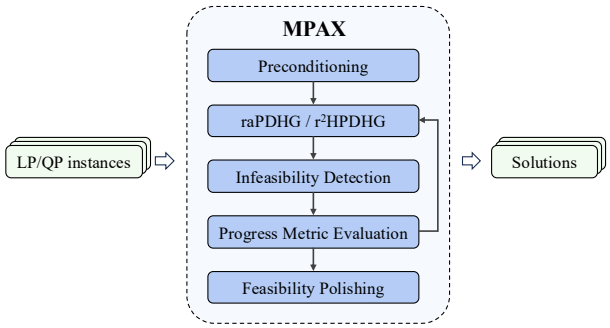

Figure 1: Design of MPAX

2.2 Just-in-time complication

Just-in-time (JIT) compilation is a critical feature of JAX, enabled by its XLA compiler, which translates high-level Python functions into highly optimized, hardware-specific executable code. This process combines operation fusion, parallel execution, and hardware-level optimization, significantly improving computational performance. In MPAX, the implementations of raPDHG and r^{2}HPDHG are designed to fully utilize JAX's JIT capabilities. By employing the control flow operators in JAX, MPAX ensures efficient computation graph optimization and execution. While the first JIT invocation incurs compilation overhead, subsequent iterations benefit from precompiled code, significantly improving the overall performance of raPDHG and r^{2}HPDHG.

2.3 Batched solving

MPAX supports efficient batch-solving of instances with identical shapes on a single accelerator, such as a GPU or TPU, through automatic vectorization. Specifically, it transforms each primitive operation into its batched equivalent using predefined batching rules. For example, a vector-vector multiplication can be transformed into a matrix-vector multiplication, allowing multiple operations to be executed in parallel within a single batched computation. This abstraction eliminates the need for explicit for-loops, reducing computational overhead and fully utilizing modern hardware's parallel processing capabilities. By solving multiple instances simultaneously, MPAX aligns with the batched computation paradigm commonly used in neural network training, where batch processing enhances both efficiency and scalability. This makes MPAX particularly effective for applications requiring the simultaneous solution of numerous similar optimization problems.

2.4 Auto-differentiation

Automatic differentiation is a key feature of MPAX. In JAX, explicit differentiation can be easily achieved by unrolling algorithms using the automatic differentiation system, which works for QPs. However, in the context of LP, the inherent discontinuity of LP solutions presents challenges for gradient computation. To address this, surrogate loss functions, such as the SPO+ loss and the PFYL loss, are used to approximate decision errors and their gradients. These surrogate losses can be seamlessly integrated into the chain rule by leveraging JAX's custom derivative rules. These loss functions typically require a forward pass to compute optimal solutions or decision losses and a backward pass to calculate gradients. The resulting gradient allows machine learning models to learn from decision errors or their approximations.

2.5 Device Parallelism

MPAX supports automatic device parallelism by combining just-in-time compilation with data-sharding techniques. Since GPUs/TPUs typically have less memory than CPUs, solving large-scale LP and QP problems might require distributing data across multiple devices. For these problems, the constraint and objective matrices are usually the most memory-intensive components, making them

the primary candidates for sharding. The device parallelism process begins by defining a mesh of devices with named axes, which organize and align the data for parallel computation. The matrix is then split into smaller shards, each sized to fit within a single device's memory. Once the data is sharded, it is passed to the JIT-compiled solve function and the JAX compiler automatically manages data partitions and inter-device communication, ensuring efficient memory usage and synchronization.

2.6 Warm start

By default, MPAX initializes with all-zero vectors as the starting point. However, it also supports a warm start with user-provided solutions, which may include a primal solution, a dual solution, or a pair of both. In many machine learning applications, solvers are frequently used to solve a series of similar problem instances where the input data undergoes only minor changes. Although the choice of warm-start strategy is typically problem-dependent, providing a well-suited warm start for both raPDHG and r^{2}HPDHG can significantly accelerate their convergence.

3 Experiments

In this section, we present the empirical results for the Warcraft shortest path problem and multi-dimensional knapsack problem that highlight the efficiency of MPAX. Although both problems are typically formulated as integer linear programming (ILP) problems, prior research [23] indicates that the performance difference between solving the ILP and its LP relaxation is negligible. Therefore, we use the LP relaxation of these two problems for our experiments.

We evaluate the computational efficiency of MPAX by comparing it with state-of-the-art commercial and open-source solvers, including Gurobi, OSQP, and SCS. Specifically, we access OSQP and SCS through the CvxpyQP wrapper in JAXOpt [5]. We use FLAX to construct neural networks and employ MPAX and JAXOpt to solve the resulting LP problems. For Gurobi, we rely on the PyEPO package, which uses PyTorch to build neural networks and Gurobi to solve LP problems. To measure performance, we use the SPO+ loss [9] in the training process and normalized regret as the evaluation metric, defined in Eq. (2).

$$\frac{\sum_{i=1}^{n_{test}} c^\top (x^*(\hat{c_i}) - x^*(c_i))}{\sum_{i=1}^{n_{test}} |x^*(c_i)|} \tag{2}$$

For the raPDHG and r^{2}HPDHG algorithm, we set the restart frequency to 64 iterations and set termination tolerance to ϵ. When warm start is enabled, the average solution from the last iteration is used as the initial starting point. Otherwise, all-zero vectors are used as the initial starting points. Feasibility polishing is disabled during the training process. Since double precision is standard in LP optimization algorithms, we evaluate the performance of raPDHG and r^{2}HPDHG exclusively in the double-precision setting to ensure consistency with common LP solver practices.

In PyEPO, we set `processes` to 16 to enable parallel computing for the forward and backward passes. By default, Gurobi chooses one from the primal simplex, dual simplex, or barrier methods to solve an LP when limited to a single thread. For OSQP and SCS, we use their default settings and run 16 parallel threads to solve LPs.

Computing environment. We use NVIDIA A100-PCIe-80GB GPU, with CUDA 12.4, for running MPAX, FLAX and PyTorch, and use Intel(R) Xeon(R) Silver 4316 CPU @ 2.30GHz with 256GB RAM and 16 threads for running Gurobi 11.0.3, OSQP 0.6.7 and SCS 3.2.7.

3.1 Warcraft Shortest Path

We use the dataset introduced by [18] for the Warcraft shortest path problem. An example of the Warcraft terrain map, vertex cost and shortest path in the dataset is provided in Figure 3 in Appendix B.1. In this task, the objective is to determine the shortest path between the top left and bottom right vertices on a Warcraft terrain map represented as a $k \times k$ 2D grid. The traversal cost of each vertex is unknown and dependent on the terrain type in the map image. The dataset includes Warcraft terrain maps of four different sizes, where $k \in \{12, 18, 24, 30\}$. For each size, the training set contains 10,000 RGB images and the test set includes 1,000 RGB images. In line with [18], we use the first five

layers of ResNet [11], followed by a max-pooling operation, to extract the latent costs for the vertices. The LP formulation of the 2D grid shortest path problem is formulated as Eq.(7) in Appendix B.2 and we use the sparse matrix format when using MPAX. The model is trained over 10 epochs to minimize the SPO+ loss using the Adam optimizer with a learning rate of 10^{-4} and batch size of 70.

The loss and normalized regret are presented in Figure 4 in Appendix B.3. The raPDHG and r^{2}HPDHG algorithms in MPAX achieve performance comparable to Gurobi's default and barrier methods when the optimality tolerance (ϵ) ranges from 10^{-3} to 10^{-6}. Specifically, both the loss and normalized regret curves of r^{2}HPDHG closely align with those of Gurobi across this range. A looser tolerance of $\epsilon = 10^{-2}$ results in slightly higher regret, indicating that $\epsilon = 10^{-3}$ offers the best trade-off between accuracy and computational cost for this problem.

The results for training time per epoch are presented in Table 1. Notably, OSQP fails to complete the training process for all problem sizes, so its results are omitted. Overall, the r^{2}HPDHG implementation with warm starts in MPAX demonstrates the best performance among all tested methods. Notably, the advantage of FLAX+MPAX over FLAX+JAXOpt and PyEPO+Gurobi becomes more pronounced as the problem size increases. For instance, when $k = 30$, FLAX+MPAX requires only half the training time compared to PyEPO+Gurobi. Consistent with the findings in [15], r^{2}HPDHG needs fewer iterations to converge than raPDHG. Furthermore, for both algorithms, employing a warm start accelerates convergence, reducing the number of iterations by approximately 30% and cutting training time by about 25%. This trend is supported by the iteration count distributions shown in Figure 2. Although some instances require more iterations to converge, MPAX remains robust across different problem sizes and optimality tolerances. Despite this variability, the warm start strategy effectively reduces both the average and maximum number of iterations. This alleviates bottlenecks in batch solving and enhances overall efficiency and scalability.

Table 1: Comparison of the training time per epoch for raPDHG and r^{2}HPDHG to converge.

Framework	Algorithm	ϵ	Training time per epoch			
			k=12	k=18	k=24	k=30
FLAX + MPAX	raPDHG	10^{-2}	35.1	64.6	110.0	187.5
		10^{-3}	38.4	72.5	127.3	211.3
		10^{-4}	41.8	83.8	148.6	243.1
		10^{-6}	44.7	92.5	163.4	269.3
FLAX + MPAX	raPDHG (warmstart)	10^{-2}	20.5	44.2	71.0	98.7
		10^{-3}	26.9	54.1	92.2	143.8
		10^{-4}	30.7	64.7	111.3	177.1
		10^{-6}	33.1	70.1	123.7	194.7
FLAX + MPAX	r^{2}HPDHG	10^{-2}	28	53.9	85	138
		10^{-3}	35.1	70.8	111	178.2
		10^{-4}	39.9	78.3	124.8	202.7
		10^{-6}	42	84	131.1	215
FLAX + MPAX	r^{2}HPDHG (warmstart)	10^{-2}	18.7	33.3	55.8	93.5
		10^{-3}	25.8	47.3	79.3	129.2
		10^{-4}	29.6	53.6	88.7	144.9
		10^{-6}	32.2	59.5	94.6	154.8
FLAX + JAXOpt	SCS	10^{-4}	389.4	1024.4	2512.7	5614.6
PyEPO + Gurobi	Default	10^{-6}	25.9	70.7	157.8	313.8

3.2 Multi-dimensional Knapsack

We consider a multi-dimensional knapsack problem with unknown item values, as described in [23, 9]. This problem involves two main tasks: a prediction task to estimate the values of items and an optimization task to maximize the total value of items selected for the knapsack within the capacity limit. The synthetic dataset of item values is generated using the same polynomial function as in [23, 9] with noise half-width changes between 0 and 0.5. The item weights are randomly sampled

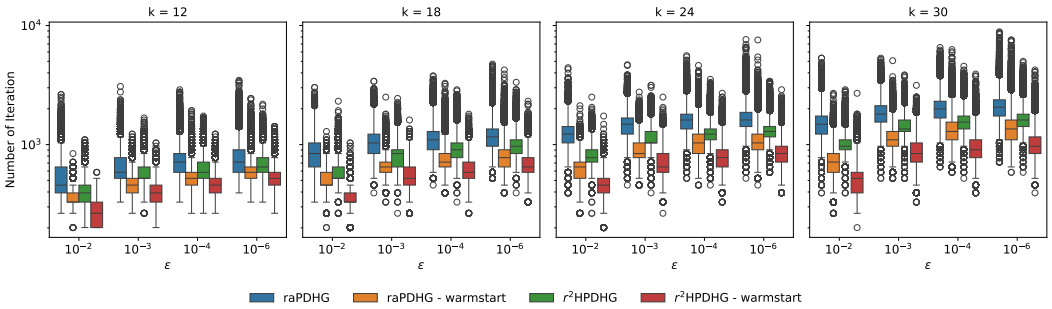

Figure 2: Distribution of the number of iterations for raPDHG and r^{2}HPDHG on the Warcraft shortest path problem.

from 3 to 8 and remain fixed throughout the dataset. To evaluate the robustness of MPAX, we tested the multi-dimensional knapsack problem across various sizes, using instances with 10,000 and 20,000 items. The capacity is set to 500 and the dimension of resources is varied among 10, 20, 50, and 100.

For the prediction model, we employ a four-layer MLP with hidden layers consisting of 32, 128, 512, and 2048 units, respectively. ReLU is used as the activation function for all hidden layers and the output layer. The model is trained using the Adam optimizer with a learning rate of 0.001 for 10 epochs, minimizing the SPO+ loss. Training is performed on a dataset of 4,000 samples and the model's performance was evaluated on a separate test set of 1,000 samples. We set a time limit of 2 hours per epoch for training. The formulation of the LP relaxation for the multi-dimensional knapsack problem is presented in Appendix C.1. In contrast to the Warcraft shortest path problem, the knapsack problem's constraint matrix is fully dense, requiring us to use the dense matrix format when testing MPAX.

The results of training time per epoch for MPAX and Gurobi are presented in Table 3.2 and the normalized regret calculated on the test set is presented in Table 3 in Appendix C.2. Notably, both OSQP and SCS fail to complete one training epoch within the 2-hour time limit, so their results are omitted. These results together demonstrate that MPAX outperforms PyEPO+Gurobi, JAXOpt+SCS, and JAXOpt+OSQP by achieving comparable normalized regret in less training time. In particular, compared to PyEPO+Gurobi, MPAX reduces training time by half for large instances. Moreover, both raPDHG and r^{2}HPDHG in MPAX exhibit robust performance across different problem settings, including variations in the number of items, dimensionality, and noise levels.

Table 2: Training time per epoch for the multi-dimensional knapsack problem. d represents the number of dimensions in the knapsack problem.

Noise half-width	Framework	Algorithm	# of items =10000				# of items =20000			
			d=10	d=20	d=50	d=100	d=10	d=20	d=50	d=100
0	FLAX+MPAX	raPDHG	12.7	21.2	46.7	88.0	21.7	38.3	86.0	186.9
	FLAX+MPAX	r^{2}HPDHG	11.9	24.1	38.7	70.9	20.9	44.1	71.9	146.1
	PyEPO+Gurobi	Default	18.0	23.8	89.9	163.1	31.2	42.8	151.8	300.6
0.5	FLAX+MPAX	raPDHG	12.4	18.7	32.8	58.9	20.4	33.0	64.2	119.2
	FLAX+MPAX	r^{2}HPDHG	12.2	21.6	29.6	51.5	20.6	40.7	58.7	105.2
	PyEPO+Gurobi	Default	18.9	25.6	94.6	119.2	33.2	51.7	159.4	225.5

4 Conclusion

In this paper, we present MPAX, a hardware-accelerated, differentiable, batchable and distributable solver for linear programming in JAX. It is designed to address optimization problems encountered in machine learning workflows and our experiments highlight its superior performance compared to traditional solvers, particularly for large-scale applications. While MPAX currently supports linear programming and quadratic programming, future work will extend its capability to more general mathematical programming problems and specialized modules for common machine-learning tasks.

References

[1] Akshay Agrawal, Brandon Amos, Shane Barratt, Stephen Boyd, Steven Diamond, and J Zico Kolter. Differentiable convex optimization layers. *Advances in neural information processing systems*, 32, 2019.

[2] Brandon Amos, Ivan Jimenez, Jacob Sacks, Byron Boots, and J Zico Kolter. Differentiable mpc for end-to-end planning and control. *Advances in neural information processing systems*, 31, 2018.

[3] Brandon Amos and J Zico Kolter. Optnet: Differentiable optimization as a layer in neural networks. In *International conference on machine learning*, pages 136–145. PMLR, 2017.

[4] David Applegate, Mateo Díaz, Oliver Hinder, Haihao Lu, Miles Lubin, Brendan O'Donoghue, and Warren Schudy. Practical large-scale linear programming using primal-dual hybrid gradient. *Advances in Neural Information Processing Systems*, 34:20243–20257, 2021.

[5] Mathieu Blondel, Quentin Berthet, Marco Cuturi, Roy Frostig, Stephan Hoyer, Felipe Llinares-López, Fabian Pedregosa, and Jean-Philippe Vert. Efficient and modular implicit differentiation. *Advances in neural information processing systems*, 35:5230–5242, 2022.

[6] Mathieu Blondel, Olivier Teboul, Quentin Berthet, and Josip Djolonga. Fast differentiable sorting and ranking. In *International Conference on Machine Learning*, pages 950–959. PMLR, 2020.

[7] Filipe de Avila Belbute-Peres, Kevin Smith, Kelsey Allen, Josh Tenenbaum, and J Zico Kolter. End-to-end differentiable physics for learning and control. *Advances in neural information processing systems*, 31, 2018.

[8] Steven Diamond and Stephen Boyd. CVXPY: A Python-embedded modeling language for convex optimization. *Journal of Machine Learning Research*, 17(83):1–5, 2016.

[9] Adam N Elmachtoub and Paul Grigas. Smart "predict, then optimize". *Management Science*, 68(1):9–26, 2022.

[10] Zhenglin Geng, Daniel Johnson, and Ronald Fedkiw. Coercing machine learning to output physically accurate results. *Journal of Computational Physics*, 406:109099, 2020.

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[12] Haihao Lu, Zedong Peng, and Jinwen Yang. cuPDLPx: A further enhanced gpu-based first-order solver for linear programming. *arXiv preprint arXiv:2507.14051*, 2025.

[13] Haihao Lu and Jinwen Yang. cuPDLP. jl: A gpu implementation of restarted primal-dual hybrid gradient for linear programming in julia. *arXiv preprint arXiv:2311.12180*, 2023.

[14] Haihao Lu and Jinwen Yang. A practical and optimal first-order method for large-scale convex quadratic programming. *arXiv preprint arXiv:2311.07710*, 2023.

[15] Haihao Lu and Jinwen Yang. Restarted halpern pdhg for linear programming. *arXiv preprint arXiv:2407.16144*, 2024.

[16] Haihao Lu, Jinwen Yang, Haodong Hu, Qi Huangfu, Jinsong Liu, Tianhao Liu, Yinyu Ye, Chuwen Zhang, and Dongdong Ge. cuPDLP-C: A strengthened implementation of cupdlp for linear programming by c language. *arXiv preprint arXiv:2312.14832*, 2023.

[17] Thomas Pock and Antonin Chambolle. Diagonal preconditioning for first order primal-dual algorithms in convex optimization. In *2011 International Conference on Computer Vision*, pages 1762–1769. IEEE, 2011.

[18] Marin Vlastelica Pogančić, Anselm Paulus, Vit Musil, Georg Martius, and Michal Rolinek. Differentiation of blackbox combinatorial solvers. In *International Conference on Learning Representations*, 2020.

[19] Daniel Ruiz. A scaling algorithm to equilibrate both rows and columns norms in matrices. Technical report, CM-P00040415, 2001.

[20] Utsav Sadana, Abhilash Chenreddy, Erick Delage, Alexandre Forel, Emma Frejinger, and Thibaut Vidal. A survey of contextual optimization methods for decision-making under uncertainty. *European Journal of Operational Research*, 2024.

[21] M. Schubiger, G. Banjac, and J. Lygeros. GPU acceleration of ADMM for large-scale quadratic programming. *Journal of Parallel and Distributed Computing*, 144:55–67, 2020.

[22] B. Stellato, G. Banjac, P. Goulart, A. Bemporad, and S. Boyd. OSQP: an operator splitting solver for quadratic programs. *Mathematical Programming Computation*, 12(4):637–672, 2020.

[23] Bo Tang and Elias B Khalil. Pyepo: A pytorch-based end-to-end predict-then-optimize library for linear and integer programming. *Mathematical Programming Computation*, 16(3):297–335, 2024.

[24] Kevin Tracy and Zachary Manchester. On the differentiability of the primal-dual interior-point method, 2024.

A Algorithms

Both raPDHG and r^{2}HPDHG solve (LP) and (QP) problems by addressing their primal-dual form:

$$\min_{x\in X}\max_{y\in Y} L(x,y) := c^\top x - y^\top K x + q^\top y\,, \tag{3}$$

$$\min_{x\in X}\max_{y\in Y} L(x,y) := \frac{1}{2}x^\top Q x + c^\top x - y^\top K x + q^\top y\,, \tag{4}$$

where $K^\top = \left(G^\top, A^\top\right)$ and $q^\top := \left(h^\top, b^\top\right)$, $X := \{x \in \mathbb{R}^n : l \le x \le u\}$, and $Y := \{y \in \mathbb{R}^{m_1+m_2} : y_{1:m_1} \ge 0\}$.

A.1 raPDHG

Let $z = (x, y)$ represent the primal-dual pair, and let $z^{k+1} = \mathrm{PDHG}(z^k)$ denote a single PDHG iteration, defined by the following update rule:

$$\begin{aligned} x^{k+1} &\leftarrow \mathrm{proj}_X(x^k - \tau(c - K^\top y^k)) \\ y^{k+1} &\leftarrow \mathrm{proj}_Y(y^k + \sigma(q - K(2x^{k+1} - x^k)))\,, \end{aligned} \tag{5}$$

However, the performance of the vanilla PDHG method for solving LPs is insufficient for general-purpose solvers [4]. To address this, several algorithmic enhancements have been proposed, including an averaging and restart scheme, adaptive step sizes, primal weighting, and preconditioning [4]. The PDHG method augmented with the averaging and restart scheme is referred to as raPDHG.

A.2 r^{2}HPDHG

The update rule for Halpern PDHG with reflection is given by:

$$z^{k+1} := \frac{k+1}{k+2}(2\cdot \mathrm{PDHG}(z^k) - z^k) + \frac{1}{k+2}z^0\,, \tag{6}$$

In this formulation, the next iterate is a weighted average of a reflected PDHG step $(2\cdot\mathrm{PDHG}(z^k)-z^k)$ and the initial solution z^0. The restarted version of Halpern PDHG with reflection, denoted as r^{2}HPDHG, offers strong theoretical guarantees and achieves significant speedups over raPDHG [15]. The detailed steps of r^{2}HPDHG is presented in Algorithm 2.

Algorithm 1: Restarted average PDHG (raPDHG)

Input: Initial point $z^{0,0}$; outer counter $n \leftarrow 0$; total iterations $k \leftarrow 0$; primal weight $\omega^0 \leftarrow \text{INITIALIZEPRIMALWEIGHT}(c, q)$

repeat
  $t \leftarrow 0$;
  repeat
    $z^{n,t+1} \leftarrow \text{ADAPTIVESTEPPDHG}(z^{n,t}, \omega^n)$;
    $\bar{z}^{n,t+1} \leftarrow \text{WEIGHTEDAVERAGE}(z^{n,0}, ..., z^{n,t+1})$;
    $z_c^{n,t+1} \leftarrow \text{GETRESTARTCANDIDATE}(z^{n,t+1}, \bar{z}^{n,t+1})$;
    $t \leftarrow t + 1$;
    $k \leftarrow k + 1$;
  until restart or termination criterion holds;
  $z^{n+1,0} \leftarrow z_c^{n,t}$;
  $n \leftarrow n + 1$;
  $\omega^n \leftarrow \text{PRIMALWEIGHTUPDATE}(z^{n,0}, z^{n-1,0}, \omega^{n-1})$;
until termination criterion holds;

Algorithm 2: Reflected restarted Halpern PDHG (r^{2}HPDHG)

Input : Initial point $z^{0,0}$, outer loop counter $n \leftarrow 0$, inner loop counter $k \leftarrow 0$.

repeat
  initialize the inner loop counter $k \leftarrow 0$;
  repeat
    $z^{n,k+1} \leftarrow$ reflected-H-PDHG$(z^{n,k}; z^{n,0})$;
    $k \leftarrow k + 1$;
  until restart condition holds;
  initialize the initial solution $z^{n+1,0} \leftarrow \text{PDHG}(z^{n,k})$;
  $n \leftarrow n + 1$;
until $z^{n+1,0}$ convergence;

B Warcraft Shortest Path Problem

B.1 An example of the Warcraft terrain map, vertex cost and shortest path

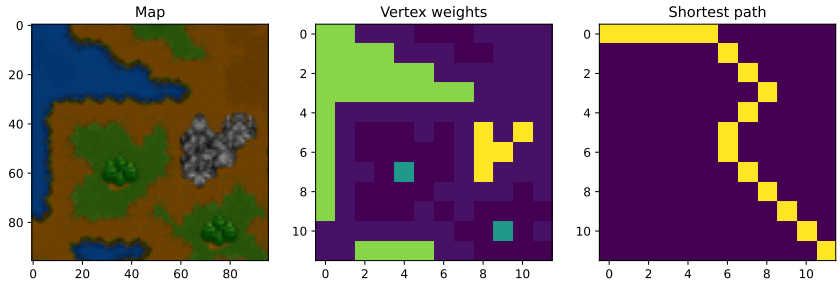

Figure 3: Warcraft shortest path dataset: Warcraft terrain map (left), vertex traversal cost (middle) and shortest path (right)

B.2 LP relaxation of the Warcraft shortest path problem

The LP relaxation of the Warcraft shortest path problem is formulated as follows.

$$
\begin{aligned}
\min \quad & \sum_{(i,j)\in E} c_{i,j} x_{i,j} \\
\text{s.t.} \quad & \sum_{j:(s,j)\in E} x_{s,j} - \sum_{i:(i,s)\in E} x_{i,s} = 1 \\
& \sum_{j:(t,j)\in E} x_{t,j} - \sum_{i:(i,t)\in E} x_{i,t} = -1 \\
& \sum_{j:(v,j)\in E} x_{v,j} - \sum_{i:(i,v)\in E} x_{i,v} = 0, \quad \forall v \in V \setminus \{s,t\} \\
& 0 \le x_{i,j} \le 1, \quad \forall (i,j) \in E.
\end{aligned}
\tag{7}
$$

where V is the set of nodes, s is the source node, t is the sink node, E is the set of edges, $c_{i,j}$ is the cost associated with travelling along edge (i,j), and x is the flow along edge (i,j).

B.3 The loss and normalized regret curve of the Warcraft shortest path problem

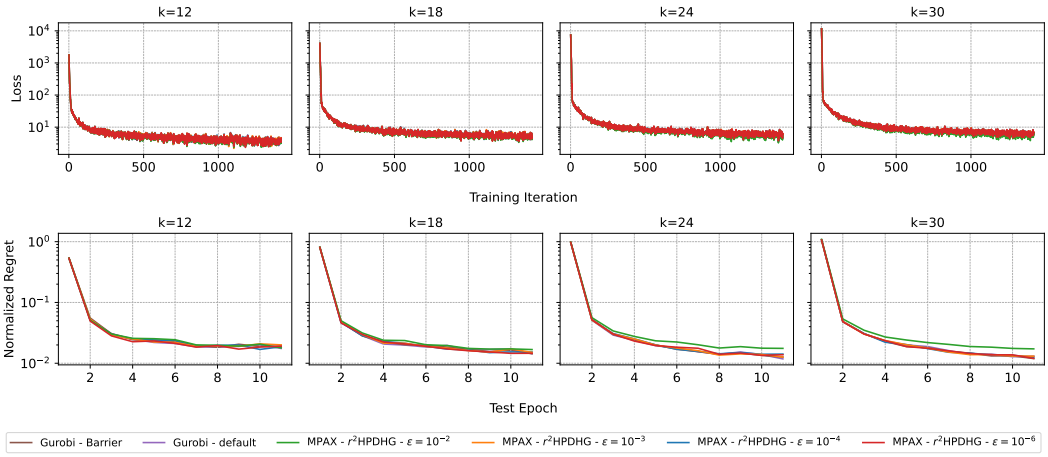

Figure 4: The loss and normalized regret curve of the Warcraft shortest path problem

C Multi-dimensional knapsack problem

C.1 LP relaxation of the multi-dimensional knapsack problem

The formulation of the LP relaxation for the multi-dimensional knapsack problem is presented as Eq. (8).

$$
\begin{aligned}
\max \quad & \sum_{i=1}^{N} c_i x_i \\
\text{s.t.} \quad & \sum_{i=1}^{N} w_{i,j} x_i \le h_j, \quad \forall j = 1, \ldots, d, \quad 0 \le x_i \le 1, \quad \forall i = 1, \ldots, N.
\end{aligned}
\tag{8}
$$

where x_i is the decision variable denoting whether to include the item, c_i is the value of each item, $w_{i,j}$ is the weight of each item, h_j is the capacity of the knapsack, N is the number of items and d is the number of dimensions.

C.2 Normalized regret for the multi-dimensional knapsack problem on the test set

Table 3: Normalized regret for the multi-dimensional knapsack problem on the test set. d represents the number of dimensions in the knapsack problem.

Noise half-width	Framework	Algorithm	# of items =10000				# of items =20000			
			d=10	d=20	d=50	d=100	d=10	d=20	d=50	d=100
0	FLAX+MPAX	raPDHG	1.2%	0.7%	0.8%	0.5%	0.9%	0.8%	0.7%	0.6%
	FLAX+MPAX	r^{2}HPDHG	1.1%	0.7%	0.8%	0.5%	0.9%	0.8%	0.7%	0.6%
	PyEPO+Gurobi	Default	0.9%	1.2%	0.7%	0.8%	1.0%	0.8%	0.8%	1.0%
0.5	FLAX+MPAX	raPDHG	22.4%	23.0%	24.6%	26.1%	23.3%	25.9%	27.5%	26.9%
	FLAX+MPAX	r^{2}HPDHG	22.4%	23.0%	24.6%	26.3%	23.5%	26.3%	27.3%	27.3%
	PyEPO+Gurobi	Default	21.9%	23.4%	24.6%	25.3%	24.0%	25.6%	27.0%	27.9%