Assignment Report #5
Linear Regression

Student Name: Dias Suleimenov (ID: 202158836)
Course: PHYS 421 Parallel Computing

Submitted: November 3, 2025

Software packages used:

GCC 13.3.0	C/C++ compilation
NVCC 13.3.0	CUDA compilation
Python 3.12.3	Data generation, analysis, plotting

AI tools used:

Gemini 2.5 Pro	Code generation, script writing
Copilot	Code generation, script writing, report assistance

Honor Statement

I affirm that this work complies with Nazarbayev University academic integrity policies and the policies regarding the use of AI tools outlined in the course syllabus

1 Methodology

1.1 Generation of Data

The synthetic datasets for linear regression benchmarks were generated using the following procedure.

Random seed = 42 was fixed. $D = 32$ was chosen and $N \in \{3\times10^4, 10^5, 3\times10^5, 10^6, 10^5, 3\times10^6\}$ was took. Let $N_{\max}$ be the largest. Then largest dataset with $N_{\max}$ samples was generated and smaller datasets were obtained by subsampling.

The steps of generation as follows:

- $X_{\max}[i, j] \sim \mathcal{N}(0, 1)$.
- $\beta_\star \sim \mathcal{N}(0, 1)$.
- Added Gaussian noise $\varepsilon \sim \mathcal{N}(0, \sigma^2)$ and set $y_{\max} = X_{\max}\beta_\star + \varepsilon$.
- Scaling was applied to zero-mean and unit-variance each column of $X_{\max}$.

- For each smaller N, sampled row indices S_N uniformly without replacement and set $X_N = X_{\max}[S_N, :]$, $y_N = y_{\max}[S_N]$.
- Each subsamples was saved for reproducibility.

With $\sigma = 1.9635$, on hold-out $R^2 \approx 0.9$ was achieved.

1.2 Measuring Performance

For performance measurements, 3 warmup runs were performed before timing. Then, 10 timing runs were conducted, and the average and standard deviation were reported. For CPU time measurement, `std::chrono::steady_clock` was used. For GPU end-to-end (E2E) time measurement, `std::chrono::steady_clock was used`, starting before data transfer to the device and ending after data transfer back to the host. For GPU kernel time measurement, CUDA events were used, starting before kernel launch and ending after kernel completion.

1.3 Vector Benchmark

The following parameteres were chosen for benchmarking DOT and SAXPY operations, $N = 10^3, \ldots, 3 \times 10^6$ with increment of 10^6; $D = 32$.

1.3.1 DOT and SAXPY kernel implementations

SAXPY Kernel was launched with 256 threads per block and $\lceil N/256 \rceil$ blocks to cover all data. Each thread computed the SAXPY operation for its assigned elements.

DOT Kernel was launched with 256 threads per block and $\lceil N/256 \rceil$ blocks to cover all data. Each thread computed partial sum for its assigned elements and stored it in shared memory. A parallel reduction was performed within each block to compute the block-level sum, which was then written to global memory. Finally, a second kernel was launched to sum the block-level results to obtain the final dot product.

The CPU implementation for both SAXPY and DOT used a simple for loop to iterate over the elements of the vectors and perform the respective operations.

1.4 Linear Regression

The FBGD algorithm worked in the following way on each iteration:

1. Prediction: $\hat{y} \leftarrow X\beta$
2. Residual: $r \leftarrow \hat{y} - y$
3. Gradient: $g \leftarrow X^\top r$
4. Update: $\beta \leftarrow \beta - \eta g$

With stopping criteria: $|g|_2/|g_0|_2 < 10^{-4}$ or after a maximum number of iterations = 1000. The relative gradient norm was checked every 100 iterations to reduce overhead. In GPU

implementation, pinned memory was used to transfer the gradient norm from device to host for checking the stopping criteria, without use of `cudaMemcpy` calls.

The learning rate $\eta = 10^{-6}$ was chosen after preliminary experiments on small datasets to ensure convergence. The small learning rate was necessary due to the exploding gradients observed with larger η values.

All arrays were kept resident on the GPU between iterations to minimize data transfer overhead. Transposed matrix vector multiplication $X^\mathsf{T} r$ was implemented using a custom kernel that used coalesced memory accesses and shared memory tiling for partial sums to optimize performance. The matrix-vector multiplication $X\beta$ was implemented using a similar approach. The `__reduce_add_sync` single-instruction warp-wide reduction was used for efficient reduction operations in $X^\mathsf{T} r$ and $X\beta$ implementations.

1.4.1 GFLOPS calculation

For FBGD, it was assumed that each iteration involves $4ND$ floating-point operations (FLOPs): $2ND$ for the matrix-vector multiplication $X\beta$, $2ND$ for the matrix-vector multiplication $X^\mathsf{T} r$, and negligible FLOPs for vector additions and scalar multiplications. The GFLOPS was calculated as:

$$\text{GFLOPS} = \frac{4ND \times \text{Number of Iterations}}{\text{Total Time (seconds)} \times 10^9}$$

.

1.5 Real-Data Application

Dataset from `sklearn.datasets` was used. Features were standardized to zero mean and unit variance. The dataset was split into training (80%) and testing (20%) sets. The bias trick was applied by adding a column of ones to the feature matrix. Model was trained using FBGD on GPU on training set with $\eta = 5 \times 10^{-5}$ and maximum iterations = 10000. Model performance was evaluated on testing set using R^2 score and Root Mean Squared Error (RMSE). The results from GPU FBGD were compared with CPU serial implementation and `sklearn`'s `LinearRegression`. Resulting training set featured D = 9, N = 16512 samples.

1.6 Correctness & Fairness

All implementations had the same update rule, the same stepsize. The same float32 datatype. The same datasets and preprocessing steps were used across all implementations to ensure a fair comparison. Performance measurements were conducted under similar conditions, with multiple runs to account for variability.

The GPU FBGD implementations were validated against CPU implementation and ground truth β_*. With tolerance $|\beta_{\text{GPU}} - \beta_{\text{CPU}}|_\infty < 2 \times 10^{-7}$ for N = 30000, D=32 and $|\beta_{\text{GPU}} - \beta_*|_\infty < 5 \times 10^{-2}$.

1.7 Environment & Reproducibility

Experiments were conducted on a HPC node with the following specifications:

- CPU: AMD EPYC 9654 @ 2.30GHz
- GPU: Nvidia V100 GPU (32 GB HBM2)
- RAM: 256 GB DDR4-2933 RAM (8-channel)
- OS: Rocky Linux 8.10
- Compiler: GCC 13.3.0 with -O3 optimization
- Compiler: nvcc 13.3.0 with -O3 optimization
- CUDA Toolkit: 12.8.0
- Python 3.12.3 with NumPy 2.3.4 and scikit-learn 1.7.2 for data generation and analysis

The cluster used was Slurm NVIDIA partition of Shabyt cluster with the following flags:

- `--gres=gpu:1`
- `--cpus-per-task=1`

Seed = 42 was used for all random number generation to ensure reproducibility. For data generation, the script `scripts/generate_data.py` was used. The main benchmarking code is in `src/part1.cu`. Dataset for real data application was generated by `scripts/regression.py` and the main code for real data application `src/par2.cu` was used.

In order to reproduce the results, follow these steps in a terminal:

```
# 0. Set up environment with specified compiler versions and libraries
python -m venv .venv
source .venv/bin/activate
pip install -r requirements.txt

# 0.5 Load necessary modules (if using a cluster with module system)
module load GCCcore/13.3.0
module load CUDA/12.8.0
module load Python/3.12.3-GCCcore-13.3.0

# 1. Generate synthetic datasets
python scripts/generate_data.py
python scripts/regression.py

# 2. Compile the CUDA code
make all

# 3. Run benchmarks for individual parts locally (optional)
./bin/part0        # Vector benchmark
./bin/fbgd         # FBGD benchmark
./bin/part2        # Real-data application
```

```
22
23 # 3.5 To run benchmarks for all parts on the cluster
24 sbatch scripts/run_part0.slurm # Vector benchmark
25 sbatch scripts/run_fbgd.slurm  # FBGD benchmark
26 sbatch scripts/run_part2.slurm # Real-data application
27
28 # 4. Generate plots from the results
29 python scripts/generate_plots.py
```

2 Results

2.1 Vector Benchmark

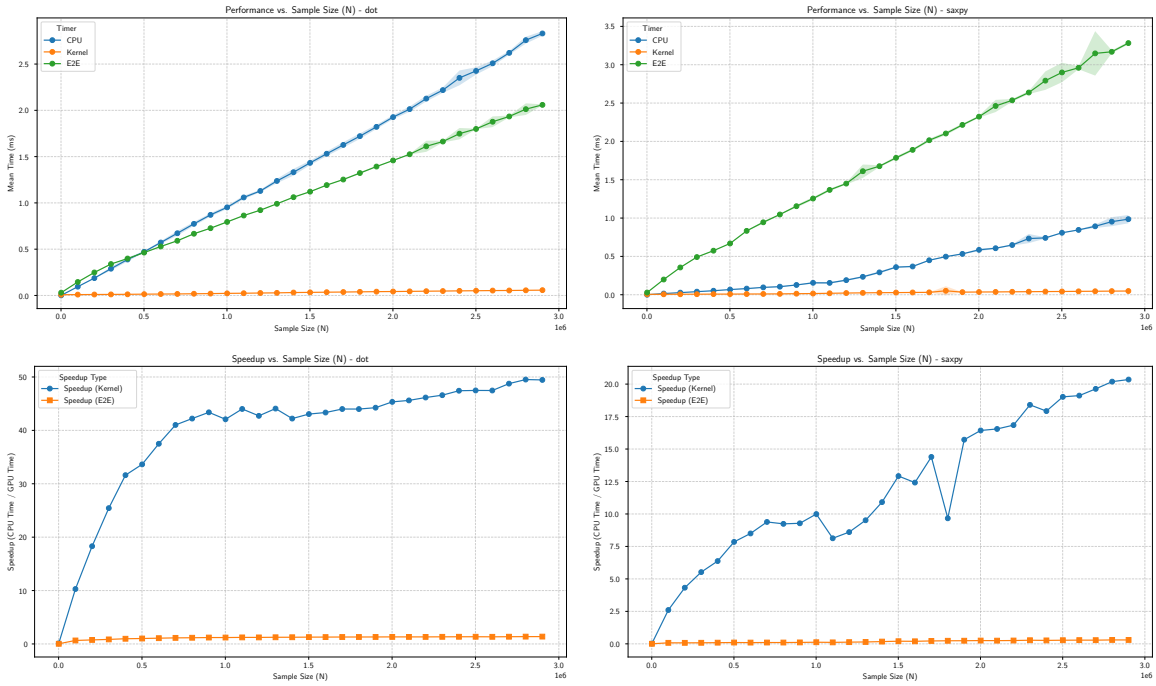

Figure 1: Performance comparison of CPU and GPU implementations for DOT and SAXPY operations across varying vector sizes N. The plots show execution time (ms) and GFLOPS achieved for each operation.

2.2 Linear Regression Benchmark

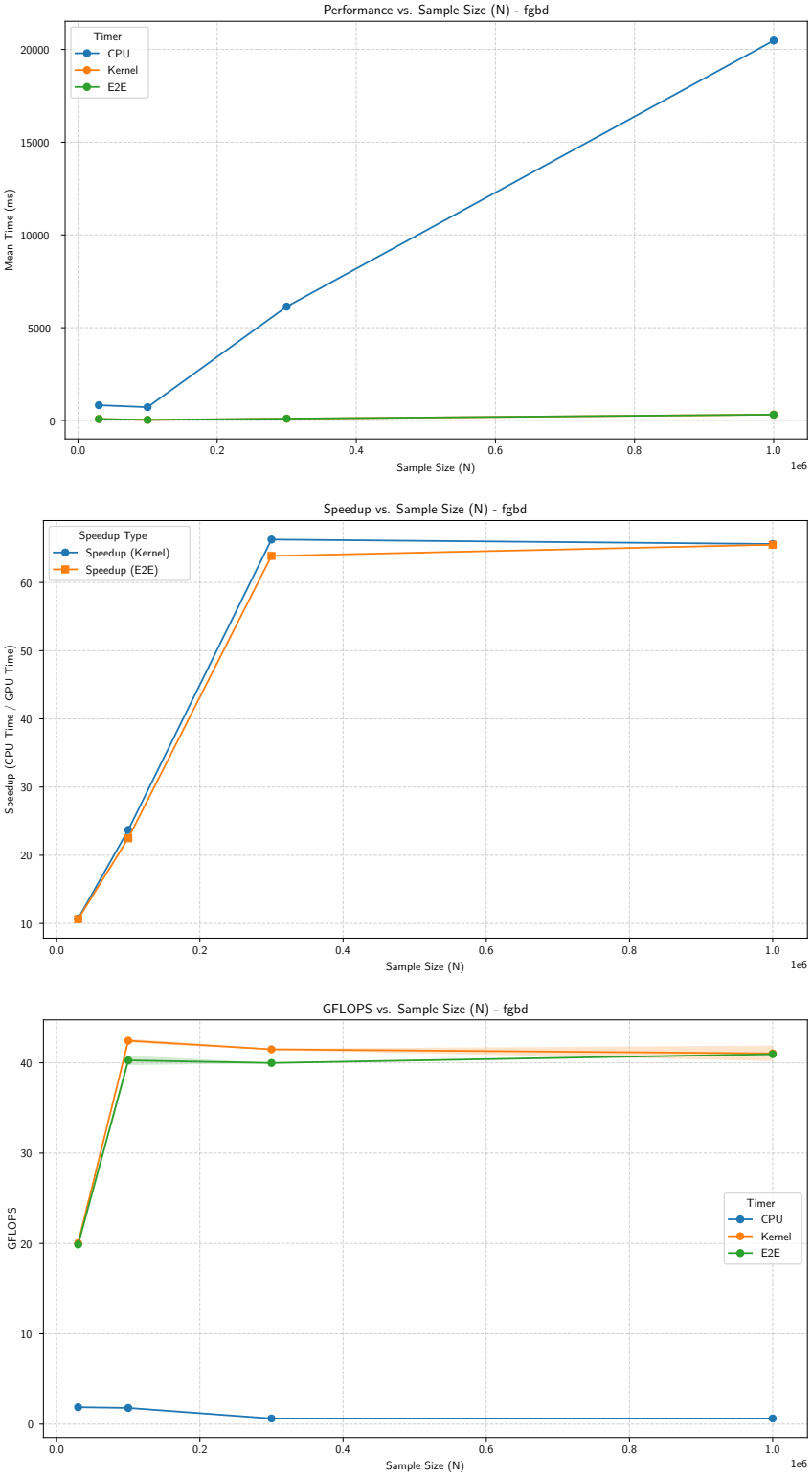

Figure 2: Performance comparison of CPU and GPU implementations for FBGD across varying dataset sizes N with fixed feature dimension $D = 32$. The plots show execution time (ms) and GFLOPS achieved for each implementation.

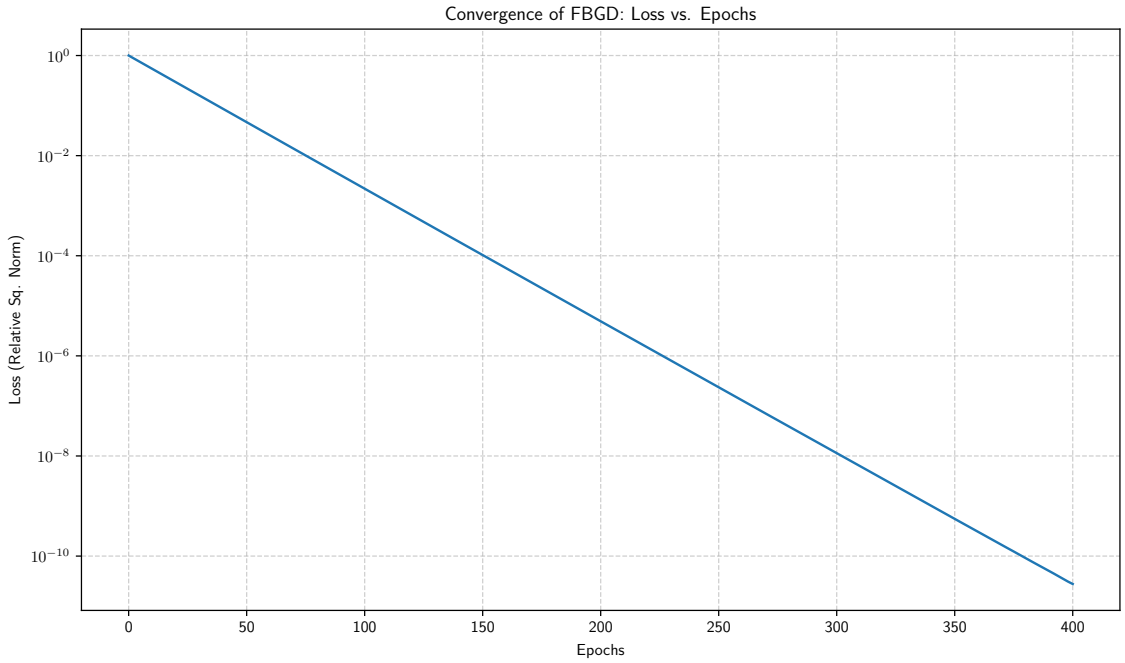

Figure 3: Convergence behavior of FBGD on GPU for dataset with $N = 30000$ with fixed feature dimension $D = 32$. The plot shows the relative gradient norm over iterations, indicating convergence to the optimal solution.

3 Real-Data Application

Implementation	Time	Iterations	R^2 Score	RMSE
CPU Serial	39.494 ms	200	0.576	0.746
GPU Kernel-Only	18.079 ms	200	-	-
GPU End-to-End	18.166 ms	200	-	-
`sklearn` LinearRegression	2.223 ms	-	0.576	0.746

Table 1: Performance comparison of different implementations on the California housing dataset. The table shows the R^2 score and Root Mean Squared Error (RMSE) for implementation on the testing set.

With difference between CPU and GPU implementations and sklearn LinearRegression within tolerance of 3×10^{-2}.

4 Discussion

5 Vector Benchmark

The vector benchmark results in Fig. 1 show the performance comparison between CPU and GPU implementations for DOT and SAXPY operations across varying vector sizes N. For small vector sizes, the CPU outperforms the GPU (E2E) due to the overhead associated with

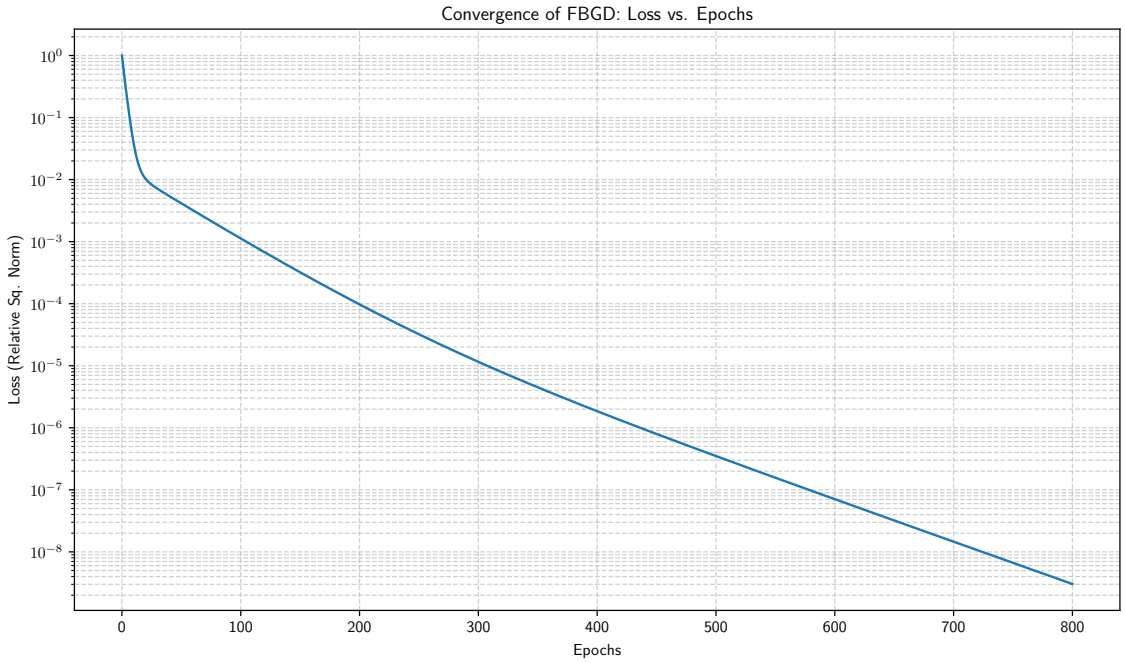

Figure 4: Convergence behavior of FBGD on GPU for California housing dataset with $N = 16512$ samples and $D = 9$ features. The plot shows the relative gradient norm over iterations, indicating convergence to the optimal solution.

data transfer and kernel launch on the GPU. As N increases, the GPU's parallel processing capabilities allow it to outperform the CPU, leading to significant speedups. The gap between kernel-only and end-to-end performance highlights the impact of data transfer overheads, which become less significant as the computation time increases with larger N. But as operations of DOT and SAXPY are relatively fast, the overheads remain significant even at larger N. The maximum speedup achieved was around 20x for SAXPY operation and 50x for DOT operation at the largest vector sizes tested. The fluctuation in saxpy graph, may be due to variability in GPU load or other system factors during benchmarking.

6 Linear Regression Benchmark

From the Fig. 2, it can be seen that the GPU overtakes the CPU in FBGD performance as N increases due to the higher arithmetic intensity of the matrix-vector multiplications involved in FBGD. The GPU's architecture is well-suited for handling large-scale parallel computations, allowing it to efficiently process the large datasets used in linear regression. Coalesced memory access patterns and shared memory optimizations further enhance the GPU's performance. The gap between GPU end-to-end and kernel-only performance is primarily due to data transfer overheads between the host and device. However, as the dataset size increases, these overheads become less significant relative to the computation time, leading to improved overall performance. The maximum speedup achieved was around 65x at the largest dataset size tested. And 42 GFLOPS was achieved at $N = 3 \times 10^6$.

Also it can be noted from Fig. 3 that convergence behavior of FBGD on GPU for dataset with

$N = 30000$ with fixed feature dimension $D = 32$ shows steady decrease in relative gradient norm over iterations, indicating effective convergence to the optimal solution.

7 Real-Data Application

The real-data application results in Table 1 demonstrate the performance of different implementations on the California housing dataset. The GPU implementations (both kernel-only and end-to-end) significantly outperform the CPU serial implementation, achieving speedups of over 2x. The `sklearn` LinearRegression implementation is the fastest due to its highly optimized algorithms and libraries, having execution time 9x faster than self-written. Also the performance of NumPy library can be explained by relatively small dataset size, where the overhead of GPU data transfer and kernel execution outweighs the benefits of parallel computation.

The convergence plot for the California housing dataset shows that the FBGD algorithm effectively converges to the optimal solution within 200 iterations. The algorithm converged in 200 iterations with learning rate 5×10^{-5}, well before the specified maximum cap of 10000, with the relative gradient norm decreasing steadily over iterations. The results from the GPU implementations closely match those from the CPU implementation and `sklearn` LinearRegression, with differences within a tolerance of 3×10^{-2}. During the testing it was noted, that larger stepsizes led to exploding gradients and divergence, highlighting the importance of careful stepsize selection for stability. In comparison the FBGD on synthetic dataset with N=30000, D=32 was stable with higher stepsize of 6×10^{-5}, indicating that real-world datasets may require more accurate tuning.

8 Conclusion

In this report, the performance of CPU and GPU implementations for vector operations (DOT and SAXPY) and linear regression using FBGD was benchmarked. The GPU implementations demonstrated significant speedups over the CPU as dataset sizes increased, showcasing the advantages of parallel processing for large-scale computations. The real-data application on the California housing dataset further validated the effectiveness of the GPU implementations, achieving comparable accuracy to established libraries like `sklearn` LinearRegression. The following speedup were achieved:

- Up to 50x speedup for DOT operation at largest vector sizes.
- Up to 20x speedup for SAXPY operation at largest vector sizes.
- Up to 65x speedup for FBGD at largest dataset size.
- Over 2x speedup for real-data application on California housing dataset.
- 42 GFLOPS achieved for FBGD at $N = 3 \times 10^6$.

Also, the importance of careful stepsize selection for stability in FBGD was highlighted, with the chosen stepsize leading to effective convergence across both synthetic and real datasets.