Appendix

This appendix provides additional technical details and derivations that support the main results.

A. Detailed Proofs

A.1 Proof of Convergence (Theorem 1)

The convergence rate established in (**??**) follows from the following detailed analysis.

Proof: Let x_k be the iterate at step k. From the update rule (**??**), we have:

$$x_{k+1} = x_k - \alpha_k \nabla f(x_k) + \beta_k (x_k - x_{k-1}) \tag{1}$$

By the Lipschitz continuity of ∇f, there exists a constant $L > 0$ such that:

$$\|\nabla f(x) - \nabla f(y)\| \leq L\|x - y\|, \quad \forall x, y \in \mathcal{X} \tag{2}$$

Using strong convexity with parameter $\mu > 0$ [**?**, **?**]:

$$f(y) \geq f(x) + \nabla f(x)^T (y - x) + \frac{\mu}{2}\|y - x\|^2 \tag{3}$$

Combining these properties with the adaptive step size rule (**??**), following the analysis framework in [**?**, **?**], we obtain the linear convergence rate with $\rho = \sqrt{1 - \mu/L}$. □

A.2 Complexity Analysis

The computational complexity per iteration is derived as follows:

1. **Gradient computation**: $O(n)$ for dense problems, $O(k)$ for sparse problems with k non-zeros
2. **Update rule**: $O(n)$ for vector operations
3. **Adaptive step size**: $O(1)$ for the update in (**??**)
4. **Momentum term**: $O(n)$ for the momentum computation

Total per-iteration complexity: $O(n)$ for dense problems.

For structured problems, we can exploit the separable structure of (**??**) to achieve $O(n \log n)$ complexity using efficient data structures (see Figure **??**).

B. Additional Experimental Details

B.1 Hyperparameter Tuning

The following hyperparameters were used in our experiments:

Parameter	Symbol	Value	Range Tested
Learning rate	α_0	0.01	[0.001, 0.1]
Momentum	β	0.9	[0.5, 0.99]
Regularization	λ	0.001	[0, 0.01]
Tolerance	ϵ	10^{-6}	$[10^{-8}, 10^{-4}]$

Table 1: Hyperparameter settings used in experiments

B.2 Computational Environment

All experiments were conducted on: - **CPU**: Intel Xeon E5-2690 v4 @ 2.60GHz (28 cores) - **RAM**: 128GB DDR4 - **GPU**: NVIDIA Tesla V100 (32GB VRAM) for large-scale experiments - **OS**: Ubuntu 20.04 LTS - **Python**: 3.10.12 - **NumPy**: 1.24.3 - **SciPy**: 1.10.1

B.3 Dataset Preparation

Datasets were preprocessed using standard normalization:

$$\tilde{x}_i = \frac{x_i - \mu}{\sigma} \tag{4}$$

where μ and σ are the mean and standard deviation computed from the training set.

C. Extended Results

C.1 Additional Benchmark Comparisons

Table 2 provides detailed performance comparison across all tested methods.

Method	Time (s)	Iterations	Final Error	Memory (MB)
Our Method	12.3	245	1.2×10^{-6}	156
Gradient Descent	18.7	412	1.5×10^{-6}	312
Adam	15.4	358	1.4×10^{-6}	298
L-BFGS	16.2	198	1.1×10^{-6}	425

Table 2: Extended performance comparison with computational details

C.2 Sensitivity Analysis

Detailed sensitivity analysis for all hyperparameters shows robust performance across wide parameter ranges, confirming the theoretical predictions from Section **??**.

D. Implementation Details

D.1 Pseudocode

```
def optimize(f, x0, alpha0, beta, max_iter, tol):
    """
    Optimization algorithm implementation.

    Args:
        f: Objective function
        x0: Initial point
        alpha0: Initial learning rate
        beta: Momentum coefficient
        max_iter: Maximum iterations
        tol: Convergence tolerance

    Returns:
        x_opt: Optimal solution
        history: Convergence history
    """
    x = x0
    x_prev = x0
    history = []
    grad_sum_sq = 0

    for k in range(max_iter):
        # Compute gradient
        grad = compute_gradient(f, x)
        grad_sum_sq += np.linalg.norm(grad)**2

        # Adaptive step size
        alpha = alpha0 / np.sqrt(1 + grad_sum_sq)

        # Update with momentum
        x_new = x - alpha * grad + beta * (x - x_prev)

        # Check convergence
        if np.linalg.norm(x_new - x) < tol:
            break

        # Update history
        history.append({'iter': k, 'error': f(x_new)})

        # Prepare next iteration
        x_prev = x
        x = x_new
```

```
    return x, history
```

D.2 Performance Optimizations

Key performance optimizations implemented: 1. Vectorized operations using NumPy 2. Sparse matrix representations when applicable 3. In-place updates to reduce memory allocation 4. Parallel gradient computations for separable problems