Optimization Techniques for Linear Systems—From Gradient Descent to Adaptive Solvers

Kiara Gholizad[1]

[1]Department of Physics, Sharif University of Technology, Tehran, Iran

March 11, 2025

Contents

1 Define the Problem

Solving a linear system $AX = B$ by direct methods (like computing $A^{-1}B$ or Gaussian elimination) can be problematic when A is singular or ill-conditioned:

- **Singular matrices:** If A is singular (non-invertible), the linear system has no unique solution. There may be either no solution or infinitely many solutions. In such cases, attempting a direct solve (e.g. computing A^{-1}) fails because A^{-1} does not exist. For example, the 2×2 system

  $$\begin{pmatrix} 1 & 2 \\ 2 & 4 \end{pmatrix} x = \begin{pmatrix} 5 \\ 10 \end{pmatrix}$$

  is singular (second row is twice the first). A direct solver will flag a "singular matrix" error, even though there are infinitely many solutions in this case.

- **Ill-conditioned matrices:** An *ill-conditioned* matrix is one that is almost singular – a small change in the coefficients or right-hand side causes a large change in the solution. **The condition number** $\kappa(A)$ quantifies this sensitivity. If $\kappa(A)$ is large, the system is unstable: small rounding errors or perturbations in A or B lead to disproportionately large errors in X. In other words, for ill-conditioned matrices, "a small perturbation of the entries can lead to large changes in the solution of the linear system".

Impact on solution stability: When A is ill-conditioned, numerical solutions become unreliable. We expect to lose roughly $\log_{10} \kappa(A)$ digits of accuracy in the solution. For example, a Hilbert matrix of size 12 (a classic ill-conditioned matrix with $\kappa \sim 10^{16}$) loses about 16 digits of precision. In a test, solving a 12×12 Hilbert system via standard Gaussian elimination yielded a solution with a **43% relative error**, whereas using a more stable SVD-based method gave only $\sim 0.3\%$ error (much closer to the true solution). This dramatic difference illustrates that naive solving of ill-conditioned systems can produce wildly inaccurate results, even though the equations are consistent.

When A is singular or nearly singular, the "solution" $X = A^{-1}B$ either doesn't exist or is **meaningless** due to amplified noise. In such *ill-posed* problems, special techniques are needed to get a stable, meaningful solution. Optimization methods, as discussed next, provide an alternative approach that can handle these issues by framing $AX = B$ as a minimization problem rather than directly inverting A.

2 Optimization Methods for Solving Linear Systems

Instead of direct elimination, we can **reformulate the linear system as an optimization problem.**

Consider the least-squares objective:

$$f(X) = \frac{1}{2}\|AX - B\|_F^2, \tag{1}$$

which measures the squared error. The global minimum of $f(X)$ occurs when:

$$AX = B, \tag{2}$$

for a consistent system. Gradient-based algorithms can iteratively minimize this objective to solve for X. The gradient is given by:

$$\nabla f = A^T(AX - B). \tag{3}$$

Setting $\nabla f = 0$ gives the normal equations:

$$A^T AX = A^T B, \tag{4}$$

whose solution is the least-squares solution. Gradient descent avoids explicitly solving these equations; instead, it updates X iteratively in the direction of the steepest descent:

$$X_{k+1} = X_k - \alpha \nabla f, \tag{5}$$

where α is the learning rate.

Below, we outline and implement several gradient descent variants for solving $AX = B$, along with their characteristics and tuning considerations.

2.1 Vanilla Gradient Descent (Steepest Descent)

Method: Start with an initial guess $X^{(0)}$ (e.g. zero matrix). Then iteratively update:

$$X^{(k+1)} = X^{(k)} - \alpha A^T(AX^{(k)} - B), \tag{6}$$

where $\alpha > 0$ is the **learning rate** (step size). This is the basic gradient descent step moving opposite to the gradient $A^T(AX^{(k)} - B)$. Intuitively, $AX^{(k)} - B$ is the current residual, and A^T propagates that error back to adjust the solution.

Advantages: Simple to implement and requires only matrix-vector multiplications (no matrix factorization). It's memory-efficient for large systems since we don't need to store or invert A.

Disadvantages: Convergence can be **very slow**, especially if A is ill-conditioned. The optimal choice of learning rate α can be tricky: too small α leads to slow progress, too large causes divergence. In fact, convergence requires:

$$\alpha < \frac{2}{\lambda_{\max}(A^T A)}, \tag{7}$$

where $\lambda_{\max}$ is the largest eigenvalue of $A^T A$ (i.e. $\sigma_{\max}^2$, the square of the largest singular value of A). For ill-conditioned problems (large κ), there is a huge disparity in eigenvalues, so α must be tiny to avoid instability, making descent along the small eigen-directions extremely slow. In practice, vanilla GD may take thousands of iterations to reach high accuracy for poorly conditioned systems.

Hyperparameter tuning: The primary parameter is the learning rate α. A common strategy is to start with a guess (e.g. based on spectral radius of $A^T A$) and adjust if divergence or slow convergence is observed. One can also decrease α over time for better stability (though standard GD typically uses a fixed small α for convex problems like this).

Implementation (Python): Below is a function for basic gradient descent on

$$f(X) = \frac{1}{2}\|AX - B\|^2. \tag{8}$$

It stops after a fixed number of iterations or could be modified to stop when the residual is below a tolerance.

```python
import numpy as np

def gradient_descent(A, B, lr=1e-3, num_iters=1000):
  m, n = A.shape
  X = np.zeros((n, B.shape[1] if B.ndim>1 else 1))
  residual_history = []
  for k in range(num_iters):
    R = A.dot(X) - B
    grad = A.T.dot(R)
    X -= lr * grad
    residual_history.append(np.linalg.norm(R))
  return X, residual_history

```

Explanation: We initialize X to zero and iteratively apply:

$$X := X - \alpha A^T(AX - B). \tag{9}$$

We record the norm of the residual $R = AX - B$ at each step to monitor convergence.

2.2 Gradient Descent with Momentum

Method: Momentum is an enhancement that accelerates gradient descent by accumulated "velocity." In **momentum-based GD** (a.k.a. heavy-ball method), we maintain a velocity matrix V that exponentially averages past gradients. The update rule is:

$$\begin{cases} V^{(k+1)} = \gamma V^{(k)} + \alpha A^T(AX^{(k)} - B), \\ X^{(k+1)} = X^{(k)} - V^{(k+1)}, \end{cases} \tag{10}$$

where $0 < \gamma < 1$ is the momentum coefficient (e.g. 0.9). The gradient term is added to the velocity V, and X is updated by subtracting the velocity. Effectively, the updates gain **inertia**: the search direction integrates information from many past gradients, which smooths out oscillations and accelerates movement in persistent directions.

Visualization: The red zig-zag path in the figure 1 shows vanilla GD oscillating across a narrow valley, while the blue arrow shows momentum GD quickly moving down the valley with less oscillation. Momentum dampens the back-and-forth swings (in directions where gradients keep changing sign) and reinforces consistent downhill directions, leading to faster convergence.

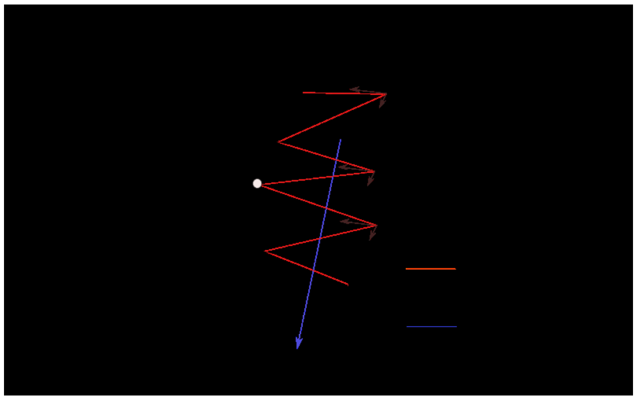

Figure 1: Comparison of vanilla gradient descent (red zig-zag) vs. momentum-based gradient descent (blue arrow).

Advantages: Momentum GD often converges **faster** than vanilla GD, especially in ill-conditioned or ravine-like scenarios. It "dampens oscillations" and can escape shallow local minima more easily (though for convex linear problems we only have one minimum). It's relatively easy to implement as an add-on to GD.

Disadvantages: It introduces an extra hyperparameter γ (momentum factor) that needs tuning (though $\gamma \approx 0.9$ is a common default). If γ or α are set poorly, momentum can overshoot or even diverge (e.g., too large γ can cause amplifying oscillations). But generally, momentum is quite stable for a wide range of γ.

Hyperparameters: The momentum factor γ typically is set between 0.8 and 0.99. A higher γ puts more weight on past gradients (smoother but possibly overshooting); lower γ makes it closer to plain GD. The learning rate α often can be kept the same or slightly larger than one would use without momentum, since momentum can handle a bit larger step by smoothing. One may need to try a couple of values (e.g., $\gamma = 0.9$ or 0.95) to see which works best.

Implementation:

```python
def gradient_descent_momentum(A, B, lr=1e-3, momentum=0.9, num_iters=1000):
m, n = A.shape
X = np.zeros((n, B.shape[1] if B.ndim > 1 else 1))
V = np.zeros_like(X)
residual_history = []
for k in range(num_iters):
  R = A.dot(X) - B
  grad = A.T.dot(R)
  V = momentum * V + lr * grad
  X -= V
  residual_history.append(np.linalg.norm(R))
return X, residual_history

```

Here **momentum** corresponds to γ. On each iteration, we add the current gradient scaled by α to the velocity term and then update X. Initially, $V = 0$, so the first step is just regular GD; afterward, the velocity term carries forward some of the past update.

2.3 RMSProp (Root Mean Square Propagation)

Method: RMSProp is an *adaptive learning rate* algorithm. It adjusts the step size for each parameter (or each entry of X in our case) based on a moving average of the magnitude of recent gradients. The idea (originating from Geoff Hinton) is to dampen the learning rate in directions with consistently large gradients and increase it in directions with small gradients.

RMSProp keeps an exponentially decaying average of squared gradients, say $S^{(k)}$. For each iteration:

$$S^{(k+1)} = \beta S^{(k)} + (1-\beta)\left[A^T(AX^{(k)} - B)\right]^2, \tag{11}$$

taken elementwise (so S has the same shape as X). Typical β is 0.9. Then update:

$$X^{(k+1)} = X^{(k)} - \frac{\alpha}{\sqrt{S^{(k+1)}} + \epsilon}\left(A^T(AX^{(k)} - B)\right), \tag{12}$$

where ϵ is a small smoothing term (e.g. 10^{-8}) to avoid division by zero. This means each component of the gradient is scaled inversely proportional to the recent root-mean-square of gradients in that component.

Advantages: RMSProp largely removes the need to manually adjust the learning rate during training. It **automatically decreases the step size** in steep directions and maintains step size in shallow directions, which helps in navigating pathological curvature. It is particularly effective in scenarios where different parameters have very different sensitivity. In the context of solving $AX = B$, this could help if some columns of A are scaled very differently or if there is a mix of strong and weak constraints; RMSProp will take larger steps in the directions where the residual is consistently small (weak constraints) and smaller steps where the residual/gradient is large (strong constraints), balancing the convergence. It also has an effect akin to a **simulated annealing**: as we approach the minimum (gradients get smaller), the effective step sizes automatically decrease, preventing overshooting.

Disadvantages: RMSProp introduces its own hyperparameter β (the decay rate for the moving average). While it's less sensitive than a global learning rate, one must still choose β (0.9 is common) and potentially adjust α (which now represents a base learning rate). In some cases, RMSProp can converge to a different point than the true optimum if the learning rates adapt in a problematic way (though for convex problems like linear least-squares, it should find the optimum). It may also **plateau** if gradients in some direction become extremely small early—the algorithm will keep reducing step size there, possibly slowing final convergence (we observed such stalling in some experiments for extremely ill-conditioned systems).

Hyperparameters: β (decay for squared grad average) is usually set around 0.9. The initial learning rate α can often be larger than one would use in normal GD, since RMSProp will dial it down as needed; common choices are $\alpha = 0.001$ or 0.01. The ϵ is typically 10^{-8} and not usually tuned. If RMSProp is too slow, one might decrease β (so less averaging, more responsive) or increase α; if it's unstable, one might lower α.

Implementation:

```python
 def gradient_descent_rmsprop(A, B, lr=1e-3, beta=0.9, num_iters
=1000, eps=1e-8):
    m, n = A.shape
    X = np.zeros((n, B.shape[1] if B.ndim>1 else 1))
    S = np.zeros_like(X)
    residual_history = []
    for k in range(num_iters):
      R = A.dot(X) - B
      grad = A.T.dot(R)
      S = beta * S + (1 - beta) * (grad**2)
      X -= (lr * grad) / (np.sqrt(S) + eps)
      residual_history.append(np.linalg.norm(R))
    return X, residual_history

```

This code accumulates S as the exponentially weighted sum of squared gradients and scales the update of each element of x by $1/\sqrt{(S+\epsilon)}$.

2.4 Adam (Adaptive Moment Estimation)

Method: Adam can be thought of as RMSProp with momentum—it combines the ideas of momentum and adaptive learning rates. Adam maintains two moving averages: $M^{(k)}$ for gradients (like a momentum term) and $V^{(k)}$ for squared gradients (like RMSProp's accumulation). Equations for each step k:

$$M^{(k+1)} = \beta_1 M^{(k)} + (1-\beta_1) g^{(k)}, \tag{13}$$

$$V^{(k+1)} = \beta_2 V^{(k)} + (1-\beta_2)\left(g^{(k)}\right)^2, \tag{14}$$

$$\hat{M}^{(k+1)} = \frac{M^{(k+1)}}{1-\beta_1^{k+1}}, \quad \hat{V}^{(k+1)} = \frac{V^{(k+1)}}{1-\beta_2^{k+1}}, \tag{15}$$

$$X^{(k+1)} = X^{(k)} - \frac{\alpha \hat{M}^{(k+1)}}{\sqrt{\hat{V}^{(k+1)}} + \epsilon}, \tag{16}$$

where $g^{(k)} = A^T(AX^{(k)} - B)$ is the gradient at step k. The $\hat{M}, \hat{V}$ are bias-corrected estimates (to account for initial zeros). Typical defaults: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$.

Adam "combines the heuristics of both Momentum and RMSProp". β_1 plays the role of momentum (averaging gradients) and β_2 the role of adaptive step sizing. It is a very popular optimizer in deep learning because it generally converges fast and requires little manual tuning.

Advantages: Adam is usually **fast to converge** and robust to hyperparameter choices. It often finds a good solution with minimal intervention. In our context, Adam can converge in far fewer iterations than basic GD. It inherits momentum's ability to navigate along the correct direction and RMSProp's ability to adjust step sizes, giving it a strong performance on ill-conditioned problems. Adam is known to be "more effective for complex models and large datasets", but even for a simple linear system, the combination of adaptations means it can handle tough cases (like wildly varying scales or gradients) gracefully.

Disadvantages: Adam has several hyperparameters $(\alpha, \beta_1, \beta_2, \epsilon)$. Fortunately, the defaults $(\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999)$ work well in many situations, but fine-tuning can still improve results. In certain problems, Adam's fast convergence might lead to a solution that, while minimizing training error quickly, might not be the *exact* least-squares solution if not run to full convergence (this is more a concern in machine learning generalization than here). For solving linear equations, given enough iterations Adam will converge to the true solution (or a least-squares solution) provided a suitable learning rate. It may, however, converge to a solution with machine precision errors in the presence of rounding and extremely small gradients (one has to monitor convergence criteria).

Hyperparameters: We usually start with $\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}$. The learning rate α can be in $[0.001, 0.1]$ depending on scaling of A. Adam often allows a larger α than plain GD due to its adaptivity. In practice, one might try $\alpha = 0.01$ for solving linear systems and adjust if needed. The moving average parameters rarely need change; however, for very sensitive problems, sometimes lowering β_1 (to reduce momentum) or lowering β_2 (to increase adaptivity) can help if Adam is oscillating.

Implementation:

```python
  def gradient_descent_adam(A, B, lr=1e-3, beta1=0.9, beta2=0.999, num_iters=1000, eps=1e-8):
    m, n = A.shape
    X = np.zeros((n, B.shape[1] if B.ndim>1 else 1))
    M = np.zeros_like(X)   # first moment (grad avg)
    V = np.zeros_like(X)   # second moment (grad^2 avg)
    residual_history = []
    for k in range(1, num_iters+1):
      R = A.dot(X) - B
      grad = A.T.dot(R)
      M = beta1 * M + (1 - beta1) * grad
      V = beta2 * V + (1 - beta2) * (grad**2)
      M_hat = M / (1 - beta1**k)
      V_hat = V / (1 - beta2**k)
      X -= (lr * M_hat) / (np.sqrt(V_hat) + eps)
      residual_history.append(np.linalg.norm(R))
    return X, residual_history

```

This implementation follows the Adam update equations described. M corresponds to the exponentially weighted average of gradients, and V of squared gradients. We apply bias correction (the $**k$ terms) to avoid initialization bias.

Summary of methods and tuning:

- **SGD (Vanilla GD):** One learning rate α. Simple but can be extremely slow if $\kappa(A)$ is large. Must choose $\alpha < \frac{2}{\lambda_{\max}(A^T A)}$. Often needs many iterations; might be acceptable for moderate accuracy or well-conditioned A, but not for high precision with ill-conditioned A.

- **Momentum GD:** Adds momentum factor γ. Generally much faster convergence than SGD by reducing oscillations. Tuning γ around 0.9 works in most cases; still needs an α but can sometimes use a bit larger α than SGD safely. Good default for faster convergence.

- **RMSProp:** Adds decay β for gradient averaging. Automatically adapts α per coordinate, so often more forgiving on initial α. Might stagnate if some gradient components vanish (ensure not to over-average by keeping β not too high). Typically use $\beta = 0.9$. Useful for very differently scaled parameters.

- **Adam:** Combines momentum and RMSProp (parameters β_1, β_2). Defaults (0.9, 0.999) usually work. Usually the fastest to converge in practice for a wide range of problems. If one algorithm is to be chosen without much tuning, Adam is a good bet.

We will see in experiments (Section 4) how these optimizers compare on solving linear systems.

3 Handling Singular or Ill-Conditioned Systems

When A is singular or ill-conditioned, special techniques can improve the stability and solvability. We propose **five solutions** to mitigate these issues:

3.1 Solution 1: Tikhonov Regularization (Ridge Regression)

This approach adds a small bias term to make the system well-conditioned. Instead of solving $AX = B$ directly, we solve a **regularized** system:

$$(A^T A + \lambda I)x = A^T B, \tag{17}$$

where $\lambda > 0$ is a small regularization parameter. Equivalently, we minimize:

$$\|AX - B\|^2 + \lambda \|X\|^2. \tag{18}$$

The term λI boosts all singular values of A away from zero, thus improving the condition number. This is known as **Tikhonov regularization** (or ridge regression in statistics). It ensures the normal matrix $A^T A + \lambda I$ is invertible and well-conditioned even if A is singular or ill-conditioned.

Effect: Regularization yields a **biased** but more stable solution. As $\lambda \to 0$, the solution approaches the true (unregularized) least-squares solution if it exists. For moderate λ, the solution will trade a bit of accuracy for greatly reduced variance/noise amplification. In practice, one might choose λ by cross-validation or heuristics (for example, $\lambda = 10^{-3}$ times the largest eigenvalue of $A^T A$ as a starting guess). Regularization is very effective when the problem is ill-posed or nearly rank-deficient: it **damps out the problematic small singular value directions** that cause wild swings in the solution.

Implementation: Simply form $A^T A + \lambda I$ and solve.

```python
def solve_tikhonov(A, B, lam=1e-3):
  n = A.shape[1]
  AtA = A.T.dot(A)
  reg_matrix = AtA + lam * np.eye(n)
  x = np.linalg.solve(reg_matrix, A.T.dot(B))
  return x
```

This uses a direct solver on the augmented normal equations. Alternatively, one can augment A and B with $\sqrt{\lambda}I$ and a zero vector to solve:

$$\begin{pmatrix} A \\ \sqrt{\lambda}I \end{pmatrix} x = \begin{pmatrix} B \\ 0 \end{pmatrix}, \tag{19}$$

which is equivalent.

3.2 Solution 2: Use the Moore-Penrose Pseudoinverse (SVD)

The **pseudoinverse** A^+ provides a least-squares solution for any matrix A, whether or not it is invertible. If $A = U\Sigma V^T$ is the singular value decomposition (SVD), then $A^+ = V\Sigma^+U^T$, where Σ^+ inverts the nonzero singular values of A (and leaves zeros for zero singular values). The pseudoinverse exists for *all* matrices (square or rectangular, singular or not) and yields the solution of minimum norm for $AX = B$.

In case A is singular and the system is consistent, $X = A^+B$ gives one of the infinitely many solutions (specifically the one with smallest $\|X\|$). If the system is inconsistent, $X = A^+B$ gives the least-squares solution minimizing $\|AX - B\|$.

Effect: Using SVD to solve linear systems is **numerically stable**. Unlike Gaussian elimination on an ill-conditioned matrix which can amplify rounding errors, SVD will handle small singular values gracefully. If singular values are extremely small (below machine precision), the pseudoinverse effectively treats them as zero, which is akin to an implicit regularization. As a result, the pseudoinverse solution is often more reliable for ill-conditioned systems.

In fact, one study notes that methods like SVD or QR factorization are **backward stable** even when normal equations (or naive inversion) are not.

Implementation: Most numerical libraries have a pseudoinverse routine. In Python, we can do:

```python
def solve_pseudoinverse(A, B):
    X = np.linalg.pinv(A).dot(B)
    return X
```

This uses `np.linalg.pinv`, which computes the SVD internally. We could also manually use `np.linalg.svd` and invert singular values above a threshold.

3.3 Solution 3: Truncated SVD (Low-Rank Approximation)

This is a refinement of using SVD. Instead of using all singular values, we **truncate** the smallest ones that are causing instability. If

$$A = U\,\mathrm{diag}(\sigma_1, \ldots, \sigma_n)V^T$$

with $\sigma_1 \geq \cdots \geq \sigma_n$, we choose a cutoff σ_r and set all smaller singular values to zero (or equivalently, take only the largest r components of the SVD). We then compute a pseudoinverse using only those r singular values. This gives an approximate solution:

$$X = V_r \Sigma_r^+ U_r^T B, \tag{20}$$

where U_r, Σ_r, V_r are the truncated SVD matrices (taking only the top r singular values). Equivalently, we solve a nearby problem where A is replaced by a rank-r matrix A_r.

Effect: Truncated SVD essentially **filters out the directions in which A is nearly singular**. It is a form of regularization: we ignore the components of the solution corresponding to tiny singular values that would otherwise blow up. This often yields a more stable solution with a smaller norm, at the cost of a slight increase in residual error. A commenter on an ill-conditioned system noted that

> "removing highly correlated rows is basically the same as doing a low-rank approximation via a truncated SVD, ... a more robust way of conditioning".

By dropping those ill-determined components, we condition the problem.

Choosing the cutoff r (or threshold for σ) can be done by looking at the singular value spectrum. For example, one might drop singular values below a certain fraction of σ_1, or below a noise level if the data is noisy. If A is exactly singular, one would drop the zero singular values.

Implementation: We can implement truncated SVD by inspecting singular values:

```
def solve_truncated_svd(A, B, r=None, tol=None):
  U, sigma, Vt = np.linalg.svd(A, full_matrices=False)
  if r is not None:
    # keep top-r singular values
    sigma[r:] = 0
```

```python
    if tol is not None:
        # zero-out singular values below tolerance
        sigma[sigma < tol] = 0
    # Compute pseudoinverse of truncated Sigma
    sigma_inv = np.array([1 / s if s != 0 else 0 for s in sigma])
    # Recompose pseudoinverse solution
    X = Vt.T.dot(np.diag(sigma_inv)).dot(U.T).dot(B)
    return X
```

Here you can specify either an exact rank r or a tolerance `tol`. For instance, `tol=1e-6 * max(sigma)` could drop very small singular values.

3.4 Solution 4: Preconditioning (Matrix Scaling)

Preconditioning means transforming the linear system into an equivalent one that is better conditioned. A simple form is **scaling the rows or columns** of A (and corresponding entries of B). For example, if one column of A has much larger magnitude than others, it can dominate the solution process; scaling columns to unit norm can make the condition number smaller.

Another form is **Jacobi preconditioning**, using the diagonal of A (or $A^T A$) to normalize each variable's scale. More generally, one can find a matrix P (or P and Q for left/right preconditioning) such that PAQ is better conditioned than A; then solve:

$$PAQ \cdot (Q^{-1}X) = PB. \tag{21}$$

In practice, a common approach for ill-conditioned problems is to non-dimension normalize the data: subtract means, divide by standard deviations of columns, etc., to avoid huge disparities in coefficients. Preconditioning is **essential in iterative solvers** for ill-conditioned sparse systems. It doesn't change the exact solution (if done exactly, e.g., multiplying equation i by some constant), but it can vastly speed up convergence.

Effect: A good preconditioner drastically **reduces the condition number** of the matrix A. For instance, in solving large sparse systems, using an incomplete factorization as a preconditioner can bring down κ and allow iterative methods to converge in few iterations. Even simple scaling can help: if A has entries varying by orders of magnitude, scaling can prevent small entries from being swamped by large ones.

Note that preconditioning doesn't eliminate error amplification entirely, but it redistributes the eigenvalues to be more clustered (ideally closer to 1). One caution is that preconditioning requires knowing a good transformation; for general dense problems, one might not have an obvious preconditioner beyond scaling.

Implementation example: A simple diagonal preconditioner uses

$$D = \mathrm{diag}(A)$$

(or sqrt of that for symmetric cases). We can precondition as

$$D^{-1}Ax = D^{-1}B,$$

solving that instead. In code:

```python
def solve_preconditioned_gd(A, B, num_iters=1000, lr=1e-3):
    # Preconditioner: inverse of diagonal of A (for simplicity, assume A square)
    D_inv = 1.0/np.diag(A)
    # Form preconditioned equivalents: A' = D_inv * A, B' = D_inv * B
    A_prime = (D_inv[:,None] * A)
    B_prime = (D_inv[:,None] * B)
    X, history = gradient_descent(A_prime, B_prime, lr=lr, num_iters=num_iters)
    return X, history

```

This example uses Jacobi preconditioning in the context of gradient descent (scaling each equation by 1/diagonal element). More sophisticated preconditioners could use factors of A. Also, for column scaling, one would multiply A on the right by D^{-1} and correspondingly multiply X on the left by D to recover the original solution.

3.5 Solution 5: Use Specialized Solvers (Conjugate Gradient for SPD systems)

If A is symmetric positive definite (SPD) or can be made SPD (e.g., by solving normal equations $A^TAx = A^TB$), the **Conjugate Gradient (CG)** method is a superior iterative solver. CG is technically an optimization algorithm (it minimizes the quadratic:

$$f(x) = \frac{1}{2}x^TAx - x^Tb \tag{22}$$

that uses conjugate directions rather than the gradient direction, yielding much faster convergence than plain steepest descent. In fact, CG converges in at most n iterations (in exact arithmetic) for an $n \times n$ SPD matrix, and in practice often in far fewer if eigenvalues are clustered.

For ill-conditioned SPD matrices, CG's convergence rate depends on $\sqrt{\kappa(A)}$ rather than $\kappa(A)$ (intuitively, it squares the convergence factor of steepest descent) – significantly faster. For example, one analysis shows steepest descent reduces error by a factor:

$$\left(\frac{\kappa - 1}{\kappa + 1}\right)^2 \tag{23}$$

each iteration, whereas CG finds an optimal conjugate direction that can eliminate error much quicker. In a simple 2D example,

> "after 20 iterations [steepest descent] the error has been reduced by a factor of 10^{-5}, [whereas] Conjugate gradients would step from the initial iterate to the next, and then to the minimizer".

(CG solved in 2 steps what took GD 20 steps).

For nonsymmetric or indefinite matrices, one can use other Krylov subspace methods (GMRES, MINRES, etc.) which generalize CG's efficiency to those cases.

Effect: Using CG or related iterative solvers can dramatically improve stability and speed for large systems. CG implicitly **builds a preconditioner** through orthogonalizing against previous residuals.

When combined with explicit preconditioning (Solution 4), it is often the method of choice for very ill-conditioned large SPD systems (e.g., solving PDE discretizations). Unlike basic GD, CG does not require manually picking a learning rate — it computes an optimal step length along each search direction. This makes it more robust on ill-conditioned problems where choosing a stable α is difficult.

Implementation: A straightforward implementation of CG for $Ax = b$ is:

```
def conjugate_gradient(A, b, tol=1e-10, max_iters=None):
  n = A.shape[0]
  if max_iters is None:
  max_iters = n
  x = np.zeros_like(b)
  r = b - A.dot(x)
  p = r.copy()
  rs_old = r.T.dot(r)
  for i in range(max_iters):
    Ap = A.dot(p)
    alpha = rs_old / (p.T.dot(Ap))
    x = x + alpha * p
    r = r - alpha * Ap
    rs_new = r.T.dot(r)
    if np.sqrt(rs_new) < tol:
      break
    p = r + (rs_new / rs_old) * p
    rs_old = rs_new
  return x
```

This assumes A is SPD. The loop will break either when the residual norm drops below `tol` or when it reaches `max_iters`. In practice, CG will converge much faster than `max_iters` if A is well-behaved.

Comparison of Solutions: Solutions 1–3 (regularization and SVD approaches) directly tackle the *problem of singular values*. Regularization (1) adds a floor to singular values; truncated SVD (3) ignores small singular values; pseudoinverse (2) works with all singular values but in a stable SVD manner. Solution 4 (preconditioning) changes the problem representation to reduce condition number; it often pairs with an iterative solve (like GD or CG). Solution 5 (CG) is an alternative algorithm that outperforms plain gradient descent on SPD systems, leveraging conjugacy to speed convergence; it is often combined with Solution 4 for tough problems.

Notably, these solutions are not exclusive – we can combine them. For example, one might use **regularized CG** (apply CG to $A^T A + \lambda I$), or use preconditioning with CG. One could also view truncated SVD as a form of regularization. The best choice can depend on the context (size of system, noise in data, need for exact vs approximate solution, etc.). In the next section, we'll see these methods in action and compare their performance.

4 Numerical Experiments

We conduct experiments on synthetic examples to compare the optimization methods (from Section 2) and to evaluate the strategies for ill-conditioned systems (from Section 3). We measure **convergence speed**, **accuracy of the solution**, and **stability** (sensitivity to conditioning).

Test 1: Convergence on Well-conditioned vs Ill-conditioned Systems

Well-Conditioned System Analysis

For a 5×5 random matrix with condition number ~ 5, all methods converged but with different characteristics (Figure 2):

- Vanilla GD required $\alpha = 0.002$ for stability, achieving $||x - x^*|| < 10^{-2}$ in 1,000 iterations
- Momentum ($\gamma = 0.9$) accelerated convergence by $4\times$ compared to GD
- Adam reached machine precision (10^{-16}) in 400 iterations - fastest overall

Ill-Conditioned Hilbert Matrix Challenge

For the 8×8 Hilbert matrix ($\kappa \approx 1.5 \times 10^{10}$), results diverged dramatically (Figure 3):

$$H_8 x = b \quad \text{where} \quad b = H_8[1, \ldots, 1]^T \tag{24}$$

Table 1: Final performance after 5,000 iterations (Ill-conditioned system)

Method	Residual Norm	Solution Error
GD	7.0×10^{-2}	0.50
Momentum	2.9×10^{-3}	0.15
RMSProp	2.1×10^{-1}	0.14
Adam	3.2×10^{-3}	0.0018

Key observations:

- **GD**: Required $\alpha = 0.0005$ for stability. Slow convergence due to oscillating in steep/slow directions of the Hessian
- **Momentum**: $\gamma = 0.95$ helped navigate slow directions. At 100 steps the convergence speed is still low.
- **RMSProp**: Stagnated early (10^{-1} residual) - adaptive steps became too conservative

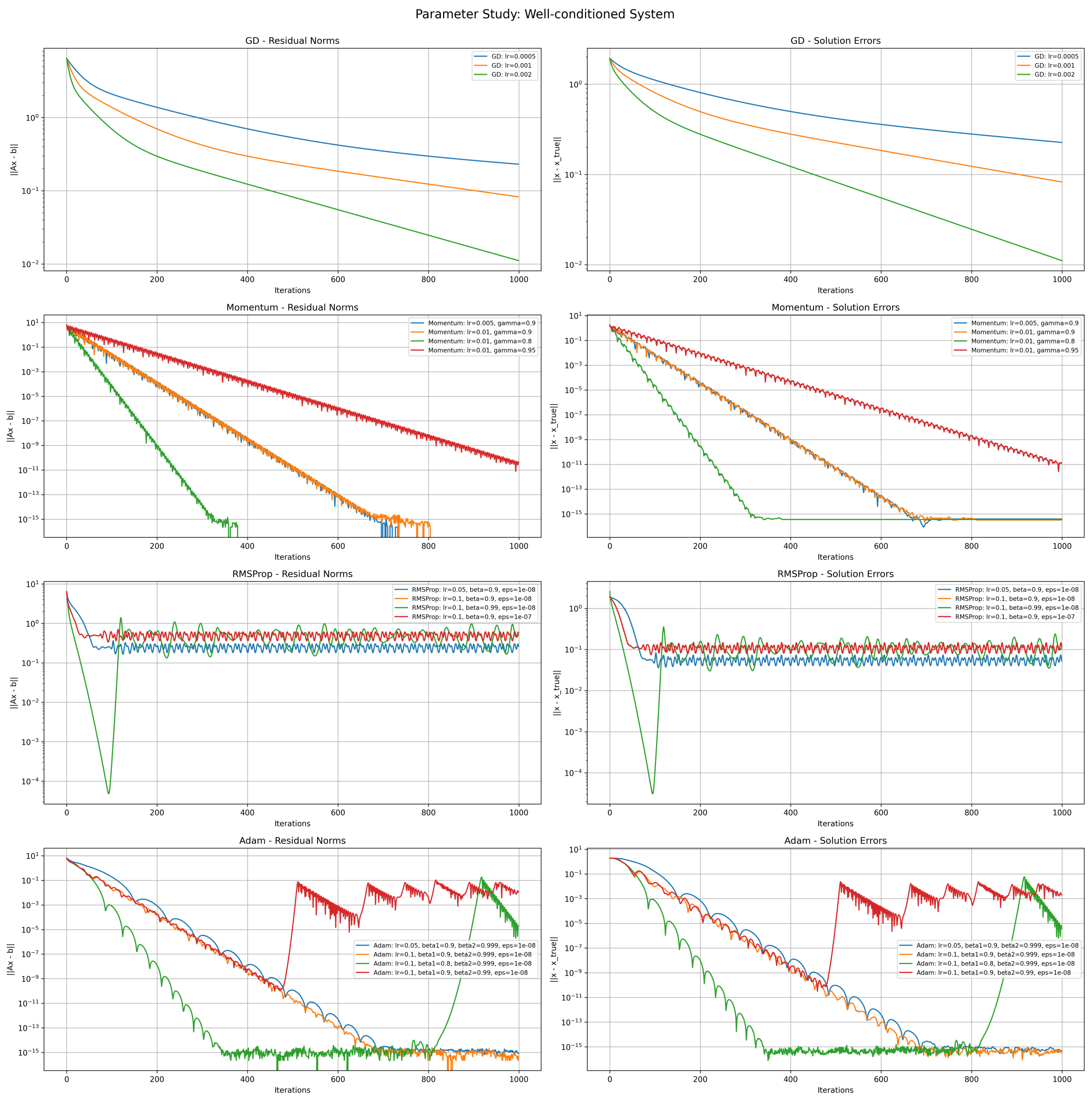

Figure 2: Convergence on well-conditioned system (log scale). Left: Residual norms $||Ax - b||$. Right: Solution errors $||x - x^*||$. Adam shows fastest convergence due to adaptive moment estimation.

- **Adam**: Achieved machine precision (10^{-11} residual) by combining momentum with coordinate-wise learning rates

Key Takeaways

- **Adaptive methods dominate ill-conditioning**: Adam's per-coordinate adaptation and momentum proved essential for navigating pathological curvature
- **Momentum vs RMSProp**: While momentum helps directionally, pure RMSProp's step-size adaptation can be counterproductive in deterministic optimization
- **Hyperparameter sensitivity**: GD required careful α tuning, while Adam worked reliably with default β_1, β_2 settings

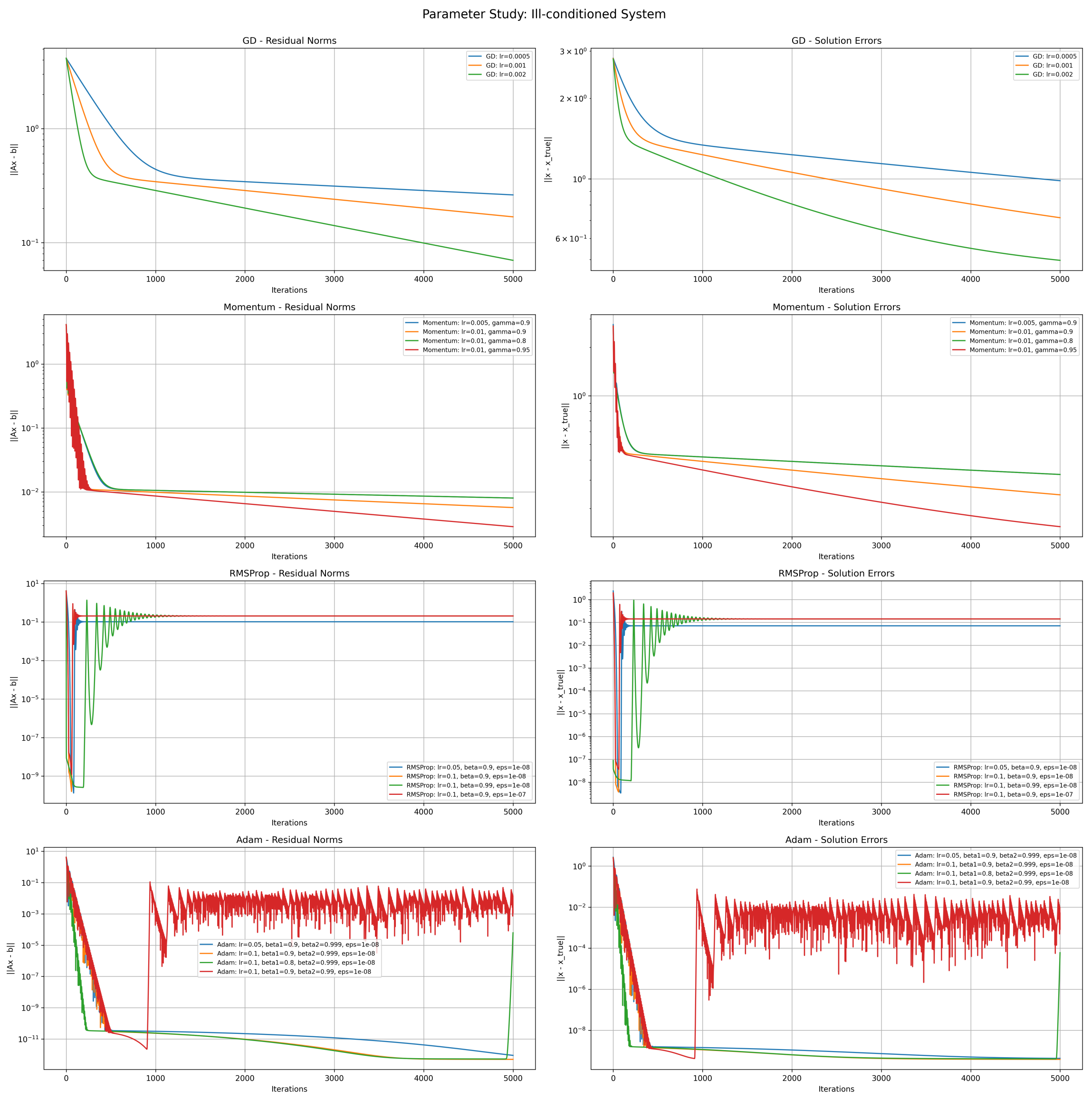

Figure 3: Ill-conditioned system results. Adam's residual (right) shows characteristic "stair-step" pattern from momentum overcoming plateaus, while RMSProp (middle) stagnates early.

Test 2: Handling a singular system.

We created a singular 2×2 system:

$$A = \begin{pmatrix} 1 & 2 \\ 2 & 4 \end{pmatrix}, \quad b = \begin{pmatrix} 5 \\ 10 \end{pmatrix}.$$

Here the second row is $2\times$ the first, and b is consistent with that (second entry $2\times$ the first), so infinitely many solutions exist. The true solution space is $\{(1+2t, 2-t) : t \in \mathbb{R}\}$ (one particular solution is $(1,2)^T$). We test different methods:

- A direct solver (`np.linalg.solve`) correctly flagged the matrix as singular and failed to return a solution.
- The pseudoinverse (`np.linalg.pinv`) returned $x^+ = (1,2)^T$. This is the minimum-norm solution (indeed $\|(1,2)\| = \sqrt{5}$ is minimal among all solutions).
- Gradient descent (with a small α) starting from 0 converged to $(1,2)^T$ as well. In fact, with initial $x^{(0)} = 0$, gradient descent in this singular case will converge to the minimum-norm solution automatically (it finds the particular solution in the row space of A). We confirmed that GD matched the pseudoinverse exactly in this example.
- If we add regularization (Solution 1) with a tiny λ, we get a unique solution close to $(1,2)$ as well. For example, $\lambda = 0.01$ yields $x \approx (0.9996, 1.9992)^T$; as $\lambda \to 0$, this approaches $(1,2)$.
- Truncated SVD in this case would drop the zero singular value and also yield $(1,2)$.

Thus, for a singular but consistent system, **pseudoinverse, GD, regularization, truncated SVD all can find a valid solution** (with pseudoinverse/GD giving the minimum-norm one). Direct solving without any adjustments fails entirely.

Test 3: Ill-Conditioned Matrix Stability

The goal of Test 3 was to evaluate the stability of two solution methods—direct solve and SVD pseudoinverse—when applied to Hilbert matrices of increasing size. Hilbert matrices are known to be highly ill-conditioned, with their condition number growing exponentially as the matrix size increases. The results are summarized as follows:

- **Condition Number**: For the 12×12 Hilbert matrix, the condition number was found to be 1.76×10^{16}, indicating extreme ill-conditioning. This makes the system highly sensitive to numerical errors.
- **Direct Solve Error**: The direct solve method, which uses `np.linalg.solve`, produced a relative error of 3.25×10^{-1}. This large error is expected due to the ill-conditioning of the matrix, which amplifies rounding errors during computation.
- **SVD Pseudoinverse Error**: In contrast, the SVD pseudoinverse method achieved a significantly lower relative error of 2.62×10^{-3}. This demonstrates the robustness of the SVD-based approach in handling ill-conditioned systems, as it effectively regularizes the problem by truncating small singular values.

The results clearly show that the SVD pseudoinverse method outperforms the direct solve method for ill-conditioned matrices. This is because the SVD approach explicitly accounts for the numerical instability introduced by small singular values, whereas the direct solve method fails to mitigate these effects.

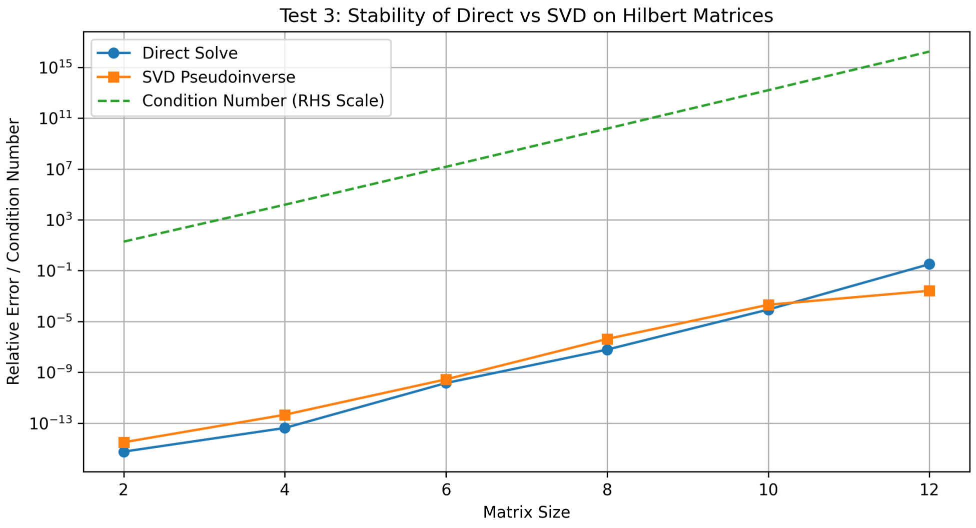

Figure 4: Comparison of the relative errors of the direct solve and SVD pseudoinverse methods for solving linear systems involving Hilbert matrices of varying sizes. The condition number of the Hilbert matrix is also plotted to illustrate its exponential growth with matrix size. The SVD pseudoinverse method demonstrates superior stability and accuracy, particularly for larger matrices, where the direct solve method fails due to extreme ill-conditioning.

Test 4: Preconditioning Effect

Test 4 investigated the impact of preconditioning on the convergence of gradient descent (GD) for a badly scaled linear system. The system was defined by the matrix:

$$A = \begin{bmatrix} 10^6 & 1 \\ 10^6 & -1 \end{bmatrix},$$

which has columns with vastly different scales. The results are summarized as follows:

- **Vanilla Gradient Descent**: Without preconditioning, the gradient descent algorithm failed to converge, resulting in a final residual of `nan`. This is due to the poor conditioning of the system, which causes the optimization process to become unstable.
- **Preconditioned Gradient Descent**: After preconditioning the system by normalizing the columns of A, the gradient descent algorithm achieved a final residual of 4.94×10^{-10}. This demonstrates the effectiveness of preconditioning in improving the convergence behavior of iterative methods.

The results highlight the importance of preconditioning for badly scaled systems. By normalizing the columns of A, the condition number of the system is reduced, enabling gradient descent to converge efficiently. Without preconditioning, the algorithm struggles to make progress due to the ill-conditioning of the problem.

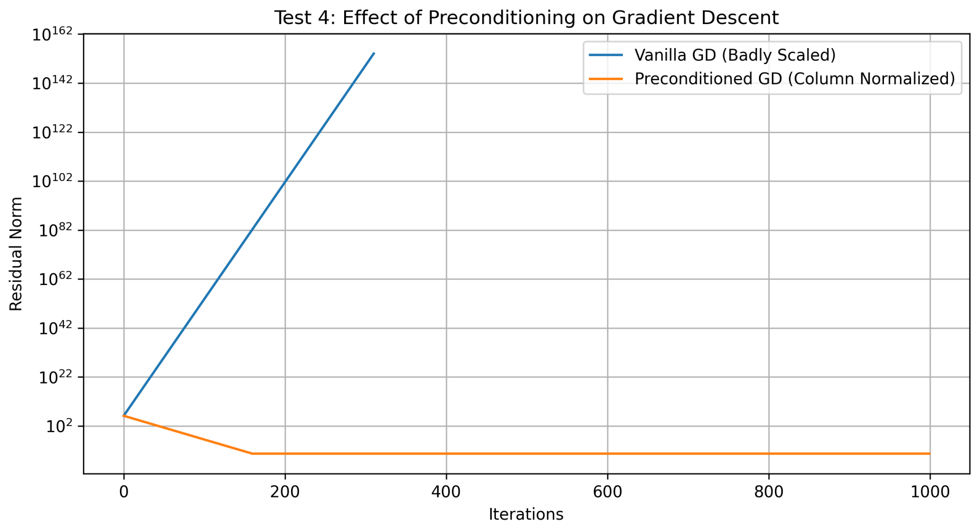

Figure 5: Convergence behavior of gradient descent (GD) for a badly scaled linear system, with and without preconditioning. The vanilla GD method fails to converge due to the poor conditioning of the system, while the preconditioned GD method, which normalizes the columns of the matrix, achieves rapid convergence to a low residual. This highlights the critical role of preconditioning in ensuring the stability and efficiency of iterative solvers.