Quasi-Newton and Levenberg-Marquardt methods

Motivation

Why Quasi-Newton Methods?

The Goal

Find a tradeoff between the advantages and disadvantages of:

- **Steepest Descent:** Low cost per iteration ($O(n)$), but slow (linear) convergence.
- **Newton-Raphson:** Very fast (quadratic) convergence, but high cost per iteration ($O(n^3)$) and requires computing the full Hessian.

The Quasi-Newton Aim

Develop methods that converge faster than steepest descent but have a lower operation count per iteration than Newton-Raphson.

The Newton-Raphson Method

The Newton-Raphson method can be seen as replacing the original problem $\min_{\mathbf{x}\in\mathbb{R}^n} f(\mathbf{x})$ with iteratively minimising a quadratic model.

Second-Order Taylor Model

At $\mathbf{x}_k$, approximate $f(\mathbf{x})$ with $q(\mathbf{x})$:

$$q(\mathbf{x}) = f(\mathbf{x}_k) + \langle \nabla f(\mathbf{x}_k), \mathbf{x} - \mathbf{x}_k \rangle + \frac{1}{2}(\mathbf{x} - \mathbf{x}_k)^T D^2 f(\mathbf{x}_k)(\mathbf{x} - \mathbf{x}_k)$$

If $D^2 f(\mathbf{x}_k)$ is positive definite, the unique minimiser of $q(\mathbf{x})$ is:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - (D^2 f(\mathbf{x}_k))^{-1} \nabla f(\mathbf{x}_k)$$

This is the Newton-Raphson iterate.

The Quasi-Newton Approach

Approximating the Hessian

The core idea is to replace the true (and expensive) Hessian $D^2 f(\mathbf{x}_k)$ with a simpler approximation B_k.

The Quasi-Newton Approach

Approximating the Hessian

The core idea is to replace the true (and expensive) Hessian $D^2 f(\mathbf{x}_k)$ with a simpler approximation B_k.

Quasi-Newton Quadratic Model

We build a new model $p(\mathbf{x})$ using $B_k \approx D^2 f(\mathbf{x}_k)$:

$$p(\mathbf{x}) = f(\mathbf{x}_k) + \langle \nabla f(\mathbf{x}_k), \mathbf{x} - \mathbf{x}_k \rangle + \frac{1}{2}(\mathbf{x} - \mathbf{x}_k)^T B_k (\mathbf{x} - \mathbf{x}_k)$$

If B_k is positive definite, minimising $p(\mathbf{x})$ yields:

$$\mathbf{x}^* = \mathbf{x}_k - B_k^{-1} \nabla f(\mathbf{x}_k)$$

From Iterate to Search Direction

Since B_k is only an approximation, the resulting step $\mathbf{x}^* - \mathbf{x}_k$ is not a full update, but rather a search direction.

From Iterate to Search Direction

Since B_k is only an approximation, the resulting step $\mathbf{x}^* - \mathbf{x}_k$ is not a full update, but rather a search direction.

Quasi-Newton Search Direction ($\mathbf{d}_k$)

$$\mathbf{d}_k = -B_k^{-1} \nabla f(\mathbf{x}_k)$$

This direction is then used in a line search to find the next iterate.

From Iterate to Search Direction

Since B_k is only an approximation, the resulting step $\mathbf{x}^* - \mathbf{x}_k$ is not a full update, but rather a search direction.

Quasi-Newton Search Direction ($\mathbf{d}_k$)

$$\mathbf{d}_k = -B_k^{-1} \nabla f(\mathbf{x}_k)$$

This direction is then used in a line search to find the next iterate.

Line Search Update

Find a suitable step length $\alpha_k > 0$ (e.g., via Wolfe conditions that we will see later in **S3**):

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{d}_k$$

From Iterate to Search Direction

Since B_k is only an approximation, the resulting step $\mathbf{x}^* - \mathbf{x}_k$ is not a full update, but rather a search direction.

Quasi-Newton Search Direction ($\mathbf{d}_k$)

$$\mathbf{d}_k = -B_k^{-1} \nabla f(\mathbf{x}_k)$$

This direction is then used in a line search to find the next iterate.

Line Search Update

Find a suitable step length $\alpha_k > 0$ (e.g., via Wolfe conditions that we will see later in **S3**):

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{d}_k$$

Open problem

The key remaining part is: **How do we compute B_{k+1} cheaply?**

Algorithm 1.1: Generic Quasi-Newton

Generic Algorithm

S0 Choose $\mathbf{x}_0 \in \mathbb{R}^n$, a nonsingular $B_0 \in S^n$ (often $B_0 = I$), and tolerance $\epsilon > 0$. Set $k = 0$.

Algorithm 1.1: Generic Quasi-Newton

Generic Algorithm

S0 Choose $\mathbf{x}_0 \in \mathbb{R}^n$, a nonsingular $B_0 \in S^n$ (often $B_0 = I$), and tolerance $\epsilon > 0$. Set $k = 0$.

S1 If $||\nabla f(\mathbf{x}_k)|| \leq \epsilon$ stop.

Algorithm 1.1: Generic Quasi-Newton

Generic Algorithm

S0 Choose $\mathbf{x}_0 \in \mathbb{R}^n$, a nonsingular $B_0 \in S^n$ (often $B_0 = I$), and tolerance $\epsilon > 0$. Set $k = 0$.

S1 If $||\nabla f(\mathbf{x}_k)|| \leq \epsilon$ stop.

S2 Compute the quasi-Newton search direction:

$$\mathbf{d}_k = -B_k^{-1} \nabla f(\mathbf{x}_k)$$

Algorithm 1.1: Generic Quasi-Newton

Generic Algorithm

S0 Choose $\mathbf{x}_0 \in \mathbb{R}^n$, a nonsingular $B_0 \in S^n$ (often $B_0 = I$), and tolerance $\epsilon > 0$. Set $k = 0$.

S1 If $||\nabla f(\mathbf{x}_k)|| \leq \epsilon$ stop.

S2 Compute the quasi-Newton search direction:

$$\mathbf{d}_k = -B_k^{-1} \nabla f(\mathbf{x}_k)$$

S3 Perform a line search to find α_k (e.g., satisfying Wolfe conditions) for $\phi(\alpha) = f(\mathbf{x}_k + \alpha \mathbf{d}_k)$.

Algorithm 1.1: Generic Quasi-Newton

Generic Algorithm

S0 Choose $\mathbf{x}_0 \in \mathbb{R}^n$, a nonsingular $B_0 \in S^n$ (often $B_0 = I$), and tolerance $\epsilon > 0$. Set $k = 0$.

S1 If $||\nabla f(\mathbf{x}_k)|| \leq \epsilon$ stop.

S2 Compute the quasi-Newton search direction:

$$\mathbf{d}_k = -B_k^{-1} \nabla f(\mathbf{x}_k)$$

S3 Perform a line search to find α_k (e.g., satisfying Wolfe conditions) for $\phi(\alpha) = f(\mathbf{x}_k + \alpha \mathbf{d}_k)$.

S4 Compute the new iterate: $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{d}_k$.

Algorithm 1.1: Generic Quasi-Newton

Generic Algorithm

S0 Choose $\mathbf{x}_0 \in \mathbb{R}^n$, a nonsingular $B_0 \in S^n$ (often $B_0 = I$), and tolerance $\epsilon > 0$. Set $k = 0$.

S1 If $||\nabla f(\mathbf{x}_k)|| \leq \epsilon$ stop.

S2 Compute the quasi-Newton search direction:

$$\mathbf{d}_k = -B_k^{-1} \nabla f(\mathbf{x}_k)$$

S3 Perform a line search to find α_k (e.g., satisfying Wolfe conditions) for $\phi(\alpha) = f(\mathbf{x}_k + \alpha \mathbf{d}_k)$.

S4 Compute the new iterate: $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{d}_k$.

S5 Compute the new approximate Hessian B_{k+1} using a specific update rule.

Algorithm 1.1: Generic Quasi-Newton

Generic Algorithm

S0 Choose $\mathbf{x}_0 \in \mathbb{R}^n$, a nonsingular $B_0 \in S^n$ (often $B_0 = I$), and tolerance $\epsilon > 0$. Set $k = 0$.

S1 If $||\nabla f(\mathbf{x}_k)|| \leq \epsilon$ stop.

S2 Compute the quasi-Newton search direction:

$$\mathbf{d}_k = -B_k^{-1} \nabla f(\mathbf{x}_k)$$

S3 Perform a line search to find α_k (e.g., satisfying Wolfe conditions) for $\phi(\alpha) = f(\mathbf{x}_k + \alpha \mathbf{d}_k)$.

S4 Compute the new iterate: $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{d}_k$.

S5 Compute the new approximate Hessian B_{k+1} using a specific update rule.

S6 Set $k \leftarrow k + 1$ and go to S1.

A Wish List for B_k (P1, P2, P3)

What properties should our approximation B_k have?

Properties P1-P3

P1: B_k should be nonsingular, so that $\mathbf{d}_k = -B_k^{-1}\nabla f(\mathbf{x}_k)$ is well-defined.

A Wish List for B_k (P1, P2, P3)

What properties should our approximation B_k have?

Properties P1-P3

P1: B_k should be nonsingular, so that $\mathbf{d}_k = -B_k^{-1}\nabla f(\mathbf{x}_k)$ is well-defined.

P2: B_k should produce a descent direction, so that $\mathbf{d}_k$ is a valid search direction (i.e., $\langle \nabla f(\mathbf{x}_k), \mathbf{d}_k \rangle < 0$).

A Wish List for B_k (P1, P2, P3)

What properties should our approximation B_k have?

Properties P1-P3

P1: B_k should be nonsingular, so that $\mathbf{d}_k = -B_k^{-1}\nabla f(\mathbf{x}_k)$ is well-defined.

P2: B_k should produce a descent direction, so that $\mathbf{d}_k$ is a valid search direction (i.e., $\langle \nabla f(\mathbf{x}_k), \mathbf{d}_k \rangle < 0$).

P3: B_k should be symmetric, because the true Hessian is symmetric.

The Power of (Symmetric) Positive Definiteness

Properties P1, P2, and P3 can all be satisfied by one simple requirement:

Requirement

Require that B_k be **symmetric positive definite (SPD)**.

- If B_k is SPD, it is automatically nonsingular (P1) and symmetric (P3).
- P2 is satisfied (assuming $\nabla f(\mathbf{x}_k) \neq \mathbf{0}$):

$$\langle \nabla f(\mathbf{x}_k), \mathbf{d}_k \rangle = -\nabla f(\mathbf{x}_k)^T B_k^{-1} \nabla f(\mathbf{x}_k) < 0$$

A Wish List for B_k (P4)

Property P4

P4: B_{k+1} should be computable cheaply, ideally by "recycling" quantities we already have.
We define two **key vectors** using this information:

$$\boldsymbol{\delta}_k = \mathbf{x}_{k+1} - \mathbf{x}_k = \alpha_k \mathbf{d}_k \quad \text{(change in position)}$$

$$\boldsymbol{\gamma}_k = \nabla f(\mathbf{x}_{k+1}) - \nabla f(\mathbf{x}_k) \quad \text{(change in gradient)}$$

The Secant Condition

How can γ_k give us info about the Hessian?

The Secant Condition

How can γ_k give us info about the Hessian?

- A Taylor expansion gives:

$$\gamma_k = \nabla f(\mathbf{x}_{k+1}) - \nabla f(\mathbf{x}_k) \approx D^2 f(\mathbf{x}_k)\boldsymbol{\delta}_k$$

- We enforce the condition that B_{k+1} would have perfectly predicted the gradient change γ_k over the step $\boldsymbol{\delta}_k$.

The Secant Condition

How can γ_k give us info about the Hessian?

- A Taylor expansion gives:

$$\gamma_k = \nabla f(\mathbf{x}_{k+1}) - \nabla f(\mathbf{x}_k) \approx D^2 f(\mathbf{x}_k)\delta_k$$

- We enforce the condition that B_{k+1} would have perfectly predicted the gradient change γ_k over the step δ_k.

The Secant Condition

The new approximation B_{k+1} must satisfy:

$$B_{k+1}\delta_k = \gamma_k$$

This is the most important equation in quasi-Newton methods.

The Secant Condition in 1D

Let's translate the N-D equation to 1D:

- The vectors $\mathbf{x}_k, \boldsymbol{\delta}_k, \boldsymbol{\gamma}_k$ all become **scalars**.
- The gradient $\nabla f(x)$ becomes the **first derivative** $f'(x)$.
- The Hessian matrix B_{k+1} becomes a **scalar approximation** b_{k+1}.

The Secant Condition in 1D

Let's translate the N-D equation to 1D:

- The vectors $\mathbf{x}_k, \boldsymbol{\delta}_k, \boldsymbol{\gamma}_k$ all become **scalars**.
- The gradient $\nabla f(x)$ becomes the **first derivative** $f'(x)$.
- The Hessian matrix B_{k+1} becomes a **scalar approximation** b_{k+1}.

The Secant Condition $B_{k+1}\delta_k = \gamma_k$ becomes:

$$b_{k+1} \cdot (x_{k+1} - x_k) = f'(x_{k+1}) - f'(x_k)$$

The Secant Condition in 1D

Let's translate the N-D equation to 1D:

- The vectors $\mathbf{x}_k, \boldsymbol{\delta}_k, \boldsymbol{\gamma}_k$ all become **scalars**.
- The gradient $\nabla f(x)$ becomes the **first derivative** $f'(x)$.
- The Hessian matrix B_{k+1} becomes a **scalar approximation** b_{k+1}.

The Secant Condition $B_{k+1}\delta_k = \gamma_k$ becomes:

$$b_{k+1} \cdot (x_{k+1} - x_k) = f'(x_{k+1}) - f'(x_k)$$

The 1D Interpretation

Solving for our new approximation b_{k+1}, we get:

$$b_{k+1} = \frac{f'(x_{k+1}) - f'(x_k)}{x_{k+1} - x_k}$$

Geometric Meaning of the 1D Condition

The equation $b_{k+1} = \dfrac{f'(x_{k+1}) - f'(x_k)}{x_{k+1} - x_k}$ has a clear geometric meaning.

- This is the definition of a slope: $\frac{\Delta y}{\Delta x}$.
- The "y" values are not from the function $f(x)$, but from the **first derivative**, $f'(x)$.

Geometric Meaning of the 1D Condition

The equation $b_{k+1} = \dfrac{f'(x_{k+1}) - f'(x_k)}{x_{k+1} - x_k}$ has a clear geometric meaning.

- This is the definition of a slope: $\frac{\Delta y}{\Delta x}$.
- The "y" values are not from the function $f(x)$, but from the **first derivative**, $f'(x)$.

The Core Idea

The secant condition forces b_{k+1} (our approximation of the second derivative) to be the **slope of the secant line** connecting the points $(x_k, f'(x_k))$ and $(x_{k+1}, f'(x_{k+1}))$ on the graph of the **first derivative**.

This is a finite difference approximation of the second derivative, $f''(x)$.

Geometric Meaning (Visualized)

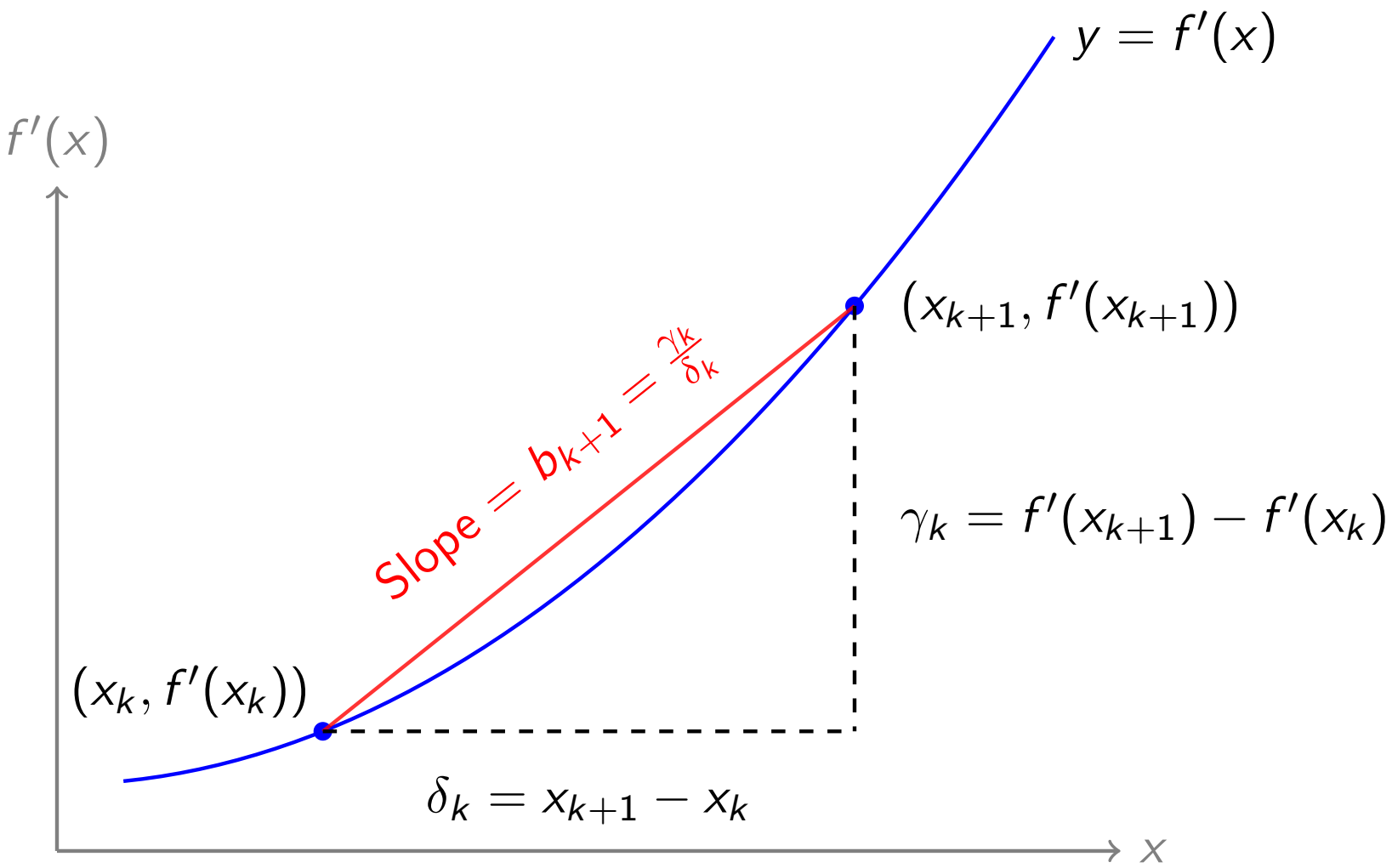

Figure: The scalar b_{k+1} is the slope of the secant line on the graph of the first derivative, $f'(x)$.

A Wish List for B_k(P5, P6)

Properties P5-P6

P5: B_{k+1} should be "close" to B_k.

- We are updating our information, not throwing it away.
- This allows $B_k \to D^2 f(\mathbf{x}^*)$ as $\mathbf{x}_k \to \mathbf{x}^*$.
- "Closeness" is often defined by a low-rank update.

A Wish List for B_k(P5, P6)

Properties P5-P6

P5: B_{k+1} should be "close" to B_k.

- We are updating our information, not throwing it away.
- This allows $B_k \to D^2 f(\mathbf{x}^*)$ as $\mathbf{x}_k \to \mathbf{x}^*$.
- "Closeness" is often defined by a low-rank update.

P6: The total work per iteration should be at most $O(n^2)$.

- This is the only way to gain a substantial speed-up over Newton-Raphson's $O(n^3)$ cost.

Symmetric Rank-1 (SR1) Updates I

Let's try to build an update that satisfies P3 (symmetric) and P5 (low-rank) in the simplest way possible: a symmetric rank-1 update.

SR1 Update Form

$$B_{k+1} = B_k + \mathbf{u}\mathbf{u}^T$$

where $\mathbf{u}$ is some vector.

- The matrix $\mathbf{u}\mathbf{u}^T$ (outer product) is symmetric and has rank 1.
- If B_0 is symmetric, all B_k will be symmetric.

Symmetric Rank-1 (SR1) Updates II

How to find $\mathbf{u}$? We enforce the Secant Condition (P4).

$$B_{k+1}\boldsymbol{\delta}_k = \boldsymbol{\gamma}_k \implies (B_k + \mathbf{u}\mathbf{u}^T)\boldsymbol{\delta}_k = \boldsymbol{\gamma}_k$$

Solving for $\mathbf{u}$

$$(\mathbf{u}^T\boldsymbol{\delta}_k)\mathbf{u} = \boldsymbol{\gamma}_k - B_k\boldsymbol{\delta}_k$$

i.e

$$\mathbf{u} = \frac{(\boldsymbol{\gamma}_k - B_k\boldsymbol{\delta}_k)}{\mathbf{u}^T\boldsymbol{\delta}_k}, \quad \mathbf{u}^T = \frac{(\boldsymbol{\gamma}_k - B_k\boldsymbol{\delta}_k)^T}{\mathbf{u}^T\boldsymbol{\delta}_k}$$

Deriving the SR1 Formula

Substituting back into the update equation yields:

$$B_{k+1} = B_k + \frac{(\gamma_k - B_k\delta_k)(\gamma_k - B_k\delta_k)^T}{((\mathbf{u}^T\delta_k)\mathbf{u}^T)\delta_k}$$

Deriving the SR1 Formula

Substituting back into the update equation yields:

$$B_{k+1} = B_k + \frac{(\gamma_k - B_k\delta_k)(\gamma_k - B_k\delta_k)^T}{((\mathbf{u}^T\delta_k)\mathbf{u}^T)\delta_k}$$

SR1 Update Formula

$$B_{k+1} = B_k + \frac{(\gamma_k - B_k\delta_k)(\gamma_k - B_k\delta_k)^T}{(\gamma_k - B_k\delta_k)^T\delta_k}$$

This is the Symmetric Rank 1 (SR1) method.

SR1: Drawbacks

The SR1 method is simple, but has serious drawbacks:

Drawbacks

- **Positive Definiteness is not guaranteed.** B_{k+1} may not be positive definite even if B_k is. Thus, $\mathbf{d}_k$ might not be a descent direction (P2 fails).

SR1: Drawbacks

The SR1 method is simple, but has serious drawbacks:

Drawbacks

- **Positive Definiteness is not guaranteed.** B_{k+1} may not be positive definite even if B_k is. Thus, $\mathbf{d}_k$ might not be a descent direction (P2 fails).
- **The denominator can be zero or small.**

$$(\gamma_k - B_k \delta_k)^T \delta_k \approx 0$$

If this happens, the update is undefined or numerically unstable.

SR1: The Complexity Problem (P6)

We have a problem with our $O(n^2)$ goal (P6).

- **Computing B_{k+1}:** This involves vector operations and an outer product. This is $O(n^2)$. (Good!)
- **Computing $\mathbf{d}_k$:** To get the next search direction, we must solve $B_k \mathbf{d}_k = -\nabla f(\mathbf{x}_k)$. This takes $O(n^3)$. (Bad!)

Problem

The $O(n^3)$ cost of solving the system dominates the $O(n^2)$ cost of the update, so the total iteration cost is still $O(n^3)$, just like Newton-Raphson.

Solution: Sherman-Morrison-Woodbury

Idea: Instead of updating B_k, let's update its inverse $H_k = B_k^{-1}$ directly.

Theorem (Sherman-Morrison-Woodbury)

If B is $n \times n$ and U, V are $n \times p$, then

$$(B + UV^T)^{-1} = B^{-1} - B^{-1}U(I_p + V^T B^{-1} U)^{-1} V^T B^{-1}$$

Applying SMW to SR1

Suppose we know $H_k = B_k^{-1}$ then applying the SMW formula to B_{k+1} with $= B = B_k, U = \mathbf{u} = (\gamma_k - B_k\delta_k)$ and $V = U^T = \mathbf{u}^T$ we have

$$\begin{aligned} H_{k+1} &= B_{k+1}^{-1} \\ &= B_k^{-1} - B_k^{-1}\mathbf{u}(1 + \mathbf{u}^T B_k^{-1}\mathbf{u})^{-1}\mathbf{u}^T B_k^{-1} \\ &= H_k + \frac{(\delta_k - H_k\gamma_{\boldsymbol{k}})(\delta_k - H_k\gamma_k)^T}{(\delta_k - H_k\gamma_{\boldsymbol{k}})^T\gamma_k} \end{aligned} \tag{1}$$

SR1 Inverse Update Formula

$$H_{k+1} = H_k + \frac{(\delta_k - H_k\gamma_k)(\delta_k - H_k\gamma_k)^T}{(\delta_k - H_k\gamma_k)^T\gamma_k}$$

This is also a symmetric rank-1 update, but for the inverse Hessian.

SR1 with Inverse Hessian: P6 Solved!

Algorithm (SR1 update H_k instead of B_k)

- **S0:** Start with $H_0 = B_0^{-1}$ (e.g., $H_0 = I$).
- **S2 (Compute d_k):**

$$\boxed{d_k = -H_k \nabla f(x_k)}$$

matrix-vector multiplication $O(n^2)$

- **S5 (Compute H_{k+1}):**

$$\boxed{H_{k+1} = H_k + \frac{(\dots)(\dots)^T}{(\dots)^T(\dots)}}$$

vector additions + outer product

$O(n^2)$

Complexity

The total work per iteration is now $O(n^2)$, satisfying P6.

Convergence

The method converges *superlinearly* close to the minimizer.

SR1 Summary

- SR1 is a simple, rank-1 update that satisfies the secant condition.
- By updating the inverse Hessian H_k, it achieves an $O(n^2)$ iteration cost.

Problem

The SR1 update for H_k still has issues:

- The denominator $(\delta_k - H_k\gamma_k)^T\gamma_k$ can be zero.
- H_{k+1} (and B_{k+1}) are not guaranteed to be positive definite.

Motivation for BFGS

- The SR1 method is good, but the lack of guaranteed positive definiteness is a serious practical problem.

Motivation for BFGS

- The SR1 method is good, but the lack of guaranteed positive definiteness is a serious practical problem.
- **Question:** Can we find an update that satisfies all 6 properties, especially P1-P6?

Motivation for BFGS

- The SR1 method is good, but the lack of guaranteed positive definiteness is a serious practical problem.
- **Question:** Can we find an update that satisfies all 6 properties, especially P1-P6?
- **Answer:** Yes, but we need a slightly more complex update.

BFGS (Broyden-Fletcher-Goldfarb-Shanno)

- The most widely used and effective quasi-Newton algorithm.
- It is a symmetric rank-2 update ($B_{k+1} = B_k + \mathbf{u}\mathbf{u}^T + \mathbf{v}\mathbf{v}^T$).
- It overcomes the weaknesses of SR1 while retaining its $O(n^2)$ speed.

The BFGS Formula (for B_k)

The BFGS update $B_k \to B_{k+1}$ that maintains symmetry, is "close" to B_k, and satisfies the secant condition is:

BFGS Update Formula

$$B_{k+1} = B_k - \frac{B_k \boldsymbol{\delta}_k \boldsymbol{\delta}_k^T B_k}{\boldsymbol{\delta}_k^T B_k \boldsymbol{\delta}_k} + \frac{\boldsymbol{\gamma}_k \boldsymbol{\gamma}_k^T}{\boldsymbol{\gamma}_k^T \boldsymbol{\delta}_k}$$

- This is a rank-2 update.
- Again, computing B_{k+1} is $O(n^2)$, but solving $B_k d_k = -\nabla f(x_k)$ is $O(n^3)$.
- We could use SMW, but there is a more stable way to maintain positive definiteness.

BFGS Strategy: Cholesky Factorisation

Idea: Instead of updating B_k or H_k, we will update the **Cholesky factor** L_k of B_k.

Definition: Cholesky Factorisation

A symmetric matrix B has a Cholesky factorisation if there exists a **lower-triangular** matrix L with **positive diagonal** entries such that:

$$B = LL^T$$

Proposition: Existence

A matrix B has a unique Cholesky factorisation $B = LL^T$ **if and only if** B is symmetric positive definite.

This is perfect! If we can maintain an update $L_k \to L_{k+1}$, we automatically guarantee $B_{k+1} = L_{k+1}L_{k+1}^T$ is positive definite (P1, P2, P3).

Solving with Cholesky: P6 Solved!

If we have the Cholesky factor L_k, we can solve the $O(n^3)$ system $B_k \mathbf{d}_k = -\nabla f(x_k)$ in $O(n^2)$ time.

Substitute $B_k = L_k L_k^T$: $\boxed{L_k L_k^T \mathbf{d}_k = -\nabla f(x_k)}$

We solve this in two (cheap) steps:

1. **Forward substitution:** Let $\mathbf{g}_k = L_k^T \mathbf{d}_k$. Solve for $\mathbf{g}_k$:

$$L_k \mathbf{g}_k = -\nabla f(x_k)$$

   (This is $O(n^2)$ because L_k is lower-triangular.)

2. **Backward substitution:** Solve for $\mathbf{d}_k$:

$$L_k^T \mathbf{d}_k = \mathbf{g}_k$$

   (This is $O(n^2)$ because L_k^T is upper-triangular.)

Result

The search direction $\mathbf{d}_k$ is found in $O(n^2)$ operations.

BFGS Derivation: High-Level Idea

How to find L_{k+1} from L_k? (Notation: $B = B_k, L = L_k, B_+ = B_{k+1}$, etc.)

1. We need $B_+ = L_+ L_+^T$ to satisfy the secant condition $B_+ \delta = \gamma$.
2. We also want L_+ to be "close" to L.

The Derivation

1. Define a general (non-triangular) factor J such that $B = JJ^T$.
2. Find a new factor J_+ that is closest to J in Frobenius norm:

$$\min_{J_+} \|J_+ - J\|_F \quad \text{s.t.} \quad B_+ \boldsymbol{\delta} = \boldsymbol{\gamma}$$

   (This is tricky, so a related problem is solved).
3. The problem solved is: $\min_{J_+} \|J_+ - J\|_F$ s.t. $J_+ \mathbf{g} = \boldsymbol{\gamma}$ for a specific vector $\mathbf{g}$.

BFGS Derivation: Step 1 (Solving for J_+) (optional)

The problem is to find the closest point J_+ to J on an affine subspace.

Minimisation Problem

$$\min_{J_+} \operatorname{tr}((J_+ - J)(J_+ - J)^T) \quad \text{s.t.} \quad J_+\mathbf{g} = \boldsymbol{\gamma}$$

The unique solution J_+^* is found when $(J_+^* - J)$ is orthogonal to the constraint subspace, which gives:

Solution for J_+

$$J_+^* = J + \frac{(\boldsymbol{\gamma} - J\mathbf{g})\mathbf{g}^T}{\mathbf{g}^T\mathbf{g}}$$

This J_+ is a rank-1 update to J. Now we must choose the vector $\mathbf{g}$.

BFGS Derivation: Step 2 (Finding g) (optional)

A second condition, $J_+^T \boldsymbol{\delta} = \mathbf{g}$, is imposed to satisfy the secant condition.

- This implies that $\mathbf{g}$ must be proportional to $J^T \boldsymbol{\delta}$.

$$\mathbf{g} = \beta J^T \boldsymbol{\delta} \quad \text{(for some scalar } \beta \neq 0\text{)}$$

- Substituting this back into the relation for $\mathbf{g}$ and solving for β yields:

$$\beta = \pm \sqrt{\frac{\boldsymbol{\gamma}^T \boldsymbol{\delta}}{\boldsymbol{\delta}^T B \boldsymbol{\delta}}}$$

- **Crucial Check:** Is the square root real?
  - From the Wolfe conditions, we know $\boldsymbol{\gamma}^T \boldsymbol{\delta} > 0$.
  - Since B_k is positive definite, $\boldsymbol{\delta}^T B_k \boldsymbol{\delta} > 0$.
  - Yes, β is a real, non-zero number.

BFGS Derivation: Step 3 (The B_+ Update) (optional) I

Now we form the new Hessian approximation $B_+ = J_+^*(J_+^*)^T$.

$$B_+ = \left(J + \frac{(\dots)\mathbf{g}^T}{\mathbf{g}^T\mathbf{g}}\right)\left(J + \frac{(\dots)\mathbf{g}^T}{\mathbf{g}^T\mathbf{g}}\right)^T$$

After substituting $\mathbf{g} = \beta J^T\boldsymbol{\delta}$ and a lot of algebra, this simplifies to:

$$B_+ = B - \frac{B\boldsymbol{\delta}\boldsymbol{\delta}^T B}{\boldsymbol{\delta}^T B\boldsymbol{\delta}} + \frac{\boldsymbol{\gamma}\boldsymbol{\gamma}^T}{\beta^2\boldsymbol{\delta}^T B\boldsymbol{\delta}}$$

Finally, substituting $\beta^2 = \frac{\boldsymbol{\gamma}^T\boldsymbol{\delta}}{\boldsymbol{\delta}^T B\boldsymbol{\delta}}$:

BFGS Formula (re-derived)

$$B_+ = B - \frac{B\boldsymbol{\delta}\boldsymbol{\delta}^T B}{\boldsymbol{\delta}^T B\boldsymbol{\delta}} + \frac{\boldsymbol{\gamma}\boldsymbol{\gamma}^T}{\boldsymbol{\gamma}^T\boldsymbol{\delta}}$$

BFGS Derivation: Step 3 (The B_+ Update) (optional) II

Key Property

This final formula for B_+ is independent of β and, most importantly, **independent of the choice of J!**

Updating the Cholesky Factor L_k (optional)

- Since the B_+ formula is independent of J, we can choose $J = L_k$, our lower-triangular Cholesky factor.
- The update formula for J_+ was:

$$J_+ = J + \frac{(\gamma - \beta B\delta)\delta^T J}{\beta \delta^T B \delta}$$

- Taking the transpose gives:

$$J_+^T = J^T + \frac{J^T \delta (\gamma - \beta B\delta)^T}{\beta \delta^T B \delta} = L_k^T + (\text{rank-1 update})$$

Problem:

L_k^T is upper-triangular, but adding a rank-1 matrix makes J_+^T *not* triangular.

The QR Factorisation Trick (optional)

We have a matrix J_+^T that is "almost" upper-triangular (it's $R + uv^T$). How do we get our new Cholesky factor L_+ from this?

Proposition: QR Factorisation

Any square matrix X has a unique factorisation $X = QR$, where Q is orthogonal ($Q^TQ = I$) and R is upper-triangular with non-negative diagonal.

1. We have our non-triangular J_+^T.
2. Compute its QR factorisation: $J_+^T = Q_+R_+$.
3. This can be done efficiently in $O(n^2)$ time because J_+^T is almost triangular (using Givens rotations).
4. Now look at B_+:

$$B_+ = J_+J_+^T = (J_+^T)^TJ_+^T = (Q_+R_+)^T(Q_+R_+)$$

$$B_+ = R_+^TQ_+^TQ_+R_+ = R_+^TIR_+ = R_+^TR_+$$

5. **Result:** $B_+ = (R_+^T)(R_+^T)^T$.

Algorithm: The BFGS Method

S0 Choose x_0, L_0 (e.g., $L_0 = I$), and $\epsilon > 0$. Set $k = 0$.

S1 If $\|\nabla f(x_k)\| \leq \epsilon$, **stop**.

S2 **(Solve for d_k in $O(n^2)$)**

- Solve $L_k g_k = -\nabla f(x_k)$ for g_k (forward-sub).
- Solve $L_k^T d_k = g_k$ for d_k (back-sub).

S3 Perform line search for $\alpha_k > 0$ satisfying Wolfe conditions.

S4 Set $\delta_k = \alpha_k d_k$, $x_{k+1} = x_k + \delta_k$. Compute $\gamma_k = \nabla f(x_{k+1}) - \nabla f(x_k)$.

S5 **(Update L_k in $O(n^2)$)**

- Compute $J_{k+1}^T = L_k^T +$ (rank-1 update vector).
- Compute QR factorisation: $J_{k+1}^T = Q_{k+1} R_{k+1}$.
- Set $L_{k+1} = R_{k+1}^T$.

S6 Set $k \leftarrow k + 1$ and go to S1.

BFGS Properties

- The BFGS algorithm spends only $O(n^2)$ computation time per iteration (P6).
- By updating the Cholesky factor L_k, it guarantees that every B_k is symmetric positive definite (P1, P2, P3).
- It uses the secant condition (P4) and is a low-rank update (P5).
- It has local **superlinear** convergence (not quite quadratic, but very fast).
- On a strictly convex quadratic function, it finds the exact minimum in at most n iterations (with exact line search).

Final Comments & Summary Table

Let $\mathcal{C}(f)$ be the cost of one function evaluation, then we have the following table:

Cost-Convergence Summary

Method	**Cost per Iteration**	**Convergence Rate**
Steepest Descent	$O(n\mathcal{C}(f))$	Linear
Quasi-Newton	$O(n^2 + n\mathcal{C}(f))$	Superlinear
Newton-Raphson	$O(n^3 + n^2\mathcal{C}(f))$	Quadratic

The Line Search Problem (S3)

The 1D Problem

Given a starting point $\mathbf{x}_k$ and a descent direction $\mathbf{d}_k$, solve:

$$\min_{\alpha>0} \phi(\alpha) \quad \text{where} \quad \phi(\alpha) = f(\mathbf{x}_k + \alpha \mathbf{d}_k)$$

Why "Practical" Line Search?

Finding the exact minimiser α^* is too computationally expensive. We just need a "good enough" α_k.

What is a "Good" Step Length?

Our α_k needs to balance two competing goals:

1. **Goal 1: Sufficient Decrease.** Avoid tiny, trivial steps.
2. **Goal 2: Sufficient Progress.** Step shouldn't be too small to avoid getting stuck.

The Problem

Just requiring $f(\mathbf{x}_{k+1}) < f(\mathbf{x}_k)$ is not enough to guarantee convergence. The Wolfe conditions provide a "reasonable spot".

The Wolfe Conditions

We need $\alpha_k > 0$ and $0 < c_1 < c_2 < 1$.

Condition 1: Sufficient Decrease (Armijo Rule)

$$f(\mathbf{x}_k + \alpha_k \mathbf{d}_k) \leq f(\mathbf{x}_k) + c_1 \alpha_k \nabla f(\mathbf{x}_k)^T \mathbf{d}_k$$

Rules out steps that are "too long".

Condition 2: Curvature Condition

$$\nabla f(\mathbf{x}_k + \alpha_k \mathbf{d}_k)^T \mathbf{d}_k \geq c_2 \nabla f(\mathbf{x}_k)^T \mathbf{d}_k$$

Rules out steps that are "too short".

- Since $\phi'(0)$ is negative, $c_2\phi'(0)$ is a *less negative* number (e.g., $-0.9 \times$ slope).
- We require the new slope to be greater (less steep) than this.
- This condition rules out α values that are "too short".

Practical Line Search Algorithms I

Backtracking Line Search

A simple and popular method that often only enforces the first condition (Armijo):

1. Start with a full step: $\alpha = 1$ (the "ideal" QN/Newton step).
2. Choose $\rho \in (0, 1)$ (e.g., $\rho = 0.5$) and c_1 (e.g., $c_1 = 10^{-4}$).
3. **while** $f(x_k + \alpha d_k) > f(x_k) + c_1 \alpha \nabla f(x_k)^T d_k$:
4. $\quad \alpha \leftarrow \rho \alpha$ (reduce step size)
5. **end while**
6. Set $\alpha_k = \alpha$.

Practical Line Search Algorithms II

Full Wolfe Conditions

- More complex algorithms (using bisection and interpolation) can find an α_k that satisfies *both* conditions.
- Enforcing both conditions is crucial for proving the global convergence of algorithms like BFGS.

Drawbacks of BFGS: The Scaling Issue

While BFGS has excellent convergence properties, it faces significant challenges as the dimension n grows large (e.g., in Machine Learning or deep learning).

The Memory Bottleneck

BFGS requires storing the approximate Hessian B_k (or its inverse/Cholesky factors) as a dense $n \times n$ matrix.

- **Memory Complexity:** $O(n^2)$.

Practical Example

Consider an optimization problem with $n = 100,000$ variables:

- The matrix contains 10^{10} entries.
- Using double precision (8 bytes per entry), this requires $\approx$ **80** GB of RAM.
- For $n = 1,000,000$, this becomes prohibitive (≈ 8 TB).

Introducing L-BFGS (Limited-Memory BFGS)

To tackle large-scale problems, we need a method that approximates Newton steps without the massive memory footprint.

The Core Idea

Instead of storing the full matrix H_k, we implicitly store the approximation using a history of recent updates.

Storage Strategy

Recall that the BFGS update is constructed using the curvature pairs:

$$\boldsymbol{\delta}_k = \mathbf{x}_{k+1} - \mathbf{x}_k \quad \text{and} \quad \boldsymbol{\gamma}_k = \nabla f(\mathbf{x}_{k+1}) - \nabla f(\mathbf{x}_k)$$

L-BFGS stores only the **most recent** m **pairs** $\{(\boldsymbol{\delta}_i, \boldsymbol{\gamma}_i)\}$ (where m is small, e.g., $m \approx 10$ to 20). Older information is discarded.

How L-BFGS Works

Implicit Multiplication

We never explicitly form the matrix H_k.

- To compute the search direction $\mathbf{d}_k = -H_k \nabla f(\mathbf{x}_k)$, we use a specialized **"Two-Loop Recursion"** algorithm.
- This algorithm performs the matrix-vector product using only the stored pairs $\{(\boldsymbol{\delta}_i, \boldsymbol{\gamma}_i)\}$ and the current gradient.

L-BFGS: The Two-Loop Recursion

Goal: Compute the search direction $\mathbf{d}_k = -H_k \nabla f(\mathbf{x}_k)$ efficiently.

The Challenge

We cannot compute $\mathbf{d}_k$ directly because we **do not store** the $n \times n$ matrix H_k. We only store m vector pairs $\{(\boldsymbol{\delta}_i, \boldsymbol{\gamma}_i)\}$.

The Core Idea

The matrix H_k is *implicitly* defined by applying the BFGS inverse update formula m times to an initial "seed" matrix H_k^0 (e.g., $H_k^0 = I$).
The inverse update formula is:

$$H_{i+1} = (I - \rho_i \boldsymbol{\delta}_i \boldsymbol{\gamma}_i^T) H_i (I - \rho_i \boldsymbol{\gamma}_i \boldsymbol{\delta}_i^T) + \rho_i \boldsymbol{\delta}_i \boldsymbol{\delta}_i^T$$

where $\rho_i = 1/(\boldsymbol{\gamma}_i^T \boldsymbol{\delta}_i)$.
The two-loop algorithm "unrolls" this recursion to compute the matrix-vector product $H_k \nabla f(\mathbf{x}_k)$ without ever forming H_k.

L-BFGS: The Algorithm I

This computes $\mathbf{r} = H_k \mathbf{q}$, where $\mathbf{q} = \nabla f(\mathbf{x}_k)$.

Input: Gradient $\mathbf{q} = \nabla f(\mathbf{x}_k)$, history $\{(\boldsymbol{\delta}_i, \boldsymbol{\gamma}_i)\}_{i=k-m}^{k-1}$.

Loop 1: Backward Pass

(Iterates from newest $k-1$ to oldest $k-m$)

for $i = k-1$ **down to** $k-m$

$\rho_i = 1/(\boldsymbol{\gamma}_i^T \boldsymbol{\delta}_i)$

$\alpha_i = \rho_i(\boldsymbol{\delta}_i^T \mathbf{q})$ (*Store this scalar*)

$\mathbf{q} = \mathbf{q} - \alpha_i \boldsymbol{\gamma}_i$

Middle Step: Apply Initial Hessian

$\mathbf{r} = H_k^0 \mathbf{q}$ (*e.g., $\mathbf{r} = \mathbf{q}$ if $H_k^0 = I$*)

L-BFGS: The Algorithm II

Loop 2: Forward Pass

(Iterates from oldest $k-m$ to newest $k-1$)

for $i = k-m$ **up to** $k-1$

$\beta = \rho_i(\boldsymbol{\gamma}_i^T \mathbf{r})$

$\mathbf{r} = \mathbf{r} + (\alpha_i - \beta)\boldsymbol{\delta}_i$ (*Use stored α_i*)

Output: The search direction is $\mathbf{d}_k = -\mathbf{r}$.

L-BFGS: Why It Works I

A Nested Expression

Multiplying $H_k\mathbf{q}$ is like evaluating a nested expression from the inside out.
$H_k \approx (\dots(V_{k-1}^T H_k^0 V_{k-1} + \dots)\dots)$
Where $V_i = (I - \rho_i \gamma_i \delta_i^T)$ and we also add the rank-1 terms.

Visualizing the Loops

- **Loop 1 (Backward Pass):** This pass moves from the "outside" of the nested expression to the "inside". It recursively applies the V_i terms (right-multiplies) to the gradient $\mathbf{q}$ until it reaches the H_k^0 core.
- **Middle Step:** The "core" matrix H_k^0 is applied.
- **Loop 2 (Forward Pass):** This pass moves back "outward". It applies the V_i^T terms (left-multiplies) and adds the rank-1 $\delta_i \delta_i^T$ components, using the α_i values saved from the first loop.

L-BFGS: Why It Works II

The Payoff

This entire operation uses only vector dot products and additions.

- No $n \times n$ matrices are ever formed or stored.
- **Computation Cost:** $O(mn)$
- **Storage Cost:** $O(mn)$

BFGS vs L-BFGS

Comparison: BFGS vs. L-BFGS

Feature	**Standard BFGS**	**L-BFGS**
Memory Usage	$O(n^2)$	$O(mn)$ (Linear!)
CPU per Iteration	$O(n^2)$	$O(mn)$
Convergence	Superlinear	Linear (but fast)
Use Case	Small/Medium n	Large scale $n > 1000$

Non-Linear LS: the Levenberg-Marquardt algorithm.

Goal

We want to find the parameters $\mathbf{w}$ that best fit a non-linear model $f(x, \mathbf{w})$ to a set of data points (x_i, y_i).

Non-Linear LS: the Levenberg-Marquardt algorithm.

Goal

We want to find the parameters $\mathbf{w}$ that best fit a non-linear model $f(x, \mathbf{w})$ to a set of data points (x_i, y_i).

Mathematical Formulation

We aim to minimize the sum of squared residuals. Let's define the residuals as:

$$r_i(\mathbf{w}) = f(x, \mathbf{w}) - y_i$$

The total cost function $f(\mathbf{w})$ to minimize is the sum of their squares:

$$f(\mathbf{w}) = \sum_{i=1}^{m} [r_i(\mathbf{w})]^2 = \boldsymbol{r}(\mathbf{w})^\top \boldsymbol{r}(\mathbf{w}) = \|\boldsymbol{r}(\mathbf{w})\|^2$$

Non-Linear LS: the Levenberg-Marquardt algorithm.

Goal

We want to find the parameters $\mathbf{w}$ that best fit a non-linear model $f(x, \mathbf{w})$ to a set of data points (x_i, y_i).

Mathematical Formulation

We aim to minimize the sum of squared residuals. Let's define the residuals as:

$$r_i(\mathbf{w}) = f(x, \mathbf{w}) - y_i$$

The total cost function $f(\mathbf{w})$ to minimize is the sum of their squares:

$$f(\mathbf{w}) = \sum_{i=1}^{m} [r_i(\mathbf{w})]^2 = \boldsymbol{r}(\mathbf{w})^\top \boldsymbol{r}(\mathbf{w}) = \|\boldsymbol{r}(\mathbf{w})\|^2$$

The problem becomes: $\boxed{\min_{\mathbf{w}} f(\mathbf{w})}$

Two Approaches

1. Gradient Descent

- **Idea:** Move in the direction of the steepest descent of $f(\mathbf{w})$.
- **Update:** $\mathbf{w}_{k+1} = \mathbf{w}_k - \gamma \nabla f(\mathbf{w}_k)$
- **Pros:** Always converges to a minimum (stable).
- **Cons:** Very slow convergence, especially near the minimum. Only uses 1st-order info.

Two Approaches

1. Gradient Descent

- **Idea:** Move in the direction of the steepest descent of $f(\mathbf{w})$.
- **Update:** $\mathbf{w}_{k+1} = \mathbf{w}_k - \gamma \nabla f(\mathbf{w}_k)$
- **Pros:** Always converges to a minimum (stable).
- **Cons:** Very slow convergence, especially near the minimum. Only uses 1st-order info.

2. Newton's Method

- **Idea:** Build a 2nd-order (quadratic) model of $f(\mathbf{w})$ at each step and jump to its minimum.
- **Update:** $\mathbf{w}_{k+1} = \mathbf{w}_k - H^{-1} \nabla f(\mathbf{w}_k)$
- **Pros:** Very fast (quadratic) convergence.
- **Cons:** Requires computing the full Hessian matrix H, which is complex and computationally very expensive.

The challenge

Aims

We want to:

- have the speed of Newton's method;
- avoid the computation of the full Hessian.

Derivation: the Gradient

We start with the goal of applying Newton's method, which requires the gradient ∇f and the Hessian $H = \nabla^2 f$.

1. The Gradient ∇f

The cost is $f = \sum_i r_i^2$. The j-th component of the gradient is:

$$\frac{\partial f}{\partial w_j} = \sum_{i=1}^{m} 2 r_i \frac{\partial r_i}{\partial w_j}$$

Let J be the **Jacobian matrix** of the residuals $\boldsymbol{r}$, with entries $J_{ij} = \partial r_i / \partial w_j$.
In vector form, the gradient is:

$$\nabla f(\mathbf{w}) = 2 J^\top \boldsymbol{r}(\mathbf{w})$$

Derivation: the Hessian

2. The Full Hessian H

Differentiating the gradient again (with $H_{jk} = \partial^2 f / \partial w_j \partial w_k$) gives:

$$H_{jk} = \sum_{i=1}^{m} 2 \left(\frac{\partial r_i}{\partial w_k} \frac{\partial r_i}{\partial w_j} + r_i \frac{\partial^2 r_i}{\partial w_j \partial w_k} \right)$$

In matrix form:

$$H = 2J^\top J + 2 \sum_{i=1}^{m} r_i \nabla^2 r_i$$

The Gauss-Newton Approximation I

The full Hessian is:

$$H = \underbrace{2J^\top J}_{\text{First-order term}} + \underbrace{2\sum_{i=1}^{m} r_i \nabla^2 r_i}_{\text{Second-order term}}$$

The Core Idea of Gauss-Newton

Approximate the Hessian by **dropping the second-order term**.

$$H \approx 2J^\top J$$

The Gauss-Newton Approximation II

When is this a good approximation?

The second term $2\sum r_i \nabla^2 r_i$ is small if...

- **The residuals r_i are small.** This happens when the model is a good fit, i.e., we are **near the minimum**. This is why GN converges so fast near the solution.
- **The model is "almost linear".** If $f(\mathbf{w})$ is close to linear, its second derivatives $\nabla^2 r_i$ are close to zero.

The Gauss-Newton Update Rule

1. **Start** with the Newton step:

$$H\delta = -\nabla f$$

The Gauss-Newton Update Rule

1. **Start** with the Newton step:

$$H\boldsymbol{\delta} = -\nabla f$$

2. **Substitute** the gradient $\nabla f = 2J^\top \boldsymbol{r}$:

$$H\boldsymbol{\delta} = -2J^\top \boldsymbol{r}$$

The Gauss-Newton Update Rule

1. **Start** with the Newton step:

$$H\boldsymbol{\delta} = -\nabla f$$

2. **Substitute** the gradient $\nabla f = 2J^\top \boldsymbol{r}$:

$$H\boldsymbol{\delta} = -2J^\top \boldsymbol{r}$$

3. **Substitute** the approximate Hessian $H \approx 2J^\top J$:

$$(2J^\top J)\boldsymbol{\delta} \approx -2J^\top \boldsymbol{r}$$

The Gauss-Newton Update Rule

1. **Start** with the Newton step:

$$H\boldsymbol{\delta} = -\nabla f$$

2. **Substitute** the gradient $\nabla f = 2J^\top \boldsymbol{r}$:

$$H\boldsymbol{\delta} = -2J^\top \boldsymbol{r}$$

3. **Substitute** the approximate Hessian $H \approx 2J^\top J$:

$$(2J^\top J)\boldsymbol{\delta} \approx -2J^\top \boldsymbol{r}$$

4. **Simplify** to get the **Gauss-Newton normal equations**:

$$(J^\top J)\boldsymbol{\delta} = -J^\top \boldsymbol{r}$$

The Gauss-Newton Update Rule

1. **Start** with the Newton step:

$$H\boldsymbol{\delta} = -\nabla f$$

2. **Substitute** the gradient $\nabla f = 2J^\top \boldsymbol{r}$:

$$H\boldsymbol{\delta} = -2J^\top \boldsymbol{r}$$

3. **Substitute** the approximate Hessian $H \approx 2J^\top J$:

$$(2J^\top J)\boldsymbol{\delta} \approx -2J^\top \boldsymbol{r}$$

4. **Simplify** to get the **Gauss-Newton normal equations**:

$$(J^\top J)\boldsymbol{\delta} = -J^\top \boldsymbol{r}$$

The Algorithm

At each step k, solve for the update $\boldsymbol{\delta}_k$ and apply it:

$$\mathbf{w}_{k+1} = \mathbf{w}_k + \boldsymbol{\delta}_k$$

Gauss-Newton: Advantages & Drawbacks

Advantages

- **Fast Convergence:** If it converges, it does so quadratically (like full Newton) when near the minimum.
- **No Hessian Needed:** It only requires the Jacobian J, not the full, complex Hessian matrix H.
- $J^\top J$ is "cheaper" to compute than H.

Gauss-Newton: Advantages & Drawbacks

Advantages

- **Fast Convergence:** If it converges, it does so quadratically (like full Newton) when near the minimum.
- **No Hessian Needed:** It only requires the Jacobian J, not the full, complex Hessian matrix H.
- $J^\top J$ is "cheaper" to compute than H.

Drawbacks

- **May Diverge:** If the initial guess is poor (far from minimum), the r_i are large, the Hessian approximation is bad, and the method can be unstable.
- **Singularity:** The matrix $J^\top J$ may be singular or ill-conditioned, meaning we cannot solve for the step $\boldsymbol{\delta}$. This happens if the Jacobian J does not have full column rank.

The Dilemma

We want the **stability** of GD but the **speed** of GN.

The Core Idea: A "Damped" Approach

Levenberg-Marquardt (LM)

The LM algorithm adaptively interpolates between Gradient Descent and the Gauss-Newton method.

The Core Idea: A "Damped" Approach

Levenberg-Marquardt (LM)

The LM algorithm adaptively interpolates between Gradient Descent and the Gauss-Newton method.

It modifies the Gauss-Newton equation by adding a "damping" term λ:

The LM Update Equation

$$(J^\top J + \lambda I)\boldsymbol{\delta} = -J^\top \boldsymbol{r}$$

The Core Idea: A "Damped" Approach

Levenberg-Marquardt (LM)

The LM algorithm adaptively interpolates between Gradient Descent and the Gauss-Newton method.

It modifies the Gauss-Newton equation by adding a "damping" term λ:

The LM Update Equation

$$(J^\top J + \lambda I)\boldsymbol{\delta} = -J^\top \boldsymbol{r}$$

Where:

- $\boldsymbol{\delta}$ is the update step ($\mathbf{w}_{k+1} = \mathbf{w}_k + \boldsymbol{\delta}$).
- $\lambda \geq 0$ is the **damping parameter**.
- I is the identity matrix.

The Core Idea: A "Damped" Approach

Levenberg-Marquardt (LM)

The LM algorithm adaptively interpolates between Gradient Descent and the Gauss-Newton method.

It modifies the Gauss-Newton equation by adding a "damping" term λ:

The LM Update Equation

$$(J^\top J + \lambda I)\boldsymbol{\delta} = -J^\top \boldsymbol{r}$$

Where:

- $\boldsymbol{\delta}$ is the update step ($\mathbf{w}_{k+1} = \mathbf{w}_k + \boldsymbol{\delta}$).
- $\lambda \geq 0$ is the **damping parameter**.
- I is the identity matrix.

How this solves the GN drawbacks:

By adding λI, the matrix $(J^\top J + \lambda I)$ is **always invertible** (for $\lambda > 0$), even if $J^\top J$ is singular.

How the Damping Parameter λ Works I

The LM update is: $\boldsymbol{\delta} = -(J^\top J + \lambda I)^{-1} J^\top \boldsymbol{r}$

How the Damping Parameter λ Works I

The LM update is: $\boldsymbol{\delta} = -(J^\top J + \lambda I)^{-1} J^\top \boldsymbol{r}$

Case 1: λ is Small (e.g., $\lambda \to 0$)

$$(J^\top J + \lambda I) \approx J^\top J$$

The update becomes:

$$\boldsymbol{\delta} \approx -(J^\top J)^{-1} J^\top \boldsymbol{r}$$

This is the Gauss-Newton method!

- **Behavior:** Fast, but potentially unstable.
- **When:** Used when the current steps are good and reducing the error f.

How the Damping Parameter λ Works I

Case 2: λ is Large (e.g., $\lambda \to \infty$)

For large λ, the $J^\top J$ term is negligible:

$$(J^\top J + \lambda I) \approx \lambda I$$

The update becomes:

$$\boldsymbol{\delta} \approx -(\lambda I)^{-1} J^\top \boldsymbol{r} = -(1/\lambda) J^\top \boldsymbol{r}$$

Recall $\nabla f = 2J^\top \boldsymbol{r}$. So $\boldsymbol{\delta} \approx -(1/2\lambda)\nabla f$.

This is Gradient Descent!

- **Behavior:** Slow, but stable. Takes small steps in the steepest direction.
- **When:** Used when steps are bad (increasing the error f).

The Algorithm: An Iterative Process

1. Choose an initial guess $\mathbf{w}_0$ and an initial damping λ_0.

The Algorithm: An Iterative Process

1. Choose an initial guess $\mathbf{w}_0$ and an initial damping λ_0.
2. **At each iteration** k**:**
   - Compute residuals $\boldsymbol{r}(\mathbf{w}_k)$ and Jacobian $J(\mathbf{w}_k)$.
   - Compute the current error $f(\mathbf{w}_k)$.
   - **Solve** $(J^\top J + \lambda_k I)\boldsymbol{\delta}_k = -J^\top \boldsymbol{r}$ for the step $\boldsymbol{\delta}_k$.
   - Calculate a candidate new point: $\mathbf{w}_{\text{new}} = \mathbf{w}_k + \boldsymbol{\delta}_k$.
   - Evaluate the new error $f(\mathbf{w}_{\text{new}})$.

The Algorithm: An Iterative Process

1. Choose an initial guess $\mathbf{w}_0$ and an initial damping λ_0.
2. **At each iteration k:**
   - Compute residuals $\boldsymbol{r}(\mathbf{w}_k)$ and Jacobian $J(\mathbf{w}_k)$.
   - Compute the current error $f(\mathbf{w}_k)$.
   - **Solve** $(J^\top J + \lambda_k I)\boldsymbol{\delta}_k = -J^\top \boldsymbol{r}$ for the step $\boldsymbol{\delta}_k$.
   - Calculate a candidate new point: $\mathbf{w}_{\text{new}} = \mathbf{w}_k + \boldsymbol{\delta}_k$.
   - Evaluate the new error $f(\mathbf{w}_{\text{new}})$.
3. **Update Strategy (The Core Logic):**
   - **If $f(\mathbf{w}_{\text{new}}) < f(\mathbf{w}_k)$ (Good Step):**
     - Accept the step: $\mathbf{w}_{k+1} \leftarrow \mathbf{w}_{\text{new}}$.
     - Trust the model more: **Decrease** λ (e.g., $\lambda_{k+1} = \lambda_k/\nu$, with $\nu > 1$). This moves towards Gauss-Newton.
   - **If $f(\mathbf{w}_{\text{new}}) \geq f(\mathbf{w}_k)$ (Bad Step):**
     - Reject the step: $\mathbf{w}_{k+1} \leftarrow \mathbf{w}_k$.
     - Trust the model less: **Increase** λ (e.g., $\lambda_{k+1} = \lambda_k \times \nu$). This moves towards Gradient Descent.

The Algorithm: An Iterative Process

1. Choose an initial guess $\mathbf{w}_0$ and an initial damping λ_0.
2. **At each iteration k:**
   - Compute residuals $\boldsymbol{r}(\mathbf{w}_k)$ and Jacobian $J(\mathbf{w}_k)$.
   - Compute the current error $f(\mathbf{w}_k)$.
   - **Solve** $(J^\top J + \lambda_k I)\boldsymbol{\delta}_k = -J^\top \boldsymbol{r}$ for the step $\boldsymbol{\delta}_k$.
   - Calculate a candidate new point: $\mathbf{w}_{\text{new}} = \mathbf{w}_k + \boldsymbol{\delta}_k$.
   - Evaluate the new error $f(\mathbf{w}_{\text{new}})$.
3. **Update Strategy (The Core Logic):**
   - **If $f(\mathbf{w}_{\text{new}}) < f(\mathbf{w}_k)$ (Good Step):**
     - Accept the step: $\mathbf{w}_{k+1} \leftarrow \mathbf{w}_{\text{new}}$.
     - Trust the model more: **Decrease** λ (e.g., $\lambda_{k+1} = \lambda_k/\nu$, with $\nu > 1$). This moves towards Gauss-Newton.
   - **If $f(\mathbf{w}_{\text{new}}) \geq f(\mathbf{w}_k)$ (Bad Step):**
     - Reject the step: $\mathbf{w}_{k+1} \leftarrow \mathbf{w}_k$.
     - Trust the model less: **Increase** λ (e.g., $\lambda_{k+1} = \lambda_k \times \nu$). This moves towards Gradient Descent.
4. Repeat until convergence (e.g., step size $\boldsymbol{\delta}$ is tiny, or error f stops decreasing).

Advantages of Levenberg-Marquardt

Robustness

It finds a solution even if the initial guess is far from the optimal minimum. By increasing λ, it can navigate "difficult" regions of the parameter space where Gauss-Newton would fail.

Advantages of Levenberg-Marquardt

Robustness

It finds a solution even if the initial guess is far from the optimal minimum. By increasing λ, it can navigate "difficult" regions of the parameter space where Gauss-Newton would fail.

Efficiency

It combines the best of both worlds:

- Far from the minimum, it acts like stable Gradient Descent.
- Close to the minimum, it acts like fast-converging Gauss-Newton.

Advantages of Levenberg-Marquardt

Robustness

It finds a solution even if the initial guess is far from the optimal minimum. By increasing λ, it can navigate "difficult" regions of the parameter space where Gauss-Newton would fail.

Efficiency

It combines the best of both worlds:

- Far from the minimum, it acts like stable Gradient Descent.
- Close to the minimum, it acts like fast-converging Gauss-Newton.

Handles Ill-Conditioning

The damping term λI **solves the main drawback of Gauss-Newton**. The matrix $(J^\top J + \lambda I)$ is always invertible (as long as $\lambda > 0$).

Drawbacks and Considerations

Computational Cost

- Requires computing (or approximating) the **Jacobian matrix** J at every iteration, which can be expensive for models with many parameters and data points.
- Requires solving an $n \times n$ linear system (where n is # of parameters) at each iteration, an $O(n^3)$ operation.

Drawbacks and Considerations

Computational Cost

- Requires computing (or approximating) the **Jacobian matrix** J at every iteration, which can be expensive for models with many parameters and data points.
- Requires solving an $n \times n$ linear system (where n is # of parameters) at each iteration, an $O(n^3)$ operation.

Local Minimum

Like Gradient Descent and Gauss-Newton, LM is a **local optimizer**. It is only guaranteed to find a local minimum, not the global one. The final solution can depend heavily on the initial guess $\mathbf{w}_0$.

Drawbacks and Considerations

Computational Cost

- Requires computing (or approximating) the **Jacobian matrix** J at every iteration, which can be expensive for models with many parameters and data points.
- Requires solving an $n \times n$ linear system (where n is # of parameters) at each iteration, an $O(n^3)$ operation.

Local Minimum

Like Gradient Descent and Gauss-Newton, LM is a **local optimizer**. It is only guaranteed to find a local minimum, not the global one. The final solution can depend heavily on the initial guess $\mathbf{w}_0$.

Parameter Tuning

The performance can be sensitive to the initial value λ_0 and the update factor ν, though standard, "good-enough" values are widely used.

Summary

- The Levenberg-Marquardt algorithm is the **de-facto standard** for solving non-linear least squares problems.

Summary

- The Levenberg-Marquardt algorithm is the **de-facto standard** for solving non-linear least squares problems.
- It improves on Gauss-Newton by adding a **damping parameter** λ that makes the update step robust and invertible.

Summary

- The Levenberg-Marquardt algorithm is the **de-facto standard** for solving non-linear least squares problems.
- It improves on Gauss-Newton by adding a **damping parameter** λ that makes the update step robust and invertible.
- It **adaptively interpolates** between:
  - Slow-but-stable **Gradient Descent** (when λ is large).
  - Fast-but-unstable **Gauss-Newton** (when λ is small).

Summary

- The Levenberg-Marquardt algorithm is the **de-facto standard** for solving non-linear least squares problems.
- It improves on Gauss-Newton by adding a **damping parameter** λ that makes the update step robust and invertible.
- It **adaptively interpolates** between:
  - Slow-but-stable **Gradient Descent** (when λ is large).
  - Fast-but-unstable **Gauss-Newton** (when λ is small).
- It is robust, efficient, and handles ill-conditioned problems, but can be costly to compute and may get stuck in local minima.