# BFGS
# & L-BFGS

-

**PALISSON Antoine**

# Definition

The **BFGS** method – **B**royden, **F**letcher, **G**oldfarb, and **S**hanno – is a **quasi-Newton** iterative optimization algorithm used in solving **unconstrained nonlinear** optimization problems.

### Non-linear

Problems with a nonlinear objective functions and/or constraints.

### Unconstrained

There are no constraints on the variables. The objective is simply to find the maximum/minimum.

### Deterministic

The outcome of the optimization is determined solely by the algorithm's input.

### Continuous

The variables can take any value within a given range.

### Local

The goal is to find the best solution within a limited region of the space i.e. the local optimum.

### Single-Objective

A single objective function needs to be optimized at the same time.

# Quasi-Newton ? Quasi ? Newton ?

To understand the BFGS method, we must first delve into the concept of **quasi-Newton methods**.

The **“quasi”** part of the name hints at an approximation approach, which is key to understanding their functionality. What about the **“Newton”** part ? To understand it, it's essential to start with the **Newton-Raphson method** which is a specific application of Newton's Method used to find roots of a function.

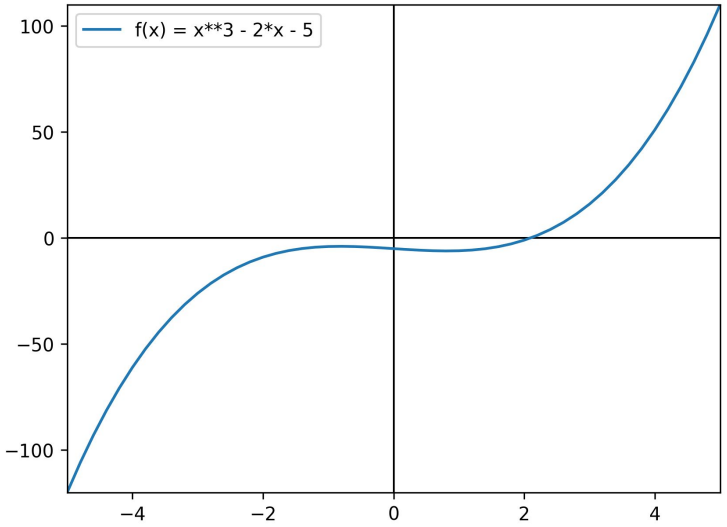
**Newton's Method** extends the ideas of the Newton-Raphson approach to find the optima – either minimum or maximum – of functions.

Thus, we will start with the Newton-Raphson method, evolves through Newton's Method and ends with the understanding of quasi-Newton methods – and especially the BFGS algorithm.

This progression not only illustrates the development of optimization algorithms but also highlights the interconnectedness of mathematical concepts in solving complex real-world problems.

# Newton Raphson Method

The **Newton-Raphson method** is a powerful and widely used tool for finding roots of real-valued functions. It is an iterative approach, meaning it starts with an initial guess and refines this guess through repeated iterations. When the method converges, it usually does so very quickly, especially when the initial guess is close to the actual root. However, convergence is not guaranteed. The method can fail to converge if the function is not well-behaved or if the initial guess is not close enough to the actual root.




A root of a function is a value at which the function equals zero.

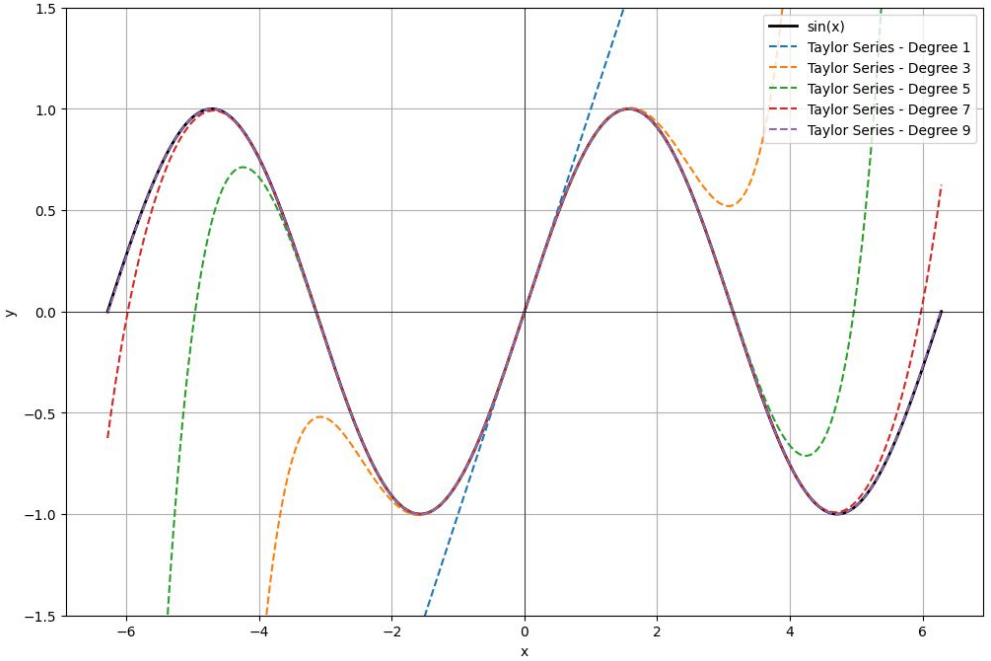
$$x^3 - 2x - 5 = 0$$

**What's the root of this function ?**

# Newton Raphson Method – Taylor Series Expansion

To understand how the Newton Raphson method works we first have to understand what the **Taylor Series Expansion** is. It is a way to represent a function as an infinite sum of terms calculated from the values of its derivatives at a single point. For a function **f(x)**, its Taylor series expansion around a point **a** is given by:

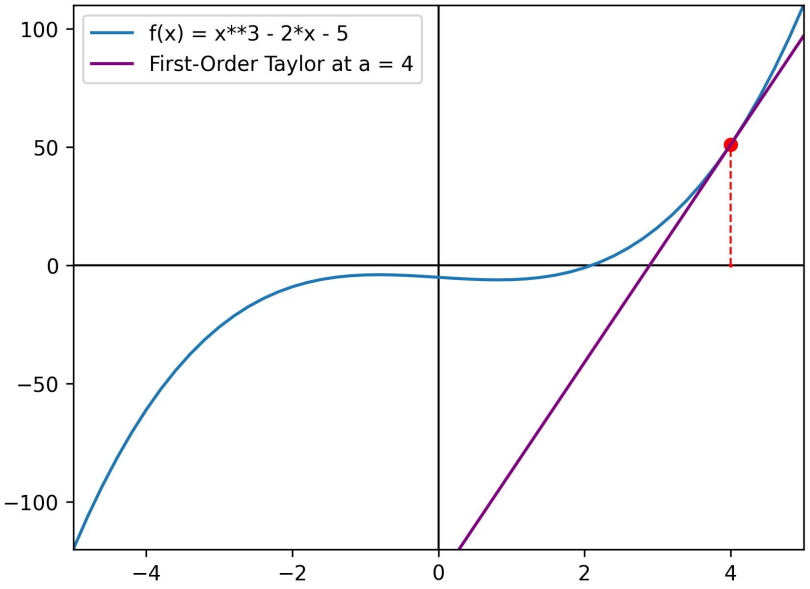
$$f(x) = f(a) + f'(a)(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f'''(a)}{3!}(x-a)^3 + \cdots$$




***sin x** and its Taylor approximations by polynomials of degree 1, 3, 5, 7 and 9 at **a = 0**.*

# Newton Raphson Method – First-Order Taylor Expansion

The **first order of the Taylor series** is the linear approximation of a function around a specific point. This is achieved by truncating the Taylor series after the first derivative term. The first-order Taylor expansion of a function **f(x)** around a point **a** is given by the following formula. It is essentially the equation of the tangent line to f(x) at a.




$$f(x) \approx f(a) + f'(a)(x - a)$$

This is a linear equation with f'(a) as the slope and f(a) as the bias.

The derivative of f(x) tells us the **slope of the function at a** which indicates how f(x) changes as x changes. If it is positive, the function is increasing at a – if negative, it's decreasing.

**How can we use this equation to find the root of f(x) ?**

# Newton Raphson Method

To find the root of the function, we need to solve this equation:

$$f(x) = 0$$
$$x^3 - 2x - 5 = 0$$

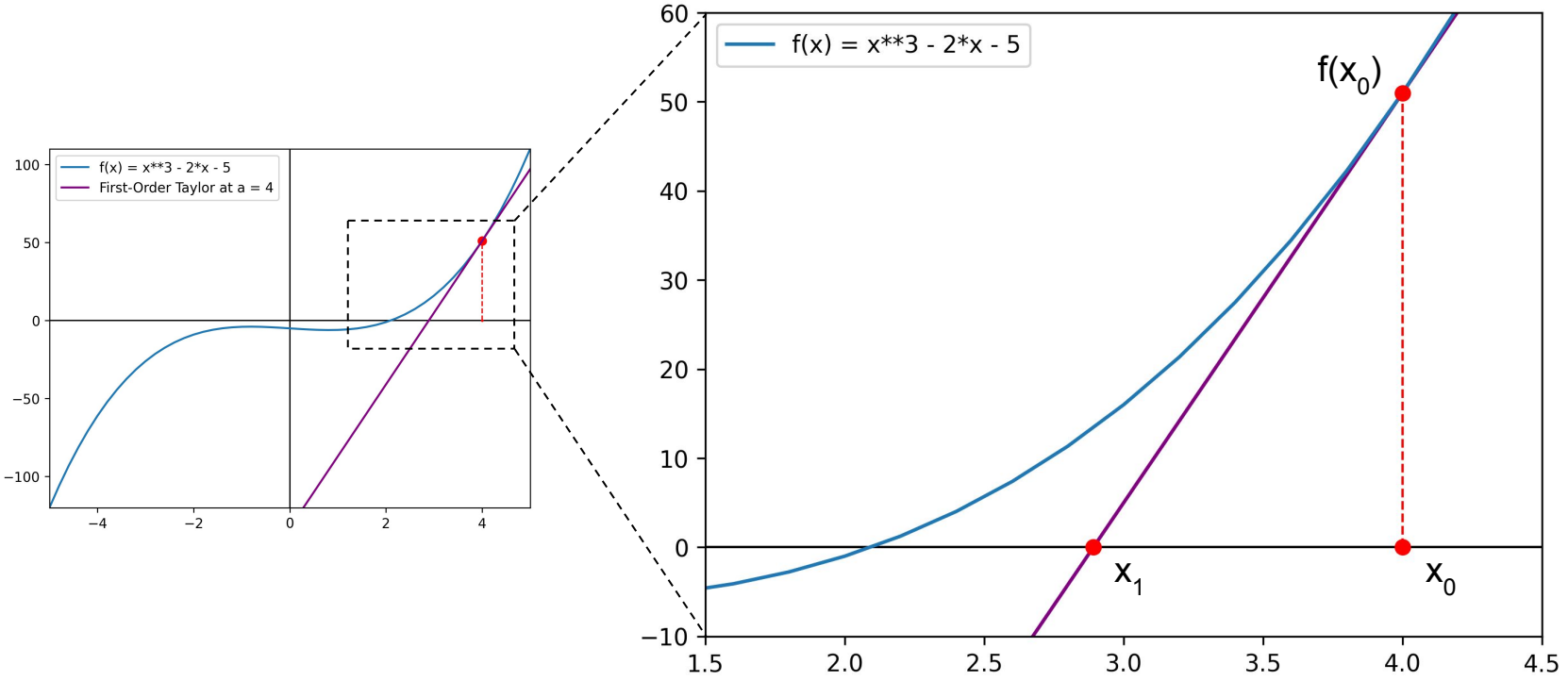
Using the **Taylor first-order** approximation, we get:

$$f(x) \approx 0$$
$$f(a) + f'(a)(x - a) \approx 0$$
$$x \approx a - \frac{f(a)}{f'(a)}$$

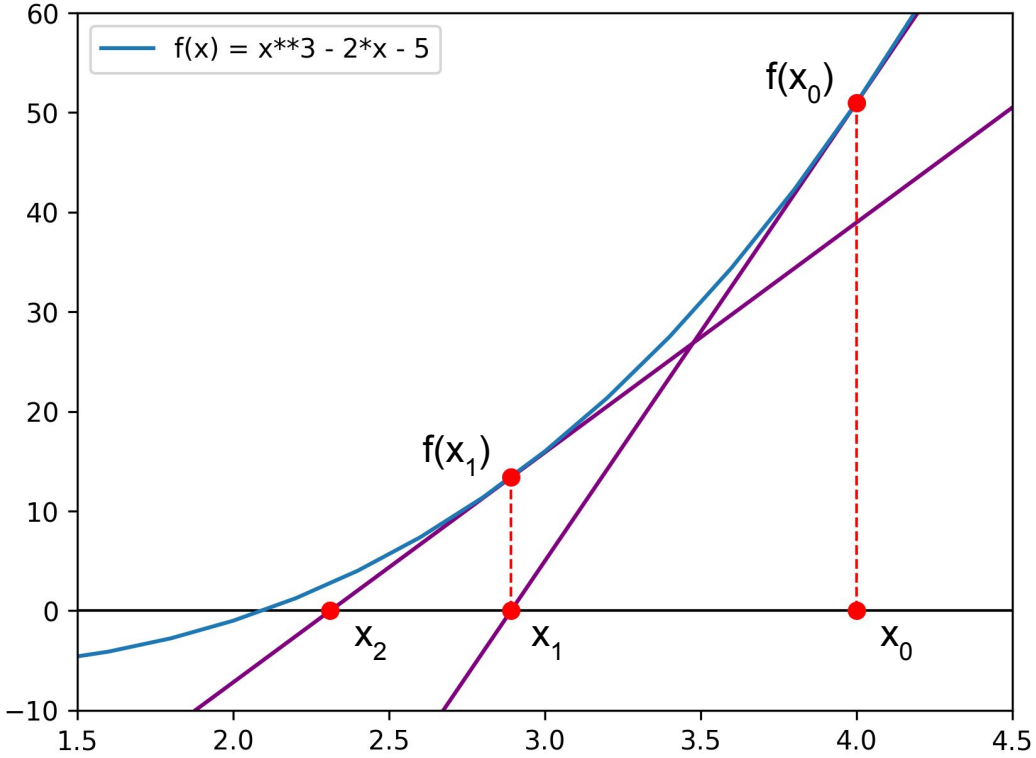
In this equation, x is the point where the function crosses the x-axis. Thus, x is an approximation of the root of the function.

# Newton Raphson Method

This first approximation can then be used as a new guess in order to refine the approximation.



# Newton Raphson Method




$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

# Newton's Method for Optimization

**Newton's method in optimization** is an extension of the basic Newton-Raphson method used for finding the roots of equations. In the context of optimization, it is used to find the minima or maxima of a function. The method identifies points where the **Gradient** is zero and use the **Hessian** to determine the nature of these points.

Similarly to the Newton Raphson method, the **Newton's method in optimization** is derived from the principles of Taylor series expansion but now, we are interested in both the first order and the second order derivatives.

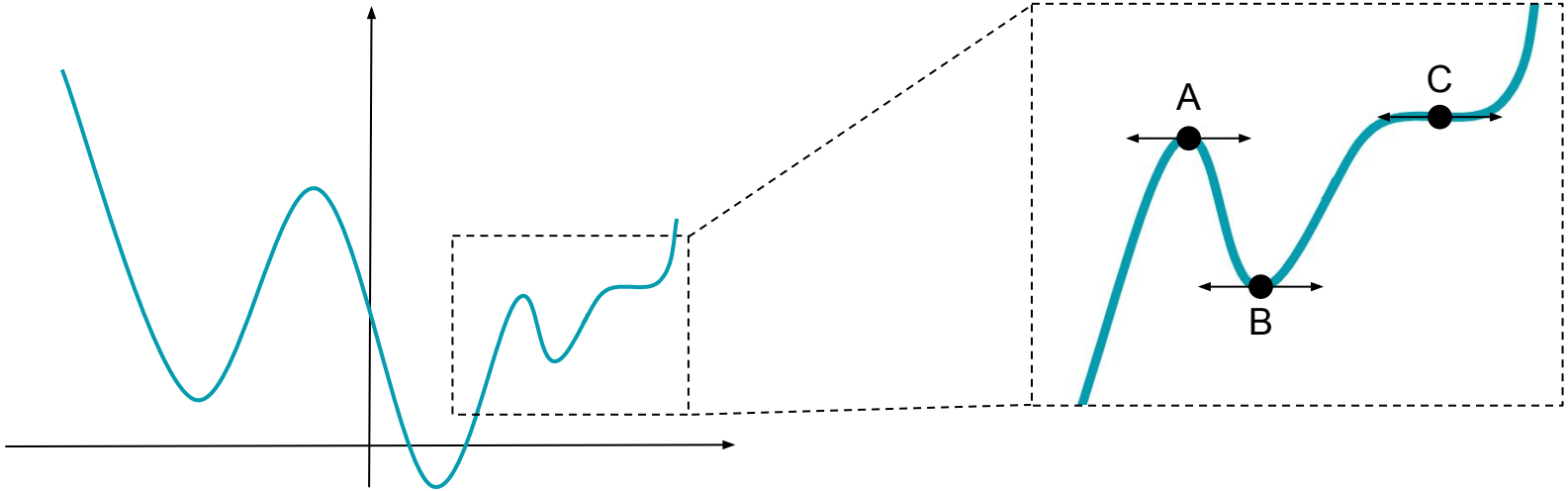
The **second derivative** is used to describe the local curvature of the function which helps in visualizing its "shape" around a point – i.e. how "sharp" or "flat" the curve is at that point. It's the rate of change of the slope.

$$f(x) \approx f(x_n) + f'(x_n)(x - x_n) + \frac{1}{2}f''(x_n)(x - x_n)^2$$

**First** derivative of the function

**Second** derivative of the function

# Newton's Method for Optimization




When the **gradient of a function is equal to zero**, it indicates that we've reached a **stationary point** meaning there's no slope in any direction. These points can be classified into several types depending on the nature of the function at these points:

- **A** – it's a local **maximum**, the second derivative is **negative**.
- **B** – it's a local **minimum**, the second derivative is **positive**.
- **C** – the point could be a **saddle point** or a higher-order stationary point

# Newton's Method – Formula

Similarly to Gradient Descent, to optimize **f(x)**, we're interested in finding where its **first derivative** is **zero**. This first derivative can be approximated by differentiating the Taylor Series of the function.

$$f(x) \approx f(x_n) + f'(x_n)(x - x_n) + \frac{1}{2}f''(x_n)(x - x_n)^2$$

$$\frac{\partial f(x)}{\partial x} \approx \frac{\partial f(x_n)}{\partial x} + \frac{\partial \left(f'(x_n)(x - x_n)\right)}{\partial x} + \frac{\partial \left(\frac{1}{2}f''(x_n)(x - x_n)^2\right)}{\partial x}$$

This is a constant with respect to x, so its **derivative is 0**

This term is linear in x. Thus, its derivative with respect to x is **$f'(x_n)$**.

This is a quadratic term in x. Thus, its derivative with respect to x is **$f''(x_n)(x - x_n)$**.

# Newton's Method – Formula

$$\frac{\partial f(x)}{\partial x} \approx \frac{\partial f(x_n)}{\partial x} + \frac{\partial \left(f'(x_n)(x - x_n)\right)}{\partial x} + \frac{\partial \left(\frac{1}{2} f''(x_n)(x - x_n)^2\right)}{\partial x}$$

$$\frac{\partial f(x)}{\partial x} \approx f'(x_n) + f''(x_n)(x - x_n)$$

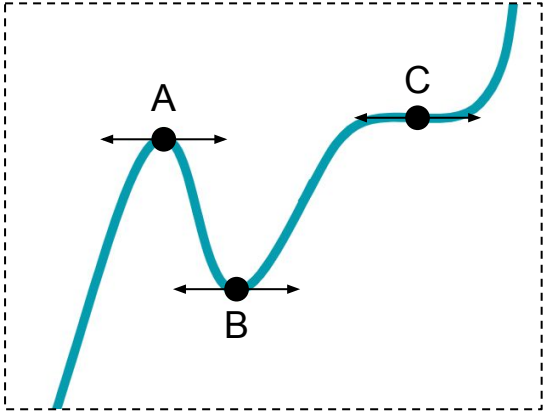
This formula gives an approximate first-order derivative of **f** at any point **x** near $\mathbf{x_n}$ while considering both the first and second-order behavior of **f** around $\mathbf{x_n}$. In optimization, we seek to find where this approximate gradient is zero as it suggests a stationary point – potential minimum, maximum or saddle point – of the actual function **f**.

$$0 \approx f'(x_n) + f''(x_n)(x - x_n)$$

$$x \approx x_n - \frac{f'(x_n)}{f''(x_n)}$$

# Newton's Method – Formula

| | A | B | C |
|---|---|---|---|
| $f'(x_n)$ | ≈ 0 | ≈ 0 | ≈ 0 |
| $f''(x_n)$ | < 0 | > 0 | ≈ 0 |
| $\frac{f'(x_n)}{f''(x_n)}$ | ≈ 0 | ≈ 0 | < 0<br>*or* ≈ 0<br>*or* > 0 |




$$x \approx x_n - \frac{f'(x_n)}{f''(x_n)}$$

# Newton's Method – Formula for a Multivariate Function

$$f(\mathbf{x}) \approx f(\mathbf{x}_n) + \nabla f(\mathbf{x}_n)^T(\mathbf{x} - \mathbf{x}_n) + \frac{1}{2}(\mathbf{x} - \mathbf{x}_n)^T H(f)(\mathbf{x}_n)(\mathbf{x} - \mathbf{x}_n)$$

$$\begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix}$$

The **Gradient** of a function is a vector consisting of the first-order partial derivatives relative to each variable.

The **Hessian** of a multivariable function is a square matrix composed of its second-order partial derivatives.

$$\begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

*If the function or its derivatives are continuous, this matrix is **symmetrical** (Clairaut's theorem)*

# Newton's Method – Formula for a Multivariate Function

$$\nabla f(\mathbf{x}) \approx \nabla\left(f(\mathbf{x}_n)\right) + \nabla\left(\nabla f(\mathbf{x}_n)^T(\mathbf{x}-\mathbf{x}_n)\right) + \nabla\left(\frac{1}{2}(\mathbf{x}-\mathbf{x}_n)^T H(f)(\mathbf{x}_n)(\mathbf{x}-\mathbf{x}_n)\right)$$

$$\approx \nabla f(\mathbf{x}_n) + H(f)(\mathbf{x}_n)(\mathbf{x}-\mathbf{x}_n)$$

$$0 \approx \nabla f(\mathbf{x}_n) + H(f)(\mathbf{x}_n)(\mathbf{x}-\mathbf{x}_n)$$

Assuming the Hessian $H(f)(x_n)$ is **invertible**.

$$-\nabla f(\mathbf{x}_n) \approx H(f)(\mathbf{x}_n)(\mathbf{x}-\mathbf{x}_n)$$

$$-H(f)(\mathbf{x}_n)^{-1}\nabla f(\mathbf{x}_n) \approx \mathbf{x}-\mathbf{x}_n$$

$$\mathbf{x} \approx \mathbf{x_n} - H(f)(\mathbf{x}_n)^{-1}\nabla f(\mathbf{x}_n)$$

The final **iterative update** rule tells us how to move from our current estimate $X_n$ to a new estimate $X_{n+1}$ by taking a step that is informed by both the direction of the gradient and the curvature of the function.

# Damped Newton's Method

As opposed to the Gradient Descent, a learning rate is not explicitly included in the Newton's Method formula because the method itself is designed to take optimal step sizes based on the curvature of the function being optimized.

However, in practice, especially in machine learning applications, a learning rate can be introduced to the update rule to provide additional control over the step size. This makes the method more robust to situations where the Hessian may not be a perfect representation of the loss function curvature due to noise for example. This modified update rule with a learning rate would be:

$$\mathbf{x}_{n+1} \approx \mathbf{x}_n - \alpha H(f)(\mathbf{x}_n)^{-1} \cdot \bigtriangledown f(\mathbf{x_n})$$

*The learning rate is generally set to be between 0 and 1.*

# Newton's Method

**Newton's method** uses the Hessian matrix which contains second-order derivative information. This tells the method not just the direction to move towards the minimum but also how the slope is changing. This allows for more informed and potentially larger steps towards the minimum.

The Hessian helps to adjust the step size dynamically by taking into account the curvature of the function:

- Far from the optimum, if the curvature is shallow, it can take a large step.
- Close to the optimum, if the curvature is steeper, it takes smaller steps to avoid overshooting.

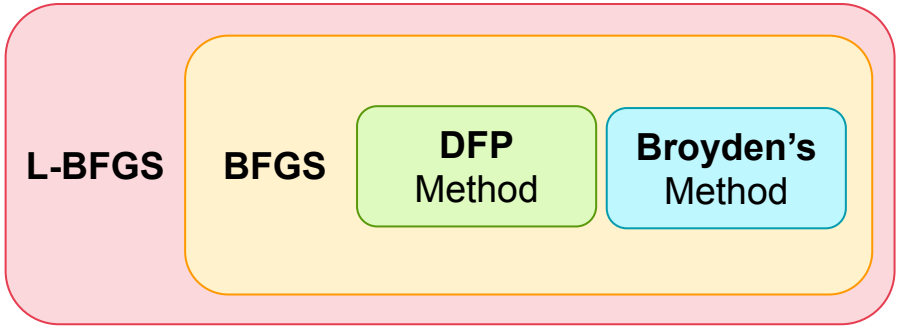
Additionally, near the solution, Newton's method converges quadratically which is much faster than the linear convergence rate of Gradient Descent.

However, the Newton's method requires to compute and invert of the Hessian matrix which are both extremely slow – between $O(n^2)$ and $O(n^3)$ – and need a lot of space to be stored.

# Quasi-Newton Method

To reduce computational costs, **Quasi-Newton methods** approximate the Hessian matrix based on gradient information from current and previous iterations to avoid the direct computation of second derivatives. Some Quasi-Newton methods are also memory-efficient by storing only vectors that represent the Hessian or its inverse which are then used to construct the approximation.

The Quasi-Newton methods encompass a variety of algorithm families. One of the most well-known families is the **Broyden** family which includes the **Broyden's method**, the **Davidon-Fletcher-Powell** (DFP) method, the **Broyden-Fletcher-Goldfarb-Shanno** (BFGS) method and the **Limited Memory BFGS** (L-BFGS) method.



# Quasi-Newton's Method and Secant Condition

Computing the Hessian matrix directly involves second derivatives which can be costly to calculate especially for high-dimensional problems. Quasi Newton methods are "looking around" to estimate the rate of change of the slope by computing the change in the gradient between two successive points rather than directly searching for the second derivative. Remember the derivative of the Taylor Expansion Series around $\mathbf{x}_n$ ?

$$\begin{aligned}\bigtriangledown f(\mathbf{x}) &\approx \bigtriangledown\left(f(\mathbf{x}_n)\right) + \bigtriangledown\left(\nabla f(\mathbf{x}_n)^T(\mathbf{x}-\mathbf{x}_n)\right) + \bigtriangledown\left(\frac{1}{2}(\mathbf{x}-\mathbf{x}_n)^T H(f)(\mathbf{x}_n)(\mathbf{x}-\mathbf{x}_n)\right) \\ &\approx \bigtriangledown f(\mathbf{x}_n) + H(f)(\mathbf{x}_n)(\mathbf{x}-\mathbf{x}_n)\end{aligned}$$

We can rearrange it to get that formula called the **quasi-Newton condition** or **secant condition**:

$$H(f)(\mathbf{x}_n)(\mathbf{x}_{n+1}-\mathbf{x}_n) \approx \bigtriangledown f(\mathbf{x}_{n+1}) - \bigtriangledown f(\mathbf{x}_n)$$

The key idea of all quasi-newton methods is to maintain an approximation of the Hessian matrix – noted **B** – that satisfies the **secant condition**:

$$B_{n+1}(\mathbf{x}_{n+1}-\mathbf{x}_n) = \bigtriangledown f(\mathbf{x}_{n+1}) - \bigtriangledown f(\mathbf{x}_n)$$

# Broyden's Method

The general form of **Broyden's update for the matrix B** is as follows. The formula is not directly derived from first principles like a theorem in mathematics but rather constructed to meet certain criteria. You can find an explanation on the update term on Broyden's original scientific article here.

The **"error"** between the actual change and the predicted change

*This represents the **actual observed change** in the gradient*

*This is the **predicted change** based on the current approximation*

The **scaling term** to transform the vector of the error into a rank-one matrix.

$$B_{n+1} = B_n + \frac{\left(\nabla f(\mathbf{x}_{n+1}) - \nabla f(\mathbf{x}_n) - B_n\left(\mathbf{x}_{n+1} - \mathbf{x}_n\right)\right)\left(\mathbf{x}_{n+1} - \mathbf{x}_n\right)^T}{\left(\mathbf{x}_{n+1} - \mathbf{x}_n\right)^T\left(\mathbf{x}_{n+1} - \mathbf{x}_n\right)}$$

The denominator **normalizes the update**, ensuring that the magnitude of the correction is scaled appropriately relative to the size of the step $\mathbf{X_{n+1}} - \mathbf{X_n}$.

# Davidon-Fletcher-Powell Method

Similarly to the Broyden's method, **DFP** maintains and updates an approximation of the Hessian matrix and ensures that it satisfies the secant condition.

However, DFP explicitly ensures that the updated Hessian approximation remains positive definite through its update formula whereas Broyden's method does not. Why is that so important ? Because it ensures that the search direction determined by the Hessian approximation is a descent direction which ensures that the algorithm makes consistent progress towards finding a local minimum.

$$B_{n+1} = \left(I - \frac{y_n s_n^T}{y_n^T s_n}\right) B_n \left(I - \frac{s_n y_n^T}{y_n^T s_n}\right) + \frac{y_n y_n^T}{y_n^T s_n}$$

$$y_n = \nabla f(x_{n+1}) - \nabla f(x_n)$$

$$s_n = x_{n+1} - x_n$$

# Davidon-Fletcher-Powell Method

$$B_{n+1} = \left( I - \frac{y_n s_n^T}{y_n^T s_n} \right) B_n \left( I - \frac{s_n y_n^T}{y_n^T s_n} \right) + \frac{y_n y_n^T}{y_n^T s_n}$$

The **outer terms** adjust $\mathbf{B_n}$ by reducing the influence of the previous approximation along the direction of the step $\mathbf{s_n}$.

This term directly adds new information about the curvature of the function as indicated by the change in gradients $\mathbf{y_n}$. It emphasizes the directions in which significant curvature changes were observed.

# Davidon-Fletcher-Powell Method

The DFP method can also be formulated in terms of directly updating the inverse of the Hessian approximation which is more computationally efficient than updating the Hessian approximation itself and then inverting it.

*Let's assume that $H_n$ is the current inverse Hessian approximation at iteration n.*

$$B_n^{-1} = H_n$$

$$H_{n+1} = H_n + \frac{s_n s_n^T}{y_n^T s_n} - \frac{H_n y_n y_n^T H_n}{y_n^T H_n y_n}$$

*A rank-one update that adds information based on the step taken.*

*This term subtracts a portion of the current approximation that may no longer be accurate given the new information.*

# BFGS

The **BFGS** method builds upon the concepts introduced by Broyden's method and the DFP method, refining and combining their approaches to update the Hessian approximation. The BFGS update formula for the inverse Hessian approximation $H_{n+1}$ is:

$$H_{n+1} = \left(I - \frac{s_n y_n^T}{y_n^T s_n}\right) H_n \left(I - \frac{y_n s_n^T}{y_n^T s_n}\right) + \frac{s_n s_n^T}{y_n^T s_n}$$


Take an initial guess for the weights $W_0$ and the inverse Hessian approximation $H_0$ such as the identity matrix.

Determine the step size **α** to be used when updating the model parameters using a backtracking line search algorithm.

Compute the search direction:

$$p_n = -H_n \bigtriangledown L(W_n)$$

Then, update the weights:

$$W_{n+1} = W_n + \alpha_n p_n$$

# BFGS – Wolfe Conditions

The line search in optimization algorithms like BFGS is a step that determines the **step size α** to be used when updating the model parameters. The goal of the line search is to find a step size that sufficiently reduces the loss function, ensuring that each step moves towards the minimum in an efficient and stable manner. Indeed, a fixed step size might be too small in flat areas, leading to slow progress or too large in steep areas, causing overshooting.

Conversely, an adaptive step size selection ensures that each step is appropriately sized for the current landscape. However, this adaptive step size must satisfy the **Wolfe conditions** for the BFGS algorithm to avoid overshooting, ensure stability, prevent slow convergence and make consistent progress toward the optimum.

**Armijo Condition**

*It prevents BFGS from taking too big a step that might overshoot the minimum or increase the function value. This ensures that each step actually moves you closer to the goal.*

+

**Curvature Condition**

*It prevents from taking too small a step. It helps to make sure BFGS is efficiently moving, not just inching forward when it could stride.*

# BFGS – Wolfe Condition

## **Armijo** Condition

*The function value after taking a step* $\alpha$ *in the direction p.*

$$L(W + \alpha p) \leq L(W) + c_1 \alpha \bigtriangledown L(W)^T p$$

*The current function value plus a small fraction of the initial predicted decrease in the function value.*

$c_1$ *is a constant that controls how much decrease in the function value is sufficient.*

$0 < c_1 < 1$

## **Curvature** Condition

*The gradient at the new point, projected onto the direction of the step.*

$$\bigtriangledown L(W + \alpha p)^T p \geq c_2 \bigtriangledown L(W)^T p$$

*A fraction* $c_2$ *of the initial gradient projected onto the same direction.*

$c_2$ *determines the required increase in the gradient's component along the direction of p.*

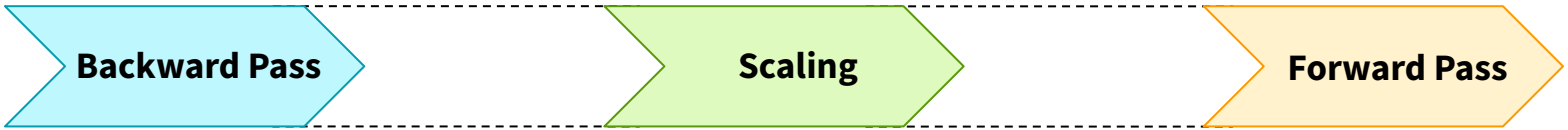
$0.9 < c_2 < 1$

# L-BFGS

**Limited-memory** Broyden-Fletcher-Goldfarb-Shanno is an extension of the BFGS algorithm designed to overcome memory limitations when dealing with high-dimensional optimization problems. The key change in L-BFGS lies in its approach to approximating the inverse of the Hessian matrix.

Indeed, instead of maintaining the full Hessian approximation, a limited history of the past vectors $\mathbf{s}_n$ – the changes in the parameters – and $\mathbf{y}_n$ – the changes in the gradients is used. These vectors are used to implicitly form the Hessian approximation without actually constructing the matrix.

The two-loop recursion algorithm can be used to approximate the inverse of the Hessian matrix using $\mathbf{s}_n$ and $\mathbf{y}_n$ :



# L-BFGS – Two-Recursion Algorithm

**Backward Pass** → **Scaling** → **Forward Pass**

*The **backward pass** accumulates information from the most recent iterations to capture the curvature of the objective function. It adjusts the **gradient** vector by applying a series of corrections based on past gradients and parameter updates.*

*For each iteration i going **backwards** from the most recent to the oldest up to m iterations back:*

$$\alpha_i = \frac{s_i^T q}{y_i^T s_i}$$

$$q = q - \alpha_i y_i$$

*The **scaling step** adjusts the magnitude of the vector resulting from the backward pass, aiming to better approximate the scale of the true inverse Hessian.*

*The scaled vector serves as an initial approximation of the Hessian's effect on the gradient.*

$$\gamma_n = \frac{s_{n-1}^T y_{n-1}}{y_{n-1}^T y_{n-1}}$$

$$r = -\gamma_n q$$

*The **forward pass** refines the vector obtained from the scaling step by sequentially applying information from the stored $\mathbf{s_n}$ and $\mathbf{y_n}$ vectors, but this time moving forward from the oldest to the most recent.*

*For each iteration i going forward, the algorithm computes:*

$$\beta = \frac{y_i^T r}{y_i^T s_i}$$

$$r = r + s_i(\alpha_i - \beta)$$