Gradient Based Optimization II

Asst. Prof. Dr.-Ing. Sudchai Boonto

Department of Control System and Instrumentation Engineering
King Mongkut's University of Technology Thonburi
Thailand

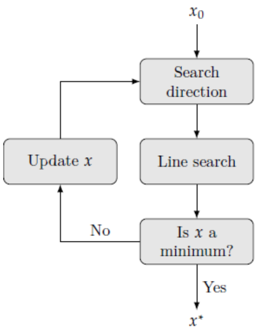
September 3, 2025

Objective

At the end of this chapter you should be able to:

- Mathematically define the optimality conditions for an unconstrained problem.
- Describe, implement, and use line-search-based methods.
- Gradient Descent based method

Two Approaches to Finding and Optimum



Line search approach

- Exact Line Search
- Approximate Line Search

Line Search : Exact Line Search

Assume we have chosen a descent direction $\mathbf{d}$. We need to choose the step factor α to obtain our next design point. One approach is to use *line search*, which selects the step factor that minimizes the one-dimensional function:

$$\underset{\alpha}{\text{minimize}} \quad f(\mathbf{x} + \alpha \mathbf{d})$$

To inform the search, we can use the derivative of the line search objective, which is simply the directional derivative along $\mathbf{d}$ at $\mathbf{x} + \alpha\mathbf{d}$.

```
function LINE_SEARCH($f, \mathbf{d}$)
    objective = $\alpha \to f(\mathbf{x} + \alpha * \mathbf{d})$
    $a, b$ = brackect_minimum(objective)
    $\alpha$ = minimize(objective, a, b)
    return $\mathbf{x} + \alpha * \mathbf{d}$
end function
```

The exact line search is expensive, if we need to do it every step of the optimization. In Matlab environment, we can use commands `fminbnd` or `fminsearch`.

One-Dimensional Unconstrained Optimization

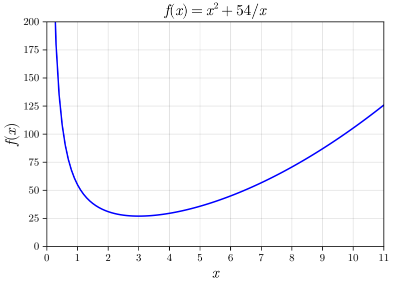
In general, exact line search in unconstrained optimization can be performed using one-dimensional techniques. Common algorithms include:

Algorithm	Requires Unimodality	Note
Fibonacci Search	✓ yes	Efficient narrowing using Fibonacci ratios
Golden Section Search	✓ yes	Uses golden ratio for interval reduction
Quadratic Fit Search	✓ yes	Fits a parabola to estimate minimum
Shubert-Piyavskii method	✗ no	Handles non-unimodal functons
Bisection Method	✗ no	Simple, robust, and effective.

Since we consider both linear and nonlinear optimization problems, we consider only the bisection method due to its simplicity and effectiveness–even when the function is not unimodal.

Unimodality

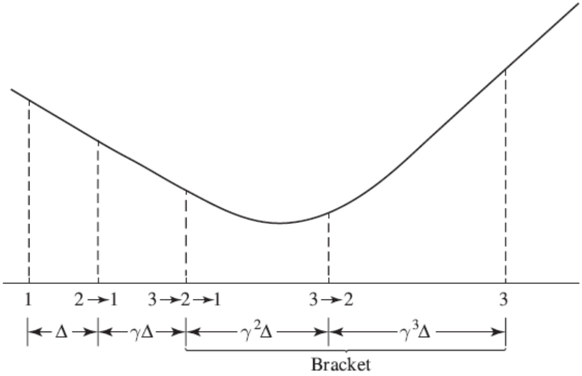
- Several of the algorithms assume unimodality of the objective function.
- A unimodal function f is one where there is a **unique** x^*, such that f is monotonically decreasing for $x \leq x^*$ and monotonically increasing for $x \geq x^*$.
- It follows from this definition that the unique global minimum is at x^* , and there are no other local minima.
- Given a unimodal function, we can bracket and inter $[a, c]$ containing the global minimum if we can find three points $a < b < c$, such that $f(a) > f(b) < f(c)$.



Finding an Initial Bracket

- When optimizing a function, we often start by first bracketing and interval containing a local minimum.
- After that, we then successively reduce the size of the bracketed interval to converge on the local minimum.
- We choose a starting point 1 with coordinate x_1 and a step size Δ in the positive direction. The distance we take is a *hyperparameter* to this algorithm. The step size Δ is 1×10^{-2}.
- We then search in the downhill direction to find a new point that exceeds the lowest point. With each step, we axpand the step size by some factor, which is another hyperparameter to the to this algorithm that is often set to $\gamma = 2$.
- We need to bracket the minimum before using bisection because the bisection method isn't a minimization algorithm–it's a root-finding algorithm. It's only guaranteed to work if we are already trapped at a root within an interval.

Finding the Initial Bracket cont.



Finding the Initial Bracket cont.

Bracketing Algorithm / Three-Point Pattern

```
1. Set $x_2 = x_1 + \Delta$
2. Evaluate $f_1$ and $f_2$
3. If $f_2 \le f_1$ Goto Step 5
4.     Else Interchange $f_1$ and $f_2$ and $x_1$ and $x_2$, and Set $\Delta = -\Delta$
5. Set $\Delta = \gamma\Delta$, $x_3 = x_2 + \Delta$, and Evaluate $f_3$ at $x_3$
6. If $f_3 > f_2$ Goto Step 8
7.     Else Rename $f_2$ as $f_1$, $f_3$ as $f_2$, $x_2$ as $x_1$, $x_3$ as $x_2$, Goto Step 5
8. Point $1, 2,$ and $3$ satisfy $f_1 \ge f_2 < f_3$ (three-point pattern)
```

Finding an Initial Bracket: Julia

Simple Julia code:

```julia
function bracket_minimum(f, x=0, s=1e-2, k = 2.0)
# f is anonymous function, x is a starting point, s is step size, k is weight
      mulitplier
# Ensure we start by going downhill
a, ya = x, f(x)
b, yb = a + s, f(a + s)

if yb > ya          # swap the direction
    a, b = b, a
    ya, yb = yb, ya
    s = -s
end
while true
    c, yc = b + s, f(b + s)
    if yc > yb
        return a < c ? (a, c) : (c,a)
    end
        a, ya , b, yb = b, yb, c, yc
        s *= k
    end
end
```

Finding an Initial Bracket: Matlab

Simple Matlab code:

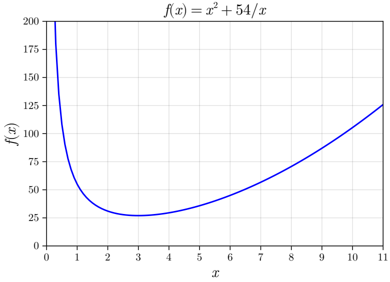
```matlab
function [a, c] = bracket_minimum(f, x, s, k)
% f: anonymous function, x: starting point, s: step size
% k: gamma (weight multiplier)

a = x; ya = f(x); b = a + s; yb = f(a + s);
% change the direction
if yb > ya
      [a, b] = swap(a,b); [ya, yb] = swap(ya, yb); s = -s;
end
flag = 1;   % yb < ya
while flag
    s = s * k; c = b + s; yc = f(b + s);
    if yc < yb
        a = b; ya = yb; b = c; yb = yc;
    else
        if c < a
            [a, c] = swap(a,c); % deal(c,a)
        end
        flag = 0;
    end
  end
end
```

Finding the Initial Bracket cont.: Example

Consider the problem:

$$\underset{x}{\text{minimize}} \qquad x^2 + \frac{54}{x}$$

in the interval $(0, 5)$.



Using a algorithm above we have the interval $(1.28, 5.12)$ by using $\Delta = 1e-2, \gamma = 2$. The interval guarantees that the minimum point lies in the interval.

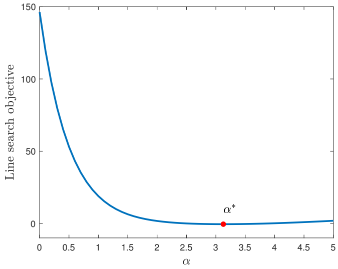
Line Search : Exact Line Search

Consider conducting a line search on $f(x_1, x_2, x_3) = \sin(x_1 x_2) + e^{(x_2+x_3)} - x_3$ from $\mathbf{x} = [1, 2, 3]$ in the direction $\mathbf{d} = [0, -1, -1]$. The corresponding optimization problem is:

$$\begin{aligned} \underset{\alpha}{\text{minimize}} \quad & \sin((1+0\alpha)(2-\alpha)) \\ & + e^{((2-\alpha)+(3-\alpha))} - (3-\alpha) \end{aligned}$$

which simplifies to:

$$\underset{\alpha}{\text{minimize}} \quad \sin(2-\alpha) + e^{(5-2\alpha)} + \alpha - 3$$



The minimum is at $\alpha \approx 3.127$ with $\mathbf{x} \approx [1, -1.126, -1.126]$.
Note I: $\nabla f(\alpha) = -\cos(2-\alpha) - 2e^{(5-2\alpha)} + 1 = 0$. We can solve for α by using `vpasolve` in `Matlab`.
Note II: We can use Nonlinear search in Matlab or Julia like **fminbnd** from the original problem.

Line Search : Exact Line Search

Nonlinear optimization using (Matlab) `fimnbnd`:

```
f = @(alpha) sin(2-alpha) + exp(5-2*alpha) + alpha - 3\\
alpha1 = fminbnd(f, 0, 4, opts)
```

Root finding of gradient (Matlab) `vpasolve`:

```
syms alpha

f1 = -cos(2-alpha) - 2*exp(5-2*alpha) + 1;
alpha1 = vpasolve(f1==0, alpha)
```

Line Search : Exact Line Search

Nonlinear optimization using (Julia) `Optim`

```julia
using Optim

f = α  -> -sin(2 - α) +  exp(5 - 2 * α) + α - 3

res = Optim.optimize(f, 0, 4)   # minimize
sol = Optim.minimizer(res)
```

Root finding of gradient (Julia) `nlsolve`

```julia
using NLsolve

f = α  -> -cos(2 - α) - 2 * exp(5 - 2 * α) + 1

sol = nlsolve(x -> [f(x[1])], [3.5])  # Initial guess between 3.5
sol.zero
```

We should have the same result around 3.127.

Bisection Method

The **bisection method** can be used to find roots of the function, or points where the function is zero. The **root-finding methods** can be used for optimization by applying them to the derivative of the objective, locating where $\nabla f(\mathbf{x}) = 0$. We must ensure that the resulting points are indeed local minima. In this method:

- The bisection method cuts the bracketed region in half with every iteration.
- The midpoint $(a+b)/2$ is evaluated, and the new bracket is formed from the midpoint and whichever side that continues to bracket a zero.
- We terminate immediately if the midpoint evaluates to zero. Otherwise we can terminate after a fixed number of iterations.
- The method is guaranteed to converge within ϵ of x^* within $\log_2\left(\frac{|b-a|}{\epsilon}\right)$ iterations, where $\log_2$ denotes the base 2 logarithm. This tells us how many iterations are need to reach a solution within tolerance ϵ.
- In optimization, we often seek points where the derivative of the objective function is zero–that is, where $\nabla f(\mathbf{x}) = 0$. These points may be minimum, maximum, or saddle point.

Bisection Method

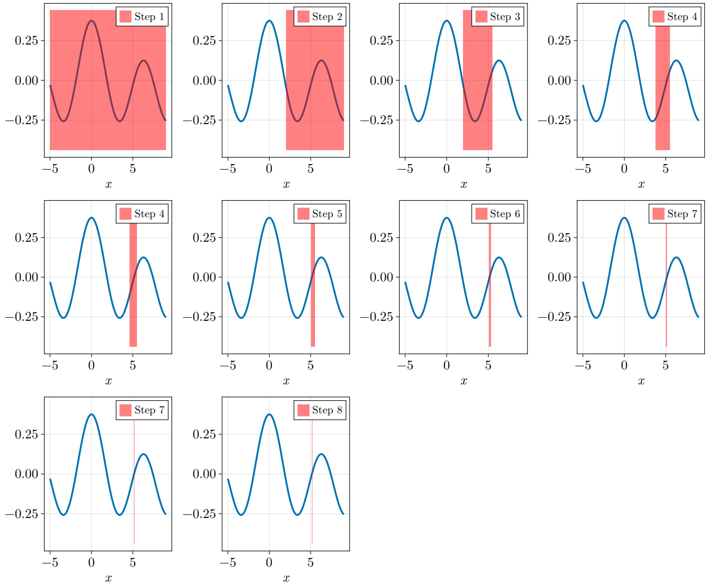
Bisection Method

```
1.  If $a > b$ Then $a, b = b, a$ EndIf
2.  $ya, yb = f(a), f(b)$
3.  If $ya == 0$ Then $b = a$ EndIf
4.  If $yb == 0$ Then $a = b$ EndIf
5.  While $b - a > \epsilon$
6.      $x = (a + b)/2$
7.      $y = f(x)$
8.      If $y == 0$
9.          $a, b = x, x$
10.     ElseIf sign($y$) == sign($ya$)
11.         $a = x$
12.     Else
13.         $b = x$
14.     EndIf
15. EndWhile
16. Return $(a, b)$
```

Example: $f(x) = \frac{1}{4}(\sin(x) + \sin(\frac{x}{2})$, and $\nabla f(x) = \frac{1}{4}(\cos(x) + \frac{1}{2}\cos(\frac{x}{2}))$
In this example, we use an approximate derivative using

$$\nabla f(x) = \frac{f(a+h) - f(a-h)}{2h}, \quad h = 1 \times 10^{-3}$$

Bisection Method: Example



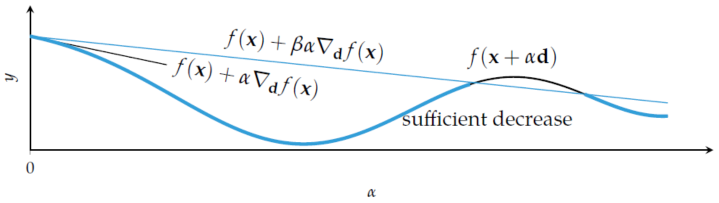
Approximate Line Search: Armijo Rule

- It is often more computationally efficient to perform more iterations of a descent method than to do an exact line search at each iteration, especially if the function and derivative calculations are expensive.
- Many methods discussed so far can benefit from using **approximate line search** to find a suitable step size with a small number of evaluations.
- Since descent methods must descend, a step size α may be suitable if it causes a decrease in the objective function value. We need the *sufficient decrease* condition. (to protect that the reductions in f values is not to small.)
- The sufficient decrease in the objective function value:

$$f(\mathbf{x}_{k+1}) \leq f(\mathbf{x}_k) + \beta \alpha \nabla_{\mathbf{d}_k} f(\mathbf{x}_k)$$

  with $\beta \in [0, 1]$ often set to $\beta = 1 \times 10^{-4}$.
- This inequality states that the function value at the new point must lie at or below the line defined by the initial slope, scaled by β. It prevents steps that are too logn.

Approximate Line Search


$f(\mathbf{x}) + \beta\alpha\nabla_{\mathbf{d}} f(\mathbf{x})$
$f(\mathbf{x}) + \alpha\nabla_{\mathbf{d}} f(\mathbf{x})$
$f(\mathbf{x} + \alpha\mathbf{d})$
sufficient decrease
y
0
α

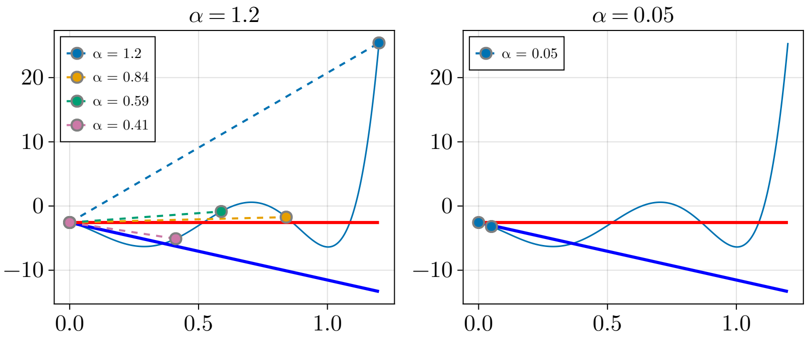
- If $\beta = 0$, then any decrease is acceptable. If $\beta = 1$, then the decrease has to be at least as much as what would be predicted by a first-order approximation.
- If $\mathbf{d}$ is a valid descent direction, then there must exist a sufficiently small step size that satisfies the sufficient decrease condition.
- We can start with a large step size and decrease it by a constant reduction factor until the sufficient decrease condition is satisfied.
- The algorithm is known as *backtracking line search* because of how it backtracks along the descent direction.

Approximate Line Search

```
function BACKTRACKING_LINE_SEARCH($f, \nabla f, \mathbf{x}, \mathbf{d}, \alpha; \rho = 0.5, \beta = 1 \times 4$)
    while $f(\mathbf{x} + \alpha * \mathbf{d}) > f(\mathbf{x}) + \beta\alpha * \nabla f(\mathbf{x})^T \mathbf{d}$ do
        $\alpha = \rho\alpha$
    end while
    return $\alpha$
end function
```

- The first condition is insufficient to guarantee convergence to a local minimum. Very small step sizes will satisfy the first condition but can prematurely converge.
- Backtracking line search avoids premature convergence by accepting the largest satisfactory step size obtained by sequential downscaling and is guaranteed to converge to ta local minimum.

Approximate Line Search



Objective function is

$$f(\mathbf{x}) = 0.1x_1^6 - 1.5x_1^4 + 5x_1^2 + 0.1x_2^4 + 3x_2^2 - 9x_2 + 0.5x_1x_2$$

Initial $\mathbf{x} = [-1.25,\ 1.25]$, $\mathbf{d} = [4,\ 0.75]$, and $\beta = 1 \times 10^{-4}$, α are 1.2 and 0.05, respectively.

Curvature Condition

The *curvature condition* requires the the directional derivative at the next iterate to be shallower (α is not too close small.):

$$\nabla_{\mathbf{d}_k} f(\mathbf{x}_{k+1}) \geq \sigma \nabla_{\mathbf{d}_k} f(\mathbf{x}_k)$$

- Where σ controls how shallow the next directional derivative must be. The α is go farer than the previous slope.
- It is common to set σ larger than β with $\sigma = 0.1$ when approximate linear search is used with the conjugate gradient method and to 0.9 when used with Newton's method.
- The *strong curvature condition*, which is more restrictive criterion in that is also required not to be too positive:

$$|\nabla_{\mathbf{d}_k} f(\mathbf{x}_{k+1})| \leq \sigma |\nabla_{\mathbf{d}_k} f(\mathbf{x}_k)|$$

- Both sufficient decrease condition (for α_U) and strong curvature condition are called **strong Wolfe conditions.**(for α_L).

Wolfe Condition

Consider approximate line search on $f(x_1, x_2) = x_1^2 + x_1x_2 + x_2^2$ from $\mathbf{x} = [1, 2]$ in the direction $\mathbf{d} = [-1, -1]$, using a maximum step size of 10, a reduction factor of 0.5, a first Wolfe condition parameter $\beta = 1 \times 10^{-4}$ and a second Wolfe condition parameter $\sigma = 0.9$.
The first Wolfe condition is $f(\mathbf{x} + \alpha\mathbf{d}) \leq f(x) + \beta\alpha(\mathbf{g}^T\mathbf{d})$, where $\mathbf{g} = \nabla f(\mathbf{x}) = [4, 5]$.
$\alpha = 10$ we have

$$f\left(\begin{bmatrix}1\\2\end{bmatrix} + 10\begin{bmatrix}-1\\-1\end{bmatrix}\right) \leq 7 + 1 \times 10^{-4}(10)\left(\begin{bmatrix}4 & 5\end{bmatrix}\begin{bmatrix}-1\\-1\end{bmatrix}\right)$$

$217 \leq 6.991$ (It is not satisfied.)

$\alpha = 0.5(10) = 5$ we have

$$f\left(\begin{bmatrix}1\\2\end{bmatrix} + 5\begin{bmatrix}-1\\-1\end{bmatrix}\right) \leq 7 + 1 \times 10^{-4}(5)\begin{bmatrix}4 & 5\end{bmatrix}\begin{bmatrix}-1\\-1\end{bmatrix}$$

$37 \leq 6.996$ (It is not satisfied.)

Wolfe Condition

$\alpha = 0.5(5) = 2.5$, we have

$$f\left(\begin{bmatrix}1\\2\end{bmatrix} + 2.5\begin{bmatrix}-1\\-1\end{bmatrix}\right) \leq 7 + 1\times 10^{-4}(2.5)\begin{bmatrix}4 & 5\end{bmatrix}^T\begin{bmatrix}-1\\-1\end{bmatrix}$$

$$3.25 \leq 6.998(\text{ The first Wolfe condition is satisfied.})$$

The candidate design point $\mathbf{x}' = \mathbf{x} + \alpha\mathbf{d} = [-1.5, -0.5]^T$ is checked against the second Wolfe condition:

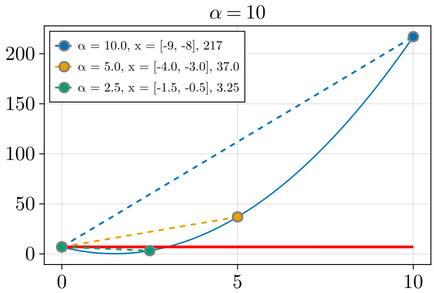
$$\nabla_{\mathbf{d}} f(\mathbf{x}') \geq \sigma \nabla_{\mathbf{d}} f(\mathbf{x})$$

$$\begin{bmatrix}-3.5 & -2.5\end{bmatrix}\begin{bmatrix}-1\\-1\end{bmatrix} \geq \sigma\begin{bmatrix}4 & 5\end{bmatrix}\begin{bmatrix}-1\\-1\end{bmatrix}$$

$$6 \geq -8.1(\text{ The second Wolfe condition is satisfied. })$$

Approximate line search terminates with $\mathbf{x} = [-1.5 \ -0.5]^T$.

Wolfe Condition



```
function BACKTRACKING_LINE_SEARCH($f, \nabla f, \mathbf{x}, \mathbf{d}, \alpha; \rho = 0.5, \beta = 1 \times 4, \sigma = 0.9$)
    $y, \mathbf{g} = f(\mathbf{x} + \alpha\mathbf{d}), \nabla f(\mathbf{x})^T\mathbf{d}$
    while ($y > f(\mathbf{x}) + \beta\alpha * \nabla f(\mathbf{x})^T\mathbf{d}$) and ($\mathbf{g} \leq \sigma\mathbf{g}$) do
        $\alpha = \rho\alpha$
    end while
    return $\alpha$
end function
```

Momentum method

- The Momentum method is an optimization is an optimization algorithm that helps accelerate the training of machine learning model, particularly deep neural networks. It is an extension of the Gradient Descent algorithm and is designed to overcome some of its limitations, such as slow convergence and the tendency to get stuck in local minima.
- The core idea is to indtoduce a "velocity" term that accumulates an exponentially decaying moving average of past gradients. This velocity is then added to the current gradient to compute the update step. Thid process helps to smooth out the updates and propel the optimizer in more consistant direction, luch like a ball rolling down a hill gathers momentum.
- The momentum update equations are:

$$\mathbf{v}_{k+1} = \beta \mathbf{v}_k - \alpha \nabla f(\mathbf{x}_k)$$
$$\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{v}_{k+1}$$

- Instead of using pure negative gradient, we add a velocity term to the update rule. If $\beta = 0$, it is a steepest descent.

Momentum method

The Momentum method can be understood through the analogy of a ball rolling down-hill.

- **Without Momentum**: Imagine a ball being placed on a hillside and only moving in the direction of the steepest descent at that exact point. If the terrain is uneven with many small bumps a valleys, the ball's path will be erratic and slow as it reacts to every little change in the slope.
- **With Momentum:** Giving a ball some mass and letting it roll. As it moves, it builds up momentum. This momentum helps it to smooth out its path, roll over small bumps (local minima), and accelerate faster down the general slope of the hill.

In term of Optimization, the "ball" is the set of parameters, and the "hill" is the objective function landscape. The momentum term helps the optimizer to:

- **Accelerate convergence:** By accumulating past gradients, the optimizer can move more quickly in directions of persistent descent.
- **Dampen oscillations:** In situations where the gradient changes direction frequently (zig-zags), the momentum term helps to average out these changes, leading to a more stable and direct path towards the minimum.

Momentum method

- **Escaping Local Minima:** The momentum term can help the optimizer overcome small local minima.

```
function GRADIENT_WITH_MOMENTUM($f, \nabla f, x_0, \alpha, \beta, N$)
    $i = 1$
    while i < N) do
        $\mathbf{g} = \nabla f(\mathbf{x})$
        $\mathbf{v} = \beta\mathbf{v} - \alpha\mathbf{g}$
        $\mathbf{x} = \mathbf{x} + \mathbf{v}$
    end while
    return $\mathbf{x}$
end function
```

Momentum method: Nesterov Momentum

- One issue of momentum is that the steps do not slow down enough at the bottom of a valley and tend to overshoot the valley floor.
- **Nesterov momentum** modifies the momentum algorithm to use the gradient at the projected future position:

$$\begin{aligned}\mathbf{v}_{k+1} &= \beta\mathbf{v}_k - \alpha\nabla f(\mathbf{x}_k + \beta\mathbf{v}_k)\\ \mathbf{x}_{k+1} &= \mathbf{x}_k + \mathbf{v}_{k+1}\end{aligned}$$

- The only difference from the classic momentum method is that the gradient is now evaluated at the point $\mathbf{x}_k + \beta\mathbf{v}_k$ instead of $\mathbf{x}_k$. This is called a **look ahead**.
- The Nesterov momentum or Nesterov's accelerated gradient (NAG) has become an industry standard algorithm in machine learning.

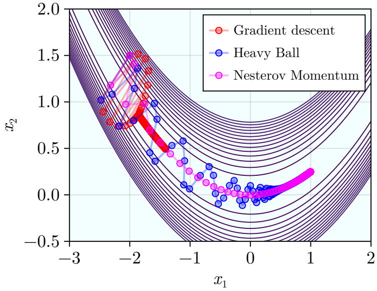
Momentum method: Nesterov Momentum

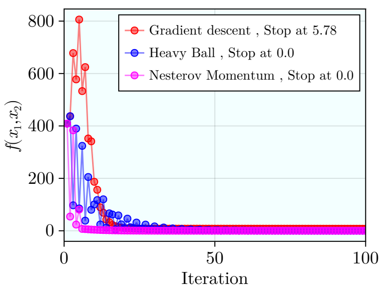
```
function GRADIENT_WITH_NESTEROV($f, \nabla f, x_0, \alpha, \beta, N$)
    $i = 1$
    while i < N) do
        $\mathbf{g} = \nabla f(\mathbf{x})$
        $\mathbf{v} = \beta\mathbf{v} - \alpha\nabla f(\mathbf{x} + \beta\mathbf{x})$
        $\mathbf{x} = \mathbf{x} + \mathbf{v}$
    end while
    return $\mathbf{x}$
end function
```

Momentum method

Consider the Rosenbrock function with $b = 100$

$$f(x_1, x_2) = (a - x_1)^2 + b(4x_2 - x_1^2)^2$$

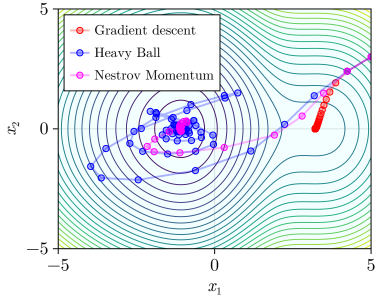


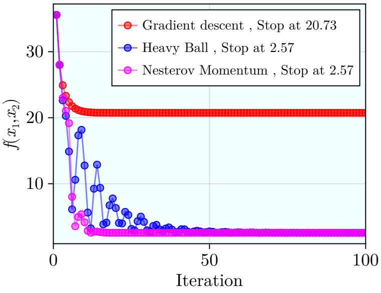


Momentum method

Consider a function, which contains a saddle point:

$$f(x_1, x_2) = 15e^{-0.25(x_1-2)^2} + 0.5e^{-2(x-2)^2} + x_1^2 + x_2^2$$





The Steepest Descent with fixed-step size get stuck at the local minima.

Momentum method: AdaGrad and RMSProp

- The momentum and Nesterov momentum update all components of $\mathbf{x}$ with the same learning rate. The **Adaptive subgradient** method or **Adagrad** adapts a learning rate for each component of $\mathbf{x}$.
- We use the sum of past gradients squared

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

The $(\nabla f(\mathbf{x}_k))^2$ is an elementwise square.

$$(\nabla f(\mathbf{x}_k))^2 = \left[\left(\frac{\partial f}{\partial x_1}\right)^2 \ \left(\frac{\partial f}{\partial x_2}\right)^2 \ \cdots \ \left(\frac{\partial f}{\partial x_n}\right)^2 \right]$$

resulting in the following iteration

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \frac{\alpha}{\sqrt{\mathbf{v}_k} + \epsilon} \nabla f(\mathbf{x}_k)$$

the small factor ϵ is added to the denomination to avoid division by zero.

Momentum method: AdaGrad and RMSProp

- In the algorithm, when we divide by $\sqrt{\mathbf{v}}$, we are actually multiplying by a vector of elementwise inverses:

$$\mathbf{w} = \left[\frac{1}{\sqrt{v_1}} \ \frac{1}{\sqrt{v_2}} \ \cdots \ \frac{1}{\sqrt{v_n}} \right]$$

- The running sum $\mathbf{v}_k$ has no discount, and so it only increases, causing the iterations to slow over time. RMSProp addresses the issue by adding a decay factor to $\mathbf{v}_k$, similar to other momentum methods:

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

Momentum method: AdaGrad and RMSProp

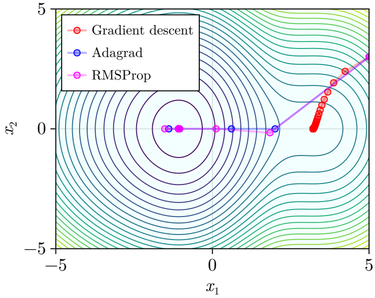
```
function GRADIENT_WITH_ADAGRAD($f, \nabla f, x_0, \alpha, \beta, N$)
    $i = 1$
    while i < N) do
        $\mathbf{g} = \nabla f(\mathbf{x})$, $\mathbf{v} = \beta \mathbf{v} + \mathbf{g} * \mathbf{g}$        element wise multiplication
        $\mathbf{x} = \mathbf{x} - \alpha \mathbf{g} / (\sqrt{v} + \epsilon)$        element wise divide
    end while
    return $\mathbf{x}$
end function
```

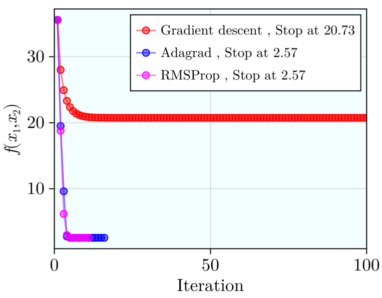
```
function GRADIENT_WITH_RMSPROP($f, \nabla f, x_0, \beta, \alpha, N$)
    $i = 1$
    while i < N) do
        $\mathbf{g} = \nabla f(\mathbf{x})$, $\mathbf{v} = \beta \mathbf{v} + (1 - \beta) \mathbf{g} * \mathbf{g}$        element wise multiplication
        $\mathbf{x} = \mathbf{x} - \alpha \mathbf{g} / (\sqrt{v} + \epsilon)$        element wise divide
    end while
    return $\mathbf{x}$
end function
```

Momentum method: AdaGrad and RMSProp

Consider a function, which contains a saddle point:

$$f(x_1, x_2) = 15e^{-0.25(x_1-2)^2} + 0.5e^{-2(x-2)^2} + x_1^2 + x_2^2$$





Reference

1. Joaquim R. R. A. Martins, and Andrew Ning, "*Engineering Design Optimization*," Cambridge University Press, 2021
2. Mykel J. Kochenderfer, and Tim A. Wheeler, "*Algorithms for Optimization*," The MIT Press, 2019