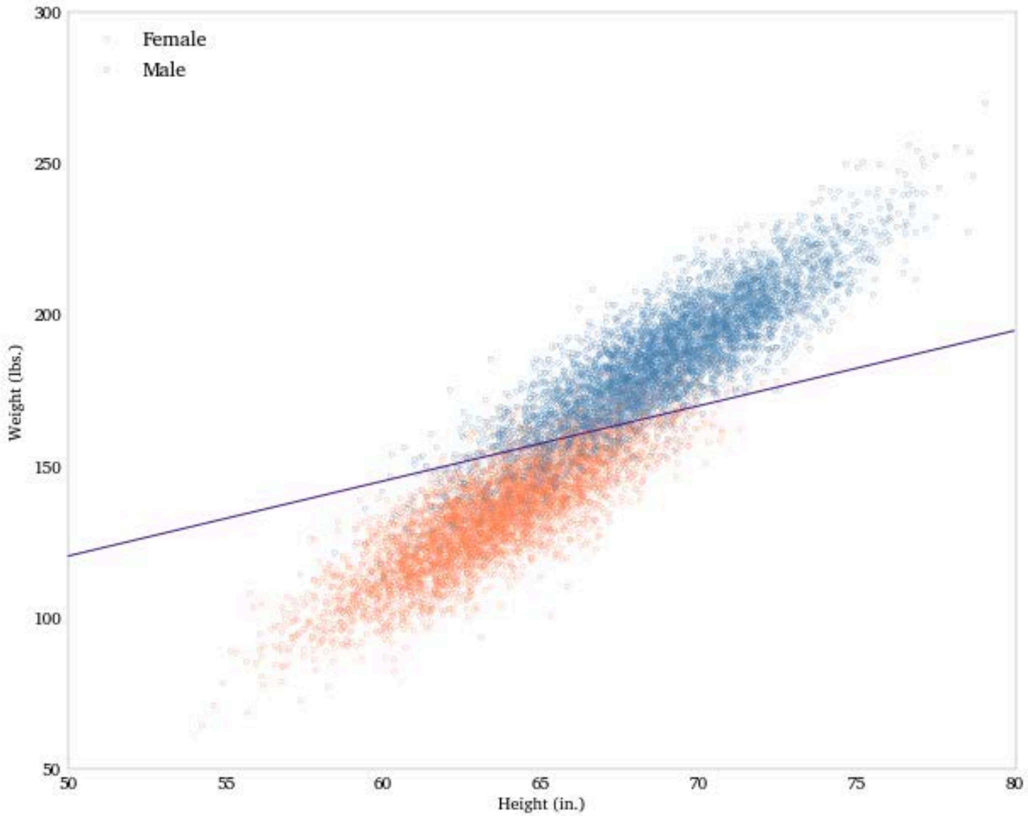
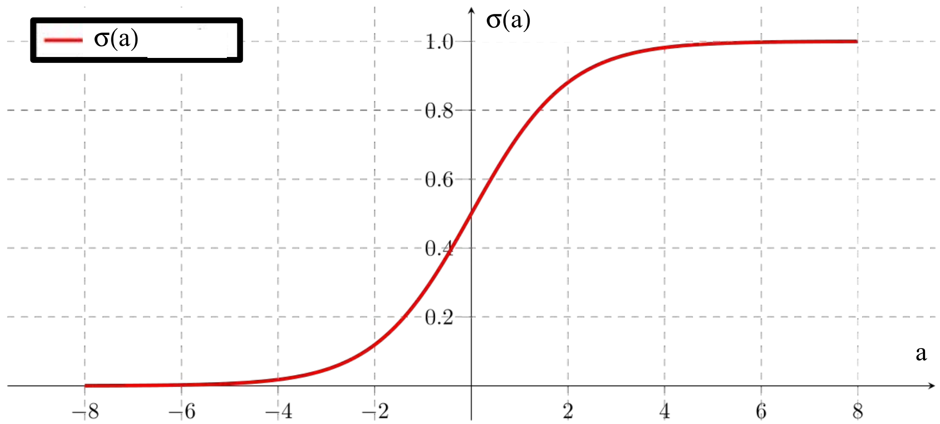
Minimizing Functions of Multiple Variables

Pascal Fua
IC-CVLab

Reminder: Logistic Regression



$$y(\mathbf{x}; \tilde{\mathbf{w}}) = \sigma(\tilde{\mathbf{w}} \cdot \tilde{\mathbf{x}})$$
$$= \frac{1}{1 + \exp(-\tilde{\mathbf{w}} \cdot \tilde{\mathbf{x}})}$$



Given a **training** set $\{(\mathbf{x}_n, t_n)_{1 \le n \le N}\}$ minimize

$$E(\tilde{\mathbf{w}}) = -\sum_n (t_n \ln y(\mathbf{x}_n) + (1 - t_n) \ln(1 - y(\mathbf{x}_n)))$$

with respect to $\tilde{\mathbf{w}}$.

—> Convex optimization problem.

Reminder: Maximizing the Margin

$$\mathbf{w}^* = min_{(\mathbf{w},\{\xi_\mathbf{n}\})}\frac{1}{2}||\mathbf{w}^2|| + C\sum_{n=1}^{N}\xi_n,$$

$$\text{subject to } \forall n, \quad t_n \cdot (\tilde{\mathbf{w}} \cdot \mathbf{x}_n) \geq 1 - \xi_n \text{ and } \xi_n \geq 0.$$

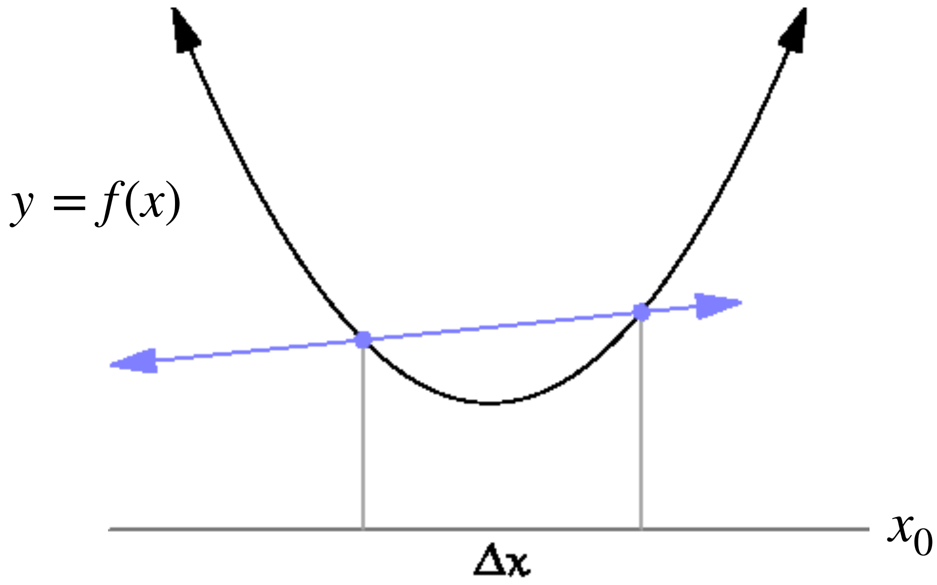
- C is constant that controls how costly constraint violations are.
- The problem is still convex.

- How do you minimize a function of several variables?
- Why does it matter that the problem is convex?

—> Let's talk about that today.

EPFL

Derivative of a 1-Variable Function

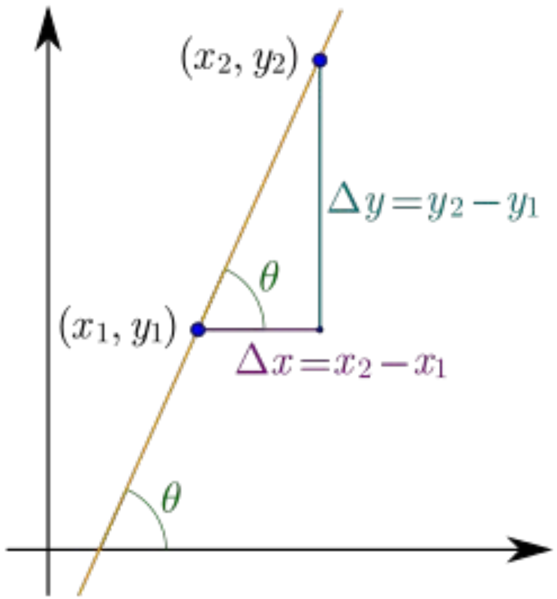


- The derivative of a function $y = f(x)$ of a single variable x is the rate at which y changes as x changes.
- It is measured for an infinitesimal change in x, starting from a point x_0, and written as

$$f'(x_0) = \frac{dy}{dx} = \lim_{\Delta x \to 0} \frac{f(x_0 + \Delta x) - f(x_0)}{\Delta x}$$

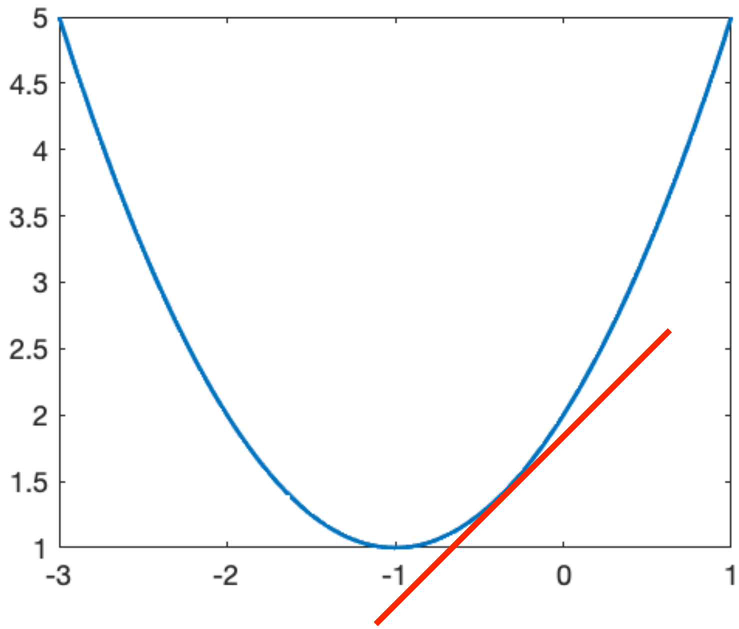
—> The derivative is the slope of the tangent at x_0.

Derivative of a Linear Function



- The tangent to the function is the function itself: The slope is constant.
- For example, $y = 2x - 1$ and $\dfrac{dy}{dx} = 2$.

Derivative of a Non-Linear Function



- The tangent (in red) to the function varies with x and so does the slope.
- For example, $y = x^2 + 2x + 2$ and $\dfrac{dy}{dx} = 2x + 2$.

Evolution of the Tangent

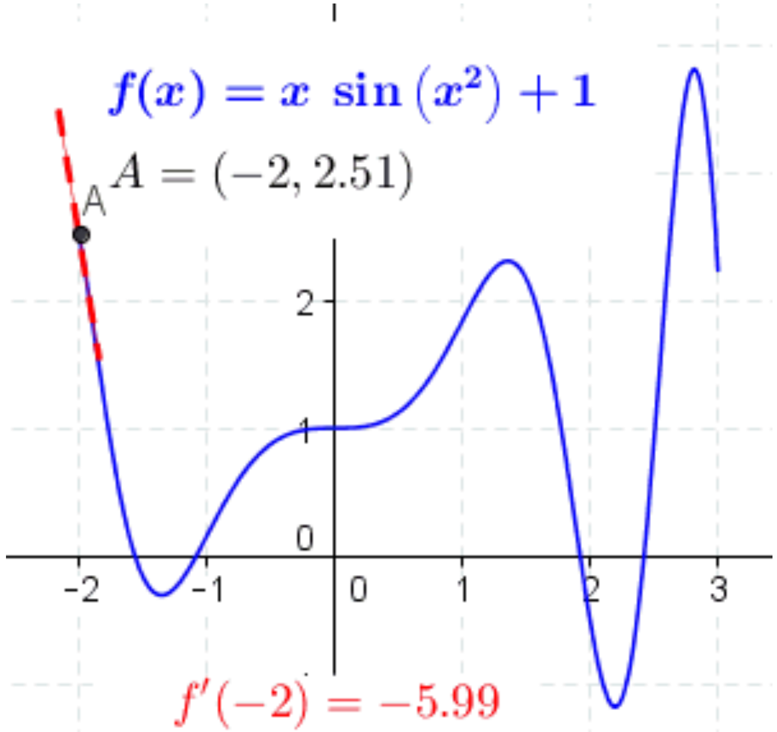
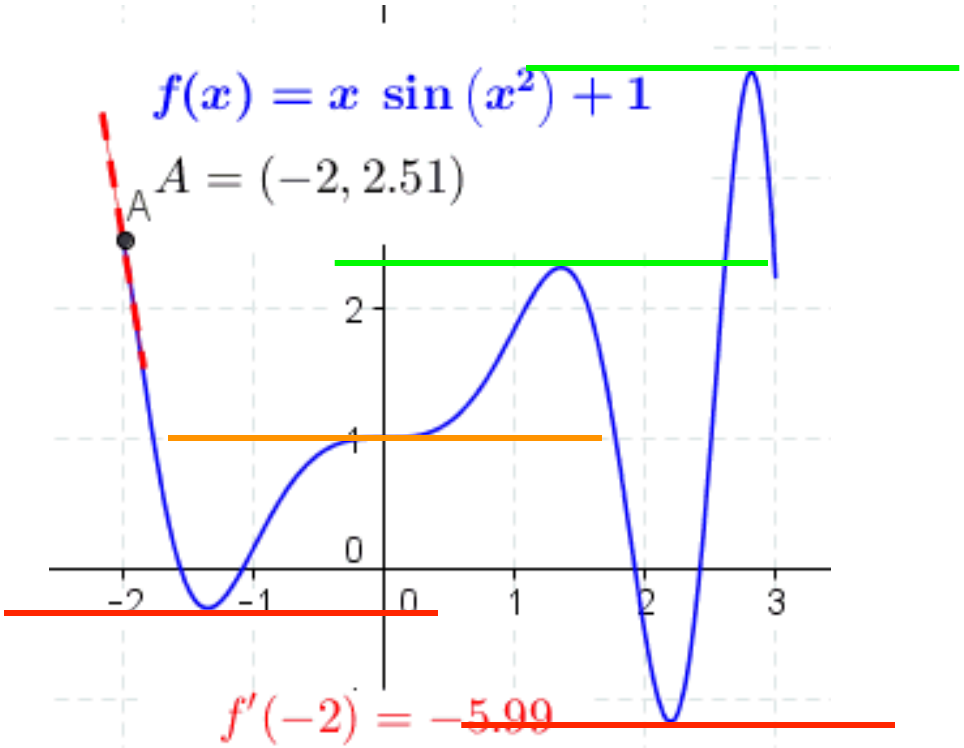


Figure from Wikipedia

$$y = x\sin(x^2) + 1$$

$$\frac{dy}{dx} = \sin(x^2) + 2x^2\cos(x^2)$$

First and Second Derivatives



$$y = x\sin(x^2) + 1$$

$$\frac{dy}{dx} = \sin(x^2) + 2x^2\cos(x^2)$$

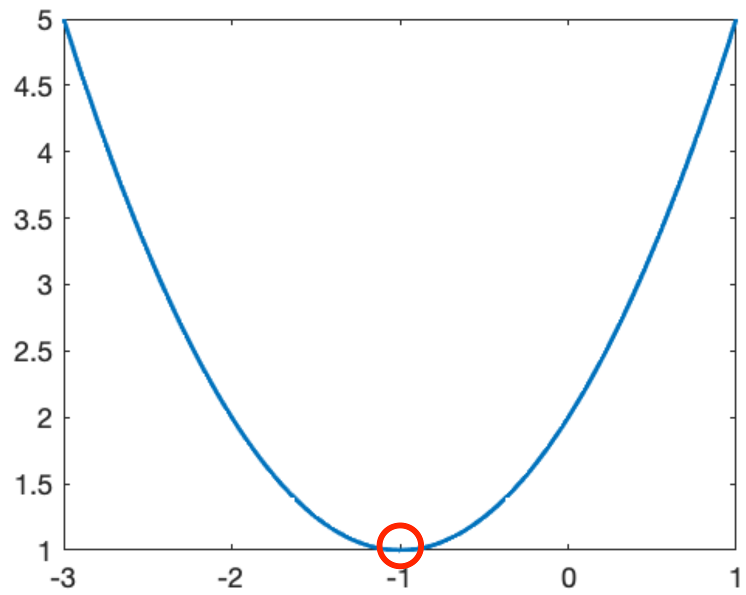
$$\frac{d^2y}{dx^2} = 6x\cos(x^2) - 4x^3\sin(x^2)$$

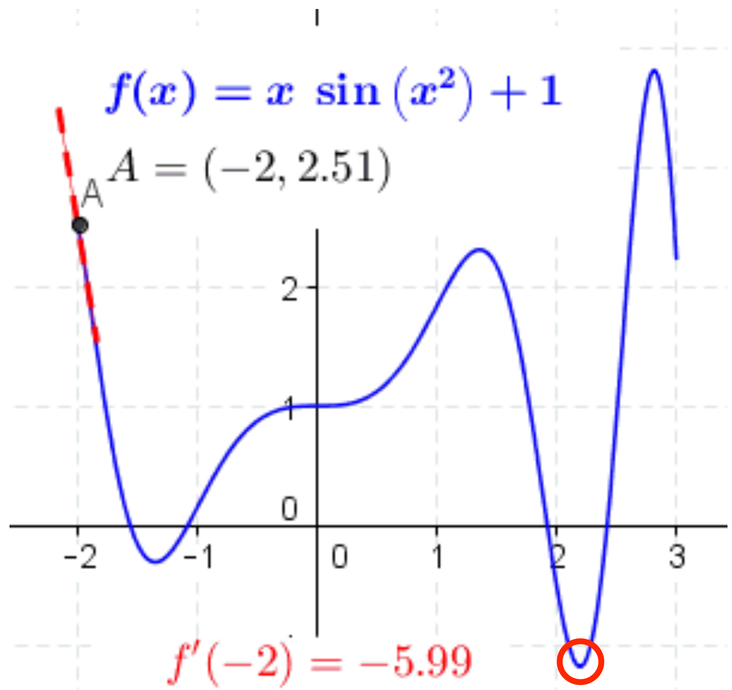
$$\frac{dy}{dx} = 0 \text{ and } \frac{d^2y}{dx^2} > 0 : \text{Minimum}$$

$$\frac{dy}{dx} = 0 \text{ and } \frac{d^2y}{dx^2} < 0 : \text{Maximum}$$

$$\frac{dy}{dx} = 0 \text{ and } \frac{d^2y}{dx^2} = 0 : \text{Saddle}$$

Convex vs Non-Convex



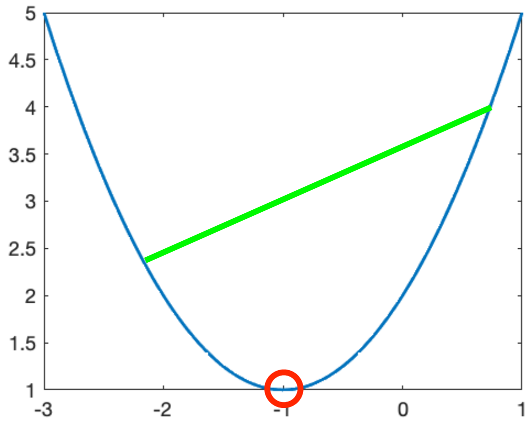


- There is only one minimum.
- The second derivative is >= 0.

- There are several **local** minima.
- There is one global minimum.

—> Non-convex functions are much more difficult to minimize than convex ones.

Minimizing a Convex Function



The line segment between any two points on the curve lies above the curve.

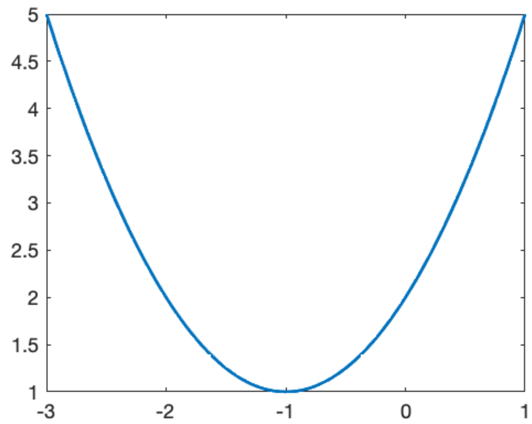
$$\frac{df(x^*)}{dx} = 0$$

For some simple functions this can be done in closed form, that is, by solving an equation.

Minimizing a Simple Convex Function

$$f(x) = x^2 + 2x + 2$$

$$\frac{df(x)}{dx} = 2x + 2$$



$$\frac{df(x^*)}{dx} = 0 \Leftrightarrow 2x^* + 2 = 0$$

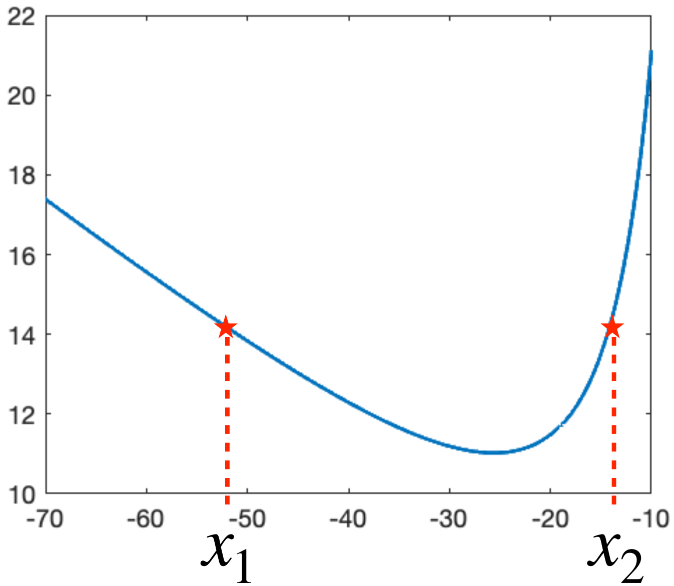
$$\Leftrightarrow 2x^* = -2$$

$$\Leftrightarrow x^* = -1$$

Minimizing a Generic Convex Function

When the minimum cannot be found in closed-form, we use the derivative:

At x_1, the slope is negative. Hence, one should move in the positive direction ($\Delta x > 0$) to go towards the minimum



At x_2, the slope is positive. Hence, one should move in the negative direction ($\Delta x < 0$) to go towards the minimum

—> One should move in the direction opposite to the derivative for minimization

Minimizing a Convex Function

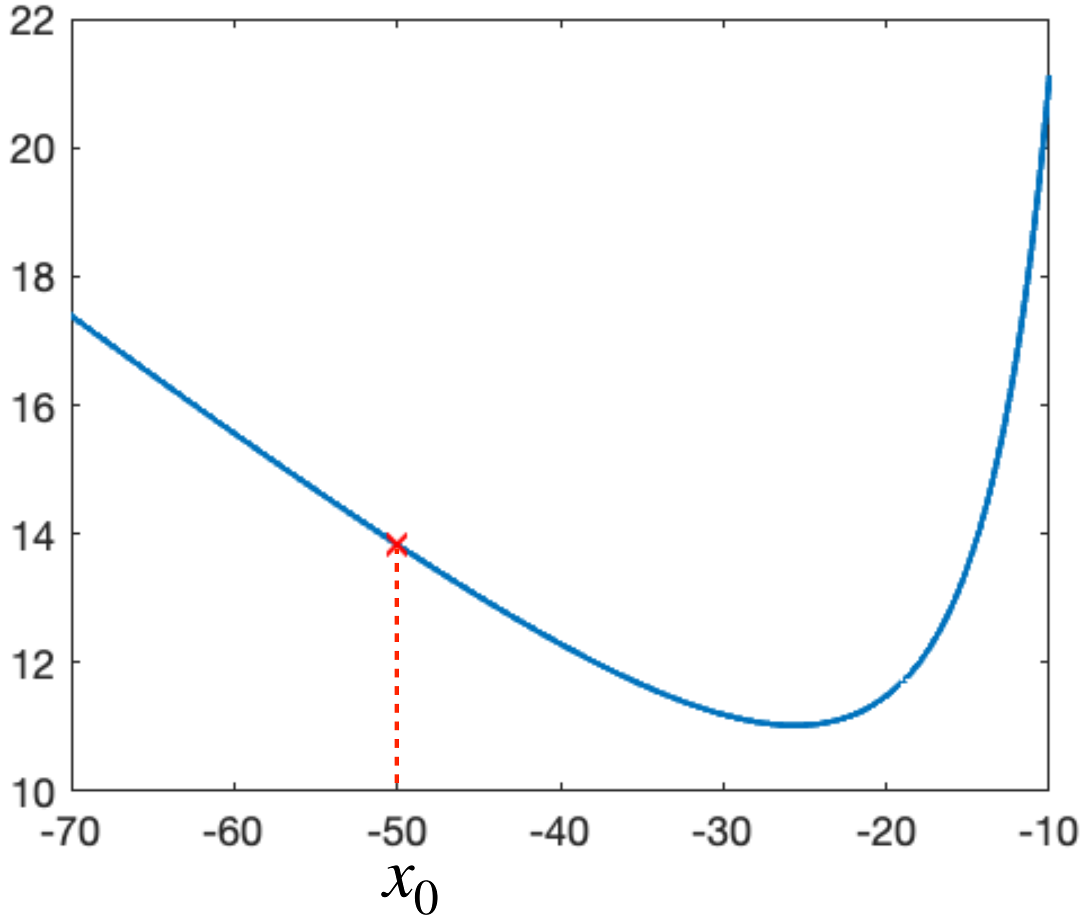
Simplest algorithm:

1. Initialize x_0 (e.g., randomly)
2. While not converged

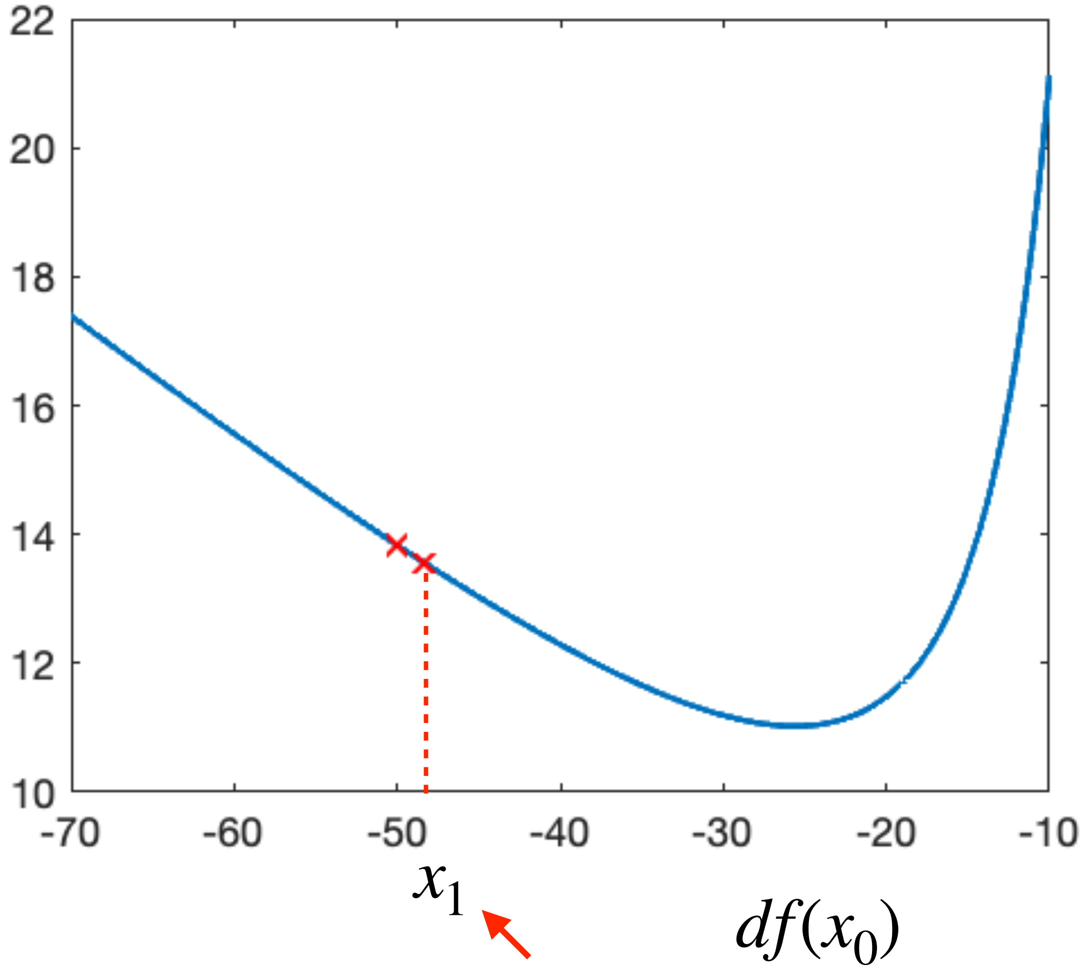
    2.1. Update $x_k \leftarrow x_{k-1} - \eta \frac{df(x_{k-1})}{dx}$

- η defines the step size of each iteration.
- In ML, it is often referred to as the *learning rate*.

Minimizing a Convex Function

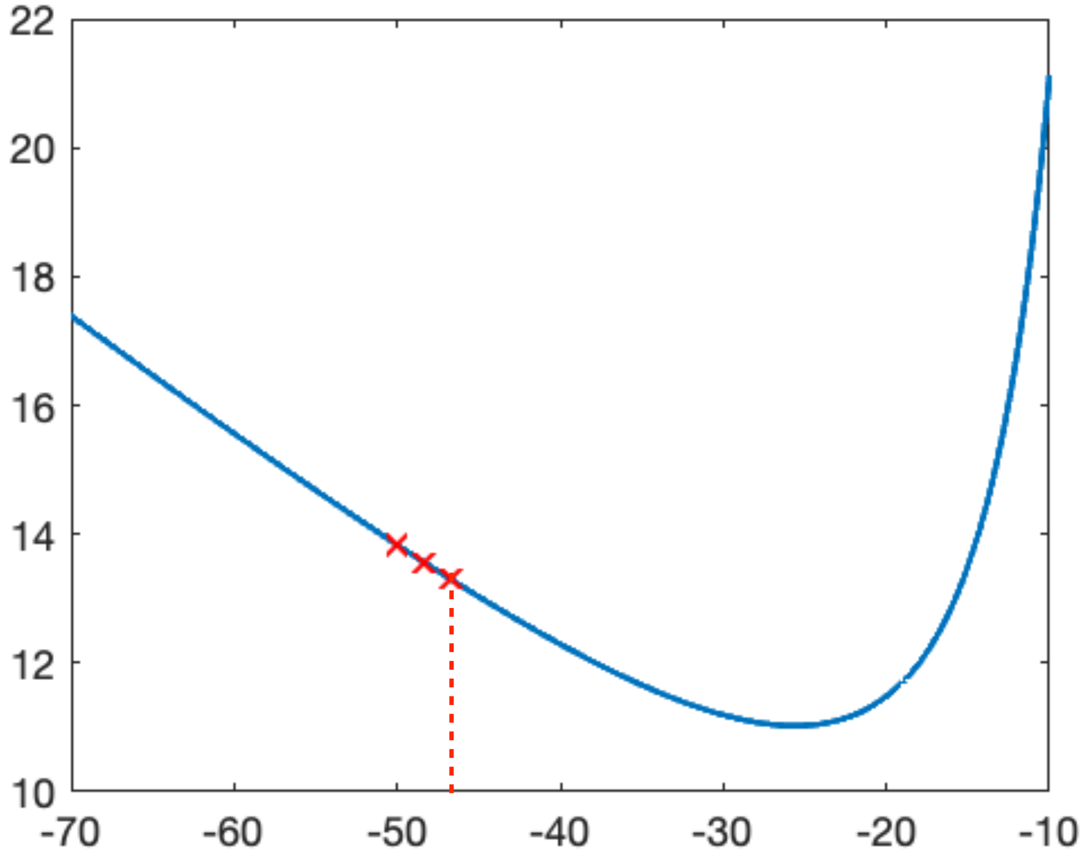


Minimizing a Convex Function



$$x_0 - \eta \frac{df(x_0)}{dx}$$

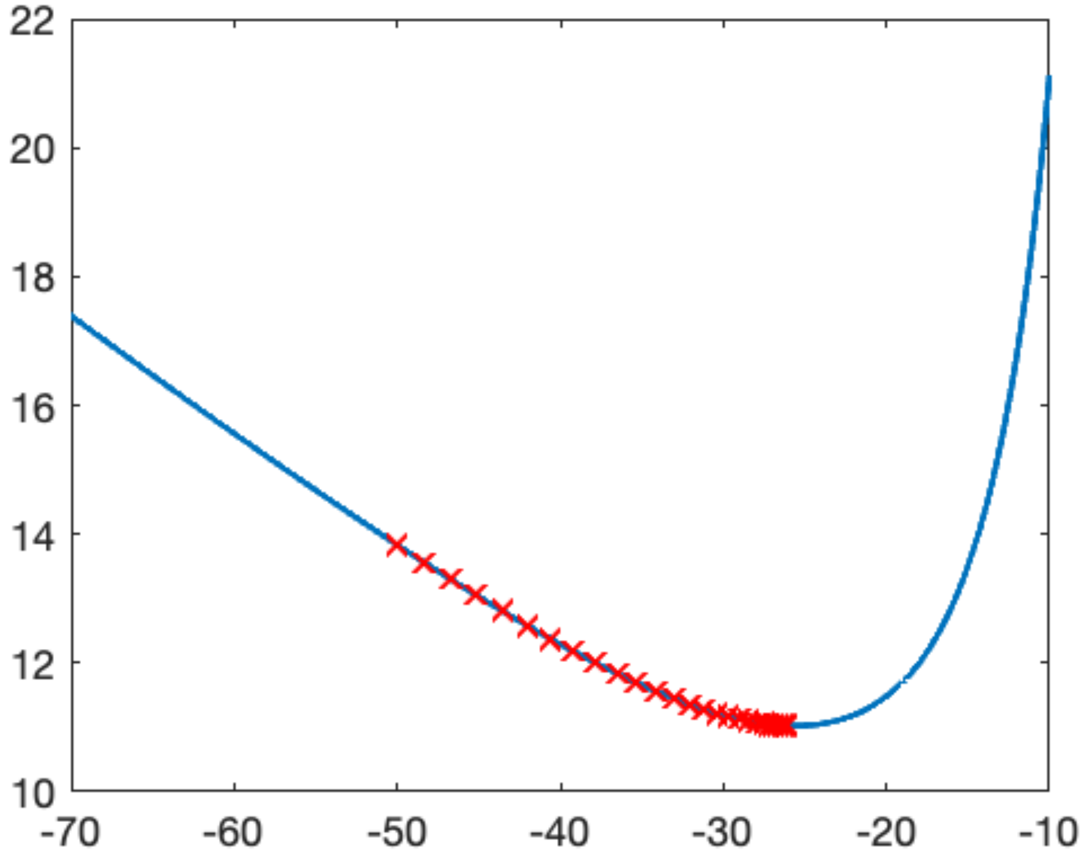
Minimizing a Convex Function



$$x_2 \leftarrow x_1 - \eta \frac{df(x_1)}{dx}$$

EPFL

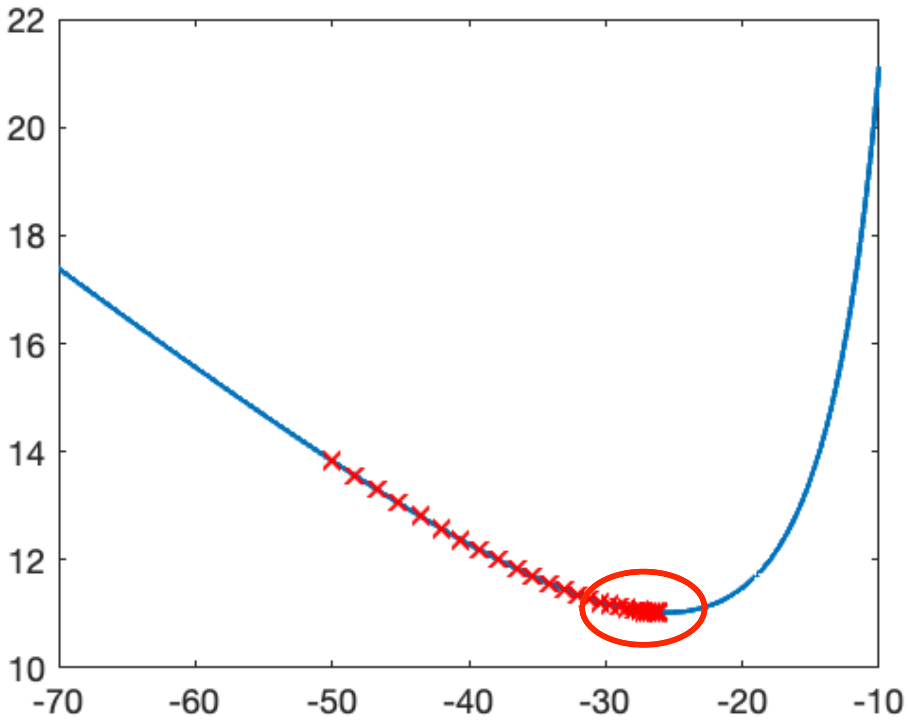
Minimizing a Convex Function



Minimizing a Convex Function

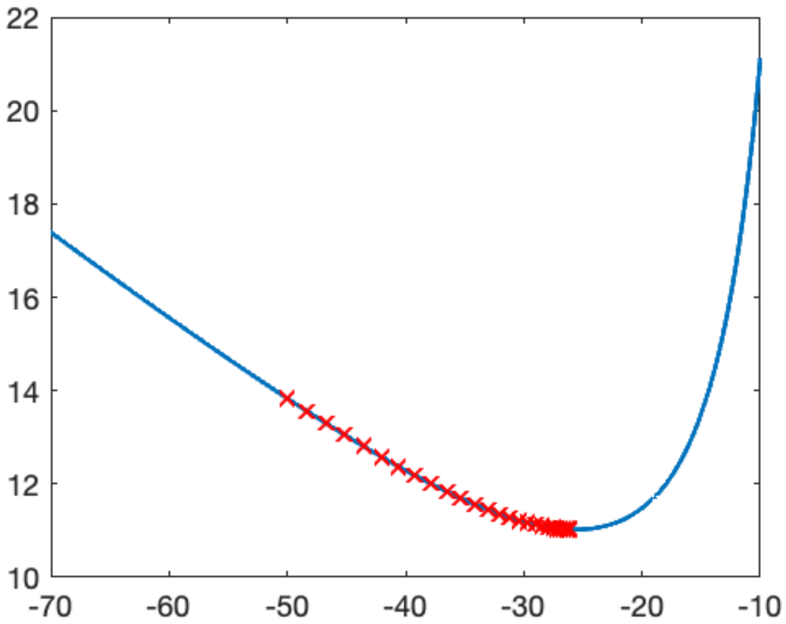
Potential stopping Criteria:

- Change in function value less than threshold: $|f(x_{i-1}) - f(x_i)| < \delta$.
- Change in parameter value less than threshold: $|x_{i-1} - x_i| < \delta$.
- Maximum number of iterations reached without a guarantee to have reached the minimum.



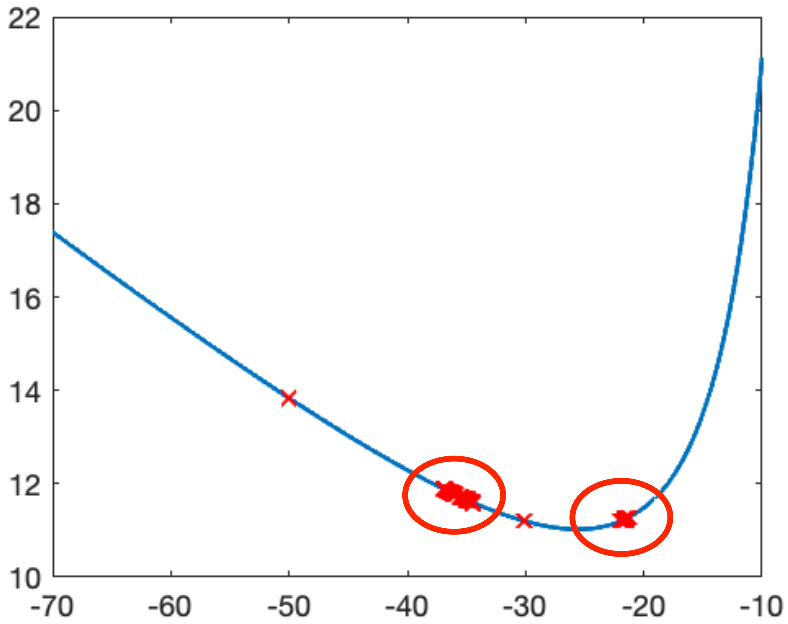
Influence of the Step Size

$\eta = 10$



The steps are of the appropriate size for convergence.

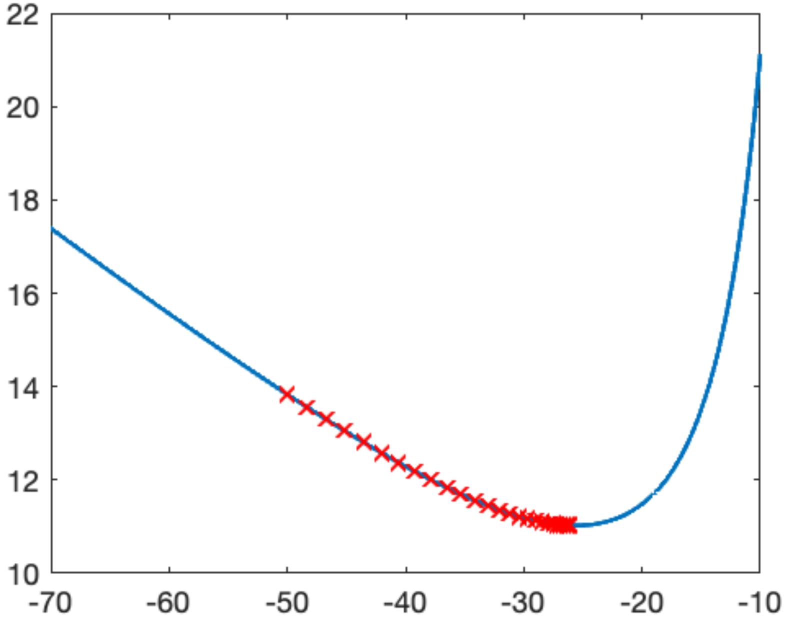
$\eta = 120$



The steps are too large and the algorithm starts jumping between these two points.

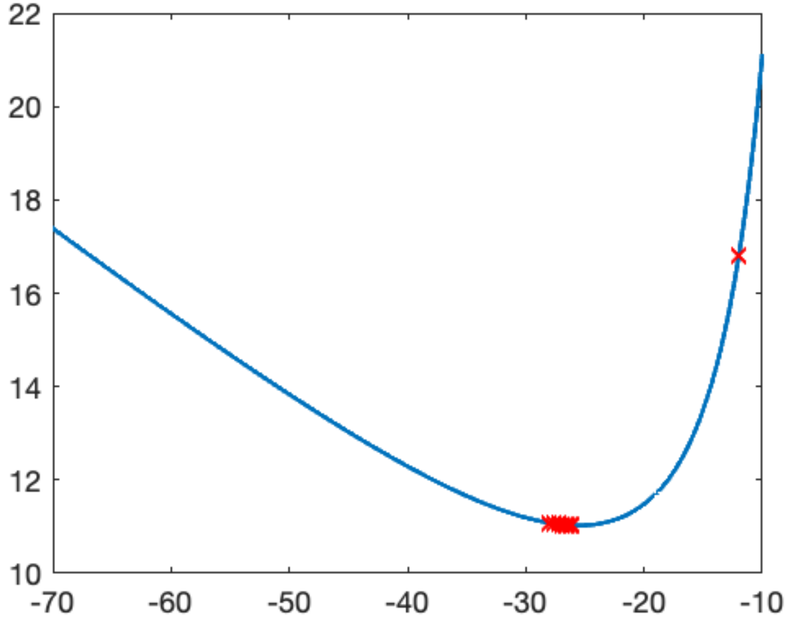
Influence of the Starting Point

$x_0 = -50$



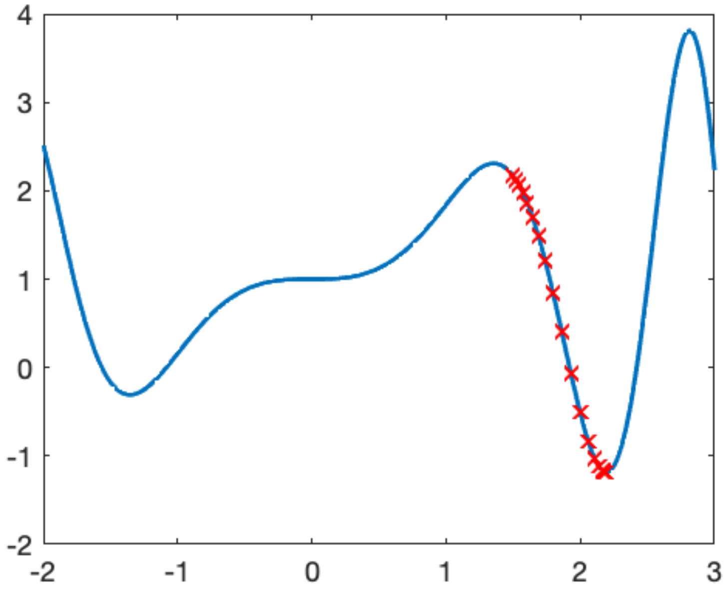
Converges.

$x_0 = -12$



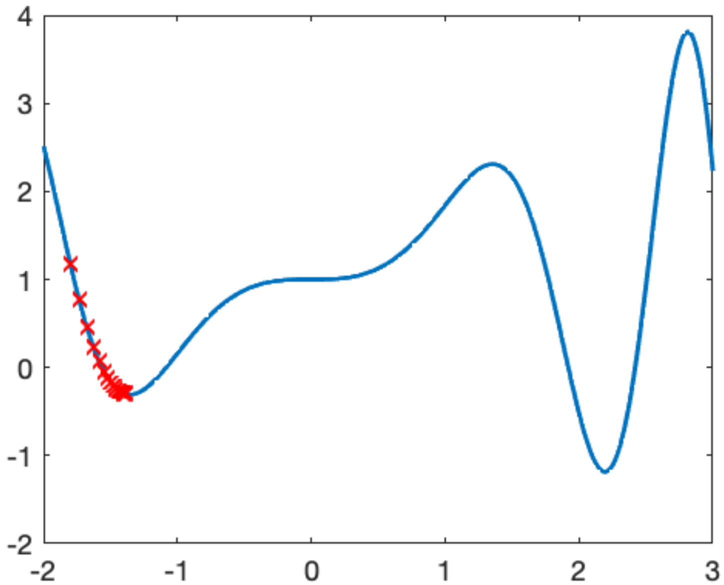
Converges to the same place, but faster.

Minimizing a Non-Convex Function



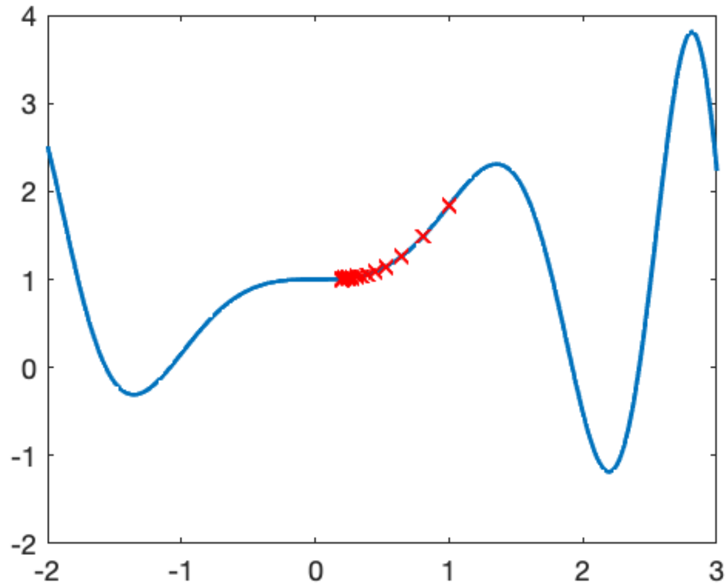
$x_0 = 1.5$

Global minimum



$x_0 = -1.8$

Local minimum



$x_0 = 1$

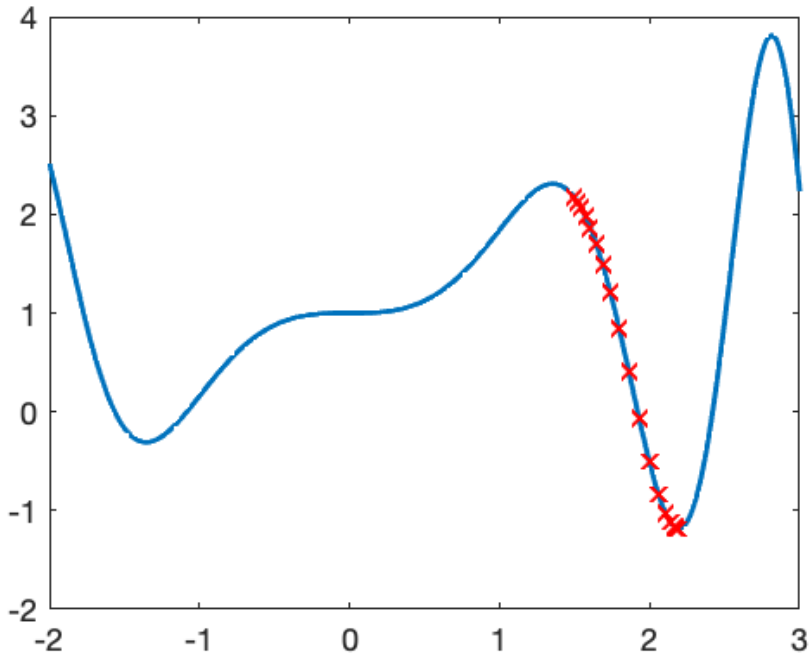
Saddle point

EPFL

Minimizing a Non-Convex Function

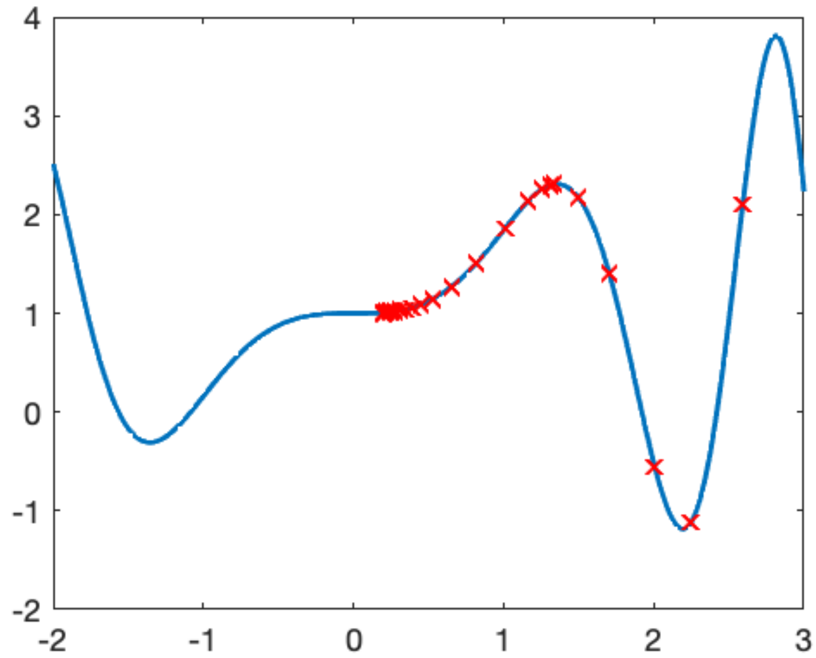
$\eta = 0.01, \ \ x_0 = 1.5$

Global minimum



$\eta = 0.1, \ \ x_0 = 1.5$

Saddle point



—> No guarantees when the function is not convex!

EPFL

Functions of Multiple Variables

Multivariate function:

$$f : R^D \rightarrow R$$

$$y = f(\mathbf{x}) = f(x_1, \ldots, x_D)$$

Partial derivative:

$$\frac{\delta y}{\delta x_d} = \lim_{\Delta x \rightarrow 0} \frac{f(\ldots, x_d + \Delta x, \ldots) - f(\ldots, x_d, \ldots)}{\Delta x}$$

Gradient vector:

$$\nabla f = [\frac{\delta f}{\delta x_1}, \ldots, \frac{\delta f}{\delta x_D}]$$

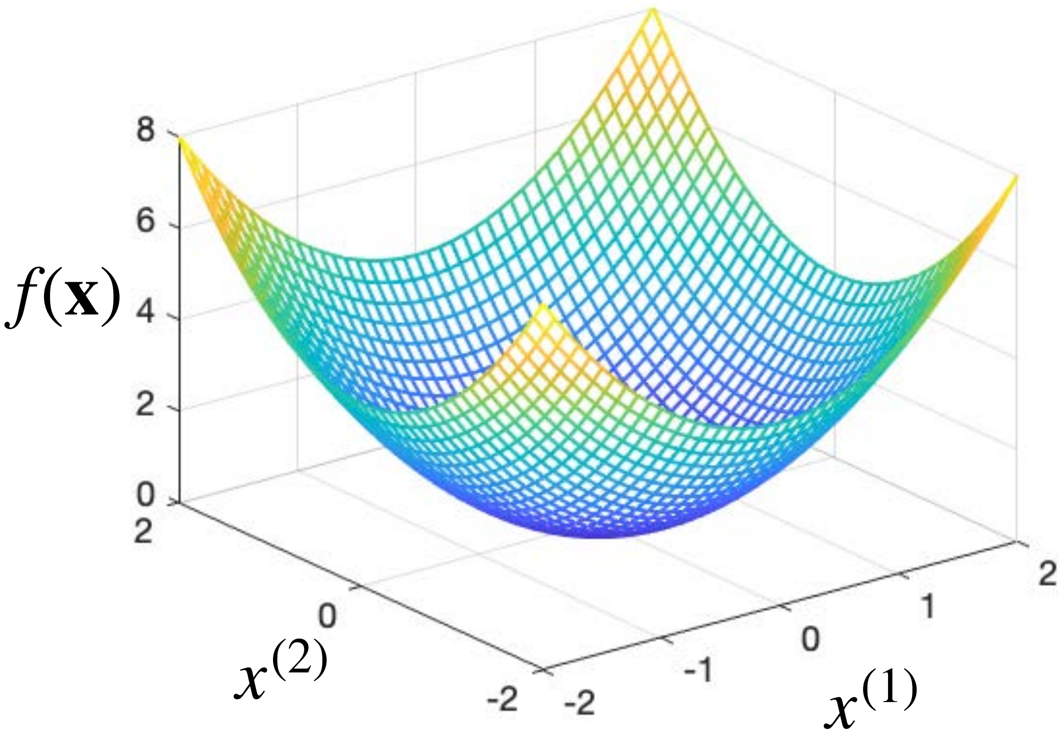
Quadratic Function

$$f(\mathbf{x}) = x_1^2 + x_2^2$$

$$\frac{\partial f}{\partial x_1} = 2x_1$$

$$\frac{\partial f}{\partial x_2} = 2x_2$$

$$\nabla f(\mathbf{x}) = \begin{bmatrix} 2x_1 \\ 2x_2 \end{bmatrix} \in \mathbb{R}^2$$



The color also represents the value of $f(\mathbf{x})$

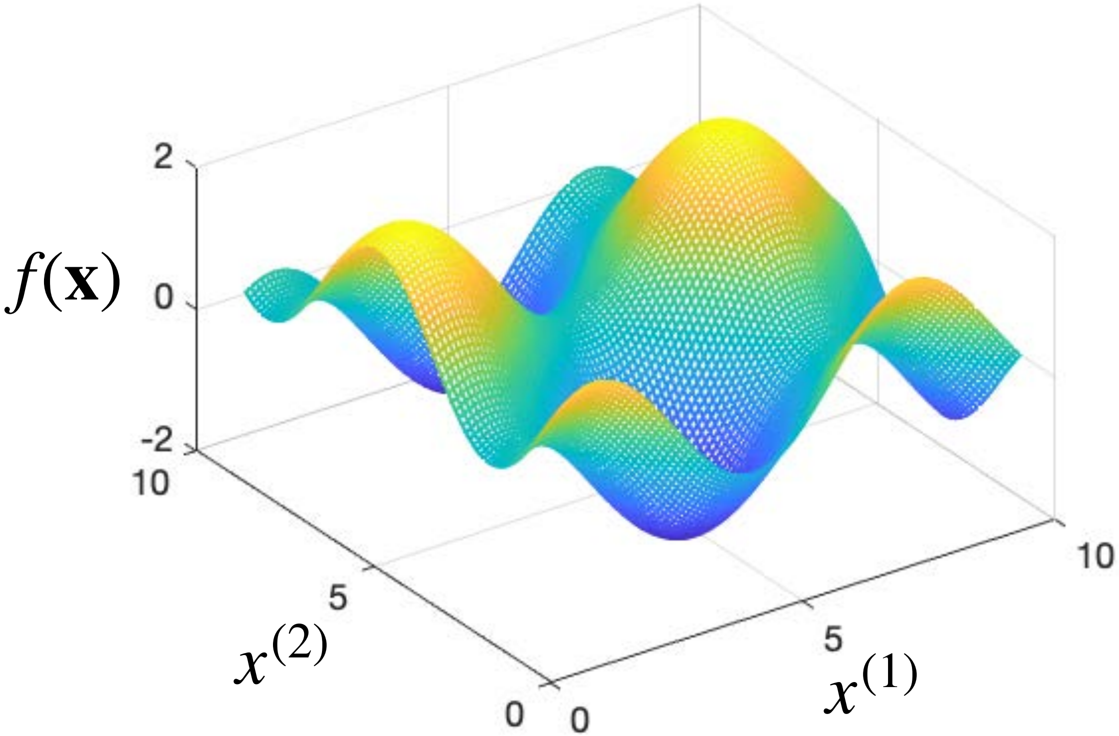
Sinusoidal Function

$$f(\mathbf{x}) = \sin x_1 + \cos x_2$$

$$\frac{\partial f}{\partial x_1} = \cos(x_1)$$

$$\frac{\partial f}{\partial x_2} = -\sin(x_2)$$

$$\nabla f(\mathbf{x}) = \begin{bmatrix} \cos(x_1) \\ -\sin(x_2) \end{bmatrix} \in \mathbb{R}^2$$



The color also represents the value of $f(\mathbf{x})$

Gradient in 4 Dimensions

$$f(\mathbf{x}) = x_1^2 x_2^2 + x_1 x_2 x_3 + x_3 x_4 + 2x_4 + 1$$

$$\frac{\partial f}{\partial x_1} = 2x_1 x_2^2 + x_2 x_3$$

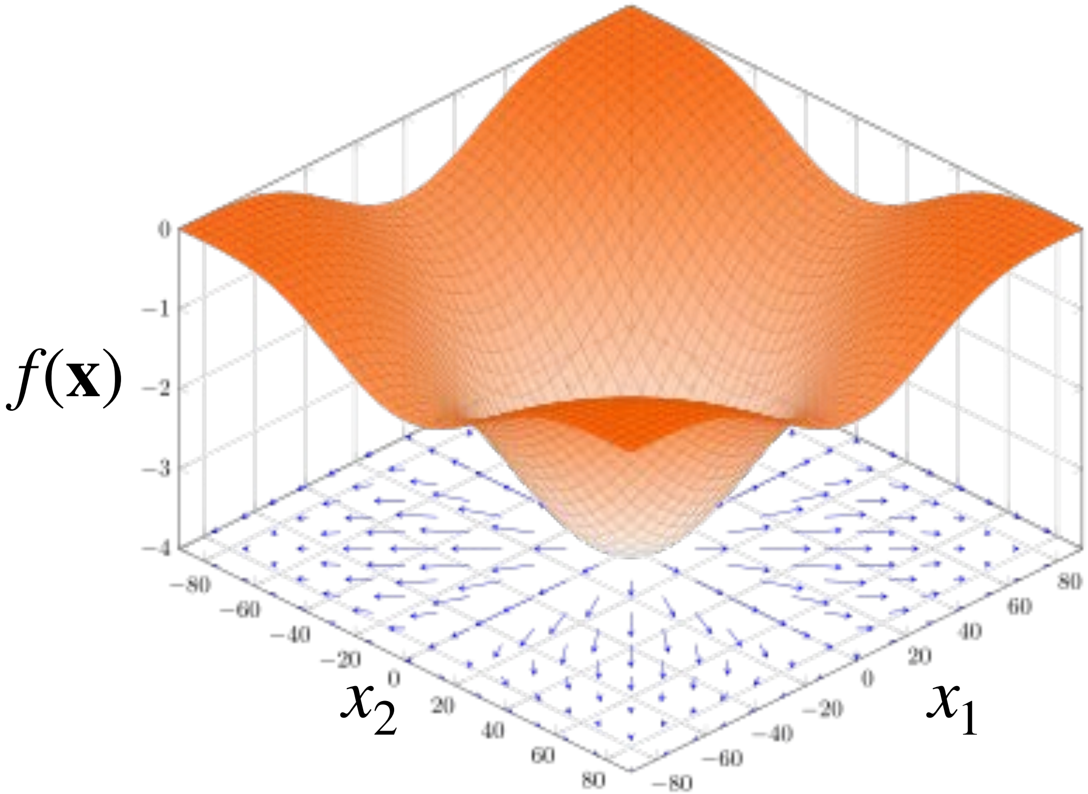
$$\frac{\partial f}{\partial x_2} = 2x_2 x_1^2 + x_1 x_3$$

$$\frac{\partial f}{\partial x_3} = x_1 x_2 + x_4$$

$$\frac{\partial f}{\partial x_4} = x_3 + 2$$

$$\nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \frac{\partial f}{\partial x_3} \\ \frac{\partial f}{\partial x_4} \end{bmatrix} \in \mathbb{R}^4$$

Gradient Properties

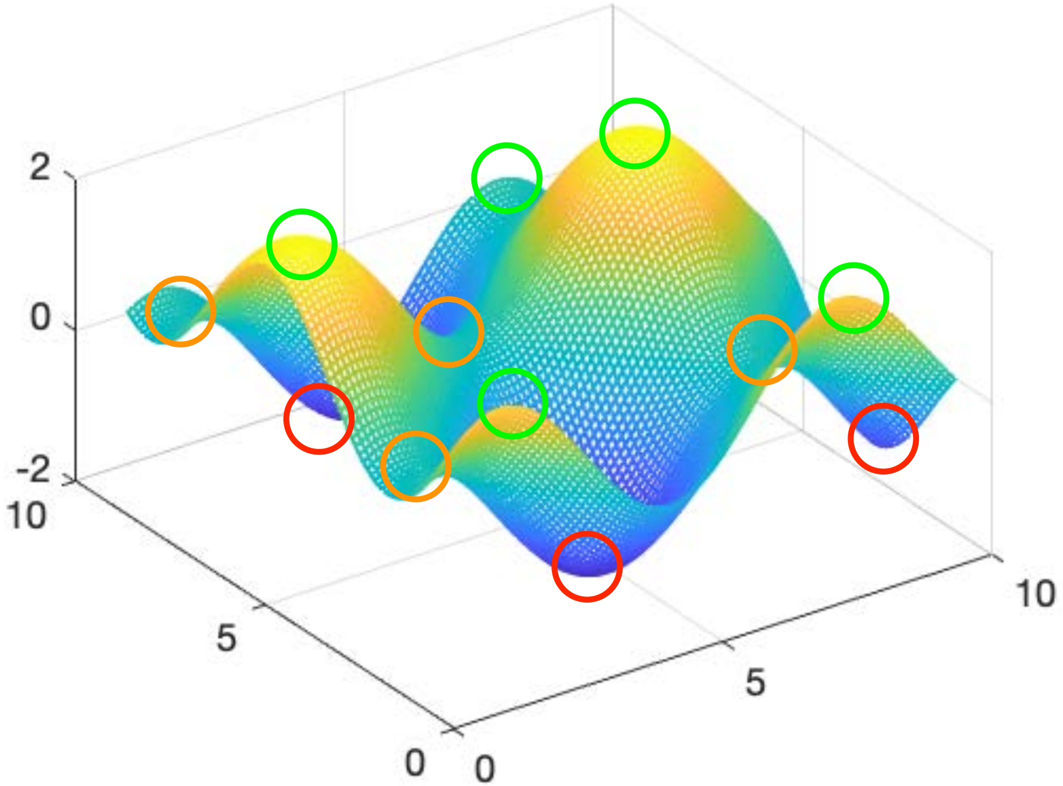


- The gradient at a point $\mathbf{x}$ indicates the direction of greatest increase of the function at $\mathbf{x}$.
- Its magnitude is the rate of increase in that direction.

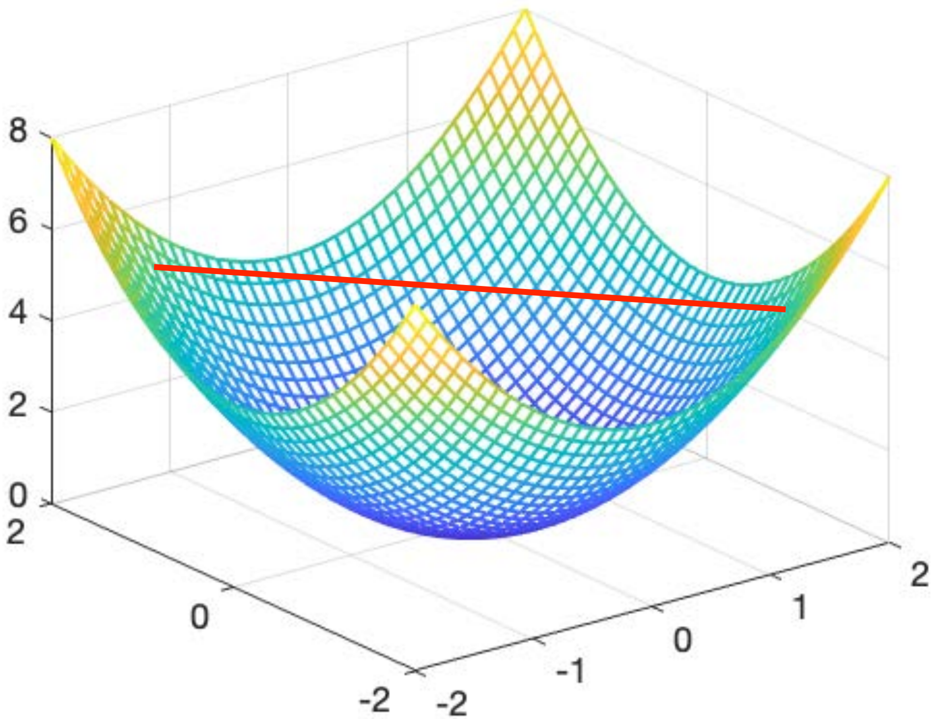
Gradient Properties

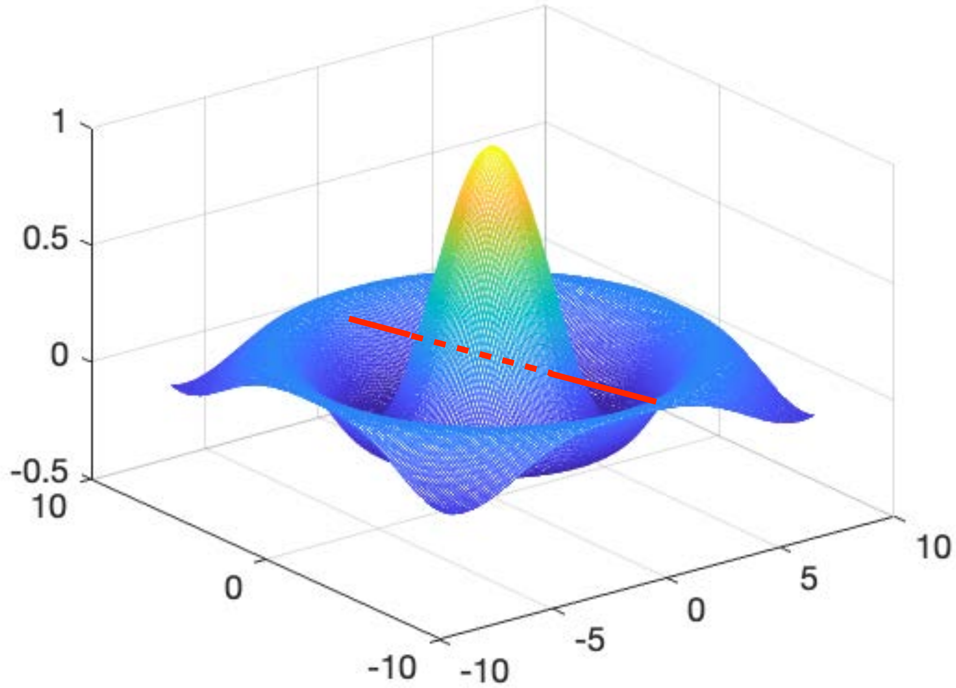
The gradient vanishes (becomes a zero vector) at the stationary points of the function:

- Minima,
- Maxima,
- Saddle points.



Convex vs Non-Convex

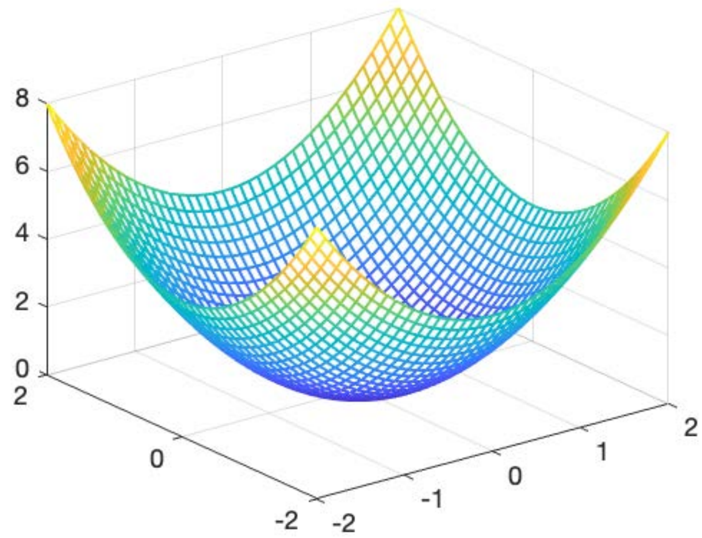




Convex: The line segment between any two points on the function lies above the function

Non-convex: At least one line segment between two points lies in part below the function.

Minimizing a Convex Function



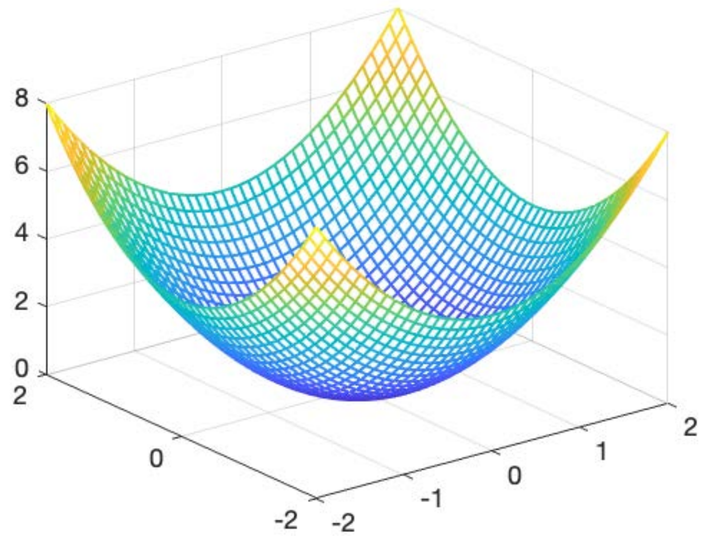
$$\nabla f(\mathbf{x}^*) = \mathbf{0}$$

- Because the gradient is a vector, this yields a system of equations.
- It can still be solved in closed form for some functions.

Minimizing a Simple Convex Function

$$f(\mathbf{x}) = x_1^2 + x_2^2$$

$$\frac{\partial f}{\partial x_1} = 2x_1 \qquad \frac{\partial f}{\partial x_2} = 2x_2$$



$$\nabla f(\mathbf{x}) = 0 \Leftrightarrow \begin{cases} 2x_1 = 0 \\ 2x_2 = 0 \end{cases}$$

$$\Leftrightarrow x_1 = x_2 = 0$$

Revisiting K means

$$\min_{\{\mu_k\},\{r_i^k\}} \sum_{i=1}^{N} \sum_{k=1}^{K} r_i^k \|\mathbf{x}_i - \mu_k\|^2$$

such that $\quad r_i^k \in \{0,1\}, \;\; \forall i, k$

$$\sum_{k=1}^{K} r_i^k = 1, \;\; \forall i$$

—> We will derive the solution by alternating between the two types of variables.

Revisiting K means

$$\min_{\{r_i^k\}} \sum_{i=1}^{N} \sum_{k=1}^{K} r_i^k \|\mathbf{x}_i - \mu_k\|^2$$

such that $\quad r_i^k \in \{0,1\}, \ \forall i, k$

$$\sum_{k=1}^{K} r_i^k = 1, \ \forall i$$

- Because of the constraints, for each sample, only one r_i^k can be 1.
- We take it to be the one corresponding to the nearest center:

$$r_i^k = \begin{cases} 1, \ \text{if } k = \underset{j}{\operatorname{argmin}} \|\mathbf{x}_i - \mu_j\|^2 \\ 0, \ \text{otherwise} \end{cases}$$

Revisiting K means

$$\min_{\{\mu_k\}} \sum_{i=1}^{N} \sum_{k=1}^{K} r_i^k \|\mathbf{x}_i - \mu_k\|^2$$

- This can be done by zeroing out the gradient for each center:

$$\frac{\partial R}{\partial \mu_k} = 2 \sum_{i=1}^{N} r_i^k (\mathbf{x}_i - \mu_k) = 0$$

- This yields:

$$\mu_k = \frac{\sum_{i=1}^{N} r_i^k \mathbf{x}_i}{\sum_{i=1}^{N} r_i^k}$$

- This corresponds to the mean of the samples assigned to cluster k.

Back to Logistic Regression

- Replace the step function by a smooth function σ.
- The prediction becomes $y(\mathbf{x};\tilde{\mathbf{w}}) = \sigma(\tilde{\mathbf{w}} \cdot \tilde{\mathbf{x}})$.
- Given the training set $\{(\mathbf{x}_n, t_n)_{1 \leq n \leq N}\}$ where $t_n \in \{0, 1\}$, minimize the cross-entropy

$$E(\tilde{\mathbf{w}}) = -\sum_n \{t_n \ln y_n + (1 - t_n) \ln(1 - y_n)\}$$

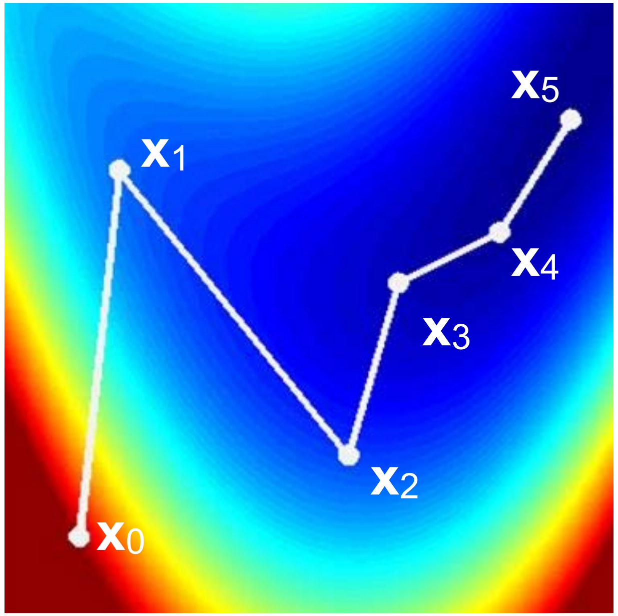
$$y_n = y(\mathbf{x}_n; \tilde{\mathbf{w}})$$

with respect to $\tilde{\mathbf{w}}$.

E is convex but cannot be minimized in closed form!

EPFL

Gradient Descent



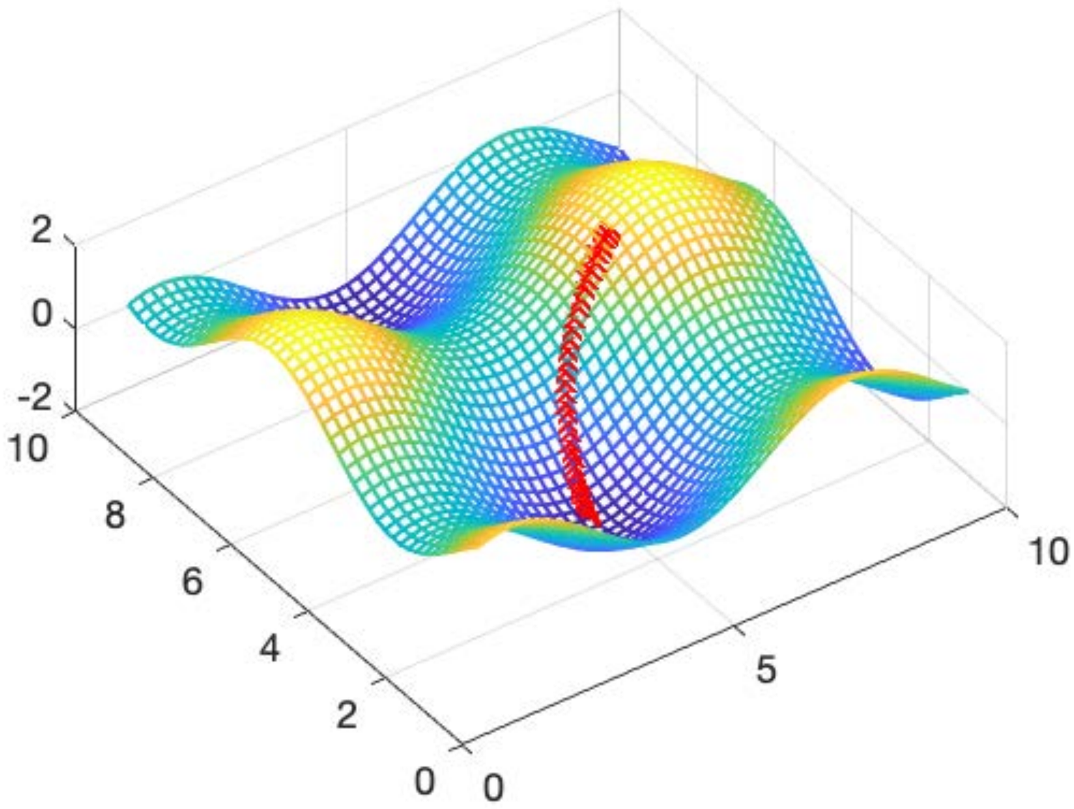
The gradient replaces the derivative.

Simplest algorithm:

1. Initialize $\mathbf{x}_0$ (e.g., randomly)
2. While not converged
   2.1. Update $\mathbf{x}_k \leftarrow \mathbf{x}_{k-1} - \eta \nabla f$

- η defines the step size of each iteration.
- In ML, it is often referred to as the *learning rate.*

Minimizing a Non-convex Function



$$f(\mathbf{x}) = \sin x_1 + \cos x_2$$

$$\nabla f(\mathbf{x}) = \begin{bmatrix} \cos(x_1) \\ -\sin(x_2) \end{bmatrix} \in \mathbb{R}^2$$

Stopping criteria:

- Thresholding the change in function value.
- Thresholding the change in parameters, i.e. $\|\mathbf{x}_{k-1} - \mathbf{x}_k\| < \delta$.

EPFL

Theoretical Justification

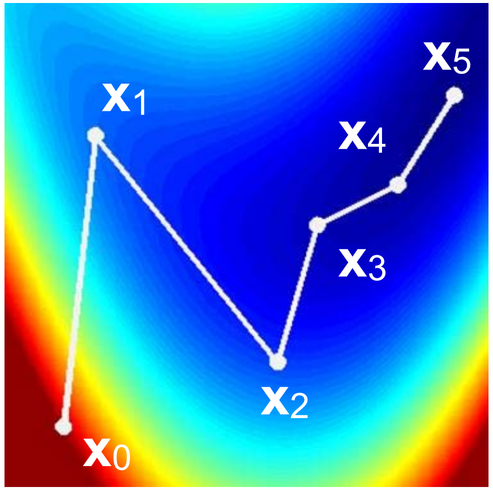
Steepest gradient descent:

$$\mathbf{x}_k \leftarrow \mathbf{x}_{k-1} - \eta \nabla f$$

First order Taylor expansion:

$$f(\mathbf{x} + \mathbf{dx}) \approx f(\mathbf{x}) + \nabla f(\mathbf{x})^T \mathbf{dx}$$

$$f(\mathbf{x} - \eta \nabla f(\mathbf{x})) \approx f(\mathbf{x}) - \eta \|\nabla f(\mathbf{x})\|^2 < f(\mathbf{x})$$

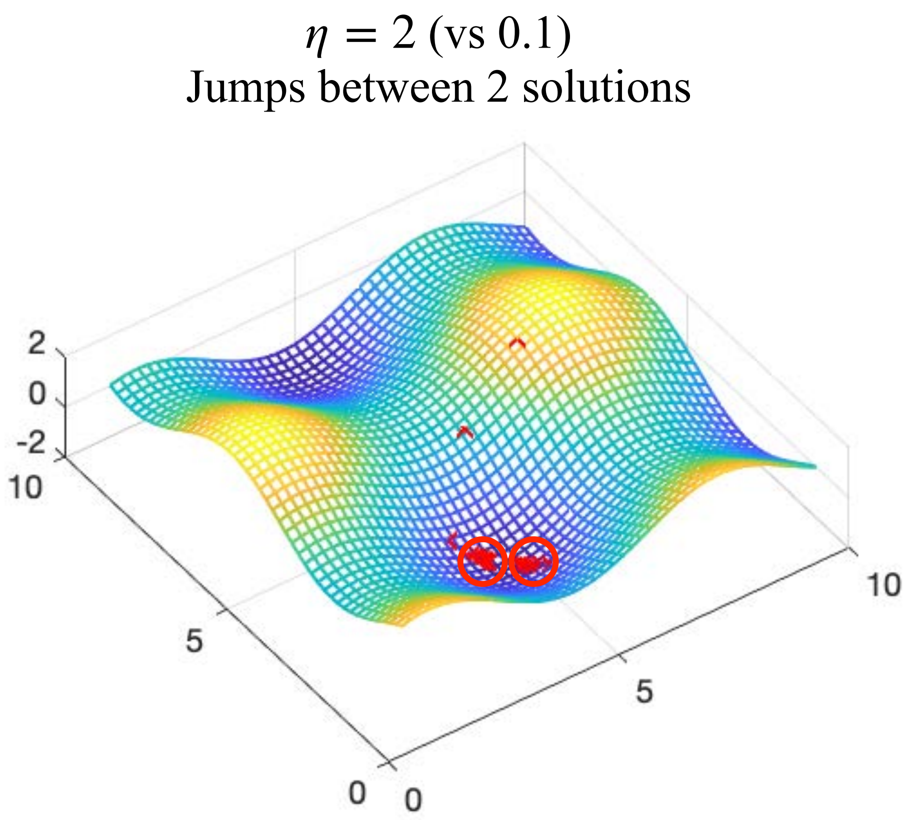


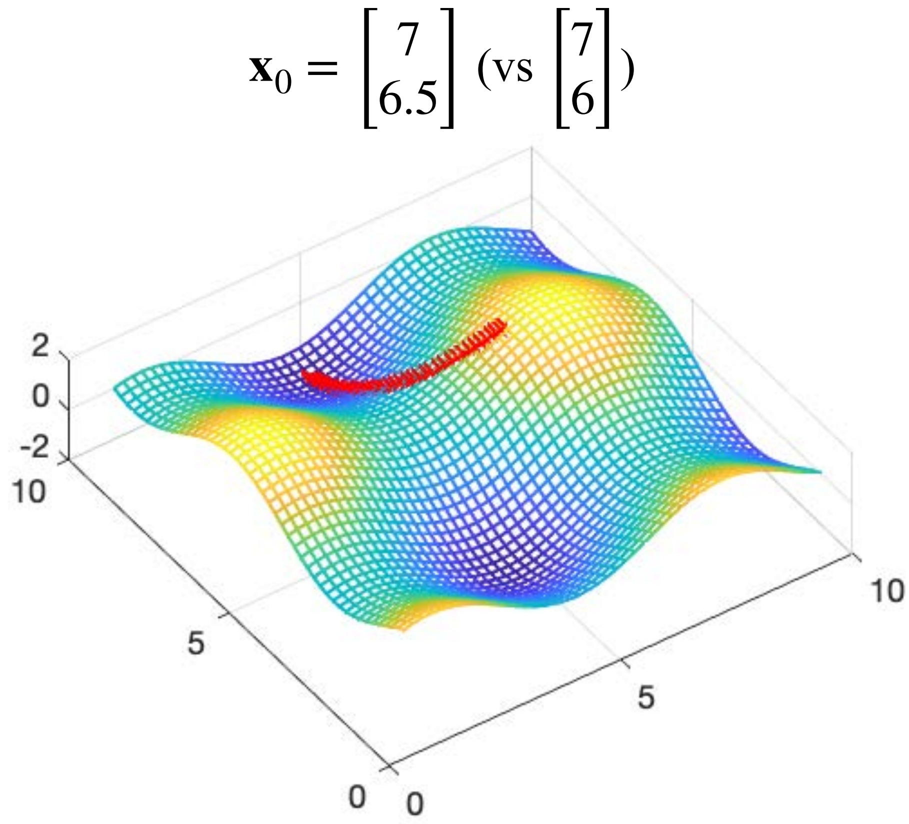
Issues:

- Justification but no guarantee
- How do we choose choose η?
- Many iterations in long and narrow valleys.

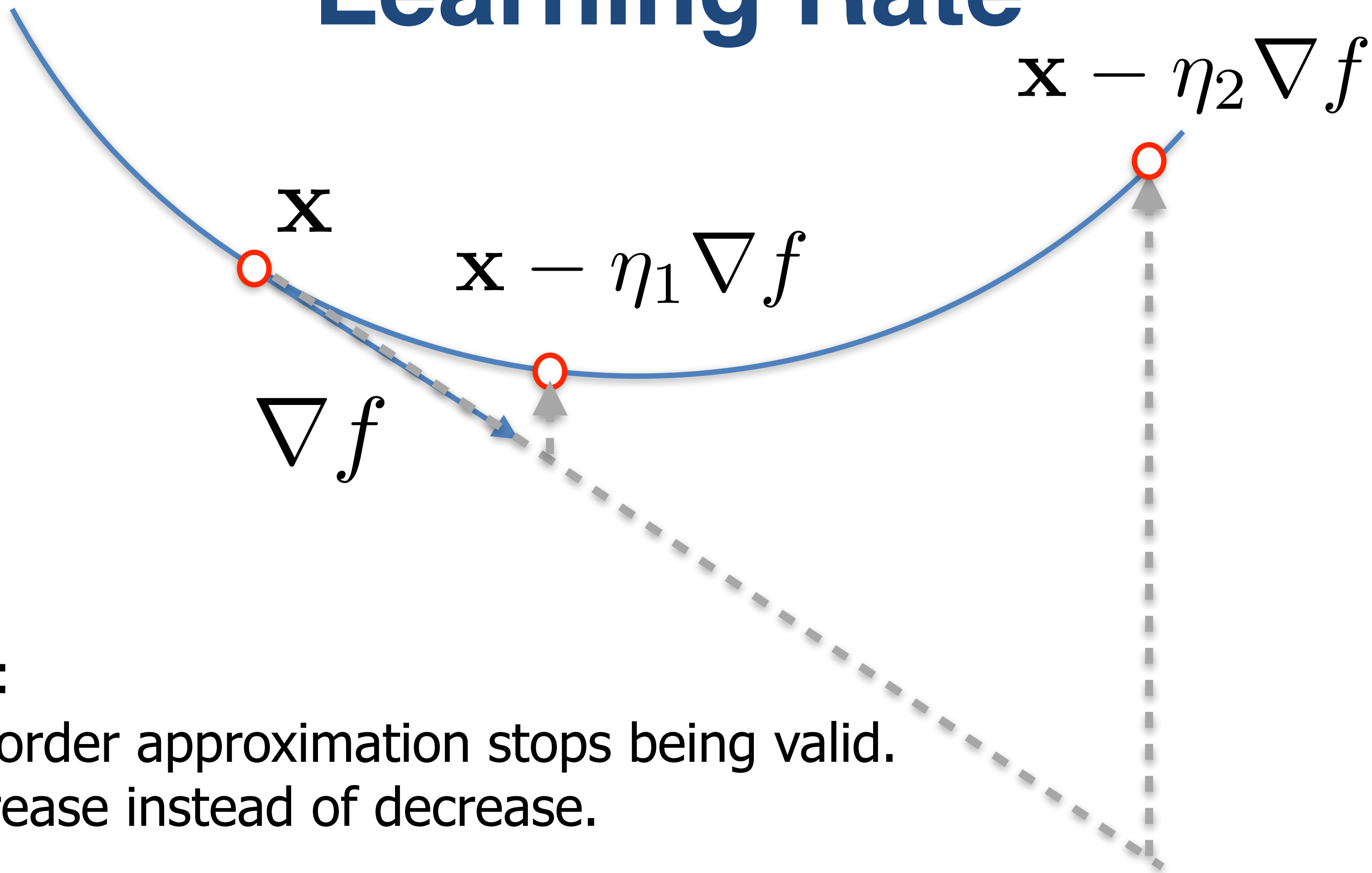
Triggs et al., Bundle Adjustment, 2000

Trouble Spots





Learning Rate



η too large:

- The first order approximation stops being valid.
- f can increase instead of decrease.

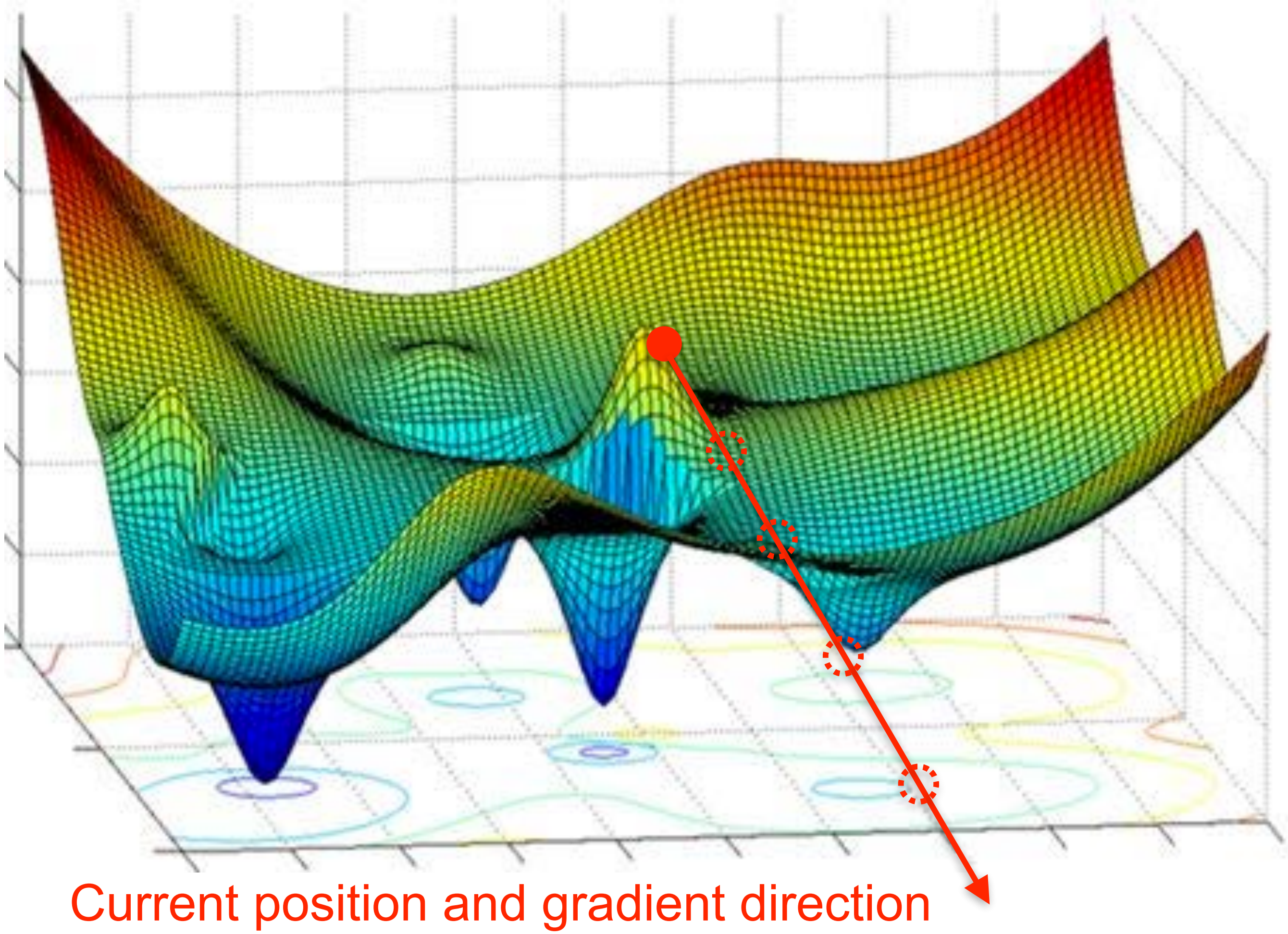
η too small:

- Convergence rate will be very slow.

Partial solution:

- Instead of using a fixed learning rate perform a line search in the direction of the gradient.

Line Search



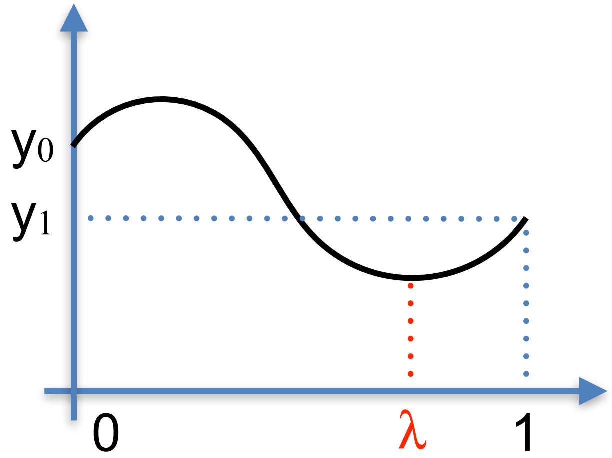
- Search along the gradient direction for a minimum.
- This is a 1D search and therefore doable.

Python Implementation

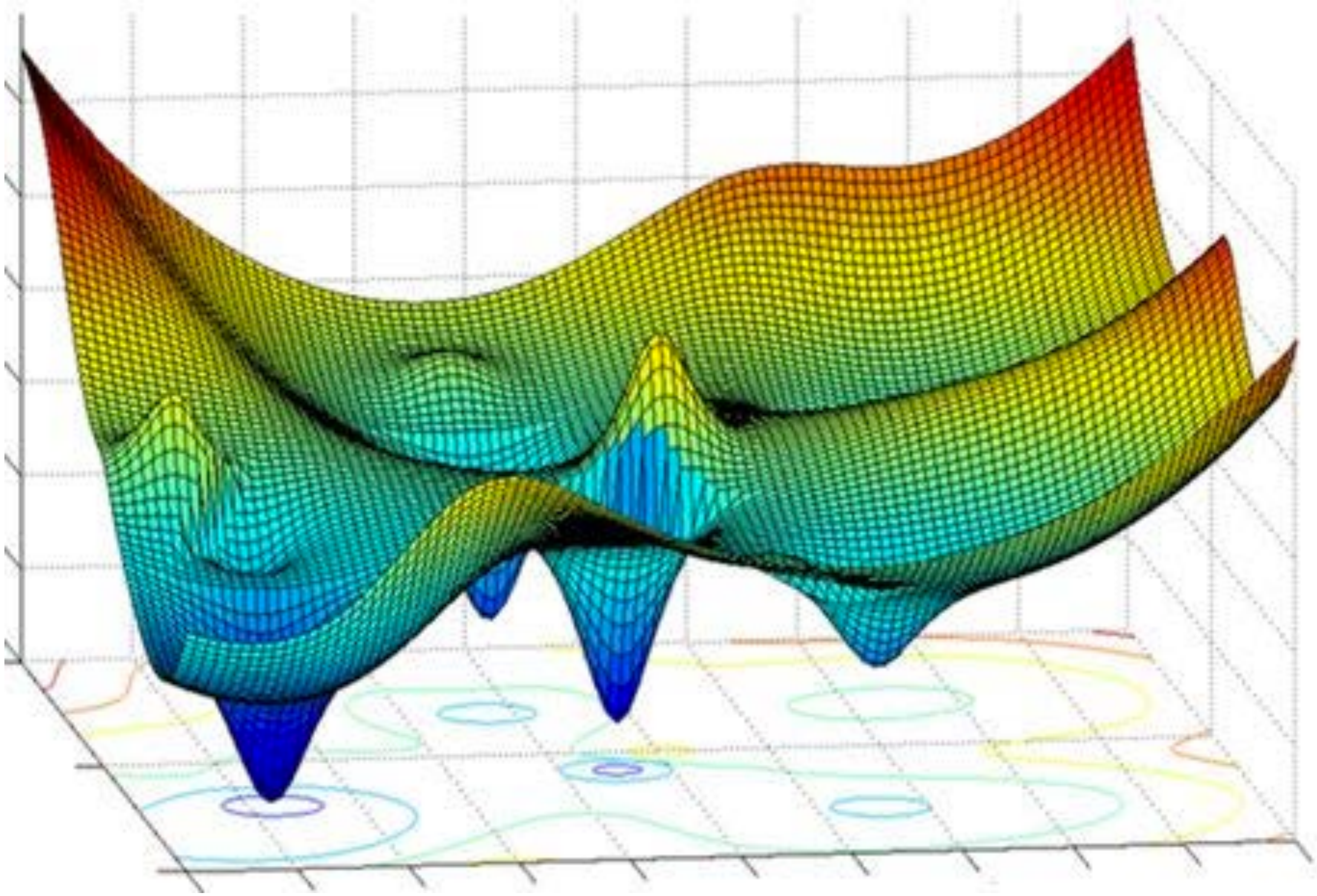
```
def steepestGrad(objF,x0,nIt=100,eps=1e-6,step=1.0):

    for i in range(nIt):

        y0,g0=objF(x0)                          # Compute the value of objF and its gradient.
        x1=x0-step*g0                           # Take a step in the direction of the gradient.
        y1,_=objF(x1)                           # Compute the new value of objF.
        while(y1>y0):                           # Check that the function value has decreased.
            if(np.allclose(x0,x1,eps)):         # Stopping condition.
                return x0
            step=step/2.0                       # Reduce the step size.
            x1   =x0-step*g0                    # Try again.
            y1,_=objF(x1)

        x0,y0=lineSearch(objF,x0,y0,x1,y1,params)
```

$\# \arg\min_{\lambda} \lambda \mathbf{x}_0 + (1-\lambda)\mathbf{x}_1$

```
    return x0
```

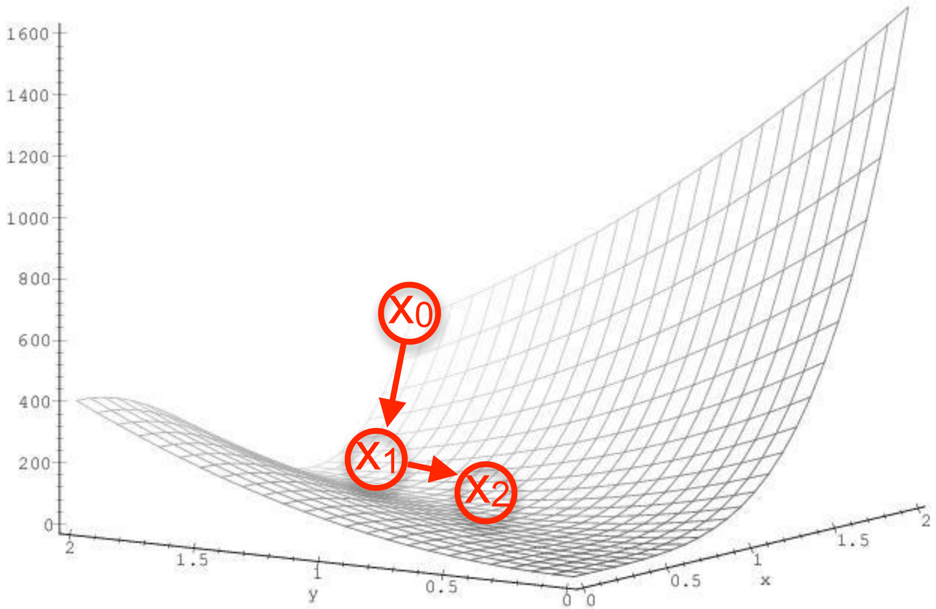


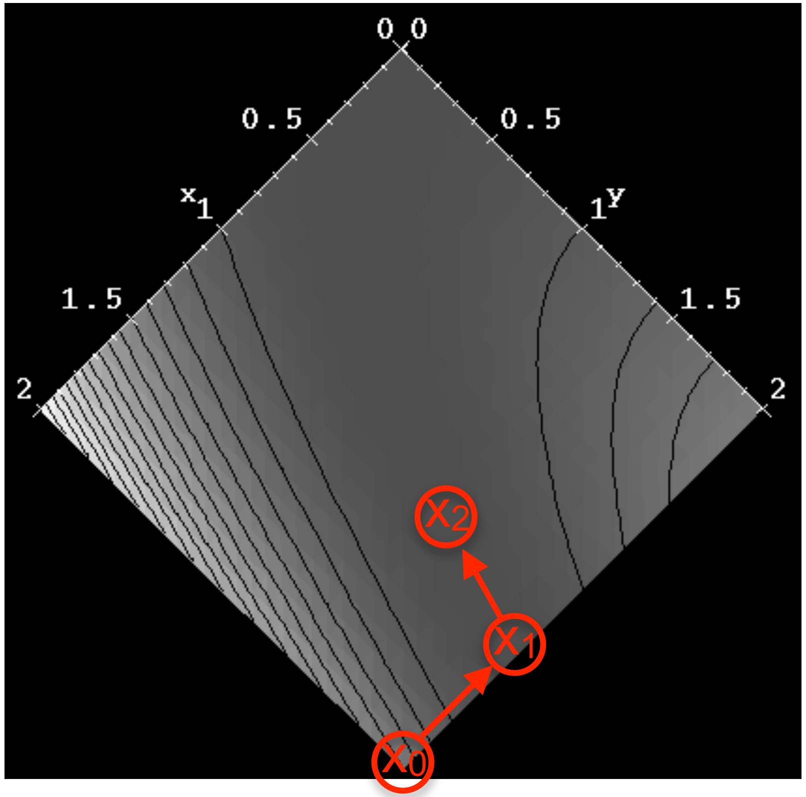
Local Minima



The result depends critically on the starting point and is very likely to be closest local minimum, which is not usually the global one.

Zig-Zagging towards the Solution



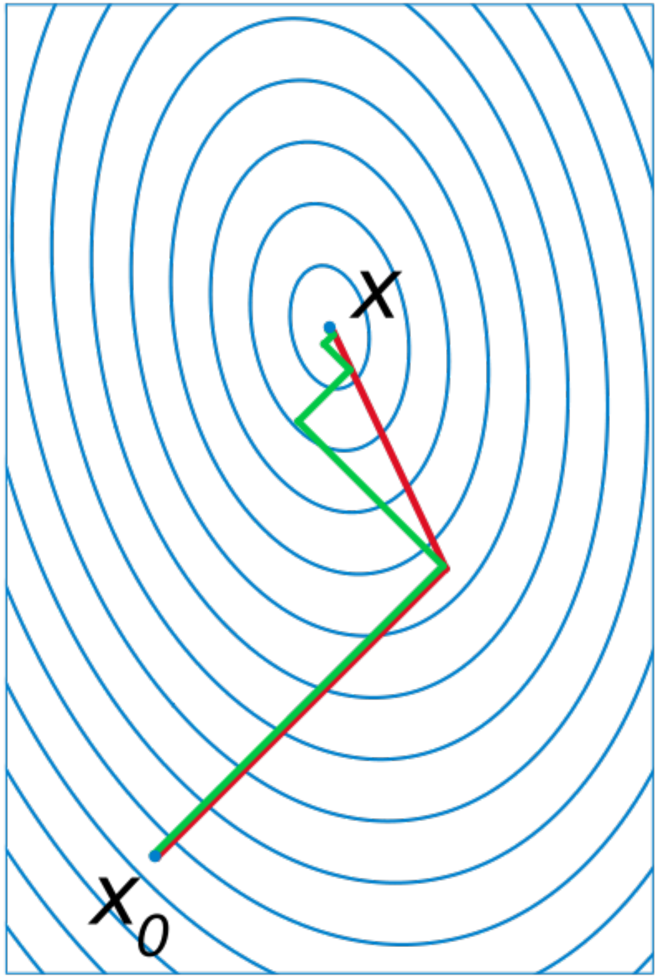


- Successive line searches tend to be perpendicular to each other.
- They would be if we found a true local minimum each time.

Conjugate Gradient

Take the search direction to be a weighted average of the gradient vector and the previous search directions:

1. Start at $\mathbf{x}_0$.
2. $\mathbf{g}_0 = \nabla F(\mathbf{x}_0)$.
3. For k from 0 to $n-1$:
   (a) Find α_k that minimizes $f(\mathbf{x}_k + \alpha_k \mathbf{g}_k)$.
   (b) $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{g}_k$.
   (c) $\beta_k = \frac{\|\nabla f(\mathbf{x}_{k+1})\|^2}{\|\nabla f(\mathbf{x}_k)\|^2}$.
   (d) $\mathbf{g}_{k+1} = -\nabla f(\mathbf{x}_{k+1}) + \beta_k \mathbf{g}_k$.
4. $\mathbf{x}_0 = \mathbf{x}_n$ and go to step 2 until convergence.



—> Faster convergence.

EPFL

Python Implementation

```
def conjugateGrad(objF,x0,nIt=100,eps=1e-10,step=1.0):

    y0,g0=objF(x0)
    h0=-g0                                  # g: Function gradient.
    g0 =h0                                  # h: Conjugate direction.

    for i in range(1,nIt):
        l0=np.linalg.norm(h0)
        if(l0<eps):
            print('Gradient has vanished.')
            break
        x1   =x0+(step/l0)*h0
        y1,_=objF(x1)
        while(y1>y0):                       # Check that the function value has decreased.
            if(np.allclose(x0,x1,eps)):     # Stopping condition.
                return x0
            step=step/2.0
            x1   =x0+(step/np.linalg.norm(h0))*h0
            y1,_=objF(x1,False)

        x1,y1=lineSearch(objF,x0,y0,x1,y1)
        y1,g1=objF(x1)                      # Recompute value and gradient.
        g1=-g1
        h1=g1

        if((i%n)>0):                        # Compute conjugate direction but reset every n iterations.
            gamma=np.dot((g1-g0),g1)/np.dot(g0,g0)  # Modified Polak Ribiere, i.e. only if gamma > 0.
            if(gamma>0):
                h1=g1+gamma*h0

        # Switch variables
        g0=g1
        h0=h1
        x0=x1
        y0=y1
```

Optional

EPFL

In Real Life (1)

```
import scipy

def f(x):
    ......   # return the value of the function.
def g(x):
    ....... # return the gradient of the function.

x0= .... # starting point.
x1= scipy.optimize.fmin_cg(f,x0,fprime=g,epsilon=eps,maxiter=nIt)
```

Optional

In Real Life (2)

Google conjugate gradient python

All Images Videos News Shopping More Settings Tools

About 196.000 results (0,81 seconds)

scipy.sparse.linalg.cg — SciPy v1.2.1 Reference Guide - SciPy.org
https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.linalg.cg.html
scipy.sparse.linalg. cg (A, b, x0=None, tol=1e-05, maxiter=None, M=None, callback=None, atol=None)¶. Use **Conjugate Gradient** iteration to solve Ax = b .

scipy.optimize.fmin_cg — SciPy v0.14.0 Reference Guide
https://docs.scipy.org/doc/scipy-0.14.0/reference/.../scipy.optimize.fmin_cg.html
Minimize a function using a nonlinear **conjugate gradient** algorithm. ... Defaults to None, in which case the gradient is approximated numerically (see epsilon, ...

conjugate gradient method implemented with python · GitHub
https://gist.github.com/sfujiwara/b135e0981d703986b6c2
from scipy.sparse.linalg import cg. ... Solve a linear equation Ax = b with **conjugate gradient** method. ... A: 2d **numpy**.array of positive semi-definite (symmetric) matrix.

The Concept of Conjugate Gradient Descent in Python - Ilya Kuzovkin
ikuz.eu/2015/04/15/the-concept-of-conjugate-gradient-descent-in-python/
Apr 15, 2015 - While reading "An Introduction to the **Conjugate Gradient** Method Without the Agonizing Pain" I decided to boost understand by repeating the ...

Optional

Second Order Methods

Second order Taylor expansion:

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

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

Newton method:

$$\text{Solve } H(\mathbf{x})\mathbf{dx} = -\nabla f(\mathbf{x})$$

$$\Rightarrow \mathbf{dx} = -H(\mathbf{x})^{-1}\nabla f(\mathbf{x})$$

$$\nabla f(\mathbf{x}+\mathbf{dx}) \approx 0$$

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

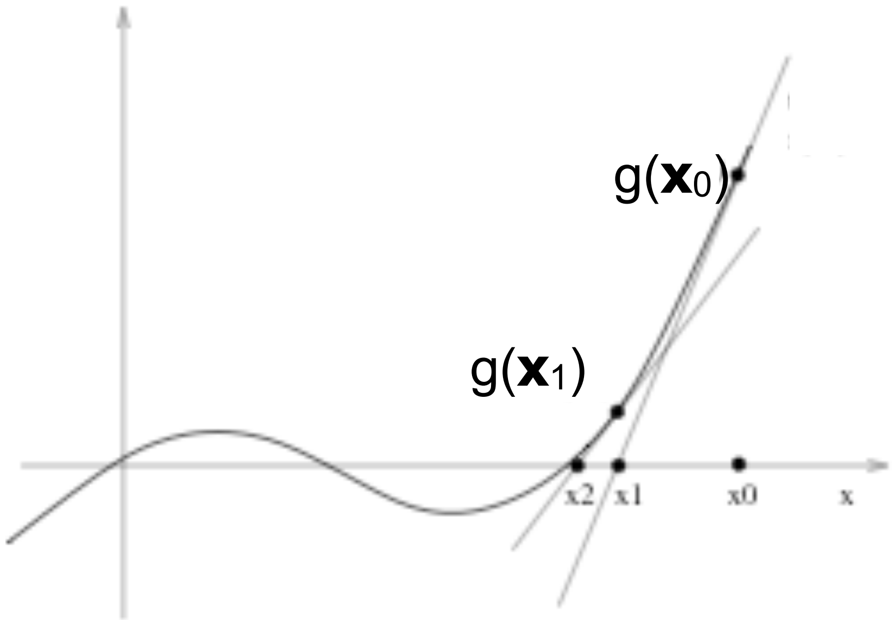
Optional

Triggs et al., Bundle Adjustment, 2000

Newton in 1D

$$0 = g(x + dx) = g(x) + g'(x)dx$$

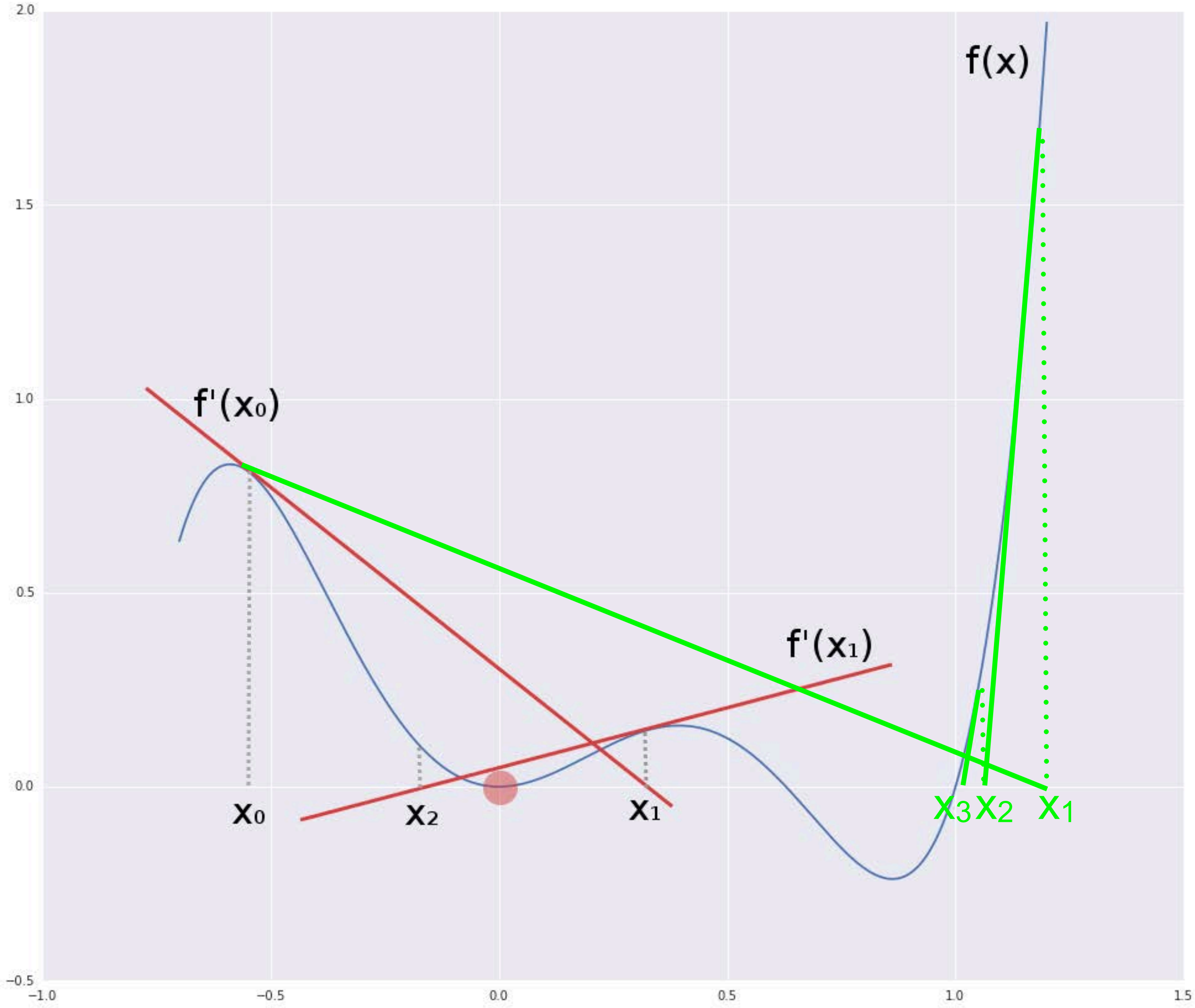
$$\Rightarrow dx = -\frac{g(x)}{g'(x)}$$



$$x \leftarrow x - \frac{g(x)}{g'(x)}$$

Optional

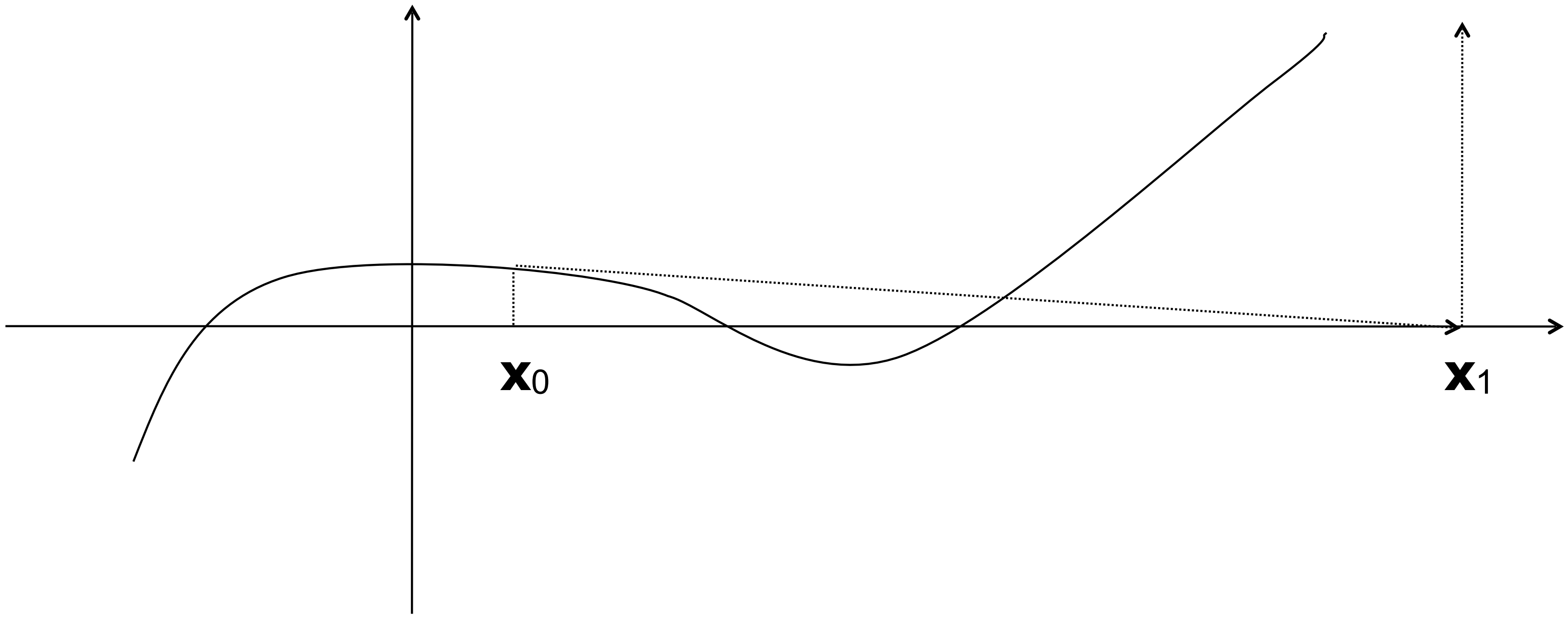
Finding the Root of a Polynomial



$$f(x) = 6x^5 - 5x^4 - 4x^3 + 3x^2$$

- There is more than one root.
- The one you find depends on the starting point.

Potential Instability



- Individual steps can be very large, leading to instability.

Optional

Damped Newton

Second order Taylor expansion:

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

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

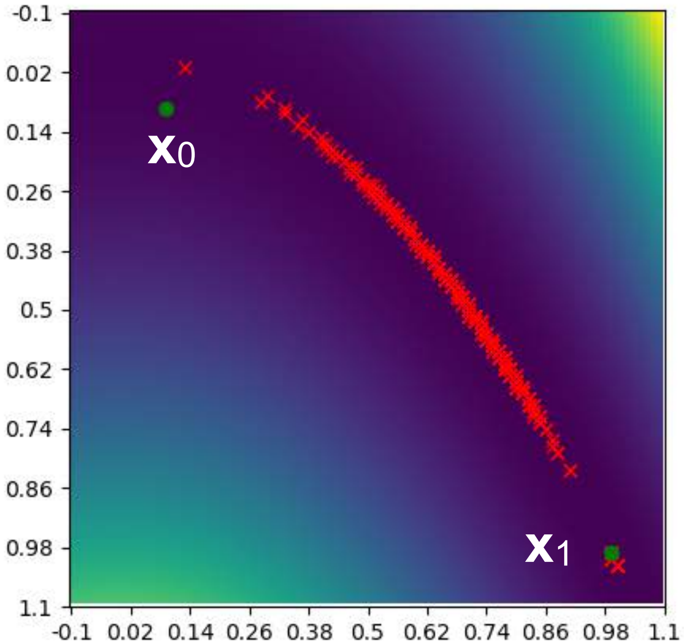
Introduce a damping term:

$$\mathbf{x} \leftarrow \mathbf{x} + \mathbf{dx} \text{ with } \begin{cases} \text{Regular Newton Method: } H(\mathbf{x})\mathbf{dx} = -\nabla f(\mathbf{x}) \\ \text{Damped Newton: } (H(\mathbf{x}) + \lambda \mathbf{I})\mathbf{dx} = -\nabla f(\mathbf{x}) \end{cases}$$

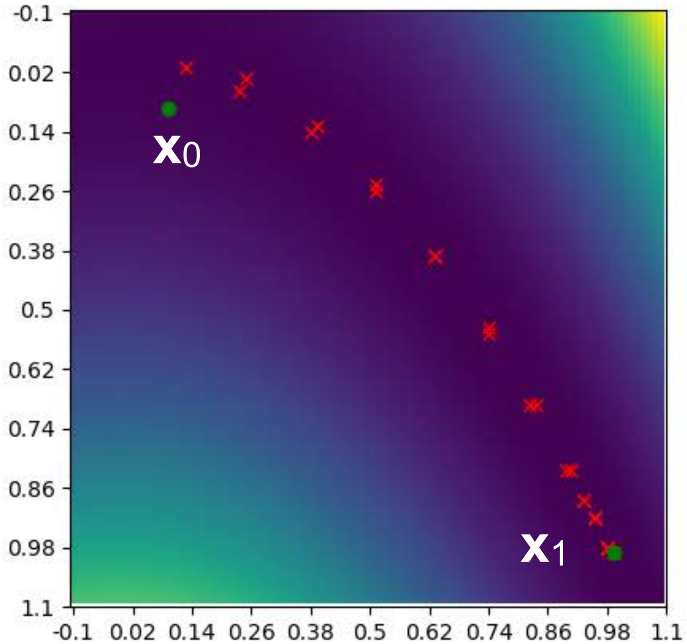
- $\lambda = 0$: Regular Newton
- $\lambda >> 0$: Gradient descent

Optional

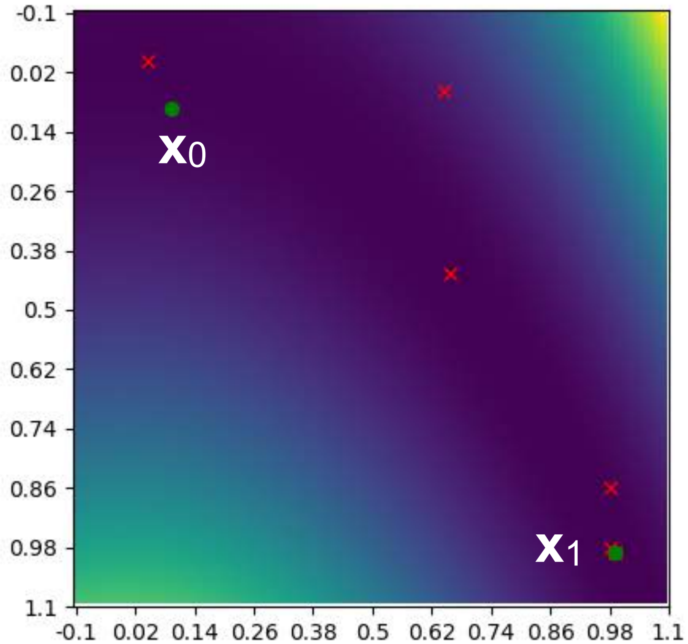
Qualitative Result



Steepest gradient



Conjugate gradient



Damped Newton

Damped Newton converges much faster!

Optional

Python Implementation

```
def dampedNewton(objF,x,nIt=10,lbda=None):

    for i in range(nIt):
        f,g,H=objF(x)                        # Evaluate f, its gradient, and its Hessian.
        x -= linSolve(H,g,lbda=lbda)         # Solve (H + λ I) x = g
    return x


def linSolve(A,b,lbda=None):

    if(lbda is not None):
        A=A+lbda*np.eye(A.shape[0])  # A <— A + λ I
    x=np.linalg.solve(A,b)           # Solve A x = b
    return(x)
```

Optional

Optimization in Short

- Convex functions have a global minimum.
- It can be found using either 1st or 2nd order methods. The latter is usually faster but requires computing second derivatives.
- Non-convex functions can be optimized in a similar manner but this will usually yield a local minimum.