

UNIVERSITÉ
TOULOUSE III
PAUL SABATIER



Introduction to Stochastic Optimization for Statistics

Ho Chi Minh City - March 2018

Xavier Gendre

KHTN



Introduction to Stochastic Optimization for Statistics

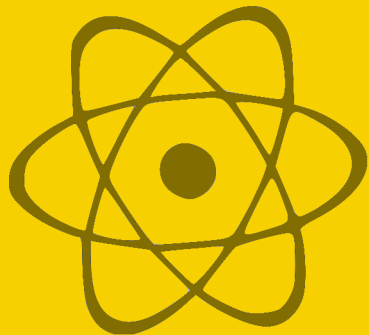
Ho Chi Minh City - March 2018

Xavier Gendre

This work is licensed under a **Creative Commons Attribution - NonCommercial - Share-Alike 4.0 International License**. To obtain a copy of this license, please visit

<https://creativecommons.org/licenses/by-nc-sa/4.0/>



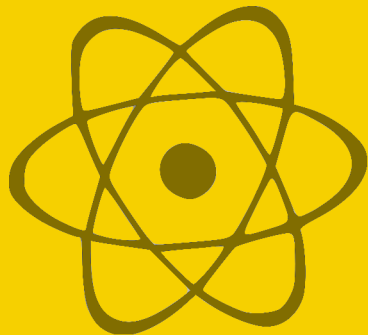


Contents

	Foreword	9
1	Motivations	11
1.1	Statistical framework	11
1.2	Examples	12
1.3	Big data	13
1.4	Optimization	14
2	Mathematical background	17
2.1	Preliminaries	17
2.2	Convexity	18
2.3	Strong convexity	23
3	Stochastic Algorithm	27
3.1	Simple examples	27
3.2	A general definition	29
3.3	Limiting differential equation	29
3.4	Theoretical guarantees	30
4	Non-asymptotic properties	35
4.1	Framework	35
4.2	Rate of projected stochastic gradient descent	35
4.3	Rates of stochastic gradient descent	39

Practicals 1 : Introduction to Python	43
What is Python?	43
About Python version	43
Where to find help?	44
First steps	44
Of Variables and Types	44
Containers	46
Control flows	49
Functions	51
Modules, packages and import	52
Standard library	54
Exceptions	56
A glance at object-oriented programming	57
A recapitulative exercise	58
 Practicals 2 : Python for scientists	 61
NumPy	61
Matplotlib	66
Pandas	70
A recapitulative exercise	74
 Practicals 3 : Adaline	 77
Introduction	77
Generate data	78
Batch approach	79
Gradient Descent	80
Stochastic Gradient Descent	81
 Practicals 4 : Applications in Statistics.....	 83
Mean estimation	83
Regression model	84
Ridge regression model	85
Logistic regression model	86

Practicals 5 : Going further.....	87
Mini-batch approach	87
An example of non-convex optimization	88



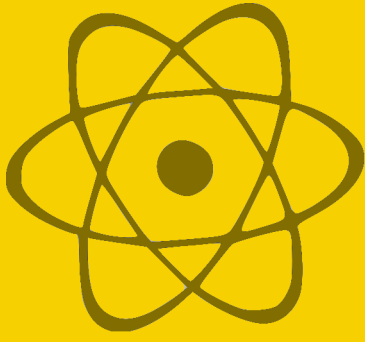
Foreword

These lecture notes are related to the course “*Introduction to Stochastic Optimization for Statistics*” taught at the **Ho Chi Minh City University of Science** from March 12, 2018 to March 21, 2018. I warmly thank **Professor Đặng Đức Trọng** and **Jade Thị Mộng Ngọc Nguyễn** for their help and their sympathy. I would also like to thank **Sébastien Gadat** for the conversations about stochastic algorithms and for his lecture notes which greatly helped me to prepare this course.

The main references used to write this document are:

- Francis Bach and Éric Moulines, *Non-asymptotic analysis of stochastic approximation algorithms for machine learning*. Advances in Neural Information Processing Systems (NIPS), 2011.
- Vivek S. Borkar, *Stochastic approximation: A dynamical systems viewpoint*. Cambridge University Press, 2008.
- Sébastien Gadat, *Stochastic optimization algorithms*. Lecture notes, 2017.
- Sébastien Gadat and Fabien Panloup, *Optimal non-asymptotic bound of the Ruppert-Polyak averaging without strong convexity*. ArXiv:1709.03342, 2017.
- Yurii Nesterov, *Introductory lectures on convex optimization: A basic course*. Springer Science & Business Media, 2013.

The present document and the data sets for the practical sessions are available on the **my web page**. For any request or comment, please contact me at **xavier.gendre@math.univ-toulouse.fr**.



I — Motivations

1.1 Statistical framework

From a global perspective, **machine learning** aims to provide efficient algorithms to estimate an unknown relationship between an observed variable $X \in \mathcal{X}$ and a variable $Y \in \mathcal{Y}$ to be predicted from a data set $\{(X_1, Y_1), \dots, (X_n, Y_n)\}$. To this end, a common approach consists in considering the **joint distribution** P of (X, Y) to deal with the **conditional distribution** of Y with respect to X through the unknown function $\Phi : \mathcal{X} \rightarrow \mathcal{Y}$ defined by the following **conditional expectation**,

$$\forall x \in \mathcal{X}, \Phi(x) = \mathbb{E}[Y \mid X = x] = \int_{\mathcal{Y}} y \, dP(x, y).$$

From a statistical point of view, such a general problem is too difficult and we need to restrict the relationship model to give a satisfactory answer. In practice, this means that we only consider a **parameterized collection** of joint distributions P_{θ} for (X, Y) where θ belongs to some parameter set Θ . For each $\theta \in \Theta$, we have at our disposal a function $\Phi_{\theta} : \mathcal{X} \rightarrow \mathcal{Y}$ defined as above. Thus, the object of interest is now an unknown parameter $\theta^* \in \Theta$ that minimizes the **risk** measurement

$$\theta^* \in \operatorname{argmin}_{\theta \in \Theta} \mathbb{E}[\ell(Y, \Phi_{\theta}(X))]$$

where $\ell : \mathcal{Y} \times \mathcal{Y} \rightarrow \mathbb{R}_+$ is a **loss function**, which intuitively measures the similarity between its arguments. Of course, θ^* is not available to the statistician and we have to estimate it from the data set. A way for that consists in considering a minimizer $\hat{\theta} \in \Theta$ of the **empirical risk**

$$\hat{\theta} \in \operatorname{argmin}_{\theta \in \Theta} \frac{1}{n} \sum_{k=1}^n \ell(Y_k, \Phi_{\theta}(X_k)).$$

Note that, as soon as the collection $\{\Phi_{\theta}\}_{\theta \in \Theta}$ is well defined, such an estimation procedure can always be considered without any assumption on the initial probability space or whatever other theoretical consideration.

1.2 Examples

Linear regression Let us consider the case of $\mathcal{X} = \mathbb{R}^d$ and $\mathcal{Y} = \mathbb{R}$. The simplest model for the relationship between the vector X and the real Y is a linear combination of the components of X , namely

$$\forall \theta \in \mathbb{R}^d, \forall x \in \mathbb{R}^d, \Phi_\theta(x) = \theta^\top x.$$

The **quadratic loss function** is defined by

$$\forall y, y' \in \mathbb{R}, \ell(y, y') = (y - y')^2.$$

Thus, we can consider the **ordinary least squares estimator** $\hat{\theta} \in \mathbb{R}^d$ given by

$$\hat{\theta} \in \operatorname{argmin}_{\theta \in \mathbb{R}^d} \frac{1}{n} \sum_{k=1}^n \left(Y_k - \theta^\top X_k \right)^2.$$

The function to minimize here is **smooth** and **convex** with respect to θ and this ensures the existence of $\hat{\theta}$ (such considerations will be at the heart of the following chapters). Denoting by $\mathbf{Y} = (Y_1, \dots, Y_n)^\top \in \mathbb{R}^n$ and by \mathbf{X} the matrix of size $n \times d$ whose lines are given by the vectors $X_1, \dots, X_n \in \mathbb{R}^d$, if $\mathbf{X}^\top \mathbf{X}$ is invertible, then we know that

$$\hat{\theta} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{Y}.$$

Ridge regression To provide an explicit expression for the ordinary least squares estimator, we assumed that the matrix $\mathbf{X}^\top \mathbf{X}$ was invertible. Such an assumption is not always satisfied in practice and $\mathbf{X}^\top \mathbf{X}$ can admit zero as an eigenvalue. In such a case, the function to minimize is not **strongly convex** and $\hat{\theta}$ can no longer be written in the previous form, although its definition as an empirical risk minimizer remains valid. The idea of the **ridge regression model** is then to consider the **Tikhonov regularization** of the problem by adding a positive part on the diagonal of $\mathbf{X}^\top \mathbf{X}$ to force it to be invertible. Thus, for $\lambda > 0$, we define

$$\hat{\theta}_{\text{ridge}}(\lambda) = (\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{Id}_p)^{-1} \mathbf{X}^\top \mathbf{Y}.$$

This is straightforward to see that $\hat{\theta}_{\text{ridge}}(\lambda)$ is a minimizer of the **regularized empirical risk**,

$$\hat{\theta}_{\text{ridge}}(\lambda) \in \operatorname{argmin}_{\theta \in \mathbb{R}^d} \frac{1}{n} \sum_{k=1}^n \left(Y_k - \theta^\top X_k \right)^2 + \lambda \theta^\top \theta.$$

The regularizer term $\lambda \theta^\top \theta$ acts here like a **Lagrange multiplier** and this problem is equivalent to minimize the empirical risk given by the quadratic loss function with the constraint $\theta^\top \theta \leq r_\lambda^2$, i.e. for θ in a centered ball $\mathcal{B}(r_\lambda) \subset \mathbb{R}^d$ of radius $r_\lambda > 0$,

$$\hat{\theta}_{\text{ridge}}(\lambda) \in \operatorname{argmin}_{\theta \in \mathcal{B}(r_\lambda)} \frac{1}{n} \sum_{k=1}^n \left(Y_k - \theta^\top X_k \right)^2.$$

Thus, we get back to the same problem of minimization as in the case of the linear regression but we now seek the solution only in a **convex subset** $\mathcal{B}(r_\lambda)$ of \mathbb{R}^d .

Logistic regression If $\mathcal{X} = \mathbb{R}^d$ and $\mathcal{Y} = \{0, 1\}$, the statistical problem stated in Section 1.1 is referred as a **supervised classification problem**. We observe a vector X and we want to predict the expected value of Y among $\{0, 1\}$. The conditional distribution of Y with respect to X is nothing else than a Bernoulli distribution that we can parameterize by some $\theta \in \Theta$,

$$\forall \theta \in \Theta, \forall x \in \mathbb{R}^d, \Phi_\theta(x) = \mathbb{P}_\theta(Y = 1 \mid X = x) = p(x, \theta).$$

The **logistic regression model** consists in considering that the logarithm of the ratio between $p(x, \theta)$ and $1 - p(x, \theta)$ can be represented by a linear combination of the components of X , namely

$$\log \left(\frac{p(x, \theta)}{1 - p(x, \theta)} \right) = \theta^\top x.$$

In other words, $\Theta = \mathbb{R}^d$ and it leads us to

$$\forall \theta \in \mathbb{R}^d, \forall x \in \mathbb{R}^d, \Phi_\theta(x) = \frac{\exp(\theta^\top x)}{1 + \exp(\theta^\top x)}.$$

The predicted value of Y for a given $X \in \mathbb{R}^d$ is then provided by $\mathbf{1}_{\Phi_\theta(X) \geq 1/2}$. Ideally, we should use the “true” loss function given by

$$\forall y, y' \in \{0, 1\}, \ell(y, y') = |y - y'|$$

but such a choice leads to a **non-convex** function to minimize with respect to $\theta \in \mathbb{R}^d$. To obtain an efficient algorithm to solve logistic regression problem, we will tend to consider the loss function given by the **opposite of the log-likelihood function**, *i.e.* for any $\theta \in \mathbb{R}^d$, $x \in \mathbb{R}^d$ and $y \in \{0, 1\}$,

$$\begin{aligned} \ell(y, \Phi_\theta(x)) &= -y \log(p(x, \theta)) - (1 - y) \log(1 - p(x, \theta)) \\ &= \log \left(1 + \exp \left(-(2y - 1) \theta^\top x \right) \right) \end{aligned}$$

which is **smooth** and **convex** with respect to θ . Thus, we define the estimator $\hat{\theta}$ as any minimizer of the empirical risk,

$$\hat{\theta} \in \operatorname{argmin}_{\theta \in \mathbb{R}^d} \frac{1}{n} \sum_{k=1}^n \log \left(1 + \exp \left(-(2Y_k - 1) \theta^\top X_k \right) \right).$$

Unfortunately, there is no explicit solution for logistic regression but the goal of the following chapters is to introduce algorithms aimed to solve such a problem in practice.

1.3 Big data

Nowadays, many domains (computer vision, bioinformatics, ...) see the amount and the complexity of the data to be processed greatly increase in various ways (volume, speed, ...). Such a phenomenon is known as **big data** and, in the above examples, it means that we have to deal with potentially (very) large n and p . From a practical point of view, handling such a huge amount or flow of data raises new challenges in statistics. Indeed, defining $\hat{\theta}$ as the minimizer of an empirical risk is always possible in theory but computing it in practice can be

more difficult if the size of the data set exceeds the whole memory of the computer or if the flow of data is faster than the speed of the processing unit.

Statistical procedures that need to handle the whole data set to provide an estimator are referred as **batch methods**. Under the practical constraints evoked above, such approaches often become infeasible even if we can properly write the function to be minimized in order to define the estimator. Alternative strategies consist in handling the data one by one, **updating an estimator through a recursion rule** to approach a solution of the minimization problem. Such approaches are known as **online strategies** and are the main topic of this course. The key tools for studying these methods are provided by **convex analysis** and will be the subject of the next chapter.

1.4 Optimization

As illustrated by the examples developed in Section 1.2, machine learning problems can often be stated as the search of a point $\theta^* \in \mathbb{R}^d$ defined as a minimizer of a function $f : \mathbb{R}^d \rightarrow \mathbb{R}$,

$$\theta^* \in \underset{\theta \in \Theta}{\operatorname{argmin}} f(\theta).$$

Such minimization (or maximization, equivalently) problem is referred as an **optimization problem**. There are various variants of this kind of problem:

- unconstrained problem: $\Theta = \mathbb{R}^p$,
- constrained problem: $\Theta \subsetneq \mathbb{R}^p$,
- smooth problem: f satisfies some regularity conditions,
- quadratic problem: f is quadratic with respect to θ ,
- ...

In general, such problems arise when there is no explicit formula for the minimizer θ^* of f . Thus, to approximate a true solution θ^* , it is natural to allow a certain degree $\varepsilon > 0$ of accuracy and to look for a **ε -solution** θ_ε^* that satisfies

$$|f(\theta^*) - f(\theta_\varepsilon^*)| \leq \varepsilon.$$

The efficiency of a method to compute a ε -solution θ_ε^* is quantified by its **numerical cost** that obviously depends on ε . The algorithms that we have in mind here to obtain a ε -solution for an optimization problem are **iterative**, *i.e.* they compute the ε -solution recursively by updating the previous state with a new data at each step. Then, the numerical cost of a method is closely related to the number of iterations needed to obtain the ε -solution.

To illustrate these considerations about the accuracy and the number of iterations, let us consider $L > 0$ and a function $f : [0, 1]^d \rightarrow \mathbb{R}$ that satisfies the quite weak assumption,

$$\forall \theta, \theta' \in [0, 1]^d, |f(\theta) - f(\theta')| \leq L \|\theta - \theta'\|_\infty \quad (1.1)$$

where we have set

$$\forall \theta = (\theta_1, \dots, \theta_d)^\top \in \mathbb{R}^d, \|\theta\|_\infty = \max_{k \in \{1, \dots, d\}} |\theta_k|.$$

In particular, such an assumption implies that f is a continuous function on the compact set $[0, 1]^d$. Then, the following result gives a very pessimistic bound on the numerical cost to find a ε -solution for the minimization of f .

Theorem 1.1. *Let $0 < \varepsilon \leq L/2$, if $f : [0, 1]^d \rightarrow \mathbb{R}$ satisfies (1.1), then a ε -solution for the minimization of f can be found in*

$$\left(2 + \left\lfloor \frac{L}{2\varepsilon} \right\rfloor\right)^d$$

operations where, for any $x \in \mathbb{R}$, $\lfloor x \rfloor$ is the greatest integer less than or equal to x .

Proof. Let $0 < \delta \leq 1$, we consider a δ -grid of $[0, 1]^d$ defined by

$$\mathcal{G}_d = \left\{ (k_1\delta, \dots, k_d\delta)^\top \text{ with } k_1, \dots, k_d \in \{0, \dots, \lfloor \delta^{-1} \rfloor\} \cup \left\{ \delta^{-1} - \frac{1}{2} \right\} \right\}.$$

By construction, the grid \mathcal{G}_d contains $(2 + \lfloor \delta^{-1} \rfloor)^d$ distinct points. Let us denote by θ^* a minimizer of f in $[0, 1]^d$ and by $\tilde{\theta}^* \in \mathcal{G}_d$ the closest point to θ^* in the grid. By definition, we know that

$$\|\tilde{\theta}^* - \theta^*\|_\infty \leq \frac{\delta}{2}.$$

We can compute $f(\theta_g)$ for all $\theta_g \in \mathcal{G}_d$ and thus find the minimizer $\theta_g^* \in \mathcal{G}_d$ of f on the grid,

$$0 \leq f(\theta_g^*) - f(\theta^*) \leq f(\tilde{\theta}^*) - f(\theta^*).$$

Assumption (1.1) leads to

$$f(\tilde{\theta}^*) - f(\theta^*) \leq L\|\tilde{\theta}^* - \theta^*\|_\infty \leq \frac{L\delta}{2}.$$

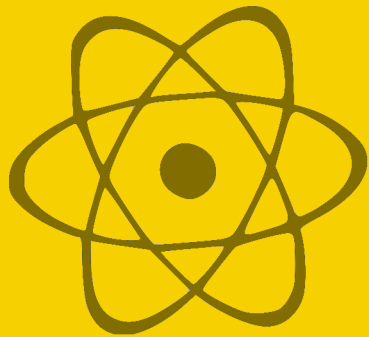
Hence, if we take $\delta = 2\varepsilon/L$, then θ_g^* is a ε -solution that has been found in $(2 + \lfloor L/(2\varepsilon) \rfloor)^d$ computations of f . \square

We can reasonably think that an exhaustive search in a grid as we did in the proof is not an optimal method to obtain a ε -solution. Nevertheless, **with no more assumption on f** , it can be proved that a method to find a ε -solution has to do **at least**

$$\left\lfloor \frac{L}{2\varepsilon} \right\rfloor^d \tag{1.2}$$

operations. Such a result is beyond the scope of this lecture but it shows that our naive approach almost reaches the optimal numerical cost to get a ε -solution for our minimization problem.

Let us conclude with a small calculation. If we want to obtain a minimizer of f in the hypercube of dimension $d = 10$ with $L = 2$ and an accuracy $\varepsilon = 10^{-2}$, then we need at least 10^{20} operations. Even if each operation is done in 10^{-9} second (*i.e.* a frequency of 1 GHz), it will take 3171 years to complete! Of course, such a procedure is unrealistic and we will have to consider other assumptions about the function f to obtain efficient algorithms.



2 — Mathematical Background

2.1 Preliminaries

The results presented in this course can be (almost) all extended to real-valued functions defined on Hilbert spaces. However, for the sake of simplicity, we will restrict ourselves to the case of \mathbb{R}^d in the following. In order to maintain a Hilbert terminology, we will still use the following standard notations. Let d be some positive integer, for any $\theta, \theta' \in \mathbb{R}^d$, we define the **scalar product** of θ and θ' by

$$\langle \theta, \theta' \rangle = \theta^\top \theta'$$

and the associated **norm** as

$$\|\theta\| = \sqrt{\langle \theta, \theta \rangle}.$$

We now introduce the first important notion that was briefly considered in the first chapter.

Definition 2.1. Let $L > 0$, a function $f : \mathbb{R}^{d_1} \rightarrow \mathbb{R}^{d_2}$ is **L -Lipschitz** if

$$\forall \theta_1, \theta_2 \in \mathbb{R}^{d_1}, \|f(\theta_1) - f(\theta_2)\| \leq L \|\theta_1 - \theta_2\|.$$

An immediate consequence of the definition is the **continuity** of f on \mathbb{R}^{d_1} .

Being Lipschitz will be a useful property of the functions in the sequel, but we will often need more regularity.

Definition 2.2. A function $f : \mathbb{R}^d \rightarrow \mathbb{R}$ is **differentiable** if, for any $\theta \in \mathbb{R}^d$, there exists a vector $\nabla f(\theta) \in \mathbb{R}^d$ such that

$$\lim_{\varepsilon \rightarrow 0} \frac{|f(\theta + \varepsilon) - f(\theta) - \langle \nabla f(\theta), \varepsilon \rangle|}{\|\varepsilon\|} = 0.$$

The vector $\nabla f(\theta)$ is called the **gradient** of f at point θ .

Combining the above definitions leads us to the common smoothness definition considered in this lecture.

Definition 2.3. Let $L > 0$, a differentiable function $f : \mathbb{R}^d \rightarrow \mathbb{R}$ is **L -smooth** if the gradient ∇f is L -Lipschitz,

$$\forall \theta_1, \theta_2 \in \mathbb{R}^d, \|\nabla f(\theta_1) - \nabla f(\theta_2)\| \leq L\|\theta_1 - \theta_2\|.$$

This definition implies the continuity of ∇f on \mathbb{R}^d . Such a function is said to be **continuously differentiable**.

The smooth functions have nice properties like the variations mainly controlled by the squared norm.

Proposition 2.1. Let $f : \mathbb{R}^d \rightarrow \mathbb{R}$ be a L -smooth function, then

$$\forall \theta_1, \theta_2 \in \mathbb{R}^d, |f(\theta_1) - f(\theta_2) - \langle \nabla f(\theta_2), \theta_1 - \theta_2 \rangle| \leq \frac{L}{2} \|\theta_1 - \theta_2\|^2.$$

Proof. A first-order Taylor expansion leads to

$$\begin{aligned} f(\theta_1) &= f(\theta_2) + \int_0^1 \langle \nabla f(\theta_2 + t(\theta_1 - \theta_2)), \theta_1 - \theta_2 \rangle dt \\ &= f(\theta_2) + \langle \nabla f(\theta_2), \theta_1 - \theta_2 \rangle + \int_0^1 \langle \nabla f(\theta_2 + t(\theta_1 - \theta_2)) - \nabla f(\theta_2), \theta_1 - \theta_2 \rangle dt. \end{aligned}$$

Cauchy–Schwarz inequality and L -smoothness give

$$\begin{aligned} |f(\theta_1) - f(\theta_2) - \langle \nabla f(\theta_2), \theta_1 - \theta_2 \rangle| &\leq \int_0^1 \|\nabla f(\theta_2 + t(\theta_1 - \theta_2)) - \nabla f(\theta_2)\| \times \|\theta_1 - \theta_2\| dt \\ &\leq L\|\theta_1 - \theta_2\|^2 \int_0^1 t dt \\ &= \frac{L}{2} \|\theta_1 - \theta_2\|^2. \end{aligned}$$

□

Finally, we introduce the main object of interest of the current chapter. How to construct such sequences and what are their properties will be the topic of the next sections.

Definition 2.4. Let $f : \mathbb{R}^d \rightarrow \mathbb{R}$ be a differentiable function. For $\theta_0 \in \mathbb{R}^d$ and a sequence of positive **step sizes** $\{\gamma_n\}_{n \geq 0}$, the **gradient descent** is the sequence $\{\theta_n\}_{n \geq 0}$ defined by

$$\forall n \geq 1, \theta_n = \theta_{n-1} - \gamma_n \nabla f(\theta_{n-1}).$$

2.2 Convexity

As it was quickly mentioned in the first chapter, the only regularity of a function is not sufficient to provide efficient algorithms for optimization problems. The notion of convexity will be the key to propose better approaches.

Definition 2.5. A function $f : \mathbb{R}^d \rightarrow \mathbb{R}$ is **convex** if

$$\forall \theta_1, \theta_2 \in \mathbb{R}^d, \forall \lambda \in [0, 1], f(\lambda \theta_1 + (1 - \lambda) \theta_2) \leq \lambda f(\theta_1) + (1 - \lambda) f(\theta_2).$$

We will not often use this definition in the following. Indeed, the functions we will handle will usually be regular enough and the following proposition offers an alternative definition of convexity in this case.

Proposition 2.2. Let $f : \mathbb{R}^d \rightarrow \mathbb{R}$ be a differentiable function, then f is convex if and only if

$$\forall \theta_1, \theta_2 \in \mathbb{R}^d, f(\theta_1) \geq f(\theta_2) + \langle \nabla f(\theta_2), \theta_1 - \theta_2 \rangle. \quad (2.1)$$

Proof. Let $\theta_1, \theta_2 \in \mathbb{R}^d$, we define the function h by

$$\forall \lambda \in [0, 1], h(\lambda) = f(\lambda \theta_1 + (1 - \lambda) \theta_2) - \lambda f(\theta_1) - (1 - \lambda) f(\theta_2).$$

Because f is differentiable, the same goes for h and we get

$$h'(\lambda) = \langle \nabla f(\lambda \theta_1 + (1 - \lambda) \theta_2), \theta_1 - \theta_2 \rangle - f(\theta_1) + f(\theta_2).$$

(\Rightarrow) If f is convex, then $h(\lambda) \leq 0$ for any $\lambda \in [0, 1]$. Since $h(0) = 0$, we deduce

$$h'(0) = \lim_{\lambda \rightarrow 0} \frac{h(\lambda)}{\lambda} \leq 0$$

which corresponds to (2.1).

(\Leftarrow) If (2.1) holds, then we have

$$f(\theta_1) \geq f(\lambda \theta_1 + (1 - \lambda) \theta_2) + (1 - \lambda) \langle \nabla f(\lambda \theta_1 + (1 - \lambda) \theta_2), \theta_1 - \theta_2 \rangle$$

and

$$f(\theta_2) \geq f(\lambda \theta_1 + (1 - \lambda) \theta_2) - \lambda \langle \nabla f(\lambda \theta_1 + (1 - \lambda) \theta_2), \theta_1 - \theta_2 \rangle.$$

From these two inequalities, we deduce

$$\begin{aligned} \lambda f(\theta_1) + (1 - \lambda) f(\theta_2) &\geq (\lambda + (1 - \lambda)) f(\lambda \theta_1 + (1 - \lambda) \theta_2) \\ &\quad + \lambda (1 - \lambda) \langle \nabla f(\lambda \theta_1 + (1 - \lambda) \theta_2), \theta_1 - \theta_2 \rangle \\ &\quad - \lambda (1 - \lambda) \langle \nabla f(\lambda \theta_1 + (1 - \lambda) \theta_2), \theta_1 - \theta_2 \rangle \\ &= f(\lambda \theta_1 + (1 - \lambda) \theta_2). \end{aligned}$$

Finally, f is convex. □

Proposition 2.2 allows us to establish two important facts that will be particularly useful in the future. First, a critical point of a convex function is a global minimum.

Proposition 2.3. Let $f : \mathbb{R}^d \rightarrow \mathbb{R}$ be a differentiable and convex function. If a point $\theta^* \in \mathbb{R}^d$ is such that $\nabla f(\theta^*) = 0$, then $f(\theta^*)$ is the global minimum of f on \mathbb{R}^d .

Proof. This is a direct consequence of (2.1). Indeed, for any $\theta \in \mathbb{R}^d$, we obtain

$$f(\theta) \geq f(\theta^*) + \langle \nabla f(\theta^*), \theta - \theta^* \rangle = f(\theta^*).$$

In other words, f reaches its global minimum at point θ^* . \square

The second useful consequence of Proposition 2.2 is the following lemma which we will invoke several times in the following.

Lemma 2.1. *Let $f : \mathbb{R}^d \rightarrow \mathbb{R}$ be a L -smooth and convex function. Then, for any $\theta_1, \theta_2 \in \mathbb{R}^d$, we have*

$$f(\theta_2) - f(\theta_1) \leq \langle \nabla f(\theta_2), \theta_2 - \theta_1 \rangle - \frac{1}{2L} \|\nabla f(\theta_2) - \nabla f(\theta_1)\|^2.$$

Proof. Let $\theta, \theta' \in \mathbb{R}^d$, Propositions 2.1 and 2.2 together imply that f satisfies the following inequalities

$$0 \leq f(\theta) - f(\theta') - \langle \nabla f(\theta'), \theta - \theta' \rangle \leq \frac{L}{2} \|\theta - \theta'\|^2.$$

Let $\theta_1, \theta_2 \in \mathbb{R}^d$, we apply the lower bound to $\theta = \theta_1 + L^{-1}(\nabla f(\theta_2) - \nabla f(\theta_1))$ and $\theta' = \theta_2$ to get

$$f(\theta_2) - f(\theta) \leq \langle \nabla f(\theta_2), \theta_2 - \theta \rangle.$$

Moreover, we apply the upper bound to the same θ and $\theta' = \theta_1$ to obtain

$$f(\theta) - f(\theta_1) \leq \langle \nabla f(\theta_1), \theta - \theta_1 \rangle + \frac{L}{2} \|\theta - \theta_1\|^2.$$

Adding these inequalities leads to the announced result,

$$\begin{aligned} f(\theta_2) - f(\theta_1) &= (f(\theta_2) - f(\theta)) + (f(\theta) - f(\theta_1)) \\ &\leq \langle \nabla f(\theta_2), \theta_2 - \theta \rangle + \langle \nabla f(\theta_1), \theta - \theta_1 \rangle + \frac{L}{2} \|\theta - \theta_1\|^2 \\ &= \langle \nabla f(\theta_2), \theta_2 - \theta_1 \rangle + \langle \nabla f(\theta_1) - \nabla f(\theta_2), \theta - \theta_1 \rangle + \frac{L}{2} \|\theta - \theta_1\|^2 \\ &= \langle \nabla f(\theta_2), \theta_2 - \theta_1 \rangle - \frac{1}{L} \|\nabla f(\theta_2) - \nabla f(\theta_1)\|^2 + \frac{1}{2L} \|\nabla f(\theta_2) - \nabla f(\theta_1)\|^2 \\ &= \langle \nabla f(\theta_2), \theta_2 - \theta_1 \rangle - \frac{1}{2L} \|\nabla f(\theta_2) - \nabla f(\theta_1)\|^2. \end{aligned}$$

\square

A wide variety of convex functions appear more or less naturally in machine learning problems. Here are some examples we have already encountered (proofs are left as exercises):

- the scalar product with a constant vector $x \in \mathbb{R}^d$ is convex,

$$\forall \theta \in \mathbb{R}^d, f(\theta) = \langle x, \theta \rangle,$$

- the squared Euclidean distance from a vector $y \in \mathbb{R}^d$ is convex,

$$\forall \theta \in \mathbb{R}^d, f(\theta) = \|y - \theta\|^2,$$

- the empirical risk given in the context of linear regression with the quadratic loss function is convex, *i.e.* for any matrix X of size $n \times d$ and any vector $y \in \mathbb{R}^n$,

$$\forall \theta \in \mathbb{R}^d, f(\theta) = \|y - X\theta\|^2,$$

- the loss function given by the opposite of the log-likelihood function defined in the example of logistic regression is convex, *i.e.* for any $x \in \mathbb{R}^d$ and $y \in \{0, 1\}$,

$$\forall \theta \in \mathbb{R}^d, f(\theta) = \log(1 + \exp(-(2y - 1)\langle x, \theta \rangle)).$$

Hint: this last example is not as straightforward as the others. First, prove that, for any $a \in \mathbb{R}$, the function $t \in \mathbb{R} \mapsto \log(1 + \exp(at))$ is convex. Then, deduce the result by composition with affine function.

When a convex function is also smooth, then a gradient descent with constant step size is an efficient algorithm to solve a minimization problem as shown by the following result.

Theorem 2.1. *Let $f : \mathbb{R}^d \rightarrow \mathbb{R}$ be a L -smooth and convex function that reaches its global minimum at point $\theta^* \in \mathbb{R}^d$. Then, the gradient descent defined with a constant step size $\gamma_n = L^{-1}$ satisfies*

$$\forall n \geq 1, f(\theta_n) - f(\theta^*) \leq \frac{2L\|\theta_0 - \theta^*\|^2}{n}$$

Proof. Let $n \geq 1$, by definition of the gradient descent, we get

$$\begin{aligned} \|\theta_n - \theta^*\|^2 &= \|(\theta_{n-1} - \theta^*) - L^{-1}\nabla f(\theta_{n-1})\|^2 \\ &= \|\theta_{n-1} - \theta^*\|^2 - \frac{2}{L}\langle \nabla f(\theta_{n-1}), \theta_{n-1} - \theta^* \rangle + \frac{1}{L^2}\|\nabla f(\theta_{n-1})\|^2. \end{aligned}$$

Applying two times Lemma 2.1, we obtain

$$\begin{aligned} -\langle \nabla f(\theta_{n-1}), \theta_{n-1} - \theta^* \rangle &\leq f(\theta^*) - f(\theta_{n-1}) - \frac{1}{2L}\|\nabla f(\theta_{n-1}) - \nabla f(\theta^*)\|^2 \\ &\leq \langle \nabla f(\theta^*), \theta^* - \theta_{n-1} \rangle - \frac{2}{2L}\|\nabla f(\theta_{n-1}) - \nabla f(\theta^*)\|^2 \\ &= -\frac{1}{L}\|\nabla f(\theta_{n-1})\|^2. \end{aligned}$$

Thus,

$$\|\theta_n - \theta^*\|^2 \leq \|\theta_{n-1} - \theta^*\|^2 - \frac{1}{L^2}\|\nabla f(\theta_{n-1})\|^2.$$

In particular, this inequality implies that $\|\theta_n - \theta^*\|^2$ is decreasing with respect to n and

$$\forall k \geq 0, \|\theta_k - \theta^*\|^2 \leq \|\theta_0 - \theta^*\|^2.$$

For any $k \geq 0$, we set $\Delta_k = f(\theta_k) - f(\theta^*)$. Proposition 2.2, Cauchy–Schwarz inequality and above result give

$$\Delta_k \leq \langle \nabla f(\theta_k), \theta_k - \theta^* \rangle \leq \|\nabla f(\theta_k)\| \times \|\theta_k - \theta^*\|. \quad (2.2)$$

Moreover, Proposition 2.1 leads to

$$\begin{aligned} f(\theta_n) &\leq f(\theta_{n-1}) + \langle \nabla f(\theta_{n-1}), \theta_n - \theta_{n-1} \rangle + \frac{L}{2} \|\theta_n - \theta_{n-1}\|^2 \\ &= f(\theta_{n-1}) - \frac{1}{2L} \|\nabla f(\theta_{n-1})\|^2. \end{aligned}$$

Then, with (2.2), we deduce

$$\Delta_n \leq \Delta_{n-1} - \frac{1}{2L} \|\nabla f(\theta_{n-1})\|^2 \leq \Delta_{n-1} - \frac{\Delta_{n-1}^2}{2L \|\theta_0 - \theta^*\|^2}.$$

Let us denote $\omega = 2L \|\theta_0 - \theta^*\|^2$. Dividing by $\Delta_n \Delta_{n-1}$, the above recursion can be rearranged as follows

$$\frac{1}{\Delta_n} - \frac{1}{\Delta_{n-1}} \geq \frac{\Delta_{n-1}}{\omega \Delta_n} \geq \frac{1}{\omega}.$$

Summing this inequality between 1 and n gives

$$\frac{1}{\Delta_n} - \frac{1}{\Delta_0} = \sum_{k=1}^n \frac{1}{\Delta_k} - \frac{1}{\Delta_{k-1}} \geq \frac{n}{\omega}.$$

Finally, we obtain the announced result since

$$\Delta_n \leq \frac{\Delta_0 \omega}{\omega + n \Delta_0} \leq \frac{\omega}{n}.$$

□

This result gives a polynomial convergence rate of the gradient descent for the minimization problem. In particular, for any $\varepsilon > 0$, the quantity θ_n is a ε -solution as soon as

$$n \geq \frac{2L \|\theta_0 - \theta^*\|^2}{\varepsilon}.$$

This lower bound has to be compared to (1.2). Up to the multiplicative factor $\|\theta_0 - \theta^*\|^2$, which can be seen as a constant price to pay in the numerical cost of the method, the minimum number of operations no longer depends on the dimension d . Of course, the dimension factor is hidden in the initial error term but this is definitely weaker than a power of the accuracy ε . With the help of the convexity assumption, for $L = 2$ and $\varepsilon = 10^{-2}$, the order of magnitude of the lower bound is now only 400 operations. Strengthening the convexity hypothesis, we will see that this result can be further improved.

2.3 Strong convexity

We have seen in the previous section that convexity is the key to obtain efficient algorithms for optimization problems. We now propose to consider a reinforced version of this notion to get better properties.

Definition 2.6. Let $\mu > 0$, a function $f : \mathbb{R}^d \rightarrow \mathbb{R}$ is μ -strongly convex if the function ϕ defined by

$$\forall \theta \in \mathbb{R}^d, \phi(\theta) = f(\theta) - \frac{\mu}{2} \|\theta\|^2 \quad (2.3)$$

is convex.

As for simple convexity, we have an alternative definition of strong convexity for regular functions.

Proposition 2.4. Let $f : \mathbb{R}^d \rightarrow \mathbb{R}$ be a differentiable function. Then, f is μ -strongly convex if and only if

$$\forall \theta_1, \theta_2 \in \mathbb{R}^d, f(\theta_1) \geq f(\theta_2) + \langle \nabla f(\theta_2), \theta_1 - \theta_2 \rangle + \frac{\mu}{2} \|\theta_1 - \theta_2\|^2.$$

Proof. Let ϕ be defined as in (2.3). Because f is differentiable, the same goes for ϕ and we have $\nabla \phi(\theta) = \nabla f(\theta) - \mu \theta$. According to Proposition 2.2, we know

$$\begin{aligned} \phi \text{ is convex} &\iff \forall \theta_1, \theta_2 \in \mathbb{R}^d, \phi(\theta_1) \geq \phi(\theta_2) + \langle \nabla \phi(\theta_2), \theta_1 - \theta_2 \rangle \\ &\iff \forall \theta_1, \theta_2 \in \mathbb{R}^d, f(\theta_1) \geq f(\theta_2) + \langle \nabla f(\theta_2), \theta_1 - \theta_2 \rangle + \frac{\mu}{2} \|\theta_1 - \theta_2\|^2. \end{aligned}$$

□

Regularity and strong convexity together lead to important properties like a gradient lower bounded by the squared norm.

Proposition 2.5. Let $f : \mathbb{R}^d \rightarrow \mathbb{R}$ be a differentiable and μ -strongly convex function. Then,

$$\forall \theta_1, \theta_2 \in \mathbb{R}^d, \langle \nabla f(\theta_1) - \nabla f(\theta_2), \theta_1 - \theta_2 \rangle \geq \mu \|\theta_1 - \theta_2\|^2.$$

Proof. This statement is a direct consequence of the Proposition 2.4. Indeed, for any $\theta_1, \theta_2 \in \mathbb{R}^d$, we get

$$f(\theta_1) - f(\theta_2) - \langle \nabla f(\theta_2), \theta_1 - \theta_2 \rangle \geq \frac{\mu}{2} \|\theta_1 - \theta_2\|^2$$

and

$$f(\theta_2) - f(\theta_1) + \langle \nabla f(\theta_1), \theta_1 - \theta_2 \rangle \geq \frac{\mu}{2} \|\theta_1 - \theta_2\|^2.$$

Summing these two inequalities leads to the announced result.

□

The inequality given by Proposition 2.5 can be strengthened under additional regularity assumption as in the next useful lemma.

Lemma 2.2. *Let $f : \mathbb{R}^d \rightarrow \mathbb{R}$ be a L -smooth and μ -strongly convex function. Then, for any $\theta_1, \theta_2 \in \mathbb{R}^d$,*

$$\langle \nabla f(\theta_1) - \nabla f(\theta_2), \theta_1 - \theta_2 \rangle \geq \frac{\mu L}{L + \mu} \|\theta_1 - \theta_2\|^2 + \frac{1}{L + \mu} \|\nabla f(\theta_1) - \nabla f(\theta_2)\|^2.$$

Proof. First, we notice that we necessarily have $\mu \leq L$. Indeed, for any $\theta_1, \theta_2 \in \mathbb{R}^d$, Proposition 2.5 and Cauchy–Schwarz inequality imply

$$\begin{aligned} \mu \|\theta_1 - \theta_2\|^2 &\leq \langle \nabla f(\theta_1) - \nabla f(\theta_2), \theta_1 - \theta_2 \rangle \\ &\leq \|\nabla f(\theta_1) - \nabla f(\theta_2)\| \times \|\theta_1 - \theta_2\| \\ &\leq L \|\theta_1 - \theta_2\|^2. \end{aligned}$$

If $\mu = L$, the announced result is a consequence of Proposition 2.5,

$$\begin{aligned} \frac{\mu}{2} \|\theta_1 - \theta_2\|^2 + \frac{1}{2\mu} \|\nabla f(\theta_1) - \nabla f(\theta_2)\|^2 &\leq \frac{\mu}{2} \|\theta_1 - \theta_2\|^2 + \frac{\mu}{2} \|\theta_1 - \theta_2\|^2 \\ &\leq \langle \nabla f(\theta_1) - \nabla f(\theta_2), \theta_1 - \theta_2 \rangle. \end{aligned}$$

We can now assume that $\mu < L$ and we consider the function ϕ as defined in (2.3). This function is differentiable with, for any $\theta \in \mathbb{R}^d$, $\nabla \phi(\theta) = \nabla f(\theta) - \mu \theta$. Thus, Proposition 2.1 leads to

$$\begin{aligned} \phi(\theta) - \phi(\theta_1) - \langle \nabla \phi(\theta_1), \theta - \theta_1 \rangle &= f(\theta) - f(\theta_1) - \langle \nabla f(\theta_1), \theta - \theta_1 \rangle - \frac{\mu}{2} \|\theta_1 - \theta\|^2 \\ &\leq \frac{L - \mu}{2} \|\theta_1 - \theta\|^2 \end{aligned} \quad (2.4)$$

which can also be written

$$\phi(\theta) - \phi(\theta_1) \leq \langle \nabla \phi(\theta_1), \theta - \theta_1 \rangle + \frac{L - \mu}{2} \|\theta_1 - \theta\|^2.$$

Moreover, by definition, ϕ is convex and Proposition 2.2 gives

$$\phi(\theta_2) - \phi(\theta) \leq \langle \nabla \phi(\theta_2), \theta_2 - \theta \rangle.$$

Let us take $\theta = \theta_1 + (L - \mu)^{-1}(\nabla \phi(\theta_2) - \nabla \phi(\theta_1))$, we sum the above inequalities to get

$$\begin{aligned} \phi(\theta_2) - \phi(\theta_1) &= (\phi(\theta_2) - \phi(\theta)) + (\phi(\theta) - \phi(\theta_1)) \\ &\leq \langle \nabla \phi(\theta_2), \theta_2 - \theta \rangle + \langle \nabla \phi(\theta_1), \theta - \theta_1 \rangle + \frac{L - \mu}{2} \|\theta_1 - \theta\|^2 \\ &= \langle \nabla \phi(\theta_2), \theta_2 - \theta_1 \rangle - \frac{1}{2(L - \mu)} \|\nabla \phi(\theta_2) - \nabla \phi(\theta_1)\|^2 \end{aligned}$$

Reorganizing the terms and arguing as for (2.4), we obtain

$$\frac{1}{2(L - \mu)} \|\nabla \phi(\theta_2) - \nabla \phi(\theta_1)\|^2 \leq \phi(\theta_1) - \phi(\theta_2) + \langle \nabla \phi(\theta_2), \theta_2 - \theta_1 \rangle \leq \frac{L - \mu}{2} \|\theta_1 - \theta_2\|^2.$$

In other words, ϕ is $(L - \mu)$ -smooth. Then, we can apply Lemma 2.1,

$$\frac{1}{2(L - \mu)} \|\nabla\phi(\theta_2) - \nabla\phi(\theta_1)\|^2 \leq \phi(\theta_1) - \phi(\theta_2) + \langle \nabla\phi(\theta_2), \theta_2 - \theta_1 \rangle.$$

Symmetrizing this inequality with respect to θ_1 and θ_2 and summing up lead to

$$\frac{1}{L - \mu} \|\nabla\phi(\theta_2) - \nabla\phi(\theta_1)\|^2 \leq \langle \nabla\phi(\theta_1) - \nabla\phi(\theta_2), \theta_1 - \theta_2 \rangle.$$

Using $\nabla\phi(\theta) = \nabla f(\theta) - \mu\theta$, we get

$$\begin{aligned} \langle \nabla f(\theta_1) - \nabla f(\theta_2), \theta_1 - \theta_2 \rangle &\geq \mu \|\theta_1 - \theta_2\|^2 + \frac{1}{L - \mu} \|(\nabla f(\theta_1) - \nabla f(\theta_2)) - \mu(\theta_1 - \theta_2)\|^2 \\ &= \frac{\mu L}{L - \mu} \|\theta_1 - \theta_2\|^2 + \frac{1}{L - \mu} \|\nabla f(\theta_1) - \nabla f(\theta_2)\|^2 \\ &\quad - \frac{2\mu}{L - \mu} \langle \nabla f(\theta_1) - \nabla f(\theta_2), \theta_1 - \theta_2 \rangle. \end{aligned}$$

This last inequality corresponds to the announced result since

$$1 + \frac{2\mu}{L - \mu} = \frac{L + \mu}{L - \mu}.$$

□

Finally, we can state the convergence of the gradient descent for the minimization of a smooth and strongly convex function.

Theorem 2.2. *Let $f : \mathbb{R}^d \rightarrow \mathbb{R}$ be a L -smooth and μ -strongly convex function that reaches its global minimum at point $\theta^* \in \mathbb{R}^d$. Then, the gradient descent defined with a constant step size $\gamma_n = 2/(L + \mu)$ satisfies*

$$\forall n \geq 1, f(\theta_n) - f(\theta^*) \leq \frac{L \|\theta_0 - \theta^*\|^2}{2} \times \exp\left(-\frac{4n}{\kappa + 1}\right)$$

where we have set $\kappa = L/\mu$.

Proof. Let $n \geq 1$, since $\nabla f(\theta^*) = 0$, Proposition 2.1 implies

$$f(\theta_n) - f(\theta^*) \leq \frac{L}{2} \|\theta_n - \theta^*\|^2.$$

Moreover, by definition of the gradient descent and Lemma 2.2,

$$\begin{aligned}
\|\theta_n - \theta^*\|^2 &= \left\| (\theta_{n-1} - \theta^*) - \frac{2}{L+\mu} \nabla f(\theta_{n-1}) \right\|^2 \\
&= \|\theta_{n-1} - \theta^*\|^2 + \frac{4}{(L+\mu)^2} \|\nabla f(\theta_{n-1})\|^2 - \frac{4}{L+\mu} \langle \nabla f(\theta_{n-1}), \theta_{n-1} - \theta^* \rangle \\
&\leq \|\theta_{n-1} - \theta^*\|^2 + \frac{4}{(L+\mu)^2} \|\nabla f(\theta_{n-1})\|^2 \\
&\quad - \frac{4}{L+\mu} \left\{ \frac{\mu L}{L+\mu} \|\theta_{n-1} - \theta^*\|^2 + \frac{1}{L+\mu} \|\nabla f(\theta_{n-1})\|^2 \right\} \\
&= \left(1 - \frac{4\mu L}{(L+\mu)^2} \right) \|\theta_{n-1} - \theta^*\|^2 \\
&= \left(1 - \frac{2}{\kappa+1} \right)^2 \|\theta_{n-1} - \theta^*\|^2.
\end{aligned}$$

Iterating this bound leads to

$$\|\theta_n - \theta^*\|^2 \leq \left(1 - \frac{2}{\kappa+1} \right)^{2n} \|\theta_0 - \theta^*\|^2 \leq \exp\left(-\frac{4n}{\kappa+1}\right) \|\theta_0 - \theta^*\|^2$$

since $1 - x \leq e^{-x}$ for any x . □

The exponential convergence rate obtained in this theorem greatly improves the convergence rate we got with simple convexity. The initial error is still present through the multiplicative factor $\|\theta_0 - \theta^*\|^2$ but the accuracy is now driven by an exponential decay with respect to the number of iterations n . That is the main consequence of strong convexity. Let $\varepsilon > 0$, the minimum number of iterations needed to get a ε -solution is now lower bounded by

$$n \geq \frac{\kappa+1}{4} \log\left(\frac{L\|\theta_0 - \theta^*\|^2}{2\varepsilon}\right).$$

For $L = 2$, $\mu = 2/3$ and $\varepsilon = 10^{-2}$, the order of magnitude of this minimum number of operations to obtain a ε -solution to the minimization problem is now only 5 operations.

There exist lower bound results for the problem of minimization of a (strongly) convex function. Actually, the convergence rates we obtained in Theorems 2.1 and 2.2 are not optimal in the sense that some better algorithms exist under the same set of hypotheses. In particular, the algorithm called **Nesterov accelerated gradient descent** outperforms the standard gradient descent. The convergence rates for this algorithm are given by:

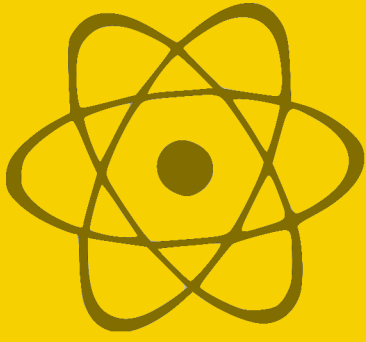
- for L -smooth and convex function,

$$f(\theta_n) - f(\theta^*) \leq \frac{2L\|\theta_0 - \theta^*\|^2}{n^2},$$

- for L -smooth and μ -strongly convex function,

$$f(\theta_n) - f(\theta^*) \leq \frac{(L+\mu)\|\theta_0 - \theta^*\|^2}{2} \times \exp\left(-\frac{n}{\sqrt{\kappa}}\right).$$

Introducing this algorithm is beyond the scope of these lecture notes, but it can be proved that these convergence rates are optimal for these two classes of functions.



3 — Stochastic Algorithm

3.1 Simple examples

Let $\{X_n\}_{n \geq 1}$ be a sequence of independent and identically distributed variables assumed to be integrable with $m = \mathbb{E}[X_1]$. For any $n \geq 1$, we can estimate m with the n first observations X_1, \dots, X_n through the empirical mean

$$\bar{X}_n = \frac{1}{n} \sum_{k=1}^n X_k.$$

This common estimation procedure can be seen as a **stochastic algorithm** as follows. Let us set $\bar{X}_0 = 0$ and, for any $n \geq 1$, rewrite \bar{X}_n with respect to \bar{X}_{n-1} ,

$$\begin{aligned} \bar{X}_n &= \frac{n-1}{n} \bar{X}_{n-1} + \frac{1}{n} X_n \\ &= \bar{X}_{n-1} + \frac{1}{n} (X_n - \bar{X}_{n-1}) \\ &= \bar{X}_{n-1} - \frac{1}{n} (\bar{X}_{n-1} - m) + \frac{1}{n} (X_n - m) \end{aligned}$$

Denoting the **step size** by $\gamma_n = n^{-1}$, if we consider the function f defined by

$$\forall x \in \mathbb{R}, f(x) = \frac{(x - m)^2}{2},$$

then the recursion may be simply written as

$$\bar{X}_n = \bar{X}_{n-1} - \gamma_n f'(\bar{X}_{n-1}) + \gamma_n M_n$$

where we have set $M_n = X_n - m$. Thus, \bar{X}_n can be seen as \bar{X}_{n-1} with the addition of a **gradient descent term** related to the function f and a **centered term** which will be considered as a **martingale increment** in the sequel, *i.e.* the sequence $\{M_n\}_{n \geq 1}$ satisfy

$$\forall n \geq 1, \mathbb{E}[M_n | X_1, \dots, X_{n-1}] = 0.$$

To go further, assuming that $\mathbb{E}[X_1^2] < +\infty$, we can consider the simultaneous estimation of the mean m and the variance σ^2 of X_1 . To this end, we introduce the usual estimator of the variance based on the n first observations,

$$\forall n \geq 1, S_n = \frac{1}{n} \sum_{k=1}^n (X_k - \bar{X}_n)^2.$$

As above, setting $S_0 = 0$, we can deduce a recursion between S_n and S_{n-1} , for any $n \geq 1$,

$$\begin{aligned} S_n &= \frac{1}{n} \sum_{k=1}^n \left(X_k - \frac{n-1}{n} \bar{X}_{n-1} - \frac{1}{n} X_n \right)^2 \\ &= \frac{1}{n} \sum_{k=1}^n \left((X_k - \bar{X}_{n-1}) - \frac{1}{n} (X_n - \bar{X}_{n-1}) \right)^2 \\ &= \frac{1}{n} \sum_{k=1}^n (X_k - \bar{X}_{n-1})^2 + \frac{1}{n^2} (X_n - \bar{X}_{n-1})^2 - \frac{2}{n} (X_n - \bar{X}_{n-1})(\bar{X}_n - \bar{X}_{n-1}) \\ &= \frac{n-1}{n} S_{n-1} + \frac{n-1}{n^2} (X_n - \bar{X}_{n-1})^2 \\ &= S_{n-1} - \frac{1}{n} (S_{n-1} - \sigma^2) + \frac{1}{n} \left\{ \frac{n-1}{n} (X_n - \bar{X}_{n-1})^2 - \sigma^2 \right\} \\ &= S_{n-1} - \gamma_n (S_{n-1} - \sigma^2) + \gamma_n M'_n + \gamma_n R_n \end{aligned}$$

where we have set the martingale increments

$$M'_n = \frac{n-1}{n} ((X_n - m)^2 - \sigma^2 - 2(X_n - m)(\bar{X}_{n-1} - m))$$

and the remainder terms

$$R_n = \frac{n-1}{n} (\bar{X}_{n-1} - m)^2 - \frac{\sigma^2}{n}.$$

Thus, we now consider the function f defined on $\mathbb{R} \times \mathbb{R}_+^*$ by

$$f(x, s) = \frac{1}{2} ((x - m)^2 + (s - \sigma^2)^2).$$

This function admits the following gradient

$$\nabla f(x, s) = \begin{pmatrix} x - m \\ s - \sigma^2 \end{pmatrix}$$

and we get the two-dimensional recursion, for any $n \geq 1$,

$$\begin{pmatrix} \bar{X}_n \\ S_n \end{pmatrix} = \begin{pmatrix} \bar{X}_{n-1} \\ S_{n-1} \end{pmatrix} - \gamma_n \nabla f(\bar{X}_{n-1}, S_{n-1}) + \gamma_n \begin{pmatrix} M_n \\ M'_n \end{pmatrix} + \gamma_n \begin{pmatrix} 0 \\ R_n \end{pmatrix}.$$

Again, we can verify that this step is nothing else than the addition of a **gradient descent term** related to f , a two-dimensional **martingale increment** and a **remainder term**.

According to the strong law of large numbers, we know that

$$\bar{X}_n \xrightarrow[n \rightarrow \infty]{a.s.} m \quad \text{and} \quad S_n \xrightarrow[n \rightarrow \infty]{a.s.} \sigma^2.$$

In other words, the sequence $\{(\bar{X}_n, S_n)\}_{n \geq 0}$ converges almost surely towards a zero of ∇f which is also the global minimizer of f . The two algorithms above are classic examples of stochastic algorithms aimed to the minimization of a differentiable function f through the search for a zero of ∇f .

3.2 A general definition

To discuss algorithms such as those introduced in the first section, we define a general framework through the following definition.

Definition 3.1. Let $h : \mathbb{R}^d \rightarrow \mathbb{R}^d$, $\{\mathcal{F}_n\}_{n \geq 0}$ be an increasing filtration and $\{\gamma_n\}_{n \geq 1}$ be a sequence of positive step sizes. A **stochastic algorithm** is a sequence of random vectors $\{\theta_n\}_{n \geq 0}$ starting from a \mathcal{F}_0 -measurable vector $\theta_0 \in \mathbb{R}^d$ and such that

$$\forall n \geq 1, \theta_n = \theta_{n-1} - \gamma_n h(\theta_{n-1}) + \gamma_n M_n + \gamma_n R_n$$

where

- $\{M_n\}_{n \geq 1}$ is a sequence of square-integrable martingale increments with respect to the filtration $\{\mathcal{F}_n\}_{n \geq 0}$,
- $\{R_n\}_{n \geq 1}$ is a predictable sequence of square-integrable remainder terms with respect to the filtration $\{\mathcal{F}_n\}_{n \geq 0}$.

An immediate consequence of this definition of a stochastic algorithm $\{\theta_n\}_{n \geq 0}$ is that, for any $n \geq 0$, θ_n is \mathcal{F}_n -measurable. As we will see in the sequel, the step sizes have to be large enough to allow the gradient descent to explore the function but not too much to keep the algorithm under control. In practice, these step sizes are commonly assumed to satisfy

$$\sum_{n \geq 1} \gamma_n = +\infty \quad \text{and} \quad \sum_{n \geq 1} \gamma_n^2 < +\infty. \quad (3.1)$$

The remainder terms R_n that appear in the definition should be considered as **negligible perturbations** and will **often be omitted** in the following for the sake of simplicity. For example, in the variance estimation example above, we had

$$\mathbb{E}[R_n] = 0 \quad \text{and} \quad \mathbb{E}[R_n^2] \leq \frac{2\mathbb{E}[(X_1 - m)^4]}{n^2}.$$

3.3 Limiting differential equation

Let us consider a stochastic algorithm $\{\theta_n\}_{n \geq 0}$ as given by Definition 3.1 with a zero remainder term, *i.e.* satisfying the following recursion,

$$\forall n \geq 1, \theta_n = \theta_{n-1} - \gamma_n h(\theta_{n-1}) + \gamma_n M_n.$$

We can easily associate this algorithm to a **continuous interpolated trajectory** $\{\theta(t)\}_{t \in \mathbb{R}_+}$ as follows. We define time instants $t_0 = 0$ and

$$\forall n \geq 1, t_n = \sum_{k=1}^n \gamma_k.$$

Let $n \geq 1$, if $t \in [t_{n-1}, t_n]$, then $\theta(t)$ is given by the linear interpolation between θ_{n-1} and θ_n , namely

$$\forall t \in [t_{n-1}, t_n], \theta(t) = \theta_{n-1} + (\theta_n - \theta_{n-1}) \times \frac{t - t_{n-1}}{t_n - t_{n-1}}. \quad (3.2)$$

We see that, in order to properly define $\theta(t)$ for any $t \in \mathbb{R}_+$, we need to ensure $t_n \rightarrow +\infty$ which corresponds to the **divergence assumption** in (3.1),

$$\sum_{n \geq 1} \gamma_n = +\infty.$$

For a small enough step size γ_n , we can roughly write

$$\theta_n = \theta(t_n) = \theta(t_{n-1} + \gamma_n) \simeq \theta(t_{n-1}) + \gamma_n \theta'(t_{n-1}) \simeq \theta_{n-1} - \gamma_n h(\theta_{n-1})$$

where the first approximation comes from a first-order Taylor expansion and the second from the omission of the martingale increment which is centered. In other words, with (3.2), we recognize an **explicit Euler method** for which we expect to asymptotically track the differential equation

$$\frac{d\theta(t)}{dt} = h(\theta(t)).$$

In the above approximations, we have neglected the martingale increments but, after $n \geq 1$ steps, they correspond to the quantity

$$\zeta_n = \sum_{k=1}^n \gamma_k M_k.$$

By hypothesis, this is straightforward to see that $\{\zeta_n\}_{n \geq 0}$ with $\zeta_0 = 0$ is a square-integrable martingale. Furthermore, the martingale increments $\{\zeta_n - \zeta_{n-1}\}_{n \geq 1}$ are such that

$$\forall n \geq 1, \sum_{k=1}^n \mathbb{E} [\|\zeta_n - \zeta_{n-1}\|^2 \mid \mathcal{F}_{n-1}] = \sum_{k=1}^n \gamma_k^2 \mathbb{E} [\|M_n\|^2 \mid \mathcal{F}_{n-1}].$$

From classical martingale theory (see Appendix C of the book of Borkar cited in Foreword), we know that the almost sure convergence of such series would imply the almost sure convergence of ζ_n as $n \rightarrow \infty$. Of course, such a result will be needed in the analysis of the stochastic algorithm $\{\theta_n\}_{n \geq 0}$. Under suitable **bounding assumptions** on the conditional expectation of $\|M_n\|^2$, we see that the step sizes should at least satisfy the second part of (3.1),

$$\sum_{n \geq 1} \gamma_n^2 < +\infty.$$

3.4 Theoretical guarantees

Providing an exhaustive theoretical analysis of stochastic algorithms is beyond the scope of this lecture. In order to introduce important properties in this section, we omit the proofs and we silently avoid developing results that come from martingales theory. For instance, the following result is a consequence of **Robbins-Siegmund's Theorem** that we do not explicitly state.

Theorem 3.1. Let $\{\theta_n\}_{n \geq 0}$ be a stochastic algorithm with step sizes satisfying (3.1). We assume that there exist $C > 0$ and a twice continuously differentiable and L -smooth function $V : \mathbb{R}^d \rightarrow \mathbb{R}$ such that:

- “Drift” assumptions:

1. $\min_{\theta \in \mathbb{R}^d} V(\theta) > 0$,
2. $\lim_{\|\theta\| \rightarrow \infty} V(\theta) = +\infty$,
3. $\forall \theta \in \mathbb{R}^d, \langle \nabla V(\theta), h(\theta) \rangle \geq 0$,
4. $\forall \theta \in \mathbb{R}^d, \|h(\theta)\|^2 + \|\nabla V(\theta)\|^2 \leq C(1 + V(\theta))$,

- “Perturbation” assumptions:

1. the martingale increments $\{M_n\}_{n \geq 1}$ satisfy

$$\forall n \geq 1, \mathbb{E} [\|M_n\|^2 \mid \mathcal{F}_{n-1}] \leq C(1 + V(\theta_{n-1})),$$

2. the remainder terms $\{R_n\}_{n \geq 1}$ satisfy

$$\forall n \geq 1, \mathbb{E} [\|R_n\|^2 \mid \mathcal{F}_{n-1}] \leq C\gamma_n^2(1 + V(\theta_{n-1})).$$

Then, the following properties hold

- $\sup_{n \geq 0} \mathbb{E} [V(\theta_n)] < +\infty$,
- $\sum_{n \geq 1} \gamma_n \langle \nabla V(\theta_{n-1}), h(\theta_{n-1}) \rangle < +\infty, a.s.$,
- $V(\theta_n) \xrightarrow[n \rightarrow \infty]{a.s.} V_\infty \in L^1$,
- $\theta_n - \theta_{n-1} \xrightarrow[n \rightarrow \infty]{a.s. \text{ and } L^2} 0$.

A function V satisfying the hypotheses of the above theorem is called a **Lyapunov function**. Such a function plays a central role in obtaining this kind of result and it is often challenging to find a “good” Lyapunov function. Because the sum $\gamma_1 + \dots + \gamma_n$ tends towards infinity when n grows and, for any $\theta \in \mathbb{R}^d$, we assume $\langle \nabla V(\theta), h(\theta) \rangle \geq 0$, the consequence of

$$\sum_{n \geq 1} \gamma_n \langle \nabla V(\theta_{n-1}), h(\theta_{n-1}) \rangle < +\infty, a.s.,$$

is only

$$\liminf_{n \rightarrow \infty} \langle \nabla V(\theta_n), h(\theta_n) \rangle = 0, a.s.$$

Thus, something more is needed to get the almost sure convergence of $\nabla V(\theta_n)$ to 0.

The next result is due to Robbins and Monro and allow to state the convergence of a stochastic algorithm aimed to the minimization of a given function.

Theorem 3.2. *Under the same assumptions as in Theorem 3.1, we also assume that the function h is continuous over \mathbb{R}^d and*

$$\left\{ \theta \in \mathbb{R}^d \text{ such that } \langle \nabla V(\theta), h(\theta) \rangle = 0 \right\} = \{\theta^*\}.$$

Then,

- θ^* is the unique minimizer of V ,
- $\langle \nabla V(\theta_n), h(\theta_n) \rangle \xrightarrow[n \rightarrow \infty]{a.s.} 0$,
- $\theta_n \xrightarrow[n \rightarrow \infty]{a.s.} \theta^*$.

The Robbins and Monro theorem implies the convergence of the stochastic algorithm to the zero of the function h but does not state any convergence rate. Such results will be developed in the next chapter. Before concluding this general presentation of theoretical properties of stochastic algorithms, we propose to discuss a **central limit theorem** due to Polyak and Juditsky. This result is specific to stochastic algorithms aimed to minimize a strongly convex and smooth enough function f through the search of a zero of $h = \nabla f$.

Theorem 3.3. *Let $f : \mathbb{R}^d \rightarrow \mathbb{R}$ be a μ -strongly convex and twice continuously differentiable function that reaches its global minimum at point $\theta^* \in \mathbb{R}^d$. We consider the stochastic algorithm given by Definition 3.1 with $h = \nabla f$ and no remainder terms. We assume that:*

- f admits a bounded Hessian operator Hf ,
- the step sizes are such that

$$\gamma_n \xrightarrow[n \rightarrow \infty]{} 0 \quad \text{and} \quad \frac{\gamma_n - \gamma_{n+1}}{\gamma_n^2} \xrightarrow[n \rightarrow \infty]{} 0, \quad (3.3)$$

- the conditional covariance matrices converge in probability,

$$\mathbb{E} \left[M_n M_n^\top \mid \mathcal{F}_{n-1} \right] \xrightarrow[n \rightarrow \infty]{\mathbb{P}} \Sigma.$$

Then,

$$\sqrt{n}(\bar{\theta}_n - \theta^*) \xrightarrow[n \rightarrow \infty]{\mathcal{L}} \mathcal{N}(0, \Sigma^*)$$

where we have set

$$\forall n \geq 1, \bar{\theta}_n = \frac{1}{n} \sum_{k=1}^n \theta_{k-1}$$

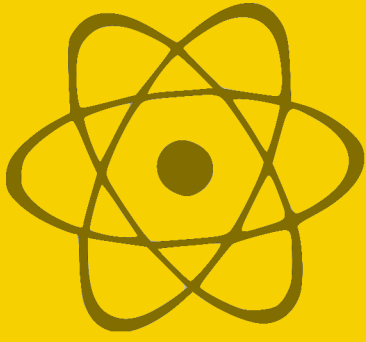
and the covariance matrix is given by

$$\Sigma^* = Hf(\theta^*)^{-1} \Sigma Hf(\theta^*)^{-1}.$$

This results illustrate the **importance of averaging** when working with stochastic algorithms. Indeed, averaging leads to an **asymptotically optimal algorithm** whose rate of convergence is $1/n$, which is **minimax optimal** in the class of strongly convex stochastic minimization problems. Moreover, the asymptotic variance is also optimal because it attains the **Cramer-Rao lower bound**. In practice, usual choices for the step sizes are $\gamma_n = \gamma n^{-\alpha}$ with $\gamma > 0$ and $\alpha \in (0, 1)$ to satisfy (3.3) because

$$\frac{\gamma_n - \gamma_{n+1}}{\gamma_n^2} = \frac{n^\alpha}{\gamma} \left(1 - \left(1 + \frac{1}{n} \right)^{-\alpha} \right) \underset{n \rightarrow \infty}{\sim} \frac{\alpha n^{\alpha-1}}{\gamma}.$$

Finally, note that as soon as the step sizes do not depend on $Hf(\theta^*)$, $\{\bar{\theta}_n\}_{n \geq 1}$ is an **adaptive sequence**.



4 — Non-asymptotic properties

4.1 Framework

Let $\{\mathcal{F}_n\}_{n \geq 0}$ be an increasing filtration and $\{\gamma_n\}_{n \geq 1}$ be a sequence of positive step sizes. Given a differentiable function $f : \mathbb{R}^d \rightarrow \mathbb{R}$ to be minimized, we consider a sequence of differentiable random functions $\{f_n\}_{n \geq 1}$ from \mathbb{R}^d to \mathbb{R} that satisfy that, for any $n \geq 1$ and any $\theta \in \mathbb{R}^d$, the random variable $\nabla f_n(\theta)$ is \mathcal{F}_n -measurable, square-integrable and such that,

$$\mathbb{E}[\nabla f_n(\theta) \mid \mathcal{F}_{n-1}] = \nabla f(\theta), \text{ a.s.}$$

Then, we define a sequence $\{\theta_n\}_{n \geq 0}$ with a \mathcal{F}_0 -measurable variable $\theta_0 \in \mathbb{R}^d$ and the recursion

$$\forall n \geq 1, \theta_n = \theta_{n-1} - \gamma_n \nabla f_n(\theta_{n-1}). \quad (4.1)$$

This is therefore a stochastic algorithm as given by Definition 3.1 with $h = \nabla f$ and no remainder terms. Indeed, an alternative way to write the above recursion is

$$\forall n \geq 1, \theta_n = \theta_{n-1} - \gamma_n \nabla f(\theta_{n-1}) + \gamma_n M_n$$

where $M_n = \nabla f(\theta_{n-1}) - \nabla f_n(\theta_{n-1})$ is a martingale increment by definition. Such a stochastic algorithm is known as **stochastic gradient descent**.

4.2 Rate of projected stochastic gradient descent

Let us assume that $f : \mathbb{R}^d \rightarrow \mathbb{R}$ is a **differentiable and convex** function to minimize. In this section, we assume that the minimization problem can be restricted to a ball of radius r , namely

$$\mathcal{B}_r = \left\{ \theta \in \mathbb{R}^d \text{ such that } \|\theta\| \leq r \right\}.$$

In practice, such an assumption is not necessarily strong because we often have the idea of a (possibly large) radius $r > 0$ such that the minimizer of f is located inside \mathcal{B}_r . To constrain the stochastic gradient descent to remain in \mathcal{B}_r , we consider the **projected stochastic gradient descent** $\{\theta_n\}_{n \geq 0}$ given by a \mathcal{F}_0 -measurable variable $\theta_0 \in \mathcal{B}_r$ and the recursion

$$\forall n \geq 1, \theta_n = \Pi_r(\theta_{n-1} - \gamma_n \nabla f_n(\theta_{n-1})) \quad (4.2)$$

where Π_r is the orthogonal projection onto \mathcal{B}_r .

Theorem 4.1. *Let $r > 0$ and $f : \mathbb{R}^d \rightarrow \mathbb{R}$ be a differentiable and convex function that reaches its global minimum at point $\theta^* \in \mathcal{B}_r$. If the random gradient functions satisfy*

$$\exists B > 0, \forall n \geq 1, \sup_{\theta \in \mathcal{B}_r} \|\nabla f_n(\theta)\| \leq B, \text{ a.s.,}$$

then, the projected stochastic gradient descent $\{\theta_n\}_{n \geq 0}$ defined by (4.2) with

$$\gamma_n = \frac{r\sqrt{2}}{B\sqrt{n}}$$

is such that

$$\forall n \geq 1, \mathbb{E} [f(\bar{\theta}_n) - f(\theta^*)] \leq \frac{2\sqrt{2}Br}{\sqrt{n}}$$

where

$$\forall n \geq 1, \bar{\theta}_n = \frac{1}{n} \sum_{k=1}^n \theta_{k-1}.$$

Proof. Let $n \geq 1$, by definition of the projected stochastic gradient descent, we get

$$\begin{aligned} \|\theta_n - \theta^*\|^2 &= \|\Pi_r(\theta_{n-1} - \gamma_n \nabla f_n(\theta_{n-1})) - \theta^*\|^2 \\ &\leq \|(\theta_{n-1} - \theta^*) - \gamma_n \nabla f_n(\theta_{n-1})\|^2 \\ &= \|\theta_{n-1} - \theta^*\|^2 + \gamma_n^2 \|\nabla f_n(\theta_{n-1})\|^2 - 2\gamma_n \langle \nabla f_n(\theta_{n-1}), \theta_{n-1} - \theta^* \rangle \\ &\leq \|\theta_{n-1} - \theta^*\|^2 + \gamma_n^2 B^2 - 2\gamma_n \langle \nabla f_n(\theta_{n-1}), \theta_{n-1} - \theta^* \rangle. \end{aligned}$$

Taking the expectation with respect to \mathcal{F}_{n-1} and using Proposition 2.2 lead to

$$\begin{aligned} \mathbb{E} [\|\theta_n - \theta^*\|^2 \mid \mathcal{F}_{n-1}] &\leq \|\theta_{n-1} - \theta^*\|^2 + \gamma_n^2 B^2 - 2\gamma_n \langle \nabla f(\theta_{n-1}), \theta_{n-1} - \theta^* \rangle \\ &\leq \|\theta_{n-1} - \theta^*\|^2 + \gamma_n^2 B^2 - 2\gamma_n (f(\theta_{n-1}) - f(\theta^*)). \end{aligned}$$

We now consider the whole expectation to obtain

$$\mathbb{E} [\|\theta_n - \theta^*\|^2] \leq \mathbb{E} [\|\theta_{n-1} - \theta^*\|^2] + \gamma_n^2 B^2 - 2\gamma_n \mathbb{E} [f(\theta_{n-1}) - f(\theta^*)]$$

which is equivalent to

$$\mathbb{E} [f(\theta_{n-1}) - f(\theta^*)] \leq \frac{\gamma_n B^2}{2} + \frac{1}{2\gamma_n} (\mathbb{E} [\|\theta_{n-1} - \theta^*\|^2] - \mathbb{E} [\|\theta_n - \theta^*\|^2]).$$

Summing this inequality from 1 to n and the decrease of $\{\gamma_n\}_{n \geq 1}$ give

$$\begin{aligned}
\sum_{k=1}^n \mathbb{E}[f(\theta_{k-1}) - f(\theta^*)] &\leq \frac{B^2}{2} \sum_{k=1}^n \gamma_k + \frac{1}{2} \sum_{k=1}^n \frac{1}{\gamma_k} (\mathbb{E}[\|\theta_{k-1} - \theta^*\|^2] - \mathbb{E}[\|\theta_k - \theta^*\|^2]) \\
&= \frac{B^2}{2} \sum_{k=1}^n \gamma_k + \frac{\mathbb{E}[\|\theta_0 - \theta^*\|^2]}{2\gamma_1} - \frac{\mathbb{E}[\|\theta_n - \theta^*\|^2]}{2\gamma_n} \\
&\quad + \frac{1}{2} \sum_{k=1}^{n-1} \mathbb{E}[\|\theta_k - \theta^*\|^2] \left(\frac{1}{\gamma_{k+1}} - \frac{1}{\gamma_k} \right) \\
&\leq \frac{B^2}{2} \sum_{k=1}^n \gamma_k + \frac{2r^2}{\gamma_1} + \frac{2r^2}{\gamma_n} - \frac{2r^2}{\gamma_1} \\
&\leq \frac{B^2}{2} \sum_{k=1}^n \gamma_k + \frac{2r^2}{\gamma_n}
\end{aligned}$$

where we used the bound $\|\theta_{k-1} - \theta^*\|^2 \leq 4r^2$. Since $\gamma_n = \frac{r\sqrt{2}}{B\sqrt{n}}$, we deduce from the convexity of f and Jensen's inequality that

$$\begin{aligned}
\mathbb{E}[f(\bar{\theta}_n) - f(\theta^*)] &\leq \frac{1}{n} \sum_{k=1}^n \mathbb{E}[f(\theta_{k-1}) - f(\theta^*)] \\
&\leq \frac{Br}{n\sqrt{2}} \sum_{k=1}^n \frac{1}{\sqrt{k}} + \frac{\sqrt{2}Br}{\sqrt{n}}
\end{aligned}$$

which gives the announced result through the simple upper bound

$$\sum_{k=1}^n \frac{1}{\sqrt{k}} \leq \int_0^n \frac{dx}{\sqrt{x}} = 2\sqrt{n}.$$

□

Note that step sizes here depend on B , which can be problematic in practice since this quantity is generally unknown. This dependency can be avoided but at the cost of more technicality. The convergence rate we obtain is slower than that of the gradient descent for which we get a rate $1/n$. Nevertheless, Nemirovski and Yudin proved that the rate $1/\sqrt{n}$ of the stochastic gradient descent is **minimax optimal** under the assumptions of the theorem. As in the gradient descent case, the following result shows that this rate can be improved with strong convexity.

Theorem 4.2. Let $r > 0$ and $f : \mathbb{R}^d \rightarrow \mathbb{R}$ be a differentiable and μ -strongly convex function that reaches its global minimum at point $\theta^* \in \mathcal{B}_r$. We assume that the random gradient functions satisfy

$$\exists B > 0, \forall n \geq 1, \sup_{\theta \in \mathcal{B}_r} \|\nabla f_n(\theta)\| \leq B, \text{ a.s.}$$

The projected stochastic gradient descent $\{\theta_n\}_{n \geq 0}$ defined by (4.2) is such that

1. if $\gamma_n = \frac{1}{\mu n}$, then

$$\forall n \geq 1, \mathbb{E} [f(\bar{\theta}_n) - f(\theta^*)] \leq \frac{B^2 (1 + \log(n))}{2\mu n}$$

where

$$\forall n \geq 1, \bar{\theta}_n = \frac{1}{n} \sum_{k=1}^n \theta_{k-1}.$$

2. if $\gamma_n = \frac{2}{\mu(n+1)}$, then

$$\forall n \geq 1, \mathbb{E} [f(\tilde{\theta}_n) - f(\theta^*)] \leq \frac{2B^2}{\mu(n+1)}$$

where

$$\forall n \geq 1, \tilde{\theta}_n = \frac{2}{n(n+1)} \sum_{k=1}^n k \theta_{k-1}.$$

Proof. Let $n \geq 1$, we argue as in the convex case to get

$$\|\theta_n - \theta^*\|^2 \leq \|\theta_{n-1} - \theta^*\|^2 + \gamma_n^2 B^2 - 2\gamma_n \langle \nabla f_n(\theta_{n-1}), \theta_{n-1} - \theta^* \rangle.$$

Taking the expectation with respect to \mathcal{F}_{n-1} and using Proposition 2.4 lead to

$$\begin{aligned} \mathbb{E} [\|\theta_n - \theta^*\|^2 \mid \mathcal{F}_{n-1}] &\leq \|\theta_{n-1} - \theta^*\|^2 + \gamma_n^2 B^2 - 2\gamma_n \langle \nabla f(\theta_{n-1}), \theta_{n-1} - \theta^* \rangle \\ &\leq (1 - \mu\gamma_n) \|\theta_{n-1} - \theta^*\|^2 + \gamma_n^2 B^2 - 2\gamma_n (f(\theta_{n-1}) - f(\theta^*)). \end{aligned}$$

We now consider the whole expectation to obtain

$$\mathbb{E} [\|\theta_n - \theta^*\|^2] \leq (1 - \mu\gamma_n) \mathbb{E} [\|\theta_{n-1} - \theta^*\|^2] + \gamma_n^2 B^2 - 2\gamma_n \mathbb{E} [f(\theta_{n-1}) - f(\theta^*)]$$

which is equivalent to

$$\mathbb{E} [f(\theta_{n-1}) - f(\theta^*)] \leq \frac{\gamma_n B^2}{2} + \frac{1 - \mu\gamma_n}{2\gamma_n} \mathbb{E} [\|\theta_{n-1} - \theta^*\|^2] - \frac{1}{2\gamma_n} \mathbb{E} [\|\theta_n - \theta^*\|^2] \quad (4.3)$$

Let us consider the first statement of the theorem. Since $\gamma_n = 1/(\mu n)$, Inequality (4.3) becomes

$$\mathbb{E} [f(\theta_{n-1}) - f(\theta^*)] \leq \frac{B^2}{2\mu n} + \frac{\mu(n-1)}{2} \mathbb{E} [\|\theta_{n-1} - \theta^*\|^2] - \frac{\mu n}{2} \mathbb{E} [\|\theta_n - \theta^*\|^2].$$

Summing this inequality from 1 to n gives

$$\sum_{k=1}^n \mathbb{E}[f(\theta_{k-1}) - f(\theta^*)] \leq \frac{B^2}{2\mu} \sum_{k=1}^n \frac{1}{k} - \frac{\mu n}{2} \mathbb{E}[\|\theta_n - \theta^*\|^2] \leq \frac{B^2}{2\mu} \sum_{k=1}^n \frac{1}{k}.$$

Thus, the convexity of f and Jensen's inequality imply

$$\mathbb{E}[f(\bar{\theta}_n) - f(\theta^*)] \leq \frac{1}{n} \sum_{k=1}^n \mathbb{E}[f(\theta_{k-1}) - f(\theta^*)] \leq \frac{B^2}{2\mu n} \sum_{k=1}^n \frac{1}{k}$$

which gives the announced result since

$$\sum_{k=1}^n \frac{1}{k} \leq 1 + \int_1^n \frac{dx}{x} \leq 1 + \log(n).$$

We now focus on the second statement of the theorem. Since $\gamma_n = 2/(\mu(n+1))$, (4.3) leads to

$$\mathbb{E}[f(\theta_{n-1}) - f(\theta^*)] \leq \frac{B^2}{\mu(n+1)} + \frac{\mu(n-1)}{4} \mathbb{E}[\|\theta_{n-1} - \theta^*\|^2] - \frac{\mu(n+1)}{4} \mathbb{E}[\|\theta_n - \theta^*\|^2].$$

Thus, we deduce that

$$\sum_{k=1}^n k \mathbb{E}[f(\theta_{k-1}) - f(\theta^*)] \leq \frac{B^2}{\mu} \sum_{k=1}^n \frac{k}{k+1} - \frac{\mu n(n+1)}{4} \mathbb{E}[\|\theta_n - \theta^*\|^2] \leq \frac{B^2 n}{\mu}.$$

Again, the convexity of f and Jensen's inequality imply

$$\mathbb{E}[f(\bar{\theta}_n) - f(\theta^*)] \leq \frac{2}{n(n+1)} \sum_{k=1}^n k \mathbb{E}[f(\theta_{k-1}) - f(\theta^*)] \leq \frac{2B^2}{\mu(n+1)}.$$

□

Although the rate of convergence of $\tilde{\theta}_n$ is asymptotically better than that of $\bar{\theta}_n$, the upper bound obtained for $\bar{\theta}_n$ is better for small values of n . In both case, the convergence rate $1/n$ we obtain with strong convexity is better than that we got in the simply convex case.

4.3 Rates of stochastic gradient descent

We now consider the general framework introduced in Section 4.1. The proofs are omitted due to their complexity but the interested reader will find them in the provided references.

The next results are adapted from the work of Bach and Moulines (2011) cited in [Foreword](#). In this paper, the authors provide important results in the non-asymptotic framework under various sets of hypotheses on the function f . We only focus on the strongly convex case but the interested reader will also find convergence rates obtained under other hypotheses.

In order to state the results of this section, we need to introduce a useful collection of functions $\varphi_\beta : \mathbb{R}_+^* \rightarrow \mathbb{R}$ given by, for any $\beta \in \mathbb{R}$ and $t > 0$,

$$\varphi_\beta(t) = \begin{cases} \frac{t^\beta - 1}{\beta} & \text{if } \beta \neq 0, \\ \log(t) & \text{if } \beta = 0. \end{cases}$$

Note that the function $\beta \mapsto \varphi_\beta(t)$ is continuous for any $t > 0$. Moreover, for $\beta > 0$, we have $\varphi_\beta(t) \leq t^\beta/\beta$ and, for $\beta < 0$, $\varphi_\beta(t) \leq -1/\beta$.

Theorem 4.3. *Let $\gamma > 0$ and step sizes $\gamma_n = \gamma n^{-\alpha}$ for some $\alpha \in [0, 1]$. We consider the stochastic gradient descent $\{\theta_n\}_{n \geq 0}$ given by (4.1) where we assume that*

- *the function $f : \mathbb{R}^d \rightarrow \mathbb{R}$ is μ -strongly convex,*
- *there exists $L > 0$ such that, for any $n \geq 1$, the random function $f_n : \mathbb{R}^d \rightarrow \mathbb{R}$ is almost surely convex and*

$$\forall \theta, \theta' \in \mathbb{R}^d, \mathbb{E} \left[\|\nabla f_n(\theta) - \nabla f_n(\theta')\|^2 \mid \mathcal{F}_{n-1} \right] \leq L^2 \|\theta - \theta'\|^2, \text{ a.s.}$$

- *let $\theta^* \in \mathbb{R}^d$ be the global minimizer of f , there exists $\sigma^2 \geq 0$ such that*

$$\forall n \geq 1, \mathbb{E} \left[\|\nabla f_n(\theta^*)\|^2 \mid \mathcal{F}_{n-1} \right] \leq \sigma^2, \text{ a.s.}$$

For any $n \geq 0$, we denote by $\delta_n = \mathbb{E}[\|\theta_n - \theta^*\|^2]$. If $0 \leq \alpha < 1$, then

$$\delta_n \leq 2 \exp \left(4L^2 \gamma^2 \varphi_{1-2\alpha}(n) - \frac{\mu\gamma}{4} n^{1-\alpha} \right) \left(\delta_0 + \frac{\sigma^2}{L^2} \right) + \frac{4\gamma\sigma^2}{\mu n^\alpha}$$

and, if $\alpha = 1$, then

$$\delta_n \leq \frac{e^{2L^2\gamma^2}}{n^{\mu\gamma}} \left(\delta_0 + \frac{\sigma^2}{L^2} \right) + 2\sigma^2\gamma^2 \frac{\varphi_{\mu\gamma/2-1}(n)}{n^{\mu\gamma/2}}.$$

To make the link with the previous results, note that we freely get bounds for the values of the function f . Indeed, as in Proposition 2.1, the second hypothesis allows us to write

$$\mathbb{E}[f(\theta_n) - f(\theta^*)] \leq \frac{L}{2} \mathbb{E}[\|\theta_n - \theta^*\|^2].$$

The above theorem illustrates the importance of the step size sequence $\{\gamma_n\}_{n \geq 1}$. First, to keep the quantities that appear in the upper bounds as small as possible, we see that small values of γ are good choices. Moreover, the behavior of the stochastic gradient descent estimator varies depending on the value of α . If $0 < \alpha \leq 1/2$, then the presence of quantity $\varphi_{1-2\alpha}(n)$ implies an increasing error for small values of n followed by the expected convergence phenomenon. This change in behavior can be catastrophic if γ is too large, which increases the duration of the increasing error period. Whatever the value of $\alpha \in (0, 1)$, the initial conditions are forgotten sub-exponentially quickly through the term involving δ_0 and we obtain a convergence rate $n^{-\alpha}$. This rate is suboptimal compared to $1/n$ but averaging can improve it at the cost of some additional assumptions.

The case $\alpha = 1$ is particular since the choice of γ becomes critical. If $\gamma < 2/\mu$, the convergence rate $n^{-\mu\gamma/2}$ is arbitrarily small and we get $\log(n)/n$ for $\gamma = 2/\mu$. For $\gamma > 2/\mu$, we obtain the optimal rate $1/n$ but we have to be careful because a too large value of γ leads to an explosion of the term involving initial conditions.

Theorem 4.4. Let $\gamma > 0$ and step sizes $\gamma_n = \gamma n^{-2/3}$. We consider the stochastic gradient descent $\{\theta_n\}_{n \geq 0}$ given by (4.1) where we assume that

- the function $f : \mathbb{R}^d \rightarrow \mathbb{R}$ is μ -strongly convex and twice differentiable,
- for any $n \geq 1$, the random function $f_n : \mathbb{R}^d \rightarrow \mathbb{R}$ is almost surely convex and L -smooth,
- for any $n \geq 1$, the random function $f_n : \mathbb{R}^d \rightarrow \mathbb{R}$ is almost surely twice differentiable with M -Lipschitz Hessian operator Hf_n , namely

$$\forall \theta, \theta' \in \mathbb{R}^d, \|Hf_n(\theta) - Hf_n(\theta')\|_{op} \leq M \|\theta - \theta'\|, \text{ a.s.}$$

where $\|\cdot\|_{op}$ is the operator norm,

- let $\theta^* \in \mathbb{R}^d$ be the global minimizer of f , there exists $\sigma^2 \geq 0$ such that

$$\forall n \geq 1, \mathbb{E} [\|\nabla f_n(\theta^*)\|^2 \mid \mathcal{F}_{n-1}] \leq \sigma^2, \text{ a.s.}$$

- there exist $\tau \geq 0$ and a non-negative self-adjoint operator Σ such that

$$\forall n \geq 1, \mathbb{E} [\|\nabla f_n(\theta^*)\|^4 \mid \mathcal{F}_{n-1}] \leq \tau, \text{ a.s.}$$

and $\Sigma - \mathbb{E} [\nabla f_n(\theta^*) \nabla f_n(\theta^*)^\top \mid \mathcal{F}_{n-1}]$ is almost surely non-negative.

Then,

$$\begin{aligned} \mathbb{E} [\|\bar{\theta}_n - \theta^*\|^2]^{1/2} &\leq \frac{\text{tr}(Hf(\theta^*)^{-1} \Sigma Hf(\theta^*)^{-1})^{1/2}}{\sqrt{n}} \\ &\quad + \frac{C_1}{n^{2/3}} + \frac{C_2}{n} \mathbb{E} [(\|\theta_0 - \theta^*\|^2 + 1)^2]^{1/2} \end{aligned}$$

where

$$\forall n \geq 1, \bar{\theta}_n = \frac{1}{n} \sum_{k=1}^n \theta_{k-1}$$

and $C_1, C_2 > 0$ are constants that depend only on $\gamma, \mu, L, M, \sigma^2$ and τ .

As above, using the L -smoothness of the function allows us to deduce a bound for the values of f ,

$$\mathbb{E} [f(\bar{\theta}_n) - f(\theta^*)] \leq \frac{L}{2} \mathbb{E} [\|\bar{\theta}_n - \theta^*\|^2].$$

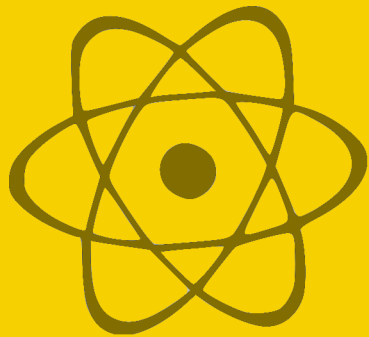
Note that the result is established for the square root of the expectation. We only focus here on the case $\alpha = 2/3$ but general case of step sizes $\gamma_n = \gamma n^{-\alpha}$ for $\alpha \in (0, 1)$ is discussed in the paper of Bach and Moulines. Note that 0 and 1 are excluded since no convergence occurs for $\alpha = 0$ and the convergence rate is already $1/n$ for $\alpha = 1$ with $\gamma > 2/\mu$. The choice of $\alpha = 2/3$ makes the statement easier and the result can be slightly improved if $M = 0$ (i.e. if f is a quadratic function).

The hypotheses are much stronger than in Theorem 4.3. In particular, the almost sure

smoothness of the random functions f_n is not easy to satisfy in practice. Note also that the existence of τ ensures that of Σ and this is indeed only one hypothesis.

The important conclusions of this result is that the convergence rate $1/\sqrt{n}$ of the main term does not depend on the step sizes γ_n and that the constant is optimal since it coincides with the Cramer-Rao lower bound. The initial conditions are now forgotten with a rate $1/n$ only and no longer a sub-exponential one.

By squaring the result of the theorem and expanding the upper bound, we obtain that the second-order term converges with a rate $n^{-7/6}$. In a recent work, Gadat and Panloup proved that this negligible term can be slightly improved for step sizes $\gamma_n = \gamma n^{-3/4}$ and similar hypotheses. They thus obtained a second-order term converging with a rate $n^{-5/4}$, which is better than the above statement but definitely harder to obtain.



Practicals I : Introduction to Python

What is Python?

Python is a widely used programming language, initially released in 1991 by Guido van Rossum. Its reference implementation, called **CPython**, is managed by the **Python Software Foundation** and distributed according to the **Python Software Foundation License**. This licence is compatible with the **GNU General Public License** and approved by the **Open Source Initiative**. Thus, CPython is a free and open-source software.

Python is an **interpreted** programming language, which means that to execute Python code, you have to use a third-party program called **interpreter**. Such softwares are available for many operating systems, allowing Python code to run on a wide variety of systems.

The philosophy of Python is summarized in a sequence of aphorisms known as **PEP 20**. It gives a great importance to code readability and to the capacity to express concepts in few lines of code. This introduction to Python does not claim to be exhaustive and all the good practices induced by the philosophy of the language can not be treated here. However, the reader interested in the **pythonic programming style** is warmly invited to read the **PEP 8**.

Python supports multiple programming paradigms, including object-oriented, imperative and functional programming. It features a **dynamic type system** and **automatic memory management**. We also have at our disposal a large and comprehensive standard library.

About Python version

Nowadays, two distinct versions of Python coexist, version 2 and version 3. These versions are **not compatible**. This is an important point to keep in mind, especially when you seek documentation or code examples on the Web. A list of the main incompatibilities (and problems to solve when porting code from Python 2.x to Python 3.x) can be found in an article of Guido van Rossum called **What's New In Python 3.0**.

The last release of version 2 is Python 2.7 and it was announced in November 2014 that this is the last one, no 2.8 will be released in the future. This version is supported until 2020 but the users are deeply encouraged to move to version 3.

The current stable release is Python 3.6.4. Some attention should be paid to that when documentation pages are browsed. In the sequel of this document, the stable release of Python 3 is silently assumed in all the examples.

Where to find help?

An undeniable force of Python language is its large community. A lot of tutorials, forum threads and articles can easily be found on the Web. The [official documentation](#) is an endless source of information about the language and its ecosystem. Pay attention to the selector at the top left of the page, it allows to pick the Python version.

First steps

Whatever software you have installed to work with Python, you will have at your disposal a **command shell**. This is a window where you can enter line-by-line code to be executed by the interpreter. It is a practical solution to make your first tests but you will quickly realize the limits of this solution that will require you to type the same piece of code every time you want to re-run it. A better solution to avoid this inconvenience and especially to keep track of your work is to write your code in a text file that you will then run. Such a file is referred as a **script**. Solutions to make this process as easy as possible are numerous and are provided by any decent **integrated development environment**.

It is now time to run your first Python command and display the traditional message “*Hello, World!*” on the screen with the following code.

```
print('Hello, World!')
```



Congratulations and welcome in a pythonic world!

Of Variables and Types

A **variable** is a symbolic name associated with a value. In Python, the name of a variable has to satisfy two rules:

- it must start with a letter or an underscore,
- remainder characters may consist of letters, numbers and underscores.

Valid variable name examples are `my_var`, `_var`, `_MyVar01`, `x42_`, ... Moreover, like everything else in Python, variable names are **case sensitive**, *i.e.* `myVar`, `MyVar` and `mYvaR` are three distinct variable names. It should also be noted that variables whose names begin with an underscore play a special role in Python and are therefore to be avoided in the general case.

To assign a value to a variable, we use the operator `=` with the variable name on the left and the value to assign on the right. If the variable does not yet exist, it is created. Otherwise, its value is simply replaced.

```
my_var = 17
print(my_var)
my_var = 8
print(my_var)
```



The content of a variable has a **type**, *i.e.* a category that determines the possible values for the variable. Usual types are boolean, integer, floating number, string, ... To know the type of a variable, use the function `type`:

```
x = True
print(type(x))
x = 42
print(type(x))
x = 42.0
print(type(x))
x = 'Am I a string?'
print(type(x))
```

The example above illustrates an important feature of Python, namely **dynamic typing**. This means that the type of a variable is that of its content at the time of its use. Since this content is allowed to change from one line to another, the type of the variable changes as it is assigned.

Below, we give some examples of manipulations of the most common **numeric types** in Python. Note in particular the syntax of comments that allow us to give details inside the code.

- Booleans:

```
# Result of a condition
test = (3 > 4)
print(test)

# Logical operations
not True
True and False
True or False

# Bitwise operators
True & False # AND
True | False # OR
```

- Integers:

```
# Usual arithmetic operations
3 + 7
5 - 8
-3 * 4
5**2 # Power
3 / 2 # Floating division

# Euclidean division
3 // 2
3 % 2
```

```
# Cast boolean
int(True)
```

- Floating numbers:

```
# Usual operations
3.14 + 1.23
1.8 - 7
6.6 * 3.3
3.0 / 2.0
3.14**2.4 # Power

# Cast integer
float(42)
```



Containers

Python offers **container types** for storing multiple objects. We first present **lists** that can receive a set of objects that can be of **different types**.

```
L1 = ['red', 'pink', 'orange', 'blue', 'black']
L2 = [42, 3.14, 'A string', ['An', 'other', 'list']]
print(type(L1))

# Indexation starts at zero
print(L1[0], L1[1])

# From the end with negative indices
print(L1[-1], L1[-2])

# Slicing a list (start included, end excluded)
L1[1:3]
L1[:3]
L1[3:]
L1[-2]
L1[:2]

# A list is mutable
L1[1] = 'brown'
L1[2:4] = ['purple', 'yellow', 'gray', 'white']
print(L1)
```



To manipulate a vector data, *i.e.* a set of numeric data of the same type, it is more efficient to use an **array**. This is a container type from the scientific module *NumPy* that we will discuss in the next practical session. Python also offers a wide range of tools for manipulating lists.

```
# Concatenation and repetition
L1 + L2
L2 * 2

# Add items
L1.append('tan')
print(L1)
L1.extend(['cyan', 'orange'])
print(L1)

# Remove and return the last item
item = L1.pop()
print(item)
print(L1)

# Reverse
L1.reverse()
print(L1)

# Sort
sorted(L1) # New object
print(L1)
L1.sort() # In place
print(L1)
```

In the examples above, we use a particular syntax with a dot “.” to designate certain functions that belong to the list object (append, extend, ...). Such internal functions to an object are called **methods**. This is an object-oriented programming concept that we will see later. For the moment, it is enough to consider that these are functions made available by an object and which act on this object (*e.g.* note the difference between the function sorted and the method sort).

The container type **tuple** is an **immutable variant** of list. The syntax is similar except that we use (optional) parentheses to create a tuple.

```
# Different types are allowed
t1 = (3, 'Hello', 42.0)
print(type(t1))

# Parentheses are optional
t2 = 'Other', 'tuple'
print(t2)

# But immutable
t2[0] = 'New' # Error...
```

A useful feature of tuples is to allow multiple assignments in one line. Such an operation is quite common with Python and can often be used.

```
# Multiple assignments
a, b, c = 4, 2, 1
# Swap variable contents
a, b = b, a
```



Let us now consider the container type **string**. Such a container stores characters and can be defined with simple quotes `' '` or double quotes `" "`, tripling them to allow multiline strings.

```
s1 = 'Hello my friend!'
s2 = "Do you like Python?"
s3 = '''Such a language
    allows amazing things.'''
s4 = """Isn't it?
Really?"""
print(type(s1))
print(s3)
print(s4)

# Indexation works like with the lists
s1[0]
s1[1]
s1[-1]
s1[3:6] # From index 3 (included) to 6 (excluded)
s1[2:10:2] # Syntax start:stop:step

# Strings are immutable
s1[0] = 'Z' # Error...

# But there are useful methods
s1.replace('l', 'z', 1)
s1.replace('l', 'z')
print(s1)
```



A very common operation with strings is **formatting**. This allows you to include the contents of other variables in a string.

```
a = 3.14
b = 42
s = 'a is %f and b is %i.' % (a, b) # It is OK
s = 'a is {} and b is {}'.format(a, b) # It is better
s = 'a is {p1} and b is {p2}'.format(p1=a, p2=b) # It is best
print(s)
```



The last container type we introduce here is **dictionary**. It is an **associative array** that connects keys to values together. It is an **unordered container**.

```
phone_prefix = {'France': 33, 'Vietnam': 84}
print(type(phone_prefix))
```




```
# Add item on the fly
phone_prefix['Cuba'] = 53
print(phone_prefix)

# Access with the key
phone_prefix['France']

# All keys and values
phone_prefix.keys()
phone_prefix.values()
phone_prefix.items()
```

Control flows

As any programming language, Python has mechanisms to control the order in which the code is executed. Such statements are followed by a colon “:” and a **block of instructions**. It is now time to warn you about the **importance of indentation** with Python. The number of spaces (or tabs) at the beginning of a line indicates the block of instructions to which it belongs. If this number of spaces is incorrect, the interpreter will report an error about the **indentation level**.

The **conditional statements** are classically done with if/elif/else. The elif statement stands for “else if” and is useful to chain the conditions. The statements elif and else are not mandatory.

```
# A simple conditional statement
if 3 < 4:
    print('Yes, this is true!')

# Chaining the conditions
if 3 > 4:
    print('Are you sure?')
elif 3 == 4:
    print('Let me doubt...')
elif 3 < 4:
    print('This is the right one!')

# If/else statement
if 3 > 4:
    print('Ouch!')
else:
    print('That is better!')

# A complete conditional statement
if 3 > 4:
    print('Are you sure?')
elif 3 == 4:
```

```
print('Let me doubt...')
else:
    print('This is the right one!')
```

To make a **loop** across elements of a container, we use the statements `for` and `in`. The instruction `range` is useful to browse integers.

```
# Loops on integers
for i in range(7):
    print(i)
for i in range(2, 7, 2): # Syntax start (included), stop (excluded), step
    print(i)

# Loops with containers
L = ['red', 'orange', 'pink']
for color in L:
    print(color)
for letter in L[1]:
    print(letter)
for i, color in enumerate(L):
    s = '{index} - {color}'.format(index=i, color=color)
    print(s)

# Case of dictionaries
D = {'Gandalf': 'wizard', 'Frodo': 'hobbit'}
for item in D:
    print(item)
for key, value in D.items():
    s = '{key} is a {value}'.format(key=key, value=value)
    print(s)
```

The loop statement `for` can also be used to create lists according to some pattern.

```
L = [i**2 for i in range(5)]
print(L)
```

To **repeat** a block of instructions until a condition becomes false, use the `while` statement. This is useful in situations where the number of iterations is not known in advance.

```
# Request a value from the user
value = input('Enter an integer: ')
i = int(value) # Convert to integer

# Search for the first multiple of 7 greater than i
mul7 = i
while mul7 % 7 != 0:
    mul7 = mul7 + 1
print(mul7)
```

Functions

To declare a function, we use the `def` statement and we list the arguments in parentheses. If no arguments are needed, there must be empty parentheses. An argument can have a **default value** which will be used if the argument is omitted when the function is called. Once again, a colon character “:” introduces the **body of the function** that has to respect indentation constraints. A function can return values with the `return` statement.

```
# No argument, no return
def say_hello():
    print('Hello!')
say_hello()

# With an argument
def say_hello_to(name):
    greeting = 'Hello, {name}!'.format(name=name)
    print(greeting)
say_hello_to('Bobby')

# Use default values
def universal_answer(x=42):
    print('The answer is {}'.format(x))
universal_answer(3.14)
universal_answer() # Argument is omitted

# Return a value
def disk_area(radius):
    return 3.14159 * radius**2
area = disk_area(2)
print(area)

# Return multiple values with a tuple
def euclidean_division(a, b=2):
    return a // b, a % b
q, r = euclidean_division(7, 3)
q, r = euclidean_division(7) # Argument b is omitted, default is 2

# Order of named arguments does not matter
q, r = euclidean_division(b=3, a=7)
```

With Python, a function can modify the content of a **mutable container** passed as an argument. This has to be kept in mind to avoid **side effects**.

```
def modify_it(container):
    container[0] = 42

L = ['Have', 'fun'] # Lists are mutable
modify_it(L)
print(L)
```

A function is an object like any other in Python. In particular, a function can be passed as an argument to another function.

```
def double_it(x):  
    return 2*x  
  
def do_something(x, f):  
    print(f(x))  
  
do_something(3, double_it)
```



Modules, packages and import

So far, we have typed all the code in the command shell or in a text editor. As the size of a project grows, it becomes more comfortable to divide the code into smaller text files called **scripts** or **modules**. A **package** is a collection of modules organized in a specific **directory hierarchy**. This is aimed to make the code easier to read, understand and manage. Python projects like those we will use in the following (*NumPy*, *Pandas*, *Matplotlib*, ...) are structured in this way. We explain here how to use the objects defined in a given module or package for our purpose. We will not explain how to write our own packages but the interested reader will find a lot of useful informations to this end in the [official documentation](#) and on the Web.

To load the code of a module, we use the `import` command. The simplest way is `import` followed by the name of the module. As an example, we import the module `os` from the standard library (see details in next section) that provides tools to access operating system features.

```
import os  
  
# List all the module definitions  
dir(os)  
  
# Call some functions from the module  
os.getcwd() # Get the current directory  
os.mkdir('test_directory') # Create a directory  
os.listdir() # List the content of the current directory  
os.rmdir('test_directory') # Remove a directory
```



It is important to notice how the objects defined in a module are called after the import. Indeed, it is necessary to **prefix** the name of these objects by the name of the module and a point “.” (e.g. `os.getcwd`, `os.mkdir`, ...). In this way, imported objects do not replace already defined objects that have the same name. We say that these objects belong to a **namespace**.

A package is composed of modules but can also contain **subpackages** with their own modules. This nested hierarchy corresponds to those of files placed in a folder structure. To import a module from a subpackage, we use the same syntax by detailing the path of the module. To shorten the namespace prefix, we can use the `as` statement.

```
# Import module path from package os
import os.path

# Use objects from os.path as usual
os.path.exists('undefined_file') # Does the file exist?

# Shorten the namespace
import os.path as op

# Same as above with short prefix
op.exists('undefined_file')
```

There is another way to import Python objects with the “from ... import” statement. The main difference with the previous method is that the objects are now imported into the **global namespace** (*i.e.* without any prefixes). If it may seem simpler at first glance, it is better to **avoid such a way**. Indeed, importing objects into the global namespace can cause several problems:

- this makes the code harder to read because the origin of the instructions is not clear,
- this can replace objects that already exist in the global namespace (*e.g.* `os.open` and the Python function `open`),
- the order in which the modules are imported becomes important because conflicts between the names defined in the modules can occur,
- this makes the code really more difficult to debug.

All of these disadvantages should be kept in mind if you use such a way to import objects from a module. Especially if you use the **star import** (*i.e.* “from ... import *” statement) which imports all the definitions from a module into the global namespace (it is definitely better to avoid this way).

```
# Importing objects into the global namespace
# Use with caution...
from os import getcwd, listdir

getcwd() # No need to prefix now

# Importing all the definitions into the global namespace
# *** Do not do that! ***
from os import *

mkdir('test_directory')
rmdir('test_directory')
```

Standard library

Python comes with a lot of modules that make up the **Python standard library**. It would be beyond the scope of this document to give an exhaustive review of all these modules. However, it is a good habit to have a look in the documentation before you start coding new features. There is a good chance that useful and well-made tools are already available. Hereafter, we mention several useful standard modules and some of their features.

Module os We have already mentioned this module in the previous section. It is aimed to provide a portable way of using operating system dependent functionalities.

```
import os

# Handle directories
os.getcwd() # Get the current directory
os.mkdir('test_directory') # Create a directory
os.listdir() # List the content of the current directory
os.rmdir('test_directory') # Remove a directory

# Deal with file paths
path = os.path.join('path', 'to', 'my', 'file')
print(path) # Output depends on your operating system

# Does a file exist?
os.path.exists('some_file')
```

Module sys This module provides access to some variables used or maintained by the interpreter.

```
import sys

# Get a platform identifier
print(sys.platform)

# What is the Python version?
print(sys.version)
```

Module math We find in this module some mathematical functions commonly defined in standard libraries. We will not use this module very much because the scientific packages presented in the following section also provide this kind of definition and many other things.

```
import math

# Some useful constants
math.pi
math.e

# Common functions
x = 4.2
```

```
math.fabs(x) # Absolute value
math.exp(x) # Exponential
math.log(x) # Logarithm
math.sqrt(x) # Square root
math.cos(x), math.sin(x), math.tan(x) # Trigonometry
```

Module random This module implements pseudo-random number generators for various distributions. As for the module `math`, we prefer to use the objects defined by the scientific package *NumPy* that we will consider in the next session.

```
import random

# Random integer between 4 and 9
random.randint(4, 9)
# Uniform distribution in [0,1]
random.random()
# Uniform distribution in [1.7, 4.2]
random.uniform(1.7, 4.2)
# Gaussian distribution with mean -8.1 and standard deviation 4.2
random.gauss(-8.1, 4.2)

# Useful tools for set
x = [i for i in range(10)]
random.sample(x, 4)
random.shuffle(x)
print(x)
```

Module datetime The `datetime` module supplies tools for manipulating dates and times as well as the associated arithmetic.

```
import datetime

now = datetime.datetime.now()
str(now) # Date and time as a string
now.weekday() # 0 is Monday, 1 is Tuesday, ...

# Direct access to the attributes
now.year
now.month
now.day
now.hour
now.minute
now.second
now.microsecond

# Arithmetic
delta = datetime.timedelta(days=17, hours=15)
str(now - 2*delta)
```

Exceptions

When the interpreter encounters an error, it **interrupts** the running script and **raise an exception**. We have already met examples of this phenomenon without discussing it. If an exception is not handled, we observe a raw error message with a name and some details about what happened.

```
x = 1 / 0 # ZeroDivisionError: division by zero
print(undefined) # NameError: name 'undefined' is not defined
x = 1 + '1' # TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

In these examples, the names of the exceptions are `ZeroDivisionError`, `NameError` and `TypeError`. These exception names are quite self-explanatory and you can get the whole list of built-in exceptions in the [official documentation](#).

To handle (or **catch**) an exception, use statements `try` and `except`. First, the block of instructions between `try` and `except` is executed. If an exception is raised, the remainder of the block is skipped and the exception is caught if its name matches the `except` statement which is then run. Such a mechanism allows multiple `except` clauses to catch various exception types.

```
try:
    x = 1 / 0
except ZeroDivisionError:
    print('Division by zero!')

try:
    function_that_can_raise_exceptions()
except ZeroDivisionError:
    print('Stop, division by zero!')
except IndexError:
    print('Ouch, out of range!')
except KeyboardInterrupt:
    print('Why did you stop me?')
```

An exception is an object and can be manipulated as any other one. To this end, we can use the statement `as` to handle it as a variable.

```
try:
    x = 1 / 0
except ZeroDivisionError as e:
    print('Type: {}'.format(type(e)))
    print('Message: {}'.format(e))
```

To raise an exception from a function, we use the `raise` statement. This is the pythonic way to handle and report an error.

```
def i_want_integer(n):
    # Check the type of n
```



```
if not isinstance(n, int):
    raise TypeError('n has to be an integer')
print('I got an integer! This is {}'.format(n))

i_want_integer('Hi!')
i_want_integer(42)
```

A glance at object-oriented programming

Python supports the **object-oriented programming (OOP)** paradigm. Such a paradigm is aimed to:

- help with code organization,
- make it easier to reuse certain portions of code.

The central concept of OOP is the **class**. This is a **model** for building objects and providing them with internal variables called **attributes** and internal functions called **methods**. To define a class in Python, we use the `class` statement and the object to be built according to the model is referred as `self`.

```
class Character(object):
    def __init__(self, name):
        self.name = name
        self.skills = []

    def add_skill(self, skill):
        self.skills.append(skill)

    def set_race(self, race):
        self.race = race

samwise = Character('Samwise Gamgee')
samwise.set_race('hobbit')
samwise.add_skill('drinking')
samwise.add_skill('eating')
print('{name} is a {race}'.format(name=samwise.name, race=samwise.race))
```

In the above example, we defined a class `Character` with three attributes (`name`, `race` and `skills`) and two methods (`add_skill` and `set_race`). The word `object` in the parentheses is not mandatory but this is a good practice. We also define a third method called `__init__` which is known as the **constructor** and is used to create an object from the class. Note that all the methods take here `self` as first argument. Moreover, we access the attributes and methods of object `samwise` through the dot syntax that we have already seen before.

Let us now assume that we want to create some special characters Wizard who are able to cast some spells. The only difference with a `Character` will be the method `cast_spell` and we want to avoid typing again the same code. To this end, we create a new class `Wizard` as a **derived class** of `Character`. We say that `Wizard` **inherits** from `Character`.

```

class Wizard(Character):
    def __init__(self, name, spell):
        Character.__init__(self, name)
        self.skills.append('casting spells')
        self.spell = spell

    def cast_spell(self):
        print('*** ' + self.spell + ' ***')

gandalf = Wizard('Gandalf', 'Pedo mellon a minno!')
print('{name} knows {s}'.format(name=gandalf.name, s=gandalf.skills[0]))
gandalf.cast_spell()

```

The syntax for declaring a derived class is the same as above except that the **parent class** name is passed in parentheses instead of object. Note that in the constructor we explicitly call the `__init__` function of the parent class. All methods and attributes of the parent class are available without any other form of declaration.

Using OOP, we can organize the code into separate classes that correspond to different parts of the project (a class for "Spell", a class for "Ring", ...) with specific attributes and methods. Then we use inheritance to create variations of these classes by reusing their code. Deriving a class defined in a module is a common operation in Python to produce objects suitable for a particular purpose.

A recapitulative exercise

To conclude this first meeting with Python, we propose to code three algorithms to approximate the number π . The first two are based on deterministic principles and the third uses randomly scattered points. Finally, we group these features into a class to manage them in a unified way.

1. Let us introduce the **Leibniz series**,

$$\forall n \geq 0, S_n = 4 \sum_{k=0}^n \frac{(-1)^k}{2k+1}.$$

It is easy to prove that this series converges to π when n grows to infinity. Write a Python function called `leibniz` that takes an argument `n` and returns the value of S_n .

2. Run `leibniz(0)`, `leibniz(10)`, `leibniz(100)`, ... and compare the results with the value `math.pi`.
3. How do you handle bad arguments like `leibniz(-1)` or `leibniz('Ouch')`? If needed, do it with built-in exceptions.
4. We now consider the **Borwein's algorithm** to approximate π . Let $\{y_n\}_{n \geq 0}$ and $\{a_n\}_{n \geq 0}$ be two sequences of real number given by $y_0 = \sqrt{2} - 1$, $a_0 = 6 - 4\sqrt{2}$ and, for any $n \geq 1$,

$$\begin{cases} y_n = \frac{1 - (1 - y_{n-1}^4)^{1/4}}{1 + (1 - y_{n-1}^4)^{1/4}} \\ a_n = a_{n-1}(1 + y_n)^4 - 2^{2n+1}y_n(1 + y_n + y_n^2). \end{cases}$$

Then, the sequence $\{1/a_n\}_{n \geq 0}$ converges to π . Write a Python function `borwein_items` that takes an argument `n` and returns the pair (y_n, a_n) as a tuple.

5. Use `borwein_items` to write a function `borwein` that takes an argument `n` and returns the approximation of π given by $1/a_n$. Correctly treat a bad value passed as an argument.
6. Run `borwein(0)`, `borwein(10)`, `borwein(100)`, ... and compare the results with the value `math.pi`. What do you notice?
7. The third method for approximating π is based on n points picked uniformly at random in the square $[0, 1] \times [0, 1]$. We denote by N_n the number of points falling at a distance less than 1 from the origin. Explain why we know that

$$F_n = \frac{4N_n}{n} \xrightarrow[n \rightarrow \infty]{a.s.} \pi.$$

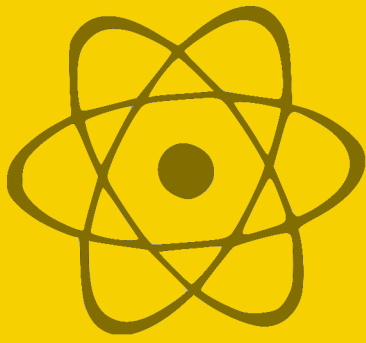
8. Write a Python function called `circle` that takes an argument `n` and returns the value of F_n .
9. Run `circle(0)`, `circle(10)`, `circle(100)`, ... and compare the results with the value `math.pi`. What happens when you run these commands again?
10. Create a class `Pi` whose the constructor takes two arguments:
 - `method`: a string to designate the algorithm to use ('leibniz', 'borwein' or 'circle'),
 - `timer`: a boolean value to indicate whether the elapsed time should be measured or not (default is `False`).

The class will also provide a method `approx` that takes a single argument `n` and returns the approximation of π calculated with the algorithm `method`. If the argument `timer` is set to `True`, the time elapsed in microseconds during the last call must be available in an attribute called `elapsed_time`.

```
P0 = Pi('borwein')
P0.approx(10)

P1 = Pi('circle', True)
P1.approx(1000000)
print('Elapsed time: {}'.format(P1.elapsed_time))
```





Practicals 2 : Python for scientists

NumPy

NumPy is a Python package that provides tools to work with **multi-dimensional arrays** (vectors, matrices, ...), along with a large library of **high-level mathematical functions** to operate on these arrays. *NumPy* is distributed under the terms of a revised BSD license and is thus a **free software**. In the sequel, to shorten the namespace prefix, we use the common alias `np` for `numpy`.

```
import numpy as np

# NumPy supplies usual constants
np.pi
np.e
```



The class `array` plays a central role in *NumPy*. Such an object can be instantiated with the function `array` from a list for a vector or from a list of lists for a matrix. To access the elements of an array, we use the usual syntax based on square brackets `[]` and slicing.

```
# Create a vector from a list
v = np.array([17.0, 42.0, 8.1, 19.0])
print(v, type(v))

# Create a matrix from a list of rows
m = np.array([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
print(m, type(m))

# Print behaves nicely with large arrays
np.array(range(10000))
np.array([range(500) for i in range(500)])

# Access the elements as usual
v[0]
v[1:3]
```



```
# Arrays are mutable (be careful with side effects)
v[2] = 15.0
v[1:3] = 0.0

# Similar syntax for matrices
m[0,1]
m[:,2]
```

To get informations about array dimensions (rows, columns, ...) the class array supplies a tuple attribute `shape`. Moreover, the method `reshape` allows to modify the dimensions of an array.

```
v.shape
m.shape

m1 = v.reshape(2, 2)
m2 = m.reshape(3, 2)
v1 = m.reshape(6)
```



The class array offers numerous useful functions to do basic computations with arrays. There are a lot of tools defined for arrays as you can see in the [index of the documentation](#). All these tools can be called as methods or as functions from the *NumPy* namespace.

```
# Sum of the elements
v.sum() # As method
np.sum(v) # From NumPy namespace

# Cumulative sum of the elements
v.cumsum()
np.cumsum(v)

# Maximum and minimum
m.min(), m.max() # Or np.min(m), np.max(m)
m.argmin(), m.argmax() # Indices of min and max

# Mean and variance
v.mean() # Or np.mean(v)
m.var() # Or np.var(m)
```



A nice feature offered by *NumPy* arrays and functions is the ability to apply a function to all elements of an array **without loop**. Such a syntax definitely makes the code easier to read and is often used in the following.

```
n = 6
x = np.array([np.pi * i / (n - 1) for i in range(n)])
np.cos(x)
```



```
# This also works with common operations ...
x + 1
2 * x
x / 5

# ... but not with math functions!
import math
math.cos(x) # Error!
```

Fortunately, we do not need to explicitly create all arrays and *NumPy* provides useful functions for the most common situations. These functions are available for vectors and matrices.

```
# Create arrays full of zeros
np.zeros(10)
np.zeros((3, 4)) # Tuple argument

# Create arrays full of ones
np.ones(10)
np.ones((3, 4)) # Tuple argument

# Create arrays with uninitialized entries
np.empty(10)
np.empty((3, 4)) # Tuple argument

# Identity matrix
np.identity(5)

# Diagonal matrix from a vector
v = np.array([1.0, 2.0, 3.0])
np.diag(v)

# Vector from the matrix diagonal
m = np.array([[1.0, 2.0], [3.0, 4.0]])
m.diagonal() # Or np.diag(m)
```

To avoid poorly readable commands based on `range` when creating arrays, *NumPy* provides two useful functions. Function `arange` is similar to `range` but the returned object is an array. Function `linspace` returns an array of evenly spaced numbers over a specified interval. This second function will be very useful to plot a function (see the section about *Matplotlib*).

```
# Create arrays in range style
np.arange(10)
np.arange(8, 17) # Start included, end excluded
np.arange(8, 17, 2) # With a step

# Useful for matrices too!
np.arange(8, 17).reshape(3, 3)
```

```
# Evenly spaced numbers
np.linspace(2, 3, 10) # Syntax: start, end, number
```

NumPy offers a syntax based on **boolean arrays** to access some elements in an array. This way can be very useful when we handle data sets and it is available to get values but **also to modify** them.

```
# Notice the concatenation operation and the negative step
a = np.append(np.arange(5), np.arange(4, -1, -1))

# With an explicit boolean array
b = [i % 3 != 0 for i in range(a.shape[0])]
a[b]

# With a condition vector
b = (a >= 2)
a[b]

# Useful to modify some values
a[b] = 42
print(a)

# Also works with matrices
a = np.arange(12).reshape(3,4)
b1 = np.array([False, True, True])
b2 = np.array([True, False, True, False])

a[b1, :] # Selecting rows
a[:, b2] # Selecting columns
a[b1, b2] # Selecting both but ...
```

Specific operations and tools are needed when handling matrices. There are common operations like addition, multiplication, ... But *NumPy* also supplies tools to deal with matrices as common Python containers.

```
m1 = np.array([[1,1], [0,1]])
m2 = np.array([[2,0], [3,4]])

# Addition works as usual
m1 + m2

# Be careful with multiplication!
m1 * m2 # Element by element
m1.dot(m2) # True matrix multiplication
np.dot(m1, m2) # Idem from NumPy namespace

# Various functions can be applied on axis
```



```
m1.sum() # Sum of all items
m1.sum(axis=0) # Sum of each column
m1.sum(axis=1) # Sum of each row

# Stacking rows and columns
v1 = np.array([1, 2, 3])
v2 = np.array([4, 5, 6])
np.vstack((v1, v2)) # Tuple argument
np.hstack((v1, v2)) # One line matrix

v1 = np.array([[1], [2], [3]])
v2 = np.array([[4], [5], [6]])
m = np.hstack((v1, v2)) # Three lines matrix

# Iterate on matrix rows
for row in m:
    print('--->', row)

# Iterate on matrix elements with flat operator
for item in m.flat:
    print('--->', item)
```

Matrix calculation with NumPy is not restricted to these elementary operations and a large set of linear algebra tools are supplied. Note that some functions come from the submodule `numpy.linalg` (see [module documentation](#) for details).

```
m1 = np.arange(12).reshape(3, 4)
m2 = np.array([[3, 1, 2], [2, 0, 5], [1, 2, 3]])

# Transpose operator
m1.T
m1.transpose() # Or np.transpose(m1)

# Determinant and trace operators
np.linalg.det(m2)
np.trace(m2) # Or m2.trace()

# Inverse matrix
np.linalg.inv(m1) # Raise a LinAlgError exception
m2_inv = np.linalg.inv(m2)
np.dot(m2, m2_inv)

# Eigenvalues and eigenvectors
np.linalg.eigh(np.dot(m1, m1.T)) # For Hermitian matrix only
np.linalg.eig(m2) # Otherwise, right eigenvectors are returned
```

More advanced procedures are at your disposal and we encourage you to browse the [documentation](#) to discover all the possibilities. As an example, we give below the commands

to perform a least squares polynomial fit.

```
x = np.array([0.0, 1.0, 2.0, 3.0, 4.0, 5.0])
y = np.array([0.0, 0.8, 0.9, 0.1, -0.8, -1.0])
z = np.polyfit(x, y, 3) # Coefficients of polynomial of degree 3

# Get a function for polynomial given by z
pol_z = np.poly1d(z)
pol_z(2.5)
```

To conclude this introduction to *NumPy*, we mention that it also supplies a **large set of random generators** in the submodule `numpy.random`. Some distributions are already available in the standard library but, with `numpy.random` functions, you are able to directly get array objects.

```
# Random arrays
np.random.rand(3) # Uniform distribution in [0,1]
np.random.randn(2, 3) # Standard normal distribution

# Random sample
np.random.choice(np.array([17, 8, 19, 1, 3, 15]), 4) # With replace
np.random.choice(np.array([17, 8, 19, 1, 3, 15]), 4, False) # No replace

# Distributions
np.random.standard_exponential(5) # Standard exponential
np.random.exponential(2, 5) # Exponential with parameter 2
np.random.poisson(np.pi, 10) # Poisson with parameter pi
```

Matplotlib

Matplotlib is a plotting library that was originally written by John D. Hunter and that is distributed under a BSD compatible licence, thus *Matplotlib* is a **free software**. *Matplotlib* is divided into several parts that are in charge of:

- **creating plots** with code similar to other scientific softwares,
- **managing** figures, text, lines, plots, ...
- handling **display devices** (or **backends**) such as *PostScript*, *SVG*, *PNG*, ... ,
- **interfacing** with widget toolkits (*Gtk+*, *Qt*, *Cocoa*, ...).

For this introduction, we will only focus on the creation and the management of graphics. The submodule of *Matplotlib* that handles this is called `pyplot` and we will use the common alias `plt`.

```
import matplotlib.pyplot as plt
```

The simplest graph consists to display points in the plane from their coordinates passed as lists to the command `plot`. Consecutive `plot` calls add more points to the current graph. To create a new graph, use the command `figure`. Depending on how you interact with Python (raw command shell in console, development environment, ...), the graph window may not appear by default. If necessary, the `show` command will pops this window (such a command will be omitted in the other examples). The function `axis` allows to adjust the axes of the current plot.

```
plt.plot([1, 2, 1, 2]) # As a sequence
plt.plot([15, 3, 1, 5], [42, 17, 8, 5]) # With X and Y coordinates

# Pop the window (not always needed)
plt.show()

# Create a new graph
plt.figure()
plt.plot([15, 3, 1, 5], [42, 17, 8, 5])
plt.show() # If needed ...

# Modify the axes
plt.axis([-5, 20, 0, 50]) # Syntax: xmin, xmax, ymin, ymax
```

An alternative to `figure` to reset a graph is the function `clf` but the current plot will be lost. The function `plot` admits an optional third argument as a string to define **basic formatting** (color, marker, linestyle, ...). The syntax is simple but not really intuitive and we refer to the [documentation](#) for details.

```
# Reset current figure
plt.clf()

# Cyan star markers
plt.plot([15, 3, 1, 5], [42, 17, 8, 5], 'c*')

# Red square markers
plt.plot([14, 2, 0, 4], [41, 16, 7, 4], 'rs')

# Green dash-dot line and circle markers
plt.plot([0, 5, 10, 15], [42, 17, 8, 5], 'go-.')
```

If *Matplotlib* were limited to working with lists, it would be fairly useless for numeric processing. Generally, you will use *NumPy* arrays (actually, all sequences are converted to *NumPy* arrays internally). This is where the *NumPy* function `linspace` becomes very interesting for producing **functional plots**. As mentioned in the documentation, more graphic options are available (alpha, linewidth, ...) and we will introduce some of them in the following examples.

```
# Gaussian increments
n = 1000
z = np.random.randn(n) # NumPy array

# Discretized Brownian motion
t = np.linspace(0, 1, n)
b = z.cumsum() / np.sqrt(n)
plt.plot(t, b, 'k-', linewidth=0.5) # Set line width
```

A great freedom is given by *Matplotlib* to tweak the figures according to our wishes. Let's start with some **decorating functions** to set title, labels and comments. Notice that these functions allow the use of TeX expressions surrounded by dollar signs “\$... \$”.

```
mu = -1.0
sigma2 = 0.5

# Plot Gaussian densities
t = np.linspace(-4, 4, 500)
plt.plot(t, np.exp(-t**2/2) / np.sqrt(2*np.pi), 'k--')
plt.plot(t, np.exp(-(t-mu)**2/(2*sigma2)) / np.sqrt(2*np.pi*sigma2), 'r-')

# Add a title
plt.title('Some Gaussian densities')

# Add labels for the axes
plt.xlabel('Value')
plt.ylabel('Density')

# Add text
plt.text(2, 0.25, 'Hello, Gauss!')

# Add annotation
plt.annotate('Local maximum', xy=(-1.0, 0.564), xytext=(0.5, 0.45),
            arrowprops={'facecolor': 'black', 'width': 1.0})

# Replace the title with TeX expression
plt.title('Parameters:  $\mu = \{m\}$ ,  $\sigma^2 = \{s2\}$ '.format(m=mu, s2=sigma2))
```

The decoration functions provided by *Matplotlib* are numerous and we can not give an exhaustive list in this document. For example, the following code shows how to add a legend and a grid. For more details, we encourage you to read the [beginner's guide](#).

```
def gauss(t, mu=0.0, sigma2=1.0):
    return np.exp(-(t-mu)**2 / (2*sigma2)) / np.sqrt(2*np.pi*sigma2)

# Plot two curves with labels (used by legend)
t = np.linspace(-4, 4, 500)
plt.plot(t, gauss(t), 'k-', label='$\sigma^2=1$')
```

```
plt.plot(t, gauss(t, sigma2=0.5), 'r-', label='$\sigma^2=1/2$')

# Add a legend
plt.legend()

# Add a grid
plt.grid()
```

A common question when plotting curves is the ability to draw **multiple graphs** on the same device with common properties (aligned axes, ...). With *Matplotlib*, the function `subplot` allows you to do that. This function takes three arguments which correspond to a **plotting grid**: the number of rows, the number of columns and the position for next plot in this grid.

```
def f(t):
    return np.exp(-t) * np.cos(2*np.pi*t)

# Distinct abscissas
t1 = np.arange(0.0, 5.0, 0.1)
t2 = np.arange(0.0, 5.0, 0.02)

# First plot
plt.subplot(2, 1, 1) # 2 rows and 1 column, position 1
plt.plot(t1, f(t1), 'bo', t2, f(t2), 'k')

# Second plot
plt.subplot(2, 1, 2) # Same grid, position 2
plt.plot(t2, np.cos(2*np.pi*t2), 'r--')
```

Of course, *Matplotlib* is not limited to simple point sequence plots. The package supplies a lot of graphical representations and tools but it would be beyond the scope of this document to make an complete review of these features. We give below an insight about the possibilities. If you are interested in **data visualization**, you should have a look to the [Matplotlib examples](#).

```
# Scatter plot
n = 256
x = np.random.randn(n)
y = np.random.randn(n)
size = 100*np.random.random(n)
plt.scatter(x, y, s=size, alpha=0.5)

# Histogram
n = 2048
x = np.random.randn(n)
u = np.linspace(-4, 4, 2*n+1)
v = gauss(u)
plt.hist(x, bins=50, normed=True, color='orange')
plt.plot(u, v, 'r-', linewidth=2)
```

```
# Pie chart
ore = ['Gold', 'Silver', 'Ore', 'Mithril']
count = [31, 58, 93, 21]
color = ['gold', 'silver', 'brown', 'lightskyblue']
explode = (0, 0.1, 0, 0)
plt.pie(count, labels=ore, colors=color, explode=explode,
        shadow=True, startangle=90)

# Boxplot
n = 256
x = np.random.randn(n)
y = np.random.randn(n)
plt.boxplot((x, y))

# Violin plot
plt.violinplot((x, y))
```

Pandas

The package **Pandas** is aimed to provide efficient data structures and data analysis tools. It is distributed under a BSD compatible licence, thus Pandas is also a **free software**. *Pandas* offers a lot of features that we will not be able to fully cover in this brief introduction. Here are the few functionalities we will present in this section:

- how to load data from a *CSV* file,
- how to work with *DataFrame* objects,
- how *Pandas* nicely integrates *Matplotlib* to produce high quality plots.

The interested reader is warmly encouraged to read the [documentation](#) for more information. Let's start by importing *Pandas* and shorten the namespace with the usual prefix *pd*.

```
import pandas as pd
```



Pandas provides various functions to read data from a file (*HTML*, *JSON*, ...). The one that interests us here is `read_csv` which allows to recover the data contained in a *CSV* file. This function can take many arguments depending on how the *CSV* file is organized (delimiter, column names, ...) but most default values are often relevant. As an example, we propose to load the data **Most Popular Baby Names by Sex and Mother's Ethnic Group, New York City** made available by the city of New York. For readability, the file has been renamed `NYnames.csv`.

```
data = pd.read_csv('NYnames.csv')
```



The object returned by `read_csv` is a `DataFrame` which is the **most commonly used** *Pandas* object. A `DataFrame` is a 2-dimensional labeled data structure with columns of potentially **different types**.

```
type(data) # pandas.core.frame.DataFrame

# Content of the DataFrame
print(data)

# First look on the data set
len(data) # Number of rows
data.shape # Dimensions as with a NumPy array
data.columns # Column names
data.dtypes # Column types
data.head() # Five first rows
data.tail() # Five last rows
```

We can get a **summary** of the data with the method `describe`. This returns several **statistical measurements** (mean, standard deviation, extremal values, quantiles, ...) only for numeric columns.

```
data.describe()
```

There are two ways to access the columns of a `DataFrame`. The first one is to use the name of the column in the manner of a dictionary. Several columns can be selected this way through a list of column names. The other way to access a column is to use the dot syntax “.” but only if the name of the column is a valid variable name for Python (no space, ...) and if it is not already used by *Pandas* (`head`, `shape`, ...). You will notice that the returned objects can be used **in place of NumPy arrays** and supply similar methods for basic computations. This integration within *Pandas* is one of the great strengths of this library.

```
# As with a dictionary
data['CNT']

# Works with multiple columns
data[['NM', 'CNT']]

# With the dot syntax
data.CNT

# Columns can be used as NumPy arrays ...
v = data['CNT']
np.log(v)
# ... and supply usual statistical methods
v.mean()
v.std()
v.argmax()
```

Of course, columns and rows can also be accessed **through indices** with `iloc` and slicing syntax.

```
# Get the first row
data.iloc[1]

# Get the first column
data.iloc[:,1]

# Works with slicing
data.iloc[5:10,3:]
data.iloc[:5] # Same as head
data.iloc[-5:] # Same as tail
```



The statistical functions at our disposal for a column generally act on the whole set of values. If we want to run our own functions **across all values** in a column (or row), we can use `apply`. Such a way is usually **more efficient** than a loop.

```
# A custom function
def name_length(name):
    return len(name)

# Apply it across a column
names = data['NM']
names.apply(name_length)
```



When working with data, it is common to select some rows according to specific conditions. With Pandas, the boolean vectors associated with such a selection are referred as **filters**. The dictionary syntax of a `DataFrame` object allows to directly use these filters in square brackets **to extract a part of the data set**. Note that to be valid a composed filter must use bitwise operators (`&`, `|`, `...`) and not boolean ones (`and`, `or`, `...`).

```
# Filter for the most given names
max_name = data.CNT > 300

# Extract the data
data[max_name]
data[data.CNT > 300] # Same result

# Composed filter
data[(data.CNT > 300) & (data.GNDR == 'FEMALE')]

# Powerful with custom filtering
def starts_with_X(name):
    return name.startswith('X')

data[data.NM.apply(starts_with_X) & (data.GNDR == 'MALE')]
```




```
# Test the presence of at least one element in the filter
max_name_300 = data.CNT > 300
max_name_300.any() # At least one name has been given 300 times
max_name_500 = data.CNT > 500
max_name_500.any() # No name has been given 500 times
```

If some variables are **categorical**, it may be useful to **partition** the data set according to these ones. This is done through the function `groupby` which returns an iterable object containing the name of each category and the corresponding **subset of data** with the same column names. It is then possible to use all the functions available for a `DataFrame` object to retrieve statistical measures.

```
# Group by gender
for gender, df in data.groupby('GNDR'):
    n = len(df)
    print("{gender}: {n} names.".format(gender=gender, n=n))

# Crossing categorical variables
for c, df in data.groupby(['ETHCTY', 'GNDR']):
    m = df.CNT.mean()
    msg = "{c}: same name is given {m} times on average.".format(c=c, m=m)
    print(msg)
```

Last functionality but not the least, *Pandas* nicely integrates *Matplotlib* through some functions that allow to quickly obtain **basic graphs** from `DataFrame` objects. Although the possibilities are not as broad as what *Matplotlib* allows, *Pandas* offers a wide variety of graphs that we will not list here. We give some examples below but, for an exhaustive presentation, the reader is invited to consult the page dedicated to the **visualization** of the official documentation.

```
# Histogram
data.CNT.hist()

# Box plots
data.boxplot(column='CNT', by='GNDR') # Group by gender

# Advanced example
df = data.groupby(['BRTH_YR', 'ETHCTY']).sum().CNT.unstack()
# Clean data
df = df.fillna(0)
df['ASIAN'] = df['ASIAN AND PACI'] + df['ASIAN AND PACIFIC ISLANDER']
df['BLACK'] = df['BLACK NON HISP'] + df['BLACK NON HISPANIC']
df['WHITE'] = df['WHITE NON HISP'] + df['WHITE NON HISPANIC']
df = df[['ASIAN', 'BLACK', 'HISPANIC', 'WHITE']]
# Plot the frequencies (bar need Pandas >= 0.17.0)
df.div(df.sum(axis=1), axis=0).plot.bar()
```

A recapitulative exercise

We propose to study meteorological data measured in the city of Rennes (France) that are provided by the file `ozone.csv`. We have at our disposal $n = 91$ observations of the following variables:

- `max03`: maximum concentration of ozone measured during the day,
- `T6`, `T9`, `T12`, `T15` and `T18`: temperature at 6:00, 9:00, 12:00, 15:00 and 18:00,
- `Ne6`, `Ne9`, `Ne12`, `Ne15` and `Ne18`: nebulosity at 6:00, 9:00, 12:00, 15:00 and 18:00,
- `Vx`: wind speed on the east-west axis.

The objective is to predict ozone concentration `max03` from other data using a linear regression model.

1. Load the data into a `DataFrame` object with `read_csv` (use argument `sep`).
2. Briefly explore the data set (`columns`, `dtypes`, `describe()`, ...).
3. What is the day when the ozone concentration is maximum?
4. We begin by considering a simple linear model in which we want to explain $y = \text{max03}$ only from $x = \text{T18}$.
 - (a) Define the *NumPy* arrays `x` and `y` associated with the values of `T18` and `max03`, respectively.
 - (b) Output the scatter plot of the data with the plotting functions of *Pandas* from the `DataFrame` object and with the *Matplotlib* functions from the vectors `x` and `y`.
 - (c) We want to minimize the least squares criterion according to $a, b \in \mathbb{R}$, namely

$$\gamma(a, b) = \frac{1}{n} \sum_{k=1}^n (y_k - a - b \times x_k)^2.$$

Show that

$$\frac{\partial \gamma}{\partial a}(a, b) = 2(a + b \times \bar{x} - \bar{y}) \quad \text{and} \quad \frac{\partial \gamma}{\partial b}(a, b) = 2(a \times \bar{x} + b \times \overline{x^2} - \overline{xy})$$

where we have set

$$\bar{x} = \frac{1}{n} \sum_{k=1}^n x_k, \quad \bar{y} = \frac{1}{n} \sum_{k=1}^n y_k, \quad \overline{x^2} = \frac{1}{n} \sum_{k=1}^n x_k^2 \quad \text{and} \quad \overline{xy} = \frac{1}{n} \sum_{k=1}^n x_k y_k.$$

- (d) Prove that the **global minimizer** of γ is given by $(\hat{a}, \hat{b}) \in \mathbb{R}^2$ where

$$\hat{a} = \bar{y} - \hat{b} \times \bar{x} \quad \text{and} \quad \hat{b} = \frac{\overline{xy} - \bar{x} \times \bar{y}}{\overline{x^2} - \bar{x}^2}.$$

- (e) Compute \hat{a} and \hat{b} from the vectors `x` and `y` with their *NumPy* functionalities.

- (f) Plot the regression line $y = \hat{a} + \hat{b} \times x$ over the scatter plot.
- (g) Add to the graph some lines to visualize the residuals,

$$\varepsilon_k = y_k - \hat{a} - \hat{b} \times x_k, \quad k \in \{1, \dots, n\}.$$

5. We now consider a linear model to explain $y = \text{max03}$ from T18 and Ne6. With the intercept, this leads us to three variables that we handle through the following $n \times 3$ matrix,

$$X = \begin{bmatrix} 1 & \text{T18}_1 & \text{Ne6}_1 \\ 1 & \text{T18}_2 & \text{Ne6}_2 \\ \vdots & \vdots & \vdots \\ 1 & \text{T18}_n & \text{Ne6}_n \end{bmatrix}$$

- (a) Define the *NumPy* arrays X and y corresponding to the matrix X and to the vector y , respectively.
- (b) Using the *Matplotlib* [toolkit](#) `mplot3d`, create a 3D scatter plot of the data.
- (c) Compute $X^\top X$ and verify that this matrix is invertible with *NumPy* functionalities.
- (d) In its matrix form, the least squares criterion to minimize according to $\theta \in \mathbb{R}^3$ is given by

$$\gamma(\theta) = \|y - X\theta\|^2.$$

Show that

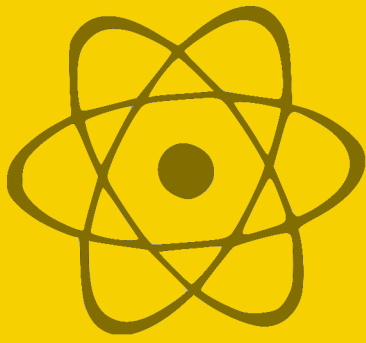
$$\nabla \gamma(\theta) = -2X^\top (y - X\theta).$$

- (e) Prove that the **global minimizer** of γ is given by $\hat{\theta} \in \mathbb{R}^3$ where

$$\hat{\theta} = \left(X^\top X\right)^{-1} X^\top y.$$

- (f) Compute $\hat{\theta}$ through the *NumPy* functionalities.
- (g) With the help of the `mplot3d` [tutorial](#), add the surface of the regression plane $z = \hat{\theta}_0 + \hat{\theta}_1 x + \hat{\theta}_2 y$ to the above scatter plot.
- (h) Add to the graph some lines to visualize the residuals,

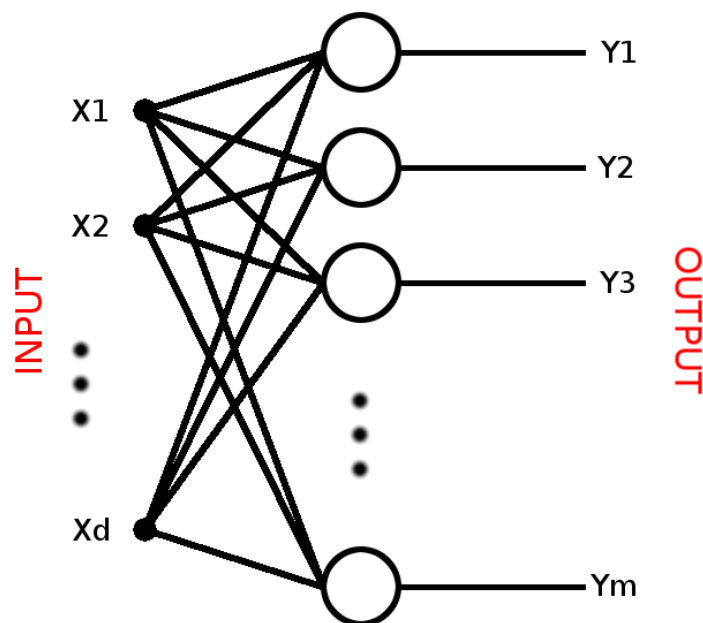
$$\varepsilon_k = y_k - (X\hat{\theta})_k, \quad k \in \{1, \dots, n\}.$$



Practicals 3 : Adaline

Introduction

An **artificial neural network** is a computing system vaguely inspired by the biological neural networks that constitute animal brains. It is said that such a system “learns” in the sense that the **parameters** that define it can be calibrated from a **training data set** to improve the quality of its **predictions on new data**. The literature on neural networks is vast and this is not the purpose of this session. We will consider here only **single-layer neural networks** taking $d \geq 1$ **real input** variables and returning one **discrete output** (or more). The usual representation of such a network is of the following form:



Let's focus now on the elemental brick of such a neural network. A **neuron** is characterized by two objects:

- a function $\phi : \mathbb{R} \rightarrow \mathbb{R}$, called **activation function**,

- a vector $\theta = (\theta_0, \dots, \theta_d)^\top \in \mathbb{R}^{d+1}$ where the θ_k are referred as the **weights**.

Given an input vector $x = (x_1, \dots, x_d)^\top \in \mathbb{R}^d$ and a discrete output value $y \in \{-1, 1\}$, the neuron compute the quantity

$$\phi \left(y - \theta_0 - \sum_{k=1}^d \theta_k x_k \right).$$

Common choices for the activation function are Heaviside, sigmoid, ... The goal is then to **fit the parameter** θ with respect to a training data set to minimize an **empirical criterion** used to measure the adequation of a prediction rule based on the output of the neuron and the observations of y . In other words, we are facing a **minimization problem** that we have to solve.

Such an optimization problem can be more or less easy to tackle according to the choice of the function ϕ . In practice, the **backpropagation** is commonly used to this end but is beyond the scope of this document. In 1960, Widrow and Hoff proposed to simply consider $\phi(z) = z$ and the quadratic loss function. The neural network we obtain, called **ADALINE** (ADaptive Llear NEuron), is really basic and simply amounts to minimize the empirical least squares criterion given by n observations of the variables,

$$\hat{\theta} \in \underset{\theta \in \mathbb{R}^{d+1}}{\operatorname{argmin}} \|Y - X\theta\|^2$$

where X is the matrix of size $n \times (d+1)$ defined by

$$X = \begin{bmatrix} 1 & x_{1,1} & \dots & x_{d,1} \\ 1 & x_{1,2} & \dots & x_{d,2} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{1,n} & \dots & x_{d,n} \end{bmatrix}$$

and $Y = (y_1, \dots, y_n)^\top \in \{-1, 1\}^n$ the observed output vector. Thus, the decision rule for a new input observation $x^{\text{new}} = (x_1^{\text{new}}, \dots, x_d^{\text{new}})^\top \in \mathbb{R}^d$ is given by the sign of the affine combination of its coordinates,

$$\operatorname{sg} \left(\hat{\theta}_0 + \sum_{k=1}^d \hat{\theta}_k x_k^{\text{new}} \right) = \begin{cases} 1 & \text{if } \hat{\theta}_0 + \sum_{k=1}^d \hat{\theta}_k x_k^{\text{new}} \geq 0, \\ -1 & \text{otherwise.} \end{cases}$$

Generate data

In order to study the role of certain parameters, in particular the impact of the number n of observations, we will not use real data here but simulated data. The following Python function makes it possible to generate n (default is 1) Gaussian observations in dimension d (default is 2) composed of centered variables associated with the label 1 and variables decentered along a unit vector u associated with the label -1 . Then, the function returns the input $n \times (d+1)$ matrix x , the output vector y of size n and the unit vector u . If the argument u is None (default), a random unit vector is sampled uniformly on the sphere of dimension d .

```
def generate_sample(n=1, d=2, u=None):
    # Get the offset
    if u is None:
        u = np.random.randn(d)
        u = u / np.linalg.norm(u)

    # Output vector
    y = np.random.choice((-1, 1), n)

    # Gaussian samples
    x = np.random.randn(n, d)
    x[y == -1] += np.log(n) * u
    x = np.hstack([np.ones((n,1)), x])

    return x, y, u
```

Batch approach

Let's start with the batch approach consisting in computing explicitly the minimizer of the function

$$\forall \theta \in \mathbb{R}^{d+1}, J(\theta) = \|Y - X\theta\|^2.$$

1. Let $\theta \in \mathbb{R}^{d+1}$, compute the gradient $\nabla J(\theta)$ and prove that J is L -smooth. What is the value of L ?
2. Prove that the function J is convex.
3. Prove that the function J is μ -strongly convex if and only if the matrix $X^\top X$ is invertible. What is the value of μ in this case?
4. Assuming that $X^\top X$ is invertible, deduce that $\theta^* = (X^\top X)^{-1} X^\top Y$ is the unique global minimizer of J .
5. Write a Python function `fit_batch` that takes `x` and `y` as arguments and return θ^* .
6. Generate a sample of size $n = 500$ and dimension $d = 2$ and run `fit_batch` with it. Produce a scatter plot of the data with red dots for label 1 and blue dots for label -1 . Add the line given by the equation $\theta_0^* + \theta_1^* x_1 + \theta_2^* x_2 = 0$ to visualize the frontier of the decision rule.
7. Run the following code to time your function `fit_batch` for various values of d and n .

```
import time

d_values = [2, 5, 10]
n_values = [10**(k+1) for k in range(6)]
elapsed = np.zeros((len(d_values), len(n_values)))
rep = 10 # Number of repetitions
```

```

for i, d in enumerate(d_values):
    for j, n in enumerate(n_values):
        elapsed_current = 0.0
        for _ in range(rep):
            start = time.time()
            # Generate a sample
            x, y, u = generate_sample(n, d=d)
            # Fit the data
            theta = fit_batch(x, y)
            end = time.time()

            elapsed_current += end - start
        elapsed[i,j] = elapsed_current / rep

for i, d in enumerate(d_values):
    label = 'd={}'.format(d)
    plt.plot(np.log(n_values), elapsed[i], 'o-', label=label)
plt.legend()

```

Gradient Descent

We now consider the gradient descent $\{\theta_n\}_{n \geq 0}$ with constant step sizes $\gamma_n = \gamma$ given by $\theta_0 = 0$ and the recursion

$$\forall n \geq 1, \theta_n = \theta_{n-1} - \gamma \nabla J(\theta_{n-1}).$$

1. Write a Python function `gradJ` that takes three arguments: the input matrix `x`, the output vector `y` and the value `theta`. This function must return the value of the gradient $\nabla J(\theta)$.
2. Write a Python function `fit_gd` that takes four arguments: the input matrix `x`, the output vector `y`, the step size `gamma` and the number of iterations `n_iter`. This function must return the **whole** sequence $\theta_1, \dots, \theta_{n_iter}$ as a $n_iter \times (d+1)$ matrix.
3. Considering the properties of the function J obtained in the previous section, state the result of convergence that we expect for $J(\theta_{n_iter}) - J(\theta^*)$.
4. Run your function `fit_gd` on a sample of size $n = 500$ and dimension $d = 2$ with 1000 iterations and a step size $\gamma = 10^{-3}$. Look at the result, what's the problem? Try again with a step size $\gamma = 10^{-5}$, is it better?
5. Run the following code to visualize the convergence of $\|\theta_{n_iter} - \theta^*\|^2$.

```

x, y, u = generate_sample(500)
theta = fit_gd(x, y, 1e-5, 1000)
theta_star = fit_batch(x, y)

```




```
err = np.zeros(theta.shape[0])
for i, theta_n in enumerate(theta):
    e = (theta_n - theta_star)**2
    err[i] = e.sum()
plt.plot(err)
```

6. Using the vector `err`, determine approximately the number of iterations necessary to obtain an accuracy $\varepsilon = 10^{-2}$.
7. Plot the logarithm of $\|\theta_{n_iter} - \theta^*\|^2$. Is the result consistent with what you expected in question 3?
8. Adapt the code of Question 7 of the previous section to time your function `fit_gd` with $\gamma = 10^{-5}$ for various values of `n_iter`. To avoid “freezing” your computer, start with small sample sizes n .

Stochastic Gradient Descent

Finally, we focus on the stochastic gradient descent $\{\hat{\theta}_n\}_{n \geq 0}$ with constant step sizes $\gamma_n = \gamma$ given by $\hat{\theta}_0 = 0$ and the recursion

$$\forall n \geq 1, \hat{\theta}_n = \hat{\theta}_{n-1} - \gamma \nabla J_n(\theta_{n-1})$$

where, for any $\theta \in \mathbb{R}^{d+1}$, $\nabla J_n(\theta)$ is the estimator of the gradient $\nabla J(\theta)$ associated to the n -th observation $(x_n, y_n) \in \mathbb{R}^d \times \{-1, 1\}$,

$$\nabla J_n(\theta) = -2 \left(y_n - \theta_0 - \sum_{k=1}^d \theta_k x_{k,n} \right) \times \begin{pmatrix} 1 \\ x_{1,n} \\ \vdots \\ x_{d,n} \end{pmatrix} \in \mathbb{R}^{d+1}.$$

1. Write a Python function `gradJn` that takes three arguments: the input vector `xn`, the output value `yn` and the current state of the stochastic gradient descent `theta`. This function must return the value of the gradient $\nabla J_n(\text{theta})$.
2. We could now start our stochastic gradient descent by simulating the data **one by one** (but with the same unit vector `u`, see Section *Generate data*) to take advantage of the on-line property of the algorithm. However, this would make it difficult to compare with θ^* . Thus, we generate a whole data set of size n with which we can compute our target θ^* . Then, to simulate the **on-line aspect** of our approach, we randomly pick a permutation of $\{1, \dots, n\}$ and we take the observations in the induced order to update the stochastic gradient descent. We repeat this as many times as necessary to make `n_iter` iterations. This procedure corresponds to the code below.

```

n = 256
x, y, u = generate_sample(n)
theta_star = fit_batch(x, y)

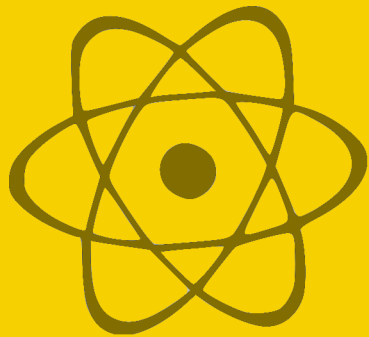
order = []
gamma = 1e-2
n_iter = 1000
theta = np.zeros((n_iter, x.shape[1]))
for k in range(1, n_iter):
    if not order:
        order = np.random.permutation(n).tolist()
    I = order.pop()
    theta[k] = theta[k-1] - gamma * gradJn(x[I], y[I], theta[k-1])

```

3. For any $k \in \{0, \dots, d\}$, plot the difference $\hat{\theta}_{n,k} - \theta_k^*$ to visualize how the stochastic gradient descent behaves relative to θ^* . Comment the aspect of this graphic.
4. Change the value of gamma. What happens when γ increases? And when γ decreases?
5. Add few lines to your code to compute the on-line averaged version of the stochastic gradient descent, namely

$$\forall n \geq 1, \bar{\theta}_n = \frac{1}{n} \sum_{k=1}^n \hat{\theta}_k = \bar{\theta}_{n-1} + \frac{1}{n} (\hat{\theta}_n - \bar{\theta}_{n-1}).$$

6. As in Question 3, for any $k \in \{0, \dots, d\}$, plot the difference $\bar{\theta}_{n,k} - \theta_k^*$ to visualize the behavior of the averaged version. Increase the number of iterations if necessary.



Practicals 4 : Applications in Statistics

Mean estimation

Let's start with the simple example we have seen in the lectures. Given a sequence $\{X_n\}_{n \geq 1}$ of independent and identically distributed random variables, we consider the following stochastic gradient descent algorithm to estimate the mean $\mathbb{E}[X_1] = \theta^* \in \mathbb{R}$. Let $\theta_0 = 0$, we define $\{\theta_n\}_{n \geq 0}$ through the recursion

$$\forall n \geq 1, \theta_n = \theta_{n-1} - \gamma_n(\theta_{n-1} - X_n)$$

where the step sizes are given by $\gamma_n = \gamma n^{-\alpha}$ with $\gamma > 0$ and $\alpha \in [0, 1]$.

1. What are the random functions f_n such that the recursion can be written as

$$\forall n \geq 1, \theta_n = \theta_{n-1} - \gamma_n \nabla f_n(\theta_{n-1}).$$

Verify that

$$\forall n \geq 1, \mathbb{E}[\nabla f_n(\theta) \mid X_1, \dots, X_{n-1}] = 0 \iff \theta = \theta^*.$$

2. If $\gamma = 1$ and $\alpha = 1$, what is θ_n ? Compute the 1000 first iterations of the algorithm with data randomly generated **inline** with a Gaussian distribution $\mathcal{N}(5, 1)$. Plot the obtained sequence to visualize how the estimation behaves.
3. Change the value of γ to 0.5 and 0.1. What happens when γ decreases?
4. Change the value of α to 0.75, 0.5, 0.25 and 0. How the estimators sequence evolves when α decreases? What happens when $\alpha = 0$?
5. Consider the averaged sequence $\{\bar{\theta}_n\}_{n \geq 0}$ given by $\bar{\theta}_0 = 0$ and the recursion

$$\forall n \geq 1, \bar{\theta}_n = \bar{\theta}_{n-1} + \frac{1}{n}(\theta_n - \bar{\theta}_{n-1}).$$

Change your code to simultaneously compute θ_n and $\bar{\theta}_n$. Plot the obtained sequences on the same graph. Explain what happens when α decreases towards 0.

Regression model

We now return to the meteorological data contained in the file `ozone.csv` (see *Practicals 2*). We propose to use a linear regression model to predict the variation of the ozone maximal concentration $y = \text{max03} - \text{max03v}$ between one day and the previous one with respect to the variables

$$x^1 = \text{T18}, x^2 = \text{Ne18} \text{ and } x^3 = \text{Vx}.$$

We have at our disposal $n = 91$ observations and, as usual, we denote by X the input matrix of size $n \times 4$ whose columns correspond to the intercept and the 3 input variables and by $Y = (y_1, \dots, y_n)^\top \in \mathbb{R}^n$ the output vector. Thus, we are looking for θ^* given by

$$\theta^* \in \underset{\theta \in \mathbb{R}^4}{\operatorname{argmin}} \|Y - X\theta\|^2.$$

To this end, we propose to use a stochastic gradient descent algorithm defined as follows. Let $\{\sigma_N\}_{N \geq 1}$ be a sequence of independent uniform random permutations of $\{1, \dots, n\}$, we define the random functions sequence $\{f_k\}_{k \geq 1}$ by

$$\forall k \geq 1, \theta \in \mathbb{R}^4, f_k(\theta) = \frac{\left(y_{\sigma_N(k)} - x_{\sigma_N(k)}^\top \theta\right)^2}{2} \quad \text{if } k \in \{(N-1)n+1, \dots, Nn\},$$

where, for any $i \in \{1, \dots, n\}$, x_i stands for the i -th row of X . Then, we introduce the sequence $\{\theta_k\}_{k \geq 0}$ given by $\theta_0 = 0$ and the recursion

$$\forall k \geq 1, \theta_k = \theta_{k-1} - \gamma_k \nabla f_k(\theta_{k-1})$$

where the positive step sizes sequence $\{\gamma_k\}_{k \geq 1}$ remain to be defined.

1. Let $N \geq 1$ and $k \in \{(N-1)n+1, \dots, Nn\}$, compute $\nabla f_k(\theta)$ for $\theta \in \mathbb{R}^4$ and prove that

$$\mathbb{E}[\nabla f_k(\theta) \mid \sigma_1, \dots, \sigma_{N-1}] = 0 \iff \theta = \theta^*.$$

2. Load the data set from the file `ozone.csv` (do not forget the column of X for the intercept) and compute the optimal value `theta_star` of $\theta^* = (X^\top X)^{-1} X^\top Y$.
3. Let us first apply this algorithm with constant step sizes $\gamma_k = \gamma > 0$. Use the *NumPy* function `np.random.permutation` to draw the random permutations of $\{1, \dots, n\}$ and write a loop to compute the `n_iter = 1000` first iterations of the stochastic gradient descent that will be stored in a matrix `theta` of size `n_iter` \times 4. Visualize how the obtained coefficients behave with respect to θ^* and experiment with several values of γ to get a “satisficient” result. Does the algorithm converge?
4. Modify your code to introduce the averaged version $\{\bar{\theta}_k\}_{k \geq 0}$ given by $\bar{\theta}_0 = 0$ and the recursion

$$\forall k \geq 1, \bar{\theta}_k = \bar{\theta}_{k-1} + k^{-1}(\theta_k - \bar{\theta}_{k-1}).$$

Visualize the obtained coefficients. Does the result seem better? Does it converge?

5. Consider the second averaging version $\{\tilde{\theta}_k\}_{k \geq 1}$ we have studied in the lectures defined by $\tilde{\theta}_0 = 0$ and

$$\forall k \geq 1, \tilde{\theta}_k = \frac{2}{k(k+1)} \sum_{\ell=1}^n \ell \theta_{\ell-1}.$$

Prove that the sequence $\{\tilde{\theta}_k\}_{k \geq 1}$ satisfies the recursion

$$\forall k \geq 1, \tilde{\theta}_k = \tilde{\theta}_{k-1} + \frac{2}{k+1} (\theta_k - \tilde{\theta}_{k-1}).$$

Modify your code to compute inline $\{\tilde{\theta}_k\}_{k \geq 1}$. Visualize and comment the obtained coefficients with respect to the previous averaging version.

6. We now consider decreasing step sizes $\gamma_k = \gamma \times \lceil k/n \rceil^{-\alpha}$ with $\alpha \in [0, 1]$ where, for any $t \in \mathbb{R}$, $\lceil t \rceil$ is the least integer greater than or equal to t (see the *NumPy* function `np.ceil`). Adapt your previous code to take these step sizes into account and experiment with several values of α . To what corresponds the case $\alpha = 0$? What happens when α increases? Take larger `n_iter` to clearly visualize the result if needed.

Ridge regression model

We consider the same notations and data set than in the previous section. During the lectures, we have seen that in some situations (e.g. when $X^\top X$ is not invertible), it is fruitful to handle a **regularized** version of the least squares criterion, namely

$$\theta_\lambda^* \in \operatorname{argmin}_{\theta \in \mathbb{R}^4} \|Y - X\theta\|^2 + \lambda \|\theta\|^2$$

where $\lambda > 0$ is a regularizing parameter. Such an estimator is known as the **Ridge regression estimator**.

1. Let $J(\theta) = \|Y - X\theta\|^2 + \lambda \|\theta\|^2$, $\theta \in \mathbb{R}^4$, compute the gradient $\nabla J(\theta)$ and show that the function J is 2λ -strongly convex.
2. Prove that the Ridge regression estimator is given by

$$\theta_\lambda^* = \left(X^\top X + \lambda \mathbf{I}_4 \right)^{-1} X^\top Y$$

where \mathbf{I}_4 is the 4×4 identity matrix.

3. For $\lambda \in \{0, \dots, 100\}$, compute θ_λ^* and visualize how its coordinates evolve with respect to the ones of θ^* . We say that the Ridge regression “shrinks” the regression coefficients.
4. Let’s fix $\lambda = 0.5$, we propose to mimic the stochastic gradient descent algorithm introduced in the previous section to compute θ_λ^* . Explain with your own words how to set up this algorithm and, in particular, what random functions J_k you have to define. The obtained sequence is denoted by $\{\theta_k^\lambda\}_{k \geq 0}$ in the sequel.
5. Implement your algorithm with decreasing step sizes and an averaged versions $\{\bar{\theta}_k^\lambda\}_{k \geq 0}$ and $\{\tilde{\theta}_k^\lambda\}_{k \geq 0}$ of the stochastic gradient descent. Visualize the obtained coefficients and compare them to the ones of θ_λ^* .

6. Compute and visualize the errors sequences $\{\|\theta_k^\lambda - \theta^*\|^2\}_{k \geq 0}$, $\{\|\bar{\theta}_k^\lambda - \theta^*\|^2\}_{k \geq 0}$ and $\{\|\tilde{\theta}_k^\lambda - \theta^*\|^2\}_{k \geq 0}$. Discuss these results with respect to what we have seen during the lectures. Using logarithmic scales should be useful to this end.

Logistic regression model

Finally, we consider the logistic regression as an example of a statistical methodology for which there is not closed expression for the minimizer. To illustrate this procedure, we use the data set given by the file `chd.csv` that contains $n = 100$ rows and two columns:

- age: the age of the patient,
- chd: a binary value equal to 1 if the patient suffers from coronary heart disease and 0 otherwise.

Thus, our goal is here to study the relation between age and chd through the logistic regression model

$$\mathbb{P}_\theta(\text{chd} = 1 \mid \text{age} = x) = \frac{\exp(\theta_0 + \theta_1 x)}{1 + \exp(\theta_0 + \theta_1 x)}$$

where $\theta = (\theta_0, \theta_1) \in \mathbb{R}^2$ is an unknown parameter. Considering the opposite of the log-likelihood function as the loss function for this problem, we define θ^* as a minimizer of the empirical risk, namely

$$\theta^* \in \operatorname{argmin}_{\theta \in \mathbb{R}^2} \sum_{k=1}^n \log(1 + \exp(-(2\text{chd}_k - 1)(\theta_0 + \theta_1 \text{age}_k))).$$

1. Load the data set from the file `chd.csv` (be careful, delimiter character is “;”) and produce the scatter plot of the data. How the probability to suffer from coronary heart disease seems to evolve when the age increases?
2. We proceed as with the regression model, let $\{\sigma_N\}_{N \geq 1}$ be a sequence of independent uniform random permutations of $\{1, \dots, n\}$, we define the random functions sequence $\{f_k\}_{k \geq 1}$ by

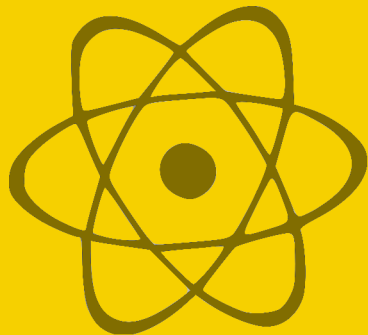
$$(N-1)n+1 \leq k \leq Nn, \theta \in \mathbb{R}^2, f_k(\theta) = \log \left(1 + e^{-(2\text{chd}_{\sigma(k)}-1)(\theta_0 + \theta_1 \text{age}_{\sigma(k)})} \right).$$

Then, we introduce the sequence $\{\theta_k\}_{k \geq 0}$ given by $\theta_0 = 0$ and the recursion

$$\forall k \geq 1, \theta_k = \theta_{k-1} - \gamma_k \nabla f_k(\theta_{k-1})$$

where, for any $k \geq 1$, we have set the step size $\gamma_k = \gamma \times [k/n]^{-\alpha}$ with $\gamma > 0$ and $\alpha \in [0, 1]$. Prove that, for any $k \geq 1$, the function f_k is convex.

3. Let $k \geq 1$ and $\theta \in \mathbb{R}^2$, compute the gradient $\nabla f_k(\theta)$. To this end, you can treat separately the cases $\text{chd}_{\sigma(k)} = 0$ and $\text{chd}_{\sigma(k)} = 1$.
4. Implement the algorithm including the averaged versions of the stochastic gradient descent and adjust the values of `n_iter`, γ and α to make it converge.
5. Plot the obtained sigmoid curve above the scatter plot. Does it fit well? What is your conclusion about the relation between age and chd?



Practicals 5 : Going further

Mini-batch approach

In the lectures, we have studied two gradient descent algorithms:

- **Batch Gradient Descent** if we know an explicit form of the gradient of the function to be minimized, we can compute it with the **entire data set** to update the gradient descent sequence. We obtain an algorithm that is not often updated with a stable result. However, each iteration can be time consuming for a large data set and the stability of the algorithm can cause it to converge too quickly in some local minima.
- **Stochastic Gradient Descent** with only estimators of the gradient of the function to be minimized, we can update the gradient descent sequence **observation-by-observation** and thus provide an on-line estimation procedure. We get a frequently updated algorithm with noisy progression that can help us avoid local pitfalls. Due to the many updates, this approach can be computationally expensive and its random nature can lead to a longer convergence time.

To find a trade-off between these two methods and try to exploit the advantages of each one, there exists an intermediate approach referred as the **mini-batch approach**. The idea consists in **updating less often** by accumulating a number m of observations called the **mini-batch size** in the sequel. Thus, each step amounts to update the gradient descent sequence with the help of these m observations. If we deal with a finite data set of size n , parameter m allows to **find a balance** between the stochastic gradient descent (m close to 1) and the batch gradient descent (m close to n). This approach is widely used nowadays when implementing machine learning algorithms.

The choice of the mini-batch size m is not straightforward in practice and may depend on various elements: data availability, semi real time needs, computational capacities of the computer, ... An interesting feature of modern computers is the **vectorization** that allows making several similar computations in parallel. These tools are commonly more efficient as the size of the data is aligned with that of the hardware. A good practice is to use powers of 2 multiplied by 32 bits (32, 64, 128, 256, ...) to take advantage of available vectorized operations.

Let us illustrate this approach with the simple example of the mean estimation. Given a sequence $\{X_n\}_{n \geq 1}$ of independent and identically distributed random variables, we plan to

estimate the mean $\mathbb{E}[X_1] = \theta^* \in \mathbb{R}$. Using the quadratic loss function with m observations, we have already seen that this problem amounts to minimize the random function

$$\forall \theta \in \mathbb{R}, J_m(\theta) = \frac{1}{2m} \sum_{j=1}^m (X_j - \theta)^2.$$

Thus, for a mini-batch size of m , the gradient used to update the algorithm is given by

$$\forall \theta \in \mathbb{R}, \nabla J_m(\theta) = \theta - \bar{X}_m$$

where \bar{X}_m is the empirical mean of the observations X_1, \dots, X_m . The mini-batch gradient descent sequence $\{\theta_k^{(m)}\}_{k \geq 0}$ is given by $\theta_0^{(m)} = 0$ and the recursion

$$\forall k \geq 1, \theta_k^{(m)} = \theta_{k-1}^{(m)} - \gamma_k \nabla J_m(\theta_{k-1}^{(m)})$$

where we set the step sizes to $\gamma_k = \gamma \times k^{-\alpha}$ with $\gamma > 0$ and $\alpha \in [0, 1]$.

1. Implement the above algorithm with data randomly generated inline with a Gaussian distribution $\mathcal{N}(5, 1)$. Run your code with various values of m to understand the role of this parameter. Plot the obtained sequence to visualize how the estimation behaves.
2. To illustrate the link with the stochastic gradient descent, modify your code to also compute the stochastic gradient descent sequence and its averaged versions. Be careful to only compare what may be compared, *i.e.* for a mini-batch size m , the stochastic gradient descent is updated for each observation while the mini-batch sequence is only updated all m observations, thus $\{\theta_k^{(m)}\}_{k \geq 0}$ has to be compared to $\{\theta_{mk}\}_{k \geq 0}$.
3. Consider again the exercises of the previous practicals and implement the mini-batch approach for each of them. In particular, special attention should be paid to how to get the m observations used by the mini-batch approach. A simple solution is to adapt your existing code by keeping in memory the last m observations and by updating the mini-batch sequence all m steps only.

An example of non-convex optimization

We now consider a variant of the regression model we have already handled. Thus, we use one more time the meteorological data set given by the file `ozone.csv`. A question that naturally arises with regression models is the **relevance** of the involved variables. In other words, we have in mind to restrict our analysis to a subset of size s of the available variables through a **variables selection procedure**. Given an input matrix X of size $n \times (d+1)$ and an output vector $Y = (y_1, \dots, y_n)^\top \in \mathbb{R}^n$, the minimization problem we have to solve amounts to seek $\theta^* \in \mathbb{R}^{d+1}$ that exactly has s non-zero coordinates, formally

$$\theta^* \in \operatorname{argmin}_{\theta \in \mathcal{B}_0(s)} \|Y - X\theta\|^2$$

where we have set

$$\mathcal{B}_0(s) = \left\{ \theta = (\theta_0, \dots, \theta_d)^\top \in \mathbb{R}^{d+1} \quad \text{such that} \quad \#\{i \in \{1, \dots, n\} : \theta_i \neq 0\} = s \right\}.$$

Such a problem is referred as **sparse regression problem**. Although the statement may seem simple at first sight, it is not an obvious problem and there is a vast literature on the topic. It should be noted in particular that the sparse regression problem is known to be a NP-hard problem. From the optimization point of view, our approach based on convex analysis tools comes up against the fact that this problem is not a convex problem because $\mathcal{B}_0(s)$ is **not a convex subset** of \mathbb{R}^{d+1} .

The approach we propose here to deal with sparse regression is called **iterative hard-thresholding** and is based on a **projected stochastic gradient descent procedure**. First, we define the projection onto $\mathcal{B}_0(s)$ as follows. Let $\theta = (\theta_0, \dots, \theta_d)^\top \in \mathbb{R}^{d+1}$, we denote by $\mathcal{J}(\theta)$ the set of the indices of the s largest coordinates of θ . Thus, the projection $p_s(\theta) = (p_0, \dots, p_d)^\top \in \mathbb{R}^{d+1}$ of θ onto $\mathcal{B}_0(s)$ is given by

$$\forall i \in \{0, \dots, d\}, p_i = \begin{cases} \theta_i & \text{if } i \in \mathcal{J}(\theta), \\ 0 & \text{otherwise.} \end{cases}$$

Then, we proceed as in the previous practicals by considering a sequence $\{\sigma_N\}_{N \geq 1}$ of independent uniform random permutations of $\{1, \dots, n\}$ and we define the random functions sequence $\{J_k\}_{k \geq 1}$ by

$$\forall k \geq 1, \theta \in \mathbb{R}^{d+1}, J_k(\theta) = \frac{(y_{\sigma_N(k)} - x_{\sigma_N(k)}^\top \theta)^2}{2} \quad \text{if } k \in \{(N-1)n+1, \dots, Nn\},$$

where, for any $i \in \{1, \dots, n\}$, x_i stands for the i -th row of X . The iterative hard-thresholding sequence $\{\theta_k\}_{k \geq 0}$ is given by $\theta_0 = 0$ and the recursion

$$\forall k \geq 1, \theta_k = p_s(\theta_{k-1} - \gamma_k \nabla J_k(\theta_{k-1}))$$

where we set the step sizes to $\gamma_k = \gamma \times \lceil k/n \rceil^{-\alpha}$ with $\gamma > 0$ and $\alpha \in [0, 1]$.

1. Prove that the subset $\mathcal{B}_0(s) \subset \mathbb{R}^{d+1}$ is not convex.
2. Let $\theta \in \mathbb{R}^{d+1}$, prove that $p_s(\theta)$ is the orthogonal projection of θ onto $\mathcal{B}_0(s)$ in the sense that it satisfies

$$p_s(\theta) \in \operatorname{argmin}_{p \in \mathcal{B}_0(s)} \|\theta - p\|^2.$$

3. Write a Python function `proj` that takes two arguments: a *NumPy* vector `theta` and an integer `s`. This function must return the projection of the vector `theta` onto $\mathcal{B}_0(s)$ as a *NumPy* vector.
4. Implement the iterative hard-thresholding algorithm and apply it to the ozone data set to select $s = 3$ variables among the 11 variables (T6, T9, T12, T15, T18, Ne6, Ne9, Ne12, Ne15, Ne18 and Vx) to explain the ozone variation $Y = \text{maxO3} - \text{maxO3v}$. Tune the parameters γ and α and the number of iterations needed to converge. Visualize the obtained coefficients and discuss how they evolve.
5. What are the selected variables? Adapt your code to always keep the intercept and selecting the s variables only among the relevant ones.
6. Modify your code to also compute the averaged versions of $\{\theta_k\}_{k \geq 0}$ (take care of how you handle the projection, the averaged versions must also belong to $\mathcal{B}_0(s)$) and the mini-batch approach.