Refresher course: Numerical Optimization

F. Iutzeler & J. Malick

Tutorial

A. Differentiability, Minima, and Convexity

◦ **A.1** (Quadratic functions)**.**

a. In $\mathbb{R}^n$, compute the gradient of the squared Euclidean norm $\|\cdot\|_2^2$ at a generic point $x \in \mathbb{R}^n$.

b. Let A be an $m \times n$ real matrix and b a size-m real vector. We define $f(x) = \|Ax - b\|_2^2$. For a generic vector $a \in \mathbb{R}^n$, compute the gradient $\nabla f(a)$ and Hessian $H_f(a)$.

c. Let C be an $n \times n$ real matrix, d a size-n real vector, and $e \in \mathbb{R}$. We define $g(x) = x^{\mathrm{T}}Cx + d^{\mathrm{T}}x + e$. For a generic vector $a \in \mathbb{R}^n$, compute the gradient $\nabla g(a)$ and Hessian $H_g(a)$.

d. Can all functions of the form of f and be written in the form of g? And conversely?

◦ **A.2** (Fundamentals of convexity)**.** This exercise proves and illustrates some results seen in the course.

a. Let f and g be two convex functions. Show that $m(x) = \max(f(x), g(x))$ is convex.

b. Show that $f_1(x) = \max(x^2 - 1, 0)$ is convex.

c. Let f be a convex function and g be a convex, non-decreasing function. Show that $c(x) = g(f(x))$ is convex.

d. Show that $f_2(x) = \exp(x^2)$ is convex. What about $f_3(x) = \exp(-x^2)$

e. Justify why the 1-norm, the 2 norm, and the squared 2-norm are convex.

◦ **A.3** (Strict and strong convexity)**.** A function $f : \mathbb{R}^n \to \mathbb{R}$ is said

- *strictly convex* if for any $x \neq y \in \mathbb{R}^n$ and any $\alpha \in]0, 1[$

$$f(\alpha x + (1 - \alpha)y) < \alpha f(x) + (1 - \alpha)f(y)$$

- *strongly convex* if there exists $\beta > 0$ such that $f - \frac{\beta}{2}\|\cdot\|_2^2$ is convex.

a. For a strictly convex function f, show that the problem

$$\begin{cases} \min f(x) \\ x \in C \end{cases}$$

where C is a convex set admits at most one solution.

b. Show that a strongly convex function is also strictly convex.
(hint: use the identity $\|\alpha x + (1 - \alpha)y\|^2 = \alpha\|x\|^2 + (1 - \alpha)\|y\|^2 - \alpha(1 - \alpha)\|x - y\|^2$.)

c. Let f be a twice differentiable function. Show that f is strongly convex if and only if there exists $\beta > 0$ such that the eigenvalues of $\nabla^2 f(x)$ are larger than β for all x.

d. Discuss the strict and strong convexity of function f_1 and f_2 of A.2.

◦ **A.4** (Optimality conditions)**.** Let $f : \mathbb{R}^n \to \mathbb{R}$ be a twice differentiable function and $\bar{x} \in \mathbb{R}^n$. We suppose that f admits a local minimum at $\bar{x}$ that is $f(x) \geq f(\bar{x})$ for all x in a neighborhood[1] of $\bar{x}$.

a. For any direction $u \in \mathbb{R}^n$, we define the $\mathbb{R} \to \mathbb{R}$ function $q(t) = f(\bar{x} + tu)$. Compute $q'(t)$.

b. By using the first order Taylor expansion of q at 0, show that $\nabla f(\bar{x}) = 0$.

c. Compute $q''(t)$. By using the second order Taylor expansion of q at 0, show that $\nabla^2 f(\bar{x})$ is positive semi-definite.

[1] Formally, one would write $\forall x \in \mathbb{R}^n$ such that $\|x - \bar{x}\| \leq \varepsilon$ for $\varepsilon > 0$ and some norm $\|\cdot\|$.

d. Give a necessary condition on $\nabla^2 f$ for f to be convex. Deduce a condition on C for g to be convex in question b of ○ A.3 and make the connection with question d.

B. Gradient Algorithm

○ **B.1** (Descent lemma). A function $f : \mathbb{R}^n \to \mathbb{R}$ is said to be L-smooth if it is differentiable and its gradient ∇f is L-Lipchitz continuous, that is

$$\forall x, y \in \mathbb{R}^n, \quad \|\nabla f(x) - \nabla f(y)\| \leq L\|x - y\|.$$

The goal of the exercise is to prove that if $f : \mathbb{R}^n \to \mathbb{R}$ is L-smooth, then for all $x, y \in \mathbb{R}^n$,

$$f(x) \leq f(y) + (x - y)^{\mathrm{T}} \nabla f(y) + \frac{L}{2}\|x - y\|^2$$

a. Starting from fundamental theorem of calculus stating that for all $x, y \in \mathbb{R}^n$,

$$f(x) - f(y) = \int_0^1 (x - y)^{\mathrm{T}} \nabla f(y + t(x - y)) \mathrm{d}t$$

prove the descent lemma.

b. Give a function for which the inequality is tight and one for which it is not.

○ **B.2** (Smooth functions). Consider the constant stepsize gradient algorithm $x_{k+1} = x_k - \gamma \nabla f(x_k)$ on an L-smooth function f.

a. Use the *descent lemma* to prove convergence of the sequence $(f(x_k))_k$ when $\gamma \leq 2/L$.

b. Did you use at some point that the function was convex? Conclude about the convergence of the gradient algorithm on smooth non-convex functions.

Lab

I. the Gradient Method

We consider here toy functions that will investigate both theoretically and practically:

$$\begin{array}{rrcl} f: & \mathbb{R}^2 & \to & \mathbb{R} \\ & (x_1,x_2) & \mapsto & 4(x_1-3)^2+2(x_2-1)^2 \\ g: & \mathbb{R}^2 & \to & \mathbb{R} \\ & (x_1,x_2) & \mapsto & \log(1+\exp(4(x_1-3)^2)+\exp(2(x_2-1)^2))-\log(3) \\ r: & \mathbb{R}^2 & \to & \mathbb{R} \\ & (x_1,x_2) & \mapsto & (1-x_1)^2+100(x_2-x_1^2)^2 \\ t: & \mathbb{R}^2 & \to & \mathbb{R} \\ & (x_1,x_2) & \mapsto & (0.6x_1+0.2x_2)^2\left((0.6x_1+0.2x_2)^2-4(0.6x_1+0.2x_2)+4\right)+(-0.2x_1+0.6x_2)^2 \\ p: & \mathbb{R}^2 & \to & \mathbb{R} \\ & (x_1,x_2) & \mapsto & |x_1-3|+2\,|x_2-1|\,. \end{array}$$

○ **I.1** (Some particular functions)**.** Let us investigate the properties of the above functions.

a. From the 3D plots of I.1, which functions are visibly non-convex.
b. For all five functions, show that they are convex or give an argument for their non-convexity.
c. For functions f, g, r, t, compute their gradient.
d. For functions f, g, compute their Hessian.

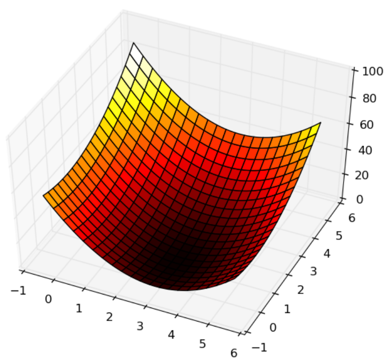
◇ **I.1.** We consider two $\mathbb{R}^2 \to \mathbb{R}$ convex functions with the same global minimizer $(3,1)$ but quite different *shapes* and see how this impacts the performance of gradient-based algorithms. In `1_simple.py` and `2_harder.py`, we aim at minimizing the functions f and g defined above.

a. In `1_simple.py`, fill the function `f` that return $f(x)$ from input vector x. Observe the 3D plot and level plot of the function by uncommenting the lines `custom_3dplot...` and `level_plot...`. Do the same for function `g` in `2_harder.py`.
b. In `1_simple.py`, fill the function `f_grad` that return $\nabla f(x)$ from input vector x. Do the same for function `g_grad` in `2_harder.py`.
c. In `my_gradient.py`, implement a constant stepsize gradient method `gradient_algorithm(f , f_grad , x0 , step , PREC , ITE_MAX )` that takes as an input:
   – `f` and `f_grad`: respectively functions and gradient simulators;
   – `x0`: starting point;
   – `step`: a stepsize;
   – `PREC` and `ITE_MAX`: stopping criteria for sought precision and maximum number of iterations;
   and return `x`, the final value, and `x_tab`, the matrix of all vectors stacked vertically[2].
d. Test your gradient descent function on f and g: i) Verify that the final point is close to the sought minimizer $(3,1)$; ii) observe the behavior of the iterates by uncommenting the line `level_points_plot`. Change the stepsize and give the values for which the algorithm (i) diverges and (ii) oscillates. Compare with theoretical limits by computing the Lipschitz constant of the gradients.

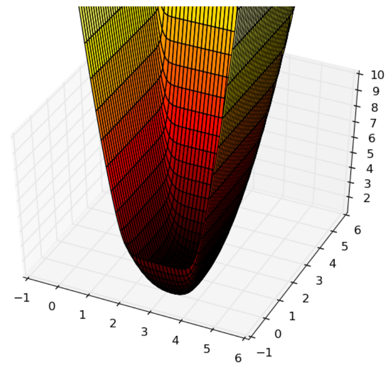
◇ **I.2.** Let us now investigate Newton's algorithm on the same functions f and g.

a. In `1_simple.py`, fill the function `f_grad_hessian` that return $\nabla f(x)$ and $H_f(x)$ from input vector x. Do the same for function `g_grad_hessian` in `2_harder.py`.
c. In `my_gradient.py`, implement Newton's method `newton_algorithm(f , f_grad_hessian , x0 , PREC , ITE_MAX )` that takes as an input:
   – `f` and `f_grad_hessian`: respectively functions and gradient + Hessian simulators;
   – `x0`: starting point;
   – `PREC` and `ITE_MAX`: stopping criteria for sought precision and maximum number of iterations;
   and return `x`, the final value, and `x_tab`, the matrix of all vectors stacked vertically.

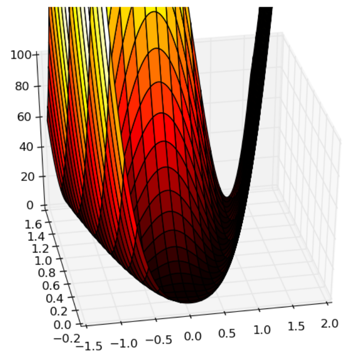
[2] Use the function `vstack` e.g. `x_tab = np.vstack(( x_tab , x ))` .



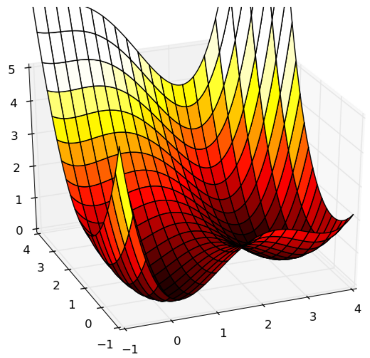
(A) a *simple* function: f



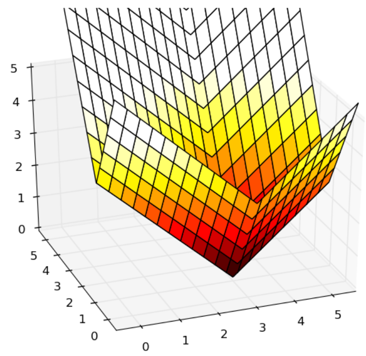
(B) some *harder* function: g



(C) *Rosenbrock*'s function: r



(D) *two pits* function: t



(E) *polyhedral* function: p

FIGURE I.1. 3D plots of the considered functions

d. Test your method on f and g: i) Verify that the final point is close to the sought minimizer $(3, 1)$; ii) observe the behavior of the iterates by uncommenting the line `level_points_plot`.

e. Compare graphically constant stepsize gradient and Newton's algorithms by uncommenting line `level_2points_plot`.

f. Newton's algorithm should take exactly one iteration to converge for function f. Why so? Is it the case for function g?

◇ **I.3.** In `3_rosenbrock.py`, we aim at minimizing the *Rosenbrock* function r.

a. In `3_rosenbrock.py`, fill the functions `r` that return $r(x)$ from input vector x; and `r_grad` that return $\nabla r(x)$ from input vector x. Observe the 3D plot and level plot of the function by uncommenting the lines `custom_3dplot...` and `level_plot...`.
b. Try to minimize r using your constant stepsize gradient function `gradient_algorithm`. Can you find a stepsize for which the algorithm converges?
c. In `my_gradient.py`, implement an *adaptive* stepsize gradient method `gradient_adaptive_algorithm(f , f_grad , x0 , step , PREC , ITE_MAX )` that takes the same inputs and returns the same as the gradient method but implements a *stepsize adaptation method*. For instance, one can use this rule:

$$\begin{aligned}\textbf{if} f(x_{k+1}) > f(x_k) :& \\ x_{k+1} &= x_k \\ step &= step/2\end{aligned}$$

which halves the stepsize if a gradient step makes the functional value increase.
d. Test your method on r: i) Verify that the final point is close to the sought minimizer $(1, 1)$; ii) observe the behavior of the iterates by uncommenting the line `level_points_plot`.
e. In `3_rosenbrock.py`, fill the function `r_grad_hessian` that return $\nabla r(x)$ and $H_r(x)$ from input vector x. Compare the above method with `newton_algorithm`.

◇ **I.4.** In `4_two_pits.py` and `5_poly.py`, we aim at minimizing the functions t and p.
a. Fill the functions and gradient simulators in both files.
b. Test adaptive gradient methods on these functions from different starting points. What do you observe?

II. Application to Regression and Classification

We now get back to the problem of predicting the final grade of a student from various features treated in the Matrix part of the course.

We remind that mathematically, from the $m_{learn} \times (n+1)$ *learning matrix*[3] A_{learn} comprising of the features values of each training student in line, and the vector of the values of the target features b_{learn}; we seek a size-$(n+1)$ *regression vector* that minimizes the squared error between $A_{learn}x$ and b_{learn}. This problem boils down to the following least square problem:

$$\min_{x\in\mathbb{R}^{n+1}} s(x) = \frac{1}{2}\|A_{learn}x - b_{learn}\|_2^2. \tag{II.1}$$

◇ **II.1.** In `1_regression.py`, we minimize the function s to retrieve a predictor.
a. Construct the suitable function and gradient simulators in order to use the `gradient_algorithm` developed in the previous section[4].
b. Compute the Lipschitz constant of the gradient of s. Find a solution to (II.1) using your `gradient_algorithm`. Compare with Numpy's Least Square routine.
c. Construct the gradient + Hessian simulator in order to use your `newton_algorithm`. Compare the execution speed of the classical gradient and Newton algorithm.
d. Generate a random Gaussian matrix/vector couple A, b with increasing size. Create simulators to compare the execution time of constant stepsize gradient, Newton, and pseudo-inverse computation *via* SVD on the least squares problem $\min_x \|Ax - b\|_2^2$. Notably change the *shape* of A from *tall* (nb. of rows >> nb. of cols.) to *fat* (nb. of rows << nb. of cols.).

Binary classification is another popular problem in machine learning. Instead of predicting a numerical value, the goal is now to classify the student into two classes: $+1$ – *pass* i.e. final grade ≥ 10; and -1 – *fail*[5]. To this purpose, we create a class vector c_{learn} from the observation vector b_{learn} by simply setting

[3] $m_{learn} = 300$, $n = 27$
[4] Copy the file `my_gradient.py` in the current folder and add `from my_gradient import *` in the preamble.
[5] Taking $+1$ and -1 instead of $0/1$ for instance simplifies the expression of the cost function.

$c_{learn}(i) = +1$ if $b_{learn}(i) \geq 10$ and -1 otherwise. Then, the most common approach is to minimize the logistic loss:

$$\min_{x\in\mathbb{R}^{n+1}} \ell(x) = \sum_{i=1}^{m_{learn}} \log\left(1 + \exp\left(-c_{learn}(i) a_i^{\mathrm{T}} x\right)\right) \tag{II.2}$$

where a_i^{T} is the i-th row of A_{learn}.

Then, from a solution $x^\star$ of this problem, one can classify a new example, represented by its feature vector a, as such: the quantity $p(a) = \frac{1}{1+\exp(-a^{\mathrm{T}} x^\star)}$ estimates the probability of belonging to class 1; thus, one can decide class $+1$ if for instance $p(a) \geq 0.5$; otherwise, decide class -1.

◇ **II.2.** In `2_classification.py`, we minimize the function ℓ to retrieve a classificator.

a. Compute the gradient of $q(t) = \log(1 + \exp(t))$. Is the function is convex? Deduce that ℓ is convex and its gradient.

b. Construct the suitable function and gradient simulators in order to use your `gradient_algorithm` to minimize ℓ.

c. Find an upper bound for the Lipschitz constant of the gradient of ℓ. *(hint: it is* $0.5\|A\|_2^2$*.)* Compare the constant stepsize gradient with stepsize based on this upper bound and your adaptive gradient.

d. From a final point of the gradient algorithm developed above, generate a decision vector corresponding to the testing set A_{test}. Evaluate the classification error.