Costrained Optimization

Simone chiominto, Francesco Pappone, Giuseppe Intilla

January 2022

The aim of this work is to test the *projected gradient method* for constrained optimization problems. This text first provides the description of the problem, then explains the idea behind this method. Then, our implementation of the method is tested onto the so-called *weighted sphere function* or *hyper-ellipsoid* in dimension $n = 10^d$ with $d = 3, 4, 5$ using both exact derivatives and finite differences to approximate the gradient.

1 Problem Setting

We start with some basic definitions

Definition 1.1 (local minimum)**.** Let $X \subset \mathbb{R}^n$ be a non-empty set and $f : X \to \mathbb{R}$ a function. A point $x^* \in X$ is said to be a *local minimum* for f if there exists $\varepsilon \geq 0$ such that $f(x^*) \leq f(x)$ for all $x \in X$ such that $|x - x^*| < \varepsilon$. A local minimum is said to be *global minimum* if $f(x^*) \leq f(x)$ for all $x \in X$.

Definition 1.2. Let $X \subset \mathbb{R}^n$ be an non-empty, convex, closed set and $f : X \to \mathbb{R}$, the problem

$$\min_{x \in X} f(x)$$

is called a *constrained optimization problem.*

Such as in the unconstrained case, the existence of a global maximum over X is not guaranteed if there are no other hypotheses on f. A necessary condition can be found if f is differentiable on X convex.

Proposition 1.3. *Let X be a non-empty convex set and $f : X \to \mathbb{R}$ be differentiable on X. If x^* is local minimum then*

$$\nabla f(x^*)^T (x^* - x) \geq 0$$

for all $x \in X$.

In addition the previous condition become a sufficient one if f is convex.

2 Projected gradient method

Before introducing the main idea of the method, we define the projection of a point onto a convex set

Proposition 2.1. *Let $X \subseteq \mathbb{R}^n$ be a non empty closed convex set. Then for all $x \in \mathbb{R}^n$ there exists and it is unique $P_X(x) \in X$ such that $min_{y \in X}||y - x|| = ||P_X(x) - x||$. The point $P_X(x)$ is called the* projection *of x onto X.*

Observation 2.2. X closed is required for the existence of a point $P_X(x) \in X$ such that $\min_{y \in X} ||y - x|| = ||P_X(x) - x||$. X convex is necessary for its uniqueness.

let $f : \mathbb{R}^n \to \mathbb{R}$ and $X \subset \mathbb{R}^n$, the *projected gradient method* is an iterative method for finding a solution to the constrained optimization problem

$$\min_{x \in X} f(x).$$

The method revolves around utilizing the projection function P_X over points reached via a regular *gradient descent method*: let x_k be the current point, we first calculate the update suggested by the gradient descent algorithm $x_{k+1} = x_k - \gamma_k \nabla f(x_k)$ and subsequently determine its projection on the set X

$$\bar{x}_{k+1} = P_X(x_{k+1}) = P_X(x_k - \gamma_k \nabla f(x_k))$$

In principle, one could implement the algorithm as it is, since the last equation ensures that the constraint is respected at each point, so that, when the next step is computed, it is by definition still contained in the set and is closer to a minimum than before. In practice, the proper implementation of the method employs a backtracking procedure with the *Armijo condition*: the step is updated with the rule $x_{k+1} = x_k + \alpha_k(\bar{x_k} - x_k)$, where $\alpha \in \mathbb{R}^n$ is determined by a second iteration, starting at $\alpha_k^{(0)} = 1$ accepting or rejecting the proposed $\alpha_k^{(i)} = \beta^{(i)}\alpha_k^{(0)}$ based on the satisfaction of the following (*Armijo*) condition:

$$f(x_k) - f(x_k + \alpha_k^{(i)} \cdot (\bar{x}_k - x_k))$$

with $\beta, \sigma \in (0, 1)$ selected a-priori. Once $\alpha_k^{(i)} \geq \sigma\alpha_k^{(i)}\nabla f(x_k)^t(\bar{(x)}_k - x_K)$ is tested, if it satisfies the previous relation the line search stops and $\alpha_k = \alpha_k^{(i)}$, otherwise $\alpha_k^{(i+1)}$ is tested.

3 Problem description

Our aim is to minimize the function $f : \mathbb{R}^n \to \mathbb{R}$ defined as

$$f(x) = \sum_{i=1}^{n} i x_i^2$$

constrained over the set $X : \{x \in \mathbb{R}^n \; s.t. \; |x_i| \leq 5.12, \;\; \forall i \in \{1, 2, \ldots, n\}\}$ It can be easily proven that X is a closed and convex set, while f is differentiable and convex. The projector function P_X thus exists and is unique, and can be written as

$$P_X(x)_i = \begin{cases} x_i & if \;\; |x| \leq 5.12 \\ 5.12 & if \;\; x \geq 5.12 \\ -5.12 & if \;\; x \leq 5.12 \end{cases}$$

the gradient ∇f is calculated both analytically and estimated via finite differences, for example via the centering approximation given by the formula

$$\nabla f_i \approx \frac{f(x + h \cdot e_i) - f(x - h \cdot e_i)}{2h} = 2ix_i$$

matching exactly the analytical calculation. In the code, we implement naively the previous condition (without expanding the squares in the numerator), so to take into account difference in the h parameter, which could otherwise be simplified.

4 Implementation and Results

Operatively, the projected gradient was implemented with coefficients $\beta = 0.6$, $\sigma = 0.000001$ and $\gamma_t = 0.01 + t$, as we found those to be the best configurations for a quicker convergence. Notice how a low value of σ allows for a more relaxed Armijo condition. As a stopping criterion for the algorithm, we employed a double condition: $|f(x_k) - f(x)| < \epsilon = 10^{-6}$ or $||\nabla f(x)|| < \epsilon = 10^{-6}$, meaning that satisfying even only one of the two is sufficient to stop the iteration. We sampled the starting point x_0 from 10^d-dimensional hypercube side length 12, centered around 0. Notice how, counterintuitively in higher dimensions peripheral volumes (for example, the volume represented by the difference between the constraint X and our hypercube), increase in magnitude (roughly speaking) exponentially. We can see a very intuitive depiction of this phenomenon in Fig.1. This effect is also known as *Curse of dimensionality*. Hence, even though our hypercube could be interpreted as "too close" to the constraint in a geometrical sense, it is actually very likely for the sampled point to be outside of the set individuated by the constraint. The algorithm was implemented in python using the Google Colab service. Our implementation made use of the `Numba` library, which helped speed up the computation of several of the functions we implemented. We ran the iterative algorithm for each of the d, k and derivative combinations, resulting in a total of 36 runs, with the results in time in seconds, value of the function at convergence and number of iterations displayed in Tab. 1. We notice how all of the iterations converge close to the expected minimum, located at $x = 0$ and hence with minimum value of the function over the constraint equal to zero. Moreover, convergence steps seem to vary little with respect to changes to the magnitude of k. Computation using exact derivatives proves to be more than twice as fast as that done with approximated methods. As we can see in Fig.

2a and 2b , the decrease of both the norm of the gradient and the value of the function is extremely quick, likely due to the effect of projection in the first steps.

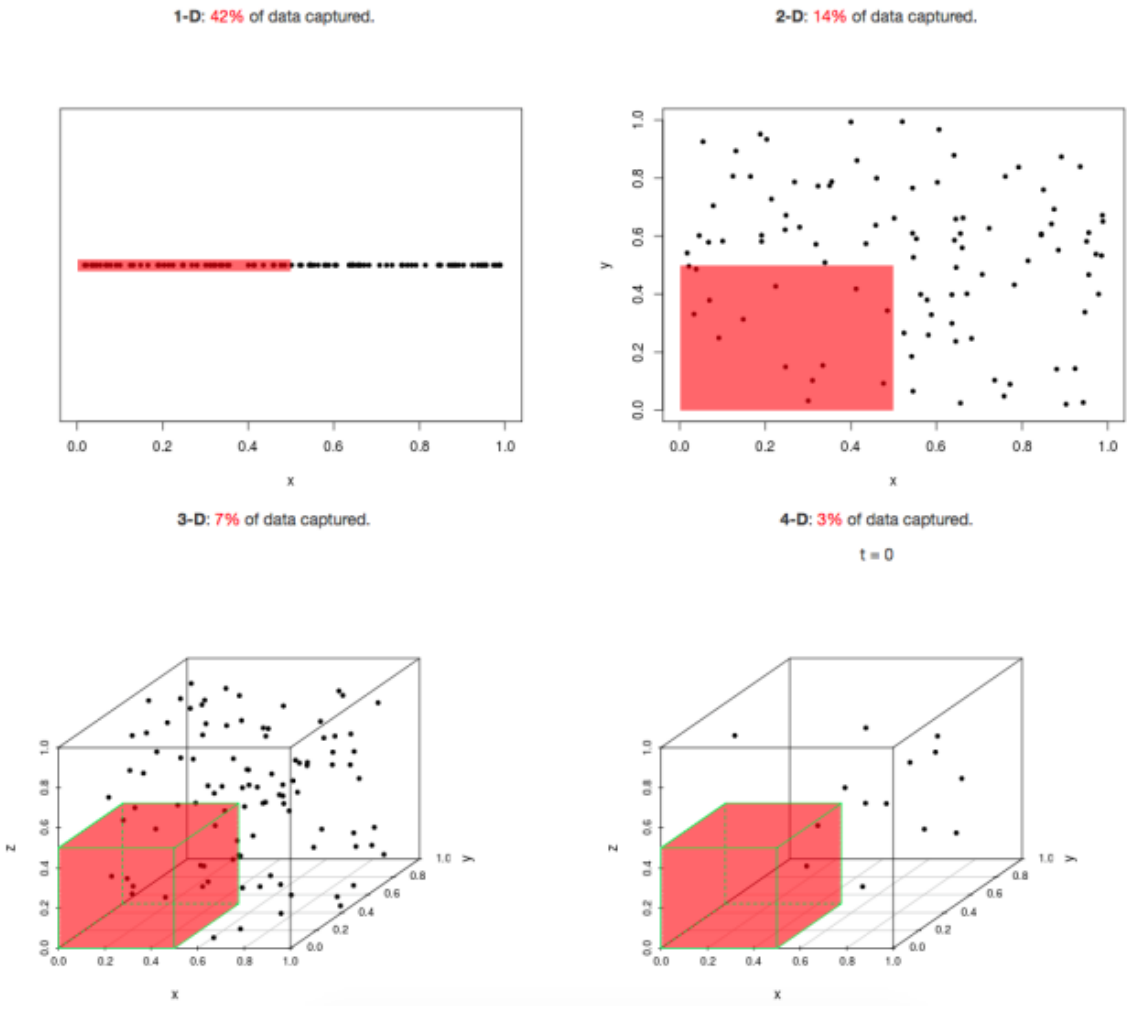
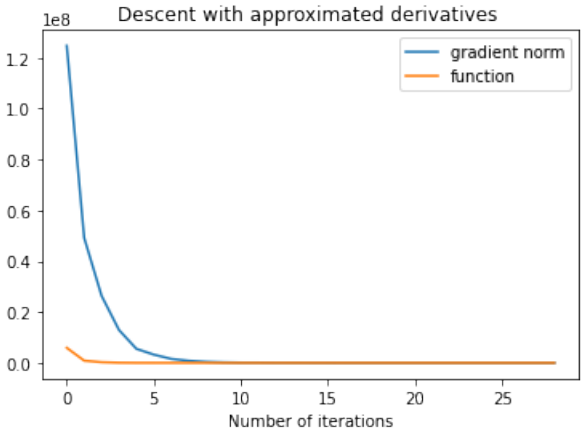
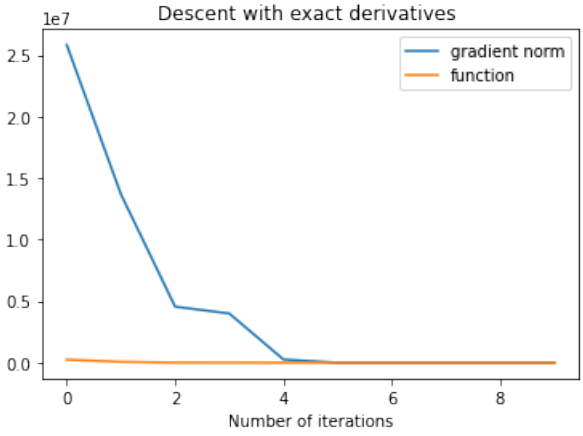


Figura 1: An intuitive depiction of the curse of dimensionality.



(a) Descent with approximated gradient for $d = 5$, $k = 12$



(b) Descent with exact gradient for $d = 5$, $k = 12$

Figura 2

	d	n iter	value of f	time [s]
exact derivatives	d1	20.0	0.00000403	0.03594374
	d2	20.0	0.00000401	0.18737315
	d3	21.0	0.00000245	2.81927402
k=2	d1	28.0	0.00000010	1.30502853
	d2	28.0	0.00000011	0.66371373
	d3	28.0	0.00000010	8.25855213
k=4	d1	28.0	0.00000012	0.13304318
	d2	28.0	0.00000011	0.65645271
	d3	28.0	0.00000011	8.24326828
k=6	d1	28.0	0.00000010	0.12979444
	d2	28.0	0.00000010	0.65034644
	d3	29.0	0.00000015	8.47672837
k=8	d1	28.0	0.00000010	0.12926705
	d2	28.0	0.00000012	0.63903647
	d3	29.0	0.00000018	8.52166820
k=10	d1	29.0	0.00000015	0.12944893
	d2	28.0	0.00000010	0.65253671
	d3	28.0	0.00000010	8.27274917
k=12	d1	29.0	0.00000015	0.13059236
	d2	28.0	0.00000009	0.63413874
	d3	28.0	0.00000011	8.16289491

Tabella 1: Results of the different combinations.

5 Code

```
!pip install numba

from numba import jit #utilising this library to speed up numpy computation
import numpy as np
from numpy import random
import math
import scipy
from scipy.sparse import identity
import matplotlib.pyplot as plt
import timeit
from timeit import default_timer as timer
import pandas as pd

@jit()
def function(x): #definition of our function.
  y=0
  for i in range(n):
    y+=(i+1)*(x[i]**2)
  return y
#Notice how one could technically optimize this by avoiding the for cycle by using np functions.
```

```
#we chose not to do so as the implementation seemed to clash with the @jit decorator, resulting in a net loss in speed.

#returns the gradient, x is where the derivative is calculated,
#the switch allows one to choose finite differences or exact derivatives
@jit()
def derivative  (finite_switch , function, x ):
  h=(10**(-k))*np.linalg.norm(x) #stepsize
  if ( finite_switch ):
    for i in range(n):
      D[i] = (i+1)*((x[i]+h)**2-(x[i]-h)**2)/(2*h) #naive center
     # D[i] = (i+1)*(2*x[i]) #center
      #D[i] = (i+1)*(2*x[i]+h) #forward
    return D
  else:
    for i in range(n):
      D[i]=2*(i+1)*x[i]
    return D

#this function performs the step in the gradient descent
@jit()
def grad_desc_step_finite(stepsize, function , startpoint,der  ): # returns updated point
  stoppoint =  startpoint - stepsize*der
  return stoppoint

#this function projects onto the hypercube defined as the constraint to the problem.
@jit()
def project (x):
  x=np.where(x>5.12,5.12,x)
  x=np.where(x<-5.12,-5.12,x)
  return x

d =0.000001 #is the sigma parameter in the armijo condition
b=0.6 #is the beta parameter
@jit()
def armijo (xbar,x,der):
  a=1
  while((function(x)-function(x+a*(xbar-x))) <= -d*a*np.matmul(np.transpose(der),xbar-x )): #checking for the armijo condition
    a=b*a

  return a

#function calling the projected gradient method.
 #returns number of steps as ctime, value of f calculated in the last x value and computation time in seconds.
 def iterate(swi,k,d):
  #definition of variables and arrays
  global n,D,stoppoint
  n=10**d
  D = np.zeros(shape=(n))
  stoppoint=np.zeros(shape=(n))
  epsilon =0.000001
  delta=delta2=1
```

```
  x =6*np.random.randint(50, size=n)/50
  x = np.where(np.random.rand(n) > 0.5, -x, x)
  grad = np.zeros(shape=(n))
  z = np.zeros(shape=(n))
  ctime=0
  gradient_plot=[]
  function_plot=[]
#iteration of the conjugate gradient algorithm
  while ( delta > epsilon and delta2>epsilon ):
    #pre-calculating values as they're called multiple times
    f=function(x)
    grad=derivative(swi,function,x)

    ctime+=1
    #calculating the projected point
    z=project(grad_desc_step_finite(0.01+ctime,function,x,grad))
    #updating via line-search with backtracking and the Armijo
    condition
    x = x + armijo(z,x,grad)*(z-x)
    #calculating stopping criteria
    delta=np.linalg.norm(grad)
    delta2 = np.max(np.abs((function(x)-f)))

    gradient_plot.append(np.linalg.norm(grad))
    function_plot.append(f)

    print(f)
  return [ctime,f]
```