Report - OPT 201

Implementation of the local SQP algorithm

Thomas Chretienne
Tom Cuel

December 1, 2024

Contents

Introduction

Optimization is central to applied mathematics, engineering and computer science. It covers problems from reducing production costs and improving process efficiency to advancing machine learning and artificial intelligence. The essence of optimization resides in the identification of the optimal solution to a function one seeks to optimize, having constraints to manage at the same time.

The report begins by defining the problem and the data set used for the analysis. We then move on to a theoretical and practical analysis, detailing and using the optimization methods seen in class, such as the KKT method and the SQP algorithm.

The software code used will be Python, which includes a direct solver, so we'll look at the results of the solver directly, then manually via the SQP algorithm. Finally, we compare the predictive performance of these different models, analyzing the results to identify the most effective approaches. With this work, we aim to highlight the strengths and limitations of the different implementation techniques.

1 Optimization Problem

We consider the following problem :

$$(P) \quad \inf_x \left\{ e^{\prod_{i=1}^{5} x_i} - \frac{1}{2}(x_1^3 + x_2^3 + 1)^2 \right\} \quad \text{subject to :} \quad \begin{cases} \sum_{i=1}^{5} x_i^2 & = 10, \\ x_2 x_3 & = 5x_4 x_5, \\ x_1^3 + x_2^3 & = -1. \end{cases}$$

In order to apply the SQP algorithm, we'll put the problem into standard form, so let's put:

$$f(x) = e^{\prod_{i=1}^{5} x_i} - \frac{1}{2}(x_1^3 + x_2^3 + 1)^2$$

$$g(x) = \begin{bmatrix} \sum_{i=1}^{5} x_i^2 - 10 \\ x_2 x_3 - 5x_4 x_5 \\ x_1^3 + x_2^3 + 1 \end{bmatrix}.$$

We can then write our problem in the following form:

$$(PCE) \quad \inf_{\substack{x \in \mathbb{R}^5 \\ g(x)=0}} f(x)$$

2 Automated solving

2.1 Calculations for our problem

Let's now calculate data relevant to our problem, regarding the previous theoretic part.

Gradient of $f(x)$

$$\nabla f(x) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \frac{\partial f}{\partial x_3} \\ \frac{\partial f}{\partial x_4} \\ \frac{\partial f}{\partial x_5} \end{bmatrix} = \mathrm{e}^{\prod_{i=1}^{5} x_i} \cdot \begin{bmatrix} \prod_{\substack{i=1 \\ i\neq 1}}^{5} x_i \\ \prod_{\substack{i=1 \\ i\neq 2}}^{5} x_i \\ \prod_{\substack{i=1 \\ i\neq 3}}^{5} x_i \\ \prod_{\substack{i=1 \\ i\neq 4}}^{5} x_i \\ \prod_{\substack{i=1 \\ i\neq 5}}^{5} x_i \end{bmatrix} + \begin{bmatrix} 3x_1^2(x_1^3 + x_2^3 + 1) \\ 3x_2^2(x_1^3 + x_2^3 + 1) \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Jacobian of $g(x)$

$$Jg(x) = \begin{bmatrix} \frac{\partial g_1}{\partial x_1} & \frac{\partial g_1}{\partial x_2} & \frac{\partial g_1}{\partial x_3} & \frac{\partial g_1}{\partial x_4} & \frac{\partial g_1}{\partial x_5} \\ \frac{\partial g_2}{\partial x_1} & \frac{\partial g_2}{\partial x_2} & \frac{\partial g_2}{\partial x_3} & \frac{\partial g_2}{\partial x_4} & \frac{\partial g_2}{\partial x_5} \\ \frac{\partial g_3}{\partial x_1} & \frac{\partial g_3}{\partial x_2} & \frac{\partial g_3}{\partial x_3} & \frac{\partial g_3}{\partial x_4} & \frac{\partial g_3}{\partial x_5} \end{bmatrix} = \begin{bmatrix} 2x_1 & 2x_2 & 2x_3 & 2x_4 & 2x_5 \\ 0 & x_3 & x_2 & -5x_5 & -5x_4 \\ 3x_1^2 & 3x_2^2 & 0 & 0 & 0 \end{bmatrix}$$

2.2 Problem solving with a Python library

The Python *scipy* library has the *minimize* function that can solve us the problem after we first get the function considered for both the objective function and the constraint, here are the code, obtained here by directly using the results calculated above :

```python
import numpy as np
import time
from scipy.optimize import minimize

# Objective function f(x) = exp(x1 * x2 * x3 * x4 * x5) - 1/2 * (x1^3 + x2^3 + 1)^2
def objective(x):
    x1, x2, x3, x4, x5 = x
    return np.exp(x1 * x2 * x3 * x4 * x5) - 0.5 * (x1**3 + x2**3 + 1)**2

# Gradient of the objective function
def objective_grad(x):
    x1, x2, x3, x4, x5 = x
    # Compute the gradient by hand (derivatives of each term)
    grad = np.zeros(5)
    # Derivatives for exp(x1 * x2 * x3 * x4 * x5)
    exp_term = np.exp(x1 * x2 * x3 * x4 * x5)
    grad[0] = exp_term * (x2 * x3 * x4 * x5) - 1.5 * (x1**2)
    grad[1] = exp_term * (x1 * x3 * x4 * x5) - 1.5 * (x2**2)
    grad[2] = exp_term * (x1 * x2 * x4 * x5)
    grad[3] = exp_term * (x1 * x2 * x3 * x5)
    grad[4] = exp_term * (x1 * x2 * x3 * x4)
    return grad

# Constraints:
# g1(x) = sum xi^2 - 10 (x1^2 + x2^2 + x3^2 + x4^2 + x5^2 = 10)
# g2(x) = x2 * x3 - 5 * x4 * x5 (x2 * x3 = 5 * x4 * x5)
# g3(x) = x1^3 + x2^3 + 1 (x1^3 + x2^3 + 1 = 0)
```

(a) implementing f and its gradient

```python
# Constraints:
# g1(x) = sum xi^2 - 10 (x1^2 + x2^2 + x3^2 + x4^2 + x5^2 = 10)
# g2(x) = x2 * x3 - 5 * x4 * x5 (x2 * x3 = 5 * x4 * x5)
# g3(x) = x1^3 + x2^3 + 1 (x1^3 + x2^3 + 1 = 0)

def constraints(x):
    x1, x2, x3, x4, x5 = x
    # g1 = sum xi^2 - 10
    g1 = sum([xi**2 for xi in x]) - 10
    # g2 = x2 * x3 - 5 * x4 * x5
    g2 = x2 * x3 - 5 * x4 * x5
    # g3 = x1^3 + x2^3 + 1
    g3 = x1**3 + x2**3 + 1
    return [g1, g2, g3]

# Gradient of the constraints
def constraint_eq_grad(x):
    x1, x2, x3, x4, x5 = x
    grad = np.zeros((3, 5))  # 3 constraints, 5 variables
    # Gradient of g1: sum xi^2 - 10
    grad[0, :] = 2 * np.array(x)
    # Gradient of g2: x2 * x3 - 5 * x4 * x5
    grad[1, 1] = x3
    grad[1, 2] = x2
    grad[1, 3] = -5 * x5
    grad[1, 4] = -5 * x4
    # Gradient of g3: x1^3 + x2^3 + 1
    grad[2, 0] = 3 * x1**2
    grad[2, 1] = 3 * x2**2
    return grad
```

(b) implementing g and its gradient

Figure 1: Getting all the objective and constraint related function

```python
# Run the optimization and measure the time spent
start_time = time.perf_counter()

# Initial guess for the optimization
# x0 = [-1.71, 1.59, 1.82, -0.763, -0.763]
x0 = [-1.9, 1.82, 2.02, -0.9, -0.9]
# x0 = [1, 0, 3, 0, 0]

# Define constraints in SciPy's format
constraints_dict = [
    {'type': 'eq', 'fun': lambda x: constraints(x)[0], 'jac': lambda x: constraint_eq_grad(x)[0]},
    {'type': 'eq', 'fun': lambda x: constraints(x)[1], 'jac': lambda x: constraint_eq_grad(x)[1]},
    {'type': 'eq', 'fun': lambda x: constraints(x)[2], 'jac': lambda x: constraint_eq_grad(x)[2]}
]

# Solve the optimization problem using SLSQP
result = minimize(
    objective,
    x0,
    jac=objective_grad,
    constraints=constraints_dict,
    method='SLSQP',
    options={'disp': True, 'maxiter': 500}
)

end_time = time.perf_counter()
elapsed_time = end_time - start_time

print(f"\nThe function took {elapsed_time:.6f} seconds to run.\n")
print("Optimal Solution : ", result.x, "\n")
print(f"Optimal Objective Value : ", result.fun, "\n")
```

Figure 2: Code to execute the algorithm

3 Manual SQP algorithm

3.1 Obtaining the different functions

We have $f(x)$, $\nabla f(x)$, $g(x)$ and $Jg(x)$, now let's calculate the last thing we need : $H_xL(x^k, \lambda^k)$:

$$H_xL(x^k, \lambda^k) = H_xf(x^k) + \sum_{i=1}^{m} \lambda_i H_x g_i(x^k) \quad \text{since} \quad L(x, \lambda) = f(x) + \lambda^T g(x)$$

where :

$$H_f(x) = \left(\frac{\partial^2 f}{\partial x_i \partial x_j} \right)_{\substack{i=1,\dots,n \\ j=1,\dots,n}}$$

We can calculated one or two of the terms but it will take too long to calculated them all, hence why we will use the *sympy* library to do it for us.

$$\frac{\partial^2 f}{\partial x_1^2} = \frac{\partial}{\partial x_1} \left(\mathrm{e}^{\prod_{i=1}^{5} x_i} \times \prod_{\substack{i=1 \\ i \neq 1}}^{5} x_i + 3x_1^2(x_1^3 + x_2^3 + 1) \right)$$

Using the chain rule and product rule, we get:

$$\frac{\partial^2 f}{\partial x_1^2} = \mathrm{e}^{\prod_{i=1}^{5} x_i} \times \left(\prod_{\substack{i=1 \\ i \neq 1}}^{5} x_i \right)^2 + 3\left(2x_1(x_1^3 + x_2^3 + 1) + 3x_1^2\right)$$

And for $j \neq 1$:

$$\frac{\partial^2 f}{\partial x_j \partial x_1} = \mathrm{e}^{\prod_{i=1}^{5} x_i} \times \left(\prod_{\substack{i=1 \\ i \neq j}}^{5} x_i \times \prod_{\substack{i=1 \\ i \neq 1}}^{5} x_i + \prod_{\substack{i=1 \\ i \neq 1 \\ i \neq j}}^{5} x_i \right) + 9x_1^2 x_2^2 \times \mathbb{I}_{j=2}$$

It will take too look to manually write them all, even if every formula does look like the same.
We can calculated the Hessians of the constraints by derivating the Jacobian matrix, we have :

$$H_x g_i(x) = \left(\frac{\partial J g_i}{\partial x_j} \right)_{j=1,\dots,n}$$

The expression are here way simplier than for the objective function, so we can show them here :

$$H_x g_1(x) = \begin{bmatrix} 2 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 2 \end{bmatrix} \quad , \quad H_x g_2(x) = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -5 \\ 0 & 0 & 0 & -5 & 0 \end{bmatrix} \quad \text{and} \quad H_x g_3(x) = \begin{bmatrix} 6x_1 & 0 & 0 & 0 & 0 \\ 0 & 6x_2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

3.2 Theory of the SQP algorithm

The local SQP algorithm for (PCE) is following these different steps :

Local SQP Algorithm

Input: Let $x^0 \in \mathbb{R}^n$ and $\lambda^0 \in \mathbb{R}^m$, set $k = 0$
S.1: If (x^k, λ^k) is a KKT-pair of (PCE): **STOP**
S.2: Solve (QS_k) and let (a^k, ζ^k) denote a KKT-pair of (PQ_k)
S.3: Let $x^{k+1} = x^k + a^k$ and $\lambda^{k+1} = \zeta^k$
S.4: Take $k = k + 1$ and go to [S.1]

As the stopping criterion is hard to verify by computer, in practice one replaces it by

$$\|x^{k+1} - x^k\| \leq \epsilon \quad \text{or} \quad \|\nabla_x L(x^k, \lambda^k)\| \leq \epsilon,$$

where $\epsilon > 0$ is an admissible error.
And, to solve (PQ_k) and the KKT system, we use the Lagrange-Newton algorithm :
Let $(x^0, \lambda^0) \in \mathbb{R}^n \times \mathbb{R}^m$, and define the sequence generated by :

$$\begin{bmatrix} x^{k+1} \\ \lambda^{k+1} \end{bmatrix} = \begin{bmatrix} x^k \\ \lambda^k \end{bmatrix} + \begin{bmatrix} a^k \\ b^k \end{bmatrix},$$

where for each $k \in \mathbb{N}$, $(a^k, b^k) \in \mathbb{R}^n \times \mathbb{R}^m$ solves the system:

$$(\text{LE}_\text{K}) \quad \begin{bmatrix} H_x L(x^k, \lambda^k) & Jg(x^k)^T \\ Jg(x^k) & 0_{m\times m} \end{bmatrix} \begin{bmatrix} a^k \\ b^k \end{bmatrix} = - \begin{bmatrix} \nabla f(x^k) + (\lambda^k)^T Jg(x^k) \\ g(x^k) \end{bmatrix}$$

Let's put :

$$Q(x^k, \lambda^k) = \begin{bmatrix} H_x L(x^k, \lambda^k) & Jg(x^k)^T \\ Jg(x^k) & 0_{m\times m} \end{bmatrix}$$

For $k \in \mathbb{N}$, taking $\zeta^k = \lambda^k + b^k$, the system (LE_K) becomes:

$$(\text{QS}_k) \quad \begin{bmatrix} H_x^L(x^k, \lambda^k) a^k + Jg(x^k)^T \zeta^k \\ g(x^k) + Jg(x^k) a^k + 0 \end{bmatrix} = \begin{bmatrix} 0_{n+m} \end{bmatrix}$$

which actually represents the KKT* system of:

$$(\text{PQ}_k) \quad \inf_{\substack{a \in \mathbb{R}^n \\ g(x^k) + Jg(x^k)a = 0_m}} \frac{1}{2} a^T H_x L(x^k, \lambda^k) a + a^T \nabla f(x^k)$$

When coding, this we can simply find (a^k, b^k) by solving a linear system (LE_k)

3.3 Implementation of the SQP algorithm

```python
class Hand_Made_SQP_Classe:
    def __init__(self):
        # Define symbolic variables
        x1, x2, x3, x4, x5 = symbols('x1 x2 x3 x4 x5')
        self.variables = [x1, x2, x3, x4, x5]

        # Define the symbolic objective function
        self.f_symbols = exp(x1 * x2 * x3 * x4 * x5) - 0.5 * (x1**3 + x2**3 + 1)**2

        # Gradient and Hessian of the objective function
        self.grad_f_symbols = [diff(self.f_symbols, var) for var in self.variables]
        self.hessian_f_symbols = [[diff(grad, var) for var in self.variables] for grad in self.grad_f_symbols]

        # Convert symbolic objective, gradient, and Hessian to numerical functions
        self.f = lambdify(self.variables, self.f_symbols, 'numpy')
        self.grad_f = lambdify(self.variables, self.grad_f_symbols, 'numpy')
        self.hessian_f = lambdify(self.variables, self.hessian_f_symbols, 'numpy')

        # Define constraints
        g1 = x1**2 + x2**2 + x3**2 + x4**2 + x5**2 - 10
        g2 = x2 * x3 - 5 * x4 * x5
        g3 = x1**3 + x2**3 + 1
        self.g_symbols = [g1, g2, g3]

        # Gradient and Hessians of constraints
        self.grad_g_symbols = [[diff(constraint, var) for var in self.variables] for constraint in self.g_symbols]
        self.hessian_g_symbols = [[[diff(diff(constraint, var1), var2) for var2 in self.variables] for var1 in self.variables] for constraint

        # Convert symbolic constraints to numerical functions
        self.g = lambdify(self.variables, self.g_symbols, 'numpy')
        self.grad_g = lambdify(self.variables, self.grad_g_symbols, 'numpy')
        self.hessian_g = lambdify(self.variables, self.hessian_g_symbols, 'numpy')
```

Figure 3: Getting all the functions

We first get all the derivative functions, the main difference between this and what we had before, it's that now it's automated, we only have to put the objective and constraint functions, the gradient, Jocabian and Hessian matrixes are all calculated by this *setup_optimization* function. To do that, we needed some libraries, for the whole project :

```python
import numpy as np
import time
from sympy import symbols, diff, lambdify, exp
```

Figure 4: Used Python libraries

We can then implemente the main loop of the SQP algorithm in Python :

```python
def optimize(self, x_k, lambda_k, use_callback, tol=1e-6, max_iter=500):
    n = np.size(x_k)
    m = np.size(lambda_k)
    nb_iter = 0

    while nb_iter < max_iter:
        # Evaluate objective and constraints
        grad_f = np.array(self.grad_f(*x_k))
        hessian_f = np.array(self.hessian_f(*x_k))
        g = np.array(self.g(*x_k))
        grad_g = np.array(self.grad_g(*x_k))
        hessian_g = np.array(self.hessian_g(*x_k))

        # KKT conditions
        # we're now looking if x_k, lambda_k is a KKT-Pair : gradf(x_k) + Jacg(x_k)lambda_k = 0n, if that's the case, we stop our search
        kkt_lhs = grad_f + np.dot(np.transpose(grad_g), lambda_k)
        if np.linalg.norm(kkt_lhs) < tol and np.linalg.norm(g) < tol:
            break

        # otherwise we solve the following problem
        # find x_solve local optimal solution of the following equation :
        #  Hess_x_lagrangien(x,lambda)x_solve + gradf(x) + Jacg(x)(lambda + lambda_solve) = 0n
        # and g(x) + Jacg(x)(x_solve) = 0m
        # where x_solve, lambda_vals + lambda_solve is a KKT-Pair : gradf(x_solve) + Jacg(x_solve)(lambda_vals + lambda_solve) = 0n

        # Construct Lagrangian Hessian
        lagrangian_hessian = hessian_f.copy()
        for i, lambda_i in enumerate(lambda_k):
            lagrangian_hessian += lambda_i * hessian_g[i]

        # Solve linear system for x_solve and lambda_solve
        A = np.block([
            [lagrangian_hessian, np.transpose(grad_g)],
            [grad_g, np.zeros((m, m))]
        ])
        b = np.hstack([-grad_f - np.dot(np.transpose(grad_g), lambda_k), -g])

        solution = np.linalg.solve(A, b)
        x_solve = solution[:n]
        lambda_solve = solution[n:]

        # Update x_k and lambda_k
        x_k = x_k + x_solve
        lambda_k = lambda_k + lambda_solve
        nb_iter += 1

        # printing the solution at each step if use_callback is True
        if use_callback:
            print(f"Current solution at the iteration n°{nb_iter}: {x_k} and f(x_k) = {self.f(*x_k)}")

        # Check for convergence
        if np.linalg.norm(x_solve) < tol:
            break

    return x_k, self.f(*x_k), nb_iter
```

Figure 5: Main loop to iterate the SQP algorithm

4 Problem reformulation

We can see that according to the last condition : $x_1^3 + x_2^3 = -1$, by implementing it in the function to be minimized : $f(x) = e^{\prod_{i=1}^{5} x_i} - \frac{1}{2}(x_1^3 + x_2^3 + 1)^2$, the problem can be rewritten as follows :

$$\boxed{(P) \quad \inf_x \left\{ e^{\prod_{i=1}^{5} x_i} \right\} \quad \text{subject to :} \quad \begin{cases} \sum_{i=1}^{5} x_i^2 & = 10, \\ x_2 x_3 & = 5 x_4 x_5, \\ x_1^3 + x_2^3 & = -1. \end{cases}}$$

In order to apply the SQP algorithm, we'll put the problem into standard form, so let's put:

$$f(x) = e^{\prod_{i=1}^{5} x_i}$$
$$g(x) = \begin{bmatrix} \sum_{i=1}^{5} x_i^2 - 10 \\ x_2 x_3 - 5 x_4 x_5 \\ x_1^3 + x_2^3 + 1 \end{bmatrix}.$$

We can then write our problem in the following form:

$$\boxed{(PCE) \quad \inf_{\substack{x \in \mathbb{R}^5 \\ g(x)=0}} f(x)}$$

Maybe, it will change the results we obtain and the time taken to compute the algorithm, so we code it by just changing f in the code :

```
class Hand_Made_SQP_Classe_with_Constraints_Changed:
    def __init__(self):
        # Define symbolic variables
        x1, x2, x3, x4, x5 = symbols('x1 x2 x3 x4 x5')
        self.variables = [x1, x2, x3, x4, x5]

        # Define the symbolic objective function
        self.f_symbols = exp(x1 * x2 * x3 * x4 * x5)

        # Gradient and Hessian of the objective function
        self.grad_f_symbols = [diff(self.f_symbols, var) for var in self.variables]
        self.hessian_f_symbols = [[diff(grad, var) for var in self.variables] for grad in self.grad_f_symbols]

        # Convert symbolic objective, gradient, and Hessian to numerical functions
        self.f = lambdify(self.variables, self.f_symbols, 'numpy')
        self.grad_f = lambdify(self.variables, self.grad_f_symbols, 'numpy')
        self.hessian_f = lambdify(self.variables, self.hessian_f_symbols, 'numpy')

        # Define constraints
        g1 = x1**2 + x2**2 + x3**2 + x4**2 + x5**2 - 10
        g2 = x2 * x3 - 5 * x4 * x5
        g3 = x1**3 + x2**3 + 1
        self.g_symbols = [g1, g2, g3]

        # Gradient and Hessians of constraints
        self.grad_g_symbols = [[diff(constraint, var) for var in self.variables] for constraint in self.g_symbols]
        self.hessian_g_symbols = [[[diff(diff(constraint, var1), var2) for var2 in self.variables] for var1 in self.variables] for constraint

        # Convert symbolic constraints to numerical functions
        self.g = lambdify(self.variables, self.g_symbols, 'numpy')
        self.grad_g = lambdify(self.variables, self.grad_g_symbols, 'numpy')
        self.hessian_g = lambdify(self.variables, self.hessian_g_symbols, 'numpy')
```

Figure 6: Change in the code

5 Results

5.1 Comparaison between methods

In a main file, we can compare the results of the three methods, here is the initialisation of the code :

```python
import time
import numpy as np

from auto_SQP_classe import Auto_SQP
from hand_made_SQP_classe import Hand_Made_SQP
from hand_made_SQP_classe_with_constraints_changes import Hand_Made_SQP_Classe_with_Constraints_Changed

# Initial guess for the optimization (x_0, lambda_0)
x_01 = np.array([-1.71, 1.59, 1.82, -0.763, -0.763])
x_02 = np.array([-1.9, 1.82, 2.02, -0.9, -0.9])
# (n=5, m=3 since there is 3 constraints, so 3 lagrange multipliers)
# lambda_0 will be modified in the optimization process, to see how it affects the optimization
lambda_0 = np.zeros(3)

# Load the optimization classes
auto_sqp = Auto_SQP()
hand_made_sqp = Hand_Made_SQP()
hand_made_sqp_with_constraints_changes = Hand_Made_SQP_Classe_with_Constraints_Changed()
```

Figure 7: Initialisation of the code

Then we've made a function to show the result of a method for a given point (x^0, λ^0) :

```python
# Showing a result progression for the by hand optimization
# method = 1 for auto_sqp, 2 for hand_made_sqp, 3 for hand_made_sqp_with_constraints_changes
def showing_a_result(method, x_0, lambda_0, use_callback):
    print()
    # Run the optimization and measure the time spent
    start_time = time.perf_counter()
    result_x, result_f_value, nb_iter = 0, 0, 0
    if method == 1:
        result_x, result_f_value, nb_iter = auto_sqp.solve(x_0, use_callback)
    elif method == 2:
        result_x, result_f_value, nb_iter = hand_made_sqp.optimize(x_0, lambda_0, use_callback)
    elif method == 3:
        result_x, result_f_value, nb_iter = hand_made_sqp_with_constraints_changes.optimize(x_0, lambda_0, use_callback)
    end_time = time.perf_counter()
    elapsed_time = end_time - start_time
    if method == 1:
        print("Auto SQP")
    elif method == 2:
        print("Hand Made SQP")
    elif method == 3:
        print("Hand Made SQP with Constraints Changes")
    print(f"The function took {elapsed_time:.6f} seconds to run.")
    print("Optimal Solution : ", result_x)
    print(f"Optimal Objective Value : ", result_f_value)
    print(f"convergence after {nb_iter} iterations\n")
```

Figure 8: Function to show the result of a method

5.2 Issue for a non found solution

For the point $x^0 = [1, 0, 3, 0, 0]^T$, the algorithm doesn't find a solution, it's because the initial point is not a feasible point, we can see that the constraints are not respected *($g_3(x) = 1 \neq -1$)*, so the algorithm can't find a solution *(but the auto solving methods can)*, here is the output of the code :

```
Traceback (most recent call last):
  File "/Users/tomcuel/Documents/Ensta-Paris/2A/Mathématiques/OPT1/TP3/main.py", line 47, in <module>
    showing_a_result(2, [1, 0, 3, 0, 0], lambda_01, True)
    ~~~~~~~~~~~~~~~~^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/Users/tomcuel/Documents/Ensta-Paris/2A/Mathématiques/OPT1/TP3/main.py", line 30, in showing_a_result
    result_x, result_f_value, nb_iter = hand_made_sqp.optimize(x_0, lambda_0, use_callback)
                                        ~~~~~~~~~~~~~~~~~~~~~~^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/Users/tomcuel/Documents/Ensta-Paris/2A/Mathématiques/OPT1/TP3/hand_made_SQP_classe.py", line 74, in optimize
    solution = np.linalg.solve(A, b)
  File "/Users/tomcuel/Documents/Ensta-Paris/2A/Mathématiques/OPT1/TP3/path/to/venv/lib/python3.13/site-packages/numpy/linalg/_linalg.py", line 413, in solve
    r = gufunc(a, b, signature=signature)
  File "/Users/tomcuel/Documents/Ensta-Paris/2A/Mathématiques/OPT1/TP3/path/to/venv/lib/python3.13/site-packages/numpy/linalg/_linalg.py", line 104, in _raise_linalgerror_singular
    raise LinAlgError("Singular matrix")
numpy.linalg.LinAlgError: Singular matrix
```

Figure 9: Error message for the point $x^0 = [1, 0, 3, 0, 0]^T$

```
Auto SQP
The function took 0.004704 seconds to run.
Optimal Solution :  [-1.  0. -3.  0.  0.]
Optimal Objective Value :  1.0
convergence after 41 iterations
```

Figure 10: Result for the auto solving method for the same point

$x^0 = [-1.71, 1.59, 1.82, -0.763, -0.763]^T$ and $x^0 = [-1.9, 1.82, 2.02, -0.9, -0.9]^T$ are two points that respect the constraints, so the algorithm can find a solution for them, we will focus on those two values for the rest of the results, one for example look like this :

```
Current solution at the iteration n°1: [-1.71644697  1.59488994  1.82858782 -0.76372206 -0.76372206] and f(x_k) = 0.05394646698976075
Current solution at the iteration n°2: [-1.71714261  1.59570867  1.82724803 -0.76364346 -0.76364346] and f(x_k) = 0.05394968227652708
Current solution at the iteration n°3: [-1.71714357  1.59570969  1.82724575 -0.76364308 -0.76364308] and f(x_k) = 0.0539498477698563
Hand Made SQP
The function took 0.000580 seconds to run.
Optimal Solution :  [-1.71714357  1.59570969  1.82724575 -0.76364308 -0.76364308]
Optimal Objective Value :  0.0539498477698563
convergence after 3 iterations
```

Figure 11: Result for the point $x^0 = [-1.71, 1.59, 1.82, -0.763, -0.763]^T$ and $\lambda^0 = [0, 0, 0]$ for the SQP algorithm by hand

5.3 Performance comparaison

Each code will provide some informations thanks to some printing of time measurement methods, function and point values, as well as some plots to compare the results.
Each way to solve the problem give us different times to compute and numbers of iterations made in the algorithm to find the solution, here are the results, for the two points where the hand made algorithm is working :

```
Auto SQP
The function took 0.000421 seconds to run.
Optimal Solution :  [-1.71714162  1.59570743  1.82724938 -0.7636433  -0.7636433 ]
Optimal Objective Value :  0.05394984775938931
convergence after 3 iterations


Current solution at the iteration n°1: [-1.71644697  1.59488994  1.82858782 -0.76372206 -0.76372206] and f(x_k) = 0.05394646698976075
Current solution at the iteration n°2: [-1.71714261  1.59570867  1.82724803 -0.76364346 -0.76364346] and f(x_k) = 0.05394968227652708
Current solution at the iteration n°3: [-1.71714357  1.59570969  1.82724575 -0.76364308 -0.76364308] and f(x_k) = 0.0539498477698563
Hand Made SQP
The function took 0.000546 seconds to run.
Optimal Solution :  [-1.71714357  1.59570969  1.82724575 -0.76364308 -0.76364308]
Optimal Objective Value :  0.0539498477698563
convergence after 3 iterations


Hand Made SQP with Constraints Changes
The function took 0.000255 seconds to run.
Optimal Solution :  [-1.71714361  1.59570974  1.82724568 -0.76364307 -0.76364307]
Optimal Objective Value :  0.05394984735500101
convergence after 2 iterations


Auto SQP
The function took 0.000438 seconds to run.
Optimal Solution :  [-1.71714355  1.59570967  1.82724579 -0.76364308 -0.76364308]
Optimal Objective Value :  0.05394984776332793
convergence after 7 iterations


Hand Made SQP
The function took 0.000515 seconds to run.
Optimal Solution :  [-1.71714362  1.59570975  1.82724567 -0.76364308 -0.76364308]
Optimal Objective Value :  0.05394984426865881
convergence after 6 iterations


Hand Made SQP with Constraints Changes
The function took 0.000417 seconds to run.
Optimal Solution :  [-1.71714347  1.59570958  1.827246   -0.76364312 -0.76364312]
Optimal Objective Value :  0.05394982862375153
convergence after 5 iterations
```

Figure 12: Comparaison of the results for the two starting points

The different methods give us the same point and value of the objective for each point, at a precision of 10^{-5}. We can better see the results in this table concerning time and number of iterations :

$\lambda^0 = [0, 0, 0]$	Auto solving method		Hand made SQP		Constraint change	
	Iterations	Time (μs)	Iterations	Time (μs)	Iterations	Time (μs)
$x^0 = [-1.71, 1.59, 1.82, -0.763, -0.763]^T$	3	421	3	546	2	255
$x^0 = [-1.9, 1.82, 2.02, -0.9, -0.9]^T$	7	438	6	515	5	417

Table 1: Tab of the number of iterations and time for each algorithm

We pretty have the same results for the two points between all methods to find the solution and the objective function value, just obtained differently. The optimization problem is too simple for us to see any difference here, but we can still see that the constraint change is faster because it demands less loops while the auto version, with all the constraints, is a bit slower, yet faster than the full by hand version. We can try to see if there is any differences for a fixed x point and by making λ^0 vary, here are the codes and the results :

```python
# making a comparaison between the 3 classes for x_01 lambda_0 that changes
def means_for_method(method, x_0, lambda_array):
    mean_time = 0
    mean_iter = 0
    mean_x = np.zeros(5)
    for lambda_0 in lambda_array:
        start_time = time.perf_counter()
        result_x, result_f_value, nb_iter = 0, 0, 0
        if method == 1:
            result_x, result_f_value, nb_iter = auto_sqp.solve(x_0, False)
        elif method == 2:
            result_x, result_f_value, nb_iter = hand_made_sqp.optimize(x_0, lambda_0, False)
        elif method == 3:
            result_x, result_f_value, nb_iter = hand_made_sqp_with_constraints_changes.optimize(x_0, lambda_0, False)
        end_time = time.perf_counter()
        elapsed_time = end_time - start_time
        mean_time += elapsed_time
        mean_iter += nb_iter
        mean_x += result_x
    mean_time /= len(lambda_array)
    mean_iter /= len(lambda_array)
    mean_x /= len(lambda_array)
    print()
    if method == 1:
        print("Auto SQP")
    elif method == 2:
        print("Hand Made SQP")
    elif method == 3:
        print("Hand Made SQP with Constraints Changes")
    print(f"Mean time : {mean_time}")
    print("Mean x : ", mean_x)
    print(f"Mean iterations : {mean_iter}")

def make_comparaison(x_0):
    nb_max_in_each_direction = 1 # above that there is some exponential overflow issues
    nb_in_each_direction = 10
    nb_values = nb_in_each_direction * nb_in_each_direction * nb_in_each_direction
    lambda_array = [np.zeros(3) for i in range(nb_values)]
    for i in range(nb_in_each_direction):
        for j in range(nb_in_each_direction):
            for k in range(nb_in_each_direction):
                lambda_array[i * nb_in_each_direction * nb_in_each_direction + j * nb_in_each_direction + k] = np.array([i * nb_
    means_for_method(1, x_0, lambda_array)
    means_for_method(2, x_0, lambda_array)
    means_for_method(3, x_0, lambda_array)

make_comparaison(x_01)
```

Figure 13: Code to compare the results for different λ^0

Yet, we're limited since the exponential function is too high for the values we can put in the exponential, and we have this type of message :

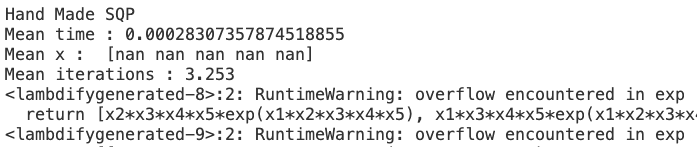

Figure 14: Error message for the overload of the exponential function that messed up the results

There are still lambda values that allow us to try to see some differences, those ones are implemented in the code, and we can see the results here :

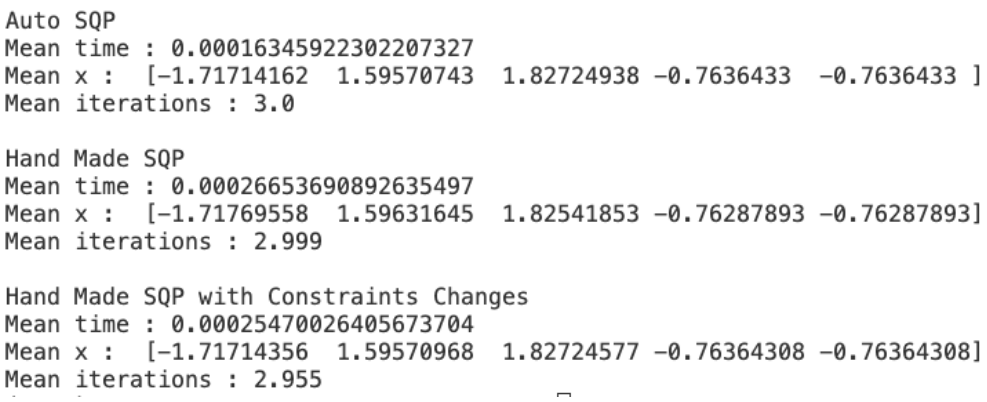

Figure 15: Comparaison of the results for the two starting points

We can see that the results are pretty much the same, the Auto SQP methods is a bit faster than the Hand made one, both the one with the constraint change and the one without it. The constraint change is still faster than the full by-hand version, but the difference is not that big of a deal.

Conclusion

The implementation of the local SQP algorithm has provided a comprehensive framework for solving constrained optimization problems. Through the development of theoretical foundations and practical implementations, this project has demonstrated the effectiveness of the SQP approach in dealing with complex, non-linear problems.

Automated solutions, implemented using Python's scipy library, offered a simple yet powerful benchmark, while manual implementation of the SQP algorithm highlighted the detailed computational steps required to solve the optimization problem iteratively. Drawing on theoretical results such as KKT conditions and gradient calculations, we validated the performance of our algorithm.

Overall, this project has highlighted the strengths and potential limitations of SQP, notably its sensitivity to initial assumptions and its computational cost for high-dimensional problems.