Lecture 2

Constrained minimization problem in Python

Classical problem of optimization methods

Mathematical programming problem:

$$f(x, y) \rightarrow max$$
$$g(x, y) \leq C$$
$$x, y \geq 0$$

Constrained minimization problem

Let us consider the problem of nonlinear programming in general form:

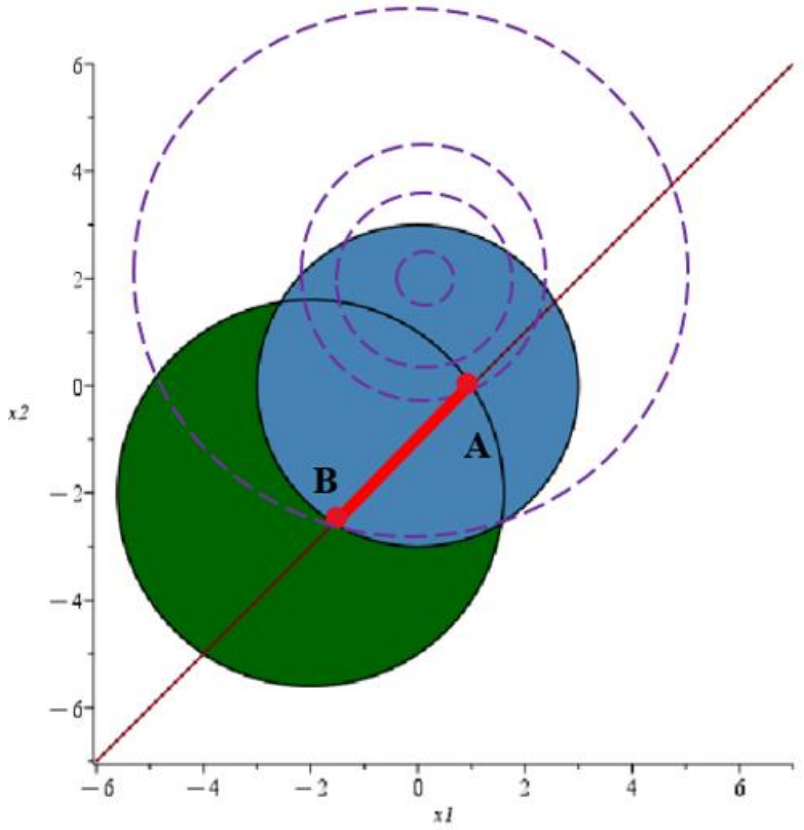
$$f(x) \to \min(\max)$$
$$g_i(x) \le 0, i = 1, \dots, m$$
$$h_i(x) = 0, i = 1, \dots, k$$

For example, task:

$$f(x) = x_1^2 + x_2^2 - 4x_2 \to min$$
$$\begin{cases} x_1^2 + x_2^2 \le 9 \\ x_1^2 + 4x_1 + x_2^2 + 4x_2 \le 5 \\ x_1 - x_2 = 1 \end{cases}$$

Constrained minimization problem. Graphic solution

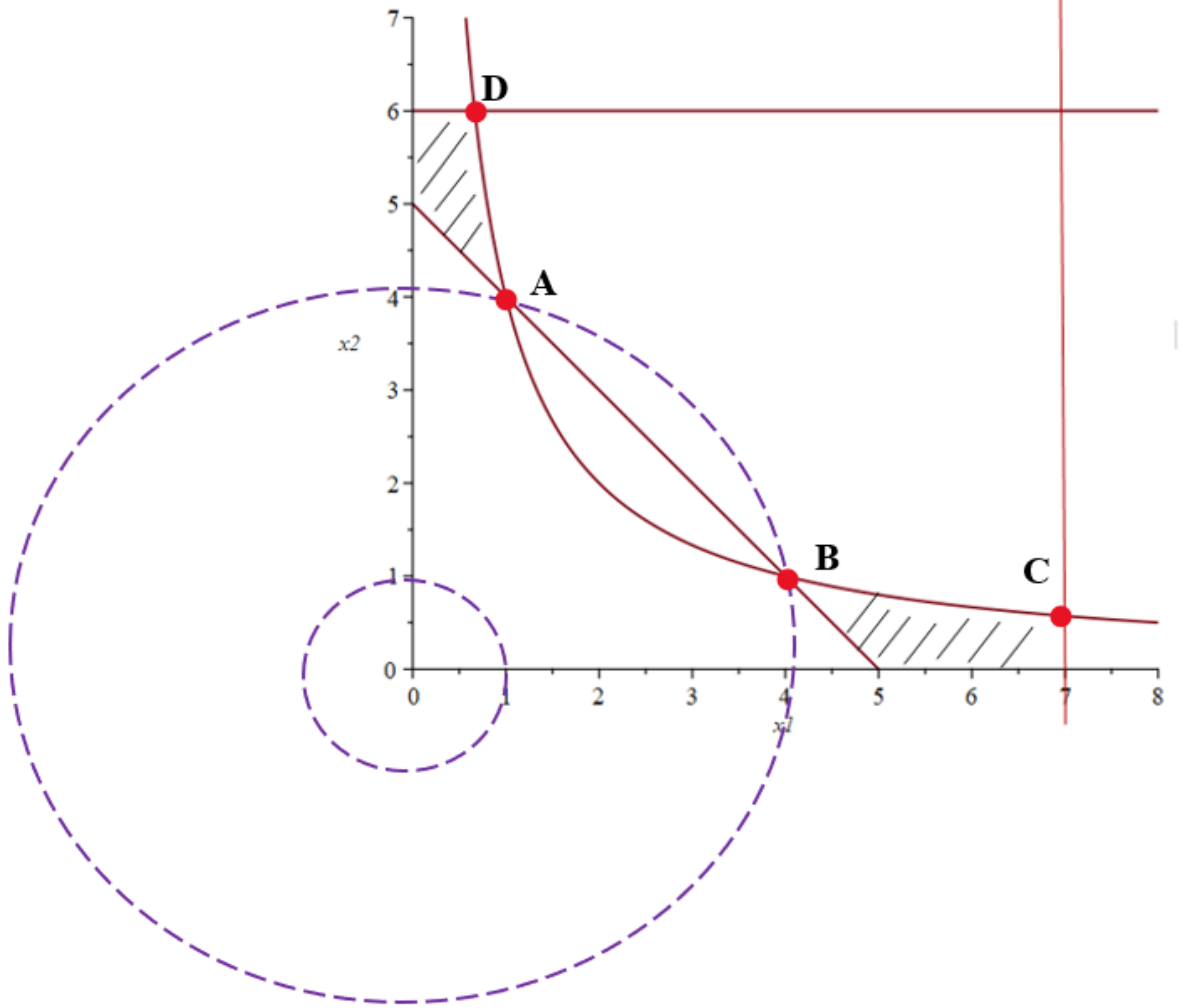
In the case of two variables, the problem can be solved graphically. It is necessary to show the constraints and level lines of the objective function



Constrained minimization problem. Graphic solution

$$f(x) = x_1^2 + x_2^2 \to min$$

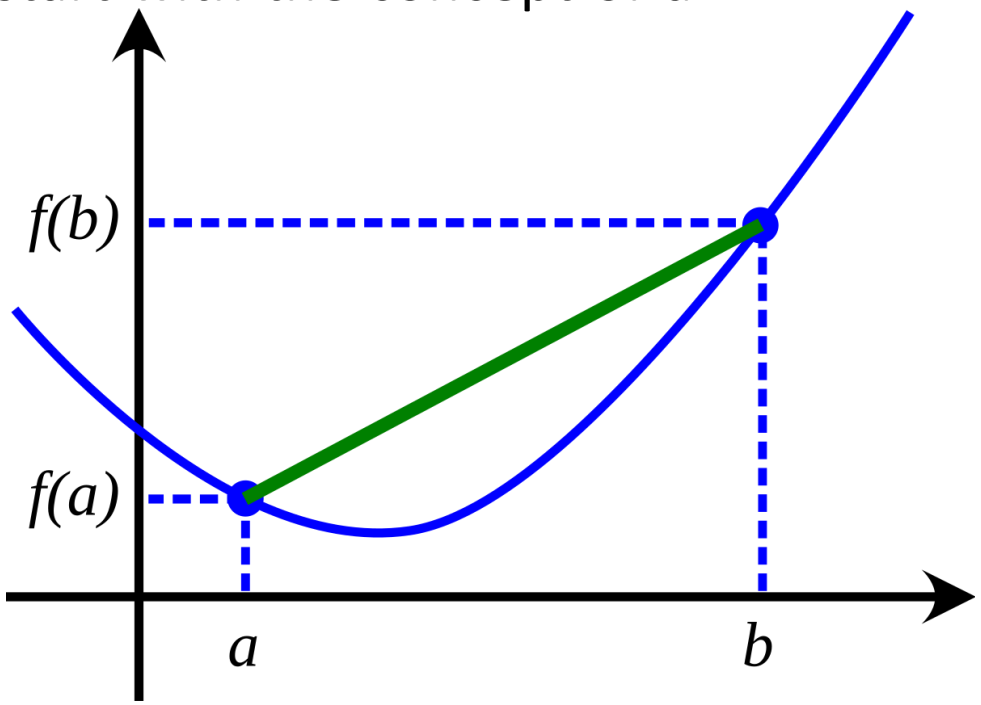
$$\begin{cases} x_1 x_2 \le 4 \\ x_1 + x_2 \ge 5 \\ x_1 \le 7 \\ x_2 \le 6 \\ x \ge 0 \end{cases}$$



Convex functions. Testing a function for convexity

Nonlinear programming problems with convex functions are well studied and belong to the section of convex programming, so let's start with the concept of a convex function.

In mathematics, a real-valued function is called **convex** if the line segment between any two distinct points on the graph of the function lies above or on the graph between the two points



Convex functions. Testing a function for convexity

Algorithm for checking a function for convexity.

1. Construct the Hessian matrix for the function in question.
2. Calculate the principal minors of the Hessian matrix. If all principal minors are non-negative, then conclude that the function is convex.

The principal minor of the *k*-th order is the determinant composed of elements of a matrix of quadratic form located at the intersection of *k* rows and *k* columns of the given matrix with the same numbers.

Convex programming problem

$$f(x) \to min$$

$$\begin{cases} g_i(x) \leq 0, i = 1, \dots, m \\ x \geq 0 \end{cases}$$

where $f(x)$ and $g_i(x), i = 1, \dots, m$ are convex

The **Lagrange function** of a convex programming problem is the function

$$L(x, y) = f(x) + \sum_{i=1}^{m} g_i(x)\lambda_i$$

$where \lambda_i, i = 1, \dots, m -$ Lagrange multipliers

Convex programming problem

A point $(x^*, \lambda^*), x^* \geq 0, \lambda^* \geq 0$ is called a **saddle point of the Lagrange function** if:

$$L(x^*, \lambda) \leq L(x^*, \lambda^*) \leq (x, \lambda^*), \forall x, \lambda \geq 0$$

1st Kuhn-Tucker theorem. If (x^*, λ^*) is a saddle point of the Lagrange function, then x^* is an optimal solution to the convex programming problem.

2nd Kuhn-Tucker theorem. For any optimal solution x^* of a regular convex programming problem, there exists a vector of Lagrange multipliers λ^*: (x^*, λ^*) is a saddle point of the Lagrange function.

Classification of optimization problems

Classification of optimization problems:

-Unconstrained optimization;
-LP (linear programming);
-MILP (mixed integer linear programming);
-NLP (nonlinear programming);
-MINLP (mixed integer nonlinear programming)

SciPy

Contains a large set of functions for scientific computing, including tools for solving optimization problems, which are located in the **scipy.optimize** module.

This module provides methods for solving both nonlinear programming (NLP) and linear programming (LP) problems, including mixed integer linear programming (MILP) problems.

SciPy

scipy.optimize module includes the implementation of the following routines:

-Constrained and unconstrained minimization of scalar functions of several variables (minim) using various algorithms (Nelder-Mead simplex, BFGS, conjugate gradient Newton, COBYLA and SLSQP);
-Global optimization (e.g.: basinhopping, diff_evolution);
-Minimization of residuals of least squares (least_squares) and nonlinear curve fitting algorithms of least squares (curve_fit);
-Minimization of scalar functions of one variable (minim_scalar) and root finding (root_scalar);
-Multivariate solvers of a system of equations (root) using various algorithms (hybrid Powell, Levenberg-Marquardt or large-scale methods such as Newton-Krylov).

SciPy. Solving the MILP problem

To solve linear programming problems, the optimize submodule has a **linprog** function. HiGHS is used by default as a solver for LP(MILP) — it implements the simplex method (highs-ds) and the interior point method (highs-ipm). When starting the solution, one of the methods is selected by default; the ability to force a method is also present.

$$f(x) = 3x_1 + 3x_2 \to \min$$
$$x_1 + x_2 \leq 8$$
$$2x_1 - x_2 \geq 1$$
$$x_1 - 2x_2 \leq 2$$
$$x_1 \geq 0, x_2 \geq 0$$

SciPy. Solving the MILP problem

```
from scipy.optimize import linprog

c = -np.array([3., 3.])
A_ub = np.array([[1., 1.],[-2., 1.],[1., -2.]])
b_ub = np.array([8.0,-1.0,2.0])

bounds = [(0, np.inf), (0, np.inf)]
res_milp = linprog(c=c, A_ub=A_ub, b_ub=b_ub, bounds=bounds, method='highs')

print(f"Solution: x = {list(np.round(res_milp['x'], 2))}, f = {-res_milp['fun']}, {res_milp['message']}")
```

```
Solution: x = [3.0, 5.0], f = 24.0, Optimization terminated successfully. (HiGHS Status 7: Optimal)
```

SciPy. Solving the MILP problem

The scipy.optimize.linprog library provides several methods for solving linear programming problems. Each of them has its own features, advantages, and limitations. Here are the main methods:

1. ‘highs’

Advantages:

High performance.

Suitable for large and complex problems.

Supports both minimization and maximization problems.

Disadvantages:

Requires installation of additional dependencies (included in the standard scipy package since version 1.6.0).

Recommendations: Use this method by default if you have a modern version of scipy.

SciPy. Solving the MILP problem

2. 'simplex’

Description: The classic simplex method implemented in scipy. This is one of the first methods available in linprog.

Advantages:

Simple and robust.

Works well for small problems.

Disadvantages:

Slower than modern methods (e.g. highs).

May not handle large problems.

Recommendation: Use for small problems or when a classical approach is required.

SciPy. Solving the MILP problem

3. 'interior-point’

Description: The interior point method. This method solves a linear programming problem by approaching the optimal solution from within a feasible region.

Advantages:

Fast convergence for large problems.

Works well for problems with many variables and constraints.

Disadvantages:

May be less accurate for problems with singular solutions.

Requires more memory than the simplex method.

Recommendation: Use for large problems where speed is important.

SciPy. Solving the MILP problem

4. 'revised simplex’

Description:

An improved version of the simplex method that uses more efficient computational approaches.

Advantages:

More efficient than the classic simplex method.

Suitable for medium-sized problems.

Disadvantages:

Slower than the interior point method for large problems.

Recommendations: Use for medium-sized problems where accuracy is important.

Conditional optimization method="trust-constr"

The **trust-constr method** in the minimize function of the **scipy.optimize** library is a constrained optimization method based on trust-regions. It is designed to solve optimization problems with constraints (both linear and nonlinear) and can work with both continuous and discrete variables.

Conditional optimization method="trust-constr"

Main features of the trust-constr method:

Support for constraints.

The method can work with constraints of different types

Suitable for problems with both smooth and non-smooth functions.

Can work with a large number of variables and constraints.

The method provides high accuracy of the solution, especially for problems with non-linear constraints.

Conditional optimization method="trust-constr"

$$f(x) = 3x_1 + 3x_2 \to \min$$

$$x_1 + x_2 \leq 8$$

$$2x_1 - x_2 \geq 1$$

$$x_1 - 2x_2 \leq 2$$

$$x_1 \geq 0, x_2 \geq 0$$

Conditional optimization method="trust-constr"

$$f(x) = 3x_1 + 3x_2 \to \min$$
$$x_1 + x_2 \le 8$$
$$2x_1 - x_2 \ge 1$$
$$x_1 - 2x_2 \le 2$$
$$x_1 \ge 0, x_2 \ge 0$$

```
from scipy.optimize import Bounds
bounds = Bounds ([0, 1e9], [0, 1e9])
import numpy as np
from scipy.optimize import minimize
from scipy.optimize import LinearConstraint
linear_constraint = LinearConstraint ([[1,1], [-2,1],[1,-2]], [8,-1,2])

[38] def objective(x):
        x1, x2 = x
        return 3 * x1 + 3 * x2

    x0 = np.array([0.5, 0])

    res = minimize(objective, x0, method='trust-constr',
                   constraints=[linear_constraint],
                   options={'verbose': 1}, bounds=bounds)

    print(res.x)
```

```
Number of iterations: 360, function evaluations: 1077, CG iterations: 350, optimality: 2.65e-14, constraint violation: 1.00e+09, execution time: 0.49 s.
[1.49999979 2.9411815 ]
```

CVXPY. Overview

CVXPY - this package was implemented to solve convex optimization problems. To solve the problem, it is necessary to perform several steps: define variables, set the objective function and constraints to form the object of the optimization problem. After the problem is formulated, before running the solver, the convexity of the objective function and constraints is checked using the **DCP** (Disciplined Convex Programming) rules.

$$f(x) = x_1^2 + x_2^2 - 2x_1 \to min$$
$$\begin{cases} x_1^2 + 4x_2^2 - 4x_1 - 4x_2 \leq 0 \\ x_1 + x_2 \geq 4 \\ x \geq 0 \end{cases}$$

```python
import cvxpy as cp

x = cp.Variable()
y = cp.Variable()

objective = cp.Minimize(x**2 + y**2-2*x)

constraints = [
    x**2+4*y**2-4*x-4*y <= 0,
    x+y >= 4,
    x >= 0
]

assert objective.is_dcp(), "Objective function is not convex"
for constraint in constraints:
    assert constraint.is_dcp(), "Constraint is not convex"

problem = cp.Problem(objective, constraints)
problem.solve()

print("Minimal value:", problem.value)
print("Optimal x and y:", x.value, y.value)
```

```
Minimal value: 3.500000000367045
Optimal x and y: 2.5000000056865748 1.4999999944357738
```

Transport problem

The transportation problem is a linear programming problem and its purpose is to determine the most economical plan for transporting goods from points of departure (warehouses) to points of destination (stores).

Let there be m warehouses $A_1, A_2, \dots, A_m$ from which goods should be transported to n stores $B_1, B_2, \dots, B_n$. Let $c_{ij}, i = 1, \dots, m, j = 1, \dots, n$ be the cost of transporting one unit of cargo from the *i*-th warehouse to the j-th store (transportation tariff). Let $a_i, i = 1, \dots, m$ be the cargo stock at the *i*-th warehouse; $b_j, j = 1, \dots, n -$ be the cargo demand at the *j*-th store; $x_{ij}, i = 1, \dots, m, j = 1, \dots, n -$ be the number of cargo units transported from the *i*-th departure point to the *j*-th destination point. The matrix $X = (x_{ij})_{m \times n}$ is called the transportation plan. The matrix $C = (c_{ij})_{m \times n}$ is called the tariff matrix.

Transport problem

As the optimality criterion, we will choose the minimum cost of transporting the entire cargo. Thus, **to solve the transport problem** is to determine the transportation plan $x_{ij}, i = 1, \dots, m, j = 1, \dots, n$, that is, the amount of cargo that should be transported from the i-th departure point to the j-th destination point so that all departure points get rid of cargo reserves, and all destination points receive the required amount of cargo, and at the same time the cost of transporting the entire cargo is the lowest.

Transport problem

The mathematical model of the transport problem will take the form:

$$F = \sum_{i=1}^{m} \sum_{j=1}^{n} c_{ij} x_{ij} \rightarrow min$$

$$\sum_{i=1}^{m} x_{ij} = b_j, j = 1, \dots, n$$

$$\sum_{j=1}^{n} x_{ij} = a_i, i = 1, \dots, m$$

$$x_{ij} \geq 0, i = 1, \dots, m, j = 1, \dots, n$$

Transport problem. Example

Warehouses	Stores				Stocks
	B_1	B_2	B_3	B_4	
A_1	2 x_{11}	3 x_{12}	4 x_{13}	2 x_{14}	140
A_2	8 x_{21}	4 x_{22}	1 x_{23}	4 x_{24}	160
A_3	9 x_{31}	7 x_{32}	3 x_{33}	6 x_{34}	120
Needs	150	90	100	80	

Warehouses	Stores				Stocks
	B_1	B_2	B_3	B_4	
A_1	2 x_{11}	3 x_{12}	4 x_{13}	2 x_{14}	140
A_2	8 x_{21}	4 x_{22}	1 x_{23}	4 x_{24}	160
A_3	9 x_{31}	7 x_{32}	3 x_{33}	6 x_{34}	120
Needs	150	90	100	80	

```
Optimal solution: [140.   0.   0.   0.   0.  90.  70.   0.  10.   0.  30.  80.]
Optimal value of the objective function: 1370.0
```

```python
from scipy.optimize import linprog

# objective function coefficients
c = [2, 3, 4, 2, 8, 4, 1, 4, 9, 7, 3, 6]


stocks = [140, 160, 120]
A_ub = [
    [1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1]
]


needs = [150, 90, 100, 80]
A_eq = [
    [1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0],
    [0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0],
    [0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0],
    [0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1]
]

result = linprog(c, A_ub=A_ub, b_ub=stocks, A_eq=A_eq, b_eq=needs)

print("Optimal solution:", result.x)
print("Optimal value of the objective function:", result.fun)
```