EL2700 - Model Predictive Control

Formulating and solving linear and quadratic programs in Python

September 2, 2024

Computer Exercise 1

Pedro Roque, Gregorio Marchesini and Mikael Johansson

Division of Decision and Control Systems
School of Electrical Engineering and Computer Science
Kungliga Tekniska Högskolan

Overview

This computer exercise aims to introduce you to linear and quadratic programming problems and how they can be solved in Python. It assumes that you are familiar with Python, the scientific libraries `numpy` and `scipy`, as well as the basic functionality for plotting in `matplotlib`.

The exercise has the following intended learning outcomes:

1. You should know what a linear program (LP) and quadratic program (QP) are, and understand how they can be formulated in matrix form and solved.
2. You should understand how an algebraic modeling language, such as `cvxpy` simplifies the definition and solution of optimization problems.
3. You should be able to formulate and solve finite-horizon optimal control problems in `cvxpy`.

To make sure that you are comfortable with defining and manipulating matrices in `numpy`, we suggest that you first run the exercises in `operations.py`.

Background on linear and quadratic programming

A linear programming problem (LP) is an optimization problem where the goal is to find values of the decision variables $x_1, x_2, \ldots, x_n$ that minimize a linear cost function under linear constraints on the decisions. Mathematically, we can write an LP on the form

$$\begin{array}{ll} \text{minimize} & c_1x_1 + c_2x_2 + \cdots + c_nx_n \\ \text{subject to} & a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \quad \leq b_1 \\ & \vdots \\ & a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \leq b_m \end{array}$$

Here, $c_1, c_2, \ldots, c_n$ are constant coefficients that define the objective function, a_{ij} are constant coefficients that define the constraints, and $b_1, b_2, \ldots, b_m$ are the upper bounds for each constraint. In this formulation, linear equality constraints can be handled by expressing them as a two inequalities. For instance, the equality constraint:

$$a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n = b_i$$

can be written as the two inequalities

$$a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n \leq b_i$$

$$a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n \geq b_i$$

Linear programming problems can be expressed more concisely in vector and matrix notation. By introducing

$$x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}, \quad c = \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix}, \quad A = \begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ \vdots & & & \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{pmatrix}, \quad b = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}$$

we can write the linear programming problem on the form

$$\begin{array}{ll} \text{minimize} & c^\top x \\ \text{subject to} & Ax \leq b \end{array}$$

This is how linear programs are formulated and solved using the `linprog` command in `scipy.optimize`.

Linear programming problems are always convex, irrespective of the values of A, b and c. In the absence of constraints, the optimal value of a linear programming problem typically goes to $-\infty$, because the objective function can be reduced indefinitely by letting each decision variable x_i have the opposite sign of c_i and increasing its magnitude. When the constraints limit the set of feasible decisions, the optimal solution is always found at the boundary of the feasible region formed by these constraints.

Task 1 : MPC Airline

MPC Airline (MPCA) is planning how many tickets should be sold from first class and second class to maximize profit. Due to airplane configurations, MPCA needs to sell at least 20 tickets from first class and 35 tickets from second class. Knowing that MPCA has a profit of 2000 SEK for second class tickets and 1500 SEK for first class tickets, and its airplanes can take a maximum of 130 passengers, how many tickets of each kind should be sold to maximize the profit?

(a) Introduce the decision variable x_1 to describe the number of first-class tickets, and x_2 to describe the number of second-class tickets. Define the objective function that allows you to maximize the total profit, and the constraints that define the minimal value for each travel class and the maximum number of tickets that the airline can sell.

(b) Write the linear programming problem in matrix form.

(c) In Python, import `linprog` from `scipy.optimize` and read its documentation to understand how you can use it to solve linear programs. Solve the ticketing problem above. What are the optimal values of x_1 and x_2?

A quadratic program (QP) is an optimization problem where the goal is to minimize a quadratic cost function subject to linear constraints on the decision variables. QPs can be written in matrix form as

$$\begin{array}{ll} \text{minimize} & x^\top P x + 2q^\top x + r \\ \text{subject to} & Ax \leq b \end{array}$$

QPs are convex (and thereby easy to solve) if the matrix P is positive semidefinite. The optimal value of a quadratic program can also be unbounded. For example,

$$x_1^2 + 2x_2$$

can be made arbitrarily negative by setting $x_1 = 0$ and letting x_2 negative and increasing its magnitude. However, if P is positive definite, then the cost is finite, and the optimum is either the minimizer of the (unconstrained) objective function, or located at the boundary of the feasible set. If you formulate your optimization problem as a QP on matrix form, you can solve it using `quadprog` in `scipy.optimize`.

Formulating and solving convex optimization problems in `cvxpy`

While the approach that we have used to formulate and solve LPs and QPs in the previous section is mathematically rigorous, it can be tedious and error-prone, particularly for complex problems with numerous variables and constraints. To address these challenges, algebraic modeling languages (AMLs) have been developed. AMLs provide a high-level, user-friendly way to formulate optimization problems. Instead of manually defining matrices, AMLs allow users to express optimization problems in a more natural and readable form, closely resembling the mathematical expressions one might write on paper. This abstraction significantly reduces the potential for errors, improves readability, and allows for easier modifications and extensions of the problem.

One popular algebraic modeling language in Python is `cvxpy`, a powerful and flexible library designed for convex optimization. It allows users to define optimization problems using intuitive mathematical expressions, making the process of defining, solving, and analyzing optimization problems much simpler and more efficient.

Defining an Optimization Problem in `cvxpy`

To define and solve an optimization problem in `cvxpy`, you can use the following steps.

1. First, you need to import the `cvxpy` library.

```
import cvxpy as cp
```

2. In `cvxpy`, you define decision variables using the `Variable` class.

```
x = cp.Variable() # For a single variable
x = cp.Variable(n) # For a vector of n variables
```

3. In `cvxpy`, you can define the objective using the `Minimize` or `Maximize` classes.

```
objective = cp.Minimize(c.T @ x) # For minimizing c^T * x
```

4. Constraints are expressed as a list of inequalities or equalities.

```
constraints = [A @ x <= b]
```

5. Once you have the objective and constraints, you create an optimization problem.

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

6. To find the optimal solution, you call the `solve` method on the problem object.

```
problem.solve()
```

7. After solving the problem, you can retrieve the optimal values of the decision variables and the optimal value of the objective function.

```
optimal_value = problem.value
optimal_solution = x.value
print("Optimal value:", optimal_value)
print("Optimal solution:", optimal_solution)
```

You can try these steps with the matrices A, b and c that you defined in the first task. Of course, if you have already defined A, b and c as matrices and vectors, the benefits of using `cvxpy` (or any other algrabraic modeling language) are rather limited. The true power of `cvxpy` shines through when you use it to define decision variables, objectives and constraints directly in their natural and readable form. The listing below shows how the first task can be defined and solved in `cvxpy`.

```
import cvxpy as cp

# Define decision variables
first_class_tickets = cp.Variable()
second_class_tickets = cp.Variable()

# Define the objective function
profit = 1500 * first_class_tickets + 2000 * second_class_tickets
objective = cp.Maximize(profit)

# Define the constraints
constraints = [
    first_class_tickets >= 20,
    second_class_tickets >= 35,
    first_class_tickets + second_class_tickets <= 130
]

# Formulate the optimization problem
problem = cp.Problem(objective, constraints)

# Solve the problem
problem.solve()

# Retrieve the optimal solution
print("Optimal profit:", problem.value)
print("Optimal number of first-class tickets:", first_class_tickets.value)
print("Optimal number of second-class tickets:", second_class_tickets.value)
```

Task 2 : Optimal control of moving masses

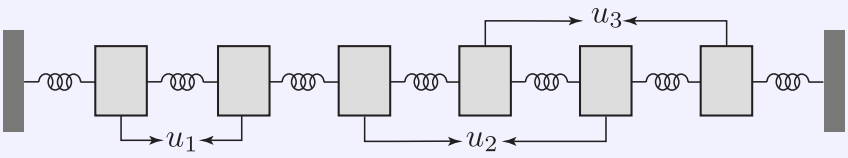

Figure 1: Moving masses in Task 2

The system in Figure 1 consists of six unit masses connected by springs to each other, and to walls on either side. There are three actuators, which exert tensions between certain pairs of masses. Its dynamics can be described by

$$\dot{x}(t) = \begin{pmatrix} 0 & I \\ A_{21} & 0 \end{pmatrix} x(t) + \begin{pmatrix} 0 \\ B_2 \end{pmatrix} u(t), \qquad y(t) = \begin{pmatrix} I & 0 \end{pmatrix} x(t)$$

where

$$A_{21} = \begin{pmatrix} -2 & 1 & 0 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 & 0 \\ 0 & 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 0 & 1 & -2 \end{pmatrix}, \text{ and } B_2 = \begin{pmatrix} 1 & 0 & 0 \\ -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{pmatrix}$$

The first six states are the velocities of the masses, and the last states are their displacements from their equilibrium positions. The actuators can exert a maximum force of ± 0.5, and the displacements of the masses cannot exceed ± 4.

(a) Use the `numpy` and `control` packages to define matrices A_c, B_c and C_c, along with a state-space system object for

$$\dot{x}(t) = A_c x(t) + B_c u(t), \qquad y(t) = C_c x(t)$$

Use a sampling time of $h = 0.5$ seconds and the `c2d` command to create a corresponding state-space description for the discrete-time dynamics

$$x_{t+1} = Ax_t + Bu_t, \qquad y_t = Cx_t$$

(b) Pose the problem of minimizing the energy required to transfer the masses from $x_0 = \begin{pmatrix} 0_{6\times 1}, & 2 \cdot \mathbf{1}_{6\times 1} \end{pmatrix}$ to $x_T = 0_{12\times 0}$ as an optimal control problem on the standard form. Solve the problem using `cvxpy`. Plot the state trajectories and the optimal input against time. What is the minimum energy $\sum_{t=0}^{T-1} \|u_t\|_2^2$ required for the state-transfer?

(c) Plot the minimal input energy against horizon length for $T \in [30, \ldots, 100]$. For readability, transform your code in (b) into a function `(u_opt, x_opt)=openLoopOptimalControl(A,B,C,ulim,ylim,T,x0,xT)` that returns the optimal control and state trajectories. Loop over T, compute the optimal input sequences, determine their energy cost, and plot the result.