Data-driven Mathematical Optimization with Python

Krzysztof Postek Alessandro Zocca Joaquim Gromicho Jeff Kantor

Version of April 3, 2022

Contents

Chapter 1

Mathematical optimization

1.1 A motivating example – A glass of water

Alice receives a single long-stemmed rose, but at home she only has a lemonade glass of cylindrical shape where she can put it. When Alice places the rose in the glass, it tumbles. Alice realizes that adding water will help to keep the rose more stable. How much water should she add? Intuitively one would say fill up the glass, but that makes it just as unstable as it was in the beginning. There should be a ideal level of water that makes the glass with the rose the most stable. We know that the stability of an object is closely related to the height of its center of gravity, so we will analyze this quantity depending on the water level.

The empty glass has a diameter of 4 (cm), height 20 (cm) and mass $m_g = 100$ (gr) and, ignoring the mass at its bottom, the height h_g (in cm) of center of gravity of the glass is at half of its total height, thus $h_g = 10$. By homogeneity, a body of water that reaches height x (in cm) when poured into the glass will has its center of gravity at height $h_w(x) = x/2$ and a total mass $m_w(x) = \pi r^2 h_w(x) = 4\pi h_w(x)$ (in gr), where we used the fact that 1 cubic cm of water weights 1 gr.

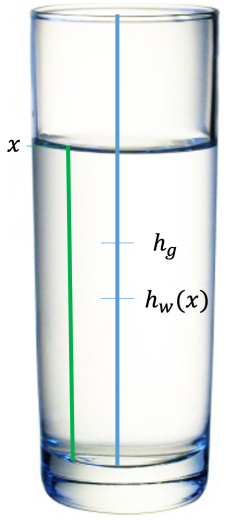


Figure 1.1: The lemonade glass with the water level x and the resulting center of mass for the water at $h_w(x)$.

The height $h(x)$ of the center of mass of the system glass-and-water does depend on the water level x and can be determined using the equation $m_g h_g + m_w(x)h_w(x) = (m_g + m_w(x))h(x)$. Solving for $h(x)$ we obtain

$$h(x) = \frac{\pi x^2 + 500}{2\pi x + 50},$$

which is plotted below as a function of x.

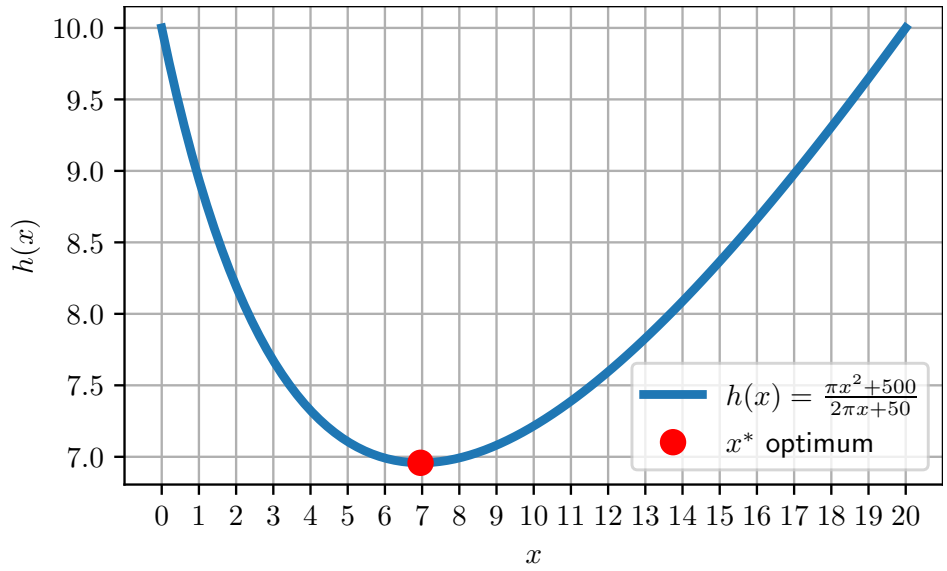


Figure 1.2: The height $h(x)$ of the center of gravity as a function of the water level x and its minimum.

As we can infer from the plot there is an optimum height x^* of the water that brings the center of gravity as low as possible. We can find it by mathematically solving the following constrained minimization problem,

$$\begin{aligned} \min \quad & h(x) \\ \text{s.t.} \quad & 0 \leq x \leq 20, \end{aligned}$$

where the **objective function** is the height $h(x)$ and we added a **constraint** $0 \leq x \leq 20$ since the water level is nonnegative and cannot exceed the glass height.

Using standard calculus techniques, we can search *analytically* for such a minimum by finding the zero(s) of the first derivative of the function $h(x)$, i.e. by solving

$$h'(x) = \frac{\pi\left(\pi x^2 + 50x - 500\right)}{2(\pi x + 25)^2} = 0.$$

Of the two solutions obtained by solving the quadratic equation at the numerator, only one lies in the interval $[0, 20]$ as desired, namely

$$x^* = \frac{5\sqrt{25 + 20\pi} - 25}{\pi} \approx 6.958,$$

hence just above one third of the glass height, that is $20/3 \approx 6.667$. Inspecting the value of the second derivative of $h(x)$ at the point x^* reveals that $h''(x^*) > 0$, confirming that x^* is indeed a minimum. The water level $x^* \approx 6.958$ is, using knowledge of physics, the one likely to make the glass most stable out of all possibilities.

The above was our starter example of formulating and solving a mathematical optimization problem. It involved making certain assumptions (about the center of gravity of the glass) about the model itself, thanks to which the problem was simple enough that we were able to find the best solution using high-school calculus techniques.

In the next section, we will discuss the key steps of modelling mathematical optimization problems in more detail.

1.2 What is mathematical optimization?

Mathematical optimization is used to to mathematically describe our decision problem and then, algorithms, to solve them. The models consist of three components:

- **decision variables** which correspond to actions or choices that we have to make in our decision problem: the height of water in the lemonade glass, or in other problems, whether to open a new manufacturing facility, which supply routes to use, or for which prices we should sell our products.

- **objective function** which is used to evaluate a specific solution (i.e. a specific choice of values for the decision variables introduced earlier). For the objective function, a 'goal' should be specified, either **maximize** or **minimize** it – in the glass example, we aimed to minimize the height of the center of gravity. In other applications, the objective can be to minimize operational costs or to maximize the number of satisfied customers.
- **constraints** that restrict the possible values of our decision variables, i.e., conditions that must be satisfied – in the glass example, the water level had to be non-negative and not higher than 20cm. Other common examples of constraints are to require that a maximum allowed budget is satisfied, that all demand of important customers is met, or that the capacities of warehouses are not exceeded. The constraints define the **feasible region** of our model, i.e., the set of all solutions that meet the constraints.

The goal of an optimization model is to find a **global optimum**: a feasible selection of the decision variable's values leading to the best objective function value. Despite being very simple, the motivating example described in Section 1.1 already includes all the key ingredients of a mathematical optimization problem.

Applied mathematical optimization requires three types of skills, which can be related to three fundamental questions:

- **What to model?** First, the modeler must translate a problem from the real world to an abstracted mathematical representation. Not every aspect of the real world can or should be taken into account by the model, so there are many choices to be made in this first step, which typically have a significant impact on the model and solution approach.
- **How to model?** There can be multiple equivalent model formulations. Conceptually, equivalent models solve the same optimization problem, but the applicable solution approaches or computational complexity may differ.
- **How to interpret the model's solution?** After solving the model, the solution has to be evaluated and translated back to the original real world problem

These three aspects should be treated as a continuous process, not as sequential steps. For example, if the final solution turns out to be impractical, we need to adjust the model. If certain desired properties cannot be modeled efficiently, perhaps we should re-evaluate what to include in the model. A mathematical model is a tool, not a goal (well, except for mathematicians to study). A model is "always flawed" and our challenge is to make a model useful.

Mathematically, we can describe optimization problems as follows. Given an objective function $f : X \to \mathbb{R}$ to be minimized, with $X \subseteq \mathbb{R}^n$ being the set of **feasible set** of candidate solutions we seek to find $x \in X$ satisfying the following condition $f(\boldsymbol{y}) \geq f(\boldsymbol{x}) \quad \forall\, \boldsymbol{y} \in X$, .i.e., that the solution we find is at least as good as all other possible solutions. Similarly, we can define a **maximization problem** by changing the last condition into $f(\boldsymbol{y}) \leq f(\boldsymbol{x}) \quad \forall\, \boldsymbol{y} \in X$. In both cases we refer to such solutions as **optimal solutions**. The general way to formulate a minimization problem is:

$$\begin{aligned} \min \quad & f(\boldsymbol{x}) \\ \text{s.t.} \quad & \boldsymbol{x} \in X, \end{aligned}$$

and similarly for a maximization problem. Different types of functions f and sets X lead to different types of optimization problems and to different solution techniques. Mathematical optimization has a long history, see [Kit11] for a nice overview of highlights in the history of optimization and [KIG14] for an overview of optimization techniques.

In what follows, we shall work through another example, in which we shall clearly indicate the various steps of the modelling process.

1.3 Example: Pop-up shop

Betty has been offered an opportunity to operate a pop-up shop to sell a unique commemorative item for each event held at a famous location. The items Betty would be selling cost 12 € each and she will be able to sell them for 40 €. Unsold items can be returned to the supplier but Betty will receive only 2 € due to their commemorative nature.

Parameter	Symbol	Value
sales price	r	40 €
unit cost	c	12 €
salvage value	w	2 €

It is clear the more Betty sells the more profit she will earn. Demand for these items, however, will be high only if the weather is good. Historical data suggests three typical scenarios, namely $S = \{sunnyskies, goodweather, poorweather\}$, as detailed in the following table.

Scenario (s)	Probability (p_s)	Demand (d_s)
sunny skies	0.1	650
good weather	0.6	400
poor weather	0.3	200

The dilemma, of course, is that the weather will not be known until after the order is placed. Ordering enough items to meet demand for a good weather day results in a financial penalty on returned goods if the weather is poor. On the other hand, ordering just enough to satisfy the demand on a poor weather day leaves "money on the table" for good weather days. What choice should Betty make?

A naive solution is to place an order equal to the expected demand, which can be calculated as

$$\mathbb{E}(D) = \sum_s p_s d_s = 0.1 \cdot 650 + 0.6 \cdot 400 + 0.3 \cdot 200 = 365.$$

By ordering exactly this amount of items, a quick analysis shows an average profit of 8,339 € , as detailed by the following table.

Scenario (s)	Demand (d_s)	Probability (p_s)	Ordered (x)	Sold (y_s)	Salvage ($x - y_s$)	Profit (f_s)
sunny skies	650	0.30	365	365	0	10,220 €
good weather	400	0.60	365	365	0	10,220 €
poor weather	200	0.10	365	200	165	3,950 €
Average profit						8,339 €

In particular, no scenario shows a loss, which appears to be a satisfactory outcome. However, can Betty find an order resulting in a higher expected profit?

Let x be a non-negative number representing the number of items that will be ordered, and y_s be the non-negative variable describing the number of items sold in scenario s in the set S comprising all scenarios under consideration. Given the constants defined in the table above, the profit f_s for scenario $s \in S$ can then be written as

$$f_s = \underbrace{ry_s}_{\text{sales revenue}} + \underbrace{w(d_s - y_s)}_{\text{salvage value}} - \underbrace{cx}_{\text{order cost}}.$$

The expected profit $\mathbb{E}(F)$ is given by $\mathbb{E}(F) = \sum_s p_s f_s$. Operationally, y_s can be no larger the number of items ordered, x, or the demand under scenario s, d_s. Putting these facts together, the problem to be solved is

$$\begin{aligned}
\max \quad & \mathbb{E}(F) = \sum_{s \in S} p_s f_s & \\
\text{s.t.} \quad & f_s = ry_s + w(d_s - y_s) - cx & \forall s \in S \\
& y_s \leq x & \forall s \in S \\
& y_s \leq d_s & \forall s \in S \\
& y_s \in \mathbb{Z}_+ & \forall s \in S \\
& x \in \mathbb{Z}_+. &
\end{aligned}$$

We can implement this problem in Pyomo as follows.

```python
# price and scenario information
r = 40
c = 12
w = 2
scenarios = {
    "sunny skies" : {"demand": 650, "p": 0.1},
    "good weather": {"demand": 400, "p": 0.6},
    "poor weather": {"demand": 200, "p": 0.3},
}

# create model instance
m = pyo.ConcreteModel('Pop-up shop')

# set of scenarios
m.S = pyo.Set(initialize=scenarios.keys())

# decision variables and their domain
m.x = pyo.Var(within=pyo.NonNegativeReals)
m.y = pyo.Var(m.S, within=pyo.NonNegativeReals)
m.f = pyo.Var(m.S)

# objective
@m.Objective(sense=pyo.maximize)
def EV(m):
    return sum([scenarios[s]["probability"]*m.f[s] for s in m.S])

# constraints
@m.Constraint(m.S)
def profit(m, s):
    return m.f[s] == r*m.y[s] + w*(m.x - m.y[s]) - c*m.x

@m.Constraint(m.S)
def sales_less_than_order(m, s):
    return m.y[s] <= m.x

@m.Constraint(m.S)
def sales_less_than_demand(m, s):
    return m.y[s] <= scenarios[s]["demand"]

# solve
solver = pyo.SolverFactory('glpk')
results = solver.solve(m)
```

The results of the optimization are shown in the following table.

Scenario (s)	Demand (d_s)	Probability (p_s)	Ordered (x)	Sold (y_s)	Salvage ($x - y_s$)	Profit (f_s)
sunny skies	650	0.30	400	400	0	11,200 €
good weather	400	0.60	400	400	0	11,200 €
poor weather	200	0.10	400	200	200	3,600 €
Average profit						8,920 €

The results of the optimization show that increasing the order to 400 items will increase the expected profit by 581€ to a level of 8,920€. In poor weather there will be more items returned to the supplier and a lower profit, but that is more than compensated by the increased profits in good weather conditions.

1.4 Pyomo: Modelling and solving optimization problems in Python

The problem in the previous section can be still considered, by standards of industrial applications, a toy problem because it involves only four variables. Large problems can involve thousands to millions of variables and constraints. Obviously thus, real-life problems are modelled and solved using computers.

A computer, however, is not able to model the problem by itself – it is the human who learns to understand the relationships between different decisions, and the constraints and (possibly multiple) objectives to be met. Optimization modelling therefore, is a marriage of human ingenuity and processing powers of computers when solving the problem.

For that reason, it is important that the programming language/tool in which we formulate optimization problems, most closely resembles the way we would think of. For that reason, the modelling language we chose to illustrate the material of this book is Pyomo, a cross-platform and open-source collection of Python software packages for implementing optimization models.

For most of the examples we present in this book, we include `Python` code snippets which, besides Pyomo, make use of a number of packages. These packages are supposed to be installed and imported prior to being used. We refer to these packages using aliases that are common, as follows:

```
import pandas as pd
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
from io import StringIO
import pyomo.environ as pyo
import pyomo.gdp as gdp
```

The full code for all the examples can be found in a dedicated GitHub repository. This companion set of notebooks contains also a Pyomo style guide, followed by the code snippets presented in the book.

A good comprehensive guide to Pyomo is the book [Har+17]. Pyomo's open source software is hosted on GitHub. In particular, examples used in the aforementioned book [Har+17] are available in this GitHub directory. The full Pyomo documentation is available online or in pdf. Additional online resources are:

- a slide deck from a 2018 Pyomo Tutorial workshop;
- a Pyomo gallery illustrating seven classical optimization problem and their implementation;
- a collection of online examples from the Pyomo software repository;
- a Pyomo Cookbook created by Prof. J. Kantor;
- Pyomo-related questions on Stack Overflow.

Pyomo is a modelling package and on its own, it cannot solve the optimization problems we pass to it. Instead, it is able to use multiple solvers – independent pieces of software that solve specific types of optimization problems. Figure 1.3 illustrates the relationship between the user, the modelling package and the solver. The user utilizes Pyomo to formulate an optimization problem in a way that is most convenient for a human to conceptualize. Pyomo, in turn, translates this optimization model into a form that is most convenient for a specific solver. The solver then find an optimal solution to the problem, and sends it back to Pyomo which presents it to the user.

Solvers can be compared to various numerical linear algebra or machine learning libraries in Python – some of them excel at certain tasks, while others do better on other types of tasks. There are multiple commercial and open-source solvers available and in order to practice optimization, one needs to be able to understand the most important differences among them, and read the solution output they provide. As this is so important, throughout the book we will also give guidance how to do exactly this.

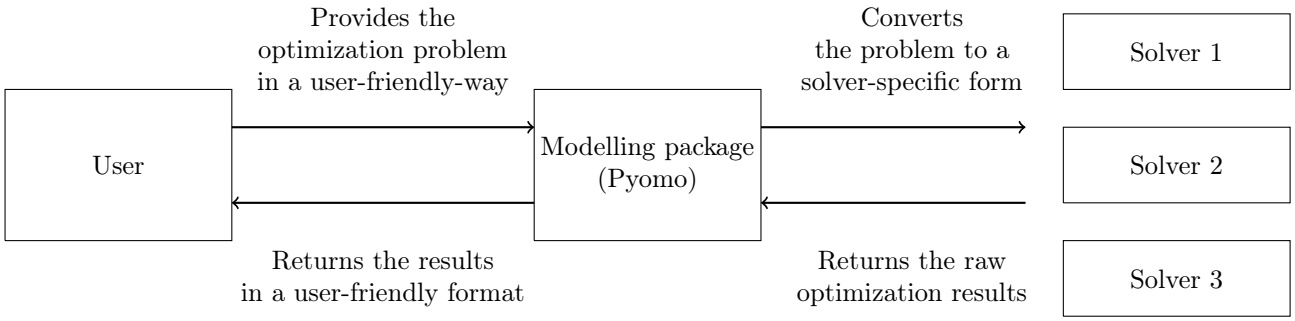


Figure 1.3: Relationship between the user, modelling package and the solver.

1.5 What to expect from this book

This book will cover both the basic theoretical as well as applied (numerical) aspects of modelling optimization problems. We find this combination to be extremely important.

First, because optimization is a domain of mathematics that has grown mostly with specific applications in mind, presenting it without the applied angle misses the point.

Secondly, understanding the mathematics underlying the optimization tools makes the user more able to choose the right approach, interpret the solutions, and to create own optimization tools.

For these reasons, we will be gradually building up our mathematical understanding and ability to build up optimization problems, from the simplest to more complicated ones. Next to that, we will be showing how to formulate them using Pyomo, solve them using the suitable solvers, and how to interpret the obtained solutions.

To combine these two smoothly, at the end of each chapter, we will present a complete example in which a real-life situation is translated into a model, indicating each time the modelling decisions we make. Also, we will show how to interpret the output provided by the solvers to the optimization models we built.

[Back to the start of Chapter 1] [Back to Table of Contents]

Chapter 2

Linear optimization

2.1 Formulation

The simplest, and most scalable class of optimization problems is the one where the objective function and the constraints are formulated using the simplest possible type of functions – linear functions. A **linear program** (LP) is an optimization problem of the form

$$\begin{aligned} \min \quad & \boldsymbol{c}^\top \boldsymbol{x} \\ \text{s.t.} \quad & A\boldsymbol{x} \geq \boldsymbol{b}, \\ & \boldsymbol{x} \geq 0, \end{aligned} \tag{2.1}$$

where the n (decision) variables are grouped in a vector $\boldsymbol{x} \in \mathbb{R}^n$, $\boldsymbol{c} \in \mathbb{R}^n$ are the objective coefficients, and the m linear constraints are described by the matrix $A \in \mathbb{R}^{m \times n}$ and the vector $\boldsymbol{b} \in \mathbb{R}^m$.

Of course, linear problems could also (i) be minimization problems, (ii) involve equality constraints and constraints of the form $\geq$, and (iii) have unbounded or non-positive decision variables x_i. In fact, any LP problem with such features can be easily converted to the 'canonical' LP form (2.1) by adding/removing variables and/or multiplying specific inequalities by -1.

Example 1 (Building microchips pt. 1 – problem formulation)**.** *The company BIM (Best International Machines) produces two types of microchips, logic chips (1gr silicon, 1gr plastic, 4gr copper) and memory chips (1gr germanium, 1gr plastic, 2gr copper). Each of the logic chips can be sold for a 12 € profit, and each of the memory chips for a 9 € profit. The current stock of raw materials is as follows: 1000gr silicon, 1500gr germanium, 1750gr plastic, 4800gr of copper. How many microchips of each type should be produced to maximize the profit while respecting the raw material stock availability?*

Let x_1 denote the number of logic chips and x_2 that of memory chips. This decision can be reformulated as an optimization problem of the following form:

$$\begin{array}{lrcrcll} \max & 12x_1 & + & 9x_2 & & & \\ \text{s.t.} & x_1 & & & \leq & 1000 & \text{(silicon)} \\ & & & x_2 & \leq & 1500 & \text{(germanium)} \\ & x_1 & + & x_2 & \leq & 1750 & \text{(plastic)} \\ & 4x_1 & + & 2x_2 & \leq & 4800 & \text{(copper)} \\ & x_1 & , & x_2 & \geq & 0 & \end{array} \tag{2.2}$$

The problem has $n = 2$ decision variables and $m = 4$ constraints. Using the standard notation introduced in (2.1) above, denote the vector of decision variables by $\boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$ and define the problem coefficients as

$$\boldsymbol{c} = \begin{pmatrix} 12 \\ 9 \end{pmatrix}, \qquad A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \\ 4 & 2 \end{bmatrix}, \quad \text{and} \quad \boldsymbol{b} = \begin{pmatrix} 1000 \\ 1500 \\ 1750 \\ 4800 \end{pmatrix}.$$

	x_1	x_2	value
0	0.0	0.0	0.0
1	0.0	1500.0	13500.0
4	250.0	1500.0	16500.0
5	650.0	1100.0	17700.0
2	1000.0	0.0	12000.0
3	1000.0	400.0	15600.0

Table 2.1: Some corner points of the feasible regions and the corresponding value of the objective function.

Aiming to build some intuition for the feasible region, we can create a small list of feasible solutions which are easy to guess and calculate the corresponding value of the objective function, see Table 2.1 below. They can be obtained by calculating points in which multiple constraints are satisfied with equalities.

Leveraging the fact that we have a two-dimensional problem, we can then represent the full feasible region.

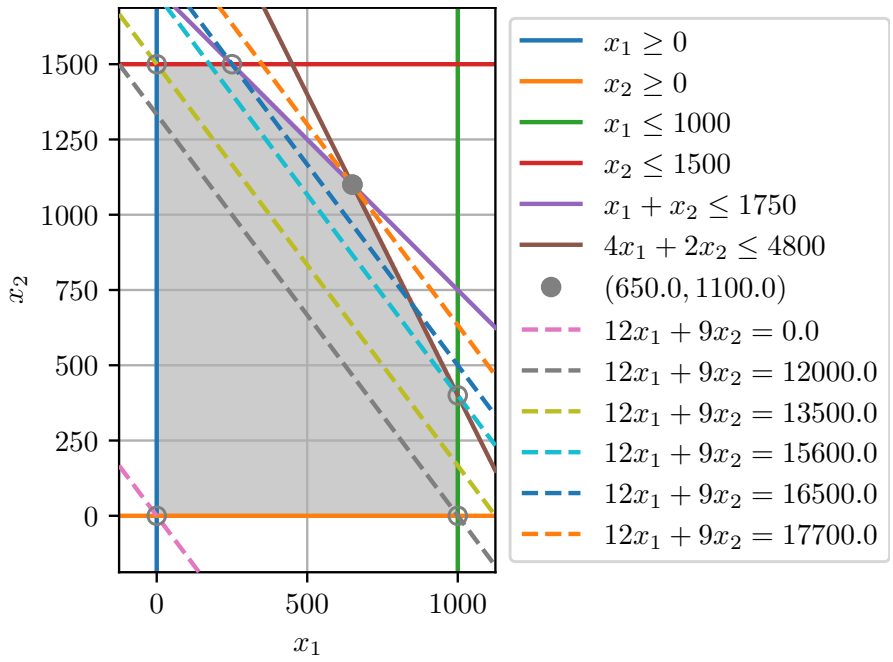


Figure 2.1: The feasible region (in gray), enclosed by the linear constraints (solid lines) and the isolines (parallell dashed lines) corresponding to the objective function.

The optimization problem in (2.2) can be implemented in Pyomo as follows.

```
m    = pyo.ConcreteModel('BIM')

m.x1 = pyo.Var()
m.x2 = pyo.Var()

m.profit    = pyo.Objective( expr = 12*m.x1 + 9*m.x2, sense= pyo.maximize  )

m.silicon   = pyo.Constraint(expr =    m.x1          <= 1000)
m.germanium = pyo.Constraint(expr =             m.x2 <= 1500)
m.plastic   = pyo.Constraint(expr =    m.x1 +   m.x2 <= 1750)
m.copper    = pyo.Constraint(expr =  4*m.x1 + 2*m.x2 <= 4800)
m.x1domain  = pyo.Constraint(expr =    m.x1          >=    0)
m.x2domain  = pyo.Constraint(expr =             m.x2 >=    0)
```

```
14
15 pyo.SolverFactory('glpk').solve(m)
```

Several things can be said about the above example. First, it was rather straightforward to model – the choice of the decision variables was obvious and the constraints were easy to formulate. Secondly, it was easy to find the optimal solution by hand. Third, it is evident to 'naked eye' that the solution found is indeed optimal.

Surprisingly, also much larger and seemingly more complicated problems can be modelled using linear constraints only. For such problems, however, we are often not able to find the solution by hand, and one typically cannot judge 'by eye' that a particular solution is an optimal one. To move on to working confidently with real-life problems, we need to gain more knowledge about LP.

In the following sections, we shall expand on what we learned so far. First, we will extend our capabilities of modelling various situations using linear constraints. Secondly, we will provide a formal definition of the search for the optimality certificate of a given solution. In the end, we will explain intuitively how numerical algorithms for LP problems work.

2.2 Modeling techniques

For some expressions, like the ones in Example 1, it is clear that we can write them down using linear functions. But, there are real-life important objective functions and constraints, for which it is difficult to immediately see the same. At the same time, because LP are by far the easiest problems to solve, a problem should be expressed using linear constraints long as it is possible. We will therefore provide a number of useful LP modeling techniques.

2.2.1 Absolute values

Consider a situation where the objective function in a minimization problem contains absolute values $|x_i|$ of (some of) the variables (or a linear combination of these with non-negative coefficients).

Although the objective is not linear when it contains absolute values $|x_i|$, we can obtain an equivalent linear formulation. For such variable x_i, introduce two new variables $x_i^-, x_i^+ \geq 0$ and replace each occurrence of x_i and $|x_i|$ by

$$\begin{aligned} x_i &= x_i^+ - x_i^-, \\ |x_i| &= x_i^+ + x_i^-, \\ x_i^+, x_i^- &\geq 0. \end{aligned}$$

It is easy to show that for any solution of the modified problem if either x_i^+ or x_i^- is zero for every i, then the problems with and without absolute values are equivalent.

Note that the same reasoning can be used to reformulate absolute values involving entire expressions such as $|x_1 - x_4|$ and, constraints such as

$$|x_1| + x_2 \leq 4,$$

but it cannot be used to do so when the coefficient in front of the absolute value is negative.

Example 2 (Least Absolute Deviation Regression)**.** *Suppose we have a finite dataset consisting of n points $\{(\boldsymbol{X}^{(i)}, y^{(i)})\}_{i=1,\ldots,n}$ with $\boldsymbol{X}^{(i)} \in \mathbb{R}^k$ and $y^{(i)} \in \mathbb{R}$. A linear regression model assumes that the relationship between the vector of k regressors $\boldsymbol{X}$ and the dependent variable y is linear. This relationship is modeled through an error or deviation term e_i, which quantifies how much each of the data points diverge from the model prediction and is defined as follows:*

$$e_i := y^{(i)} - \boldsymbol{m}^\top \boldsymbol{X}^{(i)} - b = y^{(i)} - \sum_{j=1}^{k} X_j^{(i)} m_j - b, \tag{2.3}$$

for some real numbers $m_1, \ldots, m_k$ and b.

The Least Absolute Deviation (LAD) is a possible statistical optimality criterion for such a linear regression. Similar to the well-known least-squares technique, it attempts to find a vector of linear coefficients $\boldsymbol{m} = (m_1, \ldots, m_k)$ and

intercept b so that the model closely approximates the given set of data. The method minimizes the sum of absolute errors, that is $\sum_{i=1}^{n} |e_i|$.

The LAD regression can thus be formulated as an optimization problem with the intercept b, the coefficients m_i's, and the errors e_i's as the decision variables, namely

$$\begin{aligned} \min \quad & \sum_{i=1}^{n} |e_i| \\ \text{s.t.} \quad & e_i = y^{(i)} - \boldsymbol{m}^\top \boldsymbol{X}^{(i)} - b, \qquad \forall\, i = 1, \ldots, n. \end{aligned} \tag{2.4}$$

Since it is a minimization problem and the absolute values appear in the objective function, we can use the trick illustrated above to transform it into a linear problem. More specifically, introducing for every term e_i two new variables $e_i^-, e_i^+ \geq 0$, we can rewrite (2.4) as

$$\begin{aligned} \min \quad & \sum_{i=1}^{n} (e_i^+ + e_i^-) \\ \text{s.t.} \quad & e_i^+ - e_i^- = y^{(i)} - \boldsymbol{m}^\top \boldsymbol{X}^{(i)} - b, \qquad \forall\, i = 1, \ldots, n, \\ & e_i^+, e_i^- \geq 0, \qquad \forall\, i = 1, \ldots, n. \end{aligned} \tag{2.5}$$

Such a linear problem can be implemented in Pyomo as follows.

```
m = pyo.ConcreteModel('LAD regression')

#assumes the data points have already been imported as (X,y)

n, k = X.shape
m.I = pyo.RangeSet(0, n-1)
m.J = pyo.RangeSet(0, k-1)

m.ep = pyo.Var(m.I,within=pyo.NonNegativeReals)
m.em = pyo.Var(m.I,within=pyo.NonNegativeReals)
m.m  = pyo.Var(m.J)
m.b  = pyo.Var()

@m.Constraint(m.I)
def fit(m, i):
    return y[i] == m.b + m.ep[i] - m.em[i] + sum(X[i][j]*m.m[j] for j in m.J)

@m.Objective(sense=pyo.minimize)
def obj(m):
    return sum(m.ep[i] + m.em[i] for i in m.I)

pyo.SolverFactory('glpk').solve(m)
```

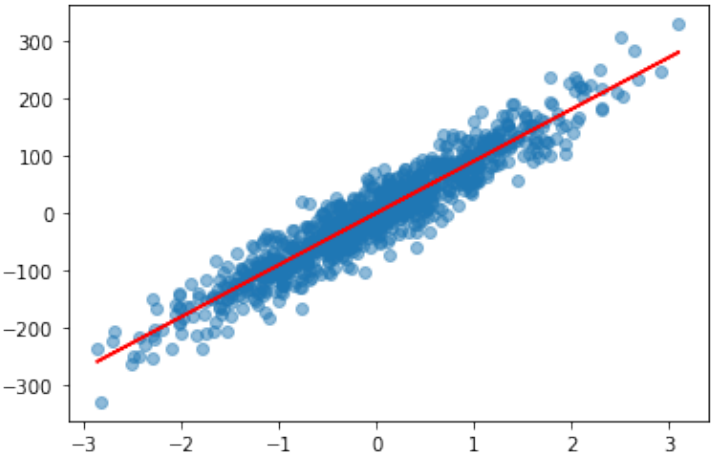


Figure 2.2: The linear approximation resulting from the LAD regression of $n = 1000$ data points $(x^{(i)}, y^{(i)})$. With a minor aboute of notation, we set $X^{(i)} = x^{(i)}$ as the vector of regressors contains only $k = 1$ feature.

Example 3 (Portfolio optimization: Minimizing the mean absolute deviation)**.** *Portfolio optimization and modern portfolio theory has a long and important history in finance and investment. The principal idea is to find a blend of investments in financial securities that achieves an optimal trade-off between financial risk and return. The introduction of modern portfolio theory is generally attributed to the 1952 doctoral thesis of Harry Markowitz who subsequently was award a share of the 1990 Nobel Memorial Prize in Economics for his fundamental contributions to this field. The well-known Markowitz Model models measure risk using covariance of the portfolio with respect to constituent assets, then solves a minimum variance problem by quadratic optimization problem subject to constraints to allocate of wealth among assets. In 1991, the authors of [KY91] proposed a different approach using the mean absolute deviation in portfolio return as a measure of financial risk. In this example, we will look at how to build such a model into a large scale linear programming problem using directly historical stock price data, like the one summarized in Figure 2.3.*

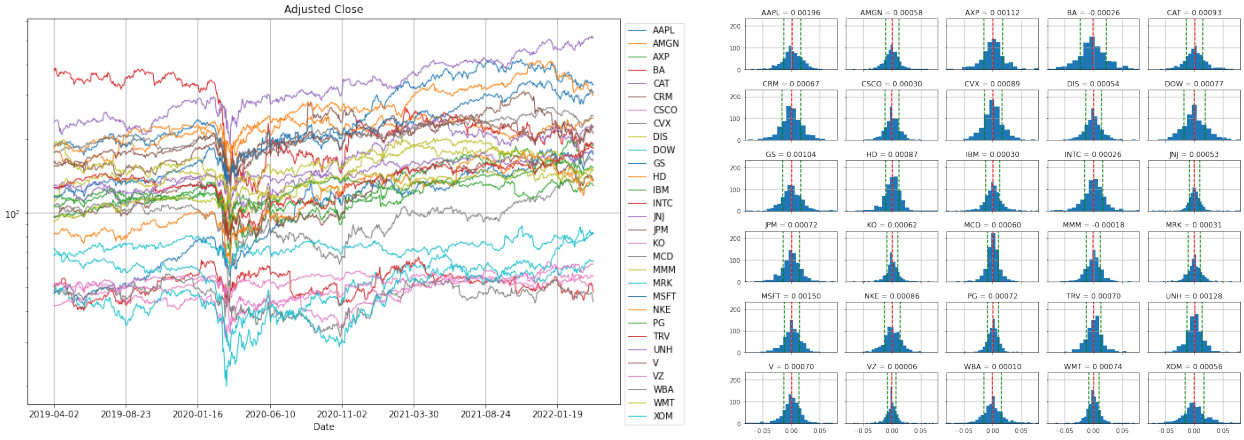


Figure 2.3: Historical price data for a few selected stocks and the distribution of their daily returns.

The portfolio optimization problem is to find an allocation of investment capital to minimize the portfolio measure of risk subject to constraints on required return and any other constraints an investor wishes to impose. Assume that we can make investment decisions at every trading day t over a fixed time horizon ranging from $t = 1, \ldots, T$ and that there is a set of J assets in which we can choose to invest. Let $r_{t,j}$ is the return on asset j at time t, $\bar{r}_j$ is the mean return for asset j, and w_j is the fraction of the total portfolio that is invested in asset j. If we aim to have a guaranteed minimum portfolio return R, but at the same time minimize risk, we could choose to have the mean

absolute deviation (MAD) in portfolio returns as objective function. The resulting optimization problem is

$$\begin{aligned} MAD = \min \quad & \frac{1}{T}\sum_{t=1}^{T}\left|\sum_{j=1}^{J} w_j(r_{t,j} - \bar{r}_j)\right| \\ \text{s.t.} \quad & \sum_{j=1}^{J} w_j \bar{r}_j \geq R \\ & \sum_{j=1}^{J} w_j = 1 \\ & w_j \geq 0 && \forall j \in 1, \ldots, J \\ & w_j \leq w_j^{ub} && \forall j \in 1, \ldots, J, \end{aligned}$$

The lower bound $w_j \geq 0$ is a "no short sales" constraint. The upper bound $w_j \leq w_j^{ub}$ enforces a required level of diversification in the portfolio. Defining $u_t \geq 0$ and $v_t \geq 0$ leads to a reformulation of the

$$\begin{aligned} MAD = \min \quad & \frac{1}{T}\sum_{t=1}^{T}(u_t + v_t) \\ \text{s.t.} \quad & u_t - v_t = \sum_{j=1}^{J} w_j(r_{t,j} - \bar{r}_j) && \forall t \in 1, \ldots, T \\ & \sum_{j=1}^{J} w_j \bar{r}_j \geq R \\ & \sum_{j=1}^{J} w_j = 1 \\ & w_j \geq 0 && \forall j \in 1, \ldots, J \\ & w_j \leq w_j^{ub} && \forall j \in 1, \ldots, J. \end{aligned}$$

We can implement this in Pyomo as follows.

```
def mad_portfolio(assets):

    daily_returns = assets.diff()[1:]/assets.shift(1)[1:]
    mean_return = daily_returns.mean()

    m = pyo.ConcreteModel()

    m.R = pyo.Param(mutable=True, default=0)
    m.w_lb = pyo.Param(mutable=True, default=0)
    m.w_ub = pyo.Param(mutable=True, default=1.0)

    m.ASSETS = pyo.Set(initialize=assets.columns)
    m.TIME = pyo.RangeSet(len(daily_returns.index))

    m.w = pyo.Var(m.ASSETS)
    m.u = pyo.Var(m.TIME, domain=pyo.NonNegativeReals)
    m.v = pyo.Var(m.TIME, domain=pyo.NonNegativeReals)

    @m.Objective(sense=pyo.minimize)
    def MAD(m):
        return sum(m.u[t] + m.v[t] for t in m.TIME)/len(m.TIME)

    @m.Constraint(m.TIME)
```

```
    def portfolio_returns(m, t):
        date = daily_returns.index[t-1]
        return m.u[t] - m.v[t] == sum(m.w[j]*(daily_returns.loc[date, j] - mean_return[j]) for j in m.ASSETS)

    @m.Constraint()
    def sum_of_weights(m):
        return sum(m.w[j] for j in m.ASSETS) == 1

    @m.Constraint()
    def mean_portfolio_return(m):
        return sum(m.w[j] * mean_return[j] for j in m.ASSETS) >= m.R

    @m.Constraint(m.ASSETS)
    def no_short(m, j):
        return m.w[j] >= m.w_lb

    @m.Constraint(m.ASSETS)
    def diversify(m, j):
        return m.w[j] <= m.w_ub

    return m

# assuming a single DataFrame named assets contains historical stock data
# the online companion notebook shows how to download and clean such data
m = mad_portfolio(assets)
m.w_lb = 0
m.w_ub = 0.2
m.R = 0.001
pyo.SolverFactory('cbc').solve(m)
```

The optimal solution of the portfolio problem above and its property are illustrated in Figure 2.4 below.

Figure 2.4: The optimal weights found by the MAD portfolio problem and other features of the solution.

Example 4 (Wine quality prediction)**.** *Physical, chemical, and sensory quality properties were collected for a large number of red and white variants of the Portuguese "Vinho Verde" wine and then then donated to the UCI machine*

learning repository, see [Cor+09]. The dataset consists of $n = 1,599$ measurements of 11 physical and chemical characteristics plus an integer measure of sensory quality recorded on a scale from 3 to 8. Due to privacy and logistic issues, there is no data about grape types, wine brand, wine selling price, etc.

The goal of the regression is find coefficients m_j and b to minimize the mean absolute deviation (MAD), that is

$$\begin{aligned} \text{MAD}\,(\hat{y}) = \min \quad & \frac{1}{n}\sum_{i=1}^{n} |y_i - \hat{y}_i| \\ \text{s.t.} \quad & \hat{y}_i = \sum_{j=1}^{J} x_{i,j} m_j + b \qquad \forall i = 1, \ldots, n \end{aligned}$$

where $x_{i,j}$ are values of 'explanatory' variables, in this case the 11 physical and chemical characteristics of the wines.

The following Pyomo code builds a model to obtain the optimal regression coefficients.

```
def l1_fit_2(df, y_col, x_cols):

    m = pyo.ConcreteModel("L1 Regression Model")

    m.I = pyo.RangeSet(len(df))
    m.J = pyo.Set(initialize=x_cols)

    @m.Param(m.I)
    def y(m, i):
        return df.loc[i-1, y_col]

    @m.Param(m.I, m.J)
    def X(m, i, j):
        return df.loc[i-1, j]

    # regression
    m.a = pyo.Var(m.J)
    m.b = pyo.Var(domain=pyo.Reals)

    m.e_pos = pyo.Var(m.I, domain=pyo.NonNegativeReals)
    m.e_neg = pyo.Var(m.I, domain=pyo.NonNegativeReals)

    @m.Expression(m.I)
    def prediction(m, i):
        return sum(m.a[j] * m.X[i, j] for j in m.J) + m.b

    @m.Constraint(m.I)
    def prediction_error(m, i):
        return m.e_pos[i] - m.e_neg[i] == m.prediction[i] - m.y[i]

    @m.Objective(sense=pyo.minimize)
    def mean_absolute_deviation(m):
        return sum(m.e_pos[i] + m.e_neg[i] for i in m.I)/len(m.I)

    pyo.SolverFactory('cbc').solve(m)

    return m

# assuming the wine measurements have been imported in the dataframe wines
# see online companion notebook to see how to download and import it
m = l1_fit_2(wines, "quality",
                    ["alcohol", "volatile acidity", "citric acid", "sulphates", \
                     "total sulfur dioxide", "density", "fixed acidity"])
```

2.2.2 Minimax objective

Another class of seemingly complicated objective functions that can be easily rewritten as a LP are those stated as maxima over several linear functions. Given a finite set of indices $\mathcal{K}$ and a collection of vectors $\{\boldsymbol{c}_k\}_{k\in\mathcal{K}}$, the minimax problem given by

$$\min \max_{k\in\mathcal{K}} \sum_{i\in\mathcal{I}} \boldsymbol{c}_k^\top \boldsymbol{x}. \tag{2.6}$$

General expressions such as (2.6) can be linearized by introducing an auxiliary variable z and setting

$$\begin{aligned} \min \quad & z \\ \text{s.t.} \quad & \boldsymbol{c}_k^\top \boldsymbol{x} \le z, \qquad \forall\, k \in \mathcal{K}. \end{aligned}$$

This trick works because if *all* the quantities corresponding to different indices $k \in \mathcal{K}$ are below the auxiliary variable z, then we are guaranteed that also their maximum is also below z and vice versa. Note that the absolute value function can be rewritten $|x_i| = \max\{x_i, -x_i\}$, hence the linearization of the optimization problem involving absolute values in the objective functions is a special case of this.

Example 5 (Building microchips pt. 2 – maximizing lowest possible profit)**.** *In the same way as minimizing the maximum like in (2.6) one can also maximize the minimum. Let us consider the same microchip production problem as in Example 1, but suppose that there is uncertainty regarding the selling prices of the microchips. Instead of the just the nominal prices* 12€ *and* 9€*, BIM estimates that the prices may more generally take the values* $P = \{(12, 9), (11, 10), (8, 11)\}$*. The optimization problem for production plan that achieves the maximum among the lowest possible profits can be formulated using the trick mentioned above and can be implemented in Pyomo as follows.*

```
1  def BIM_maxmin( costs ):
2      model    = pyo.ConcreteModel('BIM')
3
4      model.x1 = pyo.Var(within=pyo.NonNegativeReals)
5      model.x2 = pyo.Var(within=pyo.NonNegativeReals)
6      model.z  = pyo.Var()
7
8      model.profit    = pyo.Objective( sense= pyo.maximize, expr = model.z )
9
10     model.maxmin = pyo.ConstraintList()
11     for (c1,c2) in costs:
12         model.maxmin.add( expr = model.z <= c1*model.x1 + c2*model.x2 )
13
14     model.silicon   = pyo.Constraint(expr =    model.x1             <= 1000)
15     model.germanium = pyo.Constraint(expr =                 model.x2 <= 1500)
16     model.plastic   = pyo.Constraint(expr =    model.x1 +   model.x2 <= 1750)
17     model.copper    = pyo.Constraint(expr =  4*model.x1 + 2*model.x2 <= 4800)
18
19     return model
20
21 BIM = BIM_maxmin( [[12,9], [11,10], [8, 11]] )
22 results = pyo.SolverFactory('glpk').solve(BIM)
```

2.2.3 Fractional objective

In some problems, one might be interested in minimizing a ratio of one quantity to the other, where both depend on the decision variables. Although terms such as

$$\frac{x_1}{2x_2 + 5}$$

are clearly not linear expressions, under certain conditions it is also possible to minimize them using a linear problem. To demonstrate this, take a finite index set $\mathcal{I}$ and consider the optimization problem with a fractional objective

function of the form

$$\begin{array}{ll} \min & \dfrac{\boldsymbol{c}^\top \boldsymbol{x} + \alpha}{\boldsymbol{d}^\top \boldsymbol{x} + \beta} \\ \text{s.t.} & A\boldsymbol{x} \leq \boldsymbol{b}, \\ & \boldsymbol{x} \geq 0, \end{array}$$

where the term $\boldsymbol{d}^\top \boldsymbol{x} + \beta$ is either strictly positive or strictly negative over the entire feasible set of x. Setting first $t = \left(\boldsymbol{d}^\top \boldsymbol{x} + \beta\right)^{-1}$ and then $y_i = x_i t$ for every index i, we obtain the following equivalent linear optimization problem

$$\begin{array}{ll} \min & \boldsymbol{c}^\top \boldsymbol{y} + \alpha t \\ \text{s.t.} & A\boldsymbol{y} \leq t\boldsymbol{b}, \\ & \boldsymbol{d}^\top \boldsymbol{y} + \beta t = 1, \\ & t \geq 0, \\ & \boldsymbol{y} \geq 0. \end{array}$$

Note that the inequality for t should in fact be strict, i.e., $t > 0$, but in view of the assumption above for $\boldsymbol{d}^\top \boldsymbol{x} + \beta$, having relaxed the constraint does not change the optimal solution.

Example 6 (Building microchips pt. 3 – different profit formulations)**.** *Recall the BIM production model introduced in Example 1, that is*

$$\begin{array}{lrcrcrl} \max & 12x_1 & + & 9x_2 & & & \\ \text{s.t.} & x_1 & & & \leq & 1000 & \text{(silicon)} \\ & & & x_2 & \leq & 1500 & \text{(germanium)} \\ & x_1 & + & x_2 & \leq & 1750 & \text{(plastic)} \\ & 4x_1 & + & 2x_2 & \leq & 4800 & \text{(copper)} \\ & x_1 & , & x_2 & \geq & 0 & \end{array}$$

Assume the pair $(12, 9)$ *reflects the sales price (revenues) in* € *and not the profits made per unit produced. We then need to account for the production costs. Suppose that the production costs for* (x_1, x_2) *chips are equal to a fixed cost of* 100 *(independent of the number of units produced) plus* $7/6x_1$ *plus* $5/6x_2$*. It is reasonable to maximize the difference between the revenues and the costs. This yields the linear model:*

$$\begin{array}{lrcrcrcrl} \max & (12-7/6)x_1 & + & (9-5/6)x_2 & - & 100 & & \\ \text{s.t.} & x_1 & & & \leq & 1000 & & \text{(silicon)} \\ & & & x_2 & \leq & 1500 & & \text{(germanium)} \\ & x_1 & + & x_2 & \leq & 1750 & & \text{(plastic)} \\ & 4x_1 & + & 2x_2 & \leq & 4800 & & \text{(copper)} \\ & x_1 & , & x_2 & \geq & 0 & & \end{array}$$

```python
def BIM_with_revenues_minus_costs():
    model    = pyo.ConcreteModel('BIM')

    model.x1 = pyo.Var(within=pyo.NonNegativeReals)
    model.x2 = pyo.Var(within=pyo.NonNegativeReals)

    model.revenue       = pyo.Expression( expr = 12*model.x1  +   9*model.x2 )
    model.variable_cost = pyo.Expression( expr = 7/6*model.x1 + 5/6*model.x2 )
    model.fixed_cost    = 100

    model.profit    = pyo.Objective( sense= pyo.maximize
      , expr = model.revenue - model.variable_cost - model.fixed_cost )

    model.silicon   = pyo.Constraint(expr =    model.x1             <= 1000)
    model.germanium = pyo.Constraint(expr =                 model.x2 <= 1500)
    model.plastic   = pyo.Constraint(expr =    model.x1 +   model.x2 <= 1750)
    model.copper    = pyo.Constraint(expr =  4*model.x1 + 2*model.x2 <= 4800)

```

```
19     return model
20
21 BIM_linear = BIM_with_revenues_minus_costs()
22 results = pyo.SolverFactory('glpk').solve(BIM_linear)
```

Alternatively we may aim to optimize the efficiency of the plan, expressed as the ration between the revenues and the costs:

$$\begin{array}{rl} \max & \dfrac{12x_1 + 9x_2}{7/6x_1 + 5/6x_2 + 100} \\ \text{s.t.} & \end{array}$$

$$\begin{array}{rcrcrll} x_1 & & & \leq & 1000 & \text{(silicon)} \\ & & x_2 & \leq & 1500 & \text{(germanium)} \\ x_1 & + & x_2 & \leq & 1750 & \text{(plastic)} \\ 4x_1 & + & 2x_2 & \leq & 4800 & \text{(copper)} \\ x_1 & , & x_2 & \geq & 0 & \end{array}$$

The first version has the same optimal solution as the original BIM model, namely $(650, 1100)$ *with a revenue of* 17700 *and a cost of* 1775*. The second optimizes as* $(250, 1500)$ *with a revenue of* 16500 *and a cost of* 1641.667*. The efficiency of the first solution is* $\frac{17700}{1775} = 9.972$ *while the second is the highest at* $\frac{16500}{1641.667} = 10.051$*.*

In order to solve the second version we reformulate the model as

$$\begin{array}{rcrcrcrcrl} \max & 12y_1 & + & 9y_2 & & & & & & \\ \text{s.t.} & y_1 & & & & & \leq & 1000t & & \text{(silicon)} \\ & & & y_2 & & & \leq & 1500t & & \text{(germanium)} \\ & y_1 & + & y_2 & & & \leq & 1750t & & \text{(plastic)} \\ & 4y_1 & + & 2y_2 & & & \leq & 4800t & & \text{(copper)} \\ & 7/6y_1 & + & 5/6y_2 & + & 100y & = & 1 & & \text{(the fraction)} \\ & y_1 & , & y_2 & , & t & \geq & 0 & & \end{array}$$

and recover the solution as $(x_1, x_2) = (y_1/t, y_2/t)$*.*

```
1  def BIM_with_revenues_over_costs():
2      model    = pyo.ConcreteModel('BIM')
3
4      model.y1 = pyo.Var(within=pyo.NonNegativeReals)
5      model.y2 = pyo.Var(within=pyo.NonNegativeReals)
6      model.t  = pyo.Var(within=pyo.NonNegativeReals)
7
8      model.revenue       = pyo.Expression( expr = 12*model.y1  +   9*model.y2 )
9      model.variable_cost = pyo.Expression( expr = 7/6*model.y1 + 5/6*model.y2 )
10     model.fixed_cost    = 100
11
12     model.profit    = pyo.Objective( sense= pyo.maximize
13                                    , expr = model.revenue)
14
15     model.silicon   = pyo.Constraint(expr =    model.y1             <= 1000*model.t)
16     model.germanium = pyo.Constraint(expr =                 model.y2 <= 1500*model.t)
17     model.plastic   = pyo.Constraint(expr =    model.y1 +   model.y2 <= 1750*model.t)
18     model.copper    = pyo.Constraint(expr =  4*model.y1 + 2*model.y2 <= 4800*model.t)
19     model.frac      = pyo.Constraint(expr = model.variable_cost+model.fixed_cost*model.t == 1 )
20
21     return model
22
23 BIM_fractional = BIM_with_revenues_over_costs()
24 results = pyo.SolverFactory('glpk').solve(BIM_fractional)
```

2.3 Duality

As we could see in Example 1, software was certain that the solution it provided us with was an optimal solution. How could that certainty be achieved? We will now develop a general methodology of constructing such optimality certificates. First, we will show how to guess/construct such a certificate on a small problem 'manually'.

Example 7 (Building microchips pt. 3 – optimality certificate)**.** *Consider the constraints* $x_1 \leq 1000$ *and* $x_2 \leq 1500$*. If we multiply the first constraint by* 12 *and the second one by* 9*, we obtain a system*

$$\begin{aligned} 12x_1 &\leq 12000 \\ 9x_2 &\leq 13500. \end{aligned}$$

Adding the two inequalities side-wise yields

$$12x_1 + 9x_2 \leq 25500.$$

Notice that the left-hand side is the same as the objective function. Also, because we obtained this inequality by adding non-negative multiples of inequalities that hold for any feasible (x_1, x_2)*, this inequality has to hold for any feasible* (x_1, x_2) *as well. Therefore, no feasible solution can attain an objective function value higher than* 25500*. In this way, we have obtained our first upper bound on the optimal value. But, is there a solution that attains this value? You can try yourself that this is not possible.*

Now, let us look at other constraints. Summing side by side the copper constraint and the germanium constraint we get $4x_1 + 3x_2 \leq 6300$*, which is equivalent to*

$$12x_1 + 9x_2 \leq 18900.$$

Again, the left-hand side takes the same form as the objective function. At the same time, the right-hand side is lower than 25500*. Therefore, we get a tighter upper bound than the previous one. This explains why it was not possible to find a solution attaining the previous bound.*

But can the new bound be attained by some solution or can we lower this upper bound further? The latter is true – by taking 6 *times the plastic constraint and adding* 3/2 *the copper constraint one gets*

$$12x_1 + 9x_2 \leq 17700.$$

This time, we can find a feasible solution x_1, x_2 *that attains the value* 17700 *prescribed by the upper bound (that is* $(x_1, x_2) = (650, 1100)$*. At the same time, because this is an upper bound, we know that no feasible solution can attain a higher objective value. Therefore, the found solution is an optimal one. The inequality* $12x_1 + 9x_2 \leq 17700$ *acts as an optimality certificate of the found solution.*

What happened implicitly in the above example is that we constructed for our original problem, denoted from now as the *primal problem*, is associated a *dual problem* which is also an LP and which aims for determining the best bound on the objective. It is time now to show this dual problem, again, on this example.

Example 8 (Building microchips pt. 4 – dual problem)**.** *Consider again the microchip production problem:*

$$\begin{aligned} \max \quad & 12x_1 + 9x_2 \\ \text{s.t.} \quad & x_1 \leq 1000, \\ & x_2 \leq 1500, \\ & x_1 + x_2 \leq 1750, \\ & 4x_1 + 2x_2 \leq 4800, \\ & x_1, x_2 \geq 0. \end{aligned}$$

As illustrated in Example 7, constructing bounds on the objective function consisted in multiplying the constraints by non-negative numbers and adding them to each other so that the left-hand side looks like the objective function, while the right-hand side is the corresponding bound.

Let therefore $\lambda_1, \lambda_2, \lambda_3, \lambda_4$ *be non-negative numbers, such that we multiply each of the constraints by the respective number and sum all of them side by side to obtain the inequality*

$$(\lambda_1 + \lambda_3 + 4\lambda_4)x_1 + (\lambda_2 + \lambda_3 + 2\lambda_4)x_2 \leq 1000\lambda_1 + 1500\lambda_2 + 1750\lambda_3 + 4800\lambda_4.$$

It is clear that if $\lambda_1, \lambda_2, \lambda_3, \lambda_4 \geq 0$ *satisfy*

$$\lambda_1 + \lambda_3 + 4\lambda_4 \geq 12,$$
$$\lambda_2 + \lambda_3 + 2\lambda_4 \geq 9,$$

then we have the following:

$$12x_1 + 9x_2 \leq (\lambda_1 + \lambda_3 + 4\lambda_4)x_1 + (\lambda_2 + \lambda_3 + 2\lambda_4)x_2 \leq 1000\lambda_1 + 1500\lambda_2 + 1750\lambda_3 + 4800\lambda_4, \tag{2.7}$$

where the first inequality follows from the fact that $x_1, x_2 \geq 0$*, and the most right-hand expression becomes an upper bound on the optimal value of the objective.*

If we seek $\lambda_1, \lambda_2, \lambda_3, \lambda_4 \geq 0$ *such that the upper bound on the RHS is as tight as possible, that means that we need to* minimize *the expression* $1000\lambda_1 + 1500\lambda_2 + 1750\lambda_3 + 4800\lambda_4$*. This can be formulated as the following LP, which we name the* dual problem*:*

$$\begin{aligned} \min \quad & 1000\lambda_1 + 1500\lambda_2 + 1750\lambda_3 + 4800\lambda_4 \\ \text{s.t.} \quad & \lambda_1 + \lambda_3 + 4\lambda_4 \geq 12, \\ & \lambda_2 + \lambda_3 + 2\lambda_4 \geq 9, \\ & \lambda_1, \lambda_2, \lambda_3, \lambda_4 \geq 0. \end{aligned}$$

It is easy to solve and find the optimal solution $(\lambda_1, \lambda_2, \lambda_3, \lambda_4) = (0, 0, 6, 1.5)$*, for which the objective functions takes the value* 17700*. In view of* (2.7)*, such a value is (the tightest) upper bound for the original problem. Here, we present the Pyomo code for this example.*

```
m    = pyo.ConcreteModel('BIM_dual')

m.y1 = pyo.Var( within = pyo.NonNegativeReals )
m.y2 = pyo.Var( within = pyo.NonNegativeReals )
m.y3 = pyo.Var( within = pyo.NonNegativeReals )
m.y4 = pyo.Var( within = pyo.NonNegativeReals )

m.obj = pyo.Objective( sense= pyo.minimize
                     , expr = 1000*m.y1 + 1500*m.y2 + 1750*m.y3 + 4800*m.y4 )

m.x1 = pyo.Constraint( expr =      m.y1             +      m.y3 +    4*m.y4 >= 12 )
m.x2 = pyo.Constraint( expr =                  m.y2 +      m.y3 +    2*m.y4 >=  9 )

pyo.SolverFactory('glpk').solve(m)
```

In the above example, we constructed a dual for a specific maximization problem where all variables were non-negative and all constraints were inequalities of the $\leq$ form. For a general minimization problem of the form (2.1), the procedure is analogous and the dual can be formulated as follows.

Primal problem:

$$\begin{aligned} \min \quad & \boldsymbol{c}^\top \boldsymbol{x} \\ \text{s.t.} \quad & A\boldsymbol{x} \geq \boldsymbol{b}, \\ & \boldsymbol{x} \geq 0. \end{aligned} \tag{2.8}$$

Dual problem:

$$\begin{aligned} \max \quad & \boldsymbol{b}^\top \boldsymbol{\lambda} \\ \text{s.t.} \quad & \boldsymbol{\lambda}^\top A \leq \boldsymbol{c}, \\ & \boldsymbol{\lambda} \geq 0. \end{aligned} \tag{2.9}$$

For a general minimization-maximization problem with diverse constraints/variables, we can provide the following procedure for constructing the dual problem.

1. Rewrite the primal LP such that all terms with variables are on the left-hand side of each constraint and all constants on the right-hand side.
2. For each primal constraint (excluding domain restrictions), add a dual variable with objective coefficient equal to the constant on right-hand side of the constraint. If the primal problem has m constraints (i.e., the matrix A is $m \times n$), then $\boldsymbol{\lambda}$ is m-dimensional vector.

3. The objective switches between primal and dual from minimization to maximization and vice versa.
4. Add dual constraints and domain restrictions depending on the primal domain restrictions and primal constraints, respectively, as in Tables 2.2 and 2.3. In particular, if the primal problem is n-dimensional and has m primal constraints, the dual problem will be m-dimensional and have n dual constraints.

These procedural rules can be derived in two ways, either by constructing 'manually' the dual problem using bounds as in Example 8, or, more simply, by using the Lagrangian relaxation technique that will be introduced later in **??**.

		Dual constraint
Primal variable	$x_j \geq 0$	$(A^\top \boldsymbol{\lambda})_j \leq c_j$
	$x_j \leq 0$	$(A^\top \boldsymbol{\lambda})_j \geq c_j$
	$x_j \in \mathbb{R}$	$(A^\top \boldsymbol{\lambda})_j = c_j$
		Dual variable
Primal constraint	$(A\boldsymbol{x})_i \leq b_i$	$\lambda_j \leq 0$
	$(A\boldsymbol{x})_i \geq b_i$	$\lambda_j \geq 0$
	$(A\boldsymbol{x})_i = b_i$	$\lambda_j \in \mathbb{R}$

Table 2.2: Duality for primal minimization

		Dual constraint
Primal variable	$x_j \geq 0$	$(A^\top \boldsymbol{\lambda})_j \geq c_j$
	$x_j \leq 0$	$(A^\top \boldsymbol{\lambda})_j \leq c_j$
	$x_j \in \mathbb{R}$	$(A^\top \boldsymbol{\lambda})_j = c_j$
		Dual variable
Primal constraint	$(A\boldsymbol{x})_i \leq b_i$	$\lambda_j \geq 0$
	$(A\boldsymbol{x})_i \geq b_i$	$\lambda_j \leq 0$
	$(A\boldsymbol{x})_i = b_i$	$\lambda_j \in \mathbb{R}$

Table 2.3: Duality for primal maximization

Let us know derive the dual of an LP using these rules. Consider a LP with three variables x_1, x_2, x_3 (on the right) which display all types of constraints and derive its dual (on the left). Since the primal problem is a minimization problem, we need to look at Table 2.2. For sake of clarity, we display each constraint of the dual problem on the same line as the corresponding primal constraint.

$$\begin{aligned} \min \quad & x_1 - 2x_2 + 4x_3 \\ \text{s.t.} \quad & -x_1 + 3x_2 = 3 \\ & 2x_1 - x_2 + 3x_3 \geq -5 \\ & x_3 \leq 7 \\ & x_1 + x_2 - x_3 \leq 11 \\ & x_1 \geq 0 \\ & x_2 \leq 0 \\ & x_3 \in \mathbb{R}. \end{aligned} \tag{2.10}$$

$$\begin{aligned} \max \quad & 3\lambda_1 - 5\lambda_2 + 7\lambda_3 + 11\lambda_4 \\ \text{s.t.} \quad & \lambda_1 \in \mathbb{R} \\ & \lambda_2 \geq 0 \\ & \lambda_3 \leq 0 \\ & \lambda_4 \leq 0 \\ & -\lambda_1 + 2\lambda_2 + \lambda_4 \leq 1 \\ & 3\lambda_1 - \lambda_2 + \lambda_4 \geq -2 \\ & 3\lambda_2 + \lambda_3 - \lambda_4 = 4. \end{aligned} \tag{2.11}$$

Note in particular that the dual has four variables (because the primal had four linear constraints) and three linear constraints (because the primal had three variables).

Lastly, we remark that if one takes a dual problem, treats it as if it was a primal problem and derives its dual, then the original primal problem is re-obtained, see Exercise 5.

We come now to the climax of this section – how can all this duality be used to build optimality certificates for the solutions? Mathematically, what makes the optimality certificate search work is the fact that the primal and dual problems are related to each other via theorems known as the weak and strong duality. The first theorem states formally that we can use the dual problem to obtain bounds on the optimal value of the primal problem.

Theorem 1 (Weak duality theorem – linear case)**.** *For all feasible solutions $\boldsymbol{x}$ of the primal problem and all feasible solutions $\boldsymbol{y}$ of the dual problem, the following inequality holds*

$$\boldsymbol{c}^\top \boldsymbol{x} \leq \boldsymbol{b}^\top \boldsymbol{\lambda}.$$

The proof is very easy and uses the inequality constraint of the dual problem first and then that of the primal problem to get the following chain of inequalities $\boldsymbol{c}^\top \boldsymbol{x} \leq \boldsymbol{\lambda}^\top A\boldsymbol{x} \leq \boldsymbol{\lambda}^\top \boldsymbol{b} = \boldsymbol{b}^\top \boldsymbol{\lambda}$, proving the desired result.

As mentioned earlier, weak duality is useful is to certify that a candidate solution pair $(\boldsymbol{x}, \boldsymbol{\lambda})$ is in fact optimal. Suppose that $\boldsymbol{x}$ and $\boldsymbol{\lambda}$ be feasible solutions to the primal and the dual, respectively. Then, if the equality $\boldsymbol{c}^\top \boldsymbol{x} = \boldsymbol{b}^\top \boldsymbol{\lambda}$ holds, weak duality implies that $\boldsymbol{x}$ and $\boldsymbol{\lambda}$ are optimal solutions to the primal and the dual, respectively.

We say that an optimization problem is said to be **unbounded** if the objective function may be improved indefinitely without violating the constraints and is said to be **infeasible** if the feasible set is empty, i.e., there is no feasible point which satisfies all the constraints. Using these definitions, we can state three immediate consequences of the theorem above:

1. if the primal problem is unbounded, then the dual is infeasible;
2. if the dual problem is unbounded, then the primal is infeasible;
3. if a primal solution $\boldsymbol{x}$ attains the same value of a dual solution $\boldsymbol{\lambda}$, i.e., $\boldsymbol{c}^\top \boldsymbol{x} = \boldsymbol{b}^\top \boldsymbol{\lambda}$, then they are *both* optimal.

The following result shows under which conditions **strong duality** holds, i.e., under which conditions the primal and dual problem have exactly the same optimal value.

Theorem 2 (Strong duality – linear case)**.** *If a linear optimization problem has an optimal solution, so does its dual, and the respective optimal values are equal, i.e., $\boldsymbol{c}^\top \boldsymbol{x} = \boldsymbol{b}^\top \boldsymbol{\lambda}$. Therefore, strong duality holds for an LP if either of the primal or dual problem is feasible and bounded.*

Having strong duality guaranteed at the cost of hardly any assumptions is one of the huge benefits of working with linear problems. As we shall see in **??**, we do not always have this privilege for general optimization problems, and there only weak duality can be proven 'with no assumptions added'.

Strong duality allows to derive an important relation between primal and dual optimal solutions, the so-called **complementary slackness conditions**. If the optimal solution of the primal problem exists and is known, these conditions can be used to determine the unique solution to the dual problem.

Proposition 1 (Complementary slackness – linear case)**.** *Let $\boldsymbol{x}$ be a feasible solution of the primal problem and $\boldsymbol{\lambda}$ a feasible solution of the dual problem. They are both optimal for the respective problems if and only if*

$$\begin{cases} \boldsymbol{\lambda}^\top (\boldsymbol{b} - A\boldsymbol{x}) = 0, \\ (A^\top \boldsymbol{\lambda} - \boldsymbol{c})^\top \boldsymbol{x} = 0. \end{cases} \tag{2.12}$$

2.4 Solution methods

We finish this chapter by giving a high-level view of the two basic algorithms used to solve general LP problems. Although we do not go deep into the details, understanding how they work is helpful in understanding the solver output, which we will show in the last part of this chapter.

These two most used methods to solve linear optimization problems are called **simplex method** and **interior point method**, for more details see [BJS10]. First, we graphically illustrate the difference between them on the example of the feasible set of the problem of Example 1 in Figure 2.5. In the left panel, we see that the simplex algorithm operates by selecting subsequent vertices (and only vertices) of the feasible set. The interior point method, as the name suggests, explores the solution space moving between points in the interior of the feasible set (not on the boundary), converging to the same optimal solution as the one obtained by a simplex method.

How do the two different algorithms work, arriving at the same optimal solution?

Simplex method

As you can see in the picture, the simplex method visits only the vertices of the feasible set – points at which m linearly independent constraints (including the nonnegativity constraints on $\boldsymbol{x}$) hold with equality. Indeed, it can be shown that for any LP, an optimal solution, if it exists, can be found among such points.

How can one search the list of such points only and do it efficiently? To enumerate all possible sets of such constraints efficiently, we rewrite the LP to the form:

$$\begin{aligned} \max_{\boldsymbol{x}\in\mathbb{R}^n, \boldsymbol{x}'\in\mathbb{R}^m} \quad & \boldsymbol{c}^\top \boldsymbol{x} \\ \text{s.t.} \quad & A\boldsymbol{x} + \boldsymbol{x}' = \boldsymbol{b}, \\ & \boldsymbol{x}, \boldsymbol{x}' \geq 0, \end{aligned}$$

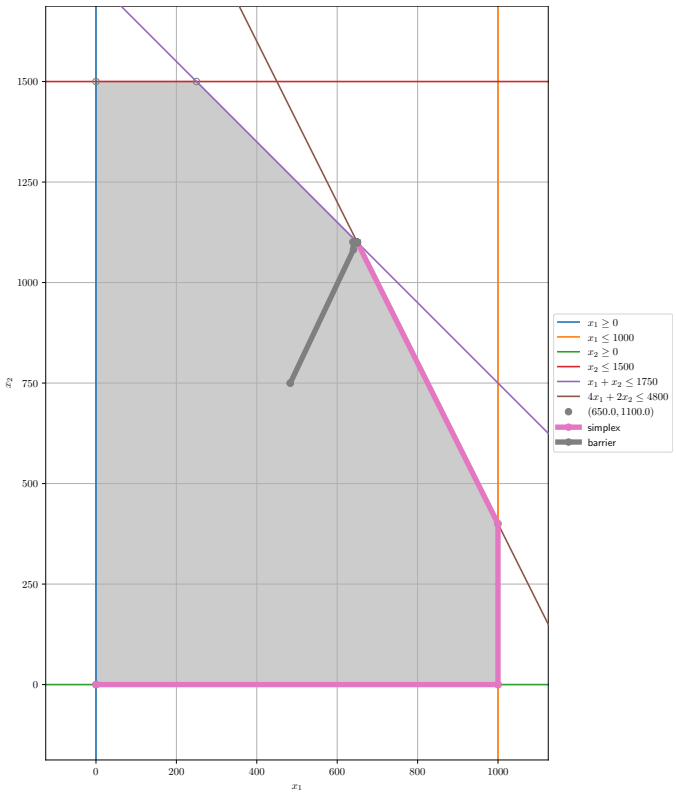


Figure 2.5: The trajectories of two LP solution methods.

where the inequalities have been rewritten as equalities using a vector $\boldsymbol{x}' \in \mathbb{R}^m$ of *slack variables*, which, intuitively, take any (non-negative) value equal to the difference between the original left and right hand side. By merging $\boldsymbol{x}$ and $\boldsymbol{x}'$ into a single vector, the problem is equivalent to

$$\begin{aligned} \max_{\boldsymbol{x} \in \mathbb{R}^{n+m}} \quad & (\boldsymbol{c}')^\top \boldsymbol{x} \\ \text{s.t.} \quad & A'\boldsymbol{x} = \boldsymbol{b}, \\ & \boldsymbol{x} \geq 0, \end{aligned}$$

where

$$\boldsymbol{c}' := \begin{bmatrix} \boldsymbol{c} \\ \boldsymbol{0} \end{bmatrix}, \quad \boldsymbol{x} := \begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{x}' \end{bmatrix}, \quad \text{and} \quad A' := [A\ I],$$

with I being the $m \times m$ identity matrix. In this notation, a vertex corresponds to a *basic solutions*, which corresponds to a selection of m linearly independent columns of matrix A with indices belonging to a set $\mathcal{B}$. If we denote $\mathcal{N} = \{1, \ldots, n+m\} \setminus \mathcal{B}$ then the basic solution corresponding to $\mathcal{B}$ is one obtained by solving the linear system

$$\begin{cases} A_{\mathcal{B}} \boldsymbol{x}_{\mathcal{B}} = \boldsymbol{b}, \\ \boldsymbol{x}_{\mathcal{N}} = \boldsymbol{0}. \end{cases}$$

The simplex method switches between different bases $\mathcal{B}_1, \mathcal{B}_2, \ldots$ until it finds one corresponding to an optimal solution. In doing so, it involves rules for choosing the columns that enter and leave the basis so that at each step, one moves to a basic solution with a better value of the objective. A good implementation aims for as few as possible computations related to inverting matrix B. In the worst case, however, it can visit all the vertices of the feasible set.

Simplex method was the first algorithm developed for general LP problems. Although it can be shown that on a very nasty problem it might have to visit all the vertices of the feasible set, its performance on realistic problems, even of very large size, is very good.

Interior point methods

Interior point methods, as the name suggests, attack the LP problem by searching only points that are 'strictly inside' the feasible set. They iteratively solve auxiliary optimization problems

$$\min_{\boldsymbol{x}\in\mathbb{R}} \boldsymbol{c}^\top \boldsymbol{x} + \mu \left(\sum_{i=1}^{m} \log(b_i - \boldsymbol{a}_i^\top \boldsymbol{x}) + \sum_{i=1}^{n} \log(x_i) \right). \tag{2.13}$$

In this auxiliary problem the constraints have been moved to the objective function as an additional *barrier term* parametrized by $\mu > 0$.

For the barrier term to make sense, all entries of $\boldsymbol{x}$ need to be positive and all the constraints need to be satisfied strictly: $\boldsymbol{a}_i^\top \boldsymbol{x} < b_i$. This explains why a particular point needs to lie inside the feasible set. Furthermore, the smaller the parameter μ, the 'less important' the barrier term for the actual value of the objective function, and the larger the role of the original objective function. In this way, for a decreasing μ the role of the barrier term becomes smaller and smaller, while still ensuring that the solution meets the problem constraints. In this way, for decreasing μ, the optimal solution of the auxiliary problem can approach the optimal solution.

In their actual implementation, interior point methods sequentially solve a sequence of problems (2.13) for decreasing values of the parameter $\mu_1 > \mu_2 > \ldots > 0$, at each step using the optimal solution for the previous value μ_{k-1} as the initial point for the problem with mu_k.

Interior point methods, although they look more complicated than the simplex method, can be proven to converge to the optimal solution efficiently. Mathematically, this means that the number of iterations needed can be upper bounded by a polynomial whose arguments is the problem size (number of variables and constraints).

2.5 A complete example

As explained in Example 1, BIM produces logic and memory chips using two types of conductors, plastic and copper. BIM hired Caroline to manage the acquisition and the inventory of these raw materials. She conducted a data analysis which lead to the following prediction of monthly demands for her chips:

chip type	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
logic chip	88	125	260	217	238	286	248	238	265	293	259	244
memory chip	47	62	81	65	95	118	86	89	82	82	84	66

As you recall, BIM has the following stock at the beginning of the year:

copper	silicon	germanium	plastic
480	1000	1500	1750

The company would like to have at least the following stock at the end of the year:

copper	silicon	germanium	plastic
200	500	500	1000

Each material can be acquired at each month, but the unit prices of each material type vary as follows:

material	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
copper	1	1	1	2	2	3	3	2	2	1	1	2
silicon	4	3	3	3	5	5	6	5	4	3	3	5
germanium	5	5	5	3	3	3	3	2	3	4	5	6
plastic	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1

The inventory is limited by a capacity of a total of 9000 units per month, regardless of the composition of products in stock. The holding costs of the inventory are 0.05€ per unit per month, regardless of the material type. Due to budget constraints, Caroline cannot spend more than 5000 per month on new material acquisition.

Note that Caroline aims at minimizing the acquisition and holding costs of the materials while meeting the required quantities for production. The production is made to order, meaning that no inventory of chips is kept. Please help Caroline to model the material planning and solve it with the data above.

2.5.1 The model

Define the set $P = \{\text{copper}, \text{silicon}, \text{germanium}, \text{plastic}\}$ and T the set of the 12 months of the year. Define the variables $x_{pt} \geq 0$ as being the amount of product $p \in P$ acquired in period $t \in T$.

Let $s_{pt} \geq 0$ be the amount of product $p \in P$ left in stock at the end of period $t \in T$. Note that these values are uniquely determined by the x variables, but we define additional variables to ease the modelling.

If π_{pt} is the unit price of product $p \in P$ in time $t \in T$ and h_{pt} the unit holding costs (which happen to be constant) we can express the objective as:

$$\min \sum_{p \in P} \sum_{t \in T} \pi_{pt} x_{pt} + \sum_{p \in P} \sum_{t \in T} h_{pt} s_{pt}$$

The constraints are easy to express as well. If $\beta \geq 0$ denotes the monthly acquisition budget, the budget constraint can be expressed as:

$$\sum_{p \in P} \pi_{pt} x_{pt} \leq \beta \quad \forall t \in T.$$

Further, we constrain the inventory to be always the storage capacity $\ell \geq 0$ using:

$$\sum_{p \in P} s_{pt} \leq \ell \quad \forall t \in T.$$

The constraints below define s_{pt} by balancing the acquired amounts with the previous inventory and the demand δ_{pt} which for each month is implied by the total demand for the chips of both types. Note that $t - 1$ is defined as the initial stock when t is the first period, that is `January`. This can be obtained with additional variables s made equal to those values or with a rule that specializes, as in the code below.

$$x_{pt} + s_{p,t-1} = \delta_{pt} + s_{pt} \quad \forall p \in P, t \in T.$$

Finally, we capture the required minimum inventory levels in December with the constraint.

$$s_{p\text{Dec}} \geq \Omega_p \quad \forall p \in P,$$

where $(\Omega_p)_{p \in P}$ is the vector with the desired end inventories.

2.5.2 The Pyomo implementation

```
demand_data = '''chip,Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec
Logic,88,125,260,217,238,286,248,238,265,293,259,244
Memory,47,62,81,65,95,118,86,89,82,82,84,66'''
demand_chips = pd.read_csv( StringIO(demand_data), index_col='chip'

price_data = '''product,Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec
copper,1,1,1,2,2,3,3,2,2,1,1,2
silicon,4,3,3,3,5,5,6,5,4,3,3,5
germanium,5,5,5,3,3,3,3,2,3,4,5,6
plastic,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1'''
price = pd.read_csv( StringIO(price_data), index_col='product' )

# We first define a simple dataframe describing the material usage for each microchip type
use = dict()
use['Logic'] = { 'silicon' : 1, 'plastic' : 1, 'copper' : 4 }
use['Memory'] = { 'germanium' : 1, 'plastic' : 1, 'copper' : 2 }
use = pd.DataFrame.from_dict( use ).fillna(0).astype( int )
```

```
18
19 # Using the microchip demand, we calculate how much of each raw material we need each month using a matrix multiplication
20 demand = use.dot( demand_chips )
21
22 def BIMProductAcquisitionAndInventory( demand, acquisition_price, existing, desired, stock_limit, month_budget ):
23     m = pyo.ConcreteModel( 'Product acquisition and inventory' )
24
25     periods  = demand.columns
26     products = demand.index
27     first    = periods[0]
28     prev     = { j : i for i,j in zip(periods,periods[1:]) }
29     last     = periods[-1]
30
31     m.T = pyo.Set( initialize=periods )
32     m.P = pyo.Set( initialize=products )
33
34     m.PT = m.P * m.T # to avoid internal set bloat
35
36     m.x = pyo.Var( m.PT, within=pyo.NonNegativeReals )
37     m.s = pyo.Var( m.PT, within=pyo.NonNegativeReals )
38
39     @m.Param( m.PT )
40     def pi(m,p,t):
41         return acquisition_price.loc[p][t]
42
43     @m.Param( m.PT )
44     def h(m,p,t):
45         return .05 # the holding cost
46
47     @m.Param( m.PT )
48     def delta(m,t,p):
49         return demand.loc[t,p]
50
51     @m.Expression()
52     def acquisition_cost( m ):
53         return pyo.quicksum( m.pi[p,t] * m.x[p,t] for p in m.P for t in m.T )
54
55     @m.Expression()
56     def inventory_cost( m ):
57         return pyo.quicksum( m.h[p,t] * m.s[p,t] for p in m.P for t in m.T )
58
59     @m.Objective( sense=pyo.minimize )
60     def total_cost( m ):
61         return m.acquisition_cost + m.inventory_cost
62
63     @m.Constraint( m.PT )
64     def balance( m, p, t ):
65         if t == first:
66             return existing[p] + m.x[p,t] == m.delta[p,t] + m.s[p,t]
67         else:
68             return m.x[p,t] + m.s[p,prev[t]] == m.delta[p,t] + m.s[p,t]
69
70     @m.Constraint( m.P )
71     def finish( m, p ):
72         return m.s[p,last] >= desired[p]
73
74     @m.Constraint( m.T )
75     def inventory( m, t ):
76         return pyo.quicksum( m.s[p,t] for p in m.P ) <= stock_limit
```

```
77
78      @m.Constraint( m.T )
79      def budget( m, t ):
80          return pyo.quicksum( m.pi[p,t]*m.x[p,t] for p in m.P ) <= month_budget
81
82      return m
83
84  m = BIMProductAcquisitionAndInventory( demand, price,
85              {'silicon' : 1000, 'germanium': 1500, 'plastic': 1750, 'copper' : 4800 },
86              {'silicon' :  500, 'germanium':  500, 'plastic': 1000, 'copper' : 2000 },
87              9000, 2000 )
88
89  pyo.SolverFactory( 'cbc' ).solve(m)
```

2.5.3 The optimal solution

Tables 2.4 and 2.5 report the optimal solution in terms of material acquisition and stock levels. The latter are also displayed below in Figure 2.6.

material	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
silicon	0.0	0.0	0.0	965.0	0.0	0.0	0.0	0.0	0.0	1078.1	217.9	0.0
plastic	0.0	0.0	0.0	0.0	0.0	0.0	266.0	327.0	347.0	375.0	343.0	1310.0
copper	0.0	0.0	3548.0	0.0	0.0	0.0	0.0	0.0	962.0	1336.0	4312.0	0.0
germanium	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Table 2.4: The optimal material acquisitions over the entire year

material	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
silicon	912.0	787.0	527.0	1275.0	1037.0	751.0	503.0	265.0	0.0	785.1	744.0	500.0
plastic	1615.0	1428.0	1087.0	805.0	472.0	68.0	0.0	0.0	0.0	0.0	0.0	1000.0
copper	4354.0	3730.0	6076.0	5078.0	3936.0	2556.0	1392.0	262.0	0.0	0.0	3108.0	2000.0
germanium	1453.0	1391.0	1310.0	1245.0	1150.0	1032.0	946.0	857.0	775.0	693.0	609.0	543.0

Table 2.5: The optimal stock levels over the entire year

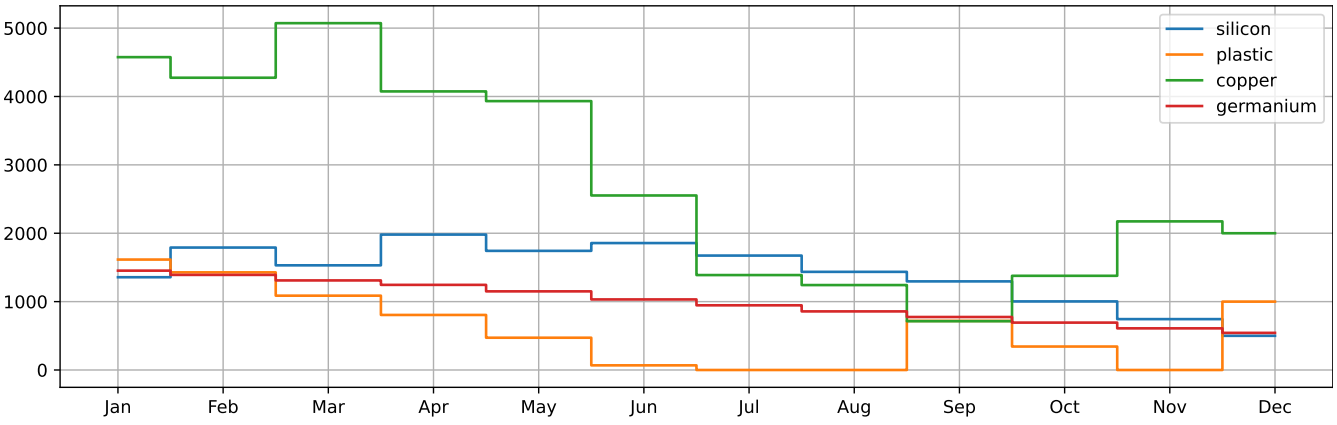


Figure 2.6: The optimal stock levels over the entire year

Looking at the solution, we can note that:

- The monthly budget is never a limitation;
- With the given budget the solution remains integer;
- Lowering the budget to 2000 forces acquiring fractional quantities, we show that solution below;
- Lower values of the budget end will up making the problem infeasible.

material	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
silicon	444.7	559.0	0.0	666.7	0.0	400.0	65.0	0.0	125.6	0.0	0.0	0.0
plastic	0.0	0.0	0.0	0.0	0.0	0.0	266.0	327.0	1065.0	0.0	0.0	1310.0
copper	221.3	323.0	2000.0	0.0	1000.0	0.0	0.0	983.6	695.5	2000.0	2000.0	934.5
germanium	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Table 2.6: The optimal material acquisitions over the entire year in the case of a budget of 2000

material	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
silicon	1356.7	1790.7	1530.7	1980.3	1742.3	1856.3	1673.4	1435.4	1296.0	1003.0	744.0	500.0
plastic	1615.0	1428.0	1087.0	805.0	472.0	68.0	0.0	0.0	718.0	343.0	0.0	1000.0
copper	4575.3	4274.3	5072.3	4074.3	3932.3	2552.3	1388.3	1242.0	713.5	1377.5	2173.5	2000.0
germanium	1453.0	1391.0	1310.0	1245.0	1150.0	1032.0	946.0	857.0	775.0	693.0	609.0	543.0

Table 2.7: The optimal stock levels over the entire year in the case of a budget of 2000

Exercises

*Exercises are labeled as **C** (calculating), **M** (modeling), **T** (theoretical) or combination of these, reflecting on which of these three aspects they focus. The most difficult/lengthy exercises are labeled with a ⋆.*

Ex. 2.1 (M) Consider a company which is working on a project. The project consists of two activities on which the company can work simultaneously. The project terminates when both the two activities are completed. The aim of the company is to complete the project as soon as possible.

According to the current estimations, activity 1 will be completed in 200 days while activity 2 requires 100 days. To speed up the process, the company is willing to deploy additional resources. A total budget of €500.000 that can be used to reduce the completing time of the project is made available. The company estimates that additional resources will impact on the two activities as follows:

- for each €10.000 spent on activity 1, the completion time of the activity is reduced by 3 days;
- for each €10.000 spent on activity 2, the completion time of the activity is reduced by 2 days;
- for each activity, investing less than €10.000 will not decrease the completion time (e.g., investing €9.999 does not reduce the completion time, investing €19.999 reduces the time by 3 days for activity 1 and 2 days for activity 2).

What is the allocation of the budget that allows to complete the project as soon as possible? Formulate this problem as a mathematical optimization problem.

Ex. 2.2 (C+T) Consider the following problem

$$\begin{aligned} \min \quad & |x_1| + |x_2| \\ \text{s.t.} \quad & 2x_1 + x_2 \geq 3 \\ & x_1 \geq -1 \\ & x_2 \geq 2. \end{aligned}$$

(a) Represent graphically and solve it.

(b) Reformulate the problem as a linear problem

(⋆ c) Prove that the technique used in (b) correctly models the minimization of a sum of absolute values.
Note: Your proof should hold for any optimization problem in which we minimize the sum of absolute values, not just for the specific instance of this question.

Ex. 2.3 (C) Consider the following optimization problem.

$$\begin{aligned} \max \quad & \frac{5x_1 + 6x_2}{2x_2 + 7} \\ \text{s.t.} \quad & 2x_1 + 3x_2 \leq 6 \\ & 2x_1 + x_2 \leq 3 \\ & x_1, x_2 \geq 0. \end{aligned}$$

Formulate the problem as a linear problem and solve it.

Ex. 2.4 (M) A truck driver has to deliver packages to houses 1, 2 and 3, in this specific order. When driving at normal speed it takes him a_1 to get from its current position to house 1, a_2 from house 1 to house 2 and a_3 to drive from house 2 to house 3. The driver has a deadline d_i by which the corresponding package should be delivered at the house i, for $i = 1, 2, 3$.

The driver has the option to slow down or speed up his normal speed, more specifically it takes the driver $a_i x_i$ to arrive at house i from the previous stop for $x_i \in \mathbb{R}^+$. More specifically, if $x_i > 1$, then the drivers is driving slower than the nominal speed and the trip to house i takes longer and the reverse scenario when $0 < x_i < 1$. However, the driver does not want to drive too fast for a long time, thus we impose the constraints $x_1 + x_2 + x_3 \geq 2.5$ and $x_1, x_2, x_3 \geq 0.5$.

The goal is to minimize the total delivery delay. The delivery delay to each house is equal to the difference between the arrival time of the driver and the scheduled delivery, but only when he/she is late.
Note: You may assume that once the driver arrives at a customer, he/she immediately drives to the next customer.

(a) Formulate the problem above as a LP.

(b) In the aforementioned model it may occur that the driver arrives before d_1 at the first house, in order to be on time for the second delivery. In some situations it might be undesirable that a driver arrives before its deadline (e.g., when the driver has to deliver a parcel in a specific time window). Adjust the previous problem such that the driver is penalized not only when he/she is too late, but also (proportionally) when he/she is too early.

Ex. 2.5 (C) Consider the dual problem (2.14) derived in Chapter 2, treat it as if it was a primal problem and derives its dual, and check that the original primal problem (2.15) is re-obtained.

$$\begin{array}{ll} \max & 3\lambda_1 - 5\lambda_2 + 7\lambda_3 + 11\lambda_4 \\ \text{s.t.} & \lambda_1 \in \mathbb{R} \\ & \lambda_2 \geq 0 \\ & \lambda_3 \leq 0 \\ & \lambda_4 \leq 0 \\ & -\lambda_1 + 2\lambda_2 + \lambda_4 \leq 1 \\ & 3\lambda_1 - \lambda_2 + \lambda_4 \geq -2 \\ & 3\lambda_2 + \lambda_3 - \lambda_4 = 4. \end{array} \tag{2.14}$$

$$\begin{array}{ll} \min & x_1 - 2x_2 + 4x_3 \\ \text{s.t.} & -x_1 + 3x_2 = 3 \\ & 2x_1 - x_2 + 3x_3 \geq -5 \\ & x_3 \leq 7 \\ & x_1 + x_2 - x_3 \leq 11 \\ & x_1 \geq 0 \\ & x_2 \leq 0 \\ & x_3 \in \mathbb{R}. \end{array} \tag{2.15}$$

Ex. 2.6 (M) In a smartphone shop, in view of the arrival of new models, a salesman wants to sell off quickly its stock composed of eight phones, four hands-free kits and nineteen prepaid cards. Thanks to a market study, she knows that she can propose an offer with a phone and two prepaid cards and that this offer will bring in a profit of seven euros. Similarly, she can prepare a box with a phone, a hands-free kit and three prepaid cards, yielding a profit of nine euros. She is assured to be able to sell any quantity of these two offers within the availability of the stock.

(a) Give a LP that determines which quantity of each offer should the salesman prepare to maximize its net profit.

(b) A sales representative of a supermarket chain proposes to buy its stock (the products, not the offers). The salesman wants to know the minimum unit prices she should negotiate for each product (phone, hands-free kits, and prepaid cards) in order to obtain at least the same profit as in (a). Formulate a LP to decides these prices.

Ex. 2.7 (C) Consider the following LP and use its dual to decide if $\boldsymbol{x} = (x_1, x_2) = (1, 4)$ is the optimal solution or not.

$$\begin{array}{ll} \max & x_1 - x_2 \\ \text{s.t.} & -2x_1 + x_2 \leq 2, \\ & x_1 - 2x_2 \leq 2, \\ & x_1 + x_2 \leq 5, \\ & \boldsymbol{x} \geq \mathbf{0}. \end{array}$$

Ex. 2.8 (C) Use duality to show that the following problem is infeasible:

$$\begin{array}{ll} \min & 2x_1 - x_2 \\ \text{s.t.} & x_1 + x_2 \geq 1, \\ & -x_1 - x_2 \geq 1, \\ & \boldsymbol{x} \geq \mathbf{0}. \end{array}$$

Ex. 2.9 (T) Consider the following linear optimization problem and assume that it has a feasible and bounded solution.

$$\begin{array}{ll} \max & \boldsymbol{c}^\top \boldsymbol{x} \\ \text{s.t.} & A\boldsymbol{x} \leq \mathbf{0}, \\ & \boldsymbol{x} \geq \mathbf{0}. \end{array}$$

(a) Write down the dual of the above problem.

(b) Find the optimal solution of the above problem.

(c) What can you say about the constraint set for the above problem?
(Hint: if it helps, think first at the situation with only two x-variables).

Exercise 5 (M)

A textile firm is capable of producing three products and let x_1, x_2 and x_3 denote the quantities of these products. The production plan for the next month must satisfy the constraints:

$$\begin{aligned} &x_1 + 2x_2 + 2x_3 \leq 12, \\ &2x_1 + 4x_2 + x_3 \leq c, \\ &x_1, x_2, x_3 \geq 0. \end{aligned}$$

The first constraint is determined by equipment availability and is fixed. The second constraint is determined by the availability of cotton c. The net profits of the products are 2, 3 and 3 respectively, minus the cost of cotton and fixed costs. Find the value of the dual variable corresponding to the cotton constraint as a function of c. Moreover, also determine the profit minus the cost of cotton as a function of c.

Ex. 2.10 (T, $\star$) Given positive constants C, p_i, and w_i for $i = 1, \ldots, n$, consider the following LP:

$$\begin{aligned} \min \quad & Cy + \sum_{i=1}^{n} z_i \\ \text{s.t.} \quad & w_i y + z_i \geq p_i, && i = 1, \ldots, n, \\ & z_i \geq 0, && i = 1, \ldots, n, \\ & y \geq 0. \end{aligned}$$

(a) Intuitively, what is the largest number of n for which you can solve *by hand* (i.e., without a computer) this problem for any values of C, p and w? How would you do that?

(b) Formulate the dual of the problem.

(c) Intuitively, what is the largest number of n for which you can solve *by hand* (i.e., without a computer) the dual? How would you do that? *(Hint: See Exercise 9 of the Exercise Sheet 1.1 on ILP)*

(d) Could you use the solution from (c) to get the optimal values for y and z_i's in the original problem?

[Back to the start of Chapter 2] [Back to Table of Contents]

Chapter 3

Mixed-integer linear programming

3.1 Formulation

The particular feature of the linear programs was that as long as the decision variables satisfy all the constraints, they can take any real value. However, there are many situations in which it makes sense to restrict the solution space in a way that cannot be expressed using linear (in-)equality constraints. For example, some numbers might need to be integers, such as the number of people to be assigned to a task. Another situation is where certain constraints are to hold only if another constraint holds – for example, the amount of power generated in a coal plant taking at least its minimum value only if the generator is turned on.

None of these two examples can be formulated using linear constraints alone. For such and many other situations, it is often possible that the problem can still be modelled as an LP, yet with an extra restriction that some variables need to take integer values only.

An mixed-integer linear program (MILP) is an LP problem in which some variables are constrained to be integers. Formally, it is defined as

$$\begin{aligned} \min_{\boldsymbol{x}\in\mathbb{R}^n} \quad & \boldsymbol{c}^\top \boldsymbol{x} \\ \text{s.t.} \quad & A\boldsymbol{x} \leq b, \\ & x_i \in \mathbb{Z}, \quad i \in \mathcal{I}, \end{aligned}$$

where $\mathcal{I} \subseteq \{1, \ldots, n\}$ is the set of indices identifying the variables that take integer values. Of course, if the decision variable are required to be nonnegative, we could use the set $\mathbb{N}$ instead of $\mathbb{Z}$ A special case of integer variables are *binary variables*, which can take only values in $\mathbb{B} = \{0, 1\}$. Consider the following example.

Example 9 (Building microchips pt.4 – Adjusting for wasted material)**.** *The company BIM realizes that a* 1% *fraction of the copper always gets wasted while producing both types of microchips, more specifically* 1% *of the required amount. This means that it actually takes* 4.04 *gr of copper to produce a logic chip and* 2.02 *gr of copper to produce a memory chip. If we rewrite the linear problem in Example 1 and modify accordingly the coefficients in the corresponding constraints, we obtain the following problem*

$$\begin{aligned} \max \quad & 12x_1 + 9x_2 & \\ \text{s.t.} \quad & x_1 \leq 1000 & \text{(silicon)} \\ & x_2 \leq 1500 & \text{(germanium)} \\ & x_1 + x_2 \leq 1750 & \text{(plastic)} \\ & 4.04x_1 + 2.02x_2 \leq 4800 & \text{(copper with waste)} \\ & x_1, x_2 \geq 0. & \end{aligned}$$

If we solve again we obtain a different optimal solution than the original one, namely $(x_1, x_2) \approx (626.238, 1123.762)$ *and an optimal value of roughly* 17628.713*. Note, in particular, that this new optimal solution is not integer, but on the other hand in the LP above there is no constraint requiring* x_1 *and* x_2 *to be such.*

In terms of production, of course we would simply produce entire chips but it is not clear how to implement the fractional solution $(x_1, x_2) \approx (626.238, 1123.762)$. *Rounding down to* $(x_1, x_2) = (626, 1123)$ *will intuitively yield a feasible solution, but we might be giving away some profit and/or not using efficiently the available material. Rounding up to* $(x_1, x_2) = (627, 1124)$ *could possibly lead to an unfeasible solution for which the available material is not enough. We can of course manually inspect by hand all these candidate integer solutions, but if the problem involved many more decision variables or had a more complex structure, this would become much harder and possibly not lead to the true optimal solution.*

A much safer approach is to explicitly require the two decision variables to be nonnnegative integers, thus transforming the original into the following MILP:

$$\begin{aligned} \max \quad & 12x_1 + 9x_2 & \\ \text{s.t.} \quad & x_1 \leq 1000 & \text{(silicon)} \\ & x_2 \leq 1500 & \text{(germanium)} \\ & x_1 + x_2 \leq 1750 & \text{(plastic)} \\ & 4.04x_1 + 2.02x_2 \leq 4800 & \text{(copper with waste)} \\ & x_1, x_2 \in \mathbb{N}. & \end{aligned}$$

The optimal solution is $(x_1, x_2) = (626, 1124)$ *with a profit of* 17628. *Note that for this specific problem both the naive rounding strategies outlined above would have not yield the true optimal solution. The Python code for obtaining the optimal solution using MILP solvers is given below.*

```
m    = pyo.ConcreteModel('BIMperturbed')

m.x1 = pyo.Var( within=pyo.NonNegativeReals )
m.x2 = pyo.Var( within=pyo.NonNegativeReals )

m.obj       = pyo.Objective(expr  =    12*m.x1 +    9*m.x2, sense= pyo.maximize)

m.silicon   = pyo.Constraint(expr =       m.x1             <= 1000 )
m.germanium = pyo.Constraint(expr =                  m.x2 <= 1500)
m.plastic   = pyo.Constraint(expr =       m.x1 +     m.x2 <= 1750)
m.copper    = pyo.Constraint(expr =  4.04*m.x1 + 2.02*m.x2 <= 4800)
```

MILP naturally applies to situations in which we need to deal with integer numbers, as when scheduling people, as the following extensive example illustrates.

Example 10 (Shift scheduling)**.** *This example concerns a model for scheduling weekly shifts for a small campus food store. It is inspired by a Towards Data Science article, whose original implementation has been revised. Let us look at the problem description from the original article.*

> *A new food store has been opened at the University Campus which will be open 24 hours a day, 7 days a week. Each day, there are three eight-hour shifts. Morning shift is from 6:00 to 14:00, evening shift is from 14:00 to 22:00 and night shift is from 22:00 to 6:00 of the next day. During the night there is only one worker while during the day there are two, except on Sunday that there is only one for each shift. Each worker will not exceed a maximum of 40 hours per week and have to rest for 12 hours between two shifts. As for the weekly rest days, an employee who rests one Sunday will also prefer to do the same that Saturday. In principle, there are available ten employees, which is clearly over-sized. The less the workers are needed, the more the resources for other stores.*

This problem requires assignment of N *workers to a predetermined set of shifts. There are three shifts per day, seven days per week. These observations suggest the need for three ordered sets:*

- WORKERS *with* N *elements representing workers.*
- DAYS *with labeling the days of the week.*
- SHIFTS *labeling the shifts each day.*

The problem describes additional considerations that suggest the utility of several additional sets.

- SLOTS *is an ordered set of (day, shift) pairs describing all of the available shifts during the week.*
- BLOCKS *is an order set of all overlapping 24 hour periods in the week. An element of the set contains the (day, shift) period in the corresponding period. This set will be used to limit worker assignments to no more than one for each 24 hour period.*
- WEEKENDS *is a the set of all (day, shift) pairs on a weekend. This set will be used to implement worker preferences on weekend scheduling.*

These additional sets improve the readability of the model.

$$\begin{aligned}
\text{WORKERS} &= \{w_1, w_2, \ldots, w_1\} \text{ set of all workers} \\
\text{DAYS} &= \{\text{Mon}, \text{Tues}, \ldots, \text{Sun}\} \text{ days of the week} \\
\text{SHIFTS} &= \{\text{morning}, \text{evening}, \text{night}\} \text{ 8 hour daily shifts} \\
\text{SLOTS} &= \text{DAYS} \times \text{SHIFTS} \text{ ordered set of all (day, shift) pairs} \\
\text{BLOCKS} &\subset \text{SLOTS} \times \text{SLOTS} \times \text{SLOTS} \text{ all 24 blocks of consecutive slots} \\
\text{WEEKENDS} &\subset \text{SLOTS} \text{ subset of slots corresponding to weekends}
\end{aligned}$$

The model parameters are

$$\begin{aligned}
N &= \text{ number of workers} \\
\text{WorkersRequired}_{d,s} &= \text{ number of workers required for each day, shift pair } (d, s)
\end{aligned}$$

The decision variables are

$$\text{assign}_{w,d,s} = \begin{cases} 1 & \text{if worker } w \text{ is assigned to day, shift pair } (d,s) \in \text{SLOTS} \\ 0 & \text{otherwise} \end{cases}$$

$$\text{weekend}_w = \begin{cases} 1 & \text{if worker } w \text{ is assigned to a weekend day, shift pair } (d,s) \in \text{WEEKENDS} \\ 0 & \text{otherwise} \end{cases}$$

$$\text{needed}_w = \begin{cases} 1 & \text{if worker } w \text{ is needed during the week} \\ 0 & \text{otherwise} \end{cases}$$

Let us now look at the model constraints. Assign workers to each shift to meet staffing requirement.

$$\sum_{w \in \text{ WORKERS}} \text{assign}_{w,d,s} \geq \text{WorkersRequired}_{d,s} \qquad \forall (d,s) \in \text{SLOTS}$$

Assign no more than 40 hours per week to each worker.

$$\sum_{d,s \in \text{ SLOTS}} \text{assign}_{w,d,s} \leq 40 \qquad \forall w \in \text{WORKERS}$$

Assign no more than one shift in each 24 hour period.

$$\text{assign}_{w,d_1,s_1} + \text{assign}_{w,d_2,s_2} + \text{w, assign}_{w,d_3,s_3} \leq 1 \qquad \begin{aligned} &\forall w \in \text{WORKERS} \\ &\forall ((d_1,s_1),(d_2,s_2),(d_3,s_3)) \in \text{BLOCKS} \end{aligned}$$

Indicator if worker has been assigned any shift.

$$\sum_{d,s \in \text{ SLOTS}} \text{assign}_{w,d,s} \leq M_{\text{SLOTS}} \cdot \text{needed}_w \qquad \forall w \in \text{WORKERS}$$

Indicator if worker has been assigned a weekend shift.

$$\sum_{d,s \in \text{ WEEKENDS}} \text{assign}_{w,d,s} \leq M_{\text{WEEKENDS}} \cdot \text{weekend}_w \qquad \forall w \in \text{WORKERS}$$

The model objective is to minimize the overall number of workers needed to fill the shift and work requirements while also attempting to meet worker preferences regarding weekend shift assignments. This is formulated here as an objective for minimizing a weighted sum of the number of workers needed to meet all shift requirements and the number of workers assigned to weekend shifts. The positive weight γ determines the relative importance of these two measures of a desirable shift schedule.

$$\min \left(\sum_{w \in \text{ WORKERS}} \text{needed}_w + \gamma \sum_{w \in \text{ WORKERS}} \text{weekend}_w) \right)$$

A Pyomo implementation of this model is as follow.

```
def shift_schedule(N, hours=40):

    m = pyo.ConcreteModel('workforce')

    # ordered set of avaiable workers
    m.WORKERS = pyo.Set(initialize=[f"W{i:02d}" for i in range(1, N+1)])

    # ordered sets of days and shifts
    m.DAYS = pyo.Set(initialize=['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun'])
    m.SHIFTS = pyo.Set(initialize=['morning', 'evening', 'night'])

    # ordered set of day, shift time slots
    m.SLOTS = pyo.Set(initialize = m.DAYS * m.SHIFTS)

    # ordered set of 24 hour time blocks
    m.BLOCKS = pyo.Set(initialize = [[m.SLOTS.at(i), m.SLOTS.at(i+1), m.SLOTS.at(i+2)]
                                     for i in range(1, len(m.SLOTS)-1)])

    # ordered set of weekend shifts
    m.WEEKENDS = pyo.Set(initialize = m.SLOTS, filter = lambda m, day, shift: day in ['Sat', 'Sun'])

    # parameter of worker requirements
    @m.Param(m.SLOTS)
    def WorkersRequired(m, day, shift):
        if shift in ['night'] or day in ['Sun']:
            return 1
        return 2

    m.Hours = pyo.Param(mutable=True, default=hours)

    # decision variable: assign[worker, day, shift] = 1 assigns worker to a time slot
    m.assign = pyo.Var(m.WORKERS, m.SLOTS, domain=pyo.Binary)

    # decision variables: weekend[worker] = 1 worker is assigned weekend shift
    m.weekend = pyo.Var(m.WORKERS, domain=pyo.Binary)

    # decision variable: needed[worker] = 1
    m.needed = pyo.Var(m.WORKERS, domain=pyo.Binary)

    # assign a sufficient number of workers for each time slot
    @m.Constraint(m.SLOTS)
    def required_workers(m, day, shift):
        return m.WorkersRequired[day, shift] == sum(m.assign[worker, day, shift] for worker in m.WORKERS)

    # workers limited to forty hours per week assuming 8 hours per shift
    @m.Constraint(m.WORKERS)
    def forty_hour_limit(m, worker):
```

```
        return 8*sum(m.assign[worker, day, shift] for day, shift in m.SLOTS) <= m.Hours

    # workers are assigned no more than one time slot per 24 time block
    @m.Constraint(m.WORKERS, m.BLOCKS)
    def required_rest(m, worker, d1, s1, d2, s2, d3, s3):
        return m.assign[worker, d1, s1] + m.assign[worker, d2, s2] + m.assign[worker, d3, s3] <= 1

    # determine if a worker is assigned to any shift
    @m.Constraint(m.WORKERS)
    def is_needed(m, worker):
        return sum(m.assign[worker, day, shift] for day, shift in m.SLOTS) <= len(m.SLOTS)*m.needed[worker]

    # determine if a worker is assigned to a weekend shift
    @m.Constraint(m.WORKERS)
    def is__weekend(m, worker):
        return 6*m.weekend[worker] >= sum(m.assign[worker, day, shift] for day, shift in m.WEEKENDS)

    # minimize a blended objective of needed workers and needed weekend workers
    @m.Objective(sense=pyo.minimize)
    def minimize_workers(m):
        return sum(i*m.needed[worker] + 0.1*i*m.weekend[worker] for i, worker in enumerate(m.WORKERS))

    pyo.SolverFactory('cbc').solve(m)

    return m

m = shift_schedule(10, 40)
```

Figure 3.1 is a visual representation of the optimal shift schedule obtain for a specific instance of the problem.

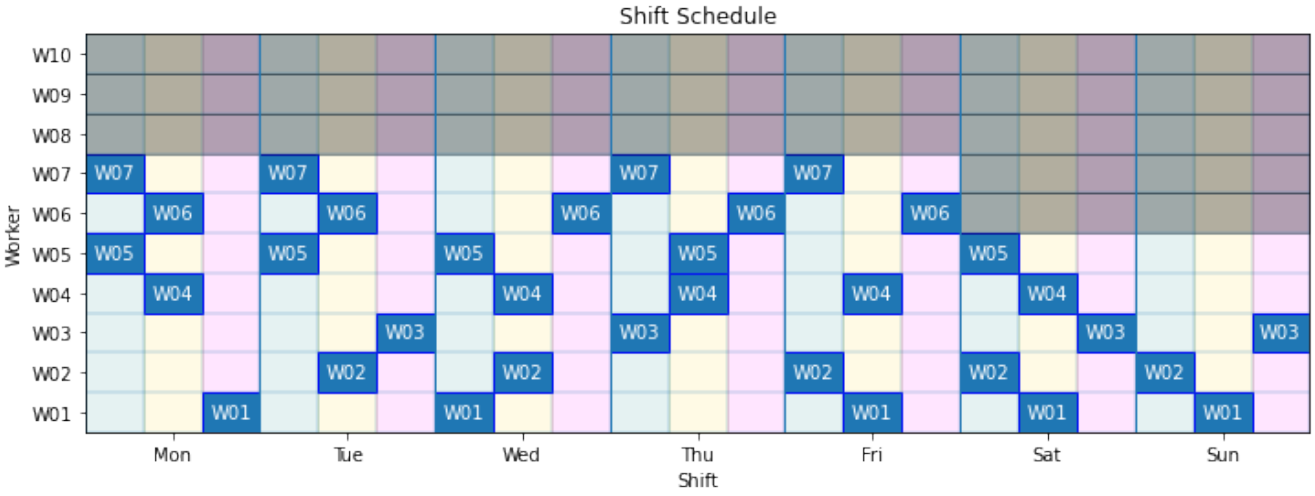


Figure 3.1: The optimal shift schedule.

In the previous chapter, we claimed that every optimization problem that can be formulated as an LP is 'easy' to solve. Does that mean that, in contrast, every MILP problem is easy to solve? Not necessarily, but due to the significantly greater modelling capacities of MILP, it can be indeed used to model problems that are fundamentally 'difficult', i.e., for which no efficient solution procedure is known, even if tools other than MILP are allowed. Here, we present an example of a classical problem like this – the knapsack problem – which can be used to model many resource allocation situations, e.g., on computational clusters.

Example 11 (Resource allocation – Knapsack 0-1 problem)**.** *A traveler can only bring a single fixed-weight knapsack and must fill it with the most valuable items. Given a finite set of n items, where each item i has a weight w_i and a value v_i, we want to select the subset of items to put in the knapsack so that the total weight is less than or equal to*

a given limit W and the total value is as large as possible. It can be formulated as an MILP as follows:

$$\begin{aligned} \max_{\boldsymbol{x}\in\mathbb{R}^n} \quad & \sum_{i=1}^{n} v_i x_i \\ \text{s.t.} \quad & \sum_{i=1}^{n} w_i x_i \leq W, \\ & x_i \in \mathbb{B}, \quad i = 1, \ldots, n. \end{aligned}$$

The knapsack problem is a one of the most fundamental combinatorial optimization problems whose variants often arise in resource allocation where the decision makers have to choose a subset of non-divisible tasks/resources under a fixed time/budget constraint, respectively. General versions of this problem are routinely solved on online computational clusters, for example.

This problem is known to be NP-complete which, roughly, means that it is widely believed that for such a problem there exists no algorithm that would not need to check, on a worst-case instance all 2^n solutions.

As visible from the example, our enthusiasm of modelling a problem we encounter as an MILP can sometimes lead us to, accidentally, modelling one of the well known NP-hard problems. Does that mean that MILP is an inefficient technology? No, because powerful solvers have been developed for MILPs that allows to solve efficiently optimization problems with thousands of integer variables.

To do that, we need to become familiar with techniques and tricks that allow us to model via integer variables and MILP constraints situations that we could not think of at first encounter. For that reason, similar to the previous chapter, we now present the new modelling opportunities of MILPs in Section 3.2 and then the key solution methods in Section 3.3. At the same time we want to keep you, the reader, aware that for some situations, there exist modelling alternatives other, and sometimes better, than 'forcing' a given constraint into an MILP form through complicated tricks. One of these alternatives is the so-called disjunctive programming which is supported by Pyomo, and in suitable places we will illustrate how the same goal can be achieved as with MILP constraints.

3.2 Modelling techniques

3.2.1 Variables taking a set of discontinuous values

MILP can be used to model variables taking on discontinuous values. For instance when either $x = 0$ or $l \leq x \leq u$ must hold. We introduce a binary variable $y \in \mathbb{B}$ with the interpretation that

$$y = \begin{cases} 0 & \text{if } x = 0, \\ 1 & \text{if } l \leq x \leq u. \end{cases} \tag{3.1}$$

Now we can model the discontinuous variable x by the following linear constraints:

$$\begin{aligned} & x \leq uy, \\ & x \geq ly, \\ & y \in \mathbb{B}. \end{aligned}$$

Indeed, by studying this system of constraints, you can see that the relationship (3.1) becomes enforced.

3.2.2 Variable enforcing a given constraint or not

In many optimization problems, we want to ensure that a certain constraint $\boldsymbol{a}^\top \boldsymbol{x} \leq b$ holds *only* under specific conditions, which we can capture with a yes-no decision variable, say a binary variable $y \in \mathbb{B}$. The so-called **big-M method** gives a way to construct such an 'optional' constraint by writing the original constraint as

$$\boldsymbol{a}^\top \boldsymbol{x} \leq b + M(1 - y),$$

where $M > 0$ is a large number. In this constraint, if the binary variable y takes a 'yes' value 1, then the right-hand side is equal to b and we recover the original constraints, which then 'needs to hold'. Otherwise, if $y = 0$ the

right-hand side becomes so large due to the $M(1-y) = M$ term that effectively the constraint does not impose any restriction on $\boldsymbol{x}$.

In this example, the value M can typically be easily guessed from the problem properties – it should be large enough to make the trick work, i.e., be greater than $\boldsymbol{a}^\top - b$ for all reasonable x, but it should not too large because that will deteriorate the solver performance. We will now see this trick applied to several specific situations.

3.2.3 Cost function with a fixed component

Often in production planning problems, cost of producing a given good consists of a part which scales with the production size, and a fixed part (machine setup costs, for example). Such a cost function $f(x)$ with a per-unit cost c and a fixed component k given by

$$f(x) = \begin{cases} 0 & \text{if } x = 0, \\ k + cx & \text{if } x > 0, \end{cases}$$

can be modeled by adding a binary variable $y \in \mathbb{B}$ as follows:

$$\begin{aligned} f(x, y) &= ky + cx, \\ x &\leq My, \\ y &\in \mathbb{B}, \end{aligned}$$

where $M > 0$ is a large positive constant that should be an upper bound on x.

3.2.4 Either-or constraints

The either-or constraints require that at least one of the following constraints must hold:

$$\boldsymbol{a}_1^\top \boldsymbol{x} \leq b_1 \qquad \text{or} \qquad \boldsymbol{a}_2^\top \boldsymbol{x} \leq b_2.$$

For this, again the big-M method can be used where we need a new binary variable $y \in \mathbb{B}$ and two large positive constants $M_1, M_2 > 0$. The linearized constraints are then

$$\begin{aligned} \boldsymbol{a}_1^\top \boldsymbol{x} &\leq b_1 + M_1 y, \\ \boldsymbol{a}_2^\top \boldsymbol{x} &\leq b_2 + M_2(1-y), \\ y &\in \mathbb{B}. \end{aligned}$$

Now that you have seen a few example of transforming logical relationships into MILP constraints you may have noticed that although nice, they are a bit artificial in the sense that they require determining the constant M and when you look at the final problem formulation, it might not be easy to see immediately that it corresponds to an 'either-or', or 'if-then' constraint. This has the benefit that a great variety of generic MILP solvers can be used.

Sometimes, however, it may be beneficial to 'remember' about the logical structure when formulating the problem and for that reason, it is a good moment to introduce the concept of disjunctive programming, which is typically not covered in standard textbooks. Disjunctive programming describes a class of optimization problems that include disjunctive ("or") constraints. The advantage of using a disjunctive programming formulation is that it retains and exploits the inherent logic structure of problems and thus reduces the combinatorics. We best illustrate it with an example

Example 12 (Multi-plant production)**.** *Consider the following production problem*

$$\begin{aligned} \max_{x,y \geq 0} \quad & \text{profit} \\ \text{s.t.} \quad & \text{profit} = 40x + 30y \\ & x \leq 40 && \text{(demand)} \\ & x + y \leq 80 && \text{(labor A)} \\ & 2x + y \leq 100 && \text{(labor B)} \end{aligned}$$

and its Pyomo implementation

```python
model = pyo.ConcreteModel("Multi-Product Plant")

# decision variables
model.profit = pyo.Var()
model.production_x = pyo.Var(domain=pyo.NonNegativeReals)
model.production_y = pyo.Var(domain=pyo.NonNegativeReals)

# profit objective
@model.Objective(sense=pyo.maximize)
def maximize_profit(model):
    return  model.profit

# constraints
@model.Constraint()
def profit_expr(model):
    return model.profit == 40*model.production_x + 30*model.production_y

@model.Constraint()
def demand(model):
    return model.production_x <= 40

@model.Constraint()
def laborA(model):
    return model.production_x + model.production_y <= 80

@model.Constraint()
def laborB(model):
    return 2*model.production_x + model.production_y <= 100

# solve
pyo.SolverFactory('cbc').solve(model)
```

The optimal solution is $(x, y) = (20, 60)$*, which results in a profit of* 2600*.*

Labor B is a relatively high cost for the production of product X. A new technology has been developed with the potential to lower cost by reducing the time required to finish product X to 1.5 *hours, but require a more highly skilled labor type C at a unit cost of 60€per hour.*

It is our task to assess if the new technology is beneficial, i.e., whether adopting it would lead to a higher profit. In this situation we have an either-or structure for the objective and for Labor B constraint:

$$\underbrace{\text{profit} = 40x + 30y,\ 2x + y \leq 100}_{\textit{old technology}} \quad \textit{or} \quad \underbrace{\text{profit} = 60x + 30y,\ 1.5x + y \leq 100}_{\textit{new technology}}.$$

Using MILP, we can formulate this problem as follows:

$$\begin{aligned}
\max_{x,y\geq 0, z\in\mathbb{B}} \quad & \text{profit} \\
\text{s.t.} \quad & x \leq 40 && \text{(demand)} \\
& x + y \leq 80 && \text{(labor A)} \\
& \text{profit} \leq 40x + 30y + Mz \\
& \text{profit} \leq 60x + 30y + M(1 - z) \\
& 2x + y \leq 100 + Mz \\
& 1.5x + y \leq 100 + M(1 - z).
\end{aligned}$$

where the variable $z \in \{0, 1\}$ *'activates' the constraints related to the old or new technology, respectively, and* M *is a big enough number. The corresponding Pyomo implementation is given by:*

```
model = pyo.ConcreteModel("Multi-Product Plant MIP")

# decision variables
model.profit = pyo.Var()
model.production_x = pyo.Var(domain=pyo.NonNegativeReals)
model.production_y = pyo.Var(domain=pyo.NonNegativeReals)
model.profit = pyo.Var(domain=pyo.NonNegativeReals)
model.z = pyo.Var(domain=pyo.Binary)
M = 10000

# profit objective
@model.Objective(sense=pyo.maximize)
def maximize_profit(model):
    return  model.profit

# constraints
@model.Constraint()
def profit_constr_1(model):
    return model.profit <= 40*model.production_x + 30*model.production_y + M * model.z

def profit_constr_2(model):
    return model.profit <= 60*model.production_x + 30*model.production_y + M * (1 - model.z)

@model.Constraint()
def demand(model):
    return model.production_x <= 40

@model.Constraint()
def laborA(model):
    return model.production_x + model.production_y <= 80

@model.Constraint()
def laborB_1(model):
    return 2*model.production_x + model.production_y <= 100 + M * model.z

@model.Constraint()
def laborB_2(model):
    return 1.5*model.production_x + model.production_y <= 100 + M * (1 - model.z)

# solve
pyo.SolverFactory('cbc').solve(model)
```

Alternatively, we can formulate our problem using a disjunction, preserving the logical structure, as follows:

$$
\begin{aligned}
\max_{x,y\geq 0} \quad & \text{profit} \\
\text{s.t.} \quad & x \leq 40 && \text{(demand)} \\
& x + y \leq 80 && \text{(labor A)} \\
& \begin{bmatrix} \text{profit} = 40x + 30y \\ 2x + y \leq 100 \end{bmatrix} \veebar \begin{bmatrix} \text{profit} = 60x + 30y \\ 1.5x + y \leq 100 \end{bmatrix}
\end{aligned}
$$

This formulation, if allowed by the software at hand, has the benefit that the software can smartly divide the solution of this problem into sub-possibilities depending on the disjunction. Pyomo natively support disjunctions, as illustrated in the following implementation of the production problem accounting for the new technology option:

```
model = pyo.ConcreteModel()

```

```
model.profit = pyo.Var(bounds=(-1000, 10000))
model.x = pyo.Var(domain=pyo.NonNegativeReals, bounds=(0, 1000))
model.y = pyo.Var(domain=pyo.NonNegativeReals, bounds=(0, 1000))

@model.Objective(sense=pyo.maximize)
def maximize_profit(model):
    return model.profit

@model.Constraint()
def demand(model):
    return model.x <= 40

@model.Constraint()
def laborA(model):
    return model.x + model.y <= 80

@model.Disjunction(xor=True)
def technologies(model):
    return [[model.profit == 40*model.x + 30*model.y,
             2*model.x + model.y <= 100],

            [model.profit == 60*model.x + 30*model.y,
             1.5*model.x + model.y <= 100]]

# solve
pyo.TransformationFactory("gdp.bigm").apply_to(model)
pyo.SolverFactory('cbc').solve(model)
```

The new optimal solution is $(x, y) = (40, 40)$*, which results in a profit of* 3600.

Disjunctive programming is particularly powerful when there are multiple logical constraints that are related to each other, i.e., where there are many logical constraints that interact/might restrict the options for other logical constraints. The following example is a good illustration.

Example 13 (Machine scheduling)**.** *Consider the problem of scheduling a sequence of jobs for a single machine. In this problem, each job has a time at which it is released to the for machine processing, the expected duration of the job, and the due date, summarized by Table 3.1 and Figure 3.2 below. The optimization objective is to find a sequence the jobs on the machine that meets the the due dates. If no such schedule exists, then the objective is to find a schedule minimizing some measure of "badness".*

j	release$_j$	duration$_j$	due$_j$
A	2	5	10
B	5	6	21
C	4	8	15
D	0	4	10
E	0	2	5
F	8	3	15
G	9	2	22

Table 3.1: Summary of the time constraints of all jobs.

To model this problem, we introduce the following three decision variables for every job j*, with the essential one being the time at which the job starts processing:*

- start$_j > 0$ *is the time when job* j *starts,*
- finish$_j > 0$ *is the time when job* j *finishes, and*
- past$_j > 0$ *is how long job* j *is past due.*

Figure 3.2: A traditional means of visualizing scheduling data in the form of a Gantt chart. For each job, the blue interval stands for the time period between the job's release and due times. The (dark) red intervals stand for an example solution of the problem with total time of tasks being past the due time being 68.

Depending on application and circumstances, one could entertain many different choices for objective function. Minimizing the number of past due jobs, or minimizing the maximum past due, or the total amount of time past due would all be appropriate objectives. The following Pyomo model minimizes the total time past due, that is

$$\min \sum_j \text{past}_j$$

The constraints describe the relationships among the decision variables. For example, a job cannot start until it is released for processing

$$\text{start}_j \geq \text{release}_j$$

Once started the processing continues until the job is finished. The finish time is compared to the due time, and the result stored the past_j *decision variable. These decision variables are needed to handle cases where it might not be possible to complete all jobs by the time they are due.*

$$\begin{aligned} \text{finish}_j &= \text{start}_j + \text{duration}_j \\ \text{past}_j &\geq \text{finish}_j - \text{due}_j \\ \text{past}_j &\geq 0 \end{aligned}$$

The final set of constraints require that no pair of jobs be operating on the same machine at the same time. For this purpose, we consider each unique pair (i, j) *where the constraint* $i < j$ *to imposed to avoid considering the same pair twice. Then for any unique pair* i *and* j*, either* i *finishes before* j *starts, or* j *finishes before* i *starts. This is expressed as the family of disjunctions:*

$$\left[\text{finish}_i \leq \text{start}_j\right] \veebar \left[\text{finish}_j \leq \text{start}_i\right] \qquad \forall i < j$$

Such constraints can be enforced using extra binary variable $z_{ij} \in \{0, 1\}$ *as:*

$$\begin{aligned} \text{finish}_i &\leq \text{start}_j + M z_{ij} \\ \text{finish}_j &\leq \text{start}_i + M(1 - z_{ij}). \end{aligned}$$

However, all these disjunctions are highly related to each other because if we have that

$$\text{finish}_i \leq \text{start}_j, \quad \text{finish}_j \leq \text{start}_k,$$

then automatically we must also have

$$\text{finish}_i \leq \text{start}_k,$$

which disjunctive programming is able to utilize very efficiently, as opposed to standard MILP reformulations blind to this structure.

This model and its constraints can be directly translated to Pyomo.

```
# The problem data as a nested Python dictionary of jobs.
# Each job is labeled by a key. For each key there is an associated data dictionary giving:
# - the time at which the job is released to the for machine processing,
# - the expected duration of the job, and
# - the due date.

jobs = pd.DataFrame({
    'A': {'release': 2, 'duration': 5, 'due': 10},
    'B': {'release': 5, 'duration': 6, 'due': 21},
    'C': {'release': 4, 'duration': 8, 'due': 15},
    'D': {'release': 0, 'duration': 4, 'due': 10},
    'E': {'release': 0, 'duration': 2, 'due':  5},
    'F': {'release': 8, 'duration': 3, 'due': 15},
    'G': {'release': 9, 'duration': 2, 'due': 22},
}).T

def build_model(jobs):

    m = pyo.ConcreteModel()

    m.JOBS = pyo.Set(initialize=jobs.index)
    m.PAIRS = pyo.Set(initialize=m.JOBS * m.JOBS, filter = lambda m, i, j: i < j)

    m.start = pyo.Var(m.JOBS, domain=pyo.NonNegativeReals, bounds=(0, 300))
    m.finish = pyo.Var(m.JOBS, domain=pyo.NonNegativeReals, bounds=(0, 300))
    m.past = pyo.Var(m.JOBS, domain=pyo.NonNegativeReals, bounds=(0, 300))

    @m.Constraint(m.JOBS)
    def job_release(m, job):
        return m.start[job] >= jobs.loc[job, "release"]

    @m.Constraint(m.JOBS)
    def job_duration(m, job):
        return m.finish[job] == m.start[job] + jobs.loc[job, "duration"]

    @m.Constraint(m.JOBS)
    def past_due_constraint(m, job):
        return m.past[job] >= m.finish[job] - jobs.loc[job, "due"]

    @m.Disjunction(m.PAIRS, xor=True)
    def machine_deconflict(m, job_a, job_b):
        return [m.finish[job_a] <= m.start[job_b],
                m.finish[job_b] <= m.start[job_a]]

    @m.Objective(sense=pyo.minimize)
    def minimize_past(m):
        return sum(m.past[job] for job in m.JOBS)

    pyo.TransformationFactory("gdp.bigm").apply_to(m)
    return m

def solve_model(m, solver_name="cbc"):
```

```
    solver = pyo.SolverFactory(solver_name)
    solver.solve(m)
    schedule = pd.DataFrame({
        "start" : {job: m.start[job]() for job in m.JOBS},
        "finish" : {job: m.finish[job]() for job in m.JOBS},
        "past" : {job: m.past[job]() for job in m.JOBS},
    })
    return schedule

model = build_model(jobs)
schedule = solve_model(model)
```

3.2.5 If-then constraints

The if-then condition requires that if one condition, say

$$A: \qquad \boldsymbol{a}_1^\top \boldsymbol{x} \leq b_1,$$

has to hold, then *another* condition, say

$$B: \qquad \boldsymbol{a}_2^\top \boldsymbol{x} \leq b_2$$

must hold. A situation like this can still be encoded as a linear model as follows. First notice that the implication $A \Rightarrow B$ is logically equivalent to $\overline{A} \vee B$. Using this trick, the if-then condition is logically equivalent to requiring The if-then condition requires that if a condition A holds, say $\boldsymbol{a}_1^\top \boldsymbol{x} \leq b_1$, then *another* condition B, say $\boldsymbol{a}_2^\top \boldsymbol{x} \leq b_2$ must hold. A situation like this can be encoded as a linear model using the big-M method.

First notice that the implication $A \Rightarrow B$ is logically equivalent to $\overline{A} \vee B$. Using this trick, the if-then condition is logically equivalent to requiring

$$\boldsymbol{a}_1^\top \boldsymbol{x} > b_1 \qquad \text{or} \qquad \boldsymbol{a}_2^\top \boldsymbol{x} \leq b_2.$$

Introducing two large constants $M_1, M_2 > 0$ and a binary variable y, the either-or constraint is equivalent to

$$\begin{aligned} \boldsymbol{a}_1^\top \boldsymbol{x} &> b_1 - M_1 y, \\ \boldsymbol{a}_2^\top \boldsymbol{x} &\leq b_2 + M_2(1-y), \\ y &\in \mathbb{B}, \end{aligned}$$

Here, one needs to be careful because in MILP, strict constraints of the form $\boldsymbol{a}_1^\top \boldsymbol{x} > b_1 - M_1 y$ cannot be enforced as such, and are always implemented as weak inequalities $\boldsymbol{a}_1^\top \boldsymbol{x} \geq b_1 - M_1 y$, which in most contexts is fine.

Example 14 (Production scheduling - ctd.)**.** *In Example 13 consider an additional constraints that if task A is started before task B, then also task C must be started before task B. Such a situation can arise, for example, then the two groups of tasks A, and B, C require a different machine setup that nobody wants to change too often. This means an if-then constraint*

$$end_A \leq start_B \quad \Rightarrow \quad end_A \leq start_B.$$

which can be formulated via MILP as

$$\begin{aligned} end_A &> start_B - M_1 y \\ end_A &\leq start_B + M_2(1-y). \end{aligned}$$

If we opted for formulating this constraint without MILP tricks, we could have used disjunctive programming as follows:

$$\begin{bmatrix} end_A \leq start_B \\ end_C \leq start_B \end{bmatrix} \quad \veebar \quad [end_A > start_B]$$

Here, we present an extensive example of the if-then logic applied to the problem of recharging an electric vehicle.

Example 15 (Recharging strategy for electric vehicle)**.** *Given the current location x, battery charge c, and planning horizon D, the task is to plan a series of recharging and rest stops for an electric vehicle. The distances to the charging stations are measured relative to an arbitrary location. The objective is to drive from location x to location $x + D$ in as little time as possible subject to the following constraints:*

- *To allow for unforeseen events, the state of charge should never drop below 20% of the maximum capacity.*
- *The the maximum charge is $c_{max} = 80$ kwh.*
- *For comfort, no more than 4 hours should pass between stops, and that a rest stop should last at least t^{rest}.*
- *Any stop includes a $t^{lost} = 10$ minutes of "lost time".*

For this first model we make several simplifying assumptions that can be relaxed as a later time.

- *Travel is at a constant speed $v = 100$ km per hour and a constant discharge rate $R = 0.24$ kwh/km*
- *The batteries recharge at a constant rate determined by the charging station.*
- *Only consider stops at the recharging stations.*

The problem statement identifies four state variables:

- *c the current battery charge*
- *r the elapsed time since the last rest stop*
- *t elapsed time since the start of the trip*
- *x the current location*

The charging stations are located at positions d_i for $i \in I$ with capacity C_i. The arrival time at charging station i is given by

$$\begin{aligned} c_i^{arr} &= c_{i-1}^{dep} - R(d_i - d_{i-1}) \\ r_i^{arr} &= r_{i-1}^{dep} + \frac{d_i - d_{i-1}}{v} \\ t_i^{arr} &= t_{i-1}^{dep} + \frac{d_i - d_{i-1}}{v} \end{aligned}$$

where the script t_{i-1}^{dep} refers to departure from the prior location. At each charging location there is a decision to make of whether to stop, rest, and rechange. If the decision is positive, then

$$\begin{aligned} c_i^{dep} &\leq c^{max} \\ r_i^{dep} &= 0 \\ t_i^{dep} &\geq t_i^{arr} + t_{lost} + \frac{c_i^{dep} - c_i^{arr}}{C_i} \\ t_i^{dep} &\geq t_i^{arr} + t_{rest} \end{aligned}$$

which account for the battery charge, the lost time and time required for battery charging, and allows for a minimum rest time. On the other hand, if a decision is make to skip the charging and rest opportunity,

$$\begin{aligned} c_i^{dep} &= c_i^{arr} \\ r_i^{dep} &= r_i^{arr} \\ t_i^{dep} &= t_i^{arr} \end{aligned}$$

The latter sets of constraints have an exclusive-or relationship. That is, either one or the other of the constraint sets hold, but not both.

$$\min \quad t_{n+1}^{arr}$$

$$\begin{aligned}
\text{s.t.} \quad & r_i^{arr} \leq r^{max} && \forall i \in I \\
& c_i^{arr} \geq c^{min} && \forall i \in I \\
& c_i^{arr} = c_{i-1}^{dep} - R(d_i - d_{i-1}) && \forall i \in I \\
& r_i^{arr} = r_{i-1}^{dep} + \frac{d_i - d_{i-1}}{v} && \forall i \in I \\
& t_i^{arr} = t_{i-1}^{dep} + \frac{d_i - d_{i-1}}{v} && \forall i \in I
\end{aligned}$$

$$\begin{bmatrix} c_i^{dep} & \leq & c^{max} \\ r_i^{dep} & = & 0 \\ t_i^{dep} & \geq & t_i^{arr} + t_{lost} + \frac{c_i^{dep} - c_i^{arr}}{C_i} \\ t_i^{dep} & \geq & t_i^{arr} + t_{rest} \end{bmatrix} \veebar \begin{bmatrix} c_i^{dep} = c_i^{arr} \\ r_i^{dep} = r_i^{arr} \\ t_i^{dep} = t_i^{arr} \end{bmatrix} \qquad \forall i \in I$$

A Pyomo implementation of this model is as follow.

```
# specify number of charging stations
n_charging_stations = 20

# randomly distribute charging stations along a fixed route
np.random.seed(1842)
d = np.round(np.cumsum(np.random.triangular(20, 150, 223, n_charging_stations)), 1)

# randomly assign changing capacities
c = np.random.choice([50, 100, 150, 250], n_charging_stations, p=[0.2, 0.4, 0.3, 0.1])

# assign names to the charging stations
s = [f"S_{i:02d}" for i in range(n_charging_stations)]

stations = pd.DataFrame([s, d, c]).T
stations.columns=["name", "location", "kw"]

# current location (km) and charge (kw)
x = 0

# planning horizon
D = 2000

# charge limits (kw)
c_max = 120
c_min = 0.2 * c_max
c = c_max

# velocity km/hr and discharge rate kwh/km
v = 100.0
R = 0.24

# lost time
t_lost = 10/60
t_rest = 10/60

```

```
# rest time
r_max = 3

def ev_plan(stations, x, D):

    # find stations between x and x + D
    on_route = stations[(stations["location"] >= x) & (stations["location"] <= x + D)]

    m = pyo.ConcreteModel()

    m.n = pyo.Param(default=len(on_route))

    # locations and road segments between location x and x + D
    m.STATIONS = pyo.RangeSet(1, m.n)
    m.LOCATIONS = pyo.RangeSet(0, m.n + 1)
    m.SEGMENTS = pyo.RangeSet(1, m.n + 1)

    # distance traveled
    m.x = pyo.Var(m.LOCATIONS, domain=pyo.NonNegativeReals, bounds=(0, 10000))

    # arrival and departure charge at each charging station
    m.c_arr = pyo.Var(m.LOCATIONS, domain=pyo.NonNegativeReals, bounds=(c_min, c_max))
    m.c_dep = pyo.Var(m.LOCATIONS, domain=pyo.NonNegativeReals, bounds=(c_min, c_max))

    # arrival and departure times from each charging station
    m.t_arr = pyo.Var(m.LOCATIONS, domain=pyo.NonNegativeReals, bounds=(0, 100))
    m.t_dep = pyo.Var(m.LOCATIONS, domain=pyo.NonNegativeReals, bounds=(0, 100))

    # arrival and departure rest from each charging station
    m.r_arr = pyo.Var(m.LOCATIONS, domain=pyo.NonNegativeReals, bounds=(0, r_max))
    m.r_dep = pyo.Var(m.LOCATIONS, domain=pyo.NonNegativeReals, bounds=(0, r_max))

    # initial conditions
    m.x[0].fix(x)
    m.t_dep[0].fix(0.0)
    m.r_dep[0].fix(0.0)
    m.c_dep[0].fix(c)

    @m.Param(m.STATIONS)
    def C(m, i):
        return on_route.loc[i-1, "kw"]

    @m.Param(m.LOCATIONS)
    def location(m, i):
        if i == 0:
            return x
        elif i == m.n + 1:
            return x + D
        else:
            return on_route.loc[i-1, "location"]

    @m.Param(m.SEGMENTS)
    def dist(m, i):
        return m.location[i] - m.location[i-1]

    @m.Objective(sense=pyo.minimize)
    def min_time(m):
        return m.t_arr[m.n + 1]

```

```
95      @m.Constraint(m.SEGMENTS)
96      def drive_time(m, i):
97          return m.t_arr[i] == m.t_dep[i-1] + m.dist[i]/v
98
99      @m.Constraint(m.SEGMENTS)
100     def rest_time(m, i):
101         return m.r_arr[i] == m.r_dep[i-1] + m.dist[i]/v
102
103     @m.Constraint(m.SEGMENTS)
104     def drive_distance(m, i):
105         return m.x[i] == m.x[i-1] + m.dist[i]
106
107     @m.Constraint(m.SEGMENTS)
108     def discharge(m, i):
109         return m.c_arr[i] == m.c_dep[i-1] - R*m.dist[i]
110
111     @m.Disjunction(m.STATIONS, xor=True)
112     def recharge(m, i):
113         # list of constraints thtat apply if there is no stop at station i
114         disjunct_1 = [m.c_dep[i] == m.c_arr[i],
115                       m.t_dep[i] == m.t_arr[i],
116                       m.r_dep[i] == m.r_arr[i]]
117
118         # list of constraints that apply if there is a stop at station i
119         disjunct_2 = [m.t_dep[i] == t_lost + m.t_arr[i] + (m.c_dep[i] - m.c_arr[i])/m.C[i],
120                       m.c_dep[i] >= m.c_arr[i],
121                       m.r_dep[i] == 0]
122
123         # return a list disjuncts
124         return [disjunct_1, disjunct_2]
125
126     pyo.TransformationFactory("gdp.bigm").apply_to(m)
127     pyo.SolverFactory('cbc').solve(m)
128
129     return m
130
131 m = ev_plan(stations, 0, 1000)
132
133 results = pd.DataFrame({
134     i : {"location": m.x[i](),
135          "t_arr": m.t_arr[i](),
136          "t_dep": m.t_dep[i](),
137          "c_arr": m.c_arr[i](),
138          "c_dep": m.c_dep[i](),
139         } for i in m.LOCATIONS
140 }).T
141
142 results["t_stop"] = results["t_dep"] - results["t_arr"]
```

The top plot of Figure 3.3 summarizes the charging stations locations and charging levels for the specific instance implemented above. The optimal charging strategy is reported in Table 3.2 and visualized in the bottom plot of Figure 3.3.

	location	t^{arr}	t^{dep}	c^{arr}	c^{dep}	t^{stop}
	0.0	-	0.00	-	120.00	-
1	191.6	1.92	2.39	74.02	120.00	0.47
2	310.6	3.58	4.03	91.44	120.00	0.45
3	516.0	6.09	6.54	70.70	84.94	0.45
4	683.6	8.21	8.21	44.71	44.71	0.00
5	769.9	9.08	9.72	24.00	47.95	0.65
6	869.7	10.72	11.20	24.00	55.27	0.48
7	1000.0	12.50	-	24.00	-	-

Table 3.2: Optimal EV charging strategy.

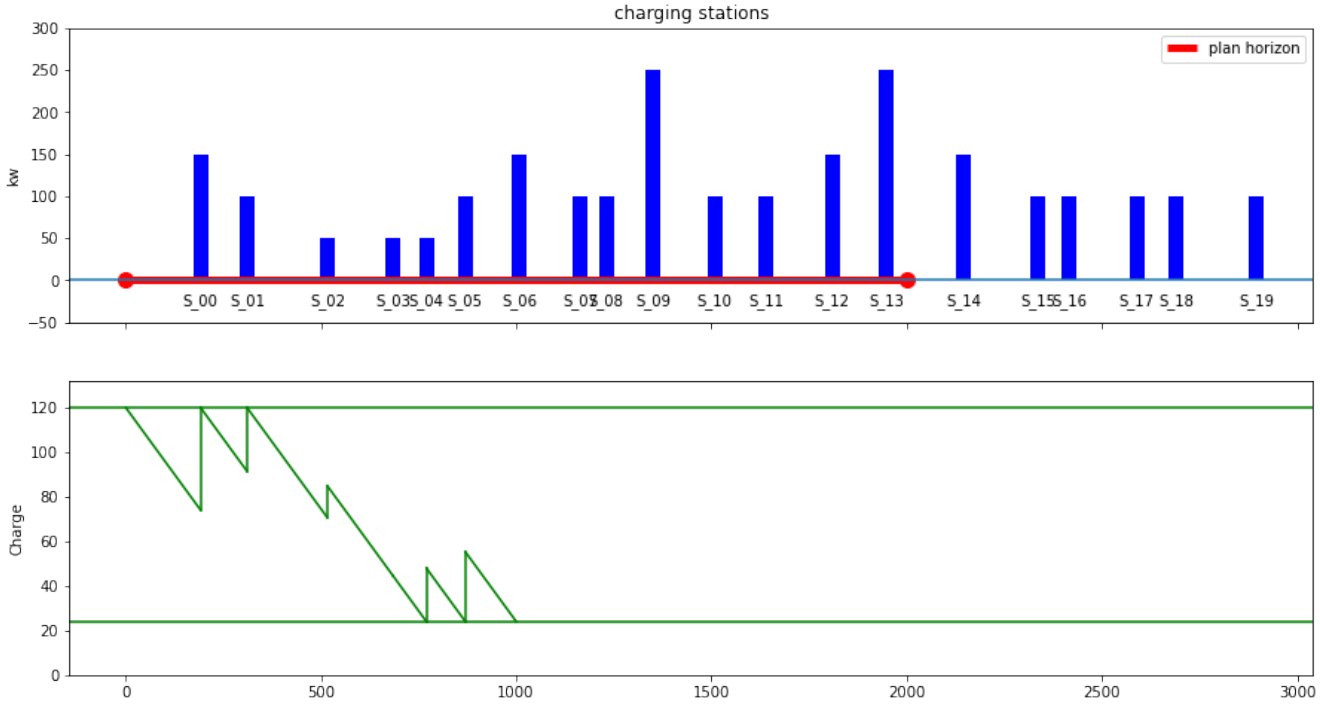


Figure 3.3: Visualization of the optimal EV changing strategy.

3.2.6 Products of variables

If the optimization problem contains the product of two variable, in a few special cases, it still is possible to "linearize" it at the cost of adding a new variable and a set of additional constraints.

The product $x_1 x_2$ of two binary variables $x_1, x_2 \in \mathbb{B}$ can be replaced by a new variable y and the following additional constraints

$$
\begin{aligned}
y &\leq x_1, \\
y &\leq x_2, \\
y &\geq x_1 + x_2 - 1, \\
y &\in \mathbb{B}.
\end{aligned}
$$

Similarly, the product x_1x_2 with $x_1 \in \mathbb{B}$ and $l \leq x_2 \leq u$ can be replaced by a new variable y and the following additional constraints

$$\begin{aligned} y &\leq ux_1, \\ y &\geq lx_1, \\ y &\leq x_2 - l(1 - x_1), \\ y &\geq x_2 - u(1 - x_1), \\ y &\in \mathbb{R}. \end{aligned}$$

3.3 Solution methods

MILP problems are much more difficult to solve than LP. At the same time, the requirements for a solution method remain the same: to find the optimal solution fast, along with a certificate of optimality. How can one go about this? An immediate (brute force) idea is, for example, to inspect each possible value of the integer variables and solve an LP for the remaining, continuous variables (if any).

Example 16. *Coming back to Example 9, if we were to enumerate all the integer solutions in the feasible set, we would end up with* 1150751 (x_1, x_2) *pairs and for each of them compute the corresponding objective function value. For such a small 2-dimensional problem, such a brute force search does quickly returns the optimal solution, but this approach rapidly becomes impractical when the number of decision variables and constraints grow large.*

However, for problems with many dimensions/feasible solutions such an approach would take too much time. We shall now present the two most well-known methods. Both of them use, as their key building block, LP solution algorithms applied to the *LP relaxation* of the given MILP which relaxes the integrality constraints on $\boldsymbol{x}$, i.e., it requires $\boldsymbol{x} \geq 0$ instead.

Since the feasible set of a MILP is a subset of the feasible set of its LP relaxation, it should be clear that in the case of a minimization the optimal value of the linear relaxation is less than or equal than the optimal value of MILP. To exclude such 'good' solutions that violate the integrality constraints, both methods skillfully add extra constraints when needed.

Branch and bound

The most common solution method is **Branch and Bound**. There, first the LP relaxation of the MILP is solved. Clearly, if the optimal solution of the relaxed problem is integer in the variables that were originally required to be integer, an optimal solution to the original problem has been found – this situation, although rare, occurs in special structure problems an example of which we discuss in the next **??**.

If at least one of the originally integer variables is fractional, the objective provides a lower bound of the optimal value of the MILP. At this moment, if we can, we use any heuristic to construct a primal feasible solution to the (unrelaxed) MILP to obtain an upper bound ('some' solution is always be at most as good as an 'optimal' solution). Again, if this solution attains the lower bound, it is an optimal solution to the MILP.

If the two bounds differ, we proceed with the **branching** step which consists of selecting an integer variable x_j that has a fractional value $\bar{x}_j$ in the optimal LP solution and create two new subproblems, one with a constraint $\bar{x}_j \leq \lfloor \bar{x}_j \rfloor$, and the other with $x_j \geq \lceil \bar{x}_j \rceil$.

Clearly, both subproblems have a stricter feasible region than the original one and the optimum to the original MILP must lie in one of the branches. For both subproblems, we can obtain bounds just as in the first step and improve (tighten) the global lower and upper bounds. These steps are repeated until we have found the optimal MILP solution.

The edge of the Branch-and-Bound method over an enumeration of all possible solutions is the use of the lower/upper bounds. These allow to "prune" certain branches from consideration and thus a lot of the possible solutions do not need to be checked. There are three reasons to prune: by infeasibility, by (local) optimality, and by bound.

First, if the LP of a subproblem is infeasible, it should be clearly discarded. Second, if the lower and upper bounds for a subproblem are equal, we have determined an optimal integer solution for that subproblem and we do not further branch this subproblem, since we cannot improve any further (hence the terminology 'local optimal'). Third, the

bounds for a subproblem tell us what is the best possible objective value for all integer solutions in the corresponding feasible region. If this best possible objective value is not better than the best feasible solution found so far anywhere else in the tree, we can discard this subproblem again.

An example of the branch-and-bound process is given in Figures 3.4 and 3.5 for the following LP:

$$\begin{aligned} \max \quad & x_1 + 2x_2 \\ \text{s.t.} \quad & -4x_1 + 5x_2 \leq 11, \\ & 5x_1 - 2x_2 \leq 9, \\ & x_1, x_2 \geq 0, \\ & x_1, x_2 \in \mathbb{N}. \end{aligned}$$

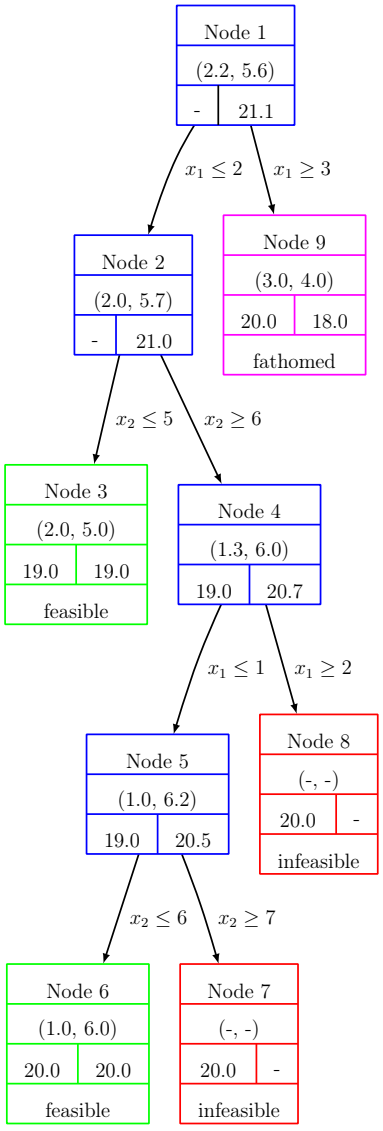


Figure 3.4: A possible Branch and Bound tree in `tikz` for BBa.

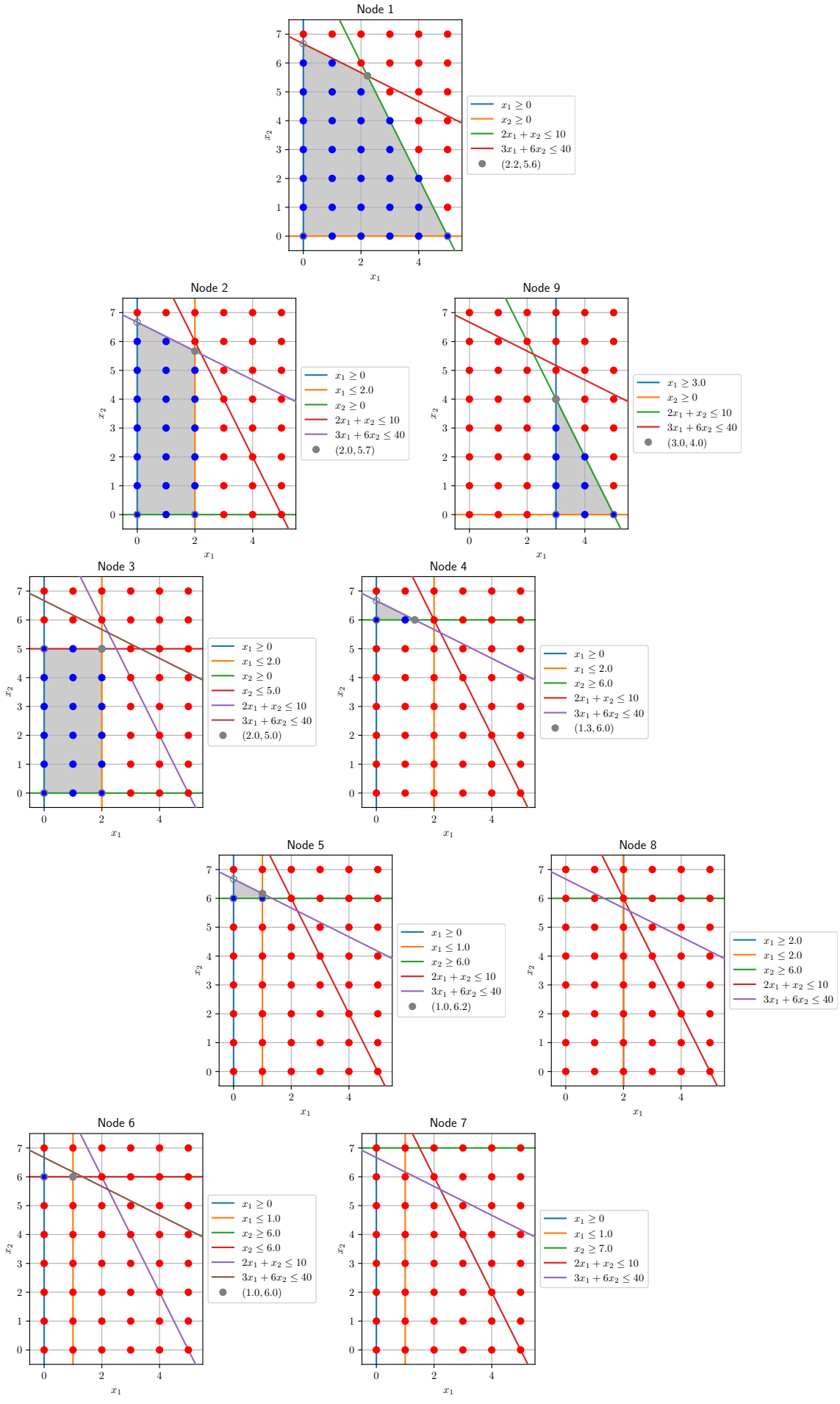


Figure 3.5: Visualizations for the branch-and-bound tree example BBa

The performance of the Branch-and-Bound method can greatly improve by having a tight problem formulation, i.e., one where the gap between the feasible region of the original problem and its LP relaxation is small. Such a tight formulation typically leads to better bounds, therefore more pruning of subproblems, speeding up computation. The extreme case is to use the convex hull for the formulation. The convex hull of a set S is the smallest convex set containing S or, equivalently, the set of all convex combinations of its points, see Figure 3.6.

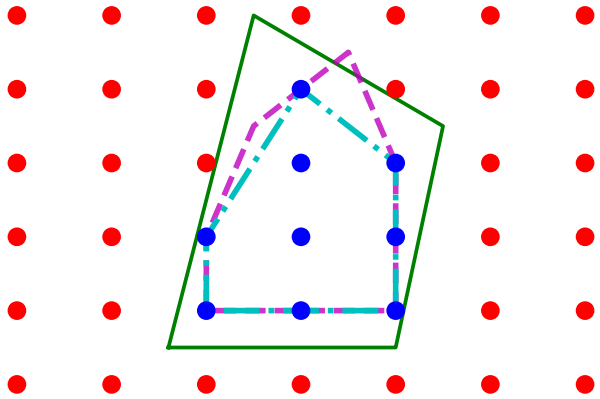


Figure 3.6: Three possible feasible regions for the LP relaxation of an MILP with 9 feasible points (in blue). The blue region is the convex hull of these points. If the problem is formulated so that the feasible region is the blue one, the LP relaxation would be exact and find the optimal solution of the original MILP.

Finding a tight formulation for a specific problem is an art that requires experience with mathematical modelling. Some 'rules of thumb' here are to (i) have as few as possible big-M constraints with the M being not too large, (ii) avoid having superfluous decision variables, (iii) avoid formulating as inequality a relation that will be known to hold with equality at the optimal solution.

In theory, each MILP has an equivalent LP formulation, for example, the convex hull of the integer feasible region. Solving the LP relaxation with the convex hull formulation would immediately lead to the optimal integer solution. However, this formulation is either difficult to find or requires excessively many variables and/or constraints. Nevertheless, if we have a formulation of the convex hull we can simply use a linear programming method to solve our MILP.

Cutting planes

Another solution method is to address directly the gap between the original problem and its relaxation. An inequality $\boldsymbol{a}^\top \boldsymbol{x} \leq \boldsymbol{b}$ is a *valid inequality* for an MILP if $\boldsymbol{a}^\top \boldsymbol{x} \leq b$ holds for all $\boldsymbol{x}$ in its (integer) feasible region. In other words, valid inequalities cut off non-integer points from the LP relaxation's feasible region.

To use that method however, one needs to know how to come up with such inequalities efficiently. A systematic way to generate valid inequalities are Chvátal-Gomory cuts. Consider the constraints $A\boldsymbol{x} \leq \boldsymbol{b}$ with $\boldsymbol{x} \in \mathbb{N}^n$, where $A \in \mathbb{R}^{m \times n}$. The expression $\boldsymbol{\lambda}^\top A\boldsymbol{x} \leq \boldsymbol{\lambda}^\top b$ with $\boldsymbol{\lambda} \in \mathbb{R}^m_{\geq 0}$ is a valid inequality. Since x is integer and non-negative, $\lfloor \boldsymbol{\lambda}^\top A \rfloor \boldsymbol{x} \leq \boldsymbol{\lambda}^\top \boldsymbol{b}$ is valid as well. Now realise that the right-hand side can be rounded as well: $\lfloor \boldsymbol{\lambda}^\top A \rfloor \boldsymbol{x} \leq \lfloor \boldsymbol{\lambda}^\top \boldsymbol{b} \rfloor$, which is a Chvátal-Gomory cut.

In certain cases, iteratively determining useful valid inequalities can be done efficiently, which opens the way to a cutting plane method to solve such problems. This method iteratively solves the LP relaxation and adds new suitable valid inequalities whenever a fractional solution is found. However, a 'pure' cutting plane method typically only works for very specific problems and will not converge fast enough in general.

Performance of MILP algorithms

Most efficient approaches to general MILPs combine Branch and Bound with valid inequalities, i.e., they simultaneously analyze the Branch and Bound tree, and add valid inequalities to the resulting subproblems. The performance of a given algorithm will typically depend greatly on the model formulation. Finding a problem formulation that will work well is, for this writing, an art with some science-driven intuition that requires experience (also with the specific

solver, as ultimately, the solution time depends on the inner workings of it). For example, it might be intuitive to think that a good formulation should involve few decision variables. However, in the following example, we show how the intuition that a good problem formulation is one with the least number of constraints, can sometimes fail.

Example 17 (Facility location – Less constraints is not always good for MILP)**.** *Consider the problem of having to meet a certain customer demand and deciding how many facilities to build and in which locations while trying to minimize costs. Let c_j be the cost of building facility j and h_{ij} the cost to meet all the demands of customer i at facility j. Introduce two sets of binary variables,*

$$x_j := \begin{cases} 1 & \text{if facility } j \text{ is built,} \\ 0 & \text{otherwise,} \end{cases} \qquad \text{and} \qquad y_{ij} := \begin{cases} 1 & \text{if customer } i \text{ is served at facility } j, \\ 0 & \text{otherwise.} \end{cases}$$

The resulting MILP is

$$\begin{aligned} \min \quad & \sum_j c_j x_j + \sum_{i,j} h_{ij} y_{ij} \\ \text{s.t.} \quad & \sum_j y_{ij} = 1, \qquad \forall\, i \qquad \textit{(every customer is served)} \\ & y_{ij} \le x_j, \qquad \forall\, i,j \qquad \textit{(facility built before use)} \\ & x_j, y_{ij} \in \mathbb{B}, \qquad \forall\, i,j. \end{aligned}$$

Note that, since $\sum_j y_{ij} = 1$ for every i, we can replace the $n \times m$ constraints $y_{ij} \le x_j$ by only n constraints, namely

$$\sum_i y_{ij} \le n x_j.$$

In this way, the model becomes more compact in its mathematical formulation, but it is not a good idea if we want to solve this problem using its LP relaxation, since by reducing the number of constraints we made its feasible region of the relaxation larger (hence less tight around the feasible integer points). This is clearly reflected in the longer run-time needed to solve the optimization problem with weaker constraints, see Figure 3.7 below.

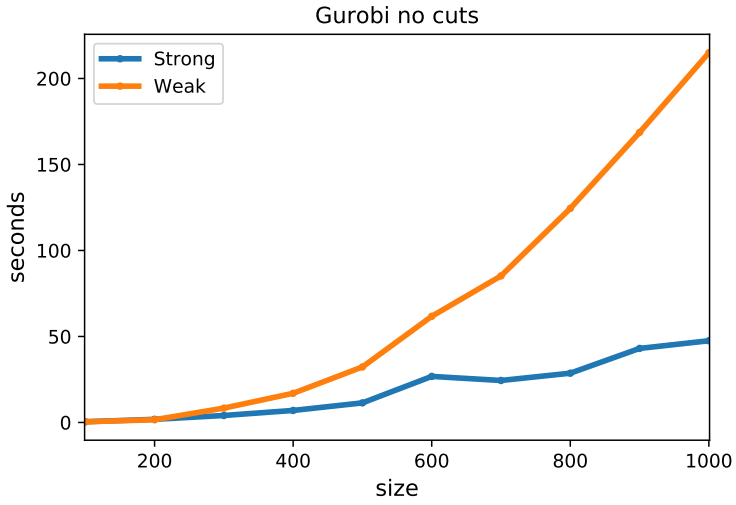


Figure 3.7: Comparison of the run-time of the commercial solver Gurobi to solve the same optimization problem, but with two different constraint formulations, one stronger than the other.

3.4 A complete example: BIM production revisited

Consider Caroline's product planning again, but now with more sophisticated pricing and acquisition protocols. There are now three suppliers who can deliver the following products:

- A: silicon, germanium and plastic
- B: copper
- C: all of the above

For the suppliers, the following conditions apply. Copper should be acquired in multiples of 100 gram, since it is delivered in sheets of 100 gram.

Unitary products such as silicon, germanium and plastic may be acquired in any real non-negative number, but the price is charged for each 'started' batch of 100 units, regardless if the full batch was used or not. That means, 30 units of silicon with 10 units of germanium and 50 units of plastic cost as much as 1 unit of silicon but half as much as 30 units of silicon with 30 units of germanium and 50 units of plastic.

Furthermore, supplier C sells all products and offers a discount if two products are purchased together: 100 gram of copper and a batch of 100 units of a unitary product cost together just 7. This set price is only applied to pairs, meaning that 100 gram of copper and 2 batches cost 13.

The summary of the prices in € is given in the following table:

Supplier	Copper per sheet of 100 gram	Batch of units	Together
A	-	5	-
B	3	-	-
C	4	6	7

Next, for stocked products inventory costs are incurred, whose summary is given by the following table:

copper per 10gr	silicon per unit	germanium per unit	plastic per unit
0.1	0.02	0.02	0.02

The holding price of copper is per 10 gram and the copper stocked is rounded up to multiples of 10 grams, meaning that 12 grams pay for 20.

The capacity limitations of the warehouse allow for a maximum of 10 kilogram of copper in stock at any moment, but there are no practical limitations to the number of units of unitary products in stock.

As you recall, BIM has the following stock at the beginning of the year:

copper	silicon	germanium	plastic
480	1000	1500	1750

The company would like to have at least the following stock at the end of the year:

copper	silicon	germanium	plastic
200	500	500	1000

Caroline's goal is to model the corresponding optimization problem, aiming to minimize the acquisition and holding costs of the products while meeting the required quantities for production. The production is made-to-order, meaning that no inventory of chips is kept.

As usual, we need to start modelling by naming our decisions and identifying the relevant index sets. In Section 2.5 we already had indices for products and time periods. These are denoted there P for the products and T for the time periods (the months). We already have a set of products, P, but now we will need to differentiate the unitary products.

Next, we need to keep track of how much we buy of what and from what supplier, which means that we need to index the suppliers. Then there is the nature of the variables. In the complete example in Chapter 2 all variables were continuous, but now Caroline needs to take care of a number of integers:

- number of batches of the unitary products;
- number of copper sheets;

- number of 'volume units' in stock, since inventory of copper pays per 10 gram;
- number of copper sheet-batch of unitary products pairs acquired in the same period from supplier C, yielding discount.

Given the above, we can clearly reuse the variables s_{pt} from the complete example of Chapter 2 to account for the total number of grams/units of each product. What regards the purchasing variables, however, we now need to differentiate between the unitary products and the suppliers. For that reason, we will use integer variables y_{ts} for the number of copper sheets purchased from supplier s, and continuous variables x_{pts} for the number of units of unitary product p purchased from supplier s. To translate these variables into 'amounts of a product purchased regardless of the supplier', we introduce another variable u_{pt}.

We now introduce the variables that we shall use to account for the 'tricky', integer numbers in our model. To account for the number of batches of unitary products at supplier s, we will introduce integer variables b_{ts}, and similarly we will introduce an integer variable r_t for the number of 10g buckets of copper in the inventory. Because we can see the buying of a batch-sheet of copper pair at supplier C as effectively, obtaining a discount of $\beta = 4+6-7$. For that reason, we will introduce an extra integer variable p_t to count the number of such pairs, so that $p_t\beta$ will amount to the total 'discount' acquired in time period t. We will need to use constraints somehow to make sure that these three variables 'take the values they should' in relation to the purchasing and inventory variables.

Taking all the above into account, we shall use the following decision variables

- y_{ts}: the number of sheets of copper bought in period t from supplier s
- x_{pts}: the number of units of unitary product p in period t from supplier s
- u_{pt}: total number of units of product p purchased in period t (from all suppliers combined)
- b_{ts}: number of batches of 100 for the unitary products in period t from supplier s
- p_t: number of batch-sheet of copper pairs purchased at supplier C in time period t
- r_t: number of 10-unit amounts of copper (result of the rounding up).

We begin by formulating the objective function. If we denote by α_s the per-batch cost of unitary products at supplier s, then the total material acquisition cost is

$$\sum_t \left(\left(\sum_s \pi_s b_{ts} + \alpha_s y_{ts} \right) - \beta p_t \right),$$

where we count the acquisition costs for supplier C as the total cost of the sheets of copper plus the batches of unitary materials minus the discount due to purchasing the pairs:

$$\pi_{\mathrm{C}} b_{t,\mathrm{C}} + \alpha_{\mathrm{C}} y_{t,\mathrm{C}} - \beta p_t.$$

The total inventory cost is

$$\sum_t \left(0.1 r_t + \sum_{p \neq \text{copper}} h_p s_{pt} \right).$$

Now, we need to construct a system of constraints to make the above costs be computed correctly.

To transform the amounts of unitary materials purchased into batches of 100, we construct the following constraints:

$$\sum_{p \neq \text{copper}} x_{pts} \leq 100 b_{ts}.$$

Because the inventory costs for copper are computed by rounding up to 10g amounts, we account for this via the following constraint

$$s_{\text{copper},t} \leq 10 r_t$$

In the end, we can account for the number of sheets of copper-unitary material batch pairs purchased at supplier C, we can use the following constraints:

$$p_t \leq b_{t,\mathrm{C}}, \quad p_t \leq y_{t,\mathrm{C}}$$

Choosing a maximal possible p_t satisfying such constraints will naturally lead to maximizing the number of pairs and thus, the total discount.

The remaining constraints such as the inventory balance constraints, total inventory capacity etc. are straightforward, which gives us the following complete model formulation:

$$
\begin{aligned}
\min \quad & \sum_t \left(\left(\sum_s \pi_s b_{ts} + \alpha_s y_{ts} \right) - \beta p_t \right) + \sum_t \left[\gamma r_t + \sum_{p \neq \text{copper}} s_t \right] \\
\text{s.t.} \quad & \sum_{p \neq \text{copper}} x_{pts} \leq 100 b_{ts} && \forall t, s \\
& s_{\text{copper},t} \leq 0.1 r_t && \forall t \\
& s_{\text{copper},t} \leq 10000 && \forall t \\
& u_{\text{copper},t} = 100 \sum_s y_{ts} && \forall t \\
& p_t \leq b_{t,\text{C}}, \quad p_t \leq y_{t,\text{C}} && \forall t \\
& u_{pt} = \sum_s x_{pts} && \forall t, p \neq \text{copper} \\
& x_{pt} + u_{p,t-1} = \delta_{pt} + s_{pt} && \forall t, p \\
& s_{p\text{Dec}} \geq \Omega_p && \forall p \in P \\
& b_{ts}, r_t \in \mathbb{Z}_+.
\end{aligned}
$$

Note that because r_t, b_{ts} appear with a nonnegative coefficient and p_t with a positive coefficient in the objective functions, at the optimal solution these variables will automatically take their minimal (r_t, b_{ts}) and maximal (p_t) values allowed by the corresponding constraints.

```
def VersionOne( demand, existing, desired, stock_limit,
                supplying_copper, supplying_batches,
                price_copper_sheet, price_batch, discounted_price,
                batch_size, copper_sheet_mass, copper_bucket_size,
                unitary_products, unitary_holding_costs
                ):

    m = pyo.ConcreteModel( 'Product acquisition and inventory with sophisticated prices' )

    periods  = demand.columns
    products = demand.index
    first    = periods[0]
    prev     = { j : i for i,j in zip(periods,periods[1:]) }
    last     = periods[-1]

    @m.Param( products, periods )
    def delta(m,p,t):
        return demand.loc[p,t]

    @m.Param( supplying_batches )
    def pi(m,s):
        return price_batch[s]

    @m.Param( supplying_copper )
    def kappa(m,s):
        return price_copper_sheet[s]

    @m.Param()
    def beta(m):
        return price_batch['C']+price_copper_sheet['C']-discounted_price
```

```
31
32      @m.Param(products)
33      def gamma(m,p):
34          return unitary_holding_costs[p]
35
36      @m.Param( products )
37      def Alpha(m,p):
38          return existing[p]
39
40      @m.Param( products )
41      def Omega(m,p):
42          return desired[p]
43
44      m.y = pyo.Var( periods, supplying_copper, within=pyo.NonNegativeIntegers )
45      m.x = pyo.Var( unitary_products, periods, supplying_batches, within=pyo.NonNegativeReals )
46      m.s = pyo.Var( products, periods, within=pyo.NonNegativeReals )
47      m.u = pyo.Var( products, periods, within=pyo.NonNegativeReals )
48      m.b = pyo.Var( periods, supplying_batches, within=pyo.NonNegativeIntegers )
49      m.p = pyo.Var( periods, within=pyo.NonNegativeIntegers )
50      m.r = pyo.Var( periods, within=pyo.NonNegativeIntegers )
51
52      @m.Constraint( periods, supplying_batches )
53      def units_in_batches( m, t, s ):
54          return pyo.quicksum( m.x[p,t,s] for p in unitary_products ) <= batch_size*m.b[t,s]
55
56      @m.Constraint( periods )
57      def copper_in_buckets( m, t ):
58          return m.s['copper',t] <= copper_bucket_size*m.r[t]
59
60      @m.Constraint( periods )
61      def inventory_capacity( m, t ):
62          return m.s['copper',t] <= stock_limit
63
64      @m.Constraint( periods )
65      def pairs_in_batches( m, t ):
66          return m.p[t] <= m.b[t,'C']
67
68      @m.Constraint( periods )
69      def pairs_in_sheets( m, t ):
70          return m.p[t] <= m.y[t,'C']
71
72      @m.Constraint( periods, products )
73      def bought( m, t, p ):
74          if p == 'copper':
75              return m.u[p,t] == copper_sheet_mass*pyo.quicksum( m.y[t,s] for s in supplying_copper )
76          else:
77              return m.u[p,t] == pyo.quicksum( m.x[p,t,s] for s in supplying_batches )
78
79      @m.Expression()
80      def acquisition_cost( m ):
81          return pyo.quicksum(
82                      pyo.quicksum( m.pi[s]*m.b[t,s] for s in supplying_batches ) \
83                    + pyo.quicksum( m.kappa[s]*m.y[t,s] for s in supplying_copper ) \
84                    - m.beta * m.p[t] for t in periods )
85
86      @m.Expression()
87      def inventory_cost( m ):
88          return pyo.quicksum( m.gamma['copper']*m.r[t] + \
89                               pyo.quicksum( m.gamma[p]*m.s[p,t] for p in unitary_products ) \
```

```
                          for t in periods )

    @m.Objective( sense=pyo.minimize )
    def total_cost( m ):
        return m.acquisition_cost + m.inventory_cost

    @m.Constraint( products, periods )
    def balance( m, p, t ):
        if t == first:
            return m.Alpha[p] + m.u[p,t] == m.delta[p,t] + m.s[p,t]
        else:
            return m.u[p,t] + m.s[p,prev[t]] == m.delta[p,t] + m.s[p,t]

    @m.Constraint( products )
    def finish( m, p ):
        return m.s[p,last] >= m.Omega[p]

    return m
```

This example has a very special structure, which repeats for each $t \in T$. At each month, e.g. $t \in T$, two types of decisions are made:

- acquisition,
- inventory.

Modelling each of these types of decisions requires its own variables and constraints. The main model can be seen as being composed of 12 pairs of smaller interconnected models. Pyomo includes the concept of 'block' which is ideal to model such types of structures. Below, we present a second version of this model, where we use a Pyomo block construction to have the model decomposed into two submodels, [A]cquisition and [I]nventory, which are related by the constraints that model the flow of products acquired, used to satisfy demand and kept in stock, as we have seen at the end of of Chapter 2.

```
def VersionTwo( demand, existing, desired,
                stock_limit,
                supplying_copper, supplying_batches, price_copper_sheet, price_batch, discounted_price,
                batch_size, copper_sheet_mass, copper_bucket_size,
                unitary_products, unitary_holding_costs
                ):
    m = pyo.ConcreteModel( 'Product acquisition and inventory with sophisticated prices in blocks' )

    periods  = demand.columns
    products = demand.index
    first    = periods[0]
    prev     = { j : i for i,j in zip(periods,periods[1:]) }
    last     = periods[-1]

    m.T = pyo.Set( initialize=periods )
    m.P = pyo.Set( initialize=products )

    m.PT = m.P * m.T # to avoid internal set bloat

    @m.Block( m.T )
    def A( b ):
        b.x = pyo.Var( supplying_batches, products, within=pyo.NonNegativeReals )
        b.b = pyo.Var( supplying_batches, within=pyo.NonNegativeIntegers )
        b.y = pyo.Var( supplying_copper, within=pyo.NonNegativeIntegers )
        b.p = pyo.Var( within=pyo.NonNegativeIntegers )

        @b.Constraint( supplying_batches )
```

```
28          def in_batches( b, s ):
29              return pyo.quicksum( b.x[s,p] for p in products ) <= batch_size*b.b[s]
30
31          @b.Constraint()
32          def pairs_in_batches( b ):
33              return b.p <= b.b['C']
34
35          @b.Constraint()
36          def pairs_in_sheets( b ):
37              return b.p <= b.y['C']
38
39          @b.Expression( products )
40          def u( b, p ):
41              if p == 'copper':
42                  return copper_sheet_mass*pyo.quicksum( b.y[s] for s in supplying_copper )
43              return pyo.quicksum( b.x[s,p] for s in supplying_batches )
44
45          @b.Expression()
46          def cost( b ):
47              discount = price_batch['C']+price_copper_sheet['C']-discounted_price
48              return pyo.quicksum( price_copper_sheet[s]*b.y[s] for s in supplying_copper ) \
49                  + pyo.quicksum( price_batch[s]*b.b[s] for s in supplying_batches ) \
50                  - discount * b.p
51
52      @m.Block( m.T )
53      def I( b ):
54          b.s = pyo.Var( products, within=pyo.NonNegativeReals )
55          b.r = pyo.Var( within=pyo.NonNegativeIntegers )
56
57          @b.Constraint()
58          def copper_in_buckets(b):
59              return b.s['copper'] <= copper_bucket_size*b.r
60
61          @b.Constraint()
62          def capacity( b ):
63              return b.s['copper'] <= stock_limit
64
65          @b.Expression()
66          def cost( b ):
67              return unitary_holding_costs['copper']*b.r + \
68                  pyo.quicksum( unitary_holding_costs[p]*b.s[p] for p in unitary_products )
69
70      @m.Param( m.PT )
71      def delta(m,t,p):
72          return demand.loc[t,p]
73
74      @m.Expression()
75      def acquisition_cost( m ):
76          return pyo.quicksum( m.A[t].cost for t in m.T )
77
78      @m.Expression()
79      def inventory_cost( m ):
80          return pyo.quicksum( m.I[t].cost for t in m.T )
81
82      @m.Objective( sense=pyo.minimize )
83      def total_cost( m ):
84          return m.acquisition_cost + m.inventory_cost
85
86      @m.Constraint( m.PT )
```

```
87      def balance( m, p, t ):
88          if t == first:
89              return existing[p] + m.A[t].u[p] == m.delta[p,t] + m.I[t].s[p]
90          else:
91              return m.A[t].u[p] + m.I[prev[t]].s[p] == m.delta[p,t] + m.I[t].s[p]
92
93      @m.Constraint( m.P )
94      def finish( m, p ):
95          return m.I[last].s[p] >= desired[p]
96
97      return m
```

3.4.1 Optimal solution

The two versions provided above are equivalent implementations of the mathematical model described.

Each of them can be used as illustrated below for the first, by calling the appropriate function, inserting the parameter values.

```
1  m = VersionOne( demand = demand,
2      existing = {'silicon' : 1000, 'germanium': 1500, 'plastic': 1750, 'copper' : 4800 },
3      desired = {'silicon' :  500, 'germanium':  500, 'plastic': 1000, 'copper' : 2000 },
4      stock_limit = 10000,
5      supplying_copper = [ 'B', 'C' ],
6      supplying_batches = [ 'A', 'C' ],
7      price_copper_sheet = { 'B': 300, 'C': 400 },
8      price_batch = { 'A': 500, 'C': 600 },
9      discounted_price = 700,
10     batch_size = 100,
11     copper_sheet_mass = 100,
12     copper_bucket_size = 10,
13     unitary_products = [ 'silicon', 'germanium', 'plastic' ],
14     unitary_holding_costs = { 'copper': 10, 'silicon' : 2, 'germanium': 2, 'plastic': 2 }
15     )
16
17 pyo.SolverFactory( 'cbc' ).solve(m)
```

The optimal solution, with a cost of 110216 cents, since we solved with the prices and costs in cents, as seen above. This is of course € 1102.16. The same solution is found by all versions presented here.

The solution consists of acquisitions of unitary products as in the next table.

supplier	materials	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
A	silicon	0	0	0	0	0	0	0	0	0	0	0	0
	germanium	0	0	0	0	0	0	0	0	0	0	0	0
	plastic	0	0	0	0	0	0	0	0	0	0	0	0
C	silicon	0	0	0	0	0	214	249	237	265	349	257	690
	germanium	0	0	0	0	0	0	0	0	0	0	0	0
	plastic	0	0	0	0	0	15	251	363	335	351	343	1310

Table 3.3: Acquisition plan for unitary materials

The next table describes the acquisition of copper sheets.

The acquisition plans of units and sheets lead to the following batches and these to the pairs of acquisitions to supplier C, yielding discount, as described below.

The last table concludes the description of the solution with the products kept in stock at each month.

supplier	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
B	0	0	0	0	0	7	7	5	7	6	6	11
C	0	0	0	0	0	3	5	6	6	7	6	20

Table 3.4: Acquisition plan for copper sheets materials

supplier	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
A	0	0	0	0	0	0	0	0	0	0	0	0
C	0	0	0	0	0	3	5	6	6	7	6	20

Table 3.5: The batches acquired

Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
0	0	0	0	0	3	5	6	6	7	6	20

Table 3.6: The pairs acquired from supplier C

product	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
silicon	912	787	527	310	72	0	1	0	0	56	54	500
plastic	1615	1428	1087	805	472	83	0	36	24	0	0	1000
copper	4354	3730	2528	1530	388	8	44	14	90	54	50	2042
germanium	1453	1391	1310	1245	1150	1032	946	857	775	693	609	543

Table 3.7: The stock levels

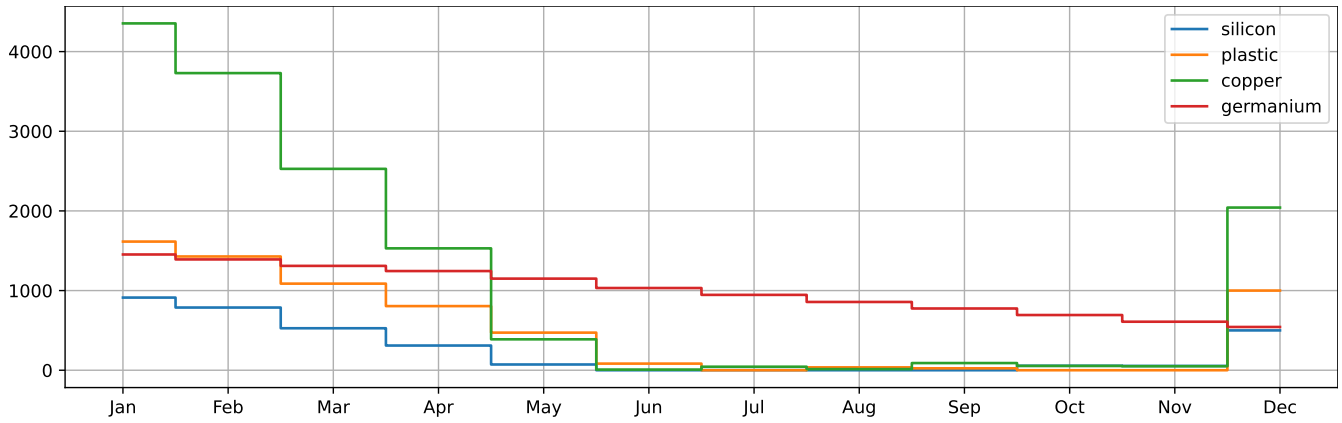


Figure 3.8: The optimal stock levels over the entire year

3.4.2 Impact on solving time

Mathematical optimization models are sometimes surprising in terms of optimization speed. We have seen in Figure 3.7 the difference that it makes to have either of the two equivalent models for the uncapacitated location problem. It is worth mentioning that on a Intel(R) Core(TM) i7-1185G7 at 3.00GHz solving the instance produced by `VersionOne` with `cbc` takes 40 seconds, while the instance produced by `VersionTwo` takes half of that time. One explanation for the difference may be that the second version has less variables, as `u` are expressions, however a third version as below solves in about the same time as `VersionTwo` while declaring the variables that were 'saved' by `VersionTwo`.

```
def VersionThree( demand, existing, desired,
                  stock_limit,
                  supplying_copper, supplying_batches, price_copper_sheet, price_batch, discounted_price,
                  batch_size, copper_sheet_mass, copper_bucket_size,
                  unitary_products, unitary_holding_costs
                  ):
    m = pyo.ConcreteModel( 'Product management with sophisticated prices, blocks and unnecessary variables' )
```

```
8
9     periods  = demand.columns
10    products = demand.index
11    first    = periods[0]
12    prev     = { j : i for i,j in zip(periods,periods[1:]) }
13    last     = periods[-1]
14
15    m.T = pyo.Set( initialize=periods )
16    m.P = pyo.Set( initialize=products )
17
18    m.PT = m.P * m.T # to avoid internal set bloat
19
20    m.x = pyo.Var( m.PT, within=pyo.NonNegativeReals )
21
22    @m.Block( m.T )
23    def A( b ):
24        b.x = pyo.Var( supplying_batches, products, within=pyo.NonNegativeReals )
25        b.b = pyo.Var( supplying_batches, within=pyo.NonNegativeIntegers )
26        b.y = pyo.Var( supplying_copper, within=pyo.NonNegativeIntegers )
27        b.p = pyo.Var( within=pyo.NonNegativeIntegers )
28
29        @b.Constraint( supplying_batches )
30        def in_batches( b, s ):
31            return pyo.quicksum( b.x[s,p] for p in products ) <= batch_size*b.b[s]
32
33        @b.Constraint()
34        def pairs_in_batches( b ):
35            return b.p <= b.b['C']
36
37        @b.Constraint()
38        def pairs_in_sheets( b ):
39            return b.p <= b.y['C']
40
41        @b.Expression( products )
42        def u( b, p ):
43            if p == 'copper':
44                return copper_sheet_mass*pyo.quicksum( b.y[s] for s in supplying_copper )
45            return pyo.quicksum( b.x[s,p] for s in supplying_batches )
46
47        @b.Expression()
48        def cost( b ):
49            discount = price_batch['C']+price_copper_sheet['C']-discounted_price
50            return pyo.quicksum( price_copper_sheet[s]*b.y[s] for s in supplying_copper ) \
51                + pyo.quicksum( price_batch[s]*b.b[s] for s in supplying_batches ) \
52                - discount * b.p
53
54    @m.Block( m.T )
55    def I( b ):
56        b.s = pyo.Var( products, within=pyo.NonNegativeReals )
57        b.r = pyo.Var( within=pyo.NonNegativeIntegers )
58
59        @b.Constraint()
60        def copper_in_buckets(b):
61            return b.s['copper'] <= copper_bucket_size*b.r
62
63        @b.Constraint()
64        def capacity( b ):
65            return b.s['copper'] <= stock_limit
66
```

```
        @b.Expression()
        def cost( b ):
            return unitary_holding_costs['copper']*b.r + \
                pyo.quicksum( unitary_holding_costs[p]*b.s[p] for p in unitary_products )

    @m.Param( m.PT )
    def delta(m,t,p):
        return demand.loc[t,p]

    @m.Expression()
    def acquisition_cost( m ):
        return pyo.quicksum( m.A[t].cost for t in m.T )

    @m.Expression()
    def inventory_cost( m ):
        return pyo.quicksum( m.I[t].cost for t in m.T )

    @m.Objective( sense=pyo.minimize )
    def total_cost( m ):
        return m.acquisition_cost + m.inventory_cost

    @m.Constraint( m.PT )
    def match( m, p, t ):
        return m.x[p,t] == m.A[t].u[p]

    @m.Constraint( m.PT )
    def balance( m, p, t ):
        if t == first:
            return existing[p] + m.x[p,t] == m.delta[p,t] + m.I[t].s[p]
        else:
            return m.x[p,t] + m.I[prev[t]].s[p] == m.delta[p,t] + m.I[t].s[p]

    @m.Constraint( m.P )
    def finish( m, p ):
        return m.I[last].s[p] >= desired[p]

    return m
```

It is therefore worth experimenting with different ways to express the same model, in as much the same way different implementations of the same algorithm also differ in execution time!

Final note: Gurobi, a leading commercial solver, solves all versions listed above in just a few hundredth of a second...

Exercises

*Exercises are labeled as **C** (calculating), **M** (modeling), **T** (theoretical) or combination of these, reflecting on which of these three aspects they focus. The most difficult/lengthy exercises are labeled with a ⋆.*

Ex. 3.1 (M+T ⋆) Assume there are three constraints of the type $\sum_j^J a_{ij}x_j \leq b_i$, $i = 1, 2, 3$, where a_j is a generic real number and x_j is a decision variable for each $j = 1, \ldots, J$. We want to construct an objective function that reflects the way the constraints are satisfied. More specifically:

- For each satisfied constraint, the objective function increases by 1;
- If at least one constraint is satisfied, then the objective function increases by 1;
- If constraints 1 and 2 are satisfied simultaneously, the objective function gets an extra 1.

Examples:

- *Satisfying all constraints yields* $3 \times 1 + 1 + 1 = 5$.

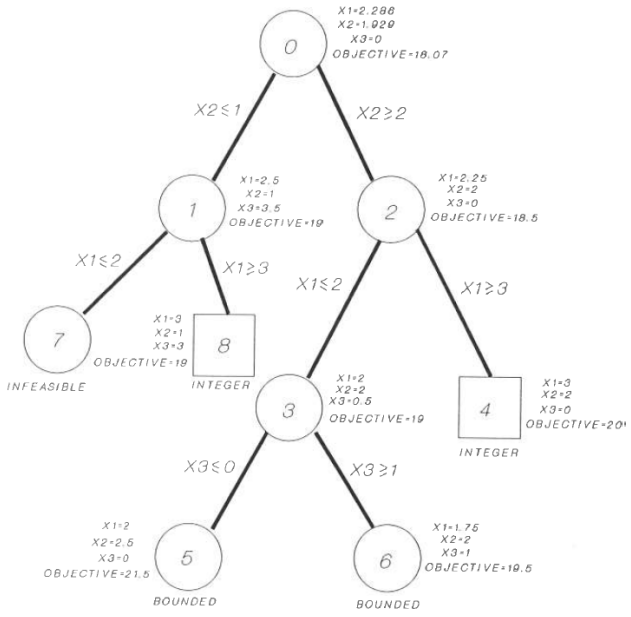
- *Satisfying only constraint 3 yields* $1 + 1 = 2$.
- *Satisfying constraints 1 and 2 yields* $2 \times 1 + 1 + 1 = 4$.
- *Satisfying no constraint yields* 0.

Formulate a mixed ILP for which the objective function is maximized.

Ex. 3.2 (C) Consider the following ILP:

$$\begin{aligned} \min \quad & 2x_1 + 7x_2 + 2x_3 \\ \text{s.t.} \quad & x_1 + 4x_2 + x_3 \geq 10, \\ & 4x_1 + 2x_2 + 2x_3 \geq 13, \\ & x_1 + x_2 - x_1 \geq 10, \\ & x_1, x_2, x_3 \geq 0 \text{ and integer.} \end{aligned}$$

Which branching sequence in the tree below results in the fewest number of subproblems to be solved?



Ex. 3.3 (M, ⋆) A company produces and sells m different products. For each product $j = 1, \ldots, m$ the forecasted demand is equal to d_j. To produce each product, the company uses n types of ingredients. In particular, the amount of ingredient i that must be used to produce product j is denoted with a_{ij}. The company purchases ingredients from a supplier that applies the following pricing policy: buying one unit of ingredient i costs c_i, but, if more than q_i units of ingredient i are bought, the company gets volume discount and the cost per unit of ingredient i goes down to c'_i for the entire order. The supply chain department must decide on the amount of ingredients to purchase in order to satisfy the demand of each product while minimizing the purchasing costs. Formulate an ILP that can be used to help the supply-chain department.

Note: Depending on the values of the parameters, it could be cheaper buying more ingredients than necessary. This happens when the cost of ordering $q_i - 1$ *units of ingredient* i*, i.e.,* $(q_i - 1)c_i$*, is higher than* $q_i c'_i$*.*

Ex. 3.4 (T+C) The knapsack problem is a well-studied problem in Operations Research. In its general form, there are n items and each item $i = 1, \ldots, n$ yields a profit p_i and has weight w_i. The goal of the problem is to decide which items to put in your knapsack to maximize your total profit under the constraint that the total weight of the packed items does not exceed C, the capacity of your knapsack . Assume the total capacity is $C = 11$ and consider the items as listed in the following table:

In the *fractional knapsack problem*, it is assumed that fractions of items can be chosen (the items are divisible). For such a problem the following simple greedy algorithm produces an optimal solution. At each step we select the item

i	1	2	3	4	5	6	7
p_i	60	60	40	15	20	10	3
w_i	3	5	4	2	3	3	1

with maximum value of p_i/w_i and we add as much as possible of that item to our knapsack (taking into account the capacity constraint). We update the used capacity and we repeat this process with the next item until the knapsack is full or until all the items have been processed.

(a) Prove (in full generality) that the greedy solution indeed produces an optimal solution.

(b) In practice, it is often not possible to split an item. The situation in which items are indivisible is harder to solve than the divisible case and give rise to the so-called *0-1 knapsack problem*. Formulate this new problem as an ILP. Verify that the optimal solution of the *linear relaxation* of the 0-1 knapsack problem for the items described in the table above is to assign items 1 and 2 completely to the knapsack and item 3 for 0.75, resulting in a total profit of 150.

(c) Solve the ILP formulated in (b) by hand using the branch-and-bound method. Use as initial upper bound of the solution the optimal fractional solution of (a) and as lower bound we consider only assigning items 1 and 2, which yields a profit of 120. In each node you can use the greedy heuristic of the fractional knapsack problem to find the solution of that node.

(d) Solve the ILP formulated in (b) by hand using (again) the branch-and-bound method. This time, use the following strategy: branch first on the nodes with the highest upper bound (i.e., the nodes with highest optimal solution value).

Ex. 3.5 (M+C, $\star$) Formulate as a mixed integer optimization problem and solve the following problem:

$$\frac{abc}{def} = \frac{1}{5},$$

where a, b, c, d, e and f can take distinct integer values in the set $\{1, 2, \ldots, 6\}$.

Hint 1: Each number/variable appears either in the numerator or in the denominator.
Hint 2: $\log(ab) = \log(a) + \log(b)$.

Ex. 3.6 (originally appeared as Exercise 3.4-8 of the book by Hillier and Lieberman)
Larry Edison is the director of the Computer Center for Buckly College. He now needs to schedule the staffing of the center. It is open from 8 A.M. until midnight. Larry has monitored the usage of the center at various times of the day, and determined that the following number of computer consultants are required:

Time of day	**Minimum number of consultants required to be on duty**
8 am – noon	4
Noon – 4 pm	8
4 pm. – 8 pm	10
8 pm. – midnight	6

Two types of computer consultants can be hired: full-time and part-time. The full-time consultants work for 8 consecutive hours in any of the following shifts: morning (8 A.M.–4 P.M.), afternoon (noon–8 P.M.), and evening (4 P.M.–midnight). Full-time consultants are paid \$40 per hour. Part-time consultants can be hired to work any of the four shifts listed in the above table. Part-time consultants are paid \$30 per hour. An additional requirement is that during every time period, there must be at least two full-time consultants on duty for every part-time consultant on duty. Larry would like to determine how many full-time and how many part-time workers should work each shift to meet the above requirements at the minimum possible cost. Formulate a LP for this problem.

Ex. 3.7 (originally appeared as Exercise 3.4-14 of the book by Hillier and Lieberman)
Oxbridge University maintains a powerful mainframe computer for research use by its faculty, Ph.D. students, and research associates. During all working hours, an operator must be available to operate and maintain the computer, as well as to perform some programming services. Beryl Ingram, the director of the computer facility, oversees the operation.

It is now the beginning of the fall semester, and Beryl is confronted with the problem of assigning different working hours to her operators. Because all the operators are currently enrolled in the university, they are available to work only a limited number of hours each day, as shown in the following table.

		Maximum Hours of Availability				
Operators	**Wage Rate**	**Mon.**	**Tue.**	**Wed.**	**Thurs.**	**Fri.**
K. C.	\$25/hour	6	0	6	0	6
D. H.	\$26/hour	0	6	0	6	0
H. B.	\$24/hour	4	8	4	0	4
S. C.	\$23/hour	5	5	5	0	5
K. S.	\$28/hour	3	0	3	8	0
N. K.	\$30/hour	0	0	0	6	2

There are six operators (four undergraduate students and two graduate students). They all have different wage rates because of differences in their experience with computers and in their programming ability. The above table shows their wage rates, along with the maximum number of hours that each can work each day. Each operator is guaranteed a certain minimum number of hours per week that will maintain an adequate knowledge of the operation. This level is set arbitrarily at 8 hours per week for the undergraduate students (K. C., D. H., H. B., and S. C.) and 7 hours per week for the graduate students (K. S. and N. K.). The computer facility is to be open for operation from 8 A.M. to 10 P.M. Monday through Friday with exactly one operator on duty during these hours. On Saturdays and Sundays, the computer is to be operated by other staff. Because of a tight budget, Beryl has to minimize cost. She wishes to determine the number of hours she should assign to each operator on each day. Formulate a LP for this problem.

[Back to the start of Chapter 3] [Back to Table of Contents]

Bibliography

[BJS10] M. S. Bazaraa, John J. Jarvis, and Hanif D. Sherali. *Linear programming and network flows*. John Wiley & Sons Inc, 2010, p. 748. ISBN: 0470462728.

[Cor+09] Paulo Cortez et al. "Modeling wine preferences by data mining from physicochemical properties". In: *Decision Support Systems* 47.4 (Nov. 2009), pp. 547–553. DOI: 10.1016/j.dss.2009.05.016.

[Har+17] William E. Hart et al. *Pyomo — Optimization Modeling in Python*. 2nd. Springer International Publishing, 2017. DOI: 10.1007/978-3-319-58821-6.

[KIG14] Serkan Kiranyaz, Turker Ince, and Moncef Gabbouj. "Optimization Techniques: An Overview". In: *Multidimensional Particle Swarm Optimization for Machine Learning and Pattern Recognition*. Springer Berlin Heidelberg, 2014, pp. 13–44. ISBN: 978-3-642-37846-1. DOI: 10.1007/978-3-642-37846-1_2.

[Kit11] Mitri Kitti. *History of Optimization*. http://www.mitrikitti.fi/opthist.html. Accessed: 2021-10-15. 2011.

[KY91] Hiroshi Konno and Hiroaki Yamazaki. "Mean-Absolute Deviation Portfolio Optimization Model and Its Applications to Tokyo Stock Market". In: *Management Science* 37.5 (1991), pp. 519–531. ISSN: 00251909, 15265501.