Mathematical Programming and Operations Research

Modeling, Algorithms, and Complexity

Examples in Excel and Python

(Work in progress)

Edited by: Robert Hildebrand

Contributors: Robert Hildebrand, Laurent Poirrier, Douglas Bish, Diego Moran

Version Compilation date: July 2, 2022

Preface

This entire book is a working manuscript. The first draft of the book is yet to be completed.

This book is being written and compiled using a number of open source materials. We will strive to properly cite all resources used and give references on where to find these resources. Although the material used in this book comes from a variety of licences, everything used here will be CC-BY-SA 4.0 compatible, and hence, the entire book will fall under a CC-BY-SA 4.0 license.

MAJOR ACKNOWLEDGEMENTS

I would like to acknowledge that substantial parts of this book were borrowed under a CC-BY-SA license. These substantial pieces include:

- "A First Course in Linear Algebra" by Lyryx Learning (based on original text by Ken Kuttler). A majority of their formatting was used along with selected sections that make up the appendix sections on linear algebra. We are extremely grateful to Lyryx for sharing their files with us. They do an amazing job compiling their books and the templates and formatting that we have borrowed here clearly took a lot of work to set up. Thank you for sharing all of this material to make structuring and formating this book much easier! See subsequent page for list of contributors.
- "Foundations of Applied Mathmatics" with many contributors. See `https://github.com/Foundations-of-Applied-Mathematics`. Several sections from these notes were used along with some formatting. Some of this content has been edited or rearranged to suit the needs of this book. This content comes with some great references to code and nice formatting to present code within the book. See subsequent page with list of contributors.
- "Linear Inequalities and Linear Programming" by Kevin Cheung. See `https://github.com/dataopt/lineqlpbook`. These notes are posted on GitHub in a ".Rmd" format for nice reading online. This content was converted to LaTeXusing Pandoc. These notes make up a substantial section of the Linear Programming part of this book.
- Linear Programming notes by Douglas Bish. These notes also make up a substantial section of the Linear Programming part of this book.

I would also like to acknowledge Laurent Porrier and Diego Moran for contributing various notes on linear and integer programming.

I would also like to thank Jamie Fravel for helping to edit this book and for contributing chapters, examples, and code.

Contents

1. Resources and Notation

Here are a list of resources that may be useful as alternative references or additional references.

Free notes and textbooks

- Linear Programming by K.J. Mtetwa, David
- A first course in optimization by Jon Lee
- Introduction to Optimizaiton Notes by Komei Fukuda
- Convex Optimization by Bord and Vandenberghe
- LP notes of Michel Goemans from MIT
- Understanding and Using Linear Programming - Matousek and Gärtner [Downloadable from Springer with University account]
- Operations Research Problems Statements and Solutions - Raúl PolerJosefa Mula Manuel Díaz-Madroñero [Downloadable from Springer with University account]

Notes, books, and videos by various solver groups

- AIMMS Optimization Modeling
- Optimization Modeling with LINGO by Linus Schrage
- The AMPL Book
- Microsoft Excel 2019 Data Analysis and Business Modeling, Sixth Edition, by Wayne Winston - Available to read for free as an e-book through Virginia Tech library at Orielly.com.
- Lesson files for the Winston Book
- Video instructions for solver and an example workbook
- 

GUROBI LINKS

- Go to `https://github.com/Gurobi` and download the example files.
- Essential ingredients
- Gurobi Linear Programming tutorial
- Gurobi tutorial MILP
- GUROBI - Python 1 - Modeling with GUROBI in Python
- GUROBI - Python II: Advanced Algebraic Modeling with Python and Gurobi
- GUROBI - Python III: Optimization and Heuristics
- Webinar Materials
- GUROBI Tutorials

HOW TO PROVE THINGS

- Hammack - Book of Proof

STATISTICS

- Open Stax - Introductory Statistics

LINEAR ALGEBRA

- Beezer - A first course in linear algebra
- Selinger - Linear Algebra
- Cherney, Denton, Thomas, Waldron - Linear Algebra

REAL ANALYSIS

- Mathematical Analysis I by Elias Zakon

Discrete Mathematics, Graphs, Algorithms, and Combinatorics

- Levin - Discrete Mathematics - An Open Introduction, 3rd edition
- Github - Discrete Mathematics: an Open Introduction CC BY SA
- Keller, Trotter - Applied Combinatorics (CC-BY-SA 4.0)
- Keller - Github - Applied Combinatorics

Programming with Python

- A Byte of Python
- Github - Byte of Python (CC-BY-SA)

Also, go to `https://github.com/open-optimization/open-optimization-or-examples` to look at more examples.

Notation

- $\mathbf{1}$ - a vector of all ones (the size of the vector depends on context)
- $\forall$ - for all
- $\exists$ - there exists
- $\in$ - in
- $\therefore$ - therefore
- $\Rightarrow$ - implies
- s.t. - such that (or sometimes "subject to".... from context?)
- $\{0,1\}$ - the set of numbers 0 and 1
- $\mathbb{Z}$ - the set of integers (e.g. $1,2,3,-1,-2,-3,...$)
- $\mathbb{Q}$ - the set of rational numbers (numbers that can be written as p/q for $p,q \in \mathbb{Z}$ (e.g. $1,1/6,27/2$)
- $\mathbb{R}$ - the set of all real numbers (e.g. 1, 1.5, π, e, $-11/5$)
- $\setminus$ - setminus, (e.g. $\{0,1,2,3\} \setminus \{0,3\} = \{1,2\}$)
- $\cup$ - union (e.g. $\{1,2\} \cup \{3,5\} = \{1,2,3,5\}$
- $\cap$ - intersection (e.g. $\{1,2,3,4\} \cap \{3,4,5,6\} = \{3,4\}$)
- $\{0,1\}^4$ - the set of 4 dimensional vectors taking values 0 or 1, (e.g. $[0,0,1,0]$ or $[1,1,1,1]$)
- $\mathbb{Z}^4$ - the set of 4 dimensional vectors taking integer values (e.g., $[1,-5,17,3]$ or $[6,2,-3,-11]$)
- $\mathbb{Q}^4$ - the set of 4 dimensional vectors taking rational values (e.g. $[1.5,3.4,-2.4,2]$)
- $\mathbb{R}^4$ - the set of 4 dimensional vectors taking real values (e.g. $[3,\pi,-e,\sqrt{2}]$)
- $\sum_{i=1}^{4} i = 1+2+3+4$
- $\sum_{i=1}^{4} i^2 = 1^2+2^2+3^2+4^4$
- $\sum_{i=1}^{4} x_i = x_1+x_2+x_3+x_4$
- $\square$ - this is a typical Q.E.D. symbol that you put at the end of a proof meaning "I proved it."

- For $x, y \in \mathbb{R}^3$, the following are equivalent (note, in other contexts, these notations can mean different things)
  - $x^\top y$ *matrix multiplication*
  - $x \cdot y$ *dot product*
  - $\langle x, y \rangle$ *inner product*

  and evaluate to $\sum_{i=1}^{3} x_i y_i = x_1 y_1 + x_2 y_2 + x_3 y_3$.

A sample sentence:

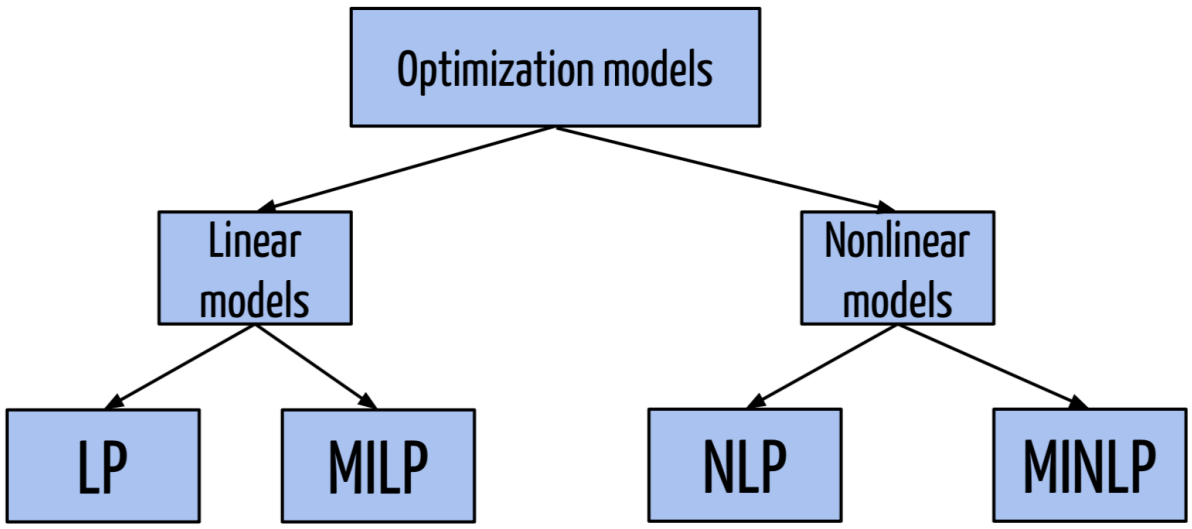
$$\forall x \in \mathbb{Q}^n \; \exists y \in \mathbb{Z}^n \setminus \{0\}^n s.t. x^\top y \in \{0, 1\}$$

"For all non-zero rational vectors x in n-dimensions, there exists a non-zero n-dimensional integer vector y such that the dot product of x with y evaluates to either 0 or 1."

2. Mathematical Programming

Add discussion of Optimization, Operations Research, and Mathematical Programming including background and applications. Also, give an introduction to the content in this book, what you will learn by working though the book, and why this book is interesting and different from other sources.

We will state main general problem classes to be associated with in these notes. These are Linear Programming (LP), Mixed-Integer Linear Programming (MILP), Non-Linear Programming (NLP), and Mixed-Integer Non-Linear Programming (MINLP).



© **problem-class-diagram**[1]

Figure 2.1: problem-class-diagram

Along with each problem class, we will associate a complexity class for the general version of the problem. See **??** for a discussion of complexity classes. Although we will often state that input data for a problem comes from $\mathbb{R}$, when we discuss complexity of such a problem, we actually mean that the data is rational, i.e., from $\mathbb{Q}$, and is given in binary encoding.

[1] **problem-class-diagram**, from **problem-class-diagram**. **problem-class-diagram**, **problem-class-diagram**.

2.1 Linear Programming (LP)

Some linear programming background, theory, and examples will be provided in **??**.

Linear Programming (LP):

Polynomial time (P)

Given a matrix $A \in \mathbb{R}^{m \times n}$, vector $b \in \mathbb{R}^m$ and vector $c \in \mathbb{R}^n$, the *linear programming* problem is

$$\begin{aligned} \max \quad & c^\top x \\ \text{s.t.} \quad & Ax \leq b \\ & x \geq 0 \end{aligned} \tag{2.1}$$

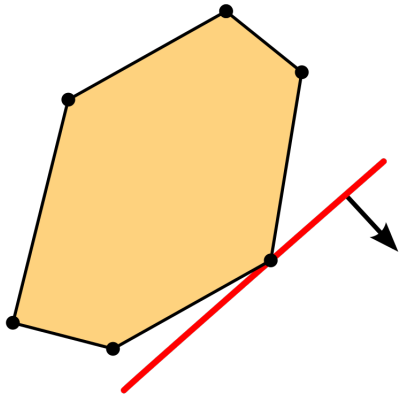
Linear programming can come in several forms, whether we are maximizing or minimizing, or if the constraints are $\leq, =$ or $\geq$. One form commonly used is *Standard Form* given as

Linear Programming (LP) Standard Form:

Polynomial time (P)

Given a matrix $A \in \mathbb{R}^{m \times n}$, vector $b \in \mathbb{R}^m$ and vector $c \in \mathbb{R}^n$, the *linear programming* problem in *standard form* is

$$\begin{aligned} \max \quad & c^\top x \\ \text{s.t.} \quad & Ax = b \\ & x \geq 0 \end{aligned} \tag{2.2}$$



© wiki/File/linear-programming.png[2]

Figure 2.2: Linear programming constraints and objective.

Figure 2.2

Exercise 2.1:

Start with a problem in form given as (2.1) and convert it to standard form (2.2) by adding at most m many new variables and by enlarging the constraint matrix A by at most m new columns.

2.2 Mixed-Integer Linear Programming (MILP)

Mixed-integer linear programming will be the focus of Sections **??**,**??**, **??**, and **??**. Recall that the notation $\mathbb{Z}$ means the set of integers and the set $\mathbb{R}$ means the set of real numbers. The first problem of interest here is a *binary integer program* (BIP) where all n variables are binary (either 0 or 1).

Binary Integer programming (BIP):

NP-Complete

Given a matrix $A \in \mathbb{R}^{m \times n}$, vector $b \in \mathbb{R}^m$ and vector $c \in \mathbb{R}^n$, the *binary integer programming* problem is

$$\begin{aligned} \max \quad & c^\top x \\ \text{s.t.} \quad & Ax \leq b \\ & x \in \{0,1\}^n \end{aligned} \tag{2.1}$$

A slightly more general class is the class of *Integer Linear Programs* (ILP). Often this is referred to as *Integer Program* (IP), although this term could leave open the possibility of non-linear parts.

Figure 2.3

Integer Linear Programming (ILP):

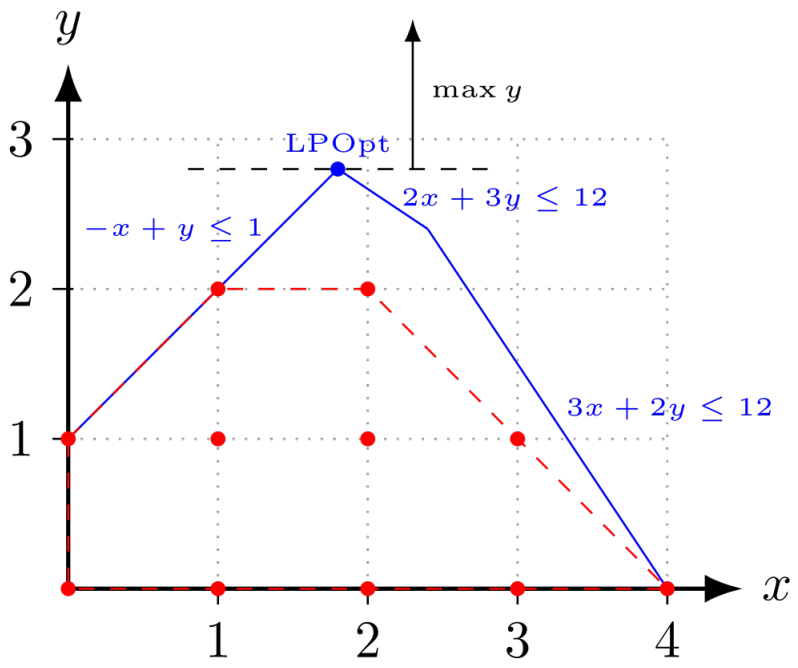
NP-Complete

Given a matrix $A \in \mathbb{R}^{m \times n}$, vector $b \in \mathbb{R}^m$ and vector $c \in \mathbb{R}^n$, the *integer linear programming* problem is

$$\begin{aligned} \max \quad & c^\top x \\ \text{s.t.} \quad & Ax \leq b \\ & x \in \mathbb{Z}^n \end{aligned} \tag{2.2}$$

[2] **wiki/File/linear-programming.png**, from **wiki/File/linear-programming.png**. **wiki/File/linear-programming.png**, **wiki/File/linear-programming.png**.

[3] **wiki/File/integer-programming.png**, from **wiki/File/integer-programming.png**. **wiki/File/integer-programming.png**, **wiki/File/integer-programming.png**.



© wiki/File/integer-programming.png[3]

Figure 2.3: Comparing the LP relaxation to the IP solutions.

An even more general class is *Mixed-Integer Linear Programming (MILP)*. This is where we have n integer variables $x_1,\ldots,x_n \in \mathbb{Z}$ and d continuous variables $x_{n+1},\ldots,x_{n+d} \in \mathbb{R}$. Succinctly, we can write this as $x \in \mathbb{Z}^n \times \mathbb{R}^d$, where $\times$ stands for the *cross-product* between two spaces.

Below, the matrix A now has $n+d$ columns, that is, $A \in \mathbb{R}^{m\times n+d}$. Also note that we have not explicitly enforced non-negativity on the variables. If there are non-negativity restrictions, this can be assumed to be a part of the inequality description $Ax \leq b$.

Mixed-Integer Linear Programming (MILP):

NP-Complete

Given a matrix $A \in \mathbb{R}^{m\times(n+d)}$, vector $b \in \mathbb{R}^m$ and vector $c \in \mathbb{R}^{n+d}$, the *mixed-integer linear programming* problem is

$$\begin{aligned} \max \quad & c^\top x \\ \text{s.t.} \quad & Ax \leq b \\ & x \in \mathbb{Z}^n \times \mathbb{R}^d \end{aligned} \tag{2.3}$$

2.3 Non-Linear Programming (NLP)

NLP:

NP-Hard

Given a function $f(x)\colon \mathbb{R}^d \to \mathbb{R}$ and other functions $f_i(x)\colon \mathbb{R}^d \to \mathbb{R}$ for $i = 1,\ldots,m$, the *nonlinear programming* problem is

$$\begin{aligned} \min \quad & f(x) \\ \text{s.t.} \quad & f_i(x) \leq 0 \quad \text{for } i = 1,\ldots,m \\ & x \in \mathbb{R}^d \end{aligned} \tag{2.1}$$

Nonlinear programming can be separated into convex programming and non-convex programming. These two are very different beasts and it is important to distinguish between the two.

2.3.1. Convex Programming

Here the functions are all **convex!**

Convex Programming:

Polynomial time (P) (typically)

Given a convex function $f(x)\colon \mathbb{R}^d \to \mathbb{R}$ and convex functions $f_i(x)\colon \mathbb{R}^d \to \mathbb{R}$ for $i = 1,\ldots,m$, the *convex programming* problem is

$$\begin{aligned} \min \quad & f(x) \\ \text{s.t.} \quad & f_i(x) \leq 0 \quad \text{for } i = 1,\ldots,m \\ & x \in \mathbb{R}^d \end{aligned} \tag{2.2}$$

Example 2.2

onvex programming is a generalization of linear programming. This can be seen by letting $f(x) = c^\top x$ *and* $f_i(x) = A_i x - b_i$.

2.3.2. Non-Convex Non-linear Programming

When the function f or functions f_i are non-convex, this becomes a non-convex nonlinear programming problem. There are a few complexity issues with this.

IP AS NLP As seen above, quadratic constraints can be used to create a feasible region with discrete solutions. For example

$$x(1-x) = 0$$

has exactly two solutions: $x = 0, x = 1$. Thus, quadratic constraints can be used to model binary constraints.

Binary Integer programming (BIP) as a NLP:

NP-Hard

Given a matrix $A \in \mathbb{R}^{m \times n}$, vector $b \in \mathbb{R}^m$ and vector $c \in \mathbb{R}^n$, the *binary integer programming* problem is

$$\begin{aligned} \max \quad & c^\top x \\ \text{s.t.} \quad & Ax \leq b \\ & \cancel{x \in \{0,1\}^n} \\ & x_i(1-x_i) = 0 \quad \text{for } i = 1, \ldots, n \end{aligned} \tag{2.3}$$

2.4 Mixed-Integer Non-Linear Programming (MINLP)

Fill in this section with formulas and discuss applications.

2.4.1. Convex Mixed-Integer Non-Linear Programming

2.4.2. Non-Convex Mixed-Integer Non-Linear Programming

Part I

Linear Programming

3. Linear Programming

Add introduction to the section. Add description of what is to come and what applications we might look at.

Outcomes

A Generic Linear Program (LP)

Decision Variables:
x_i : continuous variables ($x_i \in \mathbb{R}$, i.e., a real number), $\forall i = 1, \cdots, 3$.

Parameters (known input parameters):
c_i : cost coefficients $\forall i = 1, \ldots, n$
a_{ij} : constraint coefficients $\forall i = 1, \ldots, n,\ j = 1, \ldots, m$
b_j : right hand side coefficient for constraint j, $j = 1, \ldots, m$
The problem we will consider is

$$\begin{aligned} \max\ & z = c_1 x_1 + \cdots + c_n x_n \\ s.t.\ & a_{11} x_1 + \cdots + a_{1n} x_n \leq b_1 \\ & \vdots \\ & a_{m1} x_1 + \cdots + a_{mn} x_n \leq b_m \end{aligned} \tag{3.1}$$

For example, in 3 variables and 4 contraints this could look like the following. The following example considers other types of constraints, i.e., $\geq$ and $=$. We will show how all these forms can be converted later.

Decision Variables:
x_i : continuous variables ($x_i \in \mathbb{R}$, i.e., a real number), $\forall i = 1, \cdots, 3$.

Parameters (known input parameters):
c_i : cost coefficients $\forall i = 1, \ldots, 3$
a_{ij} : constraint coefficients $\forall i = 1, \ldots, 3,\ j = 1, \ldots, 4$
b_j : right hand side coefficient for constraint j, $j = 1, \ldots, 4$

$$\begin{aligned}
\text{Min } & z = c_1x_1 + c_2x_2 + c_3x_3 && (3.2)\\
\text{s.t. } & a_{11}x_1 + a_{12}x_2 + a_{13}x_3 \geq b_1 && (3.3)\\
& a_{21}x_1 + a_{22}x_2 + a_{23}x_3 \leq b_2 && (3.4)\\
& a_{31}x_1 + a_{32}x_2 + a_{33}x_3 = b_3 && (3.5)\\
& a_{41}x_1 + a_{42}x_2 + a_{43}x_3 \geq b_4 && (3.6)\\
& x_1 \geq 0, x_2 \leq 0, x_3 \textit{ urs.} && (3.7)
\end{aligned}$$

Definition 3.1: Linear Function

A function $z : \mathbb{R}^n \to \mathbb{R}$ is linear if there are constants $c_1, \ldots, c_n \in \mathbb{R}$ so that:

$$z(x_1, \ldots, x_n) = c_1x_1 + \cdots + c_nx_n \tag{3.8}$$

Lemma 3.2: Linear Function

If $z : \mathbb{R}^n \to \mathbb{R}$ is linear then for all $\mathbf{x}_1, \mathbf{x}_2 \in \mathbb{R}^n$ and for all scalar constants $\alpha \in \mathbb{R}$ we have:

$$z(\mathbf{x}_1 + \mathbf{x}_2) = z(\mathbf{x}_1) + z(\mathbf{x}_2) \tag{3.9}$$
$$z(\alpha\mathbf{x}_1) = \alpha z(\mathbf{x}_1) \tag{3.10}$$

Exercise 3.3

Prove Lemma 3.

For the time being, we will eschew the general form and focus exclusively on linear programming problems with two variables. Using this limited case, we will develop a graphical method for identifying optimal solutions, which we will generalize later to problems with arbitrary numbers of variables.

Example: Toy Maker Excel PuLP Gurobipy

Consider the problem of a toy company that produces toy planes and toy boats. The toy company can sell its planes for \$10 and its boats for \$8 dollars. It costs \$3 in raw materials to make a plane and \$2 in raw materials to make a boat. A plane requires 3 hours to make and 1 hour to finish while a boat requires 1 hour to make and 2 hours to finish. The toy company knows it will not sell anymore than 35 planes per week. Further, given the number of workers, the company cannot spend anymore than 160 hours per week finishing toys and 120 hours per week making toys. The company wishes to maximize the profit it makes by choosing how much of each toy to produce.

We can represent the profit maximization problem of the company as a linear programming problem. Let x_1 be the number of planes the company will produce and let x_2 be the number of boats the company will produce. The profit for each plane is $\$10 - \$3 = \$7$ per plane and the profit for each boat is $\$8 - \$2 = \$6$ per boat. Thus the total profit the company will make is:

$$z(x_1, x_2) = 7x_1 + 6x_2 \tag{3.11}$$

The company can spend no more than 120 hours per week making toys and since a plane takes 3 hours to make and a boat takes 1 hour to make we have:

$$3x_1 + x_2 \leq 120 \tag{3.12}$$

Likewise, the company can spend no more than 160 hours per week finishing toys and since it takes 1 hour to finish a plane and 2 hour to finish a boat we have:

$$x_1 + 2x_2 \leq 160 \tag{3.13}$$

Finally, we know that $x_1 \leq 35$, since the company will make no more than 35 planes per week. Thus the complete linear programming problem is given as:

$$\begin{cases} \max \ z(x_1, x_2) = 7x_1 + 6x_2 \\ s.t. \ 3x_1 + x_2 \leq 120 \\ \quad x_1 + 2x_2 \leq 160 \\ \quad x_1 \leq 35 \\ \quad x_1 \geq 0 \\ \quad x_2 \geq 0 \end{cases} \tag{3.14}$$

Exercise 3.4: Chemical Manufacturing

A chemical manufacturer produces three chemicals: A, B and C. These chemical are produced by two processes: 1 and 2. Running process 1 for 1 hour costs $4 and yields 3 units of chemical A, 1 unit of chemical B and 1 unit of chemical C. Running process 2 for 1 hour costs $1 and produces 1 units of chemical A, and 1 unit of chemical B (but none of Chemical C). To meet customer demand, at least 10 units of chemical A, 5 units of chemical B and 3 units of chemical C must be produced daily. Assume that the chemical manufacturer wants to minimize the cost of production. Develop a linear programming problem describing the constraints and objectives of the chemical manufacturer. [Hint: Let x_1 be the amount of time Process 1 is executed and let x_2 be amount of time Process 2 is executed. Use the coefficients above to express the cost of running Process 1 for x_1 time and Process 2 for x_2 time. Do the same to compute the amount of chemicals A, B, and C that are produced.]

3.1 Modeling Assumptions in Linear Programming

Outcomes

1. *Address crucial assumptions when chosing to model a problem with linear programming.*

Inspecting Example 1 (or the more general Problem 3.1) we can see there are several assumptions that must be satisfied when using a linear programming model. We enumerate these below:

Proportionality Assumption A problem can be phrased as a linear program only if the contribution to the objective function *and* the left-hand-side of each constraint by each decision variable $(x_1, \ldots, x_n)$ is proportional to the value of the decision variable.

Additivity Assumption A problem can be phrased as a linear programming problem only if the contribution to the objective function *and* the left-hand-side of each constraint by any decision variable x_i $(i = 1, \ldots, n)$ is completely independent of any other decision variable x_j $(j \neq i)$ and additive.

Divisibility Assumption A problem can be phrased as a linear programming problem only if the quantities represented by each decision variable are infinitely divisible (i.e., fractional answers make sense).

Certainty Assumption A problem can be phrased as a linear programming problem only if the coefficients in the objective function and constraints are known with certainty.

The first two assumptions simply assert (in English) that both the objective function and functions on the left-hand-side of the (in)equalities in the constraints are linear functions of the variables $x_1, \ldots, x_n$.

The third assumption asserts that a valid optimal answer could contain fractional values for decision variables. It's important to understand how this assumption comes into play–even in the toy making example. Many quantities can be divided into non-integer values (ounces, pounds etc.) but many other quantities cannot be divided. For instance, can we really expect that it's reasonable to make $\frac{1}{2}$ a plane in the toy making example? When values must be constrained to true integer values, the linear programming problem is called an *integer programming problem*. These problems are outside the scope of this course, but there is a *vast* literature dealing with them [**PS98**, **WN99**]. For many problems, particularly when the values of the decision variables may become large, a fractional optimal answer could be obtained and then rounded to the nearest integer to obtain a reasonable answer. For example, if our toy problem were re-written so that the optimal answer was to make 1045.3 planes, then we could round down to 1045.

The final assumption asserts that the coefficients (e.g., profit per plane or boat) is known with absolute certainty. In traditional linear programming, there is no lack of knowledge about the make up of the objective function, the coefficients in the left-hand-side of the constraints or the bounds on the right-hand-sides of the constraints. There is a literature on *stochastic programming* [**KW94**, **BN02**] that relaxes some of these assumptions, but this too is outside the scope of the course.

Exercise 3.5

In a short sentence or two, discuss whether the problem given in Example 1 meets all of the assumptions of a scenario that can be modeled by a linear programming problem. Do the same for Exercise 3. [Hint: Can you make $\frac{2}{3}$ of a toy? Can you run a process for $\frac{1}{3}$ of an hour?]

Exercise 3.6: Stochastic Objective

Suppose the costs are not known with certainty but instead a probability distribution for each value of c_i $(i = 1,\ldots,n)$ is known. Suggest a way of constructing a linear program from the probability distributions.
[Hint: Suppose I tell you that I'll give you a uniformly random amount of money between \$1 and \$2. How much money do you expect to receive? Use the same reasoning to answer the question.]

3.2 Examples

Outcomes

A. *Learn how to format a linear optimization problem.*
B. *Identify and understand common classes of linear optimization problems.*

We will begin with a few examples, and then discuss specific problem types that occur often.

Example: Production with welding robot Excel PuLP Gurobipy

You have 21 units of transparent aluminum alloy (TAA), LazWeld1, a joining robot leased for 23 hours, and CrumCut1, a cutting robot leased for 17 hours of aluminum cutting. You also have production code for a bookcase, desk, and cabinet, along with commitments to buy any of these you can produce for $18, $16, and $10 apiece, respectively. A bookcase requires 2 units of TAA, 3 hours of joining, and 1 hour of cutting, a desk requires 2 units of TAA, 2 hours of joining, and 2 hour of cutting, and a cabinet requires 1 unit of TAA, 2 hours of joining, and 1 hour of cutting. Formulate an LP to maximize your revenue given your current resources.

Solution.

Sets:

- The types of objects $=\{$ bookcase, desk, cabinet$\}$.

Parameters:

- Purchase cost of each object
- Units of TAA needed for each object
- Hours of joining needed for each object
- Hours of cutting needed for each object
- Hours of TAA, Joining, and Cutting available on robots

Decision variables:

x_i : number of units of product i to produce,
for all $i =$bookcase, desk, cabinet.

Objective and Constraints:

$$\begin{aligned} \max z &= 18x_1 + 16x_2 + 10x_3 && \text{(profit)} \\ s.t. 2x_1 + 2x_2 + 1x_3 &\leq 21 && (TAA) \\ 3x_1 + 2x_2 + 2x_3 &\leq 23 && (LazWeld1) \\ 1x_1 + 2x_2 + 1x_3 &\leq 17 && (CrumCut1) \\ x_1, x_2, x_3 &\geq 0. \end{aligned}$$

♠

Example 3.7: The Diet Problem

In the future (as envisioned in a bad 70's science fiction film) all food is in tablet form, and there are four types, green, blue, yellow, and red. A balanced, futuristic diet requires, at least 20 units of Iron, 25 units of Vitamin B, 30 units of Vitamin C, and 15 units of Vitamin D. Formulate an LP that ensures a balanced diet at the minimum possible cost.

Tablet	Iron	B	C	D	Cost ($)
green (1)	6	6	7	4	1.25
blue (2)	4	5	4	9	1.05
yellow (3)	5	2	5	6	0.85
red (4)	3	6	3	2	0.65

Solution. Now we formulate the problem: Sets:

- Set of tablets $\{1,2,3,4\}$

Parameters:

- Iron in each tablet
- Vitamin B in each tablet
- Vitamin C in each tablet
- Vitamin D in each tablet
- Cost of each tablet

Decision variables:
x_i : number of tablet of type i to include in the diet, $\forall i \in \{1,2,3,4\}$.

Objective and Constraints:

$$\begin{aligned} \text{Min} \quad & z = 1.25x_1 + 1.05x_2 + 0.85x_3 + 0.65x_4 \\ \text{s.t.} \quad & 6x_1 + 4x_2 + 5x_3 + 3x_4 \geq 20 \\ & 6x_1 + 5x_2 + 2x_3 + 6x_4 \geq 25 \\ & 7x_1 + 4x_2 + 5x_3 + 3x_4 \geq 30 \\ & 4x_1 + 9x_2 + 6x_3 + 2x_4 \geq 15 \\ & x_1, x_2, x_3, x_4 \geq 0. \end{aligned}$$

♠

Example 3.8: The Next Diet Problem

Progress is important, and our last problem had too many tablets, so we are going to produce a single, purple, 10 gram tablet for our futuristic diet requires, which are at least 20 units of Iron, 25 units of Vitamin B, 30 units of Vitamin C, and 15 units of Vitamin D, and 2000 calories. The tablet is made from blending 4 nutritious chemicals; the following table shows the units of our nutrients per, and cost of, grams of each chemical. Formulate an LP that ensures a balanced diet at the minimum possible cost.

Solution. Sets:

- Set of chemicals $\{1,2,3,4\}$

Tablet	Iron	B	C	D	Calories	Cost ($)
Chem 1	6	6	7	4	1000	1.25
Chem 2	4	5	4	9	250	1.05
Chem 3	5	2	5	6	850	0.85
Chem 4	3	6	3	2	750	0.65

Parameters:

- Iron in each chemical
- Vitamin B in each chemical
- Vitamin C in each chemical
- Vitamin D in each chemical
- Cost of each chemical

Decision variables:
x_i : grams of chemical i to include in the purple tablet, $\forall i = 1,2,3,4$.
Objective and Constraints:

$$
\begin{aligned}
\text{Min } & z = 1.25x_1 + 1.05x_2 + 0.85x_3 + 0.65x_4 \\
\text{s.t. } & 6x_1 + 4x_2 + 5x_3 + 3x_4 \geq 20 \\
& 6x_1 + 5x_2 + 2x_3 + 6x_4 \geq 25 \\
& 7x_1 + 4x_2 + 5x_3 + 3x_4 \geq 30 \\
& 4x_1 + 9x_2 + 6x_3 + 2x_4 \geq 15 \\
& 1000x_1 + 250x_2 + 850x_3 + 750x_4 \geq 2000 \\
& x_1 + x_2 + x_3 + x_4 = 10 \\
& x_1, x_2, x_3, x_4 \geq 0.
\end{aligned}
$$

Example 3.9: Work Scheduling Problem

You are the manager of LP Burger. The following table shows the minimum number of employees required to staff the restaurant on each day of the week. Each employees must work for five consecutive days. Formulate an LP to find the minimum number of employees required to staff the restaurant.

Day of Week	Workers Required
1 = Monday	6
2 = Tuesday	4
3 = Wednesday	5
4 = Thursday	4
5 = Friday	3
6 = Saturday	7
7 = Sunday	7

Solution. Decision variables:

Decision variables:
x_i : the number of workers that start 5 consecutive days of work on day i, $i = 1, \cdots, 7$

$$
\begin{aligned}
\text{Min} \quad & z = x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 \\
\text{s.t.} \quad & x_1 + x_4 + x_5 + x_6 + x_7 \geq 6 \\
& x_2 + x_5 + x_6 + x_7 + x_1 \geq 4 \\
& x_3 + x_6 + x_7 + x_1 + x_2 \geq 5 \\
& x_4 + x_7 + x_1 + x_2 + x_3 \geq 4 \\
& x_5 + x_1 + x_2 + x_3 + x_4 \geq 3 \\
& x_6 + x_2 + x_3 + x_4 + x_5 \geq 7 \\
& x_7 + x_3 + x_4 + x_5 + x_6 \geq 7 \\
& x_1, x_2, x_3, x_4, x_5, x_6, x_7 \geq 0.
\end{aligned}
$$

The solution is as follows:

LP Solution	IP Solution
$z_{LP} = 7.333$	$z_I = 8.0$
$x_1 = 0$	$x_1 = 0$
$x_2 = 0.333$	$x_2 = 0$
$x_3 = 1$	$x_3 = 0$
$x_4 = 2.333$	$x_4 = 3$
$x_5 = 0$	$x_5 = 0$
$x_6 = 3.333$	$x_6 = 4$
$x_7 = 0.333$	$x_7 = 1$

♠

Example 3.10: LP Burger - extended

LP Burger has changed it's policy, and allows, at most, two part time workers, who work for two consecutive days in a week. Formulate this problem.

Solution. Decision variables:
x_i : the number of workers that start 5 consecutive days of work on day i, $i = 1, \cdots, 7$
y_i : the number of workers that start 2 consecutive days of work on day i, $i = 1, \cdots, 7$.

$$
\begin{aligned}
\text{Min} \quad & z = 5(x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7) \\
& + 2(y_1 + y_2 + y_3 + y_4 + y_5 + y_6 + y_7) \\
\text{s.t.} \quad & x_1 + x_4 + x_5 + x_6 + x_7 + y_1 + y_7 \geq 6 \\
& x_2 + x_5 + x_6 + x_7 + x_1 + y_2 + y_1 \geq 4 \\
& x_3 + x_6 + x_7 + x_1 + x_2 + y_3 + y_2 \geq 5 \\
& x_4 + x_7 + x_1 + x_2 + x_3 + y_4 + y_3 \geq 4 \\
& x_5 + x_1 + x_2 + x_3 + x_4 + y_5 + y_4 \geq 3 \\
& x_6 + x_2 + x_3 + x_4 + x_5 + y_6 + y_5 \geq 7 \\
& x_7 + x_3 + x_4 + x_5 + x_6 + y_7 + y_6 \geq 7 \\
& y_1 + y_2 + y_3 + y_4 + y_5 + y_6 + y_7 \leq 2 \\
& x_i \geq 0, y_i \geq 0, \forall i = 1, \cdots, 7.
\end{aligned}
$$

♠

3.2.1. Knapsack Problem

3.2.2. Work Scheduling

3.2.3. Assignment Problem

Consider the assignment of n teams to n projects, where each team ranks the projects, where their favorite project is given a rank of n, their next favorite $n-1$, and their least favorite project is given a rank of 1. The assignment problem is formulated as follows (we denote ranks using the R-parameter):

Variables:

x_{ij} : 1 if project i assigned to team j, else 0.

$$\begin{aligned}
\text{Max } z &= \sum_{i=1}^{n}\sum_{j=1}^{n} R_{ij}x_{ij} \\
\text{s.t. } &\sum_{i=1}^{n} x_{ij} = 1, \ \ \forall j = 1,\cdots,n \\
&\sum_{j=1}^{n} x_{ij} = 1, \ \ \forall i = 1,\cdots,n \\
&x_{ij} \geq 0, \ \ \forall i = 1,\cdots,n, j = 1,\cdots,n.
\end{aligned}$$

The assignment problem has an integrality property, such that if we remove the binary restriction on the x variables (now just non-negative, i.e., $x_{ij} \geq 0$) then we still get binary assignments, despite the fact that it is now an LP. This property is very interesting and useful. Of course, the objective function might not quite what we want, we might be interested ensuring that the team with the worst assignment is as good as possible (a fairness criteria). One way of doing this is to modify the assignment problem using a max-min objective:

Max-min Assignment-like Formulation

$$\begin{aligned}
Max \quad & z \\
s.t. \quad & \sum_{i=1}^{n} x_{ij} = 1, \ \ \forall j = 1,\cdots,n \\
& \sum_{j=1}^{n} x_{ij} = 1, \ \ \forall i = 1,\cdots,n \\
& x_{ij} \geq 0, \ \ \forall i = 1,\cdots,n, J = 1,\cdots,n \\
& z \leq \sum_{i=1}^{n} R_{ij}x_{ij}, \ \ \forall j = 1,\cdots,n.
\end{aligned}$$

Does this formulation have the integrality property (it is not an assignment problem)? Consider a very simple example where two teams are to be assigned to two projects and the teams give the projects the following rankings: Both teams prefer Project 2. For both problems, if we remove the binary restriction on

	Project 1	Project 2
Team 1	2	1
Team 2	2	1

the x-variable, they can take values between (and including) zero and one. For the assignment problem the optimal solution will have $z = 3$, and fractional x-values will not improve z. For the max-min assignment problem this is not the case, the optimal solution will have $z = 1.5$, which occurs when each team is assigned half of each project (i.e., for Team 1 we have $x_{11} = 0.5$ and $x_{21} = 0.5$).

3.2.4. Multi period Models

Fill in this subsection

3.2.4.1. Production Planning

3.2.4.2. Crop Planning

3.2.5. Mixing Problems

3.2.6. Financial Planning

Fill in this subsection

3.2.7. Network Flow

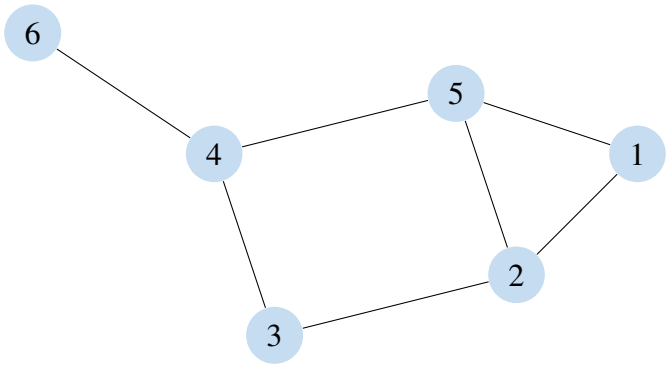
Resources

- *MIT - CC BY NC SA 4.0 license*
- *Slides for Algorithms book by Kleinberg-Tardos*

To begin a discussion on Network flow, we first need to discuss graphs.

3.2.7.1. Graphs

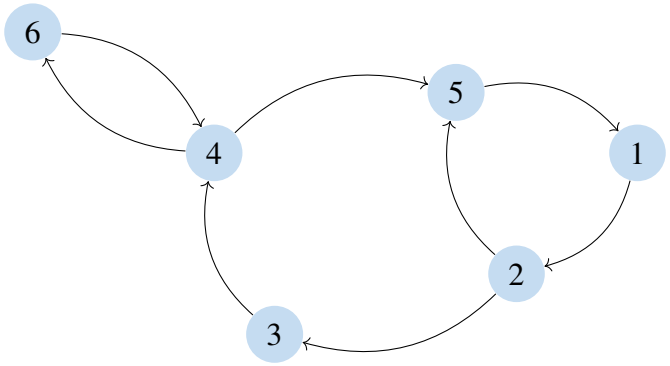
A graph $G = (V, E)$ is defined by a set of vertices V and a set of edges E that contains pairs of vertices. For example, the following graph G can be described by the vertex set $V = \{1,2,3,4,5,6\}$ and the edge set $E = \{(4,6),(4,5),(5,1)(1,2),(2,5),(2,3),(3,4)\}$.



In an undirected graph, we do not distinguish the direction of the edge. That is, for two vertices $i, j \in V$, we can equivalently write (i, j) or (j, i) to represent the edge.

Alternatively, we will want to consider directed graphs. We denote these as $G = (V, \mathscr{A})$ where $\mathscr{A}$ is a set of arcs where an arc is a directed edge.

For example, the following directed graph G can be described by the vertex set $V = \{1,2,3,4,5,6\}$ and the edge set $\mathscr{A} = \{(4,6),(6,4),(4,5),(5,1)(1,2),(2,5),(2,3),(3,4)\}$.



SETS A finite network G is described by a finite set of vertices V and a finite set $\mathscr{A}$ of arcs. Each arc (i, j) has two key attributes, namely its tail $j \in V$ and its head $i \in V$.

We think of a (single) commodity as being allowed to "flow" along each arc, from its tail to its head.

VARIABLES Indeed, we have "flow" variables

$$x_{ij} := \text{ amount of flow on arc}(i,j) \text{ from vertex } i \text{ to vertex } j,$$

for all $(i, j) \in \mathscr{A}$.

3.2.7.2. Maximum Flow Problem

$$\max \sum_{(s,i)\in\mathscr{A}} x_{si} \qquad \text{max total flow from source} \tag{3.1}$$

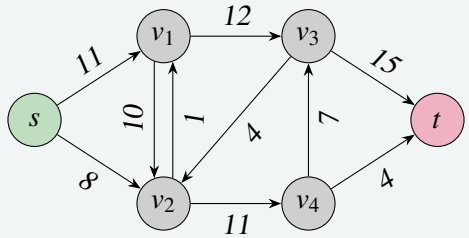
$$s.t. \sum_{i:(i,v)\in\mathscr{A}} x_{iv} - \sum_{j:(v,j)\in\mathscr{A}} x_{vj} = 0 \qquad v \in V\backslash\{s,t\} \tag{3.2}$$

$$0 \le x_{ij} \le u_{ij} \quad \forall (i,j) \in \mathscr{A} \tag{3.3}$$

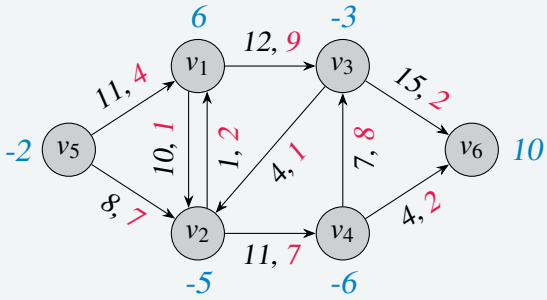
$$
\begin{aligned}
&\text{minimize} && \sum_{u\to v} \ell_{u\to v}\cdot x_{u\to v} \\
&\text{subject to } \sum_u x_{u\to s} - \sum_w x_{s\to w} && = 1 \\
&\sum_u x_{u\to t} - \sum_w x_{t\to w} = -1 && \\
&\sum_u x_{u\to v} - \sum_w x_{v\to w} = 0 && \text{for every vertex } v \neq s,t \\
&x_{u\to v} \geq 0 && \text{for every edge } u \to v
\end{aligned}
$$

SHORTEST PATH PROBLEM

Example 3.11: Max flow example



Example 3.12: Min Cost Network Flow



3.2.7.3. Minimum Cost Network Flow

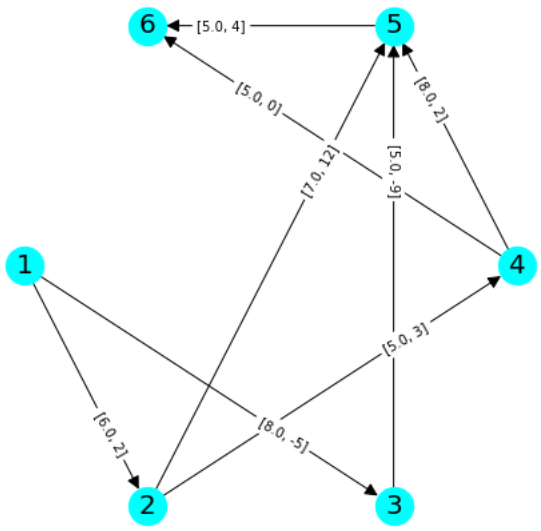
PARAMETERS We assume that flow on arc (i, j) should be non-negative and should not exceed

$$u_{ij} := \text{ the flow upper bound on arc}(i, j),$$

for $(i, j) \in \mathscr{A}$. Associated with each arc (i, j) is a cost

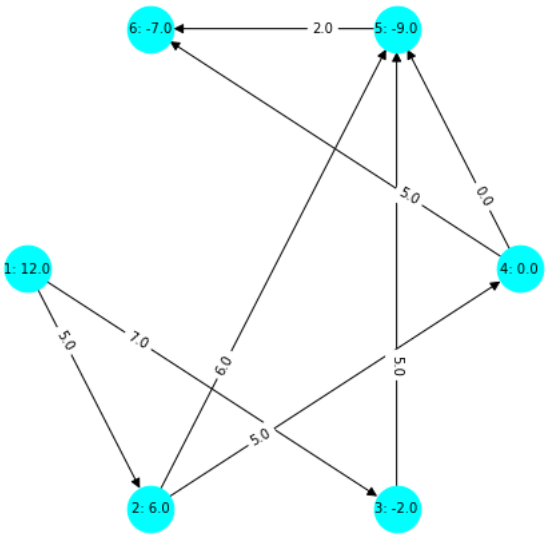
$$c_{ij} := \text{ cost per-unit-flow on arc } (i, j),$$

[1] **network-flow**, from **network-flow**. **network-flow**, **network-flow**.
[2] **network-flow-solution**, from **network-flow-solution**. **network-flow-solution**, **network-flow-solution**.



© network-flow[1]

Figure 3.1: network-flow



© network-flow-solution[2]

Figure 3.2: network-flow-solution

for $(i, j) \in \mathscr{A}$. The (total) cost of the flow x is defined to be

$$\sum_{(i,j)\in\mathscr{A}} c_{ij}x_{ij}.$$

We assume that we have further data for the nodes. Namely,

$$b_v := \text{ the net supply at node } v,$$

for $v \in V$.

A flow is conservative if the net flow out of node v, minus the net flow into node v, is equal to the net supply at node v, for all nodes $v \in V$.

The (single-commodity min-cost) network-flow problem is to find a minimumcost conservative flow that is non-negative and respects the flow upper bounds on the arcs.

OBJECTIVE AND CONSTRAINTS We can formulate this as follows:

$$\begin{aligned}
&\min \sum_{(i,j)\in\mathscr{A}} c_{ij}x_{ij} && \text{minimize cost}\\
&\sum_{(i,v)\in\mathscr{A}} x_{iv} - \sum_{(v,i)\in\mathscr{A}} x_{vi} = b_v, \quad \text{for all } v \in V, && \text{flow conservation}\\
&0 \le x_{ij} \le u_{ij}, \quad \text{for all } (i,j) \in \mathscr{A}.
\end{aligned}$$

Theorem 3.13: Integrality of Network Flow

If the capacities and demands are all integer values, then there always exists an optimal solution to the LP that has integer values.

3.2.8. Multi-Commodity Network Flow

In the same vein as the Network Flow Problem

$$\begin{aligned}
&\min \sum_{k=1}^{K} \sum_{e\in\mathscr{A}} c_e^k x_e^k\\
&\sum_{e\in\mathscr{A}\,:\,t(e)=v} x_e^k - \sum_{e\in\mathscr{A}\,:\,h(e)=v} x_e^k = b_v^k, \ \text{ for } v \in \mathscr{N},\ k = 1,2,\ldots,K;\\
&\sum_{k=0}^{K} x_e^k \le u_e, \ \text{ for } e \in \mathscr{A};\\
&x_e^k \ge 0, \ \text{ for } e \in \mathscr{A},\ k = 1,2,\ldots,K
\end{aligned}$$

Notes:

K=1 is ordinary single-commodity network flow. Integer solutions for free when node-supplies and arc capacities are integer. K=2 example below with integer data gives a fractional basic optimum. This example doesn't have any feasible integer flow at all.

Remark 3.14

Unfortunately, the same integrality theorem does not hold in the multi-commodity network flow probem. Nontheless, if the quanties in each flow are very large, then the LP solution will likely be very close to an integer valued solution.

3.3 Modeling Tricks

3.3.1. Maximizing a minimum

When the constraints could be general, we will write $x \in X$ to define general constraints. For instance, we could have $X = \{x \in \mathbb{R}^n : Ax \leq b\}$ of $X = \{x \in \mathbb{R}^n : Ax \leq b, x \in \mathbb{Z}^n\}$ or many other possibilities.

Consider the problem

$$\begin{aligned} \max \quad & \min\{x_1, \ldots, x_n\} \\ \text{such that} \quad & x \in X \end{aligned}$$

Having the minimum on the inside is inconvenient. To remove this, we just define a new variable y and enforce that $y \leq x_i$ and then we maximize y. Since we are maximizing y, it will take the value of the smallest x_i. Thus, we can recast the problem as

$$\begin{aligned} \max \quad & y \\ \text{such that} \quad & y \leq x_i \text{ for } i = 1, \ldots, n \\ & x \in X \end{aligned}$$

Example 3.15: Minimizing an Absolute Value

Note that

$$|t| = \max(t, -t),$$

Thus, if we need to minimize $|t|$ we can instead write

$$\min z \tag{3.1}$$
$$s.t. \tag{3.2}$$
$$t \leq z - t \leq z \tag{3.3}$$

3.4 Other examples

Food manfacturing - GUROBI

Optimization Methods in Finance - Corneujols, Tütüncü

4. Graphically Solving Linear Programs

Outcomes

A. *Learn how to plot the feasible region and the objective function.*
B. *Identify and compute extreme points of the feasible region.*
C. *Find the optimal solution(s) to a linear program graphically.*
D. *Classify the type of result of the problem as infeasible, unbounded, unique optimal solution, or infinitely many optimal solutions.*

Linear Programs (LP's) with two variables can be solved graphically by plotting the feasible region along with the level curves of the objective function.[1] We will show that we can find a point in the feasible region that maximizes the objective function using the level curves of the objective function.

We will begin with an easy example that is bounded and investigate the structure of the feasible region. We will then explore other examples.

4.1 Nonempty and Bounded Problem

Consider the problem

$$\begin{aligned} \max \quad & 2X + 5Y \\ \text{s.t.} \quad & X + 2Y \leq 16 \\ & 5X + 3Y \leq 45 \\ & X, Y \geq 0 \end{aligned}$$

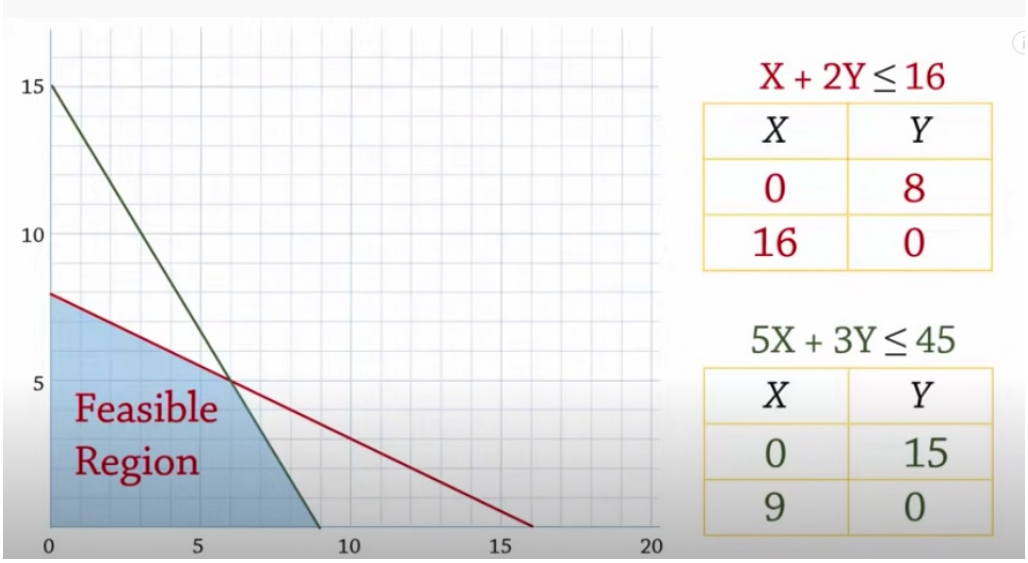
We want to start by plotting the *feasible region*, that is, the set points (X,Y) that satisfy all the constraints. We can plot this by first plotting the four lines

- $X + 2Y = 16$
- $5X + 3Y = 45$
- $X = 0$
- $Y = 0$

and then shading in the side of the space cut out by the corresponding inequality.

[1] Special thanks to Joshua Emmanual and Christopher Griffin for sharing their content to help put this section together. Proper citations and referenes are forthcoming.

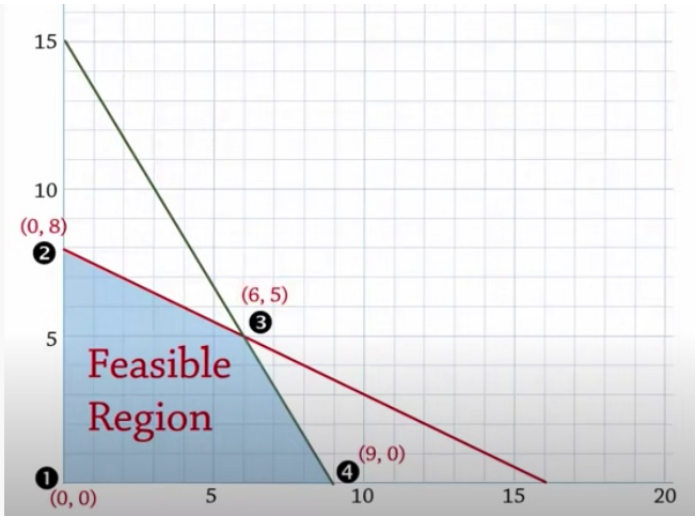
The resulting feasible region can then can be shaded in as the region that satisfies all the inequalties.



Notice that the feasible region is nonempty (it has points that satisfy all the inequalities) and also that it is bounded (the feasible points dont continue infinitly in any direction).

We want to identify the *extreme points* (i.e., the corners) of the feasible region. Understanding these points will be critical to understanding the optimal solutions of the model. Notice that all extreme points can be computed by finding the intersection of 2 of the lines. But! Not all intersections of any two lines are feasible.

We will later use the terminology *basic feasible solution* for an extreme point of the feasible region, and *basic solution* as a point that is the intersection of 2 lines, but is actually infeasible (does not satisfy all the constraints).



Point	$Z = 2X + 5Y$
❶ (0, 0)	0
❷ (0, 8)	40
❸ (6, 5)	37
❹ (9, 0)	18

Theorem 4.1: Optimal Extreme Point

If the feasible region is nonempty and bounded, then there exists an optimal solution at an extreme point of the feasible region.

We will explore why this theorem is true, and also what happens when the feasible region does not satisfy the assumtions of either nonempty or bounded. We illustrate the idea first using the problem from Example 1.

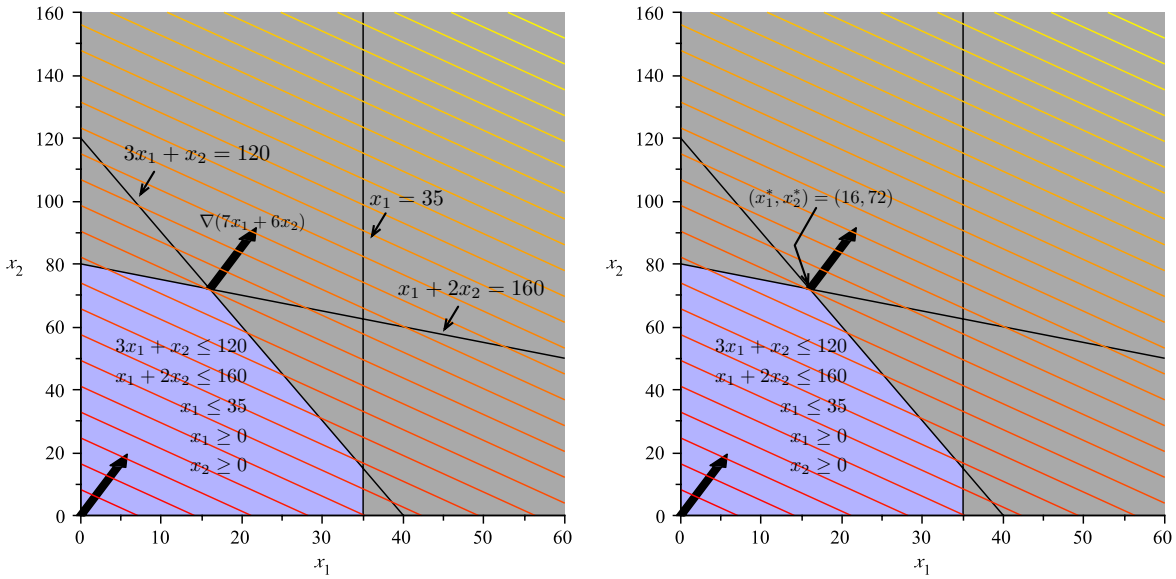


Figure 4.1: Feasible Region and Level Curves of the Objective Function: The shaded region in the plot is the feasible region and represents the intersection of the five inequalities constraining the values of x_1 and x_2. On the right, we see the optimal solution is the "last" point in the feasible region that intersects a level set as we move in the direction of increasing profit.

Example 4.2: Continuation of Example 1

Let's continue the example of the Toy Maker begin in Example 1. Solve this problem graphically.

Solution. To solve the linear programming problem graphically, begin by drawing the feasible region. This is shown in the blue shaded region of Figure 4.1.

After plotting the feasible region, the next step is to plot the level curves of the objective function. In our problem, the level sets will have the form:

$$7x_1 + 6x_2 = c \implies x_2 = \frac{-7}{6}x_1 + \frac{c}{6}$$

This is a set of parallel lines with slope $-7/6$ and intercept $c/6$ where c can be varied as needed. The level curves for various values of c are parallel lines. In Figure 4.1 they are shown in colors ranging from red to yellow depending upon the value of c. Larger values of c are more yellow.

To solve the linear programming problem, follow the level sets along the gradient (shown as the black arrow) until the last level set (line) intersects the feasible region. If you are doing this by hand, you can draw a single line of the form $7x_1 + 6x_2 = c$ and then simply draw parallel lines in the direction of the gradient $(7,6)$. At some point, these lines will fail to intersect the feasible region. The last line to intersect the feasible region will do so at a point that maximizes the profit. In this case, the point that maximizes $z(x_1,x_2) = 7x_1 + 6x_2$, subject to the constraints given, is $(x_1^*,x_2^*) = (16,72)$.

Note the point of optimality $(x_1^*, x_2^*) = (16, 72)$ is at a corner of the feasible region. This corner is formed by the intersection of the two lines: $3x_1 + x_2 = 120$ and $x_1 + 2x_2 = 160$. In this case, the constraints

$$3x_1 + x_2 \leq 120$$
$$x_1 + 2x_2 \leq 160$$

are both *binding*, while the other constraints are non-binding. In general, we will see that when an optimal solution to a linear programming problem exists, it will always be at the intersection of several binding constraints; that is, it will occur at a corner of a higher-dimensional polyhedron. ♠

We can now define an algorithm for identifying the solution to a linear programing problem in two variables with a *bounded* feasible region (see Algorithm 1):

Algorithm 1 Algorithm for Solving a Two Variable Linear Programming Problem Graphically–Bounded Feasible Region, Unique Solution Case

Algorithm for Solving a Linear Programming Problem Graphically

Bounded Feasible Region, Unique Solution

1. Plot the feasible region defined by the constraints.
2. Plot the level sets of the objective function.
3. For a maximization problem, identify the level set corresponding the greatest (least, for minimization) objective function value that intersects the feasible region. This point will be at a corner.
4. The point on the corner intersecting the greatest (least) level set is a solution to the linear programming problem.

The example linear programming problem presented in the previous section has a single optimal solution. In general, the following outcomes can occur in solving a linear programming problem:

1. The linear programming problem has a unique solution. (We've already seen this.)
2. There are infinitely many alternative optimal solutions.
3. There is no solution and the problem's objective function can grow to positive infinity for maximization problems (or negative infinity for minimization problems).
4. There is no solution to the problem at all.

Case 3 above can only occur when the feasible region is unbounded; that is, it cannot be surrounded by a ball with finite radius. We will illustrate each of these possible outcomes in the next four sections. We will prove that this is true in a later chapter.

4.2 Infinitely Many Optimal Solutions

It can happen that there is more than one solution. In fact, in this case, there are infinitly many optimal solutions. We'll study a specific linear programming problem with an infinite number of solutions by modifying the objective function in Example 1.

Example 4.3: Toy Maker Alternative Solutions

Suppose the toy maker in Example 1 finds that it can sell planes for a profit of \$18 *each instead of* \$7 *each. The new linear programming problem becomes:*

$$\begin{cases} \max\ z(x_1,x_2) = 18x_1 + 6x_2 \\ \quad s.t.\ 3x_1 + x_2 \leq 120 \\ \qquad x_1 + 2x_2 \leq 160 \\ \qquad x_1 \leq 35 \\ \qquad x_1 \geq 0 \\ \qquad x_2 \geq 0 \end{cases} \tag{4.1}$$

Solution. Applying our graphical method for finding optimal solutions to linear programming problems yields the plot shown in Figure 4.2. The level curves for the function $z(x_1,x_2) = 18x_1 + 6x_2$ are *parallel* to one face of the polygon boundary of the feasible region. Hence, as we move further up and to the right in the direction of the gradient (corresponding to larger and larger values of $z(x_1,x_2)$) we see that there is not *one* point on the boundary of the feasible region that intersects that level set with greatest value, but instead a side of the polygon boundary described by the line $3x_1 + x_2 = 120$ where $x_1 \in [16,35]$. Let:

$$S = \{(x_1,x_2) | 3x_1 + x_2 \leq 120, x_1 + 2x_2 \leq 160, x_1 \leq 35, x_1, x_2 \geq 0\}$$

that is, S is the feasible region of the problem. Then for any value of $x_1^* \in [16,35]$ and any value x_2^* so that $3x_1^* + x_2^* = 120$, we will have $z(x_1^*,x_2^*) \geq z(x_1,x_2)$ for all $(x_1,x_2) \in S$. Since there are infinitely many values that x_1 and x_2 may take on, we see this problem has an infinite number of alternative optimal solutions.

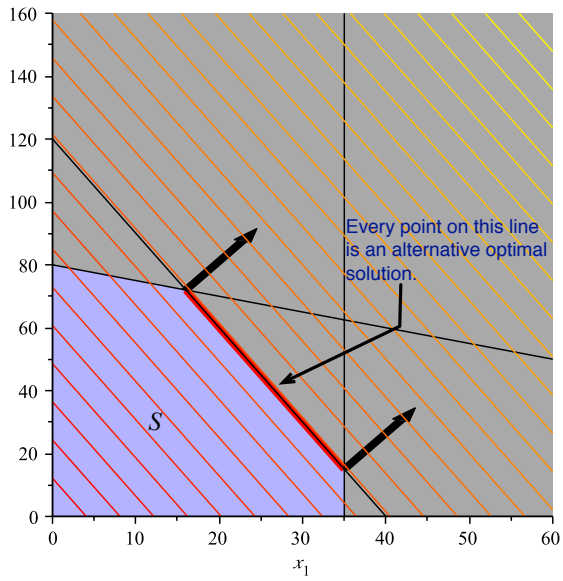


Figure 4.2: An example of infinitely many alternative optimal solutions in a linear programming problem. The level curves for $z(x_1,x_2) = 18x_1 + 6x_2$ are *parallel* to one face of the polygon boundary of the feasible region. Moreover, this side contains the points of greatest value for $z(x_1,x_2)$ inside the feasible region. Any combination of (x_1,x_2) on the line $3x_1 + x_2 = 120$ for $x_1 \in [16,35]$ will provide the largest possible value $z(x_1,x_2)$ can take in the feasible region S.

♠

Exercise 4.4

Use the graphical method for solving linear programming problems to solve the linear programming problem you defined in Exercise 3.

Based on the example in this section, we can modify our algorithm for finding the solution to a linear programming problem graphically to deal with situations with an infinite set of alternative optimal solutions (see Algorithm 2):

Algorithm 2 Algorithm for Solving a Two Variable Linear Programming Problem Graphically–Bounded Feasible Region Case

Algorithm for Solving a Linear Programming Problem Graphically

Bounded Feasible Region

1. Plot the feasible region defined by the constraints.
2. Plot the level sets of the objective function.
3. For a maximization problem, identify the level set corresponding the greatest (least, for minimization) objective function value that intersects the feasible region. This point will be at a corner.
4. The point on the corner intersecting the greatest (least) level set is a solution to the linear programming problem.
5. **If the level set corresponding to the greatest (least) objective function value is parallel to a side of the polygon boundary next to the corner identified, then there are infinitely many alternative optimal solutions and any point on this side may be chosen as an optimal solution.**

Exercise 4.5

Modify the linear programming problem from Exercise 3 to obtain a linear programming problem with an infinite number of alternative optimal solutions. Solve the new problem and obtain a description for the set of alternative optimal solutions. [Hint: Just as in the example, x_1 will be bound between two value corresponding to a side of the polygon. Find those values and the constraint that is binding. This will provide you with a description of the form for any $x_1^* \in [a,b]$ and x_2^* is chosen so that $cx_1^* + dx_2^* = v$, the point (x_1^*, x_2^*) is an alternative optimal solution to the problem. Now you fill in values for a, b, c, d and v.]

4.3 Problems with No Solution

Recall for *any* mathematical programming problem, the feasible set or region is simply a subset of $\mathbb{R}^n$. If this region is empty, then there is no solution to the mathematical programming problem and the problem is said to be *over constrained*. In this case, we say that the problem is *infeasible*. We illustrate this case for linear programming problems with the following example.

Example 4.6: Infeasible Problem

onsider the following linear programming problem:

$$\begin{cases} \max\ z(x_1,x_2) = 3x_1 + 2x_2 \\ s.t.\ \ \dfrac{1}{40}x_1 + \dfrac{1}{60}x_2 \leq 1 \\ \qquad \dfrac{1}{50}x_1 + \dfrac{1}{50}x_2 \leq 1 \\ \qquad x_1 \geq 30 \\ \qquad x_2 \geq 20 \end{cases} \tag{4.1}$$

Solution. The level sets of the objective and the constraints are shown in Figure 4.3.

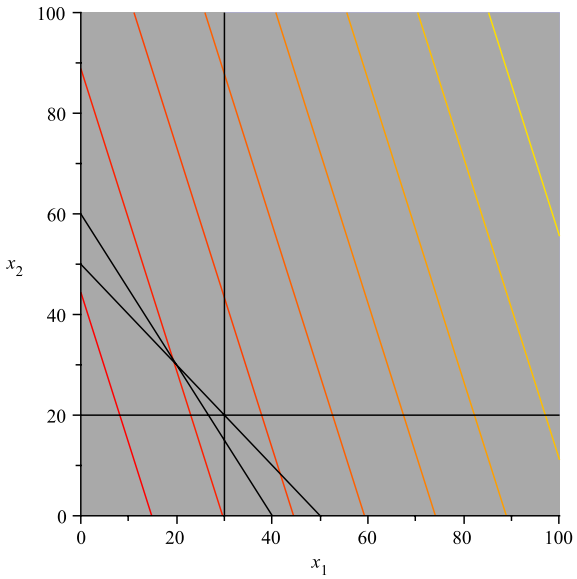


Figure 4.3: A Linear Programming Problem with no solution. The feasible region of the linear programming problem is empty; that is, there are no values for x_1 and x_2 that can simultaneously satisfy all the constraints. Thus, no solution exists.

The fact that the feasible region is empty is shown by the fact that in Figure 4.3 there is no blue region–i.e., all the regions are gray indicating that the constraints are not satisfiable. ♠

Based on this example, we can modify our previous algorithm for finding the solution to linear programming problems graphically (see Algorithm 3):

Algorithm 3 Algorithm for Solving a Two Variable Linear Programming Problem Graphically–Bounded Feasible Region Case

Algorithm for Solving a Linear Programming Problem Graphically

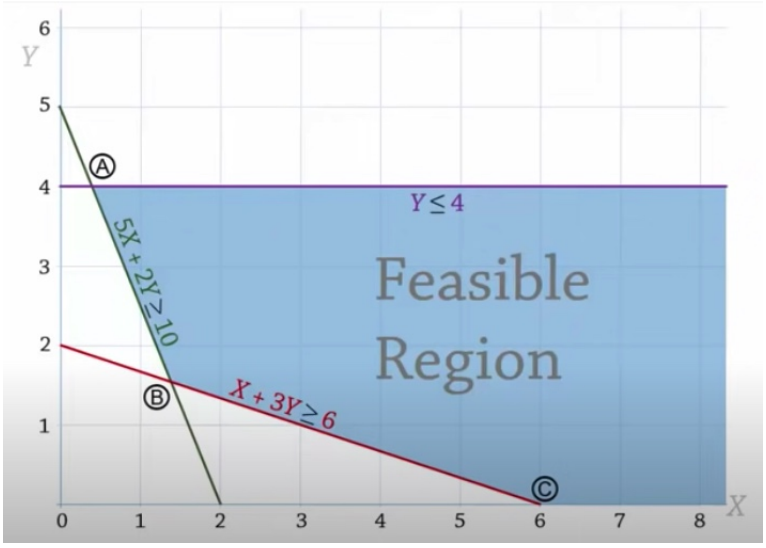
Bounded Feasible Region

1. Plot the feasible region defined by the constraints.
2. **If the feasible region is empty, then no solution exists.**
3. Plot the level sets of the objective function.
4. For a maximization problem, identify the level set corresponding the greatest (least, for minimization) objective function value that intersects the feasible region. This point will be at a corner.
5. The point on the corner intersecting the greatest (least) level set is a solution to the linear programming problem.
6. **If the level set corresponding to the greatest (least) objective function value is parallel to a side of the polygon boundary next to the corner identified, then there are infinitely many alternative optimal solutions and any point on this side may be chosen as an optimal solution.**

4.4 Problems with Unbounded Feasible Regions

Consider the problem

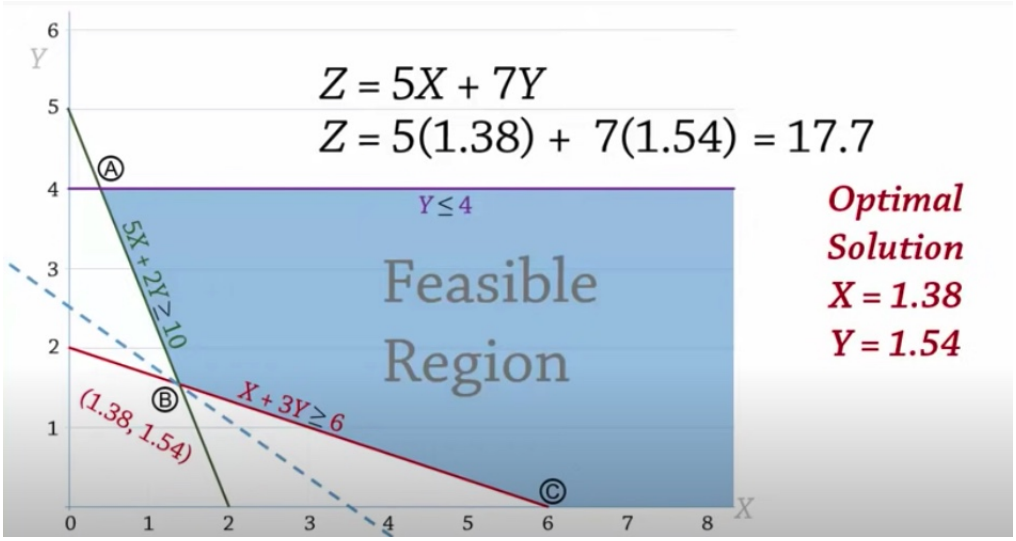
$$\begin{aligned} \min \quad & Z = 5X + 7Y \\ \text{s.t.} \quad & X + 3Y \geq 6 \\ & 5X + 2Y \geq 10 \\ & Y \leq 4 \\ & X, Y \geq 0 \end{aligned}$$



As you can see, the feasible region is *unbounded*. In particular, from any point in the feasible region, one can always find another feasible point by increasing the X coordinate (i.e., move to the right in the picture). However, this does not necessarily mean that the optimization problem is unbounded.

Indeed, the optimal solution is at the B, the extreme point in the lower left hand corner.

To do: add contours to plot to show extreme point is the optimal solution.



Consider however, if we consider a different problem where we try to maximize the objective

$$\begin{aligned} \max \quad & Z = 5X + 7Y \\ \text{s.t.} \quad & X + 3Y \geq 6 \\ & 5X + 2Y \geq 10 \\ & Y \leq 4 \\ & X, Y \geq 0 \end{aligned}$$

Solution. This optimization problem is unbounded! For example, notice that the point $(X,Y) = (n,0)$ is feasible for all $n = 1,2,3,\ldots,$. Then the objective function $Z = 5n + 0$ follows the sequence $5,10,15,\ldots,$, which diverges to infinity. ♠

Again, we'll tackle the issue of linear programming problems with unbounded feasible regions by illustrating the possible outcomes using examples.

Example 4.7

Consider the linear programming problem below:

$$\begin{cases} \max \; z(x_1,x_2) = 2x_1 - x_2 \\ \quad s.t. \; x_1 - x_2 \leq 1 \\ \qquad 2x_1 + x_2 \geq 6 \\ \qquad x_1, x_2 \geq 0 \end{cases} \tag{4.1}$$

Solution. The feasible region and level curves of the objective function are shown in Figure 4.4.

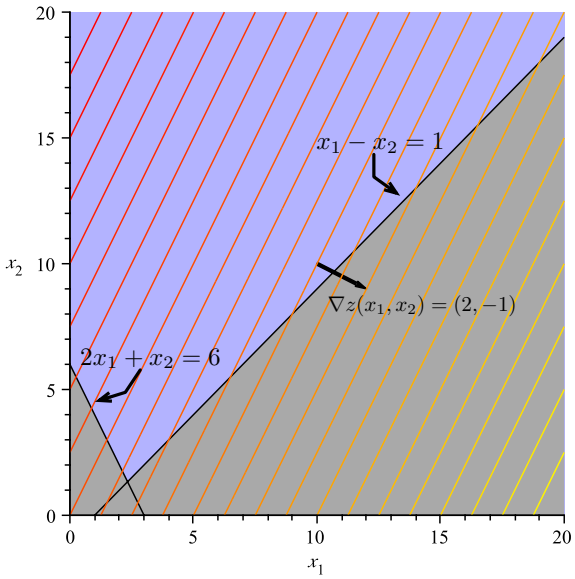


Figure 4.4: A Linear Programming Problem with Unbounded Feasible Region: Note that we can continue to make level curves of $z(x_1,x_2)$ corresponding to larger and larger values as we move down and to the right. These curves will continue to intersect the feasible region for any value of $v = z(x_1,x_2)$ we choose. Thus, we can make $z(x_1,x_2)$ as large as we want and still find a point in the feasible region that will provide this value. Hence, the optimal value of $z(x_1,x_2)$ subject to the constraints $+\infty$. That is, the problem is unbounded.

The feasible region in Figure 4.4 is clearly unbounded since it stretches upward along the x_2 axis infinitely far and also stretches rightward along the x_1 axis infinitely far, bounded below by the line $x_1 - x_2 = 1$. There is no way to enclose this region by a disk of finite radius, hence the feasible region is not bounded.

We can draw more level curves of $z(x_1,x_2)$ in the direction of increase (down and to the right) as long as we wish. There will always be an intersection point with the feasible region because it is infinite. That is, these curves will continue to intersect the feasible region for any value of $v = z(x_1,x_2)$ we choose. Thus, we can make $z(x_1,x_2)$ as large as we want and still find a point in the feasible region that will provide this value. Hence, the largest value $z(x_1,x_2)$ can take when (x_1,x_2) are in the feasible region is $+\infty$. That is, the problem is unbounded. ♠

Just because a linear programming problem has an unbounded feasible region does not imply that there is not a finite solution. We illustrate this case by modifying example 4.4.

Example 4.8: Continuation of Example

Consider the linear programming problem from Example 4.4 with the new objective function:

$z(x_1,x_2) = (1/2)x_1 - x_2$. *Then we have the new problem:*

$$\begin{cases} \max\ z(x_1,x_2) = \dfrac{1}{2}x_1 - x_2 \\ s.t.\ \ x_1 - x_2 \leq 1 \\ \qquad 2x_1 + x_2 \geq 6 \\ \qquad x_1, x_2 \geq 0 \end{cases} \tag{4.2}$$

Solution. The feasible region, level sets of $z(x_1,x_2)$ and gradients are shown in Figure 4.5. In this case note, that the direction of increase of the objective function is *away* from the direction in which the feasible region is unbounded (i.e., downward). As a result, the point in the feasible region with the largest $z(x_1,x_2)$ value is $(7/3,4/3)$. Again this is a vertex: the binding constraints are $x_1 - x_2 = 1$ and $2x_1 + x_2 = 6$ and the solution occurs at the point these two lines intersect.

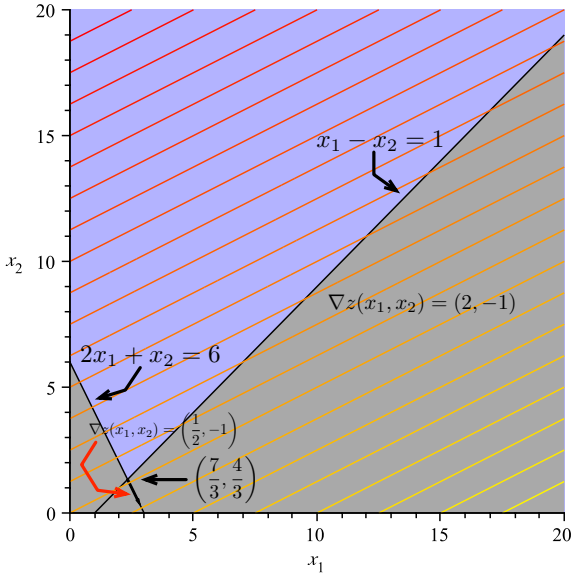


Figure 4.5: A Linear Programming Problem with Unbounded Feasible Region and Finite Solution: In this problem, the level curves of $z(x_1,x_2)$ increase in a more "southernly" direction that in Example 4.4–that is, *away* from the direction in which the feasible region increases without bound. The point in the feasible region with largest $z(x_1,x_2)$ value is $(7/3,4/3)$. Note again, this is a vertex.

♠

Based on these two examples, we can modify our algorithm for graphically solving a two variable linear programming problems to deal with the case when the feasible region is unbounded.

Algorithm 4 Algorithm for Solving a Linear Programming Problem Graphically–Bounded and Unbounded Case

Algorithm for Solving a Two Variable Linear Programming Problem Graphically

1. Plot the feasible region defined by the constraints.
2. If the feasible region is empty, then no solution exists.
3. If the feasible region is unbounded goto Line 8. Otherwise, Goto Line 4.
4. Plot the level sets of the objective function.
5. For a maximization problem, identify the level set corresponding the greatest (least, for minimization) objective function value that intersects the feasible region. This point will be at a corner.
6. The point on the corner intersecting the greatest (least) level set is a solution to the linear programming problem.
7. **If the level set corresponding to the greatest (least) objective function value is parallel to a side of the polygon boundary next to the corner identified, then there are infinitely many alternative optimal solutions and any point on this side may be chosen as an optimal solution.**
8. (The feasible region is unbounded): Plot the level sets of the objective function.
9. If the level sets intersect the feasible region at larger and larger (smaller and smaller for a minimization problem), then the problem is unbounded and the solution is $+\infty$ ($-\infty$ for minimization problems).
10. Otherwise, identify the level set corresponding the greatest (least, for minimization) objective function value that intersects the feasible region. This point will be at a corner.
11. The point on the corner intersecting the greatest (least) level set is a solution to the linear programming problem. **If the level set corresponding to the greatest (least) objective function value is parallel to a side of the polygon boundary next to the corner identified, then there are infinitely many alternative optimal solutions and any point on this side may be chosen as an optimal solution.**

Exercise 4.9

Does the following problem have a bounded solution? Why?

$$\begin{cases} \min\ z(x_1,x_2) = 2x_1 - x_2 \\ s.t.\ x_1 - x_2 \leq 1 \\ \quad 2x_1 + x_2 \geq 6 \\ \quad x_1, x_2 \geq 0 \end{cases} \tag{4.3}$$

[Hint: Use Figure 4.5 and Algorithm 4.]

Exercise 4.10

Modify the objective function in Example 4.4 or Example 4.4 to produce a problem with an infinite number of solutions.

Exercise 4.11

Modify the objective function in Exercise 4.4 to produce a ***minimization*** *problem that has a finite solution. Draw the feasible region and level curves of the objective to "prove" your example works.* [Hint: Think about what direction of increase is required for the level sets of $z(x_1,x_2)$ (or find a trick using Exercise **??**).]

4.5 Formal Mathematical Statements

Vectors and Linear and Convex Combinations

Vectors: Vector **n** has n-elements and represents a point (or an arrow from the origin to the point, denoting a direction) in $\mathscr{R}^n$ space (Euclidean or real space). Vectors can be expressed as either row or column vectors.

Vector Addition: Two vectors of the same size can be added, componentwise, e.g., for vectors $\mathbf{a} = (2,3)$ and $\mathbf{b} = (3,2)$, $\mathbf{a}+\mathbf{b} = (2+3,3+2) = (5,5)$.

Scalar Multiplication: A vector can be multiplied by a scalar k (constant) component-wise. If $k > 0$ then this does not change the direction represented by the vector, it just scales the vector.

Inner or Dot Product: Two vectors of the same size can be multiplied to produce a real number. For example, $\mathbf{ab} = 2*3+3*2 = 10$.

Linear Combination: The vector $\mathbf{b}$ is a **linear combination** of $\mathbf{a}_1,\mathbf{a}_2,\cdots,\mathbf{a}_k$ if $\mathbf{b} = \sum_{i=1}^{k} \lambda_i \mathbf{a}_i$ for $\lambda_1,\lambda_2,\cdots,\lambda_k \in \mathscr{R}$. If $\lambda_1,\lambda_2,\cdots,\lambda_k \in \mathscr{R}_{\geq 0}$ then $\mathbf{b}$ is a *non-negative linear combination* of $\mathbf{a}_1,\mathbf{a}_2,\cdots,\mathbf{a}_k$.

Convex Combination: The vector $\mathbf{b}$ is a **convex combination** of $\mathbf{a}_1,\mathbf{a}_2,\cdots,\mathbf{a}_k$ if $\mathbf{b} = \sum_{i=1}^{k} \lambda_i \mathbf{a}_i$, for $\lambda_1,\lambda_2,\cdots,\lambda_k \in \mathscr{R}_{\geq 0}$ and $\sum_{i=1}^{k} \lambda_i = 1$. For example, any convex combination of two points will lie on the line segment between the points.

Linear Independence: Vectors $\mathbf{a}_1,\mathbf{a}_2,\cdots,\mathbf{a}_k$ are *linearly independent* if the following linear combination $\sum_{i=1}^{k} \lambda_i \mathbf{a}_i = 0$ implies that $\lambda_i = 0$, $i = 1,2,\cdots,k$. In $\mathscr{R}^2$ two vectors are only linearly dependent if they lie on the same line. Can you have three linearly independent vectors in $\mathscr{R}^2$?

Spanning Set: Vectors $\mathbf{a}_1,\mathbf{a}_2,\cdots,\mathbf{a}_k$ span $\mathscr{R}^m$ is any vector in $\mathscr{R}^m$ can be represented as a linear combination of $\mathbf{a}_1,\mathbf{a}_2,\cdots,\mathbf{a}_k$, i.e., $\sum_{i=1}^{m} \lambda_i \mathbf{a_i}$ can represent any vector in $\mathscr{R}^m$.

Basis: Vectors $\mathbf{a}_1,\mathbf{a}_2,\cdots,\mathbf{a}_k$ form a basis of $\mathscr{R}^m$ if they span $\mathscr{R}^m$ and any smaller subset of these vectors does not span $\mathscr{R}^m$. Vectors $\mathbf{a}_1,\mathbf{a}_2,\cdots,\mathbf{a}_k$ can only form a basis of $\mathscr{R}^m$ if $k = m$ and they are linearly independent.

Convex and Polyhedral Sets

Convex Set: Set $\mathscr{S}$ in $\mathscr{R}^n$ is a *convex set* if a line segment joining any pair of points $\mathbf{a}_1$ and $\mathbf{a}_2$ in $\mathscr{S}$ is completely contained in $\int$, that is, $\lambda\mathbf{a}_1 + (1-\lambda)\mathbf{a}_2 \in \mathscr{S}, \forall\lambda \in [0,1]$.

Hyperplanes and Half-Spaces: A hyperplane in $\mathscr{R}^n$ divides $\mathscr{R}^n$ into 2 half-spaces (like a line does in $\mathscr{R}^2$). A hyperplane is the set $\{\mathbf{x} : \mathbf{px} = k\}$, where $\mathbf{p}$ is the gradient to the hyperplane (i.e., the coefficients of our linear expression). The corresponding half-spaces is the set of points $\{\mathbf{x} : \mathbf{px} \geq k\}$ and $\{\mathbf{x} : \mathbf{px} \leq k\}$.

Polyhedral Set: A *polyhedral set* (or polyhedron) is the set of points in the intersection of a finite set of half-spaces. Set $\mathscr{S} = \{\mathbf{x} : \mathbf{Ax} \leq \mathbf{b}, \mathbf{x} \geq 0\}$, where $\mathbf{A}$ is an $m \times n$ matrix, $\mathbf{x}$ is an n-vector, and $\mathbf{b}$ is an m-vector, is a *polyhedral set* defined by $m+n$ hyperplanes (i.e., the intersection of $m+n$ half-spaces).

- Polyhedral sets are convex.
- A polytope is a bounded polyhedral set.
- A polyhedral cone is a polyhedral set where the hyperplanes (that define the half-spaces) pass through the origin, thus $\mathscr{C} = \{\mathbf{x} : \mathbf{Ax} \leq 0\}$ is a polyhedral cone.

Edges and Faces: An *edge* of a polyhedral set $\mathscr{S}$ is defined by $n-1$ hyperplanes, and a *face* of $\mathscr{S}$ by one of more defining hyperplanes of $\mathscr{S}$, thus an extreme point and an edge are faces (an extreme point is a zero-dimensional face and an edge a one-dimensional face). In $\mathscr{R}^2$ faces are only edges and extreme points, but in $\mathscr{R}^3$ there is a third type of face, and so on...

Extreme Points: $\mathbf{x} \in \mathscr{S}$ is an extreme point of $\mathscr{S}$ if:

Definition 1: $\mathbf{x}$ is not a convex combination of two other points in $\mathscr{S}$, that is, all line segments that are completely in $\mathscr{S}$ that contain $\mathbf{x}$ must have $\mathbf{x}$ as an endpoint.

Definition 2: $\mathbf{x}$ lies on n linearly independent defining hyperplanes of $\mathscr{S}$.

If more than n hyperplanes pass through an extreme points then it is a degenerate extreme point, and the polyhedral set is considered degenerate. This just adds a bit of complexity to the algorithms we will study, but it is quite common.

Unbounded Sets:

Rays: A ray in $\mathscr{R}^n$ is the set of points $\{\mathbf{x} : \mathbf{x}_0 + \lambda\mathbf{d},\ \lambda \geq 0\}$, where $\mathbf{x}_0$ is the vertex and $\mathbf{d}$ is the direction of the ray.

Convex Cone: A *Convex Cone* is a convex set that consists of rays emanating from the origin. A convex cone is completely specified by its extreme directions. If $\mathscr{C}$ is convex cone, then for any $\mathbf{x} \in \mathscr{C}$ we have $\lambda\mathbf{x} \in \mathscr{C},\ \lambda \geq 0$.

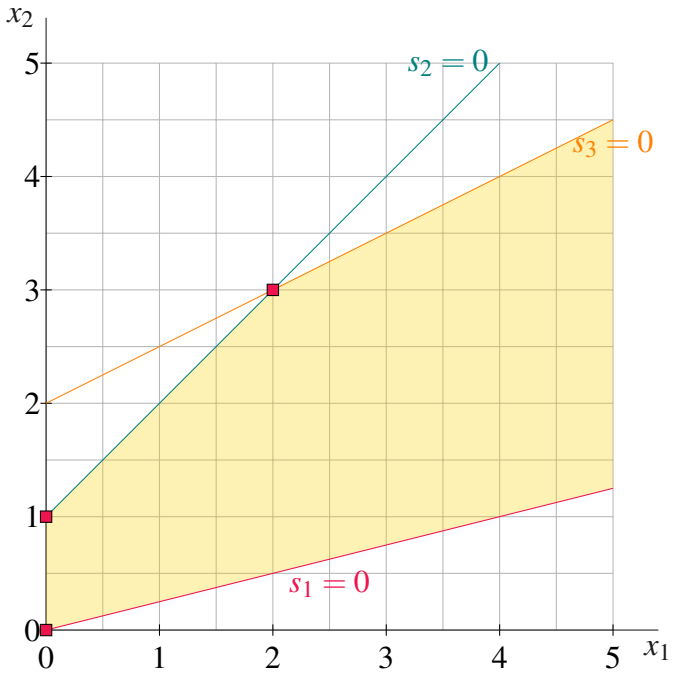
Unbounded Polyhedral Sets: If $\mathscr{S}$ is unbounded, it will have *directions*. $\mathbf{d}$ is a direction of $\mathscr{S}$ only if $\mathbf{Ax} + \lambda\mathbf{d} \leq \mathbf{b}, \mathbf{x} + \lambda\mathbf{d} \geq 0$ for all $\lambda \geq 0$ and all $\mathbf{x} \in \mathscr{S}$. In other words, consider the ray $\{\mathbf{x} : \mathbf{x}_0 + \lambda\mathbf{d},\ \lambda \geq 0\}$

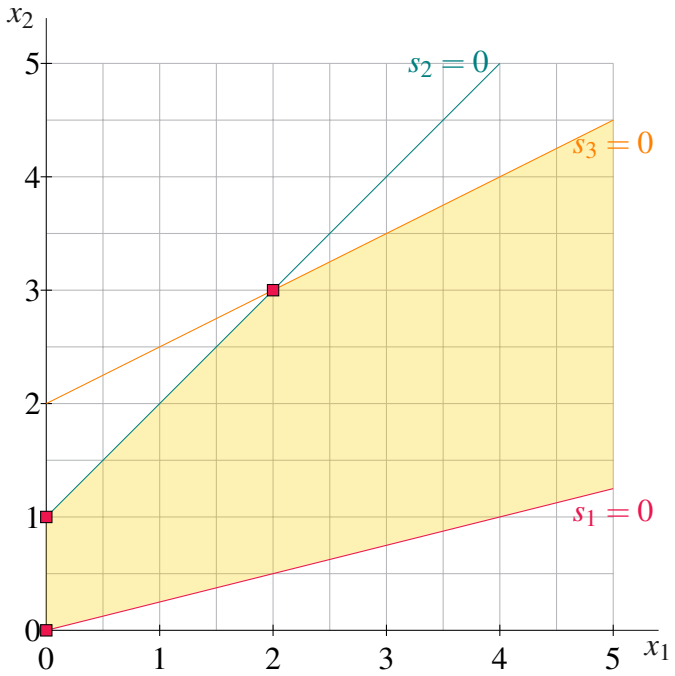
in $\mathscr{R}^n$, where $\mathbf{x}_0$ is the vertex and $\mathbf{d}$ is the direction of the ray. $\mathbf{d} \neq 0$ is a **direction** of set $\mathscr{S}$ if for each $\mathbf{x}_0$ in $\mathscr{S}$ the ray $\{\mathbf{x}_0 + \lambda\mathbf{d},\ \lambda \geq 0\}$ also belongs to $\mathscr{S}$.

Extreme Directions: An *extreme direction* of $\mathscr{S}$ is a direction that *cannot* be represented as positive linear combination of other directions of $\mathscr{S}$. A non-negative linear combination of extreme directions can be used to represent all other directions of $\mathscr{S}$. A polyhedral cone is completely specified by its extreme directions.

Let's define a procedure for finding the extreme directions, using the following LP's feasible region. Graphically, we can see that the extreme directions should follow the the $s_1 = 0$ (red) line and the $s_3 = 0$ (orange) line.

$$
\begin{aligned}
\max\ & z = -5x_1 - x_2 \\
\text{s.t.}\ & x_1 - 4x_2 + s_1 = 0 \\
& -x_1 + x_2 + s_2 = 1 \\
& -x_1 + 2x_2 + s_3 = 4 \\
& x_1, x_2, s_1, s_2, s_3 \geq 0.
\end{aligned}
$$



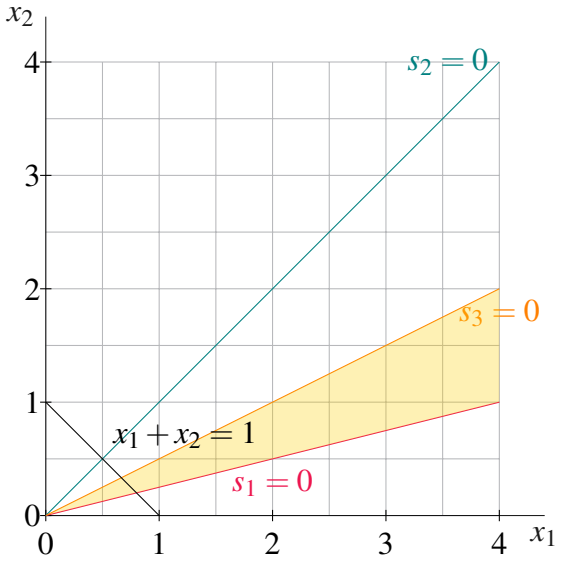


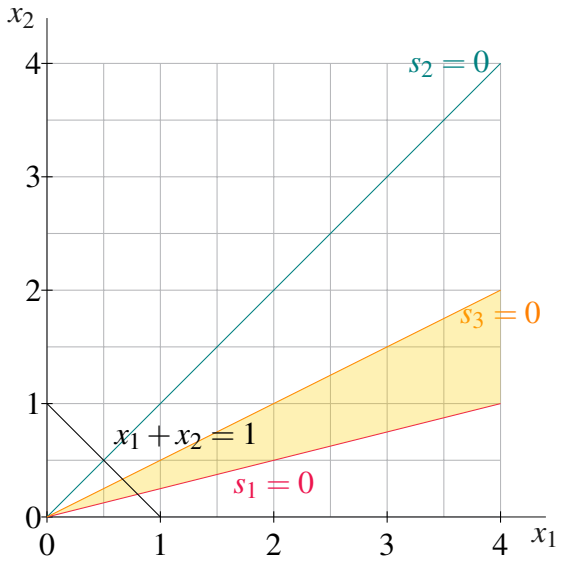
E.g., consider the $s_3 = 0$ (orange) line, to find the extreme direction start at extreme point (2,3) and find another feasible point on the orange line, say (4,4) and subtract (2,3) from (4,4), which yields (2,1).

This is related to the slope in two-dimensions, as discussed in class, the rise is 1 and the run is 2. So this direction has a slope of 1/2, but this does not carry over easily to higher dimensions where directions cannot be defined by a single number.

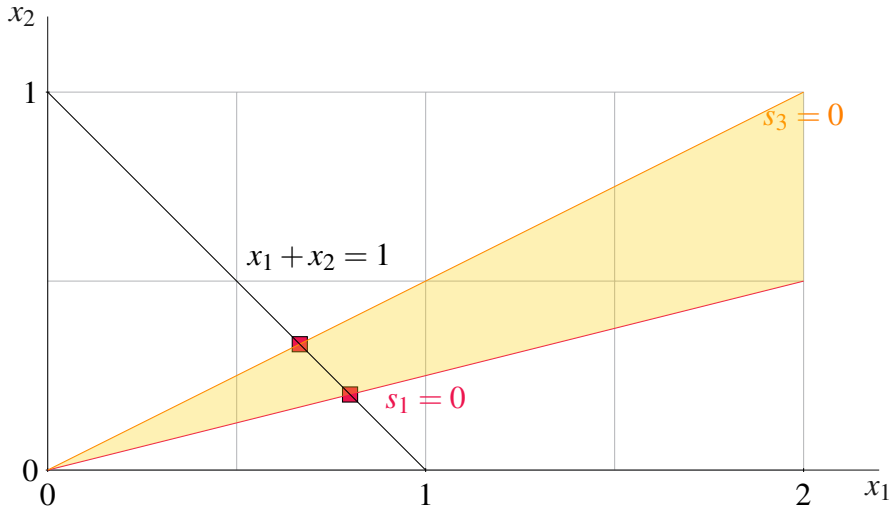
To find the extreme directions we can change the right-hand-side to $\mathbf{b} = 0$, which forms a polyhedral cone (in yellow), and then add the constraint $x_1 + x_2 = 1$. The intersection of the cone and $x_1 + x_2 = 1$ form a line segment.

$$
\begin{aligned}
\max \quad & z = -5x_1 - x_2 \\
\text{s.t.} \quad & x_1 - 4x_2 + s_1 = 0 \\
& -x_1 + x_2 + s_2 = 0 \\
& -x_1 + 2x_2 + s_3 = 0 \\
& x_1 + x_2 = 1 \\
& x_1, x_2, s_1, s_2, s_3 \geq 0.
\end{aligned}
$$





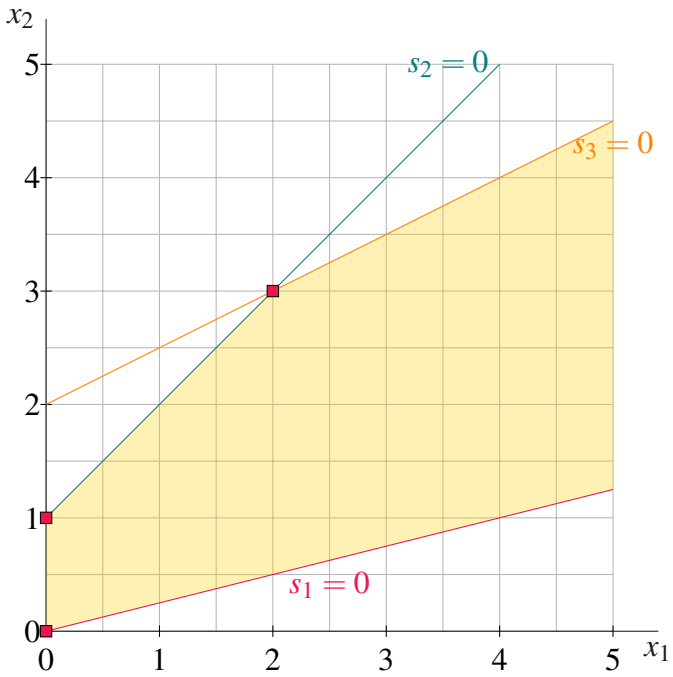
Magnifying for clarity, and removing the $s_2 = 0$ (teal) line, as it is redundant, and marking the extreme points of the new feasible region, (4/5, 1/5) and (2/3, 1/3), with red boxes, we have:



The extreme directions are thus (4/5, 1/5) and (2/3, 1/3).

Representation Theorem: Let $\mathbf{x}_1, \mathbf{x}_2, \cdots \mathbf{x_k}$ be the set of extreme points of $\mathscr{S}$, and if $\mathscr{S}$ is unbounded, $\mathbf{d}_1, \mathbf{d}_2, \cdots \mathbf{d_l}$ be the set of extreme directions. Then any $\mathbf{x} \in \mathscr{S}$ is equal to a convex combination of the extreme points and a non-negative linear combination of the extreme directions: $\mathbf{x} = \sum_{j=1}^{k} \lambda_j \mathbf{x_j} + \sum_{j=1}^{l} \mu_j \mathbf{d_j}$, where $\sum_{j=1}^{k} \lambda_j = 1$, $\lambda_j \geq 0$, $\forall j = 1, 2, \cdots, k$, and $\mu_j \geq 0$, $\forall j = 1, 2, \cdots, l$.

$$
\begin{aligned}
\max\ & z = -5x_1 - x_2 \\
\text{s.t.}\ & x_1 - 4x_2 + s_1 = 0 \\
& -x_1 + x_2 + s_2 = 1 \\
& -x_1 + 2x_2 + s_3 = 4 \\
& x_1, x_2, s_1, s_2, s_3 \geq 0.
\end{aligned}
$$



Represent point (1/2, 1) as a convex combination of the extreme points of the above LP. Find λs to solve the following system of equations:

$$
\lambda_1 \begin{bmatrix} 0 \\ 0 \end{bmatrix} + \lambda_2 \begin{bmatrix} 0 \\ 1 \end{bmatrix} + \lambda_3 \begin{bmatrix} 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 1/2 \\ 1 \end{bmatrix}
$$

5. Software - Excel

Resources

- *Excel Solver - Introduction on Youtube*
- *Some notes from MIT*

5.0.1. Excel Solver

5.0.2. Videos

Solving a linear program Optimal product mix Set Cover

Introduction to Designing Optimization Models Using Excel Solver

Traveling Salesman Problem

Also Travelin Salesman Problem

Multiple Traveling Salesman Problem

Shortest Path

5.0.3. Links

Loan Example

Several Examples including TSP

6. Software - Python

Outcomes

- *Install and get python up and running in some form*
- *Introduce basic python skills that will be helpful*

Resources

- *A Byte of Python*
- *Github - Byte of Python (CC-BY-SA)*

6.1 Installing and Managing Python

Installing and Managing Python

Lab Objective: *One of the great advantages of Python is its lack of overhead: it is relatively easy to download, install, start up, and execute. This appendix introduces tools for installing and updating specific packages and gives an overview of possible environments for working efficiently in Python.*

Installing Python via Anaconda

A *Python distribution* is a single download containing everything needed to install and run Python, together with some common packages. For this curriculum, we **strongly** recommend using the *Anaconda* distribution to install Python. Anaconda includes IPython, a few other tools for developing in Python, and a large selection of packages that are common in applied mathematics, numerical computing, and data science. Anaconda is free and available for Windows, Mac, and Linux.

Follow these steps to install Anaconda.

1. Go to `https://www.anaconda.com/download/`.
2. Download the **Python 3.6** graphical installer specific to your machine.
3. Open the downloaded file and proceed with the default configurations.

For help with installation, see `https://docs.anaconda.com/anaconda/install/`. This page contains links to detailed step-by-step installation instructions for each operating system, as well as information for updating and uninstalling Anaconda.

ACHTUNG!

This curriculum uses Python 3.6, **not** Python 2.7. With the wrong version of Python, some example code within the labs may not execute as intended or result in an error.

Managing Packages

A *Python package manager* is a tool for installing or updating Python packages, which involves downloading the right source code files, placing those files in the correct location on the machine, and linking the files to the Python interpreter. **Never** try to install a Python package without using a package manager (see `https://xkcd.com/349/`).

Conda

Many packages are not included in the default Anaconda download but can be installed via Anaconda's package manager, `conda`. See `https://docs.anaconda.com/anaconda/packages/pkg-docs` for the complete list of available packages. When you need to update or install a package, **always** try using `conda` first.

Command	Description
`conda install <package-name>`	Install the specified package.
`conda update <package-name>`	Update the specified package.
`conda update conda`	Update `conda` itself.
`conda update anaconda`	Update **all** packages included in Anaconda.
`conda --help`	Display the documentation for `conda`.

For example, the following terminal commands attempt to install and update `matplotlib`.

```
$ conda update conda                # Make sure that conda is up to date.
$ conda install matplotlib          # Attempt to install matplotlib.
$ conda update matplotlib           # Attempt to update matplotlib.
```

See `https://conda.io/docs/user-guide/tasks/manage-pkgs.html` for more examples.

NOTE

The best way to ensure a package has been installed correctly is to try importing it in IPython.

```
# Start IPython from the command line.
```

```
$ ipython
IPython 6.5.0 -- An enhanced Interactive Python. Type '?' for help.

# Try to import matplotlib.
In [1]: from matplotlib import pyplot as plt      # Success!
```

ACHTUNG!

Be careful not to attempt to update a Python package while it is in use. It is safest to update packages while the Python interpreter is not running.

Pip

The most generic Python package manager is called `pip`. While it has a larger package list, `conda` is the cleaner and safer option. Only use `pip` to manage packages that are not available through `conda`.

Command	Description
`pip install package-name`	Install the specified package.
`pip install --upgrade package-name`	Update the specified package.
`pip freeze`	Display the version number on all installed packages.
`pip --help`	Display the documentation for `pip`.

See `https://pip.pypa.io/en/stable/user_guide/` for more complete documentation.

Workflows

There are several different ways to write and execute programs in Python. Try a variety of workflows to find what works best for you.

Text Editor + Terminal

The most basic way of developing in Python is to write code in a text editor, then run it using either the Python or IPython interpreter in the terminal.

There are many different text editors available for code development. Many text editors are designed specifically for computer programming which contain features such as syntax highlighting and error detection, and are highly customizable. Try installing and using some of the popular text editors listed below.

- Atom: https://atom.io/
- Sublime Text: https://www.sublimetext.com/
- Notepad++ (Windows): https://notepad-plus-plus.org/
- Geany: https://www.geany.org/
- Vim: https://www.vim.org/
- Emacs: https://www.gnu.org/software/emacs/

Once Python code has been written in a text editor and saved to a file, that file can be executed in the terminal or command line.

```
$ ls                        # List the files in the current directory.
hello_world.py
$ cat hello_world.py        # Print the contents of the file to the terminal.
print("hello, world!")
$ python hello_world.py     # Execute the file.
hello, world!

# Alternatively, start IPython and run the file.
$ ipython
IPython 6.5.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: %run hello_world.py
hello, world!
```

IPython is an enhanced version of Python that is more user-friendly and interactive. It has many features that cater to productivity such as tab completion and object introspection.

> **NOTE**
>
> While Mac and Linux computers come with a built-in bash terminal, Windows computers do not. Windows does come with *Powershell*, a terminal-like application, but some commands in Powershell are different than their bash analogs, and some bash commands are missing from Powershell altogether. There are two good alternatives to the bash terminal for Windows:
>
> - Windows subsystem for linux: docs.microsoft.com/en-us/windows/wsl/.

- Git bash: `https://gitforwindows.org/`.

Jupyter Notebook

The Jupyter Notebook (previously known as IPython Notebook) is a browser-based interface for Python that comes included as part of the Anaconda Python Distribution. It has an interface similar to the IPython interpreter, except that input is stored in cells and can be modified and re-evaluated as desired. See `https://github.com/jupyter/jupyter/wiki/` for some examples.

To begin using Jupyter Notebook, run the command `jupyter notebook` in the terminal. This will open your file system in a web browser in the Jupyter framework. To create a Jupyter Notebook, click the **New** drop down menu and choose **Python 3** under the **Notebooks** heading. A new tab will open with a new Jupyter Notebook.

Jupyter Notebooks differ from other forms of Python development in that notebook files contain not only the raw Python code, but also formatting information. As such, Juptyer Notebook files cannot be run in any other development environment. They also have the file extension `.ipynb` rather than the standard Python extension `.py`.

Jupyter Notebooks also support Markdown—a simple text formatting language—and LaTeX, and can embedded images, sound clips, videos, and more. This makes Jupyter Notebook the ideal platform for presenting code.

Integrated Development Environments

An *integrated development environment* (IDEs) is a program that provides a comprehensive environment with the tools necessary for development, all combined into a single application. Most IDEs have many tightly integrated tools that are easily accessible, but come with more overhead than a plain text editor. Consider trying out each of the following IDEs.

- JupyterLab: `http://jupyterlab.readthedocs.io/en/stable/`
- PyCharm: `https://www.jetbrains.com/pycharm/`
- Spyder: `http://code.google.com/p/spyderlib/`
- Eclipse with PyDev: `http://www.eclipse.org/`, `https://www.pydev.org/`

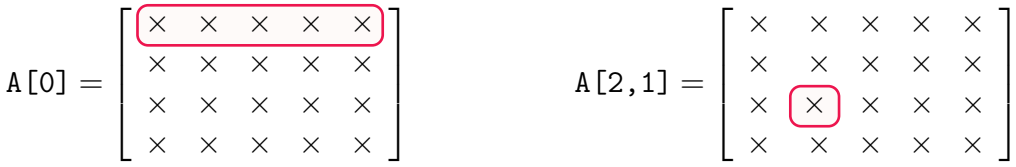
See `https://realpython.com/python-ides-code-editors-guide/` for a good overview of these (and other) workflow tools.

6.2 NumPy Visual Guide

NumPy Visual Guide **Lab Objective:** *NumPy operations can be difficult to visualize, but the concepts are straightforward. This appendix provides visual demonstrations of how NumPy arrays are used with slicing syntax, stacking, broadcasting, and axis-specific operations. Though these visualizations are for 1- or 2-dimensional arrays, the concepts can be extended to n-dimensional arrays.*

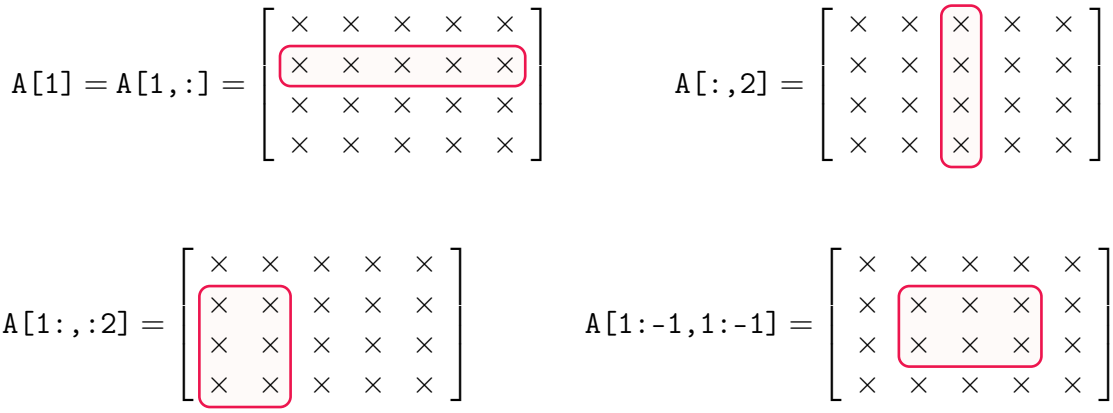
Data Access

The entries of a 2-D array are the rows of the matrix (as 1-D arrays). To access a single entry, enter the row index, a comma, and the column index. Remember that indexing begins with 0.



Slicing

A lone colon extracts an entire row or column from a 2-D array. The syntax `[a:b]` can be read as "the *a*th entry up to (but not including) the *b*th entry." Similarly, `[a:]` means "the *a*th entry to the end" and `[:b]` means "everything up to (but not including) the *b*th entry."



Stacking

`np.hstack()` stacks sequence of arrays horizontally and `np.vstack()` stacks a sequence of arrays vertically.

$$\texttt{np.hstack((A,B,A))} = \begin{bmatrix} \times & \times & \times & * & * & * & \times & \times & \times \\ \times & \times & \times & * & * & * & \times & \times & \times \\ \times & \times & \times & * & * & * & \times & \times & \times \end{bmatrix}$$

$$\texttt{np.vstack((A,B,A))} = \begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \\ * & * & * \\ * & * & * \\ * & * & * \\ \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{bmatrix}$$

Because 1-D arrays are flat, `np.hstack()` concatenates 1-D arrays and `np.vstack()` stacks them vertically. To make several 1-D arrays into the columns of a 2-D array, use `np.column_stack()`.

$$\texttt{x} = \begin{bmatrix} \times & \times & \times & \times \end{bmatrix} \qquad \texttt{y} = \begin{bmatrix} * & * & * & * \end{bmatrix}$$

$$\texttt{np.hstack((x,y,x))} = \begin{bmatrix} \times & \times & \times & \times & * & * & * & * & \times & \times & \times & \times \end{bmatrix}$$

$$\texttt{np.vstack((x,y,x))} = \begin{bmatrix} \times & \times & \times & \times \\ * & * & * & * \\ \times & \times & \times & \times \end{bmatrix} \qquad \texttt{np.column\_stack((x,y,x))} = \begin{bmatrix} \times & * & \times \\ \times & * & \times \\ \times & * & \times \\ \times & * & \times \end{bmatrix}$$

The functions `np.concatenate()` and `np.stack()` are more general versions of `np.hstack()` and `np.vstack()`, and `np.row_stack()` is an alias for `np.vstack()`.

Broadcasting

NumPy automatically aligns arrays for component-wise operations whenever possible. See `http://docs.scipy.org/doc/numpy/user/basics.broadcasting.html` for more in-depth examples and broadcasting rules.

$$\texttt{A} = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} \qquad \texttt{x} = \begin{bmatrix} 10 & 20 & 30 \end{bmatrix}$$

$$\texttt{A + x} = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} + \begin{bmatrix} 10 & 20 & 30 \end{bmatrix} = \begin{bmatrix} 11 & 22 & 33 \\ 11 & 22 & 33 \\ 11 & 22 & 33 \end{bmatrix}$$

$$\texttt{A + x.reshape((1,-1))} = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} + \begin{bmatrix} 10 \\ 20 \\ 30 \end{bmatrix} = \begin{bmatrix} 11 & 12 & 13 \\ 21 & 22 & 23 \\ 31 & 32 & 33 \end{bmatrix}$$

Operations along an Axis

Most array methods have an `axis` argument that allows an operation to be done along a given axis. To compute the sum of each column, use `axis=0`; to compute the sum of each row, use `axis=1`.

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix}$$

$$\texttt{A.sum(axis=0)} = \begin{bmatrix} \boxed{\begin{matrix}1\\1\\1\\1\end{matrix}} & \boxed{\begin{matrix}2\\2\\2\\2\end{matrix}} & \boxed{\begin{matrix}3\\3\\3\\3\end{matrix}} & \boxed{\begin{matrix}4\\4\\4\\4\end{matrix}} \end{bmatrix} = \begin{bmatrix} 4 & 8 & 12 & 16 \end{bmatrix}$$

$$\texttt{A.sum(axis=1)} = \begin{bmatrix} \boxed{1 & 2 & 3 & 4} \\ \boxed{1 & 2 & 3 & 4} \\ \boxed{1 & 2 & 3 & 4} \\ \boxed{1 & 2 & 3 & 4} \end{bmatrix} = \begin{bmatrix} 10 & 10 & 10 & 10 \end{bmatrix}$$

6.3 Plot Customization and Matplotlib Syntax Guide

Matplotlib Customization

Lab Objective: *The documentation for Matplotlib can be a little difficult to maneuver and basic information is sometimes difficult to find. This appendix condenses and demonstrates some of the more applicable and useful information on plot customizations. For an introduction to Matplotlib, see lab* **??**.

Colors

By default, every plot is assigned a different color specified by the "color cycle". It can be overwritten by specifying what color is desired in a few different ways.

-

Matplotlib recognizes some basic built-in colors.

Code	Color
`'b'`	blue
`'g'`	green
`'r'`	red
`'c'`	cyan
`'m'`	magenta
`'y'`	yellow
`'k'`	black
`'w'`	white

The following displays how these colors can be implemented. The result is displayed in Figure 6.1.

```
import numpy as np
from matplotlib import pyplot as plt

colors = np.array(["k", "g", "b", "r", "c", "m", "y", "w"])
x = np.linspace(0, 5, 1000)
y = np.ones(1000)

for i in xrange(8):
    plt.plot(x, i*y, colors[i], linewidth=18)

plt.ylim([-1, 8])
plt.savefig("colors.pdf", format='pdf')
plt.clf()
```

colors.py

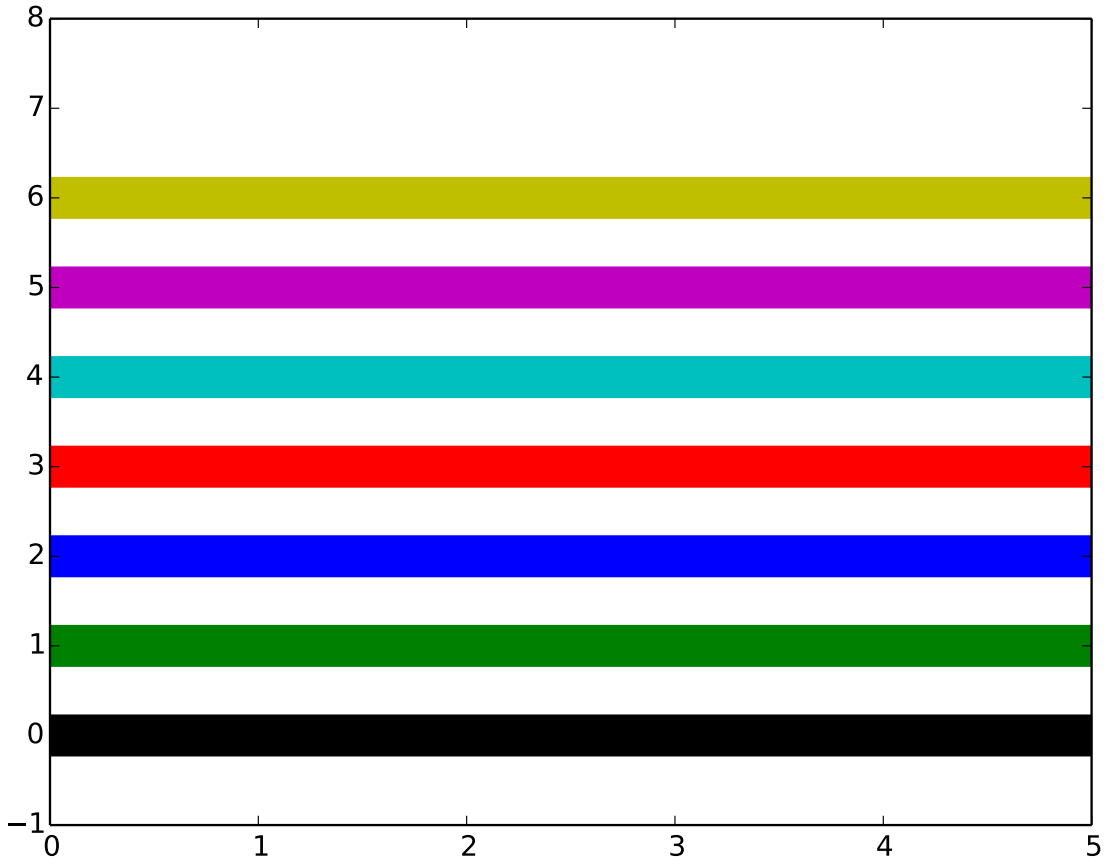


Figure 6.1: A display of all the built-in colors.

There are many other ways to specific colors. A popular method to access colors that are not built-in is to use a `RGB` tuple. Colors can also be specified using an html hex string or its associated html color name like `"DarkOliveGreen"`, `"FireBrick"`, `"LemonChiffon"`, `"MidnightBlue"`, `"PapayaWhip"`, or `"SeaGreen"`.

Window Limits

You may have noticed the use of `plt.ylim([ymin, ymax])` in the previous code. This explicitly sets the boundary of the y-axis. Similarily, `plt.xlim([xmin, xmax])` can be used to set the boundary of the x-axis. Doing both commands simultaneously is possible with the `plt.axis([xmin, xmax, ymin, ymax])`. Remember that these commands must be executed after the plot.

Lines

Thickness

You may have noticed that the width of the lines above seemed thin considering we wanted to inspect the line color. `linewidth` is a keyword argument that is defaulted to be `None` but can be given any real number to adjust the line width.

The following displays how `linewidth` is implemented. It is displayed in Figure 6.2.

```
lw = np.linspace(.5, 15, 8)

for i in xrange(8):
    plt.plot(x, i*y, colors[i], linewidth=lw[i])

plt.ylim([-1, 8])
plt.show()
```

linewidth.py

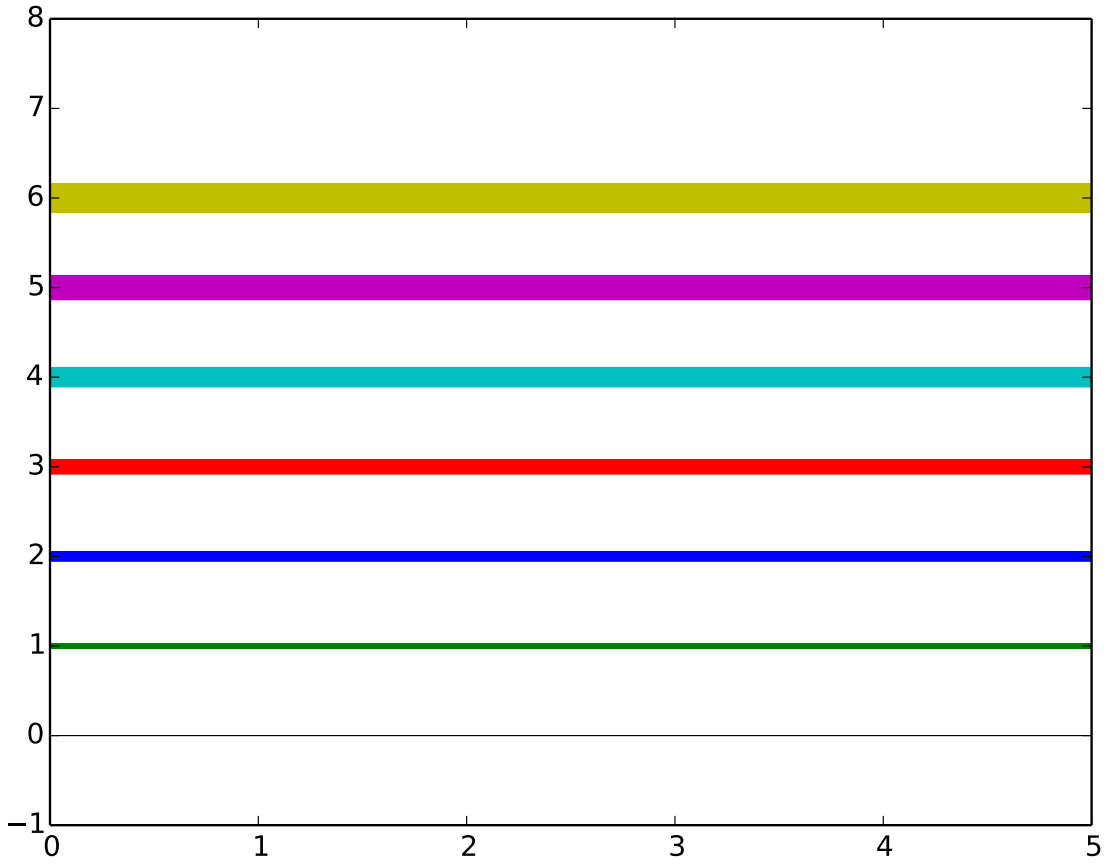


Figure 6.2: plot of varying linewidths.

Style

By default, plots are drawn with a solid line. The following are accepted format string characters to indicate line style.

character	description
-	solid line style
--	dashed line style
-.	dash-dot line style
:	dotted line style
.	point marker
,	pixel marker
o	circle marker
v	triangle_down marker
^	triangle_up marker
<	triangle_left marker
>	triangle_right marker
1	tri_down marker
2	tri_up marker
3	tri_left marker
4	tri_right marker
s	square marker
p	pentagon marker
*	star marker
h	hexagon1 marker
H	hexagon2 marker
+	plus marker
x	x marker
D	diamond marker
d	thin_diamond marker
|	vline marker
_	hline marker

The following displays how `linestyle` can be implemented. It is displayed in Figure 6.3.

```
x = np.linspace(0, 5, 10)
y = np.ones(10)
ls = np.array(['-.', ':', 'o', 's', '*', 'H', 'x', 'D'])

for i in xrange(8):
    plt.plot(x, i*y, colors[i]+ls[i])

plt.axis([-1, 6, -1, 8])
plt.show()
```

linestyle.py

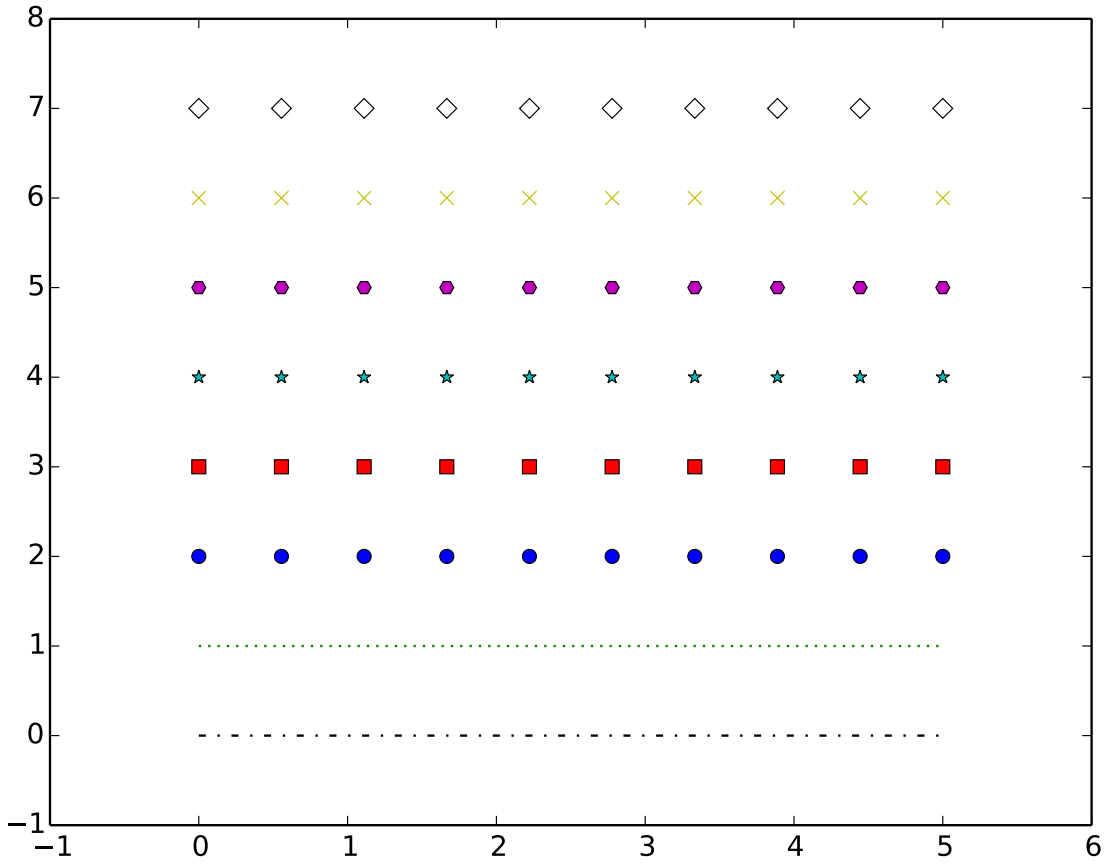


Figure 6.3: plot of varying linestyles.

Text

It is also possible to add text to your plots. To label your axes, the `plt.xlabel()` and the `plt.ylabel()` can both be used. The function `plt.title()` will add a title to a plot. If you are working with subplots, this command will add a title to the subplot you are currently modifying. To add a title above the entire figure, use `plt.suptitle()`.

All of the `text()` commands can be customized with `fontsize` and `color` keyword arguments.

We can add these elements to our previous example. It is displayed in Figure 6.4.

```
for i in xrange(8):
    plt.plot(x, i*y, colors[i]+ls[i])

plt.title("My Plot of Varying Linestyles", fontsize = 20, color = "gold")
plt.xlabel("x-axis", fontsize = 10, color = "darkcyan")
```

```
plt.ylabel("y-axis", fontsize = 10, color = "darkcyan")

plt.axis([-1, 6, -1, 8])
plt.show()
```

text.py

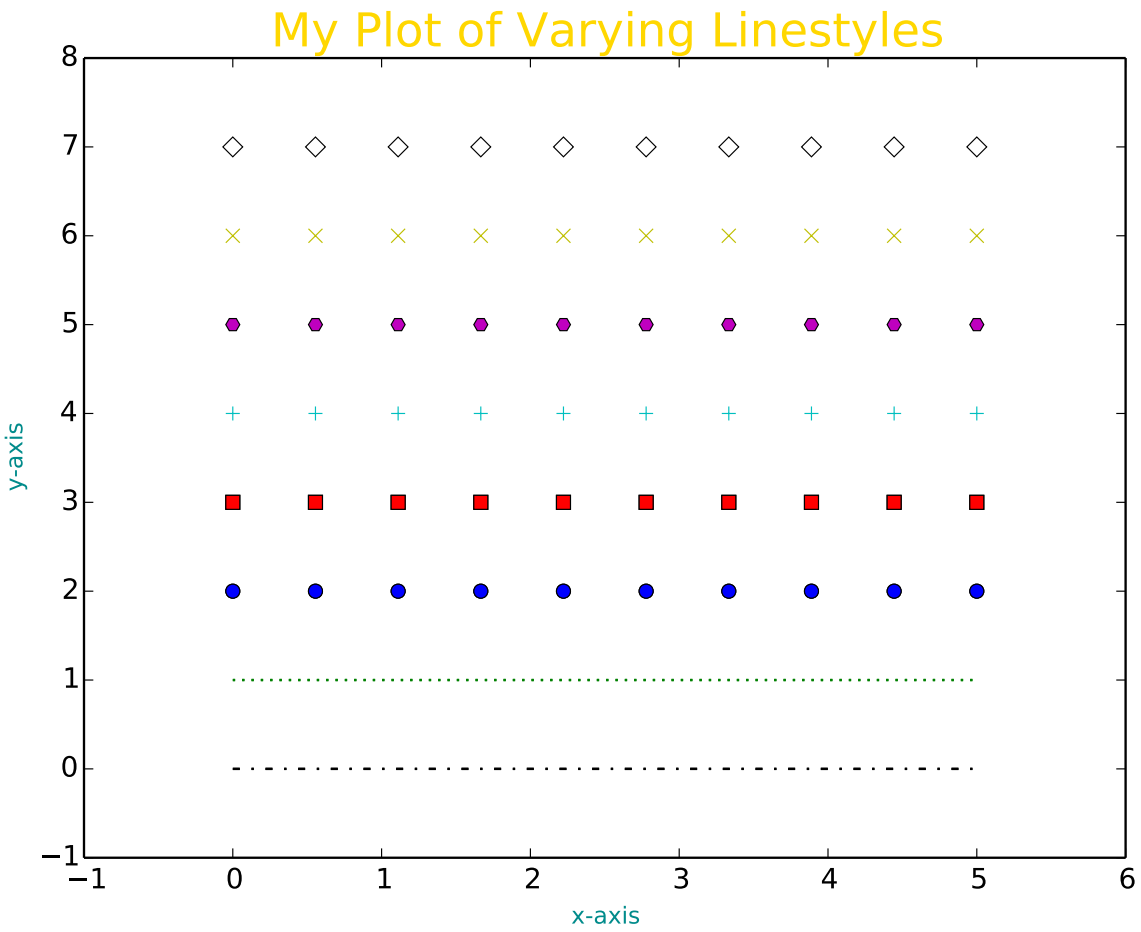


Figure 6.4: plot of varying linestyles using text labels.

See `http://matplotlib.org` for Matplotlib documentation.

6.4 Networkx - A Python Graph Algorithms Package

6.5 PuLP - An Optimization Modeling Tool for Python

Outcomes

- *Install and import PuLP*
- *Run basic first PuLP model*
- *Run "advanced" PuLP model using the algebraic modeling approach and importing data.*
- *Explore PuLP objects and possibilities*
- *Solve a Multi-Objective problem*

Resources

- *Documentation*
- *PyPi installation*
- *Examples*
- *Blog with tutorial*

PuLP is an optimizaiton modeling language that is written for Python. It is free and open source. Yay! See Section ?? for a discussion of other options for implementing your optimization problem. PuLP is convenient for it's simple syntax and easy installation.

Key benefits of using an algebraic modeling language like PuLP over Excel

- Easily readable models
- Precompute parameters within Python
- Reuse of common optimization models without recreating the equations

We will follow the introduction to pulp Jupyter Notebook Tutorial and the following application with a cleaner implementation.

6.5.1. Installation

Open a Jupyter notebook. In one of the cells, run the following command, based on which system you are running. It will take a minute to load and download the package.

[ ]:
```
## Install pulp (on windows)
!pip install pulp
```

[ ]:
```
# on a mac
pip install pulp
```

[ ]:
```
# on the VT ARC servers
import sys
!{sys.executable} -m pip install pulp
```

Installation (Continued) Now restart the kernel of your notebook (find the tab labeled Kernel in your Jupyter notebook, and in the drop down, select restart).

6.5.2. Example Problem

6.5.2.1. Product Mix Problem

$$\begin{aligned}
&\text{maximize} && Z = 3X_1 + 2X_2 && \text{(Objective function)} && (1.1)\\
&\text{subject to} && 10X_1 + 5X_2 \leq 300 && \text{(Constraint 1)} && (1.2)\\
& && 4X_1 + 4X_2 \leq 160 && \text{(Constraint 2)} && (1.3)\\
& && 2X_1 + 6X_2 \leq 180 && \text{(Constraint 3)} && (1.4)\\
&\text{and} && X_1, X_2 \geq 0 && \text{(Non-negative)} && (1.5)
\end{aligned}$$

OPTIMIZATION WITH PULP

[1]:
```
from pulp import *

# Define problem
prob = LpProblem(name='Product_Mix_Problem', sense=LpMaximize)

# Create decision variables and non-negative constraint
x1 = LpVariable(name='X1', lowBound=0, upBound=None, cat='Continuous')
x2 = LpVariable(name='X2', lowBound=0, upBound=None, cat='Continuous')

# Set objective function
prob += 3*x1 + 2*x2

# Set constraints
prob += 10*x1 + 5*x2 <= 300
prob += 4*x1 + 4*x2 <= 160
prob += 2*x1 + 6*x2 <= 180

# Solving problem
prob.solve()
print('Status', LpStatus[prob.status])
```

```
Status Optimal
```

[2]:
```
print("Status:", LpStatus[prob.status])
print("Objective value: ", prob.objective.value())
```

```
for v in prob.variables():
    print(v.name,': ', v.value())
```

```
Status: Optimal
Objective value:  100.0
X1 :  20.0
X2 :  20.0
```

6.5.3. Things we can do

[3]:
```
# print the problem
prob
```

[3]:
```
Product_Mix_Problem:
MAXIMIZE
3*X1 + 2*X2 + 0
SUBJECT TO
_C1: 10 X1 + 5 X2 <= 300

_C2: 4 X1 + 4 X2 <= 160

_C3: 2 X1 + 6 X2 <= 180

VARIABLES
X1 Continuous
X2 Continuous
```

[4]:
```
# get the objective function
prob.objective.value()
```

[4]:
```
100.0
```

[5]:
```
# get list of the variables
prob.variables()
```

[5]:
```
[X1, X2]
```

[6]:
```
for v in prob.variables():
    print(f'{v}: {v.varValue}')
```

```
X1: 20.0
X2: 20.0
```

6.5.3.1. Exploring the variables

[7]:
```
v = prob.variables()[0]
```

[9]:
```
v.name
```

[9]:
```
'X1'
```

[10]:
```
v.value()
```

[10]:
```
20.0
```

[11]:
```
v.varValue
```

[11]:
```
20.0
```

6.5.3.2. Other things you can do

[12]:
```
# get list of the constraints
prob.constraints
```

[12]:
```
OrderedDict([('_C1', 10*X1 + 5*X2 + -300 <= 0),
             ('_C2', 4*X1 + 4*X2 + -160 <= 0),
             ('_C3', 2*X1 + 6*X2 + -180 <= 0)])
```

[13]:
```
prob.to_dict()
```

[13]:
```
{'objective': {'name': 'OBJ',
  'coefficients': [{'name': 'X1', 'value': 3}, {'name': 'X2', 'value': 2}]},
 'constraints': [{'sense': -1,
   'pi': 0.2,
   'constant': -300,
   'name': None,
   'coefficients': [{'name': 'X1', 'value': 10}, {'name': 'X2', 'value': 5}]},
  {'sense': -1,
   'pi': 0.25,
   'constant': -160,
   'name': None,
   'coefficients': [{'name': 'X1', 'value': 4}, {'name': 'X2', 'value': 4}]},
  {'sense': -1,
   'pi': -0.0,
   'constant': -180,
   'name': None,
   'coefficients': [{'name': 'X1', 'value': 2}, {'name': 'X2', 'value': 6}]}],
 'variables': [{'lowBound': 0,
```

```
   'upBound': None,
   'cat': 'Continuous',
   'varValue': 20.0,
   'dj': -0.0,
   'name': 'X1'},
  {'lowBound': 0,
   'upBound': None,
   'cat': 'Continuous',
   'varValue': 20.0,
   'dj': -0.0,
   'name': 'X2'}],
 'parameters': {'name': 'Product_Mix_Problem',
  'sense': -1,
  'status': 1,
  'sol_status': 1},
 'sos1': [],
 'sos2': []}
```

[15]:
```
# Store problem information in a json
prob.to_json('Product_Mix_Problem.json')
```

6.5.4. Common issue

If you forget the <=, ==, or >= when writing a constraint, you will silently overwrite the objective function instead of adding a constraint!

6.5.4.1. Transportation Problem

Transport programming is a special form of linear programming, and in general, the objective function is cost minimization. The formula form and applicable variables of the Transport Planning Act are as follows. When supply and demand match, the constraint becomes an equation, but when supply and demand do not match, the constraint becomes an inequality.

Sets: - J = set of demand nodes - I = set of supply nodes

Parameters:

- D_j: Demand at node j
- S_i: Supply from node i
- c_{ij}: cost per unit to send supply i to demand j

Variables:

- X_ij: Transport volume from supply i to demand j (units)

- Objective function:

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

- Constraints:

$$\sum_{i=1}^{n} x_{ij} = S_i$$

$$\sum_{i=1}^{m} x_{ij} = D_j$$

$$x_{ij} \geq 0 \text{ for } i \in I, j \in J$$

6.5.4.2. Optimization with PuLP

Here we do a very basic implementation of the problem

[1]:
```
from pulp import *

prob = LpProblem('Transportation_Problem', LpMinimize)

x11 = LpVariable('X11', lowBound=0)
x12 = LpVariable('X12', lowBound=0)
x13 = LpVariable('X13', lowBound=0)
x14 = LpVariable('X14', lowBound=0)
x21 = LpVariable('X21', lowBound=0)
x22 = LpVariable('X22', lowBound=0)
x23 = LpVariable('X23', lowBound=0)
x24 = LpVariable('X24', lowBound=0)
x31 = LpVariable('X31', lowBound=0)
x32 = LpVariable('X32', lowBound=0)
x33 = LpVariable('X33', lowBound=0)
x34 = LpVariable('X34', lowBound=0)

prob += 4*x11 + 5*x12 + 6*x13 + 8*x14 + 4*x21 + 7*x22 + 9*x23 + 2*x24 + 5*x31 +␣
 ↪8*x32 + 7*x33 + 6*x34

prob += x11 + x12 + x13 + x14 == 120
prob += x21 + x22 + x23 + x24 == 150
prob += x31 + x32 + x33 + x34 == 200

prob += x11 + x21 + x31 == 100
prob += x12 + x22 + x32 == 60
prob += x13 + x23 + x33 == 130
prob += x14 + x24 + x34 == 180
```

```
# Solving problem
prob.solve();
```

[2]:
```
print("Status:", LpStatus[prob.status])
print("Objective value: ", prob.objective.value())

for v in prob.variables():
    print(v.name,': ', v.value())
```

```
Status: Optimal
Objective value:  2130.0
X11 :  60.0
X12 :  60.0
X13 :  0.0
X14 :  0.0
X21 :  0.0
X22 :  0.0
X23 :  0.0
X24 :  150.0
X31 :  40.0
X32 :  0.0
X33 :  130.0
X34 :  30.0
```

6.5.4.3. Optimization with PuLP: Round 2!

We now use set notation for this implementation

[3]:
```
from pulp import *

prob = LpProblem('Transportation_Problem', LpMinimize)


# Sets
n_suppliers = 3
n_buyers = 4

I = range(n_suppliers)
J = range(n_buyers)

routes = [(i, j) for i in I for j in J]


# Parameters
```

```
costs = [
    [4, 5, 6, 8],
    [4, 7, 9, 2],
    [5, 8, 7, 6]
]

supply = [120, 150, 200]
demand = [100, 60, 130, 180]



# Variables
x = LpVariable.dicts('X', routes, lowBound=0)

# Objective
prob += lpSum([x[i, j] * costs[i][j] for i in I for j in J])


# Constraints

## Supply Constraints
for i in range(n_suppliers):
    prob += lpSum([x[i, j] for j in J]) == supply[i], f"Supply{i}"

## Demand Constraints
for j in range(n_buyers):
    prob += lpSum([x[i, j] for i in I]) == demand[j], f"Demand{j}"

# Solving problem
prob.solve();
```

[4]:
```
print("Status:", LpStatus[prob.status])
print("Objective value: ", prob.objective.value())

for v in prob.variables():
    print(v.name,': ', v.value())
```

```
Status: Optimal
Objective value:  2130.0
X_(0,_0) :  60.0
X_(0,_1) :  60.0
X_(0,_2) :  0.0
X_(0,_3) :  0.0
X_(1,_0) :  0.0
```

```
X_(1,_1) :  0.0
X_(1,_2) :  0.0
X_(1,_3) :  150.0
X_(2,_0) :  40.0
X_(2,_1) :  0.0
X_(2,_2) :  130.0
X_(2,_3) :  30.0
```

6.5.5. Changing details of the problem

[5]:
```
original_obj = prob.objective
val = prob.objective.value()
r = 1.2
```

[6]:
```
prob += original_obj <= r*val, "Objective bound"
```

[7]:
```
prob
```

[7]:
```
Transportation_Problem:
MINIMIZE
4*X_(0,_0) + 5*X_(0,_1) + 6*X_(0,_2) + 8*X_(0,_3) + 4*X_(1,_0) + 7*X_(1,_1) +
9*X_(1,_2) + 2*X_(1,_3) + 5*X_(2,_0) + 8*X_(2,_1) + 7*X_(2,_2) + 6*X_(2,_3) + 0
SUBJECT TO
Supply0: X_(0,_0) + X_(0,_1) + X_(0,_2) + X_(0,_3) = 120

Supply1: X_(1,_0) + X_(1,_1) + X_(1,_2) + X_(1,_3) = 150

Supply2: X_(2,_0) + X_(2,_1) + X_(2,_2) + X_(2,_3) = 200

Demand0: X_(0,_0) + X_(1,_0) + X_(2,_0) = 100

Demand1: X_(0,_1) + X_(1,_1) + X_(2,_1) = 60

Demand2: X_(0,_2) + X_(1,_2) + X_(2,_2) = 130

Demand3: X_(0,_3) + X_(1,_3) + X_(2,_3) = 180

Objective_bound: 4 X_(0,_0) + 5 X_(0,_1) + 6 X_(0,_2) + 8 X_(0,_3)
 + 4 X_(1,_0) + 7 X_(1,_1) + 9 X_(1,_2) + 2 X_(1,_3) + 5 X_(2,_0) + 8 X_(2,_1)
 + 7 X_(2,_2) + 6 X_(2,_3) <= 2556

VARIABLES
X_(0,_0) Continuous
```

```
X_(0,_1) Continuous
X_(0,_2) Continuous
X_(0,_3) Continuous
X_(1,_0) Continuous
X_(1,_1) Continuous
X_(1,_2) Continuous
X_(1,_3) Continuous
X_(2,_0) Continuous
X_(2,_1) Continuous
X_(2,_2) Continuous
X_(2,_3) Continuous
```

[8]:
```
# Change the objective
prob += x[0,0]    # minimize x[0,0]
```

```
/opt/anaconda3/envs/python377/lib/python3.7/site-packages/pulp/pulp.py:1544:
UserWarning: Overwriting previously set objective.
  warnings.warn("Overwriting previously set objective.")
```

[9]:
```
prob.solve()
```

[9]:
```
1
```

[10]:
```
LpStatus[prob.status]
```

[10]:
```
'Optimal'
```

[11]:
```
print("Status:", LpStatus[prob.status])
print("Objective value: ", prob.objective.value())

for v in prob.variables():
    print(v.name,': ', v.value())
```

```
Status: Optimal
Objective value:  0.0
X_(0,_0) :  0.0
X_(0,_1) :  60.0
X_(0,_2) :  60.0
X_(0,_3) :  0.0
X_(1,_0) :  100.0
X_(1,_1) :  0.0
X_(1,_2) :  0.0
X_(1,_3) :  50.0
X_(2,_0) :  0.0
X_(2,_1) :  0.0
X_(2,_2) :  70.0
```

```
X_(2,_3) :  130.0
```

[12]:
```
original_obj
```

[12]:
```
4*X_(0,_0) + 5*X_(0,_1) + 6*X_(0,_2) + 8*X_(0,_3) + 4*X_(1,_0) + 7*X_(1,_1) +
9*X_(1,_2) + 2*X_(1,_3) + 5*X_(2,_0) + 8*X_(2,_1) + 7*X_(2,_2) + 6*X_(2,_3) + 0
```

[13]:
```
original_obj.value()
```

[13]:
```
2430.0
```

6.5.6. Changing Constraint Coefficients

[14]:
```
a = prob.constraints['Supply0']
```

[15]:
```
a.changeRHS(500)
```

[16]:
```
a
```

[16]:
```
1*X_(0,_0) + 1*X_(0,_1) + 1*X_(0,_2) + 1*X_(0,_3) + -500 = 0
```

[17]:
```
prob.constraints['Supply0'].keys()
```

[17]:
```
odict_keys([X_(0,_0), X_(0,_1), X_(0,_2), X_(0,_3)])
```

6.6 Multi Objective Optimization with PuLP

We consider two objectives and compute the pareto efficient frontier. We will implement the ε-constraint method. That is, we will add bounds based on an objective function and the optimize the alternate objective function.

6.6.0.1. Transportation Problem

Sets: - J = set of demand nodes - I = set of supply nodes

Parameters: - D_j: Demand at node j - S_i: Supply from node i - c_{ij}: cost per unit to send supply i to demand j

Variables: - x_{ij}: Transport volume from supply i to demand j (units)

- Objective function:

$$\min \left(obj1 = \sum_{i=1}^{n} \sum_{j=1}^{m} c_{ij} x_{ij}, \quad obj2 = x_{00} + x_{13} + x_{22} - x_{21} - x_{03} \right)$$

- Constraints:

$$\sum_{i=1}^{n} x_{ij} = S_i$$

$$\sum_{i=1}^{m} x_{ij} = D_j$$

- Decision variables:

$$x_{ij} \geq 0 \quad i \in I, j \in J$$

6.6.0.2. Initial Optimization with PuLP

[1]:
```
from pulp import *

prob = LpProblem('Transportation_Problem', LpMinimize)


# Sets
n_suppliers = 3
n_buyers = 4

I = range(n_suppliers)
J = range(n_buyers)

routes = [(i, j) for i in I for j in J]


# Parameters
costs = [
    [4, 5, 6, 8],
    [4, 7, 9, 2],
    [5, 8, 7, 6]
]

supply = [120, 150, 200]
demand = [100, 60, 130, 180]



# Variables
x = LpVariable.dicts('X', routes, lowBound=0)

# Objectives
obj1 = lpSum([x[i, j] * costs[i][j] for i in I for j in J])
```

```
obj2 = x[0,0] + x[1,3] + x[2,2] - x[2,1] - x[0,3]

## start with first objective
prob += obj1


# Constraints

## Supply Constraints
for i in range(n_suppliers):
    prob += lpSum([x[i, j] for j in J]) == supply[i], f"Supply{i}"

## Demand Constraints
for j in range(n_buyers):
    prob += lpSum([x[i, j] for i in I]) == demand[j], f"Demand{j}"

# Solving problem
prob.solve();
```

[2]:
```
print("Status:", LpStatus[prob.status])
print("Objective value: ", prob.objective.value())

for v in prob.variables():
    print(v.name,': ', v.value())
```

```
Status: Optimal
Objective value:  2130.0
X_(0,_0) :  60.0
X_(0,_1) :  60.0
X_(0,_2) :  0.0
X_(0,_3) :  0.0
X_(1,_0) :  0.0
X_(1,_1) :  0.0
X_(1,_2) :  0.0
X_(1,_3) :  150.0
X_(2,_0) :  40.0
X_(2,_1) :  0.0
X_(2,_2) :  130.0
X_(2,_3) :  30.0
```

[3]:
```
# Record objective value
obj1_opt = obj1.value()
obj1_opt
```

[3]: 2130.0

[4]:
```
# Add both objective values to a list and also the solution
obj1_vals = [obj1.value()]
obj2_vals = [obj2.value()]
feasible_points = [prob.variables()]
```

[5]:
```
# Change objective functions and compute optimal objective value for obj2
prob += obj2
prob.solve()

obj2_opt = obj2.value()
obj2_opt
```

```
/opt/anaconda3/envs/python377/lib/python3.7/site-packages/pulp/pulp.py:1537:
UserWarning: Overwriting previously set objective.
  warnings.warn("Overwriting previously set objective.")
```

[5]: -180.0

[6]:
```
# Append these values to the lists
obj1_vals.append(obj1.value())
obj2_vals.append(obj2.value())
feasible_points.append(prob.variables())
```

6.6.1. Creating the Pareto Efficient Frontier

[7]:
```
import numpy as np

# Create an inequality for objective 1
prob += obj1 <= obj1_opt, "Objective_bound1"
obj_constraint = prob.constraints["Objective_bound1"]
```

[8]:
```
# Set to optimize objective 2
prob += obj2
```

```
/opt/anaconda3/envs/python377/lib/python3.7/site-packages/pulp/pulp.py:1537:
UserWarning: Overwriting previously set objective.
  warnings.warn("Overwriting previously set objective.")
```

[9]:
```
# Adjusting objective bound of objective 1

r_values = np.arange(1,2000,10)
for r in r_values:
    obj_constraint.changeRHS(r + obj1_opt)
    if 1 == prob.solve():
        obj1_vals.append(obj1.value())
```

```
        obj2_vals.append(obj2.value())
        feasible_points.append(prob.variables())

# Remove objective 1 constraint
obj_constraint.changeRHS(0)
obj_constraint.clear()
```

[10]:
```
# Create constaint for objective 2
prob += obj2 <= obj2_opt, "Objective_bound2"
obj2_constraint = prob.constraints["Objective_bound2"]

# set objective to objective 1
prob += obj1
```

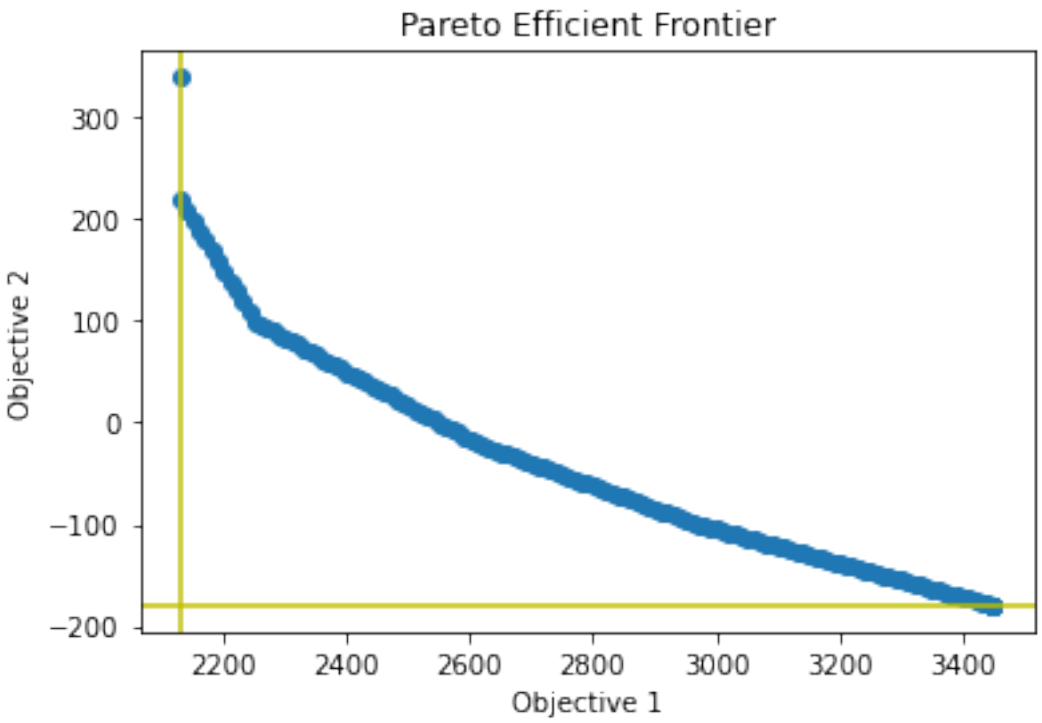
[11]:
```
# Adjusting objective bound of objective 2

r_values = np.arange(1,400,5) # may need to adjust this
for r in r_values:
    obj2_constraint.changeRHS(r*obj2_opt)
    if 1 == prob.solve():
        obj1_vals.append(obj1.value())
        obj2_vals.append(obj2.value())
        feasible_points.append(prob.variables())

# Remove objective 2 constraint
obj2_constraint.changeRHS(0)
obj2_constraint.clear()
```

[12]:
```
import matplotlib.pyplot as plt
plt.scatter(obj1_vals, obj2_vals)
plt.axvline(x=obj1_opt, color = 'y')
plt.axhline(y=obj2_opt, color = 'y')
plt.title("Pareto Efficient Frontier")
plt.xlabel("Objective 1")
plt.ylabel("Objective 2")
```

[12]:
```
Text(0, 0.5, 'Objective 2')
```



6.7 Comments

This code is a bit inefficient. It probably computes more pareto points than needed.

6.8 Jupyter Notebooks

Resources

- *https://github.com/mathinmse/mathinmse.github.io/blob/master/Lecture-00B-Notebook-Basics.ipynb*
- *https://github.com/mathinmse/mathinmse.github.io/blob/master/Lecture-00C-Writing-In-Jupyter.ipynb*

6.9 Reading and Writing

https://github.com/mathinmse/mathinmse.github.io/blob/master/Lecture-10B-Reading-and-Writing-Data.ipynb

6.10 Python Crash Course

https://github.com/rpmuller/PythonCrashCourse

6.11 Gurobi

Gurobi Log Tools

6.12 Plots, Pandas, and Geopandas

6.12.1. Geopandas

https://jcutrer.com/python/learn-geopandas-plotting-usmaps https://github.com/joncutrer/geopandas-tutorial