

# **Mathematical Programming and Operations Research**

## Modeling, Algorithms, and Complexity Examples in Python and Julia (Work in progress)

Edited by: Robert Hildebrand

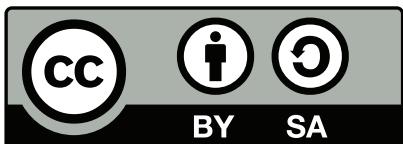
Contributors: Robert Hildebrand, Laurent Poirrier, Douglas Bish, Diego Moran

Version Compilation date: August 26, 2020





Copyright 2019 by the contributors listed on the title page.



This work is licensed under a Creative Commons  
“Attribution-ShareAlike 4.0 International” license.  
<https://creativecommons.org/>



This license allows everyone to remix, tweak, and build upon this work, even for commercial purposes, ... as long as they license their new creations under identical terms

...



... and if they credit the copyright holders.



This work aligns with the mission of UNESCO Open Educational Resources.

[https://en.unesco.org/themes/  
building-knowledge-societies/oer](https://en.unesco.org/themes/building-knowledge-societies/oer)

L<sup>A</sup>T<sub>E</sub>X

The source code of this book is available.

<https://github.com/open-optimization/>

---

<sup>1</sup>[https://en.wikipedia.org/wiki/Open\\_educational\\_resources#/media/File:Global\\_Open\\_Educational\\_Resources\\_Logo.svg](https://en.wikipedia.org/wiki/Open_educational_resources#/media/File:Global_Open_Educational_Resources_Logo.svg)

# 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 Advanced Learning. 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 Mathematics" with many contributors. See <https://github.com/Foundations-of->. 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 L<sup>A</sup>T<sub>E</sub>X using 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.



## Champions of Access to Knowledge



### OPEN TEXT

All digital forms of access to our high-quality open texts are entirely FREE! All content is reviewed for excellence and is wholly adaptable; custom editions are produced by Lyryx for those adopting Lyryx assessment. Access to the original source files is also open to anyone!



### ONLINE ASSESSMENT

We have been developing superior online formative assessment for more than 15 years. Our questions are continuously adapted with the content and reviewed for quality and sound pedagogy. To enhance learning, students receive immediate personalized feedback. Student grade reports and performance statistics are also provided.



### SUPPORT

Access to our in-house support team is available 7 days/week to provide prompt resolution to both student and instructor inquiries. In addition, we work one-on-one with instructors to provide a comprehensive system, customized for their course. This can include adapting the text, managing multiple sections, and more!



### INSTRUCTOR SUPPLEMENTS

Additional instructor resources are also freely accessible. Product dependent, these supplements include: full sets of adaptable slides and lecture notes, solutions manuals, and multiple choice question banks with an exam building tool.

## Contact Lyryx Today!

[info@lyryx.com](mailto:info@lyryx.com)

---

<sup>2</sup>This book was not produced by Lyryx, but this book has made substantial use of their open source material. We leave this page in here as a tribute to Lyryx for sharing their content.





# A First Course in Linear Algebra

## an Open Text

### BE A CHAMPION OF OER!

Contribute suggestions for improvements, new content, or errata:

A new topic

A new example

An interesting new question

A new or better proof to an existing theorem

Any other suggestions to improve the material

Contact Lyryx at [info@lyryx.com](mailto:info@lyryx.com) with your ideas.

### CONTRIBUTIONS

Ilijas Farah, York University

Ken Kuttler, Brigham Young University

#### Lyryx Learning Team

Bruce Bauslaugh

Peter Chow

Nathan Friess

Stephanie Keyowski

Claude Laflamme

Martha Laflamme

Jennifer MacKenzie

Tamsyn Murnaghan

Bogdan Sava

Larissa Stone

Ryan Yee

Ehsun Zahedi

# Foundations of Applied Mathematics

<https://github.com/Foundations-of-Applied-Mathematics>

## CONTRIBUTIONS

### List of Contributors

E. Evans	K. Finlinson
<i>Brigham Young University</i>	<i>Brigham Young University</i>
R. Evans	J. Fisher
<i>Brigham Young University</i>	<i>Brigham Young University</i>
J. Grout	R. Flores
<i>Drake University</i>	<i>Brigham Young University</i>
J. Humpherys	R. Fowers
<i>Brigham Young University</i>	<i>Brigham Young University</i>
T. Jarvis	A. Frandsen
<i>Brigham Young University</i>	<i>Brigham Young University</i>
J. Whitehead	R. Fuhriman
<i>Brigham Young University</i>	<i>Brigham Young University</i>
J. Adams	S. Giddens
<i>Brigham Young University</i>	<i>Brigham Young University</i>
J. Bejarano	C. Gigena
<i>Brigham Young University</i>	<i>Brigham Young University</i>
Z. Boyd	M. Graham
<i>Brigham Young University</i>	<i>Brigham Young University</i>
M. Brown	F. Glines
<i>Brigham Young University</i>	<i>Brigham Young University</i>
A. Carr	C. Glover
<i>Brigham Young University</i>	<i>Brigham Young University</i>
C. Carter	M. Goodwin
<i>Brigham Young University</i>	<i>Brigham Young University</i>
T. Christensen	R. Grout
<i>Brigham Young University</i>	<i>Brigham Young University</i>
M. Cook	D. Grundvig
<i>Brigham Young University</i>	<i>Brigham Young University</i>
R. Dorff	E. Hannesson
<i>Brigham Young University</i>	<i>Brigham Young University</i>
B. Ehlert	J. Hendricks
<i>Brigham Young University</i>	<i>Brigham Young University</i>
M. Fabiano	A. Henriksen
<i>Brigham Young University</i>	<i>Brigham Young University</i>

I. Henriksen	D. Reber
<i>Brigham Young University</i>	<i>Brigham Young University</i>
C. Hettinger	H. Ringer
<i>Brigham Young University</i>	<i>Brigham Young University</i>
S. Horst	C. Robertson
<i>Brigham Young University</i>	<i>Brigham Young University</i>
K. Jacobson	M. Russell
<i>Brigham Young University</i>	<i>Brigham Young University</i>
J. Leete	R. Sandberg
<i>Brigham Young University</i>	<i>Brigham Young University</i>
J. Lytle	C. Sawyer
<i>Brigham Young University</i>	<i>Brigham Young University</i>
R. McMurray	M. Stauffer
<i>Brigham Young University</i>	<i>Brigham Young University</i>
S. McQuarrie	J. Stewart
<i>Brigham Young University</i>	<i>Brigham Young University</i>
D. Miller	S. Suggs
<i>Brigham Young University</i>	<i>Brigham Young University</i>
J. Morrise	A. Tate
<i>Brigham Young University</i>	<i>Brigham Young University</i>
M. Morrise	T. Thompson
<i>Brigham Young University</i>	<i>Brigham Young University</i>
A. Morrow	M. Victors
<i>Brigham Young University</i>	<i>Brigham Young University</i>
R. Murray	J. Webb
<i>Brigham Young University</i>	<i>Brigham Young University</i>
J. Nelson	R. Webb
<i>Brigham Young University</i>	<i>Brigham Young University</i>
E. Parkinson	J. West
<i>Brigham Young University</i>	<i>Brigham Young University</i>
M. Probst	A. Zaitzeff
<i>Brigham Young University</i>	<i>Brigham Young University</i>
M. Proudfoot	
<i>Brigham Young University</i>	

This project is funded in part by the National Science Foundation, grant no. TUES Phase II DUE-1323785.

## **Revision History**

### **Current Revision: Version 2020 — Revision A**

Substantial open source content has been added to the book to fill out various sections. All content is still released under a CC-BY-SA 4.0 license.

---

2020 A            • R. Hildebrand: Added content on Linear Algebra and Linear Programming, and updated other content.

---

2019 A            • R. Hildebrand: First version of the book was created.



# Contents

---

<b>Contents</b>	<b>3</b>
<b>1 Resources and Notation</b>	<b>1</b>
<b>2 Mathematical Programming</b>	<b>5</b>
2.1 Linear Programming (LP) . . . . .	5
2.2 Mixed-Integer Linear Programming (MILP) . . . . .	7
2.3 Non-Linear Programming (NLP) . . . . .	9
2.3.1 Convex Programming . . . . .	9
2.3.2 Non-Convex Non-linear Programming . . . . .	10
2.4 Mixed-Integer Non-Linear Programming (MINLP) . . . . .	10
2.4.1 Convex Mixed-Integer Non-Linear Programming . . . . .	10
2.4.2 Non-Convex Mixed-Integer Non-Linear Programming . . . . .	10
<b>I Linear Programming [Under Construction]</b>	<b>11</b>
<b>3 LP Notes from Foundations of Applied Mathematics</b>	<b>15</b>
3.1 The Simplex Method . . . . .	21
3.1.1 Pivoting . . . . .	25
3.1.2 Termination and Reading the Dictionary . . . . .	27
3.2 The Product Mix Problem . . . . .	28
<b>4 Linear Programming Notes - Hildebrand</b>	<b>31</b>
4.0.0.1 Simplex Tableau Pivoter . . . . .	31
4.0.0.2 Videos . . . . .	31
4.1 Linear Programming Forms . . . . .	32
4.2 Linear Programming Dual . . . . .	32
4.3 Weak and Strong Duality . . . . .	32
4.3.1 Reduced Costs . . . . .	33
4.3.2 Tableau Based Pivoting . . . . .	34

<b>5 Linear Programming Book - Cheung</b>	<b>37</b>
<b>Preface</b>	<b>37</b>
<b>Notation</b>	<b>37</b>
5.1 Graphical example . . . . .	38
Exercises . . . . .	40
Solutions . . . . .	41
5.2 Definitions . . . . .	42
Exercises . . . . .	44
Solutions . . . . .	44
5.3 Farkas' Lemma . . . . .	45
5.4 Fundamental Theorem of Linear Programming . . . . .	46
Exercises . . . . .	47
Solutions . . . . .	48
5.5 Linear programming duality . . . . .	49
5.5.1 The dual problem . . . . .	52
Exercises . . . . .	54
Solutions . . . . .	54
5.6 Complementary slackness . . . . .	56
Exercises . . . . .	58
Solutions . . . . .	59
5.7 Basic feasible solution . . . . .	61
Exercises . . . . .	63
Solutions . . . . .	64
<b>6 LP Notes from ISE 5405</b>	<b>67</b>
6.1 Introduction to Optimization . . . . .	67
6.1.1 Notation . . . . .	68
6.2 Linear Optimization . . . . .	69
6.2.1 Problem Formulation . . . . .	69
6.2.2 Linear Algebra Review . . . . .	76
6.2.3 Linear Optimization Theory . . . . .	89
6.2.3.1 Optimality Conditions . . . . .	91
6.2.4 Solution Algorithms . . . . .	98
6.2.4.1 The Simplex Algorithm . . . . .	99
6.2.4.2 Dual Simplex Algorithm . . . . .	109

6.2.4.3	Primal-Dual Algorithm . . . . .	109
6.2.5	Sensitivity Analysis . . . . .	112
6.2.6	Theory Applications . . . . .	123

## II Integer Programming 127

<b>7</b>	<b>Integer Programming Formulations</b>	<b>129</b>
7.1	Knapsack Problem . . . . .	129
7.2	Capital Budgeting . . . . .	132
7.3	Set Covering . . . . .	135
7.3.1	Covering (Generalizing Set Cover) . . . . .	138
7.4	Assignment Problem . . . . .	139
7.5	Facility Location . . . . .	140
7.5.1	Capacitated Facility Location . . . . .	140
7.5.2	Uncapacitated Facility Location . . . . .	141
7.6	Network Flow . . . . .	141
7.6.1	Example - Multicommodity Flow . . . . .	141
7.6.2	Corresponding optimization problems . . . . .	142
7.6.3	Relation to other problems . . . . .	143
7.6.4	Usage . . . . .	143
7.7	Transportation Problem . . . . .	143
7.8	Jobshop Scheduling: Makespan Minimization . . . . .	143
7.9	Generalized Assignment Problem (GAP) . . . . .	144
7.9.1	In special cases . . . . .	144
7.9.2	Explanation of definition . . . . .	144
7.10	Other examples . . . . .	145
7.11	Modeling Tricks . . . . .	145
7.11.1	Either Or Constraints . . . . .	145
7.11.2	If then implications . . . . .	146
7.11.3	Binary reformulation of integer variables . . . . .	148
7.11.4	SOS1 Constraints . . . . .	149
7.11.5	SOS2 Constraints . . . . .	150
7.11.6	Piecewise linear functions with SOS2 constraint . . . . .	150
7.11.7	Maximizing a minimum . . . . .	152
7.11.8	Relaxing (nonlinear) equality constraints . . . . .	152

## 6 ■ CONTENTS

7.12 Notes from AIMMS modeling book . . . . .	152
7.12.1 Further Topics . . . . .	153
7.13 Reference guide for modeling and examples . . . . .	153
7.14 MIP Solvers and Modeling Tools . . . . .	153
<b>8 Algorithms and Complexity . . . . .</b>	<b>155</b>
8.1 Big-O Notation . . . . .	155
8.2 Algorithms - Example with Bubble Sort . . . . .	158
8.2.1 Sorting . . . . .	159
8.3 Complexity Classes . . . . .	161
8.3.1 P . . . . .	161
8.3.2 NP . . . . .	162
8.3.3 NP-Hard . . . . .	162
8.3.4 NP-Complete . . . . .	163
8.4 Relevant Terminology . . . . .	164
8.5 Matching Problem . . . . .	165
8.5.1 Greedy Algorithm for Maximal Matching . . . . .	166
8.5.2 Other algorithms to look at . . . . .	167
8.6 Minimum Spanning Tree . . . . .	167
8.6.1 Kruskal's algorithm . . . . .	168
8.6.2 Prim's Algorithm . . . . .	168
8.7 Traveling Salesman Problem . . . . .	168
8.7.1 Nearest Neighbor - Construction Heuristic . . . . .	169
8.7.2 Double Spanning Tree - 2-Apx . . . . .	169
8.7.3 Christofides - Approximation Algorithm - (3/2)-Apx . . . . .	171
<b>9 Exponential Size Formulations . . . . .</b>	<b>173</b>
9.1 Cutting Stock . . . . .	173
9.1.1 Pattern formulation . . . . .	175
9.1.2 Column Generation . . . . .	176
9.1.3 Cutting Stock - Multiple widths . . . . .	177
9.2 Spanning Trees . . . . .	178
9.3 Traveling Salesman Problem . . . . .	178
9.3.1 MTZ Model . . . . .	180
9.3.2 Dantzig-Fulkerson-Johnson Model . . . . .	184
9.3.3 Traveling Salesman Problem - Branching Solution . . . . .	186

9.4	Vehicle Routing Problem (VRP) . . . . .	186
9.4.1	Case Study: Bus Routing in Boston . . . . .	187
9.5	Literature and other notes . . . . .	187
9.5.1	Google maps data . . . . .	187
9.5.2	TSP In Excel . . . . .	187
<b>10</b>	<b>Algorithms to Solve Integer Programs</b>	<b>189</b>
10.1	LP to solve IP . . . . .	189
10.1.1	Rounding LP Solution can be bad! . . . . .	190
10.1.2	Rounding LP solution can be infeasible! . . . . .	190
10.1.3	Fractional Knapsack . . . . .	190
10.2	Branch and Bound . . . . .	191
10.2.1	Algorithm . . . . .	191
10.2.2	Knapsack Problem and 0/1 branching . . . . .	193
10.2.3	Traveling Salesman Problem solution via Branching . . . . .	194
10.3	Cutting Planes . . . . .	194
10.3.1	Chvátal Cuts . . . . .	196
10.3.2	Gomory Cuts . . . . .	198
10.4	Branching Rules . . . . .	200
10.5	Lagrangian Relaxation for Branch and Bound . . . . .	200
10.6	Benders Decomposition . . . . .	200
10.7	Literature . . . . .	200
10.8	Other material for Integer Linear Programming . . . . .	200
Exercises	. . . . .	205
Solutions	. . . . .	206
10.8.1	Other discrete problems . . . . .	206
10.8.2	Assignment Problem and the Hungarian Algorithm . . . . .	206
10.8.3	History of Computation in Combinatorial Optimization . . . . .	207
<b>11</b>	<b>Heuristics for TSP</b>	<b>209</b>
11.1	Construction Heuristics . . . . .	209
11.1.1	Random Solution . . . . .	209
11.1.2	Nearest Neighbor . . . . .	210
11.1.3	Insertion Method . . . . .	210
11.2	Improvement Heuristics . . . . .	211
11.2.1	2-Opt (Subtour Reversal) . . . . .	211

11.2.2 3-Opt . . . . .	212
11.2.3 $k$ -Opt . . . . .	212
11.3 Meta-Heuristics . . . . .	212
11.3.1 Hill Climbing (2-Opt for TSP) . . . . .	212
11.3.2 Simulated Annealing . . . . .	214
11.3.3 Tabu Search . . . . .	215
11.3.4 Genetic Algorithms . . . . .	216
11.3.5 Greedy randomized adaptive search procedure (GRASP) . . . . .	216
11.3.6 Ant Colony Optimization . . . . .	216
11.4 Computational Comparisons . . . . .	216
11.5 Polyhedra . . . . .	218
11.5.1 Convex Hull . . . . .	218
<b>12 Dynamic Programming</b>	<b>221</b>
<b>13 Policy Function Iteration</b>	<b>231</b>
<b>14 Graph Algorithms</b>	<b>239</b>
<b>III Nonlinear Programming</b>	<b>241</b>
<b>15 Non-linear Programming (NLP)</b>	<b>243</b>
15.1 Convex Sets . . . . .	244
15.2 Convex Functions . . . . .	246
15.2.1 Proving Convexity - Characterizations . . . . .	249
15.2.2 Proving Convexity - Composition Tricks . . . . .	250
15.3 Convex Optimization Examples . . . . .	251
15.3.1 Unconstrained Optimization: Linear Regression . . . . .	251
15.4 Machine Learning - SVM . . . . .	253
15.4.0.1 Feasible separation . . . . .	254
15.4.0.2 SVM . . . . .	254
15.4.0.3 Approximate SVM . . . . .	255
15.4.1 SVM with non-linear separators . . . . .	255
15.4.2 Support Vector Machines . . . . .	257
15.5 Classification . . . . .	258
15.5.1 Machine Learning . . . . .	258
15.5.2 Neural Networks . . . . .	258

15.6 Box Volume Optimization in Scipy.Minimize . . . . .	258
15.7 Modeling . . . . .	258
15.7.1 Minimum distance to circles . . . . .	260
15.8 Machine Learning . . . . .	263
15.9 Machine Learning - Supervised Learning - Regression . . . . .	263
15.10 Machine learning - Supervised Learning - Classification . . . . .	263
15.10.1 Python SGD implementation and video . . . . .	264
<b>16 NLP Algorithms</b>	<b>265</b>
16.1 Algorithms Introduction . . . . .	265
16.2 1-Dimensional Algorithms . . . . .	265
16.2.1 Golden Search Method - Derivative Free Algorithm . . . . .	266
16.2.1.1 Example: . . . . .	267
16.2.2 Bisection Method - 1st Order Method (using Derivative) . . . . .	268
16.2.2.1 Minimization Interpretation . . . . .	268
16.2.2.2 Root finding Interpretation . . . . .	268
16.2.3 Gradient Descent - 1st Order Method (using Derivative) . . . . .	268
16.2.4 Newton's Method - 2nd Order Method (using Derivative and Hessian) . . . . .	269
16.3 Multi-Variate Unconstrained Optimizaiton . . . . .	269
16.3.1 Descent Methods - Unconstrained Optimization - Gradient, Newton . . . . .	269
16.3.1.1 Choice of $\alpha_t$ . . . . .	270
16.3.1.2 Choice of $d_t$ using $\nabla f(x)$ . . . . .	270
16.3.2 Stochastic Gradient Descent - The mother of all algorithms. . . . .	270
16.3.2.1 Choice of $\Delta_k$ using the hessian $\nabla^2 f(x)$ . . . . .	272
16.4 Constrained Convex Nonlinear Programming . . . . .	273
16.4.1 Barrier Method . . . . .	273
<b>17 Computational Issues with NLP</b>	<b>275</b>
17.1 Irrational Solutions . . . . .	275
17.2 Discrete Solutions . . . . .	275
17.3 Convex NLP Harder than LP . . . . .	275
17.4 NLP is harder than IP . . . . .	276
17.5 Karush-Huhn-Tucker (KKT) Conditions . . . . .	277
17.6 Gradient Free Algorithms . . . . .	279
17.6.1 Needler-Mead . . . . .	279
<b>18 Material to add...</b>	<b>281</b>

18.0.1 Bisection Method and Newton's Method . . . . .	281
18.1 Gradient Descent . . . . .	281
18.2 Quadratic Programming . . . . .	281
<b>19 Least Squares and Computing Eigenvalues</b>	<b>285</b>
<b>20 Newton's Method</b>	<b>299</b>
<b>21 One-dimensional Optimization</b>	<b>309</b>
<b>22 Gradient Descent Methods</b>	<b>317</b>
<b>23 Interior Point 1: Linear Programs</b>	<b>329</b>
<b>24 Interior Point 2: Quadratic Programs</b>	<b>341</b>
<b>25 OpenGym AI</b>	<b>353</b>
<b>26 K-Means Clustering</b>	<b>361</b>
<b>IV Advanced Optimization</b>	<b>371</b>
<b>27 Integral polyhedra, TU matrices, TDI systems</b>	<b>373</b>
27.1 Integral polyhedra . . . . .	373
27.1.1 Basics . . . . .	373
27.1.2 Properties . . . . .	373
27.2 Unimodular and totally unimodular matrices . . . . .	374
27.2.1 Unimodular matrices . . . . .	374
27.2.2 Totally unimodular matrices . . . . .	374
27.2.3 How to detect unimodularity and totally unimodularity . . . . .	375
27.2.4 Examples of totally unimodular matrices . . . . .	375
27.3 Totally dual integral systems . . . . .	375
27.3.1 Basics . . . . .	376
27.3.2 Properties . . . . .	376
27.3.3 Totally unimodularity and TDI systems . . . . .	376
27.3.4 Examples of TDI systems . . . . .	376
<b>28 Cutting Planes</b>	<b>377</b>
28.1 Introduction . . . . .	377

28.1.1	Cutting planes . . . . .	377
28.1.2	Cutting plane algorithm . . . . .	377
28.1.3	How to compute cutting planes . . . . .	378
28.2	Computing cutting planes for general IPs . . . . .	378
28.2.1	Chvátal-Gomory cuts (for pure integer programs) . . . . .	378
28.2.1.1	A nice property of CG cuts . . . . .	379
28.2.2	Cutting planes from the Simplex tableau . . . . .	379
28.2.2.1	A simple inequality . . . . .	380
28.2.3	A stronger inequality . . . . .	380
28.2.3.1	Gomory's fractional cut . . . . .	380
28.3	Mixed-integer rounding cuts (MIR) . . . . .	381
28.3.1	Basic MIR inequality . . . . .	381
28.3.2	GMIC Derivation from MIR cut . . . . .	381
28.4	Cutting planes from lattice free sets . . . . .	384
28.4.1	The general case . . . . .	384
28.4.2	Split cuts . . . . .	385
28.4.3	MIR inequalities from one-row relaxations . . . . .	386
28.4.3.1	One-row relaxation . . . . .	386
28.4.3.2	Applying the basic MIR inequality . . . . .	386
28.5	Gomory Mixed-integer cut (GMI) . . . . .	387
<b>29</b>	<b>Cutting Planes</b>	<b>389</b>
29.1	Introduction . . . . .	389
29.1.1	Cutting planes . . . . .	389
29.1.2	Cutting plane algorithm . . . . .	389
29.1.3	How to compute cutting planes . . . . .	390
29.1.4	Cutting planes from the Simplex tableau . . . . .	390
29.1.4.1	Gomory's fractional cut . . . . .	390
29.2	Mixed-integer rounding cuts (MIR) . . . . .	391
29.2.1	Basic MIR inequality . . . . .	391
29.2.2	GMIC Derivation from MIR cut . . . . .	392
29.3	Split Cuts . . . . .	396
<b>30</b>	<b>Closures</b>	<b>399</b>
<b>31</b>	<b>Dantzig Wolfe</b>	<b>401</b>

<b>32 Lattices, IP in fixed dimensions</b>	<b>403</b>
<b>33 Introduction to computational complexity</b>	<b>405</b>
33.1 Introduction . . . . .	405
33.2 Problem, instance, size . . . . .	405
33.2.1 Problem, instance . . . . .	405
33.2.2 Format and examples of problems/instances . . . . .	406
33.2.3 Size of an instance . . . . .	406
33.3 Algorithms, running time, Big-O notation . . . . .	406
33.3.1 Basics . . . . .	406
33.3.2 Worst-time complexity . . . . .	407
33.3.3 Big-O notation . . . . .	407
33.3.4 Examples . . . . .	407
33.4 Basics . . . . .	407
33.5 Complexity classes . . . . .	408
33.5.1 Polynomial time problems . . . . .	408
33.5.2 Non-deterministic polynomial time problems . . . . .	408
33.5.3 Complements of problems in NP . . . . .	409
33.6 Relationship between the classes . . . . .	409
33.6.1 A basic result . . . . .	409
33.6.2 An \$1,000,000 open question . . . . .	410
33.7 Comparing problems, Polynomial time reductions . . . . .	410
33.8 Comparing problems, Polynomial time reductions . . . . .	410
33.8.1 Definition . . . . .	410
33.8.2 Basic properties . . . . .	411
33.9 NP-Completeness . . . . .	412
33.9.1 The basics . . . . .	412
33.9.2 Do NP-complete problems exist? . . . . .	412
33.10 NP-Hardness . . . . .	412
33.11 Exercises . . . . .	413
<b>34 Reformulation and Decomposition Techniques</b>	<b>419</b>
34.1 Lagrangean relaxation . . . . .	419
34.2 Column generation . . . . .	420
34.2.1 The master problem and the pricing subproblem . . . . .	420
34.2.2 Dantzig-Wolf decomposition . . . . .	421

34.3 Extended formulations . . . . .	422
34.4 Benders decomposition . . . . .	423
<b>35 Stochastic Programming</b>	<b>425</b>
<b>V Appendix - Linear Algebra Background</b>	<b>427</b>
<b>A Linear Transformations</b>	<b>429</b>
<b>B Linear Systems</b>	<b>443</b>
<b>C Systems of Equations</b>	<b>459</b>
C.1 Systems of Equations, Geometry . . . . .	459
C.2 Systems Of Equations, Algebraic Procedures . . . . .	463
C.2.1 Elementary Operations . . . . .	464
C.2.2 Gaussian Elimination . . . . .	468
C.2.3 Uniqueness of the Reduced Row-Echelon Form . . . . .	481
C.2.4 Rank and Homogeneous Systems . . . . .	482
<b>D Matrices</b>	<b>489</b>
D.1 Matrix Arithmetic . . . . .	489
D.1.1 Addition of Matrices . . . . .	491
D.1.2 Scalar Multiplication of Matrices . . . . .	493
D.1.3 Multiplication of Matrices . . . . .	494
D.1.4 The $i, j^{\text{th}}$ Entry of a Product . . . . .	501
D.1.5 Properties of Matrix Multiplication . . . . .	504
D.1.6 The Transpose . . . . .	505
D.1.7 The Identity and Inverses . . . . .	507
D.1.8 Finding the Inverse of a Matrix . . . . .	509
<b>E Determinants</b>	<b>515</b>
E.1 Basic Techniques and Properties . . . . .	515
E.1.1 Cofactors and $2 \times 2$ Determinants . . . . .	515
E.1.2 The Determinant of a Triangular Matrix . . . . .	521
E.2 Applications of the Determinant . . . . .	522
E.2.1 Cramer's Rule . . . . .	523
<b>F <math>\mathbb{R}^n</math></b>	<b>527</b>

F.1	Vectors in $\mathbb{R}^n$	527
F.2	Algebra in $\mathbb{R}^n$	530
F.2.1	Addition of Vectors in $\mathbb{R}^n$	530
F.2.2	Scalar Multiplication of Vectors in $\mathbb{R}^n$	532
F.3	Geometric Meaning of Vector Addition	533
F.4	Length of a Vector	536
F.5	Geometric Meaning of Scalar Multiplication	540
F.6	The Dot Product	542
F.6.1	The Dot Product	542
F.6.2	The Geometric Significance of the Dot Product	545
<b>G</b>	<b>Spectral Theory</b>	<b>549</b>
G.1	Eigenvalues and Eigenvectors of a Matrix	549
G.1.1	Definition of Eigenvectors and Eigenvalues	549
G.1.2	Finding Eigenvectors and Eigenvalues	552
G.1.3	Eigenvalues and Eigenvectors for Special Types of Matrices	558
G.2	Positive Semi-Definite Matrices	559
G.2.1	Definitions	560
G.2.2	Eigenvalues	562
G.2.3	Quadratic forms	562
G.2.4	Properties	563
G.2.4.1	Inverse of positive definite matrix	563
G.2.4.2	Scaling	563
G.2.4.3	Addition	563
G.2.4.4	Multiplication	563
G.2.4.5	Cholesky decomposition	563
G.2.4.6	Square root	564
G.2.4.7	Submatrices	564
G.2.5	Convexity	564
G.2.5.1	Further properties	564
G.2.5.2	Block matrices	564
G.2.5.3	Local extrema	565
G.2.5.4	Covariance	565
G.2.6	External links	565

<b>VI Other Appendices</b>	<b>567</b>
<b>A Some Prerequisite Topics</b>	<b>569</b>
A.1 Sets and Set Notation . . . . .	569
A.2 Well Ordering and Induction . . . . .	571
<b>B Installing and Managing Python</b>	<b>575</b>
<b>C NumPy Visual Guide</b>	<b>581</b>
<b>D Matplotlib Customization</b>	<b>585</b>
D.1 Jupyter Notebooks . . . . .	590
D.2 Reading and Writing . . . . .	590
D.3 Python Crash Course . . . . .	591
D.4 Excel Solver . . . . .	591
D.4.1 Videos . . . . .	591
D.4.2 Links . . . . .	592
D.5 GECODE and MiniZinc . . . . .	592
D.6 Optaplanner . . . . .	592
D.7 Python Modeling/Optimization . . . . .	592
D.7.1 SCIP . . . . .	592
D.7.2 Pyomo . . . . .	593
D.7.3 Python-MIP . . . . .	593
D.7.4 Local Solver . . . . .	593
D.7.5 GUROBI . . . . .	593
D.7.6 CPLEX . . . . .	593
D.7.7 Scipy . . . . .	593
D.8 Julia . . . . .	593
D.8.1 JuMP . . . . .	593
D.9 LINDO/LINGO . . . . .	594
D.10 Foundations of Machine Learning . . . . .	594
D.11 Convex Optimization . . . . .	594



# 1. Resources and Notation

---

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

## FREE NOTES AND TEXTBOOKS

- A first course in optimization by Jon Lee
- Introduction to Optimization Notes by Komei Fukuda
- Convex Optimization by Boyd and Vandenberghe
- LP notes of Michel Goemans from MIT
- Understanding and Using Linear Programming - Matousek and Gärtner [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 O'Reilly.com.
- Lesson files for the Winston Book
- Video instructions for solver and an example workbook



## 2 ■ Resources and Notation

### **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.  $1, 2, 3, -1, -2, -3, \dots$ )
- $\mathbb{Q}$  - 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.  $1, 1.5, \pi, e, -11/5$ )

## 4 ■ Resources and Notation

- \ - 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.  $[3, \pi, -e, \sqrt{2}]$ )
- $\sum_{i=1}^4 i = 1 + 2 + 3 + 4$
- $\sum_{i=1}^4 i^2 = 1^2 + 2^2 + 3^2 + 4^2$
- $\sum_{i=1}^4 x_i = x_1 + x_2 + x_3 + x_4$
- $\square$  - 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:

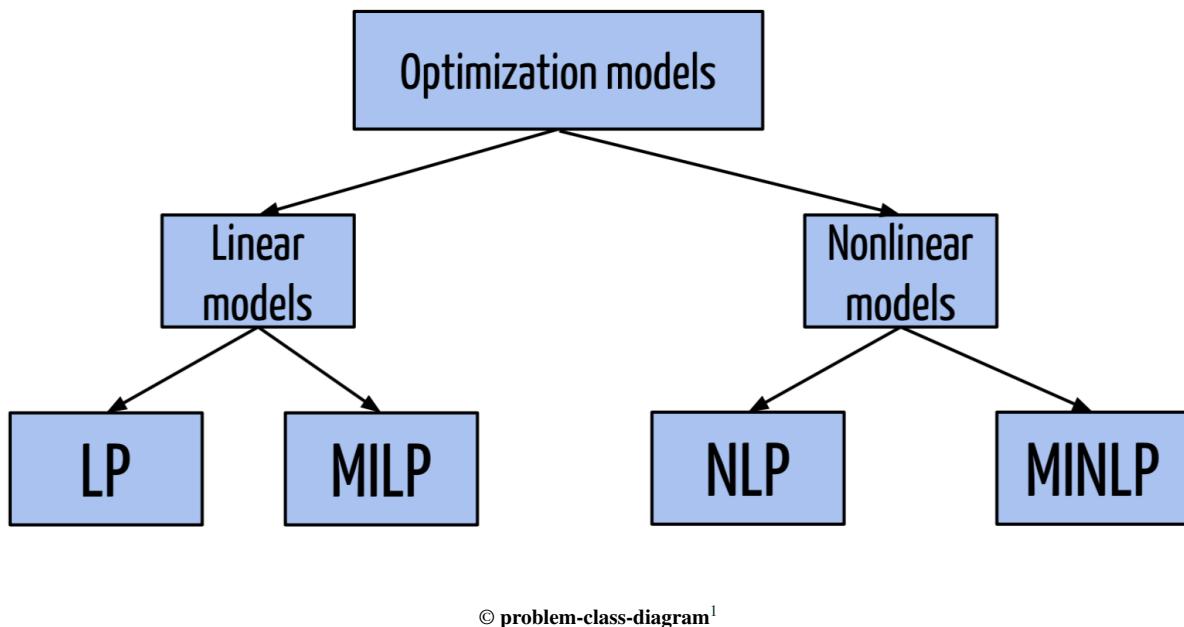
$$\forall x \in \mathbb{Q}^n \exists y \in \mathbb{Z}^n \setminus \{0\}^n s.t. x^\top y \in \{0, 1\}$$

"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

---

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<sup>1</sup>

**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 chapter 8 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.

### 2.1 Linear Programming (LP)

---

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

<sup>1</sup>problem-class-diagram, from problem-class-diagram. problem-class-diagram, problem-class-diagram.

**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

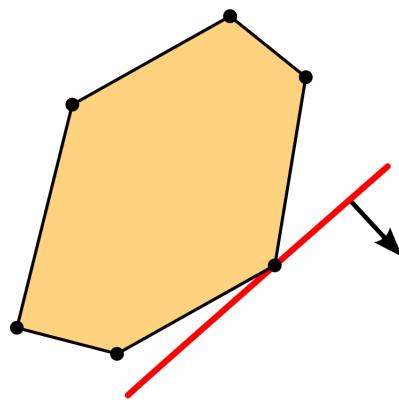
$$\begin{aligned} \max \quad & c^\top x \\ \text{s.t.} \quad & Ax \leq b \\ & x \geq 0 \end{aligned} \tag{2.1}$$

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<sup>2</sup>

**Figure 2.2: Linear programming constraints and objective.**

Figure 2.2

<sup>2</sup>wiki/File/linear-programming.png, from wiki/File/linear-programming.png. wiki/File/linear-programming.png, wiki/File/linear-programming.png.

**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 7, 9, 10, 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.

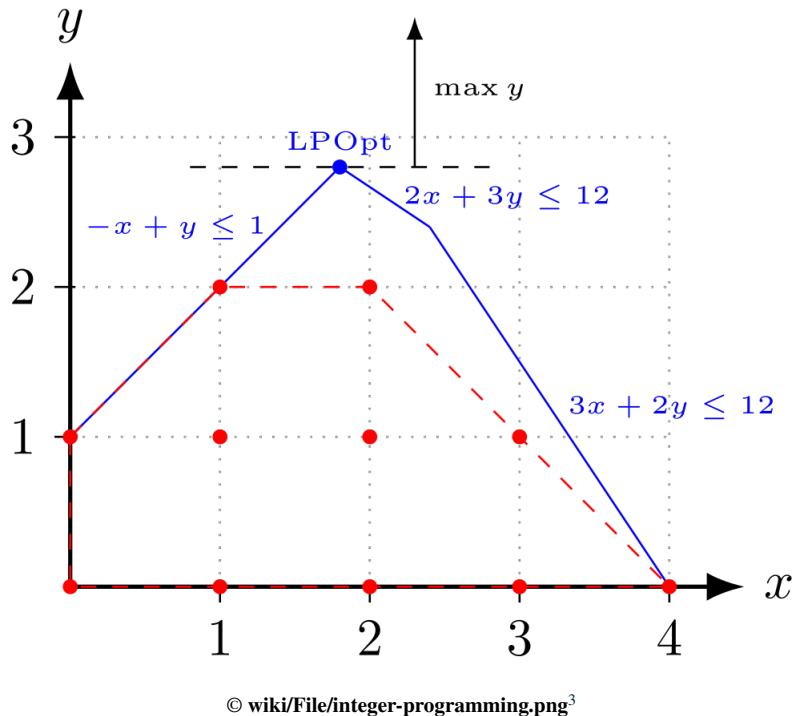
Figure 2.3

**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

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

<sup>3</sup>wiki/File/integer-programming.png, from wiki/File/integer-programming.png. wiki/File/integer-programming.png, wiki/File/integer-programming.png.



**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  $x_1, \dots, x_n \in \mathbb{Z}$  and  $d$  continuous variables  $x_{n+1}, \dots, x_{n+d} \in \mathbb{R}$ . 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  $f(x): \mathbb{R}^d \rightarrow \mathbb{R}$  and other functions  $f_i(x): \mathbb{R}^d \rightarrow \mathbb{R}$  for  $i = 1, \dots, m$ , the *nonlinear programming* problem is

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

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  $f(x): \mathbb{R}^d \rightarrow \mathbb{R}$  and convex functions  $f_i(x): \mathbb{R}^d \rightarrow \mathbb{R}$  for  $i = 1, \dots, m$ , the *convex programming* problem is

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

### Example 2.2

Convex 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

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

## 2.4 Mixed-Integer Non-Linear Programming (MINLP)

---

### 2.4.1. Convex Mixed-Integer Non-Linear Programming

---

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

---

## **Part I**

# **Linear Programming [Under Construction]**





This part of the book needs substantial revision. It is currently comprised of 3 sets of notes pasted together.



# 3. LP Notes from Foundations of Applied Mathematics

---

## Linear Programs

---

A *linear program* is a linear constrained optimization problem. Such a problem can be stated in several different forms, one of which is

$$\begin{aligned} & \text{minimize} && \mathbf{c}^T \mathbf{x} \\ & \text{subject to} && G\mathbf{x} \leq \mathbf{h} \\ & && A\mathbf{x} = \mathbf{b}. \end{aligned}$$

The symbol  $\leq$  denotes that the components of  $G\mathbf{x}$  are less than the components of  $\mathbf{h}$ . In other words, if  $\mathbf{x} \leq \mathbf{y}$ , then  $x_i < y_i$  for all  $x_i \in \mathbf{x}$  and  $y_i \in \mathbf{y}$ .

Define vector  $\mathbf{s} \geq 0$  such that the constraint  $G\mathbf{x} + \mathbf{s} = \mathbf{h}$ . This vector is known as a *slack variable*. Since  $\mathbf{s} \geq 0$ , the constraint  $G\mathbf{x} + \mathbf{s} = \mathbf{h}$  is equivalent to  $G\mathbf{x} \leq \mathbf{h}$ .

With a slack variable, a new form of the linear program is found:

$$\begin{aligned} & \text{minimize} && \mathbf{c}^T \mathbf{x} \\ & \text{subject to} && G\mathbf{x} + \mathbf{s} = \mathbf{h} \\ & && A\mathbf{x} = \mathbf{b} \\ & && \mathbf{s} \geq 0. \end{aligned}$$

This is the formulation used by CVXOPT. It requires that the matrix  $A$  has full row rank, and that the block matrix  $[G \ A]^T$  has full column rank.

Consider the following example:

$$\begin{aligned} & \text{minimize} && -4x_1 - 5x_2 \\ & \text{subject to} && x_1 + 2x_2 \leq 3 \\ & && 2x_1 + x_2 = 3 \\ & && x_1, x_2 \geq 0 \end{aligned}$$

Recall that all inequalities must be less than or equal to, so that  $G\mathbf{x} \leq \mathbf{h}$ . Because the final two constraints are  $x_1, x_2 \geq 0$ , they need to be adjusted to be  $\leq$  constraints. This is easily done by multiplying by  $-1$ , resulting in the constraints  $-x_1, -x_2 \leq 0$ . If we define

$$G = \begin{bmatrix} 1 & 2 \\ -1 & 0 \\ 0 & -1 \end{bmatrix}, \quad \mathbf{h} = \begin{bmatrix} 3 \\ 0 \\ 0 \end{bmatrix}, \quad A = [2 \ 1], \quad \text{and} \quad \mathbf{b} = [3]$$

then we can express the constraints compactly as

$$\begin{aligned} G\mathbf{x} &\leq \mathbf{h}, \\ A\mathbf{x} &= \mathbf{b}, \end{aligned} \quad \text{where} \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}.$$

By adding a slack variable  $s$ , we can write our constraints as

$$G\mathbf{x} + s = \mathbf{h},$$

which matches the form discussed above.

### Problem 3.1: Linear Optimization

Solve the following linear optimization problem:

$$\begin{aligned} \text{minimize} \quad & 2x_1 + x_2 + 3x_3 \\ \text{subject to} \quad & x_1 + 2x_2 \geq 3 \\ & 2x_1 + 10x_2 + 3x_3 \geq 10 \\ & x_i \geq 0 \quad \text{for } i = 1, 2, 3 \end{aligned}$$

Return the minimizer  $\mathbf{x}$  and the primal objective value.

(Hint: make the necessary adjustments so that all inequality constraints are  $\leq$  rather than  $\geq$ ).

## $l_1$ Norm

---

The  $l_1$  norm is defined

$$\|\mathbf{x}\|_1 = \sum_{i=1}^n |x_i|.$$

A  $l_1$  minimization problem is minimizing a vector's  $l_1$  norm, while fitting certain constraints. It can be written in the following form:

$$\begin{aligned} \text{minimize} \quad & \|\mathbf{x}\|_1 \\ \text{subject to} \quad & A\mathbf{x} = \mathbf{b}. \end{aligned}$$

This problem can be converted into a linear program by introducing an additional vector  $\mathbf{u}$  of length  $n$ . Define  $\mathbf{u}$  such that  $|x_i| \leq u_i$ . Thus,  $-u_i - x_i \leq 0$  and  $-u_i + x_i \leq 0$ . These two inequalities can be added to the linear system as constraints. Additionally, this means that  $\|\mathbf{x}\|_1 \leq \|\mathbf{u}\|_1$ . So minimizing  $\|\mathbf{u}\|_1$  subject to the given constraints will in turn minimize  $\|\mathbf{x}\|_1$ . This can be written as follows:

$$\begin{aligned} \text{minimize} \quad & [1^T \ 0^T] \begin{bmatrix} \mathbf{u} \\ \mathbf{x} \end{bmatrix} \\ \text{subject to} \quad & \begin{bmatrix} -I & I \\ -I & -I \\ -I & 0 \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ \mathbf{x} \end{bmatrix} \leq \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \\ & [0 \ A] \begin{bmatrix} \mathbf{u} \\ \mathbf{x} \end{bmatrix} = \mathbf{b}. \end{aligned}$$

Solving this gives values for the optimal  $\mathbf{u}$  and the optimal  $\mathbf{x}$ , but we only care about the optimal  $\mathbf{x}$ .

### Problem 3.2: $\ell_1$ Norm Minimization

Write a function called `l1Min()` that accepts a matrix  $A$  and vector  $\mathbf{b}$  as NumPy arrays and solves the  $\ell_1$  minimization problem. Return the minimizer  $\mathbf{x}$  and the primal objective value. Remember to first discard the unnecessary  $u$  values from the minimizer.

To test your function consider the matrix  $A$  and vector  $\mathbf{b}$  below.

$$A = \begin{bmatrix} 1 & 2 & 1 & 1 \\ 0 & 3 & -2 & -1 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} 7 \\ 4 \end{bmatrix}$$

The linear system  $A\mathbf{x} = \mathbf{b}$  has infinitely many solutions. Use `l1Min()` to verify that the solution which minimizes  $\|\mathbf{x}\|_1$  is approximately  $\mathbf{x} = [0., 2.571, 1.857, 0.]^T$  and the minimum objective value is approximately 4.429.

## The Transportation Problem

---

Consider the following transportation problem: A piano company needs to transport thirteen pianos from their three supply centers (denoted by 1, 2, 3) to two demand centers (4, 5). Transporting a piano from a supply center to a demand center incurs a cost, listed in Table 3.3. The company wants to minimize shipping costs for the pianos while meeting the demand.

Supply Center	Number of pianos available
1	7
2	2
3	4

**Table 3.1:** Number of pianos available at each supply center

Demand Center	Number of pianos needed
4	5
5	8

**Table 3.2:** Number of pianos needed at each demand center

Supply Center	Demand Center	Cost of transportation	Number of pianos
1	4	4	$p_1$
1	5	7	$p_2$
2	4	6	$p_3$
2	5	8	$p_4$
3	4	8	$p_5$
3	5	9	$p_6$

**Table 3.3: Cost of transporting one piano from a supply center to a demand center**

A system of constraints is defined for the variables  $p_1, p_2, p_3, p_4, p_5$ , and  $p_6$ . First, there cannot be a negative number of pianos so the variables must be nonnegative. Next, the Tables 3.1 and 3.2 define the following three supply constraints and two demand constraints:

$$\begin{aligned} p_1 + p_2 &= 7 \\ p_3 + p_4 &= 2 \\ p_5 + p_6 &= 4 \\ p_1 + p_3 + p_5 &= 5 \\ p_2 + p_4 + p_6 &= 8 \end{aligned}$$

The objective function is the number of pianos shipped from each location multiplied by the respective cost (found in Table 3.3):

$$4p_1 + 7p_2 + 6p_3 + 8p_4 + 8p_5 + 9p_6.$$

### NOTE

Since our answers must be integers, in general this problem turns out to be an NP-hard problem. There is a whole field devoted to dealing with integer constraints, called *integer linear programming*, which is beyond the scope of this lab. Fortunately, we can treat this particular problem as a standard linear program and still obtain integer solutions.

Recall the variables are nonnegative, so  $p_1, p_2, p_3, p_4, p_5, p_6 \geq 0$ . Thus,  $G$  and  $\mathbf{h}$  constrain the variables to be non-negative.

### Problem 3.3: Transportation problem

Solve the transportation problem by converting the last equality constraint into an inequality constraint. Return the minimizer  $\mathbf{x}$  and the primal objective value.

## Eating on a Budget

---

In 2009, the inmates of Morgan County jail convinced Judge Clemon of the Federal District Court in Birmingham to put Sheriff Barlett in jail for malnutrition. Under Alabama law, in order to encourage less spending, "the chief lawman could go light on prisoners' meals and pocket the leftover change."<sup>1</sup>. Sheriffs had to ensure a minimum amount of nutrition for inmates, but minimizing costs meant more money for the sheriffs themselves. Judge Clemon jailed Sheriff Barlett one night until a plan was made to use all allotted funds, 1.75 per inmate, to feed prisoners more nutritious meals. While this case made national news, the controversy of feeding prisoners in Alabama continues as of 2019<sup>2</sup>.

The problem of minimizing cost while reaching healthy nutritional requirements can be approached as a convex optimization problem. Rather than viewing this problem from the sheriff's perspective, we view it from the perspective of a college student trying to minimize food cost in order to pay for higher education, all while meeting standard nutritional guidelines.

The file `food.npy` contains a dataset with nutritional facts for 18 foods that have been eaten frequently by college students working on this text. A subset of this dataset can be found in Table 3.4, where the "Food" column contains the list of all 18 foods.

The columns of the full dataset are:

- Column 1:  $p$ , price (dollars)
- Column 2:  $s$ , number of servings
- Column 3:  $c$ , calories per serving
- Column 4:  $f$ , fat per serving (grams)
- Column 5:  $\hat{s}$ , sugar per serving (grams)
- Column 6:  $\hat{c}$ , calcium per serving (milligrams)
- Column 7:  $\hat{f}$ , fiber per serving (grams)
- Column 8:  $\hat{p}$ , protein per serving (grams)

---

<sup>1</sup>Nossiter, Adam, 8 Jan 2009, "As His Inmates Grew Thinner, a Sheriff's Wallet Grew Fatter", *New York Times*,<https://www.nytimes.com/2009/01/09/us/09sheriff.html>

<sup>2</sup>Sheets, Connor, 31 January 2019, "Alabama sheriffs urge lawmakers to get them out of the jail food business", <https://www.al.com/news/2019/01/alabama-sheriffs-urge-lawmakers-to-get-them-out-of-the-jail-food-business.html>

Food	Price $p$ dollars	Serving Size $s$	Calories $c$	Fat $f$ g	Sugar $\hat{s}$ g	Calcium $\hat{c}$ mg	Fiber $\hat{f}$ g	Protein $\hat{p}$ g
Ramen	6.88	48	190	7	0	0	0	5
Potatoes	0.48	1	290	0.4	3.2	53.8	6.9	7.9
Milk	1.79	16	130	5	12	250	0	8
Eggs	1.32	12	70	5	0	28	0	6
Pasta	3.88	8	200	1	2	0	2	7
Frozen Pizza	2.78	5	350	11	5	150	2	14
Potato Chips	2.12	14	160	11	1	0	1	1
Frozen Broccoli	0.98	4	25	0	1	25	2	1
Carrots	0.98	2	52.5	0.3	6.1	42.2	3.6	1.2
Bananas	0.24	1	105	0.4	14.4	5.9	3.1	1.3
Tortillas	3.48	18	140	4	0	0	0	3
Cheese	1.88	8	110	8	0	191	0	6
Yogurt	3.47	5	90	0	7	190	0	17
Bread	1.28	6	120	2	2	60	0.01	4
Chicken	9.76	20	110	3	0	0	0	20
Rice	8.43	40	205	0.4	0.1	15.8	0.6	4.2
Pasta Sauce	3.57	15	60	1.5	7	20	2	2
Lettuce	1.78	6	8	0.1	0.6	15.5	1	0.6

**Table 3.4: Subset of table containing food data**

According to the FDA<sup>1</sup> and US Department of Health, someone on a 2000 calorie diet should have no more than 2000 calories, no more than 65 grams of fat, no more than 50 grams of sugar<sup>2</sup>, at least 1000 milligrams of calcium<sup>1</sup>, at least 25 grams of fiber, and at least 46 grams of protein<sup>2</sup> per day.

We can rewrite this as a convex optimization problem below.

<sup>1</sup>url`https://www.accessdata.fda.gov/scripts/InteractiveNutritionFactsLabel/pdv.html`

<sup>2</sup>`https://www.today.com/health/4-rules-added-sugars-how-calculate-your-daily-limit-t34731`

<sup>1</sup>26 Sept 2018, `https://ods.od.nih.gov/factsheets/Calcium-HealthProfessional/`

<sup>2</sup>`https://www.accessdata.fda.gov/scripts/InteractiveNutritionFactsLabel/protein.html`

$$\begin{aligned}
 & \text{minimize} \sum_{i=1}^{18} p_i x_i, \\
 & \text{subject to} \sum_{i=1}^{18} c_i x_i \leq 2000, \\
 & \quad \sum_{i=1}^{18} f_i x_i \leq 65, \\
 & \quad \sum_{i=1}^{18} \hat{s}_i x_i \leq 50, \\
 & \quad \sum_{i=1}^{18} \hat{c}_i x_i \geq 1000, \\
 & \quad \sum_{i=1}^{18} \hat{f}_i x_i \geq 25, \\
 & \quad \sum_{i=1}^{18} \hat{p}_i x_i \geq 46, \\
 & \quad x_i \geq 0.
 \end{aligned}$$

### Problem 3.4: Eating on a Budget

Read in the file `food.npy`. Identify how much of each food item a college student should eat to minimize cost spent each day. Return the minimizing vector and the total amount of money spent. What is the food you should eat most each day? What are the three foods you should eat most each week?

(Hint: Each nutritional value must be multiplied by the number of servings to get the nutrition value of the whole product).

## 3.1 The Simplex Method

---

**The Simplex Method Lab Objective:** *The Simplex Method is a straightforward algorithm for finding optimal solutions to optimization problems with linear constraints and cost functions. Because of its simplicity and applicability, this algorithm has been named one of the most important algorithms invented within the last 100 years. In this lab we implement a standard Simplex solver for the primal problem.*

## Standard Form

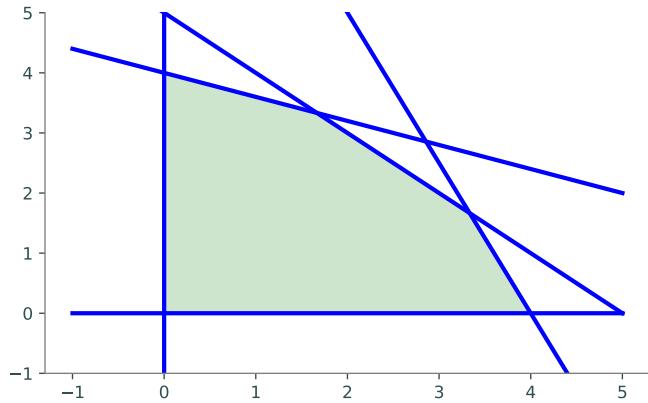
---

The Simplex Algorithm accepts a linear constrained optimization problem, also called a *linear program*, in the form given below:

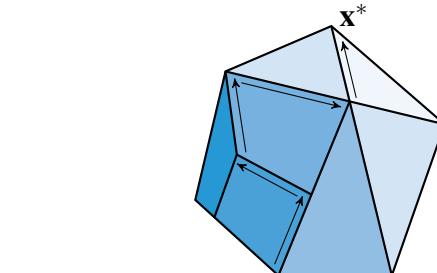
$$\begin{array}{ll} \text{minimize} & \mathbf{c}^T \mathbf{x} \\ \text{subject to} & A\mathbf{x} \leq \mathbf{b} \\ & \mathbf{x} \geq 0 \end{array}$$

Note that any linear program can be converted to standard form, so there is no loss of generality in restricting our attention to this particular formulation.

Such an optimization problem defines a region in space called the *feasible region*, the set of points satisfying the constraints. Because the constraints are all linear, the feasible region forms a geometric object called a *polytope*, having flat faces and edges (see Figure 3.1). The Simplex Algorithm jumps among the vertices of the feasible region searching for an optimal point. It does this by moving along the edges of the feasible region in such a way that the objective function is always increased after each move.



(a) The feasible region for a linear program with 2-dimensional constraints.



(b) The feasible region for a linear program with 3-dimensional constraints.

**Figure 3.1:** If an optimal point exists, it is one of the vertices of the polyhedron. The simplex algorithm searches for optimal points by moving between adjacent vertices in a direction that increases the value of the objective function until it finds an optimal vertex.

Implementing the Simplex Algorithm is straightforward, provided one carefully follows the procedure. We will break the algorithm into several small steps, and write a function to perform each one. To become familiar with the execution of the Simplex algorithm, it is helpful to work several examples by hand.

## The Simplex Solver

---

Our program will be more lengthy than many other lab exercises and will consist of a collection of functions working together to produce a final result. It is important to clearly define the task of each function and how all the functions will work together. If this program is written haphazardly, it will be much longer and more difficult to read than it needs to be. We will walk you through the steps of implementing the Simplex Algorithm as a Python class.

For demonstration purposes, we will use the following linear program.

$$\begin{array}{ll} \text{minimize} & -3x_0 - 2x_1 \\ \text{subject to} & x_0 - x_1 \leq 2 \\ & 3x_0 + x_1 \leq 5 \\ & 4x_0 + 3x_1 \leq 7 \\ & x_0, x_1 \geq 0. \end{array}$$

## Accepting a Linear Program

---

Our first task is to determine if we can even use the Simplex algorithm. Assuming that the problem is presented to us in standard form, we need to check that the feasible region includes the origin. For now, we only check for feasibility at the origin. A more robust solver sets up the auxiliary problem and solves it to find a starting point if the origin is infeasible.

### Problem 3.5: Check feasibility at the origin.

*Write a class that accepts the arrays  $\mathbf{c}$ ,  $A$ , and  $\mathbf{b}$  of a linear optimization problem in standard form. In the constructor, check that the system is feasible at the origin. That is, check that  $A\mathbf{x} \leq \mathbf{b}$  when  $\mathbf{x} = 0$ . Raise a `ValueError` if the problem is not feasible at the origin.*

## Adding Slack Variables

---

The next step is to convert the inequality constraints  $A\mathbf{x} \leq \mathbf{b}$  into equality constraints by introducing a slack variable for each constraint equation. If the constraint matrix  $A$  is an  $m \times n$  matrix, then there are  $m$  slack variables, one for each row of  $A$ . Grouping all of the slack variables into a vector  $\mathbf{w}$  of length  $m$ , the constraints now take the form  $A\mathbf{x} + \mathbf{w} = \mathbf{b}$ . In our example, we have

$$\mathbf{w} = \begin{bmatrix} x_2 \\ x_3 \\ x_4 \end{bmatrix}$$

When adding slack variables, it is useful to represent all of your variables, both the original primal variables and the additional slack variables, in a convenient manner. One effective way is to refer to a

variable by its subscript. For example, we can use the integers 0 through  $n - 1$  to refer to the original (non-slack) variables  $x_0$  through  $x_{n-1}$ , and we can use the integers  $n$  through  $n + m - 1$  to track the slack variables (where the slack variable corresponding to the  $i$ th row of the constraint matrix is represented by the index  $n + i - 1$ ).

We also need some way to track which variables are *independent* (non-zero) and which variables are *dependent* (those that have value 0). This can be done using the objective function. At anytime during the optimization process, the non-zero variables in the objective function are *independent* and all other variables are *dependent*.

## Creating a Dictionary

---

After we have determined that our program is feasible, we need to create the *dictionary* (sometimes called the *tableau*), a matrix to track the state of the algorithm.

There are many different ways to build your dictionary. One way is to mimic the dictionary that is often used when performing the Simplex Algorithm by hand. To do this we will set the corresponding dependent variable equations to 0. For example, if  $x_5$  were a dependent variable we would expect to see a -1 in the column that represents  $x_5$ . Define

$$\bar{A} = [ A \quad I_m ],$$

where  $I_m$  is the  $m \times m$  identity matrix we will use to represent our slack variables, and define

$$\bar{\mathbf{c}} = \begin{bmatrix} \mathbf{c} \\ 0 \end{bmatrix}.$$

That is,  $\bar{\mathbf{c}} \in \mathbb{R}^{n+m}$  such that the first  $n$  entries are  $\mathbf{c}$  and the final  $m$  entries are zeros. Then the initial dictionary has the form

$$D = \begin{bmatrix} 0 & \bar{\mathbf{c}}^T \\ \mathbf{b} & -\bar{A} \end{bmatrix} \tag{3.1}$$

The columns of the dictionary correspond to each of the variables (both primal and slack), and the rows of the dictionary correspond to the dependent variables.

For our example the initial dictionary is

$$D = \begin{bmatrix} 0 & -3 & -2 & 0 & 0 & 0 \\ 2 & -1 & 1 & -1 & 0 & 0 \\ 5 & -3 & -1 & 0 & -1 & 0 \\ 7 & -4 & -3 & 0 & 0 & -1 \end{bmatrix}.$$

The advantage of using this kind of dictionary is that it is easy to check the progress of your algorithm by hand.

### Problem 3.6: Initialize the dictionary.

dd a method to your Simplex solver that takes in arrays  $c$ ,  $A$ , and  $b$  to create the initial dictionary ( $D$ ) as a NumPy array.

### 3.1.1. Pivoting

---

Pivoting is the mechanism that really makes Simplex useful. Pivoting refers to the act of swapping dependent and independent variables, and transforming the dictionary appropriately. This has the effect of moving from one vertex of the feasible polytope to another vertex in a way that increases the value of the objective function. Depending on how you store your variables, you may need to modify a few different parts of your solver to reflect this swapping.

When initiating a pivot, you need to determine which variables will be swapped. In the dictionary representation, you first find a specific element on which to pivot, and the row and column that contain the pivot element correspond to the variables that need to be swapped. Row operations are then performed on the dictionary so that the pivot column becomes a negative elementary vector.

Let's break it down, starting with the pivot selection. We need to use some care when choosing the pivot element. To find the pivot column, search from left to right along the top row of the dictionary (ignoring the first column), and stop once you encounter the first negative value. The index corresponding to this column will be designated the *entering index*, since after the full pivot operation, it will enter the basis and become a dependent variable.

Using our initial dictionary  $D$  in the example, we stop at the second column:

$$D = \left[ \begin{array}{c|ccccc} 0 & -3 & -2 & 0 & 0 & 0 \\ 2 & -1 & 1 & -1 & 0 & 0 \\ 5 & -3 & -1 & 0 & -1 & 0 \\ 7 & -4 & -3 & 0 & 0 & -1 \end{array} \right]$$

We now know that our pivot element will be found in the second column. The entering index is thus 1.

Next, we select the pivot element from among the negative entries in the pivot column (ignoring the entry in the first row). *If all entries in the pivot column are non-negative, the problem is unbounded and has no solution.* In this case, the algorithm should terminate. Otherwise, assuming our pivot column is the  $j$ th column of the dictionary and that the negative entries of this column are  $D_{i_1,j}, D_{i_2,j}, \dots, D_{i_k,j}$ , we calculate the ratios

$$\frac{-D_{i_1,j}}{D_{i_1,j}}, \frac{-D_{i_2,j}}{D_{i_2,j}}, \dots, \frac{-D_{i_k,j}}{D_{i_k,j}},$$

and we choose our pivot element to be one that minimizes this ratio. If multiple entries minimize the ratio, then we utilize *Bland's Rule*, which instructs us to choose the entry in the row corresponding to the smallest index (obeying this rule is important, as it prevents the possibility of the algorithm cycling back on itself infinitely). The index corresponding to the pivot row is designated as the *leaving index*, since after the full pivot operation, it will leave the basis and become an independent variable.

In our example, we see that all entries in the pivot column (ignoring the entry in the first row, of course) are negative, and hence they are all potential choices for the pivot element. We then calculate the ratios, and obtain

$$\frac{-2}{-1} = 2, \quad \frac{-5}{-3} = 1.66\dots, \quad \frac{-7}{-4} = 1.75.$$

We see that the entry in the third row minimizes these ratios. Hence, the element in the second column (index 1), third row (index 2) is our designated pivot element.

$$D = \begin{bmatrix} 0 & -3 & -2 & 0 & 0 & 0 \\ 2 & -1 & 1 & -1 & 0 & 0 \\ 5 & \boxed{-3} & -1 & 0 & -1 & 0 \\ 7 & -4 & -3 & 0 & 0 & -1 \end{bmatrix}$$

**Definition 3.7: Bland's Rule**

choose the independent variable with the smallest index that has a negative coefficient in the objective function as the leaving variable. Choose the dependent variable with the smallest index among all the binding dependent variables.

Bland's Rule is important in avoiding cycles when performing pivots. This rule guarantees that a feasible Simplex problem will terminate in a finite number of pivots.

Finally, we perform row operations on our dictionary in the following way: divide the pivot row by the negative value of the pivot entry. Then use the pivot row to zero out all entries in the pivot column above and below the pivot entry. In our example, we first divide the pivot row by -3, and then zero out the two entries above the pivot element and the single entry below it:

$$\begin{array}{c} \left[ \begin{array}{cccccc} 0 & -3 & -2 & 0 & 0 & 0 \\ 2 & -1 & 1 & -1 & 0 & 0 \\ 5 & -3 & -1 & 0 & -1 & 0 \\ 7 & -4 & -3 & 0 & 0 & -1 \end{array} \right] \rightarrow \left[ \begin{array}{cccccc} 0 & -3 & -2 & 0 & 0 & 0 \\ 2 & -1 & 1 & -1 & 0 & 0 \\ 5/3 & -1 & -1/3 & 0 & -1/3 & 0 \\ 7 & -4 & -3 & 0 & 0 & -1 \end{array} \right] \rightarrow \\ \left[ \begin{array}{cccccc} -5 & 0 & -1 & 0 & 1 & 0 \\ 2 & -1 & 1 & -1 & 0 & 0 \\ 5/3 & -1 & -1/3 & 0 & -1/3 & 0 \\ 7 & -4 & -3 & 0 & 0 & -1 \end{array} \right] \rightarrow \left[ \begin{array}{cccccc} -5 & 0 & -1 & 0 & 1 & 0 \\ 1/3 & 0 & -4/3 & 1 & -1/3 & 0 \\ 5/3 & -1 & -1/3 & 0 & -1/3 & 0 \\ 7 & -4 & -3 & 0 & 0 & -1 \end{array} \right] \rightarrow \\ \left[ \begin{array}{cccccc} -5 & 0 & -1 & 0 & 1 & 0 \\ 1/3 & 0 & 4/3 & -1 & 1/3 & 0 \\ 5/3 & -1 & -1/3 & 0 & -1/3 & 0 \\ 1/3 & 0 & -5/3 & 0 & 4/3 & -1 \end{array} \right]. \end{array}$$

The result of these row operations is our updated dictionary, and the pivot operation is complete.

**Problem 3.8: Pivoting**

Add a method to your solver that checks for unboundedness and performs a single pivot operation from start to completion. If the problem is unbounded, raise a `ValueError`.

### 3.1.2. Termination and Reading the Dictionary

---

Up to this point, our algorithm accepts a linear program, adds slack variables, and creates the initial dictionary. After carrying out these initial steps, it then performs the pivoting operation iteratively until the optimal point is found. But how do we determine when the optimal point is found? The answer is to look at the top row of the dictionary, which represents the objective function. More specifically, before each pivoting operation, check whether all of the entries in the top row of the dictionary (ignoring the entry in the first column) are nonnegative. If this is the case, then we have found an optimal solution, and so we terminate the algorithm.

The final step is to report the solution. The ending state of the dictionary and index list tells us everything we need to know. The minimal value attained by the objective function is found in the upper leftmost entry of the dictionary. The dependent variables all have the value 0 in the objective function or first row of our dictionary array. The independent variables have values given by the first column of the dictionary. Specifically, the independent variable whose index is located at the  $i$ th entry of the index list has the value  $T_{i+1,0}$ .

In our example, suppose that our algorithm terminates with the dictionary and index list in the following state:

$$D = \begin{bmatrix} -5.2 & 0 & 0 & 0 & 0.2 & 0.6 \\ 0.6 & 0 & 0 & -1 & 1.4 & -0.8 \\ 1.6 & -1 & 0 & 0 & -0.6 & 0.2 \\ 0.2 & 0 & -1 & 0 & 0.8 & -0.6 \end{bmatrix}$$

Then the minimal value of the objective function is  $-5.2$ . The independent variables have indices 4, 5 and have the value 0. The dependent variables have indices 3, 1, and 2, and have values  $.6, 1.6$ , and  $.2$ , respectively. In the notation of the original problem statement, the solution is given by

$$x_0 = 1.6$$

$$x_1 = .2.$$

#### Problem 3.9: SimplexSolvers.solve()

*Write an additional method in your solver called `solve()` that obtains the optimal solution, then returns the minimal value, the dependent variables, and the independent variables. The dependent and independent variables should be represented as two dictionaries that map the index of the variable to its corresponding value.*

*For our example, we would return the tuple*

*( $-5.2$ , {0: 1.6, 1: .2, 2: .6}, {3: 0, 4: 0}).*

At this point, you should have a Simplex solver that is ready to use. The following code demonstrates how your solver is expected to behave:

```

>>> import SimplexSolver

# Initialize objective function and constraints.
>>> c = np.array([-3., -2.])
>>> b = np.array([2., 5, 7])
>>> A = np.array([[1., -1], [3, 1], [4, 3]])

# Instantiate the simplex solver, then solve the problem.
>>> solver = SimplexSolver(c, A, b)
>>> sol = solver.solve()
>>> print(sol)
(-5.2,
 {0: 1.6, 1: 0.2, 2: 0.6},
 {3: 0, 4: 0})

```

If the linear program were infeasible at the origin or unbounded, we would expect the solver to alert the user by raising an error.

Note that this simplex solver is *not* fully operational. It can't handle the case of infeasibility at the origin. This can be fixed by adding methods to your class that solve the *auxiliary problem*, that of finding an initial feasible dictionary when the problem is not feasible at the origin. Solving the auxiliary problem involves pivoting operations identical to those you have already implemented, so adding this functionality is not overly difficult.

## 3.2 The Product Mix Problem

---

We now use our Simplex implementation to solve the *product mix problem*, which in its dependent form can be expressed as a simple linear program. Suppose that a manufacturer makes  $n$  products using  $m$  different resources (labor, raw materials, machine time available, etc). The  $i$ th product is sold at a unit price  $p_i$ , and there are at most  $m_j$  units of the  $j$ th resource available. Additionally, each unit of the  $i$ th product requires  $a_{j,i}$  units of resource  $j$ . Given that the demand for product  $i$  is  $d_i$  units per a certain time period, how do we choose the optimal amount of each product to manufacture in that time period so as to maximize revenue, while not exceeding the available resources?

Let  $x_1, x_2, \dots, x_n$  denote the amount of each product to be manufactured. The sale of product  $i$  brings revenue in the amount of  $p_i x_i$ . Therefore our objective function, the profit, is given by

$$\sum_{i=1}^n p_i x_i.$$

Additionally, the manufacture of product  $i$  requires  $a_{j,i} x_i$  units of resource  $j$ . Thus we have the resource constraints

$$\sum_{i=1}^n a_{j,i} x_i \leq m_j \text{ for } j = 1, 2, \dots, m.$$

Finally, we have the demand constraints which tell us not to exceed the demand for the products:

$$x_i \leq d_i \text{ for } i = 1, 2, \dots, n$$

The variables  $x_i$  are constrained to be nonnegative, of course. We therefore have a linear program in the appropriate form that is feasible at the origin. It is a simple task to solve the problem using our Simplex solver.

### Problem 3.10: Product mix problem.

Solve the product mix problem for the data contained in the file `productMix.npz`. In this problem, there are 4 products and 3 resources. The archive file, which you can load using the function `np.load`, contains a dictionary of arrays. The array with key '`A`' gives the resource coefficients  $a_{i,j}$  (i.e. the  $(i, j)$ -th entry of the array give  $a_{i,j}$ ). The array with key '`p`' gives the unit prices  $p_i$ . The array with key '`m`' gives the available resource units  $m_j$ . The array with key '`d`' gives the demand constraints  $d_i$ .

Report the number of units that should be produced for each product. Hint: Because this is a maximization problem and your solver works with minimizations, you will need to change the sign of the array  $c$ .

## Beyond Simplex

---

The *Computing in Science and Engineering* journal listed Simplex as one of the top ten algorithms of the twentieth century [Nash2000]. However, like any other algorithm, Simplex has its drawbacks.

In 1972, Victor Klee and George Minty Cube published a paper with several examples of worst-case polytopes for the Simplex algorithm [Klee1972]. In their paper, they give several examples of polytopes that the Simplex algorithm struggles to solve.

Consider the following linear program from Klee and Minty.

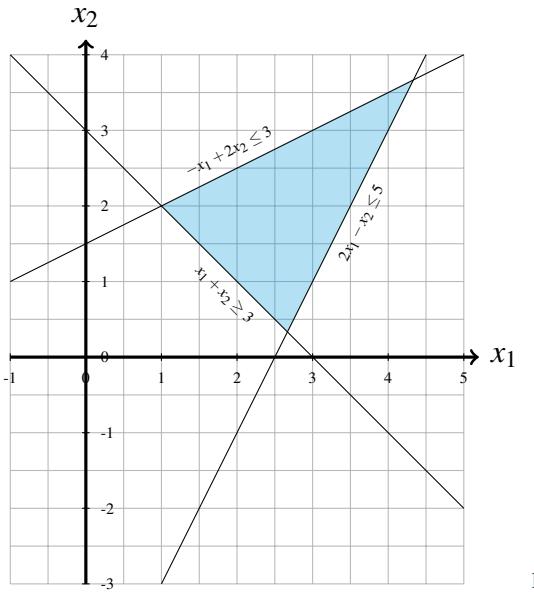
$$\begin{array}{lllll} \max & 2^{n-1}x_1 & +2^{n-2}x_2 & +\cdots & +2x_{n-1} & +x_n \\ \text{subject to } x_1 & & & & & \leq 5 \\ 4x_1 & & +x_2 & & & \leq 25 \\ 8x_1 & & +4x_2 & & +x_3 & \leq 125 \\ \vdots & & & & & \vdots \\ 2^n x_1 & & +2^{n-1}x_2 & & +\cdots & +4x_{n-1} & +x_n \leq 5 \end{array}$$

Klee and Minty show that for this example, the worst case scenario has exponential time complexity. With only  $n$  constraints and  $n$  variables, the simplex algorithm goes through  $2^n$  iterations. This is because there are  $2^n$  extreme points, and when starting at the point  $x = 0$ , the simplex algorithm goes through all of the extreme points before reaching the optimal point  $(0, 0, \dots, 0, 5^n)$ . Other algorithms, such as interior point methods, solve this problem much faster because they are not constrained to follow the edges.



# 4. Linear Programming Notes - Hildebrand

---



1

## 4.0.0.1. Simplex Tableau Pivot

---

[http://www.tutor-homework.com/Simplex\\_Tableau\\_Homework\\_Help.html](http://www.tutor-homework.com/Simplex_Tableau_Homework_Help.html)

## 4.0.0.2. Videos

---

For some nice videos of doing simplex method with tableaus, I recommend:

<https://www.youtube.com/watch?v=M8P0tpPtQZc>

LPP using simplex method [Minimization with 3 variables]: <https://youtu.be/SNc9NGCJmns>

LPP using Dual simplex method : <https://youtu.be/KLHWtBpPbEc>

LPP using TWO PHASE method: <https://youtu.be/zJhncZ5XUSU>

LPP using BIG M method: <https://youtu.be/MZ843VviaOA>

[1] LPP using Graphical method [Maximization with 2 constraints]: <https://youtu.be/8IRrgDoV8Eo>

[2] LPP using Graphical method [Minimization with 3 constraints]: [https://youtu.be/06Q03J\\_85as](https://youtu.be/06Q03J_85as)

---

<sup>1</sup><https://tex.stackexchange.com/questions/75933/how-to-draw-the-region-of-inequality>

## 4.1 Linear Programming Forms

---

## 4.2 Linear Programming Dual

---

Consider the linear program in standard form. The dual is the following problem

**Dual of LP in Standard Form:**

*Polynomial time (P)*

**Primal**

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

**Dual**

$$\begin{aligned} \min \quad & b^\top y \\ \text{s.t.} \quad & A^\top y \geq c \\ & y \text{ free} \end{aligned} \tag{4.1}$$

## 4.3 Weak and Strong Duality

---

**Theorem 4.1: Weak Duality**

Let  $x$  be feasible for the primal LP and  $y$  feasible for the dual LP. Then

$$c^\top x \leq b^\top y. \tag{4.1}$$

**Theorem 4.2: Strong Duality**

The primal LP is feasible and has a bounded objective value if and only if the dual LP is also feasible and has a bounded objective value. In this case, the optimal values to both problems coincide.

In particular, suppose  $x^*$  is optimal for the primal LP and  $y^*$  is optimal for the dual LP.

Then

$$c^\top x^* = b^\top y^*. \tag{4.2}$$

### 4.3.1. Reduced Costs

---

Consider the LP in standard form (2.2) given by

$$\begin{aligned} \max \quad & c^\top x \\ \text{s.t.} \quad & Ax = b \\ & x \geq 0 \end{aligned} \tag{4.3}$$

A basis  $B$  is a subset of the columns of  $A$  that form an invertible matrix. The remaining columns for the matrix  $N$ , that is,  $A = (B|N)$  (after permuting the columns of  $A$ ).

The *basic variables*  $x_B$  are the variables corresponding to the columns of  $B$  and the *non-basic variables* are those corresponding to the columns of  $N$ .

Since  $B$  is invertible we can convert the formulation by multiplying through by  $B^{-1}$ . This produces

$$\begin{aligned} \max \quad & c^\top x \\ \text{s.t.} \quad & B^{-1}Ax = B^{-1}b \\ & x \geq 0 \end{aligned} \tag{4.4}$$

Since  $A = (B|N)$ , we have

$$\begin{aligned} \max \quad & c^\top x \\ \text{s.t.} \quad & (B^{-1}B, B^{-1}N)x = B^{-1}b \\ & x \geq 0 \end{aligned} \tag{4.5}$$

which becomes

$$\begin{aligned} \max \quad & c^\top x \\ \text{s.t.} \quad & x_B + B^{-1}N x_N = B^{-1}b \\ & x \geq 0 \end{aligned} \tag{4.6}$$

$B^{-1}b \geq 0$ , then  $x_B = B^{-1}b, x_N = 0$  called a *basic feasible solution*.

Manipulating the formulation again, we can multiply the equations by  $c_B$  and subtract that from the objective function. This leaves us with

$$\begin{aligned} \max \quad & c_N x_N - c_B B^{-1} N x_N + c_B B^{-1} b \\ \text{s.t.} \quad & x_B + B^{-1} N x_N = B^{-1} b \\ & x \geq 0 \end{aligned} \tag{4.7}$$

combining terms creates

$$\begin{aligned} \max \quad & (c_N - c_B B^{-1} N) x_N + c_B B^{-1} b \\ \text{s.t.} \quad & x_B + B^{-1} N x_N = B^{-1} b \\ & x \geq 0 \end{aligned} \tag{4.8}$$

Now clearly we see that if  $B^{-1}b \geq 0$ , then setting  $x = (x_B, x_N) = (B^{-1}b, 0)$  is a feasible solution with objective value  $c_B B^{-1}b$ .

### 4.3.2. Tableau Based Pivoting

---

#### Problem 3

---

$$\text{Minimize } Z = 2x_1 + 3x_2$$

$$\begin{aligned} \text{s.t.} \quad & 2x_1 + x_2 \leq 16 \\ & x_1 + 3x_2 \geq 20 \\ & x_1 + x_2 = 10 \\ & x_1, x_2 \geq 0 \end{aligned} \Rightarrow$$

$$\text{Maximize } -Z = -2x_1 - 3x_2 - M\bar{x}_5 - M\bar{x}_6$$

$$\begin{aligned} \text{s.t.} \quad & 2x_1 + x_2 + x_3 = 16 \\ & x_1 + 3x_2 - x_4 + \bar{x}_5 = 20 \\ & x_1 + x_2 + \bar{x}_6 = 10 \\ & x_1, x_2, x_3, x_4, \bar{x}_5, \bar{x}_6 \geq 0 \end{aligned}$$

Initial Set-up

Basic Variable	Z	Coefficient of:						Right Side
		$x_1$	$x_2$	$x_3$	$x_4$	$\bar{x}_5$	$\bar{x}_6$	
Z	-1	2	3	0	0	M	M	0
$x_3$	0	2	1	1	0	0	0	16
$\bar{x}_5$	0	1	3	0	-1	1	0	20
$\bar{x}_6$	0	1	1	0	0	0	1	10

Standard Form

Basic Variable	Z	Coefficient of:						Right Side
		$x_1$	$x_2$	$x_3$	$x_4$	$\bar{x}_5$	$\bar{x}_6$	
Z	-1	2 - 2M	3 - 4M	0	M	0	0	-30M
$x_3$	0	2	1	1	0	0	0	16
$\bar{x}_5$	0	1	3	0	-1	1	0	20
$\bar{x}_6$	0	1	1	0	0	0	1	10

Iteration 1 - Let  $x_2$  enter and  $\bar{x}_5$  leaves.

Basic Variable	Z	Coefficient of:						Right Side
		$x_1$	$x_2$	$x_3$	$x_4$	$\bar{x}_5$	$\bar{x}_6$	
Z	-1	1 - 2M/3	0	0	1 - M/3	-1 + 4M/3	0	-20 - 10M/3
$x_3$	0	2/3	0	1	1/3	-1/3	0	-28/3
$x_2$	0	1/3	1	0	-1/3	1/3	0	20/3
$\bar{x}_6$	0	2/3	0	0	1/3	-1/3	1	10/3

Iteration 2 - Let  $x_1$  enter and  $\bar{x}_6$  leaves.

Basic Variable	Z	Coefficient of:						Right Side
		$x_1$	$x_2$	$x_3$	$x_4$	$\bar{x}_5$	$\bar{x}_6$	
Z	-1	0	0	0	1/2	-1/2 + M	-3/2 + M	-25
$x_3$	0	0	0	1	-1/2	1/2	-5/2	1
$x_2$	0	0	1	0	-1/2	1/2	-1/2	5
$x_1$	0	1	0	0	1/2	-1/2	3/2	5

We have reached an optimal solution since all coefficients in the objective function are positive. Thus our solution to the initial minimization problem is

$$x_1 = 5, \quad x_2 = 5, \quad Z(5,5) = 25 \quad \text{with slack } x_3 = 1$$



# 5. Linear Programming Book - Cheung

---

## Preface

---

This book covers the fundamentals of linear programming through studying systems of linear inequalities using only basic facts from linear algebra. It is suitable for a crash course on linear programming that emphasizes theoretical aspects of the subject. Discussion on practical solution methods such as the simplex method and interior point methods, though not present in this book, is planned for a future book.

Two excellent references for further study are [**Bertsimas:1997**] and [**Schrijver:1986**].



The book is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License.

## Notation

---

The set of real numbers is denoted by  $\mathbb{R}$ . The set of rational numbers is denoted by  $\mathbb{Q}$ . The set of integers is denoted by  $\mathbb{Z}$ .

The set of  $n$ -tuples with real entries is denoted by  $\mathbb{R}^n$ . Similar definitions hold for  $\mathbb{Q}^n$  and  $\mathbb{Z}^n$ .

The set of  $m \times n$  matrices (that is, matrices with  $m$  rows and  $n$  columns) with real entries is denoted  $\mathbb{R}^{m \times n}$ . Similar definitions hold for  $\mathbb{Q}^{m \times n}$  and  $\mathbb{Z}^{m \times n}$ .

All  $n$ -tuples are written as columns (that is, as  $n \times 1$  matrices). An  $n$ -tuple is normally represented by a lowercase Roman letter in boldface; for example,  $\mathbf{x}$ . For an  $n$ -tuple  $\mathbf{x}$ ,  $x_i$  denotes the  $i$ th entry (or component) of  $\mathbf{x}$  for  $i = 1, \dots, n$ .

Matrices are normally represented by an uppercase Roman letter in boldface; for example,  $\mathbf{A}$ . The  $j$ th column of a matrix  $\mathbf{A}$  is denoted by  $A_j$  and the  $(i, j)$ -entry (that is, the entry in row  $i$  and column  $j$ ) is denoted by  $a_{ij}$ .

Scalars are usually represented by lowercase Greek letters; for example,  $\lambda, \alpha, \beta$  etc.

An  $n$ -tuple consisting of all zeros is denoted by  $0$ . The dimension of the tuple is inferred from the context.

For a matrix  $\mathbf{A}$ ,  $\mathbf{A}^\top$  denotes the transpose of  $\mathbf{A}$ . For an  $n$ -tuple  $\mathbf{x}$ ,  $\mathbf{x}^\top$  denotes the transpose of  $\mathbf{x}$ .

If  $\mathbf{A}$  and  $\mathbf{B}$  are  $m \times n$  matrices,  $\mathbf{A} \geq \mathbf{B}$  means  $a_{ij} \geq b_{ij}$  for all  $i = 1, \dots, m$ ,  $j = 1, \dots, n$ . Similar definitions hold for  $\mathbf{A} \leq \mathbf{B}$ ,  $\mathbf{A} = \mathbf{B}$ ,  $\mathbf{A} < \mathbf{B}$  and  $\mathbf{A} > \mathbf{B}$ . In particular, if  $\mathbf{u}$  and  $\mathbf{v}$  are  $n$ -tuples,  $\mathbf{u} \geq \mathbf{v}$  means  $u_i \geq v_i$  for

$i = 1, \dots, n$  and  $\mathbf{u} > 0$  means  $u_i > 0$  for  $i = 1, \dots, n$ .

Superscripts in brackets are used for indexing tuples. For example, we can write  $\mathbf{u}^{(1)}, \mathbf{u}^{(2)} \in \mathbb{R}^3$ . Then  $\mathbf{u}^{(1)}$  and  $\mathbf{u}^{(2)}$  are elements of  $\mathbb{R}^3$ . The second entry of  $\mathbf{u}^{(1)}$  is denoted by  $u_2^{(1)}$ .

## 5.1 Graphical example

---

To motivate the subject of linear programming (LP), we begin with a planning problem that can be solved graphically.

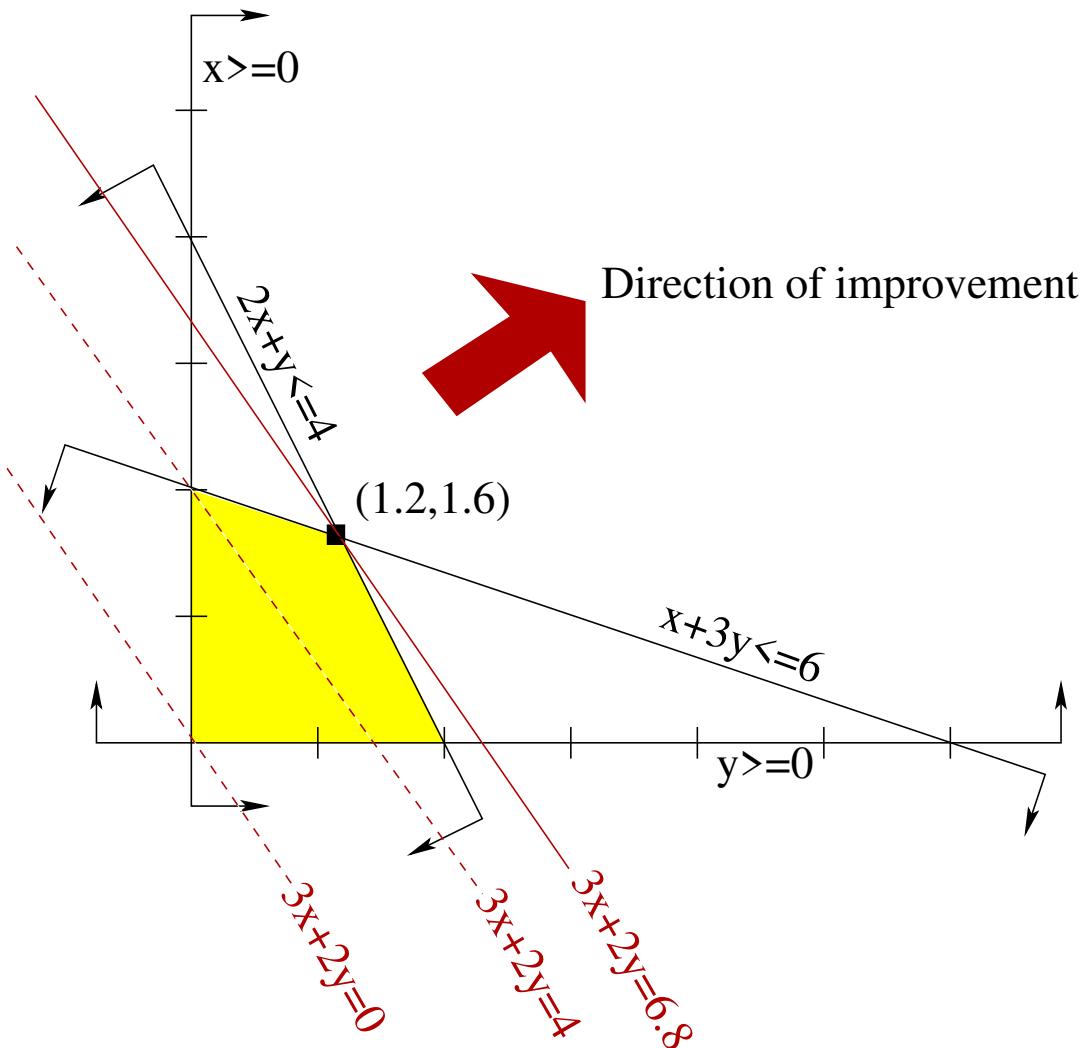
Say you are a vendor of lemonade and lemon juice. Each unit of lemonade requires 1 lemon and 2 litres of water. Each unit of lemon juice requires 3 lemons and 1 litre of water. Each unit of lemonade gives a profit of three dollars. Each unit of lemon juice gives a profit of two dollars. You have 6 lemons and 4 litres of water available. How many units of lemonade and lemon juice should you make to maximize profit?

If we let  $x$  denote the number of units of lemonade to be made and let  $y$  denote the number of units of lemon juice to be made, then the profit is given by  $3x + 2y$  dollars. We call  $3x + 2y$  the objective function. Note that there are a number of constraints that  $x$  and  $y$  must satisfy. First of all,  $x$  and  $y$  should be nonnegative. The number of lemons needed to make  $x$  units of lemonade and  $y$  units of lemon juice is  $x + 3y$  and cannot exceed 6. The number of litres of water needed to make  $x$  units of lemonade and  $y$  units of lemon juice is  $2x + y$  and cannot exceed 4. Hence, to determine the maximum profit, we need to maximize  $3x + 2y$  subject to  $x$  and  $y$  satisfying the constraints  $x + 3y \leq 6$ ,  $2x + y \leq 4$ ,  $x \geq 0$ , and  $y \geq 0$ .

A more compact way to write the problem is as follows:

$$\begin{aligned} &\text{maximize} && 3x + 2y \\ &\text{subject to} && x + 3y \leq 6 \\ & && 2x + y \leq 4 \\ & && x \geq 0 \\ & && y \geq 0. \end{aligned}$$

We can solve this maximization problem graphically as follows. We first sketch the set of  $\begin{bmatrix} x \\ y \end{bmatrix}$  satisfying the constraints, called the feasible region, on the  $(x, y)$ -plane. We then take the objective function  $3x + 2y$  and turn it into an equation of a line  $3x + 2y = z$  where  $z$  is a parameter. Note that as the value of  $z$  increases, the line defined by the equation  $3x + 2y = z$  moves in the direction of the normal vector  $\begin{bmatrix} 3 \\ 2 \end{bmatrix}$ . We call this direction the direction of improvement. Determining the maximum value of the objective function, called the optimal value, subject to the constraints amounts to finding the maximum value of  $z$  so that the line defined by the equation  $3x + 2y = z$  still intersects the feasible region.



In the figure above, the lines with  $z$  at 0, 4 and 6.8 have been drawn. From the picture, we can see that if  $z$  is greater than 6.8, the line defined by  $3x + 2y = z$  will not intersect the feasible region. Hence, the profit cannot exceed 6.8 dollars.

As the line  $3x + 2y = 6.8$  does intersect the feasible region, 6.8 is the maximum value for the objective function. Note that there is only one point in the feasible region that intersects the line  $3x + 2y = 6.8$ , namely  $\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1.2 \\ 1.6 \end{bmatrix}$ . In other words, to maximize profit, we want to make 1.2 units of lemonade and 1.6 units of lemon juice.

The above solution method can hardly be regarded as rigorous because we relied on a picture to conclude that  $3x + 2y \leq 6.8$  for all  $\begin{bmatrix} x \\ y \end{bmatrix}$  satisfying the constraints. But we can actually show this *algebraically*.

Note that multiplying both sides of the constraint  $x + 3y \leq 6$  gives  $0.2x + 0.6y \leq 1.2$ , and multiplying both sides of the constraint  $2x + y \leq 4$  gives  $2.8x + 1.4y \leq 5.6$ . Hence, any  $\begin{bmatrix} x \\ y \end{bmatrix}$  that satisfies both  $x + 3y \leq 6$  and  $2x + y \leq 4$  must also satisfy  $(0.2x + 0.6y) + (2.8x + 1.4y) \leq 1.2 + 5.6$ , which simplifies to  $3x + 2y \leq 6.8$  as desired! (Here, we used the fact that if  $a \leq b$  and  $c \leq d$ , then  $a + c \leq b + d$ .)

Now, one might ask if it is always possible to find an algebraic proof like the one above for similar problems. If the answer is yes, how does one find such a proof? We will see answers to this question later on.

Before we end this segment, let us consider the following problem:

$$\begin{array}{ll} \text{minimize} & -2x + y \\ \text{subject to} & -x + y \leq 3 \\ & x - 2y \leq 2 \\ & x \geq 0 \\ & y \geq 0. \end{array}$$

Note that for any  $t \geq 0$ ,  $\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} t \\ t \end{bmatrix}$  satisfies all the constraints. The value of the objective function at  $\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} t \\ t \end{bmatrix}$  is  $-t$ . As  $t \rightarrow \infty$ , the value of the objective function tends to  $-\infty$ . Therefore, there is no minimum value for the objective function. The problem is said to be unbounded. Later on, we will see how to detect unboundedness algorithmically.

As an exercise, check that unboundedness can also be established by using  $\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 2t+2 \\ t \end{bmatrix}$  for  $t \geq 0$ .

## Exercises

---

1. Sketch all  $\begin{bmatrix} x \\ y \end{bmatrix}$  satisfying

$$x - 2y \leq 2$$

on the  $(x,y)$ -plane.

2. Determine the optimal value of

$$\begin{array}{ll} \text{Minimize} & x + y \\ \text{Subject to} & 2x + y \geq 4 \\ & x + 3y \geq 1. \end{array}$$

3. Show that the problem

$$\begin{array}{ll} \text{Minimize} & -x + y \\ \text{Subject to} & 2x - y \geq 0 \\ & x + 3y \geq 3 \end{array}$$

is unbounded.

4. Suppose that you are shopping for dietary supplements to satisfy your required daily intake of 0.40mg of nutrient  $M$  and 0.30mg of nutrient  $N$ . There are three popular products on the market. The costs and the amounts of the two nutrients are given in the following table:

	Product 1	Product 2	Product 3
Cost	\$27	\$31	\$24
Daily amount of $M$	0.16 mg	0.21 mg	0.11 mg
Daily amount of $N$	0.19 mg	0.13 mg	0.15 mg

You want to determine how much of each product you should buy so that the daily intake requirements of the two nutrients are satisfied at minimum cost. Formulate your problem as a linear programming problem, assuming that you can buy a fractional number of each product.

## Solutions

---

1. The points  $(x, y)$  satisfying  $x - 2y \leq 2$  are precisely those above the line passing through  $(2, 0)$  and  $(0, -1)$ .
2. We want to determine the minimum value  $z$  so that  $x + y = z$  defines a line that has a nonempty intersection with the feasible region. However, we can avoid referring to a sketch by setting  $x = z - y$  and substituting for  $x$  in the inequalities to obtain:

$$\begin{aligned} 2(z - y) + y &\geq 4 \\ (z - y) + 3y &\geq 1, \end{aligned}$$

or equivalently,

$$\begin{aligned} z &\geq 2 + \frac{1}{2}y \\ z &\geq 1 - 2y, \end{aligned}$$

Thus, the minimum value for  $z$  is  $\min\{2 + \frac{1}{2}y, 1 - 2y\}$ , which occurs at  $y = -\frac{9}{5}$ . Hence, the optimal value is  $\frac{9}{5}$ .

We can verify our work by doing the following. If our calculations above are correct, then an optimal solution is given by  $x = \frac{11}{5}$ ,  $y = -\frac{2}{5}$  since  $x = z - y$ . It is easy to check that this satisfies both inequalities and therefore is a feasible solution.

Now, taking  $\frac{2}{5}$  times the first inequality and  $\frac{1}{5}$  times the second inequality, we can infer the inequality  $x + y \geq \frac{9}{5}$ . The left-hand side of this inequality is precisely the objective function. Hence, no feasible solution can have objective function value less than  $\frac{9}{5}$ . But  $x = \frac{11}{5}$ ,  $y = -\frac{2}{5}$  is a feasible solution with objective function value equal to  $\frac{9}{5}$ . As a result, it must be an optimal solution.

**Remark.** We have not yet discussed how to obtain the multipliers  $\frac{2}{5}$  and  $\frac{1}{5}$  for inferring the inequality  $x + y \geq \frac{9}{5}$ . This is an issue that will be taken up later. In the meantime, think about how one could have obtained these multipliers for this particular exercise.

3. We could glean some insight by first making a sketch on the  $(x,y)$ -plane.

The line defined by  $-x + y = z$  has  $x$ -intercept  $-z$ . Note that for  $z \leq -3$ ,  $\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} -z \\ 0 \end{bmatrix}$  satisfies both inequalities and the value of the objective function at  $\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} -z \\ 0 \end{bmatrix}$  is  $z$ . Hence, there is no lower bound on the value of objective function.

4. Let  $x_i$  denote the amount of Product  $i$  to buy for  $i = 1, 2, 3$ . Then, the problem can be formulated as

$$\begin{aligned} & \text{minimize} && 27x_1 + 31x_2 + 24x_3 \\ & \text{subject to} && 0.16x_1 + 0.21x_2 + 0.11x_3 \geq 0.30 \\ & && 0.19x_1 + 0.13x_2 + 0.15x_3 \geq 0.40 \\ & && x_1, x_2, x_3 \geq 0. \end{aligned}$$

**Remark.** If one cannot buy fractional amounts of the products, the problem can be formulated as

$$\begin{aligned} & \text{minimize} && 27x_1 + 31x_2 + 24x_3 \\ & \text{subject to} && 0.16x_1 + 0.21x_2 + 0.11x_3 \geq 0.30 \\ & && 0.19x_1 + 0.13x_2 + 0.15x_3 \geq 0.40 \\ & && x_1, x_2, x_3 \geq 0. \\ & && x_1, x_2, x_3 \in \mathbb{Z}. \end{aligned}$$

## 5.2 Definitions

---

The following is an example of a problem in **linear programming**:

$$\begin{aligned} & \text{Maximize} && x + y - 2z \\ & \text{Subject to} && 2x + y + z \leq 4 \\ & && 3x - y + z = 0 \\ & && x, y, z \geq 0 \end{aligned}$$

**Solving** this problem means finding real values for the **variables**  $x, y, z$  satisfying the **constraints**  $2x + y + z \leq 4$ ,  $3x - y + z = 0$ , and  $x, y, z \geq 0$  that gives the maximum possible value (if it exists) for the **objective function**  $x + y - 2z$ .

For example,  $\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}$  satisfies all the constraints and is called a **feasible solution**. Its **objective**

**function value**, obtained by evaluating the objective function at  $\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}$ , is  $0 + 1 - 2(1) = -1$ . The set of feasible solutions to a linear programming problem is called the **feasible region**.

More formally, a linear programming problem is an optimization problem of the following form:

$$\begin{aligned} & \text{Maximize (or Minimize)} \quad \sum_{j=1}^n c_j x_j \\ & \text{Subject to} \quad P_i(x_1, \dots, x_n) \quad i = 1, \dots, m \end{aligned}$$

where  $m$  and  $n$  are positive integers,  $c_j \in \mathbb{R}$  for  $j = 1, \dots, n$ , and for each  $i = 1, \dots, m$ ,  $P_i(x_1, \dots, x_n)$  is a **linear constraint** on the **(decision) variables**  $x_1, \dots, x_n$  having one of the following forms:

- $a_1 x_1 + \dots + a_n x_n \geq \beta$
- $a_1 x_1 + \dots + a_n x_n \leq \beta$
- $a_1 x_1 + \dots + a_n x_n = \beta$

where  $\beta, a_1, \dots, a_n \in \mathbb{R}$ . To save writing, the word “Minimize” (“Maximize”) is replaced with “min” (“max”) and “Subject to” is abbreviated as “s.t.”.

A feasible solution  $\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$  that gives the maximum possible objective function value in the case of a maximization problem is called an **optimal solution** and its objective function value is the **optimal value** of the problem.

The following example shows that it is possible to have multiple optimal solutions:

$$\begin{aligned} & \max \quad x + y \\ & \text{s.t.} \quad 2x + 2y \leq 1 \end{aligned}$$

The constraint says that  $x + y$  cannot exceed  $\frac{1}{2}$ . Now, both  $\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \frac{1}{2} \\ 0 \end{bmatrix}$  and  $\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 0 \\ \frac{1}{2} \end{bmatrix}$  are feasible solutions having objective function value  $\frac{1}{2}$ . Hence, they are both optimal solutions. (In fact, this problem has infinitely many optimal solutions. Can you specify all of them?)

Not all linear programming problems have optimal solutions. For example, a problem can have no feasible solution. Such a problem is said to be **infeasible**. Here is an example of an infeasible problem:

$$\begin{aligned} & \min \quad x \\ & \text{s.t.} \quad x \leq 1 \\ & \quad x \geq 2 \end{aligned}$$

There is no value for  $x$  that is at the same time at most 1 and at least 2.

Even if a problem is not infeasible, it might not have an optimal solution as the following example shows:

$$\begin{aligned} & \min \quad x \\ & \text{s.t.} \quad x \leq 0 \end{aligned}$$

Note that now matter what real number  $M$  we are given, we can always find a feasible solution whose objective function value is less than  $M$ . Such a problem is said to be **unbounded**. (For a maximization problem, it is unbounded if one can find feasible solutions who objective function value is larger than any given real number.)

So far, we have seen that a linear programming problem can have an optimal solution, be infeasible, or be unbounded. Is it possible for a linear programming problem to be not infeasible, not unbounded, and with no optimal solution?

The following optimization problem, though not a linear programming problem, is not infeasible, not unbounded, and has no optimal solution:

$$\begin{aligned} \min \quad & 2^x \\ \text{s.t.} \quad & x \leq 0 \end{aligned}$$

The objective function value is never negative and can get arbitrarily close to 0 but can never attain 0.

A main result in linear programming states that if a linear programming problem is not infeasible and is not unbounded, then it must have an optimal solution. This result is known as the **Fundamental Theorem of Linear Programming** (Theorem 5.4) and we will see a proof of this important result. In the meantime, we will consider the seemingly easier problem of determining if a system of linear constraints has a solution.

## Exercises

---

1. Determine all values of  $a$  such that the problem

$$\begin{aligned} \min \quad & x + y \\ \text{s.t.} \quad & -3x + y \geq a \\ & 2x - y \geq 0 \\ & x + 2y \geq 2 \end{aligned}$$

is infeasible.

2. Show that the problem

$$\begin{aligned} \min \quad & 2^x \cdot 4^y \\ \text{s.t.} \quad & e^{-3x+y} \geq 1 \\ & |2x - y| \leq 4 \end{aligned}$$

can be solved by solving a linear programming problem.

## Solutions

---

1. Adding the first two inequalities gives  $-x \geq a$ . Adding 2 times the second inequality and the third inequality gives  $5x \geq 2$ , implying that  $x \geq \frac{2}{5}$ . Hence, if  $a > -\frac{2}{5}$ , there is no solution.

Note that if  $a \leq -\frac{2}{5}$ , then  $(x, y) = \left(\frac{2}{5}, \frac{4}{5}\right)$  satisfies all the inequalities. Hence, the problem is infeasible if and only if  $a > -\frac{2}{5}$ .

2. Note that the constraint  $|2x - y| \leq 4$  is equivalent to the constraints  $2x - y \leq 4$  and  $2x - y \geq -4$  taken together, and the constraint  $e^{-3x+y} \geq 1$  is equivalent to  $-3x + y \geq 0$ . Hence, we can rewrite the problem with linear constraints.

Finally, minimizing  $2^x \cdot 4^y$  is the same as minimizing  $2^{x+2y}$ , which is equivalent to minimizing  $x + 2y$ .

## 5.3 Farkas' Lemma

---

A well-known result in linear algebra states that a system of linear equations  $\mathbf{Ax} = \mathbf{b}$ , where  $\mathbf{A} \in \mathbb{R}^{m \times n}$ ,  $\mathbf{b} \in \mathbb{R}^m$ , and  $\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$  is a tuple of variables, has no solution if and only if there exists  $\mathbf{y} \in \mathbb{R}^m$  such that  $\mathbf{y}^\top \mathbf{A} = 0$  and  $\mathbf{y}^\top \mathbf{b} \neq 0$ .

It is easily seen that if such a  $\mathbf{y}$  exists, then the system  $\mathbf{Ax} = \mathbf{b}$  cannot have a solution. (Simply multiply both sides of  $\mathbf{Ax} = \mathbf{b}$  on the left by  $\mathbf{y}^\top$ .) However, proving the converse requires a bit of work. A standard elementary proof involves using Gauss-Jordan elimination to reduce the original system to an equivalent system  $\mathbf{Qx} = \mathbf{d}$  such that  $\mathbf{Q}$  has a row of zero, say in row  $i$ , with  $\mathbf{d}_i \neq 0$ . The process can be captured by a square matrix  $\mathbf{M}$  satisfying  $\mathbf{MA} = \mathbf{Q}$ . We can then take  $\mathbf{y}^\top$  to be the  $i$ th row of  $\mathbf{M}$ .

An analogous result holds for systems of linear inequalities. The following result is one of the many variants of a result known as the **Farkas' Lemma**:

### Theorem 5.1: Farkas' Lemma

With  $\mathbf{A}$ ,  $\mathbf{x}$ , and  $\mathbf{b}$  as above, the system  $\mathbf{Ax} \geq \mathbf{b}$  has no solution if and only if there exists  $\mathbf{y} \in \mathbb{R}^m$  such that

$$\mathbf{y} \geq 0, \mathbf{y}^\top \mathbf{A} = 0, \mathbf{y}^\top \mathbf{b} > 0.$$

In other words, the system  $\mathbf{Ax} \geq \mathbf{b}$  has no solution if and only if one can infer the inequality  $0 \geq \gamma$  for some  $\gamma > 0$  by taking a nonnegative linear combination of the inequalities.

This result essentially says that there is always a certificate (the  $m$ -tuple  $\mathbf{y}$  with the prescribed properties) for the infeasibility of the system  $\mathbf{Ax} \geq \mathbf{b}$ . This allows third parties to verify the claim of infeasibility without having to solve the system from scratch.

### Example 5.2

For the system

$$\begin{aligned} 2x - y + z &\geq 2 \\ -x + y - z &\geq 0 \\ -y + z &\geq 0, \end{aligned}$$

adding two times the second inequality and the third inequality to the first inequality gives  $0 \geq 2$ .

Hence,  $\mathbf{y} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}$  is a certificate of infeasibility for this example.

We now give a proof of Theorem 5.3. It is easy to see that if such a  $\mathbf{y}$  exists, then the system  $\mathbf{Ax} \geq \mathbf{b}$  has no solution.

## 5.4 Fundamental Theorem of Linear Programming

---

Having used Fourier-Motzkin elimination to solve a linear programming problem, we now will go one step further and use the same technique to prove the following important result.

### Theorem 5.3: Fundamental Theorem of Linear Programming

*For any given linear programming problem, exactly one of the following holds:*

1. *the problem is infeasible;*
2. *the problem is unbounded;*
3. *the problem has an optimal solution.*

*Proof.* Without loss of generality, we may assume that the linear programming problem is of the form

$$\begin{aligned} \min \quad & \mathbf{c}^T \mathbf{x} \\ \text{s.t.} \quad & \mathbf{A} \mathbf{x} \geq \mathbf{b} \end{aligned} \tag{5.1}$$

where  $m$  and  $n$  are positive integers,  $\mathbf{A} \in \mathbb{R}^{m \times n}$ ,  $\mathbf{b} \in \mathbb{R}^m$ ,  $\mathbf{c} \in \mathbb{R}^n$ , and  $\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$  is a tuple of variables.

Indeed, any linear programming problem can be converted to a linear programming problem in the form of (5.1) having the same feasible region and optimal solution set. To see this, note that a constraint of the form  $\mathbf{a}^T \mathbf{x} \leq \beta$  can be written as  $-\mathbf{a}^T \mathbf{x} \geq -\beta$ ; a constraint of the form  $\mathbf{a}^T \mathbf{x} = \beta$  written as a pair of constraints  $\mathbf{a}^T \mathbf{x} \geq \beta$  and  $-\mathbf{a}^T \mathbf{x} \geq -\beta$ ; and a maximization problem is equivalent to the problem that minimizes the negative of the objective function subject to the same constraints.

Suppose that (5.1) is not infeasible. Form the system

$$\begin{aligned} z - \mathbf{c}^T \mathbf{x} &\geq 0 \\ -z + \mathbf{c}^T \mathbf{x} &\geq 0 \\ \mathbf{A} \mathbf{x} &\geq \mathbf{b}. \end{aligned} \tag{5.2}$$

Solving (5.1) is equivalent to finding among all the solutions to (5.2) one that minimizes  $z$ , if it exists. Eliminating the variables  $x_1, \dots, x_n$  (in any order) using Fourier-Motzkin elimination gives a system of linear inequalities (S) containing at most the variable  $z$ . By scaling, we may assume that the each coefficient of  $z$  in (S) is 1,  $-1$ , or 0. Note that any  $z$  satisfying (S) can be extended to a solution to (5.2) and the  $z$  value from any solution to (5.2) must satisfy (S).

That (5.1) is not unbounded implies that (S) must contain an inequality of the form  $z \geq \beta$  for some  $\beta \in \mathbb{R}$ . (Why?) Let all the inequalities in which the coefficient of  $z$  is positive be

$$z \geq \beta_i$$

where  $\beta_i \in \mathbb{R}$  for  $i = 1, \dots, p$  for some positive integer  $p$ . Let  $\gamma = \max\{\beta_1, \dots, \beta_p\}$ . Then for any solution  $x, z$  to (5.2),  $z$  is at least  $\gamma$ . But we can set  $z = \gamma$  and extend it to a solution to (5.2). Hence, we obtain an optimal solution for (5.1) and  $\gamma$  is the optimal value. This completes the proof of the theorem.

□

**Remark.** We can construct multipliers to infer the inequality  $\mathbf{c}^\top \mathbf{x} \geq \gamma$  from the system  $\mathbf{Ax} \geq \mathbf{b}$ . Because we obtained the inequality  $z \geq \gamma$  using Fourier-Motzkin elimination, there must exist real numbers  $\alpha, \beta, y_1^*, \dots, y_m^* \geq 0$  such that

$$[\alpha \ \beta \ y_1^* \ \cdots \ y_m^*] \begin{bmatrix} 1 & -\mathbf{c}^\top \\ -1 & \mathbf{c}^\top \\ 0 & \mathbf{A} \end{bmatrix} \begin{bmatrix} z \\ \mathbf{x} \end{bmatrix} \geq [\alpha \ \beta \ y_1^* \ \cdots \ y_m^*] \begin{bmatrix} 0 \\ 0 \\ \mathbf{b} \end{bmatrix}$$

is identically  $z \geq \gamma$ . Note that we must have  $\alpha - \beta = 1$  and

$$\mathbf{y}^* \geq 0, \mathbf{y}^{*\top} \mathbf{A} = \mathbf{c}^\top, \text{ and } \mathbf{y}^{*\top} \mathbf{b} = \gamma$$

where  $\mathbf{y}^* = [y_1^*, \dots, y_m^*]^\top$ . Hence,  $y_1^*, \dots, y_m^*$  are the desired multipliers.,

The significance of the fact that we can infer  $\mathbf{c}^\top \mathbf{x} \geq \gamma$  where  $\gamma$  will be discussed in more details when we look at duality theory for linear programming.

## Exercises

---

1. Determine the optimal value of the following linear programming problem:

$$\begin{aligned} \min \quad & x \\ \text{s.t.} \quad & x + y \geq 2 \\ & x - 2y + z \geq 0 \\ & y - 2z \geq -1. \end{aligned}$$

2. Determine if the following linear programming problem has an optimal solution:

$$\begin{aligned} \min \quad & x_1 + 2x_2 \\ \text{s.t.} \quad & x_1 + 3x_2 \geq 4 \\ & -x_1 + x_2 \geq 0. \end{aligned}$$

3. A set  $S \subset \mathbb{R}^n$  is said to be bounded if there exists a real number  $M > 0$  such that for every  $\mathbf{x} \in S$ ,  $|x_i| < M$  for all  $i = 1, \dots, n$ . Prove that every linear programming problem with a bounded nonempty feasible region has an optimal solution.

## Solutions

---

1. The problem is equivalent to determining the minimum value for  $x$  among all  $x, y, z$  satisfying

$$\begin{aligned} x + y &\geq 2 & (1) \\ x - 2y + z &\geq 0 & (2) \\ y - 2z &\geq -1. & (3) \end{aligned}$$

We use Fourier-Motzkin Elimination Method to eliminate  $z$ . Multiplying (3) by  $\frac{1}{2}$ , we get

$$\begin{aligned} x + y &\geq 2 & (1) \\ x - 2y + z &\geq 0 & (2) \\ \frac{1}{2}y - z &\geq -\frac{1}{2}. & (4) \end{aligned}$$

Eliminating  $z$ , we obtain

$$\begin{aligned} x + y &\geq 2 & (1) \\ x - \frac{3}{2}y &\geq -\frac{1}{2} & (5) \end{aligned}$$

where (5) is given by (2) + (4).

Multiplying (5) by  $\frac{2}{3}$ , we get

$$\begin{aligned} x + y &\geq 2 & (1) \\ \frac{2}{3}x - y &\geq -\frac{1}{3} & (6) \end{aligned}$$

Eliminating  $y$ , we get

$$\frac{5}{3}x \geq \frac{5}{3} \quad (7)$$

where (7) is given by (1) + (6). Multiplying (7) by  $\frac{3}{5}$ , we obtain  $x \geq 1$ . Hence, the minimum possible value for  $x$  is 1.

Note that setting  $x = 1$ , the system (1) and (6) forces  $y = 1$ . And (2) and (3) together force  $z = 1$ . One can check that  $(x, y, z) = (1, 1, 1)$  is a feasible solution.

**Remark.** Note that the inequality  $x \geq 1$  is given by

$$\begin{aligned} \frac{3}{5}(7) &\Leftarrow \frac{3}{5}(1) + \frac{3}{5}(6) \\ &\Leftarrow \frac{3}{5}(1) + \frac{2}{5}(5) \\ &\Leftarrow \frac{3}{5}(1) + \frac{2}{5}(2) + \frac{2}{5}(4) \\ &\Leftarrow \frac{3}{5}(1) + \frac{2}{5}(2) + \frac{1}{5}(3) \end{aligned}$$

2. It suffices to determine if there exists a minimum value for  $z$  among all the solutions to the system

$$\begin{aligned} z - x_1 - 2x_2 &\geq 0 & (1) \\ -z + x_1 + 2x_2 &\geq 0 & (2) \\ x_1 + 3x_2 &\geq 4 & (3) \\ -x_1 + x_2 &\geq 0 & (4) \end{aligned}$$

Using Fourier-Motzkin elimination to eliminate  $x_1$ , we obtain:

$$\begin{aligned} (1) + (2) : \quad & 0 \geq 0 \\ (1) + (3) : \quad & z + x_2 \geq 4 \quad (5) \\ (2) + (4) : \quad & -z + 3x_2 \geq 0 \quad (6) \\ (3) + (4) : \quad & 4x_2 \geq 4 \quad (7) \end{aligned}$$

Note that all the coefficients of  $x_2$  is nonnegative. Hence, eliminating  $x_2$  will result in a system with no constraints. Therefore, there is no lower bound on the value of  $z$ . In particular, if  $z = t$  for  $t \leq 0$ , then from (5)–(6), we need  $x_2 \geq 4 - t$ ,  $3x_2 \geq t$ , and  $x_2 \geq 1$ . Hence, we can set  $x_2 = 4 - t$  and  $x_1 = -8 + 3t$ . This gives a feasible solution for all  $t \leq 0$  with objective function value that approaches  $-\infty$  as  $t \rightarrow -\infty$ . Hence, the linear programming problem is unbounded.

3. Let  $(P)$  denote a linear programming problem with a bounded nonempty feasible region with objective function  $\mathbf{c}^T \mathbf{x}$ . By assumption,  $(P)$  is not infeasible. Note that  $(P)$  is not unbounded because  $|\mathbf{c}^T \mathbf{x}| \leq \sum_i |c_i| |x_i| \leq M \sum_i |c_i|$ . Thus, by Theorem 5.4,  $(P)$  has an optimal solution.

## 5.5 Linear programming duality

---

Consider the following problem:

$$\begin{array}{ll} \min & \mathbf{c}^T \mathbf{x} \\ \text{s.t.} & \mathbf{A} \mathbf{x} \geq \mathbf{b}. \end{array} \quad (5.1)$$

In the remark at the end of Chapter ??, we saw that if (5.1) has an optimal solution, then there exists  $\mathbf{y}^* \in \mathbb{R}^m$  such that  $\mathbf{y}^* \geq 0$ ,  $\mathbf{y}^{*\top} \mathbf{A} = \mathbf{c}^T$ , and  $\mathbf{y}^{*\top} \mathbf{b} = \gamma$  where  $\gamma$  denotes the optimal value of (5.1).

Take any  $\mathbf{y} \in \mathbb{R}^m$  satisfying  $\mathbf{y} \geq 0$  and  $\mathbf{y}^T \mathbf{A} = \mathbf{c}^T$ . Then we can infer from  $\mathbf{A} \mathbf{x} \geq \mathbf{b}$  the inequality  $\mathbf{y}^T \mathbf{A} \mathbf{x} \geq \mathbf{y}^T \mathbf{b}$ , or more simply,  $\mathbf{c}^T \mathbf{x} \geq \mathbf{y}^T \mathbf{b}$ . Thus, for any such  $\mathbf{y}$ ,  $\mathbf{y}^T \mathbf{b}$  gives a lower bound for the objective function value of any feasible solution to (5.1). Since  $\gamma$  is the optimal value of  $(P)$ , we must have  $\gamma \geq \mathbf{y}^T \mathbf{b}$ .

As  $\mathbf{y}^{*\top} \mathbf{b} = \gamma$ , we see that  $\gamma$  is the optimal value of

$$\begin{array}{ll} \max & \mathbf{y}^T \mathbf{b} \\ \text{s.t.} & \mathbf{y}^T \mathbf{A} = \mathbf{c}^T \\ & \mathbf{y} \geq 0. \end{array} \quad (5.2)$$

Note that (5.2) is a linear programming problem! We call it the **dual problem** of the **primal problem** (5.1). We say that the dual variable  $y_i$  is **associated** with the constraint  $\mathbf{a}^{(i)\top} \mathbf{x} \geq b_i$  where  $\mathbf{a}^{(i)\top}$  denotes the  $i$ th row of  $\mathbf{A}$ .

In other words, we define the dual problem of (5.1) to be the linear programming problem (5.2). In the discussion above, we saw that if the primal problem has an optimal solution, then so does the dual problem and the optimal values of the two problems are equal. Thus, we have the following result:

**Theorem 5.4: strong-duality-special**

Suppose that (5.1) has an optimal solution. Then (5.2) also has an optimal solution and the optimal values of the two problems are equal.

At first glance, requiring all the constraints to be  $\geq$ -inequalities as in (5.1) before forming the dual problem seems a bit restrictive. We now see how the dual problem of a primal problem in general form can be defined. We first make two observations that motivate the definition.

**Observation 1**

Suppose that our primal problem contains a mixture of all types of linear constraints:

$$\begin{aligned} \min \quad & \mathbf{c}^T \mathbf{x} \\ \text{s.t.} \quad & \mathbf{Ax} \geq \mathbf{b} \\ & \mathbf{A}' \mathbf{x} \leq \mathbf{b}' \\ & \mathbf{A}'' \mathbf{x} = \mathbf{b}'' \end{aligned} \tag{5.3}$$

where  $\mathbf{A} \in \mathbb{R}^{m \times n}$ ,  $\mathbf{b} \in \mathbb{R}^m$ ,  $\mathbf{A}' \in \mathbb{R}^{m' \times n}$ ,  $\mathbf{b}' \in \mathbb{R}^{m'}$ ,  $\mathbf{A}'' \in \mathbb{R}^{m'' \times n}$ , and  $\mathbf{b}'' \in \mathbb{R}^{m''}$ .

We can of course convert this into an equivalent problem in the form of (5.1) and form its dual. However, if we take the point of view that the function of the dual is to infer from the constraints of (5.3) an inequality of the form  $\mathbf{c}^T \mathbf{x} \geq \gamma$  with  $\gamma$  as large as possible by taking an appropriate linear combination of the constraints, we are effectively looking for  $\mathbf{y} \in \mathbb{R}^m$ ,  $\mathbf{y} \geq 0$ ,  $\mathbf{y}' \in \mathbb{R}^{m'}$ ,  $\mathbf{y}' \leq 0$ , and  $\mathbf{y}'' \in \mathbb{R}^{m''}$ , such that

$$\mathbf{y}^T \mathbf{A} + \mathbf{y}'^T \mathbf{A}' + \mathbf{y}''^T \mathbf{A}'' = \mathbf{c}^T$$

with  $\mathbf{y}^T \mathbf{b} + \mathbf{y}'^T \mathbf{b}' + \mathbf{y}''^T \mathbf{b}''$  to be maximized.

(The reason why we need  $\mathbf{y}' \leq 0$  is because inferring a  $\geq$ -inequality from  $\mathbf{A}' \mathbf{x} \leq \mathbf{b}'$  requires nonpositive multipliers. There is no restriction on  $\mathbf{y}''$  because the constraints  $\mathbf{A}'' \mathbf{x} = \mathbf{b}''$  are equalities.)

This leads to the dual problem:

$$\begin{aligned} \max \quad & \mathbf{y}^T \mathbf{b} + \mathbf{y}'^T \mathbf{b}' + \mathbf{y}''^T \mathbf{b}'' \\ \text{s.t.} \quad & \mathbf{y}^T \mathbf{A} + \mathbf{y}'^T \mathbf{A}' + \mathbf{y}''^T \mathbf{A}'' = \mathbf{c}^T \\ & \mathbf{y} \geq 0 \\ & \mathbf{y}' \leq 0. \end{aligned} \tag{5.4}$$

In fact, we could have derived this dual by applying the definition of the dual problem to

$$\begin{aligned} \min \quad & \mathbf{c}^T \mathbf{x} \\ \text{s.t.} \quad & \begin{bmatrix} \mathbf{A} \\ -\mathbf{A}' \\ \mathbf{A}'' \\ -\mathbf{A}'' \end{bmatrix} \mathbf{x} \geq \begin{bmatrix} \mathbf{b} \\ -\mathbf{b}' \\ \mathbf{b}'' \\ -\mathbf{b}'' \end{bmatrix}, \end{aligned}$$

which is equivalent to (5.3). The details are left as an exercise.

**Observation 2**

Consider the primal problem of the following form:

$$\begin{aligned} \min \quad & \mathbf{c}^T \mathbf{x} \\ \text{s.t.} \quad & \mathbf{A} \mathbf{x} \geq \mathbf{b} \\ & x_i \geq 0 \quad i \in P \\ & x_i \leq 0 \quad i \in N \end{aligned} \tag{5.5}$$

where  $P$  and  $N$  are disjoint subsets of  $\{1, \dots, n\}$ . In other words, constraints of the form  $x_i \geq 0$  or  $x_i \leq 0$  are separated out from the rest of the inequalities.

Forming the dual of (5.5) as defined under Observation 1, we obtain the dual problem

$$\begin{aligned} \max \quad & \mathbf{y}^T \mathbf{b} \\ \text{s.t.} \quad & \mathbf{y}^T \mathbf{a}^{(i)} = c_i \quad i \in \{1, \dots, n\} \setminus (P \cup N) \\ & \mathbf{y}^T \mathbf{a}^{(i)} + p_i = c_i \quad i \in P \\ & \mathbf{y}^T \mathbf{a}^{(i)} + q_i = c_i \quad i \in N \\ & p_i \geq 0 \quad i \in P \\ & q_i \leq 0 \quad i \in N \end{aligned} \tag{5.6}$$

where  $\mathbf{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix}$ . Note that this problem is equivalent to the following without the variables  $p_i$ ,  $i \in P$  and  $q_i$ ,  $i \in N$ :

$$\begin{aligned} \max \quad & \mathbf{y}^T \mathbf{b} \\ \text{s.t.} \quad & \mathbf{y}^T \mathbf{a}^{(i)} = c_i \quad i \in \{1, \dots, n\} \setminus (P \cup N) \\ & \mathbf{y}^T \mathbf{a}^{(i)} \leq c_i \quad i \in P \\ & \mathbf{y}^T \mathbf{a}^{(i)} \geq c_i \quad i \in N, \end{aligned} \tag{5.7}$$

which can be taken as the dual problem of (5.5) instead of (5.6). The advantage here is that it has fewer variables than (5.6).

Hence, the dual problem of

$$\begin{aligned} \min \quad & \mathbf{c}^T \mathbf{x} \\ \text{s.t.} \quad & \mathbf{A} \mathbf{x} \geq \mathbf{b} \\ & \mathbf{x} \geq 0 \end{aligned}$$

is simply

$$\begin{aligned} \max \quad & \mathbf{y}^T \mathbf{b} \\ \text{s.t.} \quad & \mathbf{y}^T \mathbf{A} \leq \mathbf{c}^T \\ & \mathbf{y} \geq 0. \end{aligned}$$

As we can see from above, there is no need to associate dual variables to constraints of the form  $x_i \geq 0$  or  $x_i \leq 0$  provided we have the appropriate types of constraints in the dual problem. Combining all the observations lead to the definition of the dual problem for a primal problem in general form as discussed next.

### 5.5.1. The dual problem

---

Let  $\mathbf{A} \in \mathbb{R}^{m \times n}$ ,  $\mathbf{b} \in \mathbb{R}^m$ ,  $\mathbf{c} \in \mathbb{R}^n$ . Let  $\mathbf{a}^{(i)^\top}$  denote the  $i$ th row of  $\mathbf{A}$ . Let  $\mathbf{A}_j$  denote the  $j$ th column of  $\mathbf{A}$ .

Let  $(P)$  denote the minimization problem with variables in the tuple  $\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$  given as follows:

- The objective function to be minimized is  $\mathbf{c}^\top \mathbf{x}$
- The constraints are

$$\mathbf{a}^{(i)^\top} \mathbf{x} \sqcup_i b_i$$

where  $\sqcup_i$  is  $\leq$ ,  $\geq$ , or  $=$  for  $i = 1, \dots, m$ .

- For each  $j \in \{1, \dots, n\}$ ,  $x_j$  is constrained to be nonnegative, nonpositive, or free (i.e. not constrained to be nonnegative or nonpositive.)

Then the **dual problem** is defined to be the maximization problem with variables in the tuple  $\mathbf{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix}$  given as follows:

- The objective function to be maximized is  $\mathbf{y}^\top \mathbf{b}$
- For  $j = 1, \dots, n$ , the  $j$ th constraint is

$$\begin{cases} \mathbf{y}^\top \mathbf{A}_j \leq c_j & \text{if } x_j \text{ is constrained to be nonnegative} \\ \mathbf{y}^\top \mathbf{A}_j \geq c_j & \text{if } x_j \text{ is constrained to be nonpositive} \\ \mathbf{y}^\top \mathbf{A}_j = c_j & \text{if } x_j \text{ is free.} \end{cases}$$

- For each  $i \in \{1, \dots, m\}$ ,  $y_i$  is constrained to be nonnegative if  $\sqcup_i$  is  $\geq$ ;  $y_i$  is constrained to be nonpositive if  $\sqcup_i$  is  $\leq$ ;  $y_i$  is free if  $\sqcup_i$  is  $=$ .

The following table can help remember the above.

Primal (min)	Dual (max)
$\geq$ constraint	$\geq 0$ variable
$\leq$ constraint	$\leq 0$ variable
$=$ constraint	free variable
$\geq 0$ variable	$\leq$ constraint
$\leq 0$ variable	$\geq$ constraint
free variable	$=$ constraint

Below is an example of a primal-dual pair of problems based on the above definition:

Consider the primal problem:

$$\begin{array}{lllll} \min & x_1 & - & 2x_2 & + & 3x_3 \\ \text{s.t.} & -x_1 & & + & 4x_3 & = & 5 \\ & 2x_1 & + & 3x_2 & - & 5x_3 & \geq & 6 \\ & & & & 7x_2 & \leq & 8 \\ & x_1 & & & & & \geq & 0 \\ & & & x_2 & & & & \text{free} \\ & & & & & & & x_3 \leq 0. \end{array}$$

Here,  $\mathbf{A} = \begin{bmatrix} -1 & 0 & 4 \\ 2 & 3 & -5 \\ 0 & 7 & 0 \end{bmatrix}$ ,  $\mathbf{b} = \begin{bmatrix} 5 \\ 6 \\ 8 \end{bmatrix}$ , and  $\mathbf{c} = \begin{bmatrix} 1 \\ -2 \\ 3 \end{bmatrix}$ .

The primal problem has three constraints. So the dual problem has three variables. As the first constraint in the primal is an equation, the corresponding variable in the dual is free. As the second constraint in the primal is a  $\geq$ -inequality, the corresponding variable in the dual is nonnegative. As the third constraint in the primal is a  $\leq$ -inequality, the corresponding variable in the dual is nonpositive. Now, the primal problem has three variables. So the dual problem has three constraints. As the first variable in the primal is nonnegative, the corresponding constraint in the dual is a  $\leq$ -inequality. As the second variable in the primal is free, the corresponding constraint in the dual is an equation. As the third variable in the primal is nonpositive, the corresponding constraint in the dual is a  $\geq$ -inequality. Hence, the dual problem is:

$$\begin{array}{lllll} \max & 5y_1 & + & 6y_2 & + & 8y_3 \\ \text{s.t.} & -y_1 & + & 2y_2 & & \leq 1 \\ & & & 3y_2 & + & 7y_3 = -2 \\ & 4y_1 & - & 5y_2 & & \geq 3 \\ & y_1 & & & & \text{free} \\ & & & y_2 & & \geq 0 \\ & & & & & y_3 \leq 0. \end{array}$$

**Remarks.** Note that in some books, the primal problem is always a maximization problem. In that case, what is our primal problem is their dual problem and what is our dual problem is their primal problem.

One can now prove a more general version of Theorem 5.5 as stated below. The details are left as an exercise.

### Theorem 5.5: Duality Theorem for Linear Programming

Let  $(P)$  and  $(D)$  denote a primal-dual pair of linear programming problems. If either  $(P)$  or  $(D)$  has an optimal solution, then so does the other. Furthermore, the optimal values of the two problems are equal.

Theorem 5.5.1 is also known informally as **strong duality**.

## Exercises

---

1. Write down the dual problem of

$$\begin{array}{lll} \min & 4x_1 - 2x_2 \\ \text{s.t.} & x_1 + 2x_2 \geq 3 \\ & 3x_1 - 4x_2 = 0 \\ & x_2 \geq 0. \end{array}$$

2. Write down the dual problem of the following:

$$\begin{array}{lll} \min & 3x_2 + x_3 \\ \text{s.t.} & x_1 + x_2 + 2x_3 = 1 \\ & x_1 - 3x_3 \leq 0 \\ & x_1, x_2, x_3 \geq 0. \end{array}$$

3. Write down the dual problem of the following:

$$\begin{array}{lll} \min & x_1 - 9x_3 \\ \text{s.t.} & x_1 - 3x_2 + 2x_3 = 1 \\ & x_1 \leq 0 \\ & x_2 \text{ free} \\ & x_3 \geq 0. \end{array}$$

4. Determine all values  $c_1, c_2$  such that the linear programming problem

$$\begin{array}{lll} \min & c_1x_1 + c_2x_2 \\ \text{s.t.} & 2x_1 + x_2 \geq 2 \\ & x_1 + 3x_2 \geq 1. \end{array}$$

has an optimal solution. Justify your answer

## Solutions

---

1. The dual is

$$\begin{array}{lll} \max & 3y_1 \\ \text{s.t.} & y_1 + 3y_2 = 4 \\ & 2y_1 - 4y_2 \leq -2 \\ & y_1 \geq 0. \end{array}$$

2. The dual is

$$\begin{array}{lll} \max & y_1 \\ \text{s.t.} & y_1 + y_2 \leq 0 \\ & y_1 \leq 3 \\ & 2y_1 - 3y_2 \leq 1 \\ & y_1 \text{ free} \\ & y_2 \leq 0. \end{array}$$

3. The dual is

$$\begin{array}{ll} \max & y_1 \\ \text{s.t.} & y_1 \geq 1 \\ & -3y_1 = 0 \\ & 2y_1 \leq -9 \\ & y_1 \text{ free.} \end{array}$$

4. Let (P) denote the given linear programming problem.

Note that  $\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$  is a feasible solution to (P). Therefore, by Theorem ??, it suffices to find all values  $c_1, c_2$  such that

(P) is not unbounded. This amounts to finding all values  $c_1, c_2$  such that the dual problem of (P) has a feasible solution.

The dual problem of (P) is

$$\begin{array}{ll} \max & 2y_1 + y_2 \\ & 2y_1 + y_2 = c_1 \\ & y_1 + 3y_2 = c_2 \\ & y_1, y_2 \geq 0. \end{array}$$

The two equality constraints gives  $\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} \frac{3}{5}c_1 - \frac{1}{5}c_2 \\ -\frac{1}{5}c_1 + \frac{2}{5}c_2 \end{bmatrix}$ . Thus, the dual problem is feasible if and only if  $c_1$  and  $c_2$  are real numbers satisfying

$$\begin{array}{ll} \frac{3}{5}c_1 - \frac{1}{5}c_2 \geq & 0 \\ -\frac{1}{5}c_1 + \frac{2}{5}c_2 \geq & 0, \end{array}$$

or more simply,

$$\frac{1}{3}c_2 \leq c_1 \leq 2c_2.$$

## 5.6 Complementary slackness

### Theorem 5.6: Weak Duality

Let  $(P)$  and  $(D)$  denote a primal-dual pair of linear programming problems in generic form as defined previously. Let  $\mathbf{x}^*$  be a feasible solution to  $(P)$  and  $\mathbf{y}^*$  is a feasible solution to  $(D)$ . Then the following hold:

1.  $\mathbf{c}^\top \mathbf{x}^* \geq \mathbf{y}^{*\top} \mathbf{b}$ .
2.  $\mathbf{x}^*$  and  $\mathbf{y}^*$  are optimal solutions to the respective problems if and only if the following conditions (known as the **complementary slackness conditions**) hold:

$$\begin{aligned} x_j^* = 0 \quad \text{or} \quad \mathbf{y}^{*\top} \mathbf{A}_j &= c_j \quad \text{for } j = 1, \dots, n \\ y_i^* = 0 \quad \text{or} \quad \mathbf{a}^{(i)\top} \mathbf{x}^* &= b_i \quad \text{for } i = 1, \dots, m \end{aligned}$$

Part 1 of the theorem is known as **weak duality**. Part 2 of the theorem is often called the **Complementary Slackness Theorem**.

#### Proof. [Proof of Theorem 5.6]

Note that if  $x_j^*$  is constrained to be nonnegative, its corresponding dual constraint is  $\mathbf{y}^{*\top} \mathbf{A}_j \leq c_j$ . Hence,  $(c_j - \mathbf{y}^{*\top} \mathbf{A}_j)x_j^* \geq 0$  with equality if and only if  $x_j^* = 0$  or  $\mathbf{y}^{*\top} \mathbf{A}_j = c_j$  (or both).

If  $x_j^*$  is constrained to be nonpositive, its corresponding dual constraint is  $\mathbf{y}^{*\top} \mathbf{A}_j \geq c_j$ . Hence,  $(c_j - \mathbf{y}^{*\top} \mathbf{A}_j)x_j^* \geq 0$  with equality if and only if  $x_j^* = 0$  or  $\mathbf{y}^{*\top} \mathbf{A}_j = c_j$  (or both).

If  $x_j^*$  is free, its corresponding dual constraint is  $\mathbf{y}^{*\top} \mathbf{A}_j = c_j$ . Hence,  $(c_j - \mathbf{y}^{*\top} \mathbf{A}_j)x_j^* = 0$ .

We can combine these three cases and obtain that  $(\mathbf{c}^\top - \mathbf{y}^{*\top} \mathbf{A})\mathbf{x}^* = \sum_{j=1}^n (c_j - \mathbf{y}^{*\top} \mathbf{A}_j)x_j^* \geq 0$  with equality if and only if for each  $j = 1, \dots, n$ ,

$$x_j^* = 0 \text{ or } \mathbf{y}^{*\top} \mathbf{A}_j = c_j.$$

(Here, the usage of “or” is not exclusive.)

Similarly,  $\mathbf{y}^{*\top}(\mathbf{A}\mathbf{x}^* - \mathbf{b}) = \sum_{i=1}^n y_i^*(\mathbf{a}^{(i)\top} \mathbf{x}^* - b_i) \geq 0$  with equality if and only if for each  $i = 1, \dots, n$ ,

$$y_i^* = 0 \text{ or } \mathbf{a}^{(i)\top} \mathbf{x}^* = b_i.$$

(Again, the usage of “or” is not exclusive.)

Adding the inequalities  $(\mathbf{c}^\top - \mathbf{y}^{*\top} \mathbf{A})\mathbf{x}^* \geq 0$  and  $\mathbf{y}^{*\top}(\mathbf{A}\mathbf{x}^* - \mathbf{b}) \geq 0$ , we obtain  $\mathbf{c}^\top \mathbf{x}^* - \mathbf{y}^{*\top} \mathbf{b} \geq 0$  with equality if and only if the complementary slackness conditions hold. By strong duality,  $\mathbf{x}^*$  is optimal  $(P)$  and  $\mathbf{y}^*$  is optimal for  $(D)$  if and only if  $\mathbf{c}^\top \mathbf{x}^* = \mathbf{y}^{*\top} \mathbf{b}$ . The result now follows.



The complementary slackness conditions give a characterization of optimality which can be useful in solving certain problems as illustrated by the following example.

### Example 5.7: Checking Optimality

Let  $(P)$  denote the following linear programming problem:

$$\begin{aligned} \min \quad & 2x_1 + 4x_2 + 2x_3 \\ \text{s.t.} \quad & x_1 + x_2 + 3x_3 \leq 1 \\ & -x_1 + 2x_2 + x_3 \geq 1 \\ & 3x_2 - 6x_3 = 0 \\ & x_1, \quad x_3 \geq 0 \\ & x_2 \quad \text{free.} \end{aligned}$$

Is  $\mathbf{x}^* = \begin{bmatrix} 0 \\ \frac{2}{5} \\ \frac{1}{5} \end{bmatrix}$  an optimal solution to  $(P)$ ?

**Solution.** One could answer this question by solving  $(P)$  and then see if the objective function value of  $\mathbf{x}^*$ , assuming that its feasibility has already been verified, is equal to the optimal value. However, there is a way to make use of the given information to save some work.

Let  $(D)$  denote the dual problem of  $(P)$ :

$$\begin{aligned} \max \quad & y_1 + y_2 \\ \text{s.t.} \quad & y_1 - y_2 \leq 2 \\ & y_1 + 2y_2 + 3y_3 = 4 \\ & 3y_1 + y_2 - 6y_3 \leq 2 \\ & y_1 \leq 0 \\ & y_2 \geq 0 \\ & y_3 \quad \text{free.} \end{aligned}$$

One can check that  $\mathbf{x}^*$  is a feasible solution to  $(P)$ . If  $\mathbf{x}^*$  is optimal, then there must exist a feasible solution  $\mathbf{y}^*$  to  $(D)$  satisfying together with  $\mathbf{x}^*$  the complementary slackness conditions:

$$\begin{aligned} y_1^* = 0 \quad \text{or} \quad & x_1^* + x_2^* + 3x_3^* = 1 \\ y_2^* = 0 \quad \text{or} \quad & -x_1^* + 2x_2^* + x_3^* = 1 \\ y_3^* = 0 \quad \text{or} \quad & 3x_2^* - 6x_3^* = 0 \\ x_1^* = 0 \quad \text{or} \quad & y_1^* - y_2^* = 2 \\ x_2^* = 0 \quad \text{or} \quad & y_1^* + 2y_2^* + 3y_3^* = 4 \\ x_3^* = 0 \quad \text{or} \quad & 3y_1^* + y_2^* - 6y_3^* = 2. \end{aligned}$$

As  $x_2^*, x_3^* > 0$ , satisfying the above conditions require that

$$\begin{aligned} y_1^* + 2y_2^* + 3y_3^* &= 4 \\ 3y_1^* + y_2^* - 6y_3^* &= 2. \end{aligned}$$

Solving for  $y_2^*$  and  $y_3^*$  in terms of  $y_1^*$  gives  $y_2^* = 2 - y_1^*$ ,  $y_3^* = \frac{1}{3}y_1^*$ . To make  $\mathbf{y}^*$  feasible to (D), we can set  $y_1^* = 0$  to obtain the feasible solution  $y_1^* = 0, y_2^* = 2, y_3^* = 0$ . We can check that this  $\mathbf{y}^*$  satisfies the complementary slackness conditions with  $\mathbf{x}^*$ . Hence,  $\mathbf{x}^*$  is an optimal solution to (P) by Theorem 5.6, part 2. ♠

## Exercises

---

1. Let (P) and (D) denote a primal-dual pair of linear programming problems. Prove that if (P) is not infeasible and (D) is infeasible, then (P) is unbounded.
2. Let (P) denote the following linear programming problem:

$$\begin{array}{lll} \min & 4x_2 + 2x_3 \\ \text{s.t.} & x_1 + x_2 + 3x_3 \leq 1 \\ & x_1 - 2x_2 + x_3 \geq 1 \\ & x_1 + 3x_2 - 6x_3 = 0 \\ & x_1, x_2, x_3 \geq 0 \\ & x_2 \text{ free.} \end{array}$$

Determine if  $\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} \frac{3}{5} \\ -\frac{1}{5} \\ 0 \end{bmatrix}$  is an optimal solution to (P).

3. Let (P) denote the following linear programming problem:

$$\begin{array}{lll} \min & x_1 + 2x_2 - 3x_3 \\ \text{s.t.} & x_1 + 2x_2 + 2x_3 = 2 \\ & -x_1 + x_2 + x_3 = 1 \\ & -x_1 + x_2 - x_3 \geq 0 \\ & x_1, x_2, x_3 \geq 0 \end{array}$$

Determine if  $\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$  is an optimal solution to (P).

4. Let  $m$  and  $n$  be positive integers. Let  $\mathbf{A} \in \mathbb{R}^{m \times n}$ . Let  $\mathbf{b} \in \mathbb{R}^m$ . Let  $\mathbf{c} \in \mathbb{R}^n$ . Let (P) denote the linear programming problem

$$\begin{array}{ll} \min & \mathbf{c}^\top \mathbf{x} \\ \text{s.t.} & \mathbf{Ax} = \mathbf{b} \\ & \mathbf{x} \geq 0. \end{array}$$

Let (D) denote the dual problem of (P):

$$\begin{array}{ll} \max & \mathbf{y}^\top \mathbf{b} \\ \text{s.t.} & \mathbf{y}^\top \mathbf{A} \leq \mathbf{c}^\top. \end{array}$$

Suppose that  $\mathbf{A}$  has rank  $m$  and that (P) has at least one optimal solution. Prove that if  $x_j^* = 0$  for every optimal solution  $\mathbf{x}^*$  to (P), then there exists an optimal solution  $\mathbf{y}^*$  to (D) such that  $\mathbf{y}^* \top \mathbf{A}_j < c_i$  where  $\mathbf{A}_j$  denotes the  $j$ th column of  $\mathbf{A}$ .

## Solutions

---

1. By the Fundamental Theorem of Linear Programming, (P) either is unbounded or has an optimal solution. If it is the latter, then by strong duality, (D) has an optimal solution, which contradicts that (D) is infeasible. Hence, (P) must be unbounded.
2. We show that it is not an optimal solution to (P). First, note that the dual problem of (P) is

$$\begin{array}{lll} \max & y_1 & + y_2 \\ \text{s.t.} & y_1 & + y_2 + y_3 \leq 0 \\ & y_1 - 2y_2 + 3y_3 = 4 \\ & 3y_1 + y_2 - 6y_3 \leq 2 \\ & y_1 & \leq 0 \\ & y_2 & \geq 0 \\ & y_3 & \text{free.} \end{array}$$

\end{bmatrix}) were an optimal solution, there would exist  $\mathbf{y}^*$  feasible to (D) satisfying the complementary slackness conditions with  $\mathbf{x}^*$ :

$$\begin{array}{ll} y_1^* = 0 & \text{or } x_1^* + x_2^* + 3x_3^* = 1 \\ y_2^* = 0 & \text{or } x_1^* - 2x_2^* + x_3^* = 1 \\ y_3^* = 0 & \text{or } x_1^* + 3x_2^* - 6x_3^* = 0 \\ x_1^* = 0 & \text{or } y_1^* + y_2^* + y_3^* = 0 \\ x_2^* = 0 & \text{or } y_1^* - 2y_2^* + 3y_3^* = 4 \\ x_3^* = 0 & \text{or } 3y_1^* + y_2^* - 6y_3^* = 2. \end{array}$$

Since  $x_1^* + x_2^* + 3x_3^* < 1$ , we must have  $y_1^* = 0$ . Also,  $x_1^*, x_2^*$  are both nonzero. Hence,

$$\begin{aligned} y_1^* + y_2^* + y_3^* &= 0 \\ y_1^* - 2y_2^* + 3y_3^* &= 4, \end{aligned}$$

implying that

$$\begin{aligned} y_2^* + y_3^* &= 0 \\ -2y_2^* + 3y_3^* &= 4. \end{aligned}$$

Solving gives  $y_2^* = -\frac{4}{5}$  and  $y_3^* = \frac{4}{5}$ . But this implies that  $\mathbf{y}^*$  is not a feasible solution to the dual problem since we need  $y_2^* \geq 0$ . Hence,  $\mathbf{x}^*$  is not an optimal solution to (P).

3. We show that it is not an optimal solution to (P). First, note that the dual problem of (P) is

$$\begin{aligned} \max \quad & 2y_1 + y_2 \\ \text{s.t.} \quad & y_1 - y_2 - y_3 \leq 1 \\ & 2y_1 + y_2 + y_3 \leq 2 \\ & 2y_1 + y_2 - y_3 \leq -3 \\ & y_1, y_2 \quad \text{free.} \\ & y_3 \geq 0 \end{aligned}$$

Note that  $\mathbf{x}^* = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$  is a feasible solution to (P). If it were an optimal solution to (P), there would exist  $\mathbf{y}^*$  feasible to the dual problem (D) satisfying the complementary slackness conditions with  $\mathbf{x}^*$ :

$$\begin{aligned} y_1^* = 0 \quad & \text{or} \quad x_1^* + 2x_2^* + 2x_3^* = 2 \\ y_2^* = 0 \quad & \text{or} \quad -x_1^* + x_2^* + x_3^* = 1 \\ y_3^* = 0 \quad & \text{or} \quad -x_1^* + x_2^* - x_3^* = 0 \\ x_1^* = 0 \quad & \text{or} \quad y_1^* - y_2^* - y_3^* = 1 \\ x_2^* = 0 \quad & \text{or} \quad 2y_1^* + y_2^* + y_3^* = 2 \\ x_3^* = 0 \quad & \text{or} \quad 2y_1^* + y_2^* - y_3^* = -3. \end{aligned}$$

Since  $-x_1^* + x_2^* - x_3^* > 0$ , we must have  $y_3^* = 0$ . Also,  $x_2^* > 0$  implies that  $2y_1^* + y_2^* + y_3^* = 2$ . Simplifying gives  $y_2^* = 2 - 2y_1^*$ .

Hence, for  $\mathbf{y}^*$  to be feasible to the dual problem, it needs to satisfy the third constraint,  $2y_1^* + (2 - 2y_1^*) \leq -3$ , which simplifies to the absurdity  $2 \leq -3$ . Hence,  $\mathbf{x}^*$  is not an optimal solution to (P).

4. Let  $v$  denote the optimal value of (P). Let (P') denote the problem

$$\begin{aligned} \min \quad & -x_i \\ \text{s.t.} \quad & \mathbf{Ax} = \mathbf{b} \\ & \mathbf{c}^\top \mathbf{x} \leq v \\ & \mathbf{x} \geq 0 \end{aligned}$$

Note that  $x^*$  is a feasible solution to (P') if and only if it is an optimal solution to (P). Since  $x_i^* = 0$  for every optimal solution to (P), we see that the optimal value of (P') is 0.

Let (D') denote the dual problem of (P'):

$$\begin{aligned} \max \quad & \mathbf{y}^\top \mathbf{b} + vu \\ \text{s.t.} \quad & \mathbf{y}^\top \mathbf{A}_p + c_p u \leq 0 \quad \text{for all } p \neq i \\ & \mathbf{y}^\top \mathbf{A}_i + c_i u \leq -1 \\ & u \leq 0. \end{aligned}$$

Suppose that an optimal solution to (D') is given by  $\mathbf{y}', u'$ . Let  $\bar{\mathbf{y}}$  be an optimal solution to (D). We consider two cases.

**Case 1:**  $u' = 0$ .

Then  $\mathbf{y}'^\top \mathbf{b} = 0$ . Hence,  $\mathbf{y}^* = \bar{\mathbf{y}} + \mathbf{y}'$  is an optimal solution to (D) with  $\mathbf{y}^{*\top} \mathbf{A}_i < c_i$ .

**Case 2:**  $u' < 0$ .

Then  $\mathbf{y}'^\top \mathbf{b} + vu' = 0$ , implying that  $\frac{1}{|u'|} \mathbf{y}'^\top \mathbf{b} = v$ . Let  $\mathbf{y}^* = \frac{1}{|u'|} \mathbf{y}'$ . Then  $\mathbf{y}^*$  is an optimal solution to (D) with  $\mathbf{y}^{*\top} \mathbf{A}_i < c_i$ .

## 5.7 Basic feasible solution

For a linear constraint  $\mathbf{a}^\top \mathbf{x} \sqcup \gamma$  where  $\sqcup$  is  $\geq$ ,  $\leq$ , or  $=$ , we call  $\mathbf{a}^\top$  the **coefficient row-vector** of the constraint.

Let  $S$  denote a system of linear constraints with  $n$  variables and  $m$  constraints given by  $\mathbf{a}^{(i)\top} \mathbf{x} \sqcup_i b_i$  where  $\sqcup_i$  is  $\geq$ ,  $\leq$ , or  $=$  for  $i = 1, \dots, m$ .

For  $\mathbf{x}' \in \mathbb{R}^n$ , let  $J(S, \mathbf{x}')$  denote the set  $\{i : \mathbf{a}^{(i)\top} \mathbf{x}' = b_i\}$  and define  $\mathbf{A}_{S, \mathbf{x}'}$  to be the matrix whose rows are precisely the coefficient row-vectors of the constraints indexed by  $J(S, \mathbf{x}')$ .

### Example 5.8

Suppose that  $S$  is the system

$$\begin{aligned} x_1 + x_2 - x_3 &\geq 2 \\ 3x_1 - x_2 + x_3 &= 2 \\ 2x_1 - x_2 &\leq 1 \end{aligned}$$

If  $\mathbf{x}' = \begin{bmatrix} 1 \\ 3 \\ 2 \end{bmatrix}$ , then  $J(S, \mathbf{x}') = \{1, 2\}$  since  $\mathbf{x}'$  satisfies the first two constraints with equality but not the third. Hence,  $\mathbf{A}_{S, \mathbf{x}'} = \begin{bmatrix} 1 & 1 & -1 \\ 3 & -1 & 1 \end{bmatrix}$ .

### Definition 5.9

A solution  $\mathbf{x}^*$  to  $S$  is called a **basic feasible solution** if the rank of  $\mathbf{A}_{S, \mathbf{x}^*}$  is  $n$ .

A basic feasible solution to the system in Example 5.7 is  $\begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}$ .

It is not difficult to see that in two dimensions, basic feasible solutions correspond to “corner points” of the set of all solutions. Therefore, the notion of a basic feasible solution generalizes the idea of a corner point to higher dimensions.

The following result is the basis for what is commonly known as the **corner method** for solving linear programming problems in two variables.

### Theorem 5.10: Basic Feasible Optimal Solution

*Let  $(P)$  be a linear programming problem. Suppose that  $(P)$  has an optimal solution and there exists a basic feasible solution to its constraints. Then there exists an optimal solution that is a basic feasible solution.*

We first state the following simple fact from linear algebra:

#### Lemma 5.11

*Let  $\mathbf{A} \in \mathbb{R}^{m \times n}$  and  $\mathbf{d} \in \mathbb{R}^n$  be such that  $\mathbf{A}\mathbf{d} = 0$ . If  $\mathbf{q} \in \mathbb{R}^n$  satisfies  $\mathbf{q}^\top \mathbf{d} \neq 0$  then  $\mathbf{q}^\top$  is not in the row space of  $\mathbf{A}$ .*

#### Proof.

*Proof of Theorem 5.7.*

Suppose that the system of constraints in  $(P)$ , call it  $S$ , has  $m$  constraints and  $n$  variables. Let the objective function be  $\mathbf{c}^\top \mathbf{x}$ . Let  $v$  denote the optimal value.

Let  $\mathbf{x}^*$  be an optimal solution to  $(P)$  such that the rank of  $\mathbf{A}_{S, \mathbf{x}^*}$  is as large as possible. We claim that  $\mathbf{x}^*$  must be a basic feasible solution.

To ease notation, let  $J = J(S, \mathbf{x}^*)$ . Let  $N = \{1, \dots, m\} \setminus J$ .

Suppose to the contrary that the rank of  $\mathbf{A}_{S, \mathbf{x}^*}$  is less than  $n$ . Let  $\mathbf{Px} = \mathbf{q}$  denote the system of equations obtained by setting the constraints indexed by  $J$  to equalities. Then  $\mathbf{Px} = \mathbf{A}_{S, \mathbf{x}^*}$ . Since  $\mathbf{P}$  has  $n$  columns and its rank is less than  $n$ , there exists a nonzero  $\mathbf{d}$  such that  $\mathbf{Pd} = 0$ .

As  $\mathbf{x}^*$  satisfies each constraint indexed by  $N$  strictly, for a sufficiently small  $\varepsilon > 0$ ,  $\mathbf{x}^* + \varepsilon\mathbf{d}$  and  $\mathbf{x}^* - \varepsilon\mathbf{d}$  are solutions to  $S$  and therefore are feasible to  $(P)$ . Thus,

$$\begin{aligned} \mathbf{c}^\top (\mathbf{x}^* + \varepsilon\mathbf{d}) &\geq v \\ \mathbf{c}^\top (\mathbf{x}^* - \varepsilon\mathbf{d}) &\geq v. \end{aligned} \tag{5.1}$$

Since  $\mathbf{x}^*$  is an optimal solution, we have  $\mathbf{c}^\top \mathbf{x}^* = v$ . Hence, (5.1) simplifies to

$$\begin{aligned} \varepsilon\mathbf{c}^\top \mathbf{d} &\geq 0 \\ -\varepsilon\mathbf{c}^\top \mathbf{d} &\geq 0, \end{aligned}$$

giving us  $\mathbf{c}^\top \mathbf{d} = 0$  since  $\varepsilon > 0$ .

Without loss of generality, assume that the constraints indexed by  $N$  are  $\mathbf{Qx} \geq \mathbf{r}$ . As  $(P)$  does have a basic feasible solution, implying that the rank of  $\begin{bmatrix} \mathbf{P} \\ \mathbf{Q} \end{bmatrix}$  is  $n$ , at least one row of  $\mathbf{Q}$ , which we denote by  $\mathbf{t}^\top$ ,

must satisfy  $\mathbf{t}^\top \mathbf{d} \neq 0$ . Without loss of generality, we may assume that  $\mathbf{t}^\top \mathbf{d} > 0$ , replacing  $\mathbf{d}$  with  $-\mathbf{d}$  if necessary.

Consider the linear programming problem

$$\begin{aligned} \min \quad & \lambda \\ \text{s.t.} \quad & \mathbf{Q}(\mathbf{x}^* + \lambda \mathbf{d}) \geq \mathbf{p} \end{aligned}$$

Since at least one entry of  $\mathbf{Q}\mathbf{d}$  is positive (namely,  $\mathbf{t}^\top \mathbf{d}$ ), this problem must have an optimal solution, say  $\lambda'$ . Setting  $\mathbf{x}' = \mathbf{x}^* + \lambda' \mathbf{d}$ , we have that  $\mathbf{x}'$  is an optimal solution since  $\mathbf{c}^\top \mathbf{x}' = v$ .

Now,  $\mathbf{x}'$  must satisfy at least one constraint in  $\mathbf{Q} \geq \mathbf{p}$  with equality. Let  $\mathbf{q}^\top$  be the coefficient row-vector of one such constraint. Then the rows of  $\mathbf{A}_{S,\mathbf{x}'}$  must have all the rows of  $\mathbf{A}_{S,\mathbf{x}^*}$  and  $\mathbf{q}^\top$ . Since  $\mathbf{q}^\top \mathbf{d} \neq 0$ , by Lemma 5.7, the rank of  $\mathbf{A}_{S,\mathbf{x}'}$  is larger than the rank of  $\mathbf{A}_{S,\mathbf{x}^*}$ , contradicting our choice of  $\mathbf{x}^*$ . Thus,  $\mathbf{x}^*$  must be a basic feasible solution. ♠

## Exercises

---

1. Find all basic feasible solutions to

$$\begin{aligned} x_1 + 2x_2 - x_3 &\geq 1 \\ x_2 + 2x_3 &\geq 3 \\ -x_1 + 2x_2 + x_3 &\geq 3 \\ -x_1 + x_2 + x_3 &\geq 0. \end{aligned}$$

2. A set  $S \subset \mathbb{R}^n$  is said to be bounded if there exists a real number  $M > 0$  such that for every  $\mathbf{x} \in S$ ,  $|x_i| < M$  for all  $i = 1, \dots, n$ . Let  $\mathbf{A} \in \mathbb{R}^{m \times n}$  and  $\mathbf{b} \in \mathbb{R}^m$ . Prove that if  $\{\mathbf{x} : \mathbf{Ax} \geq \mathbf{b}\}$  is nonempty and bounded, then there is a basic feasible solution to  $\mathbf{Ax} \geq \mathbf{b}$ .
3. Let  $\mathbf{A} \in \mathbb{R}^{m \times n}$  and  $\mathbf{b} \in \mathbb{R}^m$  where  $m$  and  $n$  are positive integers with  $m \leq n$ . Suppose that the rank of  $\mathbf{A}$  is  $m$  and  $\mathbf{x}'$  is a basic feasible solution to

$$\mathbf{Ax} = \mathbf{b}$$

$$\mathbf{x} \geq 0.$$

Let  $J = \{i : x'_i > 0\}$ . Prove that the columns of  $\mathbf{A}$  indexed by  $J$  are linearly independent.

## Solutions

---

1. To obtain all the basic feasible solutions, it suffices to enumerate all subsystems  $\mathbf{A}'\mathbf{x} \geq \mathbf{b}'$  of the given system such that the rank of  $\mathbf{A}'$  is three and solve  $\mathbf{A}'\mathbf{x} = \mathbf{b}'$  for  $\mathbf{x}$  and see if it is a solution to the system, in which case it is a basic feasible solution. Observe that every basic feasible solution can be discovered in this manner.

We have at most four subsystems to consider.

Setting the first three inequalities to equality gives the unique solution  $\begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}$  which satisfies the given system.. Hence,  $\begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}$  is a basic feasible solution.

Setting the first, second, and fourth inequalities to equality gives the unique solution  $\begin{bmatrix} \frac{5}{3} \\ \frac{1}{3} \\ \frac{4}{3} \end{bmatrix}$  which violates the third inequality of the given system.

Setting the first, third, and fourth inequalities to equality leads to no solution. (In fact, the coefficient matrix of the system does not have rank 3 and therefore this case can be ignored.)

Setting the last three inequalities to equality gives the unique solution  $\begin{bmatrix} 3 \\ 3 \\ 0 \end{bmatrix}$  which satisfies the given system. Hence,  $\begin{bmatrix} 3 \\ 3 \\ 0 \end{bmatrix}$  is a basic feasible solution.

Thus,  $\begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}$  and  $\begin{bmatrix} 3 \\ 3 \\ 0 \end{bmatrix}$  are the only basic feasible solutions.

2. Let  $S$  denote the system  $\mathbf{Ax} \geq \mathbf{b}$ . Let  $\mathbf{x}'$  be a solution to  $S$  such that the rank of  $\mathbf{A}_{S,\mathbf{x}'}$  is as large as possible. If the rank is  $n$ , then we are done. Otherwise, there exists nonzero  $\mathbf{d} \in \mathbb{R}^n$  such  $\mathbf{A}_{S,\mathbf{x}'}\mathbf{d} = 0$ . Since the set of solutions to  $S$  is a bounded set, at least one of the following values is finite:

- $\max\{\lambda : \mathbf{A}(\mathbf{x}' + \lambda\mathbf{d}) \geq \mathbf{b}\}$
- $\min\{\lambda : \mathbf{A}(\mathbf{x}' + \lambda\mathbf{d}) \geq \mathbf{b}\}$

Without loss of generality, assume that the maximum is finite and is equal to  $\lambda^*$ . Setting  $\mathbf{x}^*$  to  $\mathbf{x}' + \lambda^*\mathbf{d}$ , we have that the rows of  $\mathbf{A}_{S,\mathbf{x}^*}$  contains all the rows of  $\mathbf{A}_{S,\mathbf{x}'}$  plus at least one additional row, say  $\mathbf{q}^\top$ . Since  $\mathbf{q}^\top\mathbf{d} \neq 0$ , by Lemma 5.7, the rank of  $\mathbf{A}_{S,\mathbf{x}^*}$  is larger than the rank of  $\mathbf{A}_{S,\mathbf{x}'}$ , contradicting our choice of  $\mathbf{x}'$ .

3. The system of equations obtained from taking all the constraints satisfied with equality by  $\mathbf{x}'$  is

$$\begin{aligned} \mathbf{Ax} &= \mathbf{b} \\ x_j &= 0 \quad j \notin J. \end{aligned} \tag{5.2}$$

Note that the coefficient matrix of this system has rank  $n$  if and only if it has a unique solution. Now, (5.2) simplifies to

$$\sum_{j \in J} x_j \mathbf{A}_j = \mathbf{b},$$

which has a unique solution if and only if the columns of  $\mathbf{A}$  indexed by  $J$  are linearly independent.



# 6. LP Notes from ISE 5405

---

## 6.1 Introduction to Optimization

---

Optimization (i.e., Mathematical Programming) seeks to select, from a set of alternative solutions (decisions), a solution that is “best” for a given performance criteria (i.e., maximize or minimizes the criteria). The following is a general optimization problem:

$$\max\{f(\mathbf{x}, \mathbf{y}) : A(\mathbf{x}) + G(\mathbf{y}) \leq \mathbf{b}_1, H(\mathbf{x}) + W(\mathbf{y}) = \mathbf{b}_2, \mathbf{x} \in \mathcal{X}^+, \mathbf{y} \in \mathcal{R}^+\},$$

where  $\mathbf{x}$  and  $\mathbf{y}$  are vectors of decision variables,  $f(\mathbf{x}, \mathbf{y})$  is the *objective function*, which defines the “best” solution (in this case the optimization problem seeks to maximize the objective function), and  $A(\mathbf{x}) + G(\mathbf{y}) \leq \mathbf{b}_1$ ,  $H(\mathbf{x}) + W(\mathbf{y}) = \mathbf{b}_2$ ,  $\mathbf{x} \in \mathcal{X}^+$ , and  $\mathbf{y} \in \mathcal{R}^+$  are the *constraints* that define the set of possible solutions.

Depending on the nature of the objective function, the constraints, and the input parameters, we can make some broad classifications of optimization problems, as follows:

**Linear Optimization:** Linear optimization, i.e., a linear program (LP), has a linear objective function subject to a set of linear constraints and continuous decision variables.

**Definition:** A function  $f(x_1, x_2, \dots, x_n)$  is linear if, and only if, we have  $f(x_1, x_2, \dots, x_n) = c_1x_1 + c_2x_2 + \dots + c_nx_n$ , where the  $c_1, c_2, \dots, c_n$  coefficients are constants.

An LP has the following general form:

$$\max\{\mathbf{c}\mathbf{x} : \mathbf{Ax} = \mathbf{b}, \mathbf{x} \in \mathcal{R}\},$$

where  $\mathbf{x}$  is a vector of decision variables, and the vectors  $\mathbf{c}$  and  $\mathbf{b}$ , as well as the matrix  $\mathbf{A}$ , are constant problem parameters.

**Nonlinear Optimization:** Nonlinear optimization, i.e., a nonlinear program, is similar to an LP, but objective function and/or the constraints are nonlinear.

**Integer Optimization:** Integer optimization, , i.e., an integer program (IP), is much like an LP, but some) variables restricted to take only integer values.

To use optimization, first you must formulate your model, based on the system of interest and any simplifications required (i.e., assumptions). Formulating the model is not enough, we are also interested in solving the problem, and in a reasonable amount of time (however that is determined). To solve these problems, algorithms are developed. An algorithms is a step-by-step process for finding a solution. We can broadly define different types of algorithms as follows:

- Optimal algorithms - processes that solve the model to optimality (and proves optimality).
- Near-optimal algorithms with bounds (heuristics), processes that do not guarantee optimality, but provides "good" solutions with known bounds.
- Other heuristic algorithms, processes that provide a "good" solution, but bounds are not provided, or are not that useful.

Algorithms can also be categorized based on performance, for instance, usually a *polynomial time algorithm* is better than an *exponential time algorithm*.

The class will almost exclusively focus on Linear Programs (LP) because: 1) LP are useful for many problems; 2) LPs are, relatively, easy to solve; and most importantly 3) LP are an important foundation for further courses in optimization.

### 6.1.1. Notation

---

- We use bold text to indicate a matrix or vector, e.g., the matrix **A** or the vector **x**.

## 6.2 Linear Optimization

---

In this section, we study on linear optimization problems, i.e., linear programs (LPs).

### 6.2.1. Problem Formulation

---

Remember, for a linear program (LP), we want to maximize or minimize a linear **objective function** of the continuous decision variables, while considering linear constraints on the values of the decision variables.

#### Definition 6.1: Linear Function

*function  $f(x_1, x_2, \dots, x_n)$  is linear if, and only if, we have  $f(x_1, x_2, \dots, x_n) = c_1x_1 + c_2x_2 + \dots + c_nx_n$ , where the  $c_1, c_2, \dots, c_n$  coefficients are constants.*

#### A Generic Linear Program (LP)

##### Decision Variables:

$x_i$  : continuous variables ( $x_i \in \mathcal{R}$ , i.e., a real number),  $\forall i = 1, \dots, 3$ .

##### Parameters (known input parameters):

$c_i$  : cost coefficients  $\forall i = 1, \dots, 3$

$a_{ij}$  : constraint coefficients  $\forall i = 1, \dots, 3, j = 1, \dots, 4$

$b_j$  : right hand side coefficient for constraint  $j$ ,  $j = 1, \dots, 4$

$$\text{Min } z = c_1x_1 + c_2x_2 + c_3x_3 \quad (6.1)$$

$$\text{s.t. } a_{11}x_1 + a_{12}x_2 + a_{13}x_3 \geq b_1 \quad (6.2)$$

$$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 \leq b_2 \quad (6.3)$$

$$a_{31}x_1 + a_{32}x_2 + a_{33}x_3 = b_3 \quad (6.4)$$

$$a_{41}x_1 + a_{42}x_2 + a_{43}x_3 \geq b_4 \quad (6.5)$$

$$x_1 \geq 0, x_2 \leq 0, x_3 \text{ urs.} \quad (6.6)$$

Eq. (6.1) is the objective function, (6.2)-(6.5) are the functional constraints, while (6.6) is the sign restrictions (*urs* signifies that the variable is unrestricted). If we were to add any one of these following constraints  $x_2 \in \{0, 1\}$  ( $x_2$  is binary-valued) or  $x_3 \in \mathbb{Z}$  ( $x_3$  is integer-valued) we would have an Integer Program. For the purposes of this class, an Integer Program (IP) is just an LP with added integer restrictions on (some) variables.

While, in general, solvers will take any form of the LP, there are some special forms we use in analysis:

**LP Standard Form:** The standard form has all constraints as equalities, and all variables as non-negative. The generic LP is not in standard form, but any LP can be converted to standard form.

Since  $x_2$  is non-positive and  $x_3$  unrestricted, perform the following substitutions  $x_2 = -\hat{x}_2$  and  $x_3 = x_3^+ - x_3^-$ , where  $\hat{x}_2, x_3^+, x_3^- \geq 0$ . Eqs. (6.2) and (6.5) are in the form left-hand side (LHS)  $\geq$  right-hand side (RHS), so to make an equality, subtract a non-negative slack variable from the LHS ( $s_1$  and  $s_4$ ). Eq. (6.3) is in the form LHS  $\leq$  RHS, so add a non-negative slack variable to the LHS.

$$\begin{aligned} \text{Min } z &= c_1x_1 - c_2\hat{x}_2 + c_3(x_3^+ - x_3^-) \\ \text{s.t. } a_{11}x_1 - a_{12}x_2 + a_{13}(x_3^+ - x_3^-) - s_1 &= b_1 \\ a_{21}x_1 - a_{22}\hat{x}_2 + a_{23}(x_3^+ - x_3^-) + s_2 &= b_2 \\ a_{31}x_1 - a_{32}\hat{x}_2 + a_{33}(x_3^+ - x_3^-) &= b_3 \\ a_{41}x_1 - a_{42}\hat{x}_2 + a_{43}x_3 - s_4 &= b_4 \\ x_1, \hat{x}_2, x_3^+, x_3^-, s_1, s_2, s_4 &\geq 0. \end{aligned}$$

**LP Canonical Form:** For a minimization problem the canonical form of the LP has the LHS of each constraint greater than or equal to the the RHS, and a maximization the LHS less than or equal to the RHS, and non-negative variables.

Next we consider some formulation examples:

**Production Problem:** 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.

Decision variables:

$x_i$  : number of units of product  $i$  to produce,

$\forall i = \{\text{bookcase, desk, cabinet}\}$ .

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

**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.

Decision variables:

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

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

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

The solution is as follows:

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

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

Decision variables:

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

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

**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

Now we formulate the problem:

Decision variables:

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

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

**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	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

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.

Decision variables:

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

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

**The 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.

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

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

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

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  $x_{21} = 0.5$ ).

**Linear Data Models:** Consider a data set that consists of  $n$  data points  $(x_i, y_i)$ . We want to fit the best line to this data, such that given an  $x$ -value, we can predict the associated  $y$ -value. Thus, the form is  $y_i = \alpha x_i + \beta$  and we want to choose the  $\alpha$  and  $\beta$  values such that we minimize the error for our  $n$  data points.

### Variables:

$e_i$  : error for data point  $i$ ,  $i = 1, \dots, n$ .

$\alpha$  : slope of fitted line.

$\beta$  : intercept of fitted line.

$$\begin{aligned} \text{Min } & \sum_{i=1}^n |e_i| \\ \text{s.t. } & \alpha x_i + \beta - y_i = e_i, \quad i = 1, \dots, n \\ & e_i, \alpha, \beta \text{ urs.} \end{aligned}$$

Of course, absolute values are not linear function, so we can linearize as follows:

### Decision variables:

$e_i^+$  : positive error for data point  $i$ ,  $i = 1, \dots, n$ .

$e_i^-$  : negative error for data point  $i$ ,  $i = 1, \dots, n$ .

$\alpha$  : slope of fitted line.

$\beta$  : intercept of fitted line.

$$\begin{aligned} \text{Min } & \sum_{i=1}^n e_i^+ + e_i^- \\ \text{s.t. } & \alpha x_i + \beta - y_i = e_i^+ - e_i^-, \quad i = 1, \dots, n \\ & e_i^+, e_i^- \geq 0, \alpha, \beta \text{ urs.} \end{aligned}$$

**Two-Person Zero-Sum Games:** Consider a game with two players,  $\mathcal{A}$  and  $\mathcal{B}$ . In each round of the game,  $\mathcal{A}$  chooses one out of  $m$  possible actions, while  $\mathcal{B}$  chooses one out of  $n$  actions. If  $\mathcal{A}$  takes action  $j$  while  $\mathcal{B}$  takes action  $i$ , then  $c_{ij}$  is the payoff for  $\mathcal{A}$ , if  $c_{ij} > 0$ ,  $\mathcal{A}$  “wins”  $c_{ij}$  (and  $\mathcal{B}$  losses that amount), and if  $c_{ij} < 0$  if  $\mathcal{B}$  “wins”  $-c_{ij}$  (and  $\mathcal{A}$  losses that amount). This is a two-person zero-sum game.

Rock, Paper, Scissors is a two-person zero-sum game, with the following payoff matrix.

		$\mathcal{A}$		
		R	P	S
$\mathcal{B}$	R	0	1	-1
	P	-1	0	1
	S	1	-1	0

We can have a similar game, but with a different payoff matrix, as follows:

		$\mathcal{A}$		
		R	P	S
$\mathcal{B}$	R	4	-1	-1
	P	-2	4	-2
	S	-3	-3	4

What is the optimal strategy for  $\mathcal{A}$  (for either game)? We define  $x_j$  as the probability that  $\mathcal{A}$  takes action  $j$  (related to the columns). Then the payoff for  $\mathcal{A}$ , if  $\mathcal{B}$  takes action  $i$  is  $\sum_{j=1}^m c_{ij}x_j$ . Of course,  $\mathcal{A}$  does not know what action  $\mathcal{B}$  will take, so let's find a strategy that maximizes the minimum expected winnings of  $\mathcal{A}$  given any random strategy of  $\mathcal{B}$ , which we can formulate as follows:

$$\begin{aligned} \text{Max } & \left( \min_{i=1,\dots,n} \sum_{j=1}^m c_{ij}x_j \right) \\ \text{s.t. } & \sum_{j=1}^m x_j = 1 \\ & x_j \geq 0, \quad i = 1, \dots, m, \end{aligned}$$

which can be linearized as follows:

$$\begin{aligned} \text{Max } & z \\ \text{s.t. } & z \leq \sum_{j=1}^m c_{ij}x_j, \quad i = 1, \dots, n \\ & \sum_{j=1}^m x_j = 1 \\ & x_j \geq 0, \quad i = 1, \dots, m. \end{aligned}$$

The last two constraints ensure the that  $x_i$ -variables are valid probabilities. If you solved this LP for the first game (i.e., payoff matrix) you find the best strategy is  $x_1 = 1/3$ ,  $x_2 = 1/3$ , and  $x_3 = 1/3$  and there is no expected gain for player  $\mathcal{A}$ . For the second game, the best strategy is  $x_1 = 23/107$ ,  $x_2 = 37/107$ , and  $x_3 = 47/107$ , with  $\mathcal{A}$  gaining, on average,  $8/107$  per round.

### 6.2.2. Linear Algebra Review

---

#### Vectors and Linear and Convex Combinations

**Vectors:** Vector  $\mathbf{n}$  has  $n$ -elements and represents a point (or an arrow from the origin to the point, denoting a direction) in  $\mathbb{R}^n$  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  $\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_k$  if  $\mathbf{b} = \sum_{i=1}^k \lambda_i \mathbf{a}_i$  for  $\lambda_1, \lambda_2, \dots, \lambda_k \in \mathbb{R}$ . If  $\lambda_1, \lambda_2, \dots, \lambda_k \in \mathbb{R}_{\geq 0}$  then  $\mathbf{b}$  is a *non-negative linear combination* of  $\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_k$ .

**Convex Combination:** The vector  $\mathbf{b}$  is a **convex combination** of  $\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_k$  if  $\mathbf{b} = \sum_{i=1}^k \lambda_i \mathbf{a}_i$ , for  $\lambda_1, \lambda_2, \dots, \lambda_k \in \mathbb{R}_{\geq 0}$  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  $\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_k$  are *linearly independent* if the following linear combination  $\sum_{i=1}^k \lambda_i \mathbf{a}_i = 0$  implies that  $\lambda_i = 0$ ,  $i = 1, 2, \dots, k$ . In  $\mathbb{R}^2$  two vectors are only linearly dependent if they lie on the same line. Can you have three linearly independent vectors in  $\mathbb{R}^2$ ?

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

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

## Convex and Polyhedral Sets

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

**Hyperplanes and Half-Spaces:** A hyperplane in  $\mathbb{R}^n$  divides  $\mathbb{R}^n$  into 2 half-spaces (like a line does in  $\mathbb{R}^2$ ). A hyperplane is the set  $\{\mathbf{x} : \mathbf{p}\mathbf{x} = k\}$ , 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  $\{\mathbf{x} : \mathbf{p}\mathbf{x} \geq k\}$  and  $\{\mathbf{x} : \mathbf{p}\mathbf{x} \leq k\}$ .

**Polyhedral Set:** A *polyhedral set* (or polyhedron) is the set of points in the intersection of a finite set of half-spaces. Set  $\mathcal{S} = \{\mathbf{x} : \mathbf{A}\mathbf{x} \leq \mathbf{b}, \mathbf{x} \geq 0\}$ , 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  $\mathcal{C} = \{\mathbf{x} : \mathbf{A}\mathbf{x} \leq 0\}$  is a polyhedral cone.

**Edges and Faces:** An *edge* of a polyhedral set  $\mathcal{S}$  is defined by  $n-1$  hyperplanes, and a *face* of  $\mathcal{S}$  by one or more defining hyperplanes of  $\mathcal{S}$ , 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  $\mathbb{R}^2$  faces are only edges and extreme points, but in  $\mathbb{R}^3$  there is a third type of face, and so on...

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

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

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

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  $\mathbb{R}^n$  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  $\mathcal{C}$  is convex cone, then for any  $\mathbf{x} \in \mathcal{C}$  we

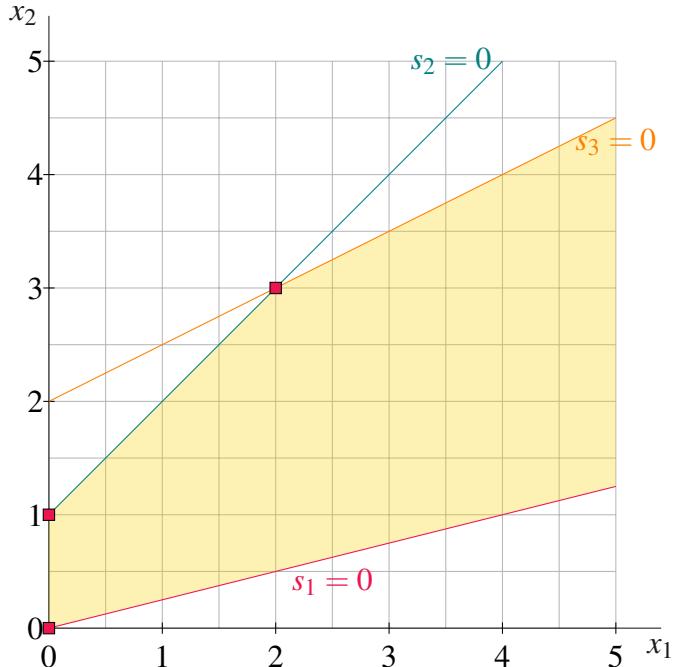
have  $\lambda \mathbf{x} \in \mathcal{C}$ ,  $\lambda \geq 0$ .

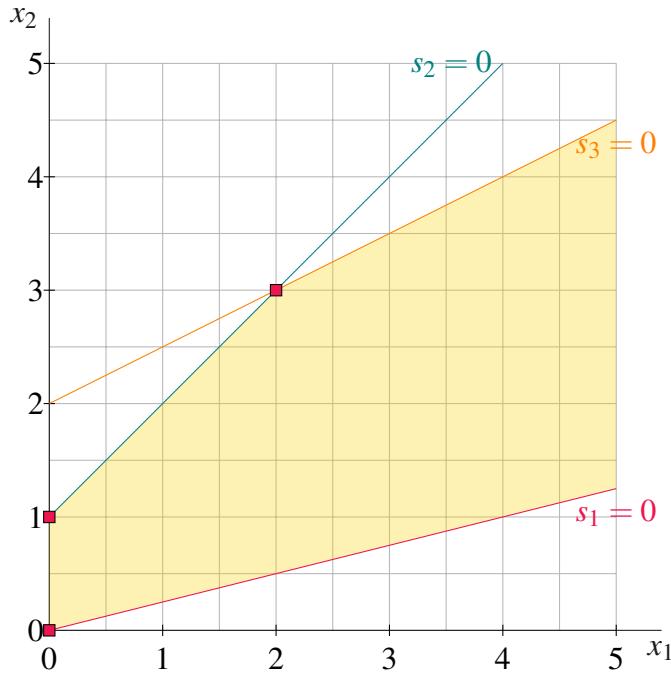
**Unbounded Polyhedral Sets:** If  $\mathcal{S}$  is unbounded, it will have *directions*.  $\mathbf{d}$  is a direction of  $\mathcal{S}$  only if  $\mathbf{Ax} + \lambda \mathbf{d} \leq \mathbf{b}, \mathbf{x} + \lambda \mathbf{d} \geq 0$  for all  $\lambda \geq 0$  and all  $\mathbf{x} \in \mathcal{S}$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

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

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



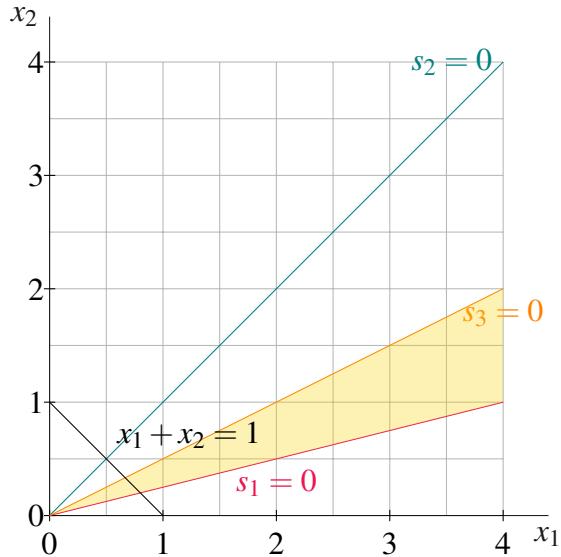


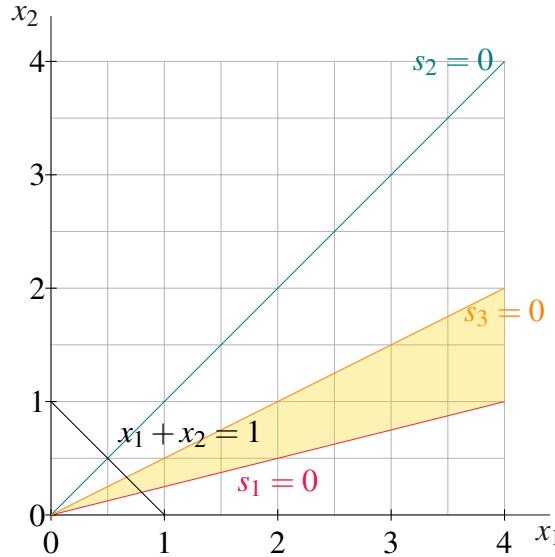
E.g., consider the  $s_3 = 0$  (orange) line, to find the extreme direction start at extreme point  $(2,3)$  and find another feasible point on the orange line, say  $(4,4)$  and subtract  $(2,3)$  from  $(4,4)$ , which yields  $(2,1)$ .

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.

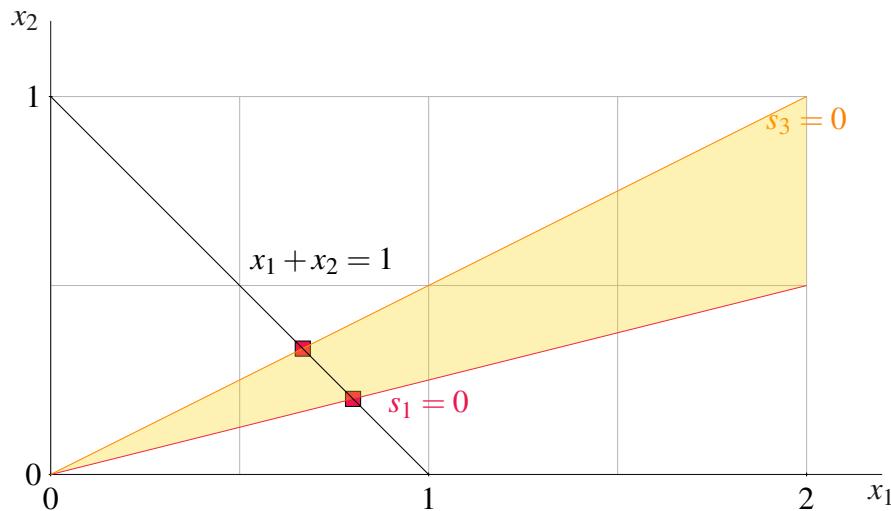
To find the extreme directions we can change the right-hand-side to  $\mathbf{b} = 0$ , which forms a polyhedral cone (in yellow), and then add the constraint  $x_1 + x_2 = 1$ . The intersection of the cone and  $x_1 + x_2 = 1$  form a line segment.

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





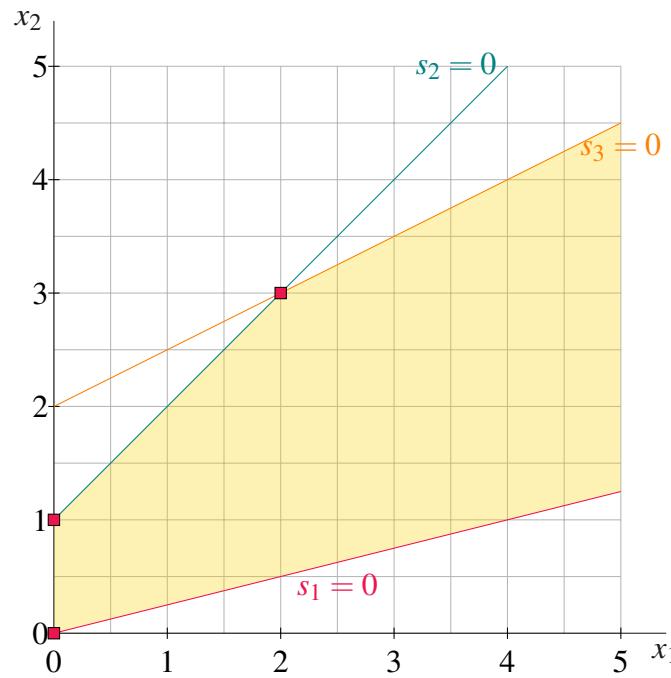
Magnifying for clarity, and removing the  $s_2 = 0$  (teal) line, as it is redundant, and marking the extreme points of the new feasible region,  $(4/5, 1/5)$  and  $(2/3, 1/3)$ , with red boxes, we have:



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

**Representation Theorem:** Let  $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_k$  be the set of extreme points of  $\mathcal{S}$ , and if  $\mathcal{S}$  is unbounded,  $\mathbf{d}_1, \mathbf{d}_2, \dots, \mathbf{d}_l$  be the set of extreme directions. Then any  $\mathbf{x} \in \mathcal{S}$  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  $\sum_{j=1}^k \lambda_j = 1$ ,  $\lambda_j \geq 0$ ,  $\forall j = 1, 2, \dots, k$ , and  $\mu_j \geq 0$ ,  $\forall j = 1, 2, \dots, l$ .

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

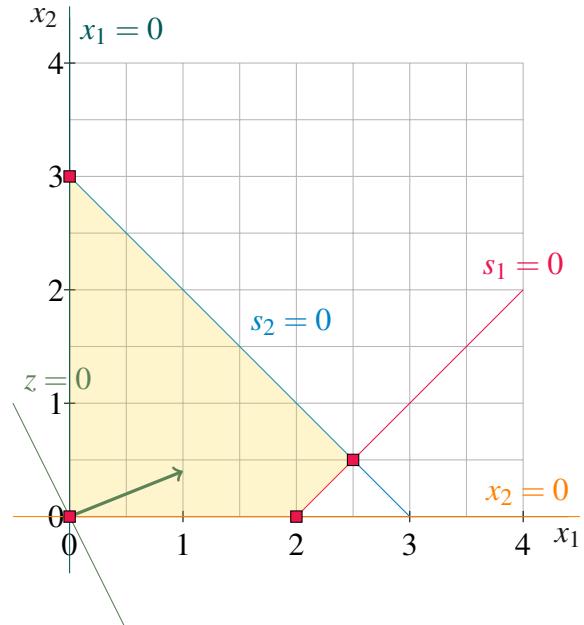


Represent point  $(1/2, 1)$  as a convex combination of the extreme points of the above LP. Find  $\lambda$ s to solve the following system of equations:

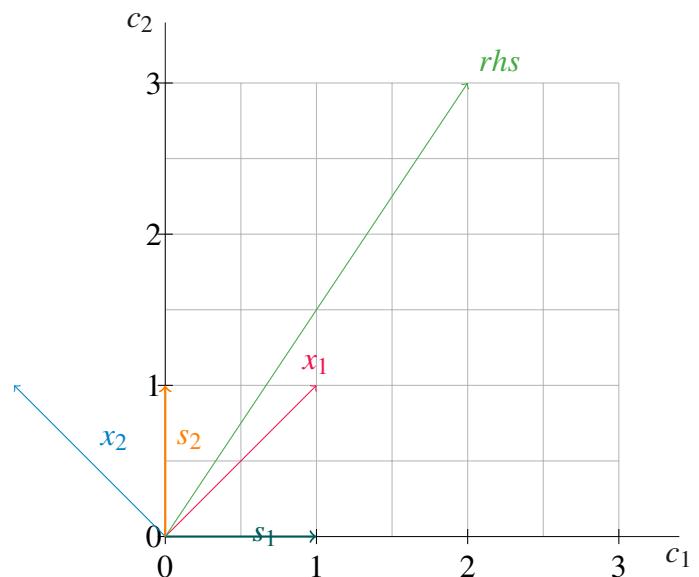
$$\lambda_1 \begin{bmatrix} 0 \\ 0 \end{bmatrix} + \lambda_2 \begin{bmatrix} 0 \\ 1 \end{bmatrix} + \lambda_3 \begin{bmatrix} 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 1/2 \\ 1 \end{bmatrix}$$

## The Variable (Canonical Form) and Requirement Space

$$\begin{aligned} \max \quad & z = 2x_1 + x_2 \\ \text{s.t.} \quad & x_1 - x_2 + s_1 = 2 \\ & x_1 + x_2 + s_2 = 3 \\ & x_1, x_2, s_1, s_2 \geq 0. \end{aligned}$$



$$\begin{aligned} \max \quad & z = 2x_1 + x_2 \\ \text{s.t.} \quad & x_1 - x_2 + s_1 = 2 \\ & x_1 + x_2 + s_2 = 3 \\ & x_1, x_2, s_1, s_2 \geq 0. \end{aligned}$$



### Tableaus

After putting an LP into standard form, we can put the system of equations into a table form, the “tableau”.

$$\begin{aligned} \max \quad & z = 2x_1 + x_2 \\ \text{s.t.} \quad & x_1 - x_2 + s_1 = 2 \\ & x_1 + x_2 + s_2 = 3 \\ & x_1, x_2, s_1, s_2 \geq 0. \end{aligned}$$

max	$z$	$x_1$	$x_2$	$x_3$	$x_4$	$rhs$
(r0 - z)	1	-2	-1	0	0	0
(r1 - $x_3$ )	0	1	-1	1	0	2
(r2 - $x_4$ )	0	1	1	0	1	3

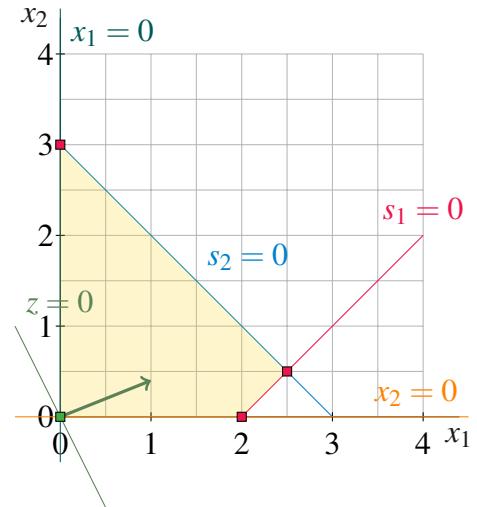
Why are the coefficients negative in row zero, we change  $z = 2x_1 + x_2$  to  $z - 2x_1 - x_2 = 0$  so we have only constants on the right-hand-side (rhs).

This tableau represents a basic solution, because it contains an identity matrix. The basic variables are those variables having columns in the identity matrix (here,  $x_3$  and  $x_4$ ), and it is feasible because the rhs for row  $1 - m$  are non-negative.

We can consider  $z$  a permanent member of an expanded basis if we treat row zero like any other row (although the rhs of row 0 can be negative).

### Basic Solutions and Extreme Points

max	$z$	$x_1$	$x_2$	$x_3$	$x_4$	$rhs$
(r0 - z)	1	-2	-1	0	0	0
(r1 - $x_3$ )	0	1	-1	1	0	2
(r2 - $x_4$ )	0	1	1	0	1	3



Here the basic variables are  $x_3 = 2$  and  $x_4 = 3$ , and the  $z$ -value of objective function value is 0.

Let's go to the extreme point  $(2,0)$  which has basic variables  $x_1$  and  $x_4$  (this new extreme point is adjacent to the extreme point  $(0,0)$ ). Why?

max	$z$	$x_1$	$x_2$	$x_3$	$x_4$	$rhs$
(r0 - z)	1	-2	-1	0	0	0
(r1 - $x_3$ )	0	1	-1	1	0	2
(r2 - $x_4$ )	0	1	1	0	1	3

First get a 1 coefficient in row 1 in the  $x_1$  column by multiplying row 1 by a scalar (no action needed, already equals one).

max	$z$	$x_1$	$x_2$	$x_3$	$x_4$	$rhs$
(r0 - z)	1	-2	-1	0	0	0
(r1 - $x_3$ )	0	1	-1	1	0	2
(r2 - $x_4$ )	0	1	1	0	1	3

Then use row 1 to zero out the row 0 coefficient for  $x_1$  by multiplying row 1 by 2 and adding it to row 0 to get a new row 0.

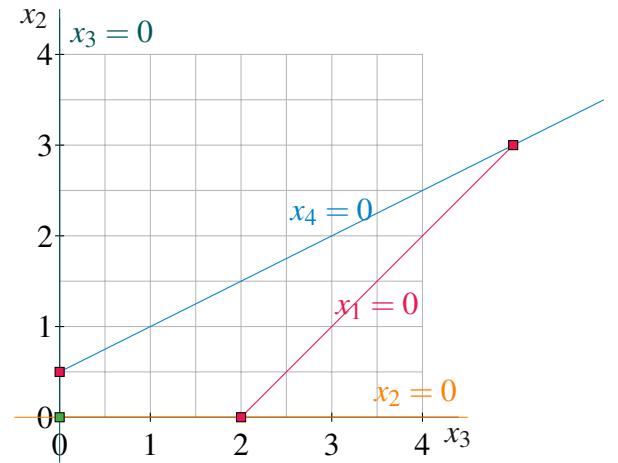
max	$z$	$x_1$	$x_2$	$x_3$	$x_4$	$rhs$
(r0 - z)	1	0	-3	2	0	4
(r1 - $x_3$ )	0	1	-1	1	0	2
(r2 - $x_4$ )	0	1	1	0	1	3

Lastly use row 1 to zero out the row 2 coefficient for  $x_1$  by multiplying row 1 by -1 and adding it to row 2 to get a new row 2.

max	$z$	$x_1$	$x_2$	$x_3$	$x_4$	$rhs$
(r0 -z)	1	0	-3	2	0	4
(r1 - $x_3$ )	0	1	-1	1	0	2
(r2 - $x_4$ )	0	0	2	-1	1	1

The nonbasic variables for this tableau are  $x_2$  and  $x_3$ , so we can graph the new LP in the nonbasic variable space.

max	$z$	$x_1$	$x_2$	$x_3$	$x_4$	$rhs$
(r0 -z)	1	0	-3	2	0	4
(r1 - $x_3$ )	0	1	-1	1	0	2
(r2 - $x_4$ )	0	0	2	-1	1	1



## Matrix Math

An  $m \times n$  matrix is an array of real numbers with  $m$  rows and  $n$  columns. Any matrix can be represented by its constituent set of row or column vectors.

$$\mathbf{A} = \begin{bmatrix} 1 & 3 \\ 6 & 4 \end{bmatrix} = [\mathbf{a}_1 \quad \mathbf{a}_2] = \begin{bmatrix} \mathbf{a}^1 \\ \mathbf{a}^2 \end{bmatrix},$$

where  $\mathbf{a}_1 = \begin{bmatrix} 1 \\ 6 \end{bmatrix}$ ,  $\mathbf{a}_2 = \begin{bmatrix} 3 \\ 4 \end{bmatrix}$ ,  $\mathbf{a}^1 = [1 \ 3]$ , and  $\mathbf{a}^2 = [6 \ 4]$ . Additionally,  $a_{11} = 1$ ,  $a_{12} = 3$ ,  $a_{21} = 6$ ,  $a_{22} = 4$ .

**Matrix Addition:** Two matrices of the same dimension can be added componentwise, thus  $\mathbf{C} = \mathbf{A} + \mathbf{B}$  means that  $c_{ij} = a_{ij} + b_{ij}$ .

**Scalar Multiplication:** Just like it sounds. If  $k$  is a scalar, then  $k\mathbf{A}$  means that every component of  $\mathbf{A}$  is multiplied by  $k$ .

**Matrix Multiplication:**  $\mathbf{A}$  is a  $m \times n$  matrix and  $\mathbf{B}$  is a  $p \times q$  matrix.  $\mathbf{AB}$  (matrix multiplication) is only defined if  $n = p$  and the result is a  $m \times q$  matrix,  $\mathbf{BA}$  is only defined if  $q = m$  and the result is a  $p \times n$ .  $\mathbf{AB}$  is not necessarily equal to  $\mathbf{BA}$ , thus  $\mathbf{C} = \mathbf{AB}$  where  $c_{ij} = \sum_{k=1}^n a_{ik}b_{kj}$ ,  $i = 1, \dots, m$ ,  $j = 1, \dots, p$ , e.g.,  $c_{11}$  is the sum of the componentwise multiplication of the first row of  $\mathbf{A}$  and the first column of  $\mathbf{B}$ .

**Identity Matrix:** A square matrix (denoted by  $\mathbf{I}$ ) with all zero components, except for the diagonal:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

**Elementary Matrix Operations:** These operations are used to solve systems of linear equations or inverting a matrix. The three operations are as follows (for any matrix  $\mathbf{A}$ ):

- Interchange two rows of  $\mathbf{A}$ .
- Multiply a row by a nonzero scalar.
- Replace row  $i$  with row  $i$  plus row  $j$  multiplied by a nonzero scalar.

### Inverting a Matrix:

$$\mathbf{A} = \begin{bmatrix} 3 & 9 & 2 \\ 1 & 1 & 1 \\ 5 & 4 & 7 \end{bmatrix} \text{ and } \mathbf{A}^{-1} = \begin{bmatrix} -\frac{3}{11} & \frac{5}{11} & -\frac{7}{11} \\ \frac{2}{11} & -1 & \frac{1}{11} \\ \frac{1}{11} & -3 & \frac{6}{11} \end{bmatrix}$$

To find  $\mathbf{A}^{-1}$  using elementary row operations to transform  $\mathbf{A}$  into an identity matrix, while performing these same operations on the attached identity matrix.

$$\left[ \begin{array}{ccc|ccc} 3 & 9 & 2 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 \\ 5 & 4 & 7 & 0 & 0 & 1 \end{array} \right] \Rightarrow \left[ \begin{array}{ccc|ccc} 1 & 0 & 0 & -\frac{3}{11} & 5 & -\frac{7}{11} \\ 0 & 1 & 0 & \frac{2}{11} & -1 & \frac{1}{11} \\ 0 & 0 & 1 & \frac{1}{11} & -3 & \frac{6}{11} \end{array} \right].$$

**Rank of a Matrix:**  $\mathbf{A}$  is a  $m \times n$  matrix then  $\text{rank}(\mathbf{A}) \leq \min\{m, n\}$ , if  $\text{rank}(\mathbf{A}) = \min\{m, n\}$ , then  $\mathbf{A}$  is of full rank. If  $\mathbf{A}$  is not full rank, but of rank  $k$ , where  $k < \min\{m, n\}$ , then using the elementary row operations, we can transform  $\mathbf{A}$  to the following:  $\begin{bmatrix} \mathbf{I}_k & \mathbf{Q} \\ 0 & 0 \end{bmatrix}$

### 6.2.3. Linear Optimization Theory

---

Consider an arbitrary LP, which we will call the primal ( $P$ ):

$$(P) : \max\{\mathbf{c}\mathbf{x} : \mathbf{Ax} \leq \mathbf{b}, \mathbf{x} \geq 0\},$$

where  $\mathbf{A}$  is an  $m \times n$  matrix, and  $\mathbf{x}$  is a  $n$  element column vector. Every primal LP has a related LP, which we call the dual, the dual of ( $P$ ) is:

$$(D) : \min\{\mathbf{w}\mathbf{b} : \mathbf{w}\mathbf{A} \geq \mathbf{c}, \mathbf{w} \geq 0\}.$$

Before we discuss properties of duality, and why it is important, we start with how to formulate the dual for any given LP. If the LP has a different form like  $P$ , we find the dual based on the  $P$  and  $D$  example above. If the LP does not have this form, we can transform it to this form, or use the rules in the following table, first noting that:

- The dual of problem  $D$  is problem  $P$ .
- Each primal constraint has an associated dual variable ( $w_i$ ) and each dual constraint has an associated primal variable ( $x_i$ ).
- When the primal is a maximization, the dual is a minimization, and vice versa.

$$\max \mathbf{c}\mathbf{x} :$$

$$\mathbf{a}_{1*}\mathbf{x} \leq b_1 (w_1 \geq 0)$$

$$\mathbf{a}_{2*}\mathbf{x} = b_2 (w_2 \text{ urs})$$

$$\mathbf{a}_{3*}\mathbf{x} \geq b_3 (w_3 \leq 0)$$

$$\vdots$$

$$x_1 \geq 0, x_2 \text{ urs}, x_3 \leq 0, \dots$$

$$\min \mathbf{w}\mathbf{b} :$$

$$\mathbf{w}\mathbf{a}_{*1} \geq c_1 (x_1 \geq 0)$$

$$\mathbf{w}\mathbf{a}_{*2} = c_2 (x_2 \text{ urs})$$

$$\mathbf{w}\mathbf{a}_{*3} \leq c_3 (x_3 \leq 0)$$

$$\vdots$$

$$w_1 \geq 0, w_2 \text{ urs}, w_3 \leq 0, \dots$$

To illustrate the relationship between the primal and dual, consider this production problem we previously formulated:

**Production Problem:** 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.

Decision variables:

$x_i$  : number of units of product  $i$  to produce,

$\forall i = \{\text{bookcase, desk, cabinet}\}$ .

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

Considering the formulation above as the primal, consider a new, related, problem: You have an offer to buy all your resources (the leased hours for the two robots, and the TAA). Formulate an LP to find the minimum value of the resources given the above plans for the three products and commitments to buy them.

Decision variables:

$w_i$  : selling price, per unit, for resource  $i$ ,  $\forall i = \{\text{TAA, LazWeld1, CrumCut1}\}$ .

$$\begin{aligned} \min & 21w_1 + 23w_2 + 17w_3 : \\ 2w_1 + 3w_2 + 1w_3 &\geq 18 \\ 2w_1 + 2w_2 + 2w_3 &\geq 16 \\ 1w_1 + 2w_2 + 1w_3 &\geq 10 \\ w_1, w_2, w_3 &\geq 0. \end{aligned}$$

Define  $\mathbf{w} = \mathbf{c}_B \mathbf{B}^{-1}$  as the vector of *shadow prices*, where  $w_i$  represents the change in the objective function value caused by a unit change to the associated  $b_i$  parameter (i.e., increasing the amount of resource  $i$  by one unit, see dual objective function).

Consider the following primal tableau (where  $z_P$  is the primal objective function value) for  $(P)$  :  $(\max \{\mathbf{c}\mathbf{x} : \mathbf{Ax} \leq \mathbf{b}, \mathbf{x} \geq 0\})$

	$z_P$	$x_i$	rhs
$z_P$	1	$\mathbf{c}_B \mathbf{B}^{-1} \mathbf{a}_i - c_i$	$\mathbf{c}_B \mathbf{B}^{-1} \mathbf{b}$
$BV$	0	$\mathbf{B}^{-1} \mathbf{a}_i$	$\mathbf{B}^{-1} \mathbf{b}$

Observe that if a basis for  $P$  is optimal, then the row zero coefficients for the variables are greater than, or equal to, zero, that is,  $c_B B^{-1} a_i - c_i \geq 0$  for each  $x_i$  (if the variable is a slack, this simplifies to  $c_B B^{-1} \geq 0$ ).

Substituting  $w = c_B B^{-1}$  we get  $wA \geq c, w \geq 0$  which corresponds to dual feasibility.

$$(D) : \min\{wb : wA \geq c, w \geq 0\}.$$

### Weak Duality Property

If  $\mathbf{x}$  and  $\mathbf{w}$  are feasible solutions to  $P$  and  $D$ , respectively, then  $\mathbf{c}\mathbf{x} \leq \mathbf{w}\mathbf{A}\mathbf{x} \leq \mathbf{w}\mathbf{b}$ .

$$(P) : \max\{\mathbf{c}\mathbf{x} : \mathbf{A}\mathbf{x} \leq \mathbf{b}, \mathbf{x} \geq 0\}.$$

$$(D) : \min\{wb : wA \geq c, w \geq 0\}.$$

This implies that the objective function value for a feasible solution to  $P$  is a lower bound on the objective function value for the optimal solution to  $D$ , and the objective function value for a feasible solution to  $D$  is an upper bound on the objective function value for the optimal solution to  $P$ .

Thus if the objective function values are equal, i.e.,  $\mathbf{c}\mathbf{x} = \mathbf{w}\mathbf{b}$ , then the solutions  $\mathbf{x}$  and  $\mathbf{w}$  are optimal.

### Fundamental Theorem of Duality

For problems  $P$  and  $D$  (i.e., any primal dual set) exactly one of the following is true:

1. Both have optimal solutions  $\mathbf{x}$  and  $\mathbf{w}$  where  $\mathbf{c}\mathbf{x} = \mathbf{w}\mathbf{b}$ .
2. One problem is unbounded (i.e., the objective function value can become arbitrarily large for a maximization, or arbitrarily small for a minimization), and the other is infeasible.
3. Both are infeasible.

#### 6.2.3.1. Optimality Conditions

---

### Farka's Lemma

Consider the following two systems:

1.  $\mathbf{A}\mathbf{x} \geq 0, \mathbf{c}\mathbf{x} < 0$ .
2.  $\mathbf{w}\mathbf{A} = \mathbf{c}, \mathbf{w} \geq 0$ .

Farka's Lemma - exactly one of these systems has a solution.

**Suppose system 1 has  $\mathbf{x}$  as a solution:**

- If  $\mathbf{w}$  were a solution to system 2, then post-multiplying each side of  $\mathbf{w}\mathbf{A} = \mathbf{c}$  by  $\mathbf{x}$  would yield  $\mathbf{w}\mathbf{A}\mathbf{x} = \mathbf{c}\mathbf{x}$ .
- Since  $\mathbf{Ax} \geq 0$  and  $\mathbf{w} \geq 0$ , this implies that  $\mathbf{cx} \geq 0$ , which violates  $\mathbf{cx} < 0$ .
- Thus we show that if system 1 has a solution, system 2 cannot have one.

**Suppose system 1 has no solution:**

- Consider the following LP:  $\min\{\mathbf{cx} : \mathbf{Ax} \geq 0\}$ .
- The optimal solution is  $\mathbf{cx} = 0$  and  $\mathbf{x} = 0$ .
- The LP in standard form (substitute  $\mathbf{x} = \mathbf{x}' - \mathbf{x}''$ ,  $\mathbf{x}' \geq 0$  and  $\mathbf{x}'' \geq 0$  and add  $\mathbf{x}^s \geq 0$ ) follows:  

$$\min\{\mathbf{cx}' - \mathbf{cx}'' : \mathbf{Ax}' - \mathbf{Ax}'' - \mathbf{x}^s = 0, \mathbf{x}', \mathbf{x}'', \mathbf{x}^s \geq 0\}$$
- $\mathbf{x}' = 0, \mathbf{x}'' = 0, \mathbf{x}^s = 0$  is an optimal extreme point solution.
- Using  $\mathbf{x}^s$  as an initial feasible basis, solve with the simplex algorithm (with cycling prevention) to find a basis where  $\mathbf{c}_B \mathbf{B}^{-1} \mathbf{a}_i - c_i \leq 0$  for all variables. Define  $\mathbf{w} = \mathbf{c}_B \mathbf{B}^{-1}$ .
- This yields  $\mathbf{w}\mathbf{A} - \mathbf{c} \leq 0, -\mathbf{w}\mathbf{A} + \mathbf{c} \leq 0, -\mathbf{w} \leq 0\}$ , from the columns for variables  $\mathbf{x}', \mathbf{x}'', \mathbf{x}^s$ , respectively. Thus,  $\mathbf{w} \geq 0$  and  $\mathbf{w}\mathbf{A} = \mathbf{c}$ , and system 2 has a solution.

**Karush-Kuhn-Tucker (KKT) Conditions**

$$(P) : \max\{\mathbf{cx} : \mathbf{Ax} \leq \mathbf{b}, \mathbf{x} \geq 0\}.$$

$$(D) : \min\{\mathbf{wb} : \mathbf{wA} \geq \mathbf{c}, \mathbf{w} \geq 0\}.$$

For problems  $P$  and  $D$ , with solutions  $\mathbf{x}$  and  $\mathbf{w}$ , respectively, we have the following conditions, which for LPs are necessary and sufficient conditions for optimality:

1.  $\mathbf{Ax} \leq \mathbf{b}, \mathbf{x} \geq 0$  (primal feasibility).
2.  $\mathbf{wA} \geq \mathbf{c}, \mathbf{w} \geq 0$  (dual feasibility).
3.  $\mathbf{w}(\mathbf{Ax} - \mathbf{b}) = 0$  and  $\mathbf{x}(\mathbf{c} - \mathbf{wA}) = 0$  (complementary slackness).

Note we can rewrite the third condition as  $\mathbf{w}(\mathbf{Ax} - \mathbf{b}) = \mathbf{wx}^s = 0$  and  $\mathbf{x}(\mathbf{c} - \mathbf{wA}) = \mathbf{xw}^s = 0$ , where  $\mathbf{x}^s$  and  $\mathbf{w}^s$  are the slack variables for the primal and dual problems, respectively.

**Why do the KKT conditions hold?**

Suppose that the LP  $\min\{\mathbf{cx} : \mathbf{Ax} \geq \mathbf{b}, \mathbf{x} \geq 0\}$  has an optimal solution  $\mathbf{x}^*$  (the dual is  $\max\{\mathbf{wb} : \mathbf{wA} \leq \mathbf{c}, \mathbf{w} \geq 0\}$ ).

- Since  $\mathbf{x}^*$  is optimal there is no direction  $\mathbf{d}$  such that  $\mathbf{c}(\mathbf{x}^* + \lambda \mathbf{d}) < \mathbf{c}\mathbf{x}^*$ ,  $\mathbf{A}(\mathbf{x}^* + \lambda \mathbf{d}) \geq \mathbf{b}$ , and  $\mathbf{x}^* + \lambda \mathbf{d} \geq 0$  for  $\lambda > 0$ .
- Let  $\mathbf{Gx} \geq \mathbf{g}$  be the binding inequalities in  $\mathbf{Ax} \geq \mathbf{b}$  and  $\mathbf{x} \geq 0$  for solution  $\mathbf{x}^*$  that is,  $\mathbf{Gx}^* = \mathbf{g}$ .
- Based on the optimality of  $\mathbf{x}^*$ , there is no direction  $\mathbf{d}$  at  $\mathbf{x}^*$  such that  $\mathbf{cd} < 0$  and  $\mathbf{Gd} \geq 0$  (else we could improve the solution).
- Based on Farka's Lemma, if the system  $\mathbf{cd} < 0$ ,  $\mathbf{Gd} \geq 0$  does not have a solution, the system  $\mathbf{wG} = \mathbf{c}$ ,  $\mathbf{w} \geq 0$  must have a solution.
- $\mathbf{G}$  is composed of rows from  $\mathbf{A}$  where  $\mathbf{a}_{i*}\mathbf{x}^* = b_i$  and vectors  $\mathbf{e}_i$  for any  $x_i^* = 0$ .
- We can divide the  $\mathbf{w}$  into two sets:
  - $\{w_i, i : \mathbf{a}_{i*}\mathbf{x}^* = b_i\}$  - those corresponding to the binding functional constraints in the primal.
  - $\{w_i^s, j : x_i^* = 0\}$  - those corresponding to the binding non-negativity constraints in the primal.
- Thus  $\mathbf{G}$  has the columns  $\mathbf{a}_{i*}^T$  for  $w_i$  and  $e_i^T$  for  $w_i^s$ .
- Since  $\mathbf{wG} = \mathbf{c}$ ,  $\mathbf{w} \geq 0$  must have a solution, this solution is feasible for  $\mathbf{wA} \leq \mathbf{c}, \mathbf{w} \geq 0$  where  $w_i^s$  are added slacks. Thus,  $\mathbf{G}$  is missing some columns from  $\mathbf{A}$  (and thus some  $w$  variables) and some slack variables if  $\mathbf{wA} \leq \mathbf{c}, \mathbf{w} \geq 0$  were put into standard form, but those are not needed for feasibility based on the result, and thus can be thought of as set to zero, giving us complementary slackness.

**Example:** Consider a production LP (the primal  $P$ ) where the variables represent the amount of three products to produce, using three resources, represented by the functional constraints. In standard form  $P$  and  $D$  have  $x_4^s, x_5^s, x_6^s$  and  $w_4^s, w_5^s, w_6^s$  as slack variables, respectively.

Decision variables:

$x_i$  : number of units of product  $i$  to produce,  $\forall i = \{1, 2, 3\}$ .

$$(P) : \begin{aligned} & \max z_P = 18x_1 + 16x_2 + 10x_3 \\ & \text{s.t. } 2x_1 + 2x_2 + 1x_3 + x_4^s = 21 \quad (w_1) \\ & \quad 3x_1 + 2x_2 + 2x_3 + x_5^s = 23 \quad (w_2) \\ & \quad 1x_1 + 2x_2 + 1x_3 + x_6^s = 17 \quad (w_3) \\ & \quad x_1, x_2, x_3, x_4^s, x_5^s, x_6^s \geq 0. \end{aligned}$$

$$(D) : \begin{aligned} & \min z_D = 21w_1 + 23w_2 + 17w_3 \\ & \text{s.t. } 2w_1 + 3w_2 + 1w_3 \geq 18 \quad (x_1) \\ & \quad 2w_1 + 2w_2 + 2w_3 \geq 16 \quad (x_2) \\ & \quad 1w_1 + 2w_2 + 1w_3 \geq 10 \quad (x_3) \\ & \quad 1w_1 \geq 0 \\ & \quad 1w_2 \geq 0 \\ & \quad 1w_3 \geq 0 \\ & \quad w_1, w_2, w_3 \text{ urs.} \end{aligned}$$

Decision variables:

$w_i$  : unit selling price for resource  $i$ ,  $\forall i = \{1, 2, 3\}$ .

$$(D) : \begin{aligned} & \min z_D = 21w_1 + 23w_2 + 17w_3 : \\ & \quad 2w_1 + 3w_2 + 1w_3 - w_4^s = 18 \quad (x_1) \\ & \quad 2w_1 + 2w_2 + 2w_3 - w_5^s = 16 \quad (x_2) \\ & \quad 1w_1 + 2w_2 + 1w_3 - w_6^s = 10 \quad (x_3) \\ & \quad w_1, w_2, w_3, w_4^s, w_5^s, w_6^s \geq 0. \end{aligned}$$

The initial basic feasible tableau for the primal, i.e., having the slack variables form the basis, follows:

$P : \max$	$z_P$	$x_1$	$x_2$	$x_3$	$x_4^s$	$x_5^s$	$x_6^s$	rhs
$z_P$	1	-18	-16	-10	0	0	0	0
$x_4^s$	0	2	2	1	1	0	0	21
$x_5^s$	0	3	2	2	0	1	0	23
$x_6^s$	0	1	2	1	0	0	1	17

$$x_1, x_2, x_3 = 0, x_4^s = 21, x_5^s = 23, x_6^s = 17, z_P = 0$$

The following dual tableau **conforms with the primal tableau through complementary slackness**.

$D : \min$	$z_D$	$w_1$	$w_2$	$w_3$	$w_4^s$	$w_5^s$	$w_6^s$	rhs
	$z_D$	1	-21	-23	-17	0	0	0
	$w_4^s$	0	-2	-3	-1	1	0	-18
	$w_5^s$	0	-2	-2	-2	0	1	-16
	$w_6^s$	0	-1	-2	-1	0	0	-10

$$w_1, w_2, w_3 = 0, w_4^s = -18, w_5^s = -16, w_6^s = -10, z_D = 0$$

**Complementary slackness:**  $w_1 x_4^s = 0, w_2 x_5^s = 0, w_3 x_6^s = 0, x_1 w_4^s = 0, x_2 w_5^s = 0, x_3 w_6^s = 0$ .

- If a primal variable is basic, then its corresponding dual variable must be nonbasic, and vice versa.
- The primal is suboptimal, and the dual tableau has a basic infeasible solution.
- Row 0 of the primal tableau has dual variable values in the corresponding primal variable columns.

The primal basis is not optimal, so enter  $x_1$  into the basis, and remove  $x_5^s$ , which yields:

P: Max	$z_P$	$x_1$	$x_2$	$x_3$	$x_4^s$	$x_5^s$	$x_6^s$	rhs	
	$z_P$	1	0	-4	2	0	6	0	138
	$x_4^s$	0	0	2/3	-1/3	1	-2/3	0	17/3
	$x_1$	0	1	2/3	2/3	0	1/3	0	23/3
	$x_6^s$	0	0	4/3	1/3	0	-1/3	1	28/3

D: Min	$z_D$	$w_1$	$w_2$	$w_3$	$w_4^s$	$w_5^s$	$w_6^s$	rhs	
	$z_D$	1	-17/3	0	-28/3	-23/3	0	0	138
	$w_2$	0	2/3	1	1/3	-1/3	0	0	6
	$w_5^s$	0	-2/3	0	-4/3	-2/3	1	0	-4
	$w_6^s$	0	1/3	0	-1/3	-2/3	0	1	2

The primal tableau does not represent an optimal basic solution, and the dual tableau does not represent a feasible basic solution.

Using Dantzig's rule, we enter  $x_2$  into the basis, and using the ratio test we find that  $x_6^s$  leaves the basis. This change in basis yields the following tableau:

P: Max	$z_P$	$x_1$	$x_2$	$x_3$	$x_4^s$	$x_5^s$	$x_6^s$	rhs	
	$z_P$	1	0	0	3	0	5	3	166
	$x_4^s$	0	0	0	-1/2	1	-1/2	-1/2	1
	$x_1$	0	1	0	1/2	0	1/2	-1/2	3
	$x_2$	0	0	1	1/4	0	-1/4	3/4	7

D: Min	$z_D$	$w_1$	$w_2$	$w_3$	$w_4^s$	$w_5^s$	$w_6^s$	rhs	
	$z_D$	1	-1	0	0	-3	-7	0	166
	$w_2$	0	1/2	1	0	-1/2	1/4	0	5
	$w_3$	0	1/2	0	1	1/2	-3/4	0	3
	$w_6^s$	0	1/2	0	0	-1/2	-1/4	1	3

Decision variables:

$x_i$  : number of units of product  $i$  to produce,  $\forall i = \{1, 2, 3\}$ .

$$(P) : \max z_P = 18x_1 + 16x_2 + 10x_3 :$$

$$2x_1 + 2x_2 + 1x_3 + x_4^s = 21 \quad (w_1)$$

$$3x_1 + 2x_2 + 2x_3 + x_5^s = 23 \quad (w_2)$$

$$1x_1 + 2x_2 + 1x_3 + x_6^s = 17 \quad (w_3)$$

$$x_1, x_2, x_3, x_4^s, x_5^s, x_6^s \geq 0.$$

The LP  $\max\{\mathbf{c}\mathbf{x} : \mathbf{Ax} \leq \mathbf{b}, \mathbf{x} \geq 0\}$  has an optimal solution  $\mathbf{x}^*$  (the dual is  $\min\{\mathbf{wb} : \mathbf{wA} \geq \mathbf{c}, \mathbf{w} \geq 0\}$ ).

- Since  $\mathbf{x}^*$  is optimal there is no direction  $\mathbf{d}$  such that  $\mathbf{c}(\mathbf{x}^* + \lambda \mathbf{d}) > \mathbf{c}\mathbf{x}^*$ ,  $\mathbf{A}(\mathbf{x}^* + \lambda \mathbf{d}) \leq \mathbf{b}$ , and  $\mathbf{x}^* + \lambda \mathbf{d} \geq 0$  for  $\lambda > 0$ .
- Let  $\mathbf{Gx} \leq \mathbf{g}$  be the binding inequalities in  $\mathbf{Ax} \leq \mathbf{b}$  and  $\mathbf{x} \geq 0$  for solution  $\mathbf{x}^*$ , that is,  $\mathbf{Gx}^* = \mathbf{g}$ .

For our example,

$$\mathbf{G|g} = \left[ \begin{array}{ccc|c} 3 & 2 & 2 & 23 \\ 1 & 2 & 1 & 17 \\ 0 & 0 & -1 & 0 \end{array} \right]$$

- Based on the optimality of  $\mathbf{x}^*$ , there is no direction  $\mathbf{d}$  at  $\mathbf{x}^*$  such that  $\mathbf{cd} > 0$  and  $\mathbf{Gd} \leq 0$  (this includes  $\mathbf{d} \leq 0$ ) (else we could improve the solution).
- From Farka's Lemma, if the system  $\mathbf{cd} > 0$ ,  $\mathbf{Gd} \leq 0$  does not have a solution, the system  $\mathbf{wG} = \mathbf{c}$ ,  $\mathbf{w} \geq 0$  must have a solution.

$$3w_2 + 1w_3 = 18 \quad (x_1)$$

$$2w_2 + 2w_3 = 16 \quad (x_2)$$

$$2w_2 + 1w_3 - w_6^s = 10 \quad (x_3)$$

$$w_2, w_3, w_6^s \geq 0.$$

D: Min	$z_D$	$w_1$	$w_2$	$w_3$	$w_4^s$	$w_5^s$	$w_6^s$	rhs	
	1	-1	0	0	-3	-7	0	166	
	$w_2$	0	1/2	1	0	-1/2	1/4	0	5
	$w_3$	0	1/2	0	1	1/2	-3/4	0	3
	$w_6^s$	0	1/2	0	0	-1/2	-1/4	1	3

**Challenge 1:** Solve the following LP (as represented in the tableau), using the given tableau as a starting point. Provide the details of the algorithm to do so, and make it valid for both maximization and minimization problems.

$D : \min$	$z_D$	$w_1$	$w_2$	$w_3$	$w_4^s$	$w_5^s$	$w_6^s$	rhs
	$z_D$	1	-21	-23	-17	0	0	0
	$w_4^s$	0	-2	-3	-1	1	0	-18
	$w_5^s$	0	-2	-2	-2	0	1	-16
	$w_6^s$	0	-1	-2	-1	0	0	-10

**Challenge 2:** Given the following optimal tableau to our production LP, we can buy 12 units of resource 2 for \$4 a unit. Should we, please provide the analysis needed to make this decision.

$P : \max$	$z_P$	$x_1$	$x_2$	$x_3$	$x_4^s$	$x_5^s$	$x_6^s$	rhs	
	$z_P$	1	0	0	3	0	5	3	166
	$x_4^s$	0	0	0	-1/2	1	-1/2	-1/2	1
	$x_1$	0	1	0	1/2	0	1/2	-1/2	3
	$x_2$	0	0	1	1/4	0	-1/4	3/4	7

## 6.2.4. Solution Algorithms

---

We start with some preliminaries, and then discuss the simplex algorithm, assuming an initial basic feasible solution, including tableau formulas. Next two extensions to the algorithm, for finding an initial basic feasible solution, are discussed. We then explore a useful algorithm for solving certain LPs that have “too many” columns.

Consider an LP  $\max\{\mathbf{c}\mathbf{x} : \mathbf{Ax} = \mathbf{b}, \mathbf{x} \geq 0\}$  in standard form, where:

- $\mathbf{A}$  is an  $m \times n$  matrix of rank  $m$  and  $n \geq m$ ;  $\mathbf{A}$  consists of  $n$  column vectors,  $\mathbf{a}_1, \mathbf{a}_2, \mathbf{a}_3, \dots, \mathbf{a}_n$ .
- $\mathbf{c}$  and  $\mathbf{x}$  are  $n$ -vectors.
- $\mathbf{b}$  is an  $m$ -vector with non-negative elements.

We can partition the problem as  $\mathbf{x} = [\mathbf{x}_B, \mathbf{x}_N]$ ,  $\mathbf{A} = [\mathbf{B}, \mathbf{N}]$ ,  $\mathbf{c} = [\mathbf{c}_B, \mathbf{c}_N]$ , where:

- $\mathbf{x}_B$  is the vector of basic variables
- $\mathbf{B}$  is the *basis matrix*, a nonsingular (i.e., it consists of  $m$  linearly independent columns of  $\mathbf{A}$ )  $m \times m$  matrix.

- $\mathbf{c}_B$  is the vector of cost coefficients for the basic variables.
- $\mathbf{x}_N$  is the vector of nonbasic variables
- $\mathbf{N}$  is the *nonbasic matrix*, a  $m \times n - m$  matrix.
- $\mathbf{c}_N$  is the vector of cost coefficients for the basic variables.

The LP can then be written as:

$$\max \{\mathbf{c}_B \mathbf{x}_B + \mathbf{c}_N \mathbf{x}_N : \mathbf{B} \mathbf{x}_B + \mathbf{N} \mathbf{x}_N = \mathbf{b}, \mathbf{x}_B, \mathbf{x}_N \geq 0\}.$$

For the feasible region, we can write the system of equations as follows:

$$\mathbf{B} \mathbf{x}_B + \mathbf{N} \mathbf{x}_N = \mathbf{b}.$$

$$\mathbf{B} \mathbf{x}_B = \mathbf{b} - \mathbf{N} \mathbf{x}_N.$$

Premultiplying by  $\mathbf{B}^{-1}$  yields:

$$\mathbf{x}_B = \mathbf{B}^{-1} \mathbf{b} - \mathbf{B}^{-1} \mathbf{N} \mathbf{x}_N.$$

By setting  $\mathbf{x}_N = 0$  and solving we (potentially) find a *basic feasible solution* to the system, which corresponds to an extreme point of the feasible region. Remember that the nonbasic variables  $\mathbf{x}_N = 0$  represent the defining hyperplanes for a solution.

For any set of  $m$  variables, the result can be:

1. a basic feasible solution,  $\mathbf{x}_B \geq 0$ .
2. a basic infeasible solution, some  $x \in \mathbf{x}_B \leq 0$ .
3. a set of linearly dependent columns that does not span the  $m$ -space.

For this system there are possibly  $n$  choose  $m$   $\binom{n}{m}$  basic solutions, that is, the number of basic feasible solutions is bounded by  $n! / m!(n-m)!$  from above.

#### 6.2.4.1. The Simplex Algorithm

---

It is common practice to put an LP into a tableau. To do so, we first modify the objective function by bringing all the variables to the left-hand side, yielding the following tableau of LP data:

	$z$	$x_i$	$rhs$
Row 0 ( $z$ )	1	$-c_i$	0
Rows 1-m	0	$\mathbf{a}_i$	$\mathbf{b}$

We are interested in tableaus that represent basic solutions, which have a special form; the columns of coefficients for the basis to form an identity matrix, which we can obtain using elementary row operations.

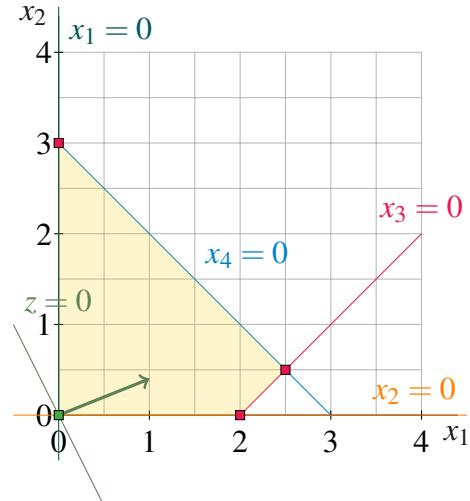
Consider the following LP:

$$\begin{aligned} \text{Max } z &= 2x_1 + x_2 \\ \text{s.t. } x_1 - x_2 + x_3 &= 2 \\ x_1 + x_2 + x_4 &= 3 \\ x_1, x_2, x_3, x_4 &\geq 0, \end{aligned}$$

where  $m=2$ ,  $n=4$ ,  $\mathbf{A} = \begin{bmatrix} 1 & -1 & 1 & 0 \\ 1 & 1 & 0 & 1 \end{bmatrix}$ ,  
 $\mathbf{a}_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$ ,  $\mathbf{a}_2 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}$ ,  $\mathbf{a}_3 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ ,  $\mathbf{a}_4 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$ ,  
 $\mathbf{c} = [2 \ 1 \ 0 \ 0]$ ,  
 $\mathbf{x} = [x_1 \ x_2 \ x_3 \ x_4]^T$ , (T for transpose,  $\mathbf{x}$  is a column vector),  
and  $\mathbf{b} = \begin{bmatrix} 2 \\ 3 \end{bmatrix}$ .

The tableau and graph for this LP follow:

max	$z$	$x_1$	$x_2$	$x_3$	$x_4$	$rhs$
(z)	1	-2	-1	0	0	0
( $x_3$ )	0	1	-1	1	0	2
( $x_4$ )	0	1	1	0	1	3



Luckily, this tableau already represents a basis, which has basic variables  $\mathbf{x}_B = [x_3 \ x_4]^T$  (we can consider  $z$  as a basic variable of sorts to complete the identity matrix). Thus for this tableau we have  $\mathbf{x}_N = [x_1 \ x_2]^T$ ,  $\mathbf{B} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ ,  $\mathbf{N} = \begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix}$ . This basic solution represents an extreme point if we set the nonbasic variables to zero, as we see in the graph of the LP in the nonbasic variable space.

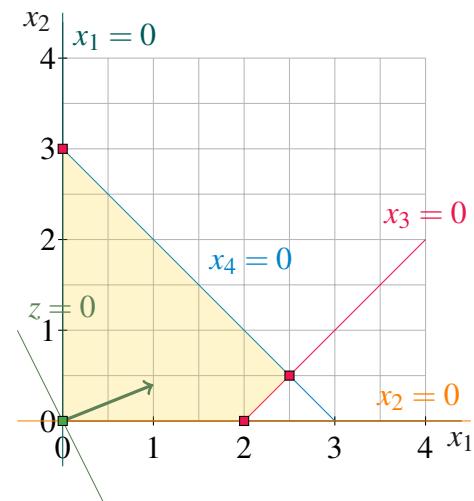
Here the basic variable values are  $x_3 = 2$  and  $x_4 = 3$ , and  $z = 0$ .

The simplex algorithm (mostly), tableau version:

1. Put the LP into a standard form tableau, find a set of basic variables that form a feasible basis, and modify the tableau to represent the basis using elementary row operations, we want the coefficients of the basic variables to form an identity matrix.
2. Check optimality, for a maximization (minimization), if the Row 0 coefficients for the nonbasic variables are all nonnegative (nonpositive) then the basis is optimal.
3. If the basis is not optimal, then find an adjacent basis that improves the solution. This will involve swapping one of the basic variables with a nonbasic variable to form a new basis.
4. Select a nonbasic entering variable using Dantzig's rule, specifically, for a maximization (minimization) problem pick the nonbasic variable with the smallest negative (largest positive) reduced cost.
5. Select a variable to leave the basis. Conceptually, as we increase the entering variable's value from zero, the values of the basis variables should change, the basic variable that goes to zero first is the leaving variable. To find the leaving variable, use the ratio test. For each row  $1-m$  having a positive coefficient in the entering variable column, divide the  $rhs$  by the entering variable's (positive) coefficient. The basic variable corresponding to row with the smallest ratio is the leaving variable.
6. Put the tableau into the proper form for the new basis using elementary row operations and go to Step 2.

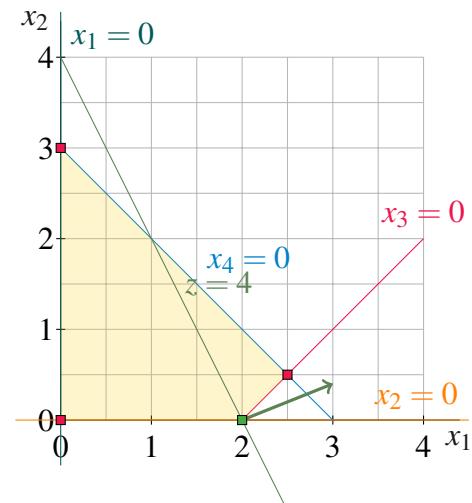
Consider the following example:

max	$z$	$x_1$	$x_2$	$x_3$	$x_4$	$rhs$
( $z$ )	1	-2	-1	0	0	0
( $x_3$ )	0	1	-1	1	0	2
( $x_4$ )	0	1	1	0	1	3

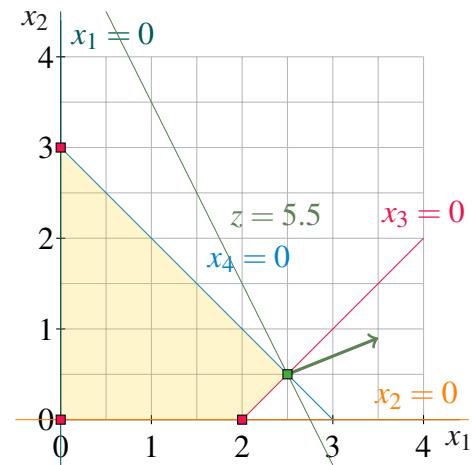


From the graph and tableau we can see that we are at the extreme point  $(0,0)$  and if we increase  $x_1$  by one unit, the objective function ( $z$ -value) increases by 2, thus improving the solution. We can increase  $x_1$  by 2 and still remain in the feasible region, moving to extreme point  $(2,0)$ . Likewise, if we increase  $x_2$  by one unit the objective function increases by 1, and we can increase  $x_2$  by three and still remain in the feasible region, moving to extreme point  $(0,3)$ . As  $x_2$  increases the basic variable  $x_3$  gets larger, while the basic variable  $x_4$  gets smaller, so  $x_2$  enters the basis and  $x_4$  leaves.

max	$z$	$x_1$	$x_2$	$x_3$	$x_4$	$rhs$
$z$	1	0	-3	2	0	4
$x_1$	0	1	-1	1	0	2
$x_4$	0	0	2	-1	1	1



max	$z$	$x_1$	$x_2$	$x_3$	$x_4$	$rhs$
$z$	1	0	0	1/2	3/2	11/2
$x_1$	0	1	0	1/2	1/2	5/2
$x_2$	0	0	1	-1/2	1/2	1/2



Let's consider this algorithm, and what we know, and see if there are any missing parts, or other information we would find valuable.

- Unique optimal solution
- Multiple optimal solutions
- Unbounded optimal objective value
- Empty feasible region (an infeasible LP)

**Tableau Formulas:**

We can modify the tableau for a particular basis  $\mathbf{B}$  using the following formulas:

	$z$	$x_i$	$rhs$
(z)	1	$\mathbf{c}_B \mathbf{B}^{-1} \mathbf{a}_i - \mathbf{c}_i$	$\mathbf{c}_B \mathbf{B}^{-1} \mathbf{b}$
( $x_B$ )	0	$\mathbf{B}^{-1} \mathbf{a}_i$	$\mathbf{B}^{-1} \mathbf{b}$

If we partition the variables, the formulas simplify as follows:

	$z$	$x_B$	$x_N$	$rhs$
$z$	1	$\mathbf{c}_B \mathbf{B}^{-1} \mathbf{B} - \mathbf{c}_B = 0$	$\mathbf{c}_B \mathbf{B}^{-1} \mathbf{N} - \mathbf{c}_N$	$\mathbf{c}_B \mathbf{B}^{-1} \mathbf{b}$
$x_B$	0	$\mathbf{B}^{-1} \mathbf{B} = \mathbf{I}$	$\mathbf{B}^{-1} \mathbf{N}$	$\mathbf{B}^{-1} \mathbf{b}$

The formulas for the coefficients for rows 1- $m$ , that is  $\mathbf{B}^{-1} \mathbf{a}_i$  (or  $\mathbf{B}^{-1} \mathbf{A}$  for all the columns on the left-hand side) is fairly straight forward; Multiplying by  $\mathbf{B}^{-1}$  is essentially the same as doing the elementary row operations required to get an identity matrix in the basic variable columns.

Now consider the formula  $\mathbf{c}_B \mathbf{B}^{-1} \mathbf{a}_i - \mathbf{c}_i$  for the row 0 coefficients. Where did this come from?

Consider an expanded basis matrix  $\hat{\mathbf{B}}$ , which includes the  $z$ -variable column and row, as follows:  

$$\begin{bmatrix} 1 & -\mathbf{c}_B \\ 0 & \mathbf{B} \end{bmatrix}$$
, which yields  $\hat{\mathbf{B}}^{-1}$  of  $\begin{bmatrix} 1 & \mathbf{c}_B \mathbf{B}^{-1} \\ 0 & \mathbf{B}^{-1} \end{bmatrix}$ , and the column for  $x_i$  is  $[-c_i, \mathbf{a}_i]^T$ . Multiplying these yields  $\begin{bmatrix} 1 & \mathbf{c}_B \mathbf{B}^{-1} \\ 0 & \mathbf{B}^{-1} \end{bmatrix} \begin{bmatrix} -c_i \\ \mathbf{a}_i \end{bmatrix}$ , which results in dot product of  $[1, \mathbf{c}_B \mathbf{B}^{-1}] [-c_i, \mathbf{a}_i] = \mathbf{c}_B \mathbf{B}^{-1} \mathbf{a}_i - c_i$  for the first element of the resulting column vector.

For example, consider the following LP:

$$\begin{aligned} \text{Max } z &= 2x_1 + 3x_2 \\ \text{s.t. } 1x_1 + 1x_2 &\geq 2 \\ 4x_1 + 6x_2 &\leq 9 \\ x_1, x_2 &\geq 0. \end{aligned}$$

For this problem, if we have  $x_1$  ad  $x_2$  as the basic variables, then

$$\hat{\mathbf{B}} = \begin{bmatrix} 1 & -2 & -3 \\ 0 & 1 & 1 \\ 0 & 4 & 6 \end{bmatrix} \text{ and } \hat{\mathbf{B}}^{-1} = \begin{bmatrix} 1 & 0 & 1/2 \\ 0 & 3 & -1/2 \\ 0 & -2 & 1/2 \end{bmatrix},$$

and the column for  $x_1$  is  $[-2, 1, 4]^T$ .

When we multiply  $\hat{\mathbf{B}}^{-1}$  and  $[-2, 1, 4]^T$  we get

$$\begin{bmatrix} 1 & 0 & 1/2 & -2 & 0 \\ 0 & 3 & -1/2 & 1 & = 1 \\ 0 & -2 & 1/2 & 4 & 0 \end{bmatrix}$$

The simplex algorithm again:

1. Put the LP into a standard form tableau, find a feasible basis  $\mathbf{B}$  and modify the tableau using  $\mathbf{B}^{-1}$  and the tableau formulas.
2. Check optimality, if  $\mathbf{c}_B \mathbf{B}^{-1} \mathbf{a}_i - \mathbf{c}_i \geq (\leq) 0$  for  $i : x_i \in \mathbf{x}_N$  for a maximization (minimization) problem, then the current basic solution is optimal. Stop.
3. Select an entering nonbasic variable using Dantzig's rule, specifically, entering variable  $x_i$ , where  
 $i = \min(\max)_i \{ \mathbf{c}_B \mathbf{B}^{-1} \mathbf{a}_i - \mathbf{c}_i < (>) 0 : (i : x_i \in \mathbf{x}_N) \}$  for a maximization (minimization) problem.
4. Select a variable to leave the basis using the ratio test. For entering variable  $x_i$  the leaving variable is the basic variable corresponding to row  $j$ , where :  
 $\min_j \{ [\mathbf{B}^{-1} \mathbf{b}]_j / [\mathbf{B}^{-1} \mathbf{a}_i]_j, (j : j = 1, \dots, m, [\mathbf{B}^{-1} \mathbf{a}_i]_j > 0) \}.$
5. Put the tableau into the proper form for the new basis and go to Step 2.

**Finding an Initial BFS** When a basic feasible solution is not apparent, we can produce one using *artificial variables*. This *artificial* basis is undesirable from the perspective of the original problem, we do not want the artificial variables in our solution, so we penalize them in the objective function, and allow the simplex algorithm to drive them to zero (if possible) and out of the basis. There are two such methods, the **Big M method** and the **Two-phase method**, which we illustrate below:

Solve the following LP using the Big M Method and the simplex algorithm:

$$\begin{aligned} \max \quad & z = 9x_1 + 6x_2 \\ \text{s.t.} \quad & 3x_1 + 3x_2 \leq 9 \\ & 2x_1 - 2x_2 \geq 3 \\ & 2x_1 + 2x_2 \geq 4 \\ & x_1, x_2 \geq 0. \end{aligned}$$

Here is the LP is transformed into standard form by using slack variables  $x_3$ ,  $x_4$ , and  $x_5$ , with the required artificial variables  $x_6$  and  $x_7$ , which allow us to easily find an initial basic feasible solution (to the

artificial problem).

$$\begin{aligned}
 \max \quad & z_a = 9x_1 + 6x_2 - Mx_6 - Mx_7 \\
 \text{s.t.} \quad & 3x_1 + 3x_2 + x_3 = 9 \\
 & 2x_1 - 2x_2 - x_4 + x_6 = 3 \\
 & 2x_1 + 2x_2 - x_5 + x_7 = 4 \\
 & x_i \geq 0, \quad i = 1, \dots, 7.
 \end{aligned}$$

$z$	$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	$x_6$	$x_7$	RHS	ratio
1	-9	-6	0	0	0	M	M	0	
0	3	3	1	0	0	0	0	9	
0	2	-2	0	-1	0	1	0	3	
0	2	2	0	0	-1	0	1	4	

This tableau is not in the correct form, it does not represent a basis, the columns for the artificial variables need to be adjusted.

$z$	$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	$x_6$	$x_7$	RHS	ratio
1	-9 - 4M	-6	0	M	M	0	0	-7M	
0	3	3	1	0	0	0	0	9	3
0	2	-2	0	-1	0	1	0	3	3/2
0	2	2	0	0	-1	0	1	4	2

The current solution is not optimal, so  $x_1$  enters the basis, and by the ratio test,  $x_6$  (an artificial variable) leaves the basis.

$z$	$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	$x_6$	$x_7$	RHS	ratio
1	0	-15 - 4M	0	-9/2 - M	M	9/2 + 2M	0	27/2 - M	
0	0	6	1	3/2	0	-3/2	0	3/2	3/4
0	1	-1	0	-1/2	0	1/2	0	3/2	-
0	0	4	0	1	-1	-1	1	1	1/4

The current solution is not optimal, so  $x_2$  enters the basis, and by the ratio test,  $x_7$  (an artificial variable) leaves the basis.

$z$	$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	$x_6$	$x_7$	RHS	ratio
1	0	0	0	-3/4	-15/4	-	-	17 1/4	
0	0	0	1	0	3/2	0	-3/2	3	-
0	1	0	0	-1/4	-1/4	1/2	1/4	7/4	-
0	0	1	0	1/4	-1/4	-1/4	1/4	1/4	1

The current solution is not optimal, so  $x_4$  enters the basis, and by the ratio test,  $x_2$  leaves the basis.

$z$	$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	$x_6$	$x_7$	RHS	ratio
1	0	3	0	0	-9/2	-	-	18	
0	0	0	1	0	3/2	0	-3/2	3	-
0	1	1	0	0	-1/2	0	1/2	2	-
0	0	4	0	1	-1	-1	1	1	1

The current solution is not optimal, so  $x_5$  enters the basis, and by the ratio test,  $x_3$  leaves the basis.

$z$	$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	$x_6$	$x_7$	RHS	ratio
1	0	3	3	0	0	-	-	27	
0	0	0	2/3	0	1	0	-1	2	
0	1	1	1/3	0	0	0	0	3	
0	0	4	2/3	1	0	-1	0	3	

The current solution is optimal!

Solve the following LP using the Two-phase Method and Simplex Algorithm.

$$\begin{aligned} \max z &= 2x_1 + 3x_2 \\ \text{s.t. } & 3x_1 + 3x_2 \geq 6 \\ & 2x_1 - 2x_2 \leq 2 \\ & -3x_1 + 3x_2 \leq 6 \\ & x_1, x_2 \geq 0. \end{aligned}$$

Here is first phase LP (in standard form), where  $x_3$ ,  $x_4$ , and  $x_5$  are slack variables, and  $x_6$  is an artificial variable.

$$\begin{aligned} \min z_a &= x_6 \\ \text{s.t. } & 3x_1 + 3x_2 - x_3 + x_6 = 6 \\ & 2x_1 - 2x_2 + x_4 = 2 \\ & -3x_1 + 3x_2 + x_5 = 6 \\ & x_i \geq 0, \quad i = 1, \dots, 6. \end{aligned}$$

Next, we put the LP into a tableau, which, still is not in the right form for our basic variables ( $x_6$ ,  $x_4$ , and  $x_5$ ).

$z$	$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	$x_6$	RHS	ratio
1	0	0	0	0	0	-1	0	
0	3	3	-1	0	0	1	6	
0	2	-2	0	1	0	0	2	
0	-3	3	0	0	1	0	6	

To remedy this, we use row operation to modify the row 0 coefficients, yielding the following:

$z$	$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	$x_6$	RHS	ratio
1	3	3	-1	0	0	0	6	
0	3	3	-1	0	0	1	6	2
0	2	-2	0	1	0	0	2	-
0	-3	3	0	0	1	0	6	2

The current solution is not optimal, either  $x_1$  or  $x_2$  can enter the basis, let's choose  $x_2$ . Then by the ratio test, either  $x_6$  (an artificial variable) or  $x_5$  (a slack variable) can leave the basis. Let's choose  $x_6$ .

$z$	$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	$x_6$	RHS	ratio
1	0	0	0	0	0	-1	0	
0	1	1	-1/3	0	0	1/3	2	
0	4	0	-2/3	1	0	2/3	6	
0	-6	0	1	0	1	-1	0	

The current solution is optimal, so we end the first phase with a basic feasible solution to the original problem, with  $x_2$ ,  $x_4$ , and  $x_5$  as the basic variables. Now we provide a new row zero that corresponds to the original problem.

$z$	$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	$x_6$	RHS	ratio
1	1	0	-1	0	0	0	6	
0	1	1	-1/3	0	0	1/3	2	
0	4	0	-2/3	1	0	2/3	6	
0	-6	0	1	0	1	-1	0	

$z$	$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	$x_6$	RHS	ratio
1	-5	0	0	0	1	-1	6	
0	-1	1	0	0	1/3	0	2	
0	0	0	0	1	2/3	0	6	
0	-6	0	1	0	1	-1	0	

From this tableau we can see that the LP is unbounded and an extreme point is  $[0, 2, 0, 6, 0]$  and an extreme direction is  $[1, 1, 6, 0, 0]$ .

### Degeneracy and the Simplex Algorithm

Degeneracy must be considered in the simplex algorithm, as it causes some trouble. For instance, it might mislead us into thinking there are multiple optimal solutions, or provide faulty insight. Further, the algorithm as described can *cycle*, that is, remain on a degenerate extreme point repeatedly cycling through a subset of bases that represent that point, never leaving.

min	$z$	$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	$x_6$	$x_7$	$rhs$
	1	0	0	0	3/4	-20	1/2	-6	0
	0	1	0	0	1/4	-8	-1	9	0
	0	0	1	0	1/2	-12	-1/2	3	0
	0	0	0	1	0	0	1	0	1

Solve the following LP using the Simplex Algorithm:

$$\begin{aligned} \max \quad & z = 40x_1 + 30x_2 \\ \text{s.t.} \quad & 6x_1 + 4x_2 \leq 40 \\ & 4x_1 + 2x_2 \leq 20 \\ & x_1, x_2 \geq 0. \end{aligned}$$

By adding slack variables, we have the following tableau. Luckily, this tableau represents a basis,

$z$	$x_1$	$x_2$	$s_1$	$s_2$	RHS
1	-40	-30	0	0	0
0	6	4	1	0	40
0	4	2	0	1	20

where  $\text{BV}=\{s_1, s_2\}$ , but by inspecting the row 0 (objective function row) coefficients, we can see that this is not optimal. By Dantzig's Rule, we enter  $x_1$  into the basis, and by the ratio test we see that  $s_2$  leaves the basis. By performing elementary row operations, we obtain the following tableau for the new basis  $\text{BV}=\{s_1, x_1\}$ .

$z$	$x_1$	$x_2$	$s_1$	$s_2$	RHS
1	0	-10	0	10	200
0	0	1	1	-3/2	10
0	1	1/2	0	1/4	5

This tableau is not optimal, entering  $x_2$  into the basis can improve the objective function value. The basic variables  $s_1$  and  $x_1$  tie in the ration test. If we have  $x_1$  leave the basis, we get the following tableau ( $\text{BV}=\{s_1, x_2\}$ ).

$z$	$x_1$	$x_2$	$s_1$	$s_2$	RHS
1	20	0	0	15	300
0	-2	0	1	-2	0
0	2	1	0	1/2	10

This is an optimal tableau, with an objective function value of 300, If instead of  $x_1$  leaving the basis, suppose  $s_1$  left, this would lead to the following tableau ( $\text{BV}=\{x_2, x_1\}$ ).

$z$	$x_1$	$x_2$	$s_1$	$s_2$	RHS
1	0	0	10	-5	300
0	0	1	1	-3/2	10
0	1	0	-1/2	1	0

This tableau does not look optimal, yet the objective function value is the same as the optimal solution's. This occurs because the optimal extreme point is a degenerate.

### 6.2.4.2. Dual Simplex Algorithm

---

The dual simplex algorithm is essentially performing the simplex algorithm, on the dual problem, on the primal tableau. Remember, the Simplex algorithm, in relation to the KKT conditions, maintains primal feasibility, complementary slackness, and strives for dual feasibility. The dual simplex algorithm maintains dual feasibility, complementary slackness, and strives for primal feasibility.

1. Pick the row with the smallest  $\bar{b}_i$ , where  $\bar{b}_i < 0$  ( $\bar{\mathbf{b}} = \mathbf{B}^{-1}\mathbf{b}$ ), this corresponds to the leaving variable.
2. Pick a column with the minimum  $\{|z_j - c_j/y_{ij}| : y_{ij} < 0\}$ , this corresponds to the entering variable.
3. Pivot, and repeat until primal feasibility is achieved.

### 6.2.4.3. Primal-Dual Algorithm

---

The primal dual algorithm is another method for solving LPs. This algorithm starts with a feasible dual solution (not necessarily basic) and searches for a primal feasible solution will maintaining complementary slackness between the primal and dual solutions. Consider the following primal dual pair:

$$(P) : \min\{\mathbf{c}\mathbf{x} : \mathbf{Ax} = \mathbf{b}, \mathbf{x} \geq 0\}$$

$$(D) : \max\{\mathbf{w}\mathbf{b} : \mathbf{wA} \leq \mathbf{c}, \mathbf{w} \text{ urs}\}.$$

To solve  $P$  using the primal-dual algorithm, first find a feasible solution to  $D$ , it does not necessarily have to be basic, and form the following restricted primal problem:

$$(P_R) : \min\{z_R = \mathbf{1}\mathbf{x}^a : \mathbf{a}_j x_j + \mathbf{x}^a = \mathbf{b}, x_j \geq 0, j \in Q, \mathbf{x}^a \geq 0\},$$

where  $Q = \{j : w_a j = c_j\}$ , that is,  $Q$  is the set of indexes for binding dual constraints, and  $x_j, j \in Q$  are the primal variables that can be non-zero given the current dual solution and complementary slackness. The vector  $\mathbf{x}^a$  is a vector of artificial variables; this problem looks very similar to the phase 1 problem in the two-phase method. The objective is the same, to find a basic feasible primal solution, but here we use a *restricted* or limited number of primal variables, those with indexes in set  $Q$ .

Solve  $P_R$  (using simplex) and if  $z_R = 0$ , stop, the solution is optimal for  $P$  because we have satisfied the KKT conditions, else let  $(v)^*$  be the corresponding optimal dual solution  $D_R$ .

$$D_R : \max\{\mathbf{v}\mathbf{b} : \mathbf{v}\mathbf{a}_j \leq 0, i \in Q, \mathbf{v} \leq 1, \mathbf{v} \text{ urs}\}.$$

Note that for each  $j \in Q$ ,  $\mathbf{v}^*\mathbf{a}_j \leq 0$ . For  $i \notin Q$  calculate  $\mathbf{v}^*\mathbf{a}_i$ , if  $\mathbf{v}^*\mathbf{a}_i > 0$  then  $x_i$  can be added to the restricted primal to improve  $z_R$ . To get  $x_i$  into  $Q$  we must modify the original dual solution  $\mathbf{w}$ , first we calculate  $\theta$ .

$$\theta = \min_{i \notin Q} \{ |(\mathbf{w}\mathbf{a}_i - c_i)| / \mathbf{v}^* \mathbf{a}_i : \mathbf{v}^* \mathbf{a}_i > 0 \} > 0$$

We take the absolute value of  $\mathbf{w}\mathbf{a}_i - c_i$  because if the primal problem is a minimization, this term will always be non-positive, for a primal maximization this will always be non-negative (thus the absolute value is not needed).

We then replace  $\mathbf{w}$  by  $\mathbf{w} + \theta \mathbf{v}^*$ . We use this  $\theta$ -step like a ratio test, changing the current dual solution such that we can enter a new primal variable into the restricted primal, one that will improve the solution (and move us closer to feasibility), while maintaining dual feasibility and complementary slackness.

If we think about this algorithm on a tableau, we get the following formulas, where the  $z_P$  row corresponds to our dual solution, and defines the set  $Q$ , and simplex is performed on the restricted problem (using row  $z_P^R$  for the objective function row, and using column defined by  $Q$ ).

	$z$	$\mathbf{x}$	$\mathbf{x}^s$	$\mathbf{x}^a$	rhs
$z_P$	1	$\mathbf{w}\mathbf{a}_i - c_i$	$\mathbf{w}\mathbf{a}_i$	0	0
$z_P^R$	1	$\mathbf{v}\mathbf{a}_i$	$\mathbf{v}\mathbf{a}_i$	$\mathbf{v}\mathbf{a}_i - 1$	0
	0	$\mathbf{B}^{-1}\mathbf{a}_i$	$\mathbf{B}^{-1}\mathbf{a}_i$	$\mathbf{B}^{-1}\mathbf{a}_i$	$\mathbf{B}^{-1}\mathbf{b}$

### Example 6.2: Primal-dual algorithm

Consider again the following LP and solve using the primal-dual algorithm, using a starting feasible dual solution of  $\mathbf{w} = [10, 0, 0]$ .

#### Solution.

$$\begin{aligned}
 \max \quad & 18x_1 + 16x_2 + 10x_3 \\
 \text{s.t.} \quad & 2x_1 + 2x_2 + 1x_3 + x_4^s = 21 \quad (w_1) \\
 & 3x_1 + 2x_2 + 2x_3 + x_5^s = 23 \quad (w_2) \\
 & 1x_1 + 2x_2 + 1x_3 + x_6 = 17 \quad (w_3) \\
 & x_1, x_2, x_3, x_4, x_5, x_6 \geq 0.
 \end{aligned}$$

$$\begin{aligned}
 D1 : \min \quad & 21w_1 + 23w_2 + 17w_3 \\
 \text{s.t.} \quad & 2w_1 + 3w_2 + 1w_3 \geq 18 \quad (x_1) \\
 & 2w_1 + 2w_2 + 2w_3 \geq 16 \quad (x_2) \\
 & 1w_1 + 2w_2 + 1w_3 \geq 10 \quad (x_3) \\
 & w_1, w_2, w_3 \geq 0.
 \end{aligned}$$

Use the modified tableau method, and write  $Q$  and the restricted problem for each step. Be able to explain the complete process.

For  $\mathbf{w} = [10, 0, 0]$  we have  $Q = \{3, 5, 6\}$

	$x_1$	$x_2$	$x_3$	$x_4^s$	$x_5^s$	$x_6^s$	$x_7^a$	rhs
$z$	2	4	0	10	0	0	0	210
$z_R$	0	0	0	0	0	0	-1	0
	2	2	1	1	0	0	1	21
	3	2	2	0	1	0	0	23
	1	2	1	0	0	1	0	17

We need to make a minor adjustment to the artificial variable column.

	$z_P$	$x_1$	$x_2$	$x_3$	$x_4^s$	$x_5^s$	$x_6^s$	$x_7^a$	rhs
$z_P$	1	2	4	0	10	0	0	0	210
$z_R$	1	2	2	1	1	0	0	0	21
	0	2	2	1	1	0	0	1	21
	0	3	2	2	0	1	0	0	23
	0	1	2	1	0	0	1	0	17

$x_3$  enters the restricted basis and  $x_5^s$  leaves the restricted basis, resulting in the following tableau:

	$x_1$	$x_2$	$x_3$	$x_4^s$	$x_5^s$	$x_6^s$	$x_7^a$	rhs
$z$	2	4	0	10	0	0	0	210
$z_R$	1/2	1	0	1	-1/2	0	0	19/2
	1/2	1	0	1	-1/2	0	1	19/2
	3/2	1	1	0	1/2	0	0	23/2
	-1/2	1	0	0	-1/2	1	0	11/2

To find  $\Theta$  we take the minimum of  $4/1$  and  $2/(1/2)$ , which are equal, and thus  $\Theta = 4$ , using this we adjust the  $z$ -row, yielding the following tableau. Notice that this operation ensures that the  $z$ -row remains non-negative. Now  $Q = \{1, 2, 3, 6\}$ .

	$x_1$	$x_2$	$x_3$	$x_4^s$	$x_5^s$	$x_6^s$	$x_7^a$	rhs
$z$	0	0	0	6	2	0	0	172
$z_R$	1/2	1	0	1	-1/2	0	0	19/2
	1/2	1	0	1	-1/2	0	1	19/2
	3/2	1	1	0	1/2	0	0	23/2
	-1/2	1	0	0	-1/2	1	0	11/2

$x_2$  enters the basis (of the restricted problem) and  $x_6$  leaves.

	$x_1$	$x_2$	$x_3$	$x_4^s$	$x_5^s$	$x_6^s$	$x_7^a$	rhs
$z$	0	0	0	6	2	0	0	172
$z_R$	1	0	0	1	0	-1	0	4
	1	0	0	1	0	-1	1	4
	2	0	1	0	1	-1	0	6
	-1/2	1	0	0	-1/2	1	0	11/2

This is not an optimal solution, so now  $x_1$  enters the restricted basis and  $x_3$  leaves.

	$x_1$	$x_2$	$x_3$	$x_4^s$	$x_5^s$	$x_6^s$	$x_7^a$	rhs
$z$	0	0	0	6	2	0	0	172
$z_R$	0	0	-1/2	1	-1/2	-1/2	0	1
	0	0	-1/2	1	-1/2	-1/2	1	1
	1	0	1/2	0	1/2	-1/2	0	3
	0	1	1/4	0	-1/4	3/4	0	7

Here  $\Theta = 6$ , which yields the following, having Now  $Q = \{1, 2, 4\}$ ; we can see that  $x_4$  enters and  $a_1$  leaves, which will change the  $z_R$ -row, this makes the restricted primal optimal, but does not change any other rows, and thus this is the optimal solution.

	$x_1$	$x_2$	$x_3$	$x_4^s$	$x_5^s$	$x_6^s$	$x_7^a$	rhs
$z$	0	0	3	0	5	3	0	166
$z_R$	0	0	-1/2	1	-1/2	-1/2	0	1
	0	0	-1/2	1	-1/2	-1/2	1	1
	1	0	1/2	0	1/2	-1/2	0	3
	0	1	1/4	0	-1/4	3/4	0	7



## 6.2.5. Sensitivity Analysis

---

Consider an arbitrary LP, which we will call the primal ( $P$ ):

$$(P) : \max\{\mathbf{c}\mathbf{x} : \mathbf{Ax} \leq \mathbf{b}, \mathbf{x} \geq 0\},$$

where  $\mathbf{A}$  is an  $m \times n$  matrix, and  $\mathbf{x}$  is a  $n$  element column vector. Every primal LP has a related LP, which we call the dual, the dual of ( $P$ ) is:

$$(D) : \min\{\mathbf{w}\mathbf{b} : \mathbf{w}\mathbf{A} \geq \mathbf{c}, \mathbf{w} \geq 0\}.$$

If an LP has a different form from  $P$ , we can convert it to the above form to find the dual, or use the rules in the following table:

$\max \mathbf{c}\mathbf{x} :$

$$\mathbf{a}_1 * \mathbf{x} \leq b_1 \quad (w_1 \geq 0)$$

$$\mathbf{a}_2 * \mathbf{x} = b_2 \quad (w_2 \text{ urs})$$

$$\mathbf{a}_3 * \mathbf{x} \geq b_3 \quad (w_3 \leq 0)$$

$\vdots$

$$x_1 \geq 0, x_2 \text{ urs}, x_3 \leq 0, \dots$$

$\min \mathbf{w}\mathbf{b} :$

$$\mathbf{w}\mathbf{a}_1 \geq c_1 \quad (x_1 \geq 0)$$

$$\mathbf{w}\mathbf{a}_2 = c_2 \quad (x_2 \text{ urs})$$

$$\mathbf{w}\mathbf{a}_3 \leq c_3 \quad (x_3 \leq 0)$$

$\vdots$

$$w_1 \geq 0, w_2 \text{ urs}, w_3 \leq 0, \dots$$

Define  $\mathbf{w} = \mathbf{c}_B \mathbf{B}^{-1}$  as the vector of *shadow prices*, where  $w_i$  represents the change in the objective function value caused by a unit change to the associated  $b_i$  parameter (i.e., increasing the amount of resource  $i$  by one unit, see dual objective function).

Some observations:

- The dual of  $D$  is  $P$ .
- Each primal constraint has an associated dual variable ( $w_i$ ) and each dual constraint has an associated primal variable ( $x_i$ ).
- When the primal is a maximization, the dual is a minimization, and vice versa.

Consider the following primal tableau (where  $z_P$  is the primal objective function value) for  $(P) : \text{Max } \{\mathbf{c}\mathbf{x} : \mathbf{Ax} \leq \mathbf{b}, \mathbf{x} \geq 0\}$ .

	$z_P$	$x_i$	rhs
$z_P$	1	$\mathbf{c}_B \mathbf{B}^{-1} \mathbf{a}_i - c_i$	$\mathbf{c}_B \mathbf{B}^{-1} \mathbf{b}$
BV	0	$\mathbf{B}^{-1} \mathbf{a}_i$	$\mathbf{B}^{-1} \mathbf{b}$

Observe that if a basis for  $P$  is optimal, then the row zero coefficients for the variables are greater than, or equal to, zero, that is,  $c_B B^{-1} a_i - c_i \geq 0$  for each  $x_i$  (if the variable is a slack, this simplifies to  $c_B B^{-1} \geq 0$ ).

Substituting  $w = c_B B^{-1}$  we get  $\mathbf{wA} \geq \mathbf{c}, \mathbf{w} \geq 0$  which corresponds to dual feasibility.

$$(D) : \min \{\mathbf{wb} : \mathbf{wA} \geq \mathbf{c}, \mathbf{w} \geq 0\}.$$

### Weak Duality Property

If  $\mathbf{x}$  and  $\mathbf{w}$  are feasible solutions to  $P$  and  $D$ , respectively, then  $\mathbf{cx} \leq \mathbf{wAx} \leq \mathbf{wb}$ .

$$(P) : \max \{\mathbf{cx} : \mathbf{Ax} \leq \mathbf{b}, \mathbf{x} \geq 0\}.$$

$$(D) : \min \{\mathbf{wb} : \mathbf{wA} \geq \mathbf{c}, \mathbf{w} \geq 0\}.$$

This implies that the objective function value for a feasible solution to  $P$  is a lower bound on the objective function value for the optimal solution to  $D$ , and the objective function value for a feasible solution to  $D$  is an upper bound on the objective function value for the optimal solution to  $P$ .

Thus if the objective function values are equal, i.e.,  $\mathbf{cx} = \mathbf{wb}$ , then the solutions  $\mathbf{x}$  and  $\mathbf{w}$  are optimal.

**Theorem 6.3: Fundamental Theorem of Duality**

or problems  $P$  and  $D$  (i.e., any primal dual set) exactly one of the following is true:

1. Both have optimal solutions  $\mathbf{x}$  and  $\mathbf{w}$  where  $\mathbf{c}\mathbf{x} = \mathbf{w}\mathbf{b}$ .
2. One problem is unbounded (i.e., the objective function value can become arbitrarily large for a maximization, or arbitrarily small for a minimization), and the other is infeasible.
3. Both are infeasible.

**Theorem 6.4: Farkás Lemma**

Consider the following two systems:

1.  $\mathbf{Ax} \geq 0, \mathbf{cx} < 0$ .
2.  $\mathbf{wA} = \mathbf{c}, \mathbf{w} \geq 0$ .

Exactly one of these systems has a solution.

**Suppose system 1 has  $\mathbf{x}$  as a solution:**

- If  $\mathbf{w}$  were a solution to system 2, then post-multiplying each side of  $\mathbf{wA} = \mathbf{c}$  by  $\mathbf{x}$  would yield  $\mathbf{wAx} = \mathbf{cx}$ .
- Since  $\mathbf{Ax} \geq 0$  and  $\mathbf{w} \geq 0$ , this implies that  $\mathbf{cx} \geq 0$ , which violates  $\mathbf{cx} < 0$ .
- Thus we show that if system 1 has a solution, system 2 cannot have one.

**Suppose system 1 has no solution:**

- Consider the following LP:  $\min\{\mathbf{cx} : \mathbf{Ax} \geq 0\}$ .
- The optimal solution is  $\mathbf{cx} = 0$  and  $\mathbf{x} = 0$ .
- The LP in standard form (substitute  $\mathbf{x} = \mathbf{x}' - \mathbf{x}''$ ,  $\mathbf{x}' \geq 0$  and  $\mathbf{x}'' \geq 0$  and add  $\mathbf{x}^s \geq 0$ ) follows:  

$$\min\{\mathbf{cx}' - \mathbf{cx}'' : \mathbf{Ax}' - \mathbf{Ax}'' - \mathbf{x}^s = 0, \mathbf{x}', \mathbf{x}'', \mathbf{x}^s \geq 0\}$$
- $\mathbf{x}' = 0, \mathbf{x}'' = 0, \mathbf{x}^s = 0$  is an optimal extreme point solution.
- Using  $\mathbf{x}^s$  as an initial feasible basis, solve with the simplex algorithm (with cycling prevention) to find a basis where  $\mathbf{c}_B \mathbf{B}^{-1} \mathbf{a}_i - c_i \leq 0$  for all variables. Define  $\mathbf{w} = \mathbf{c}_B \mathbf{B}^{-1}$ .
- This yields  $\mathbf{wA} - \mathbf{c} \leq 0, -\mathbf{wA} + \mathbf{c} \leq 0, -\mathbf{w} \leq 0\}$ , from the columns for variables  $\mathbf{x}', \mathbf{x}'', \mathbf{x}^s$ , respectively. Thus,  $\mathbf{w} \geq 0$  and  $\mathbf{wA} = \mathbf{c}$ , and system 2 has a solution.

### Karush-Kuhn-Tucker (KKT) Conditions

$$(P) : \max\{\mathbf{c}\mathbf{x} : \mathbf{Ax} \leq \mathbf{b}, \mathbf{x} \geq 0\}.$$

$$(D) : \min\{\mathbf{w}\mathbf{b} : \mathbf{wA} \geq \mathbf{c}, \mathbf{w} \geq 0\}.$$

For problems  $P$  and  $D$ , with solutions  $\mathbf{x}$  and  $\mathbf{w}$ , respectively, we have the following conditions, which for LPs are necessary and sufficient conditions for optimality:

1.  $\mathbf{Ax} \leq \mathbf{b}, \mathbf{x} \geq 0$  (primal feasibility).
2.  $\mathbf{wA} \geq \mathbf{c}, \mathbf{w} \geq 0$  (dual feasibility).
3.  $\mathbf{w}(\mathbf{Ax} - \mathbf{b}) = 0$  and  $\mathbf{x}(\mathbf{c} - \mathbf{wA}) = 0$  (complementary slackness).

Note we can rewrite the third condition as  $\mathbf{w}(\mathbf{Ax} - \mathbf{b}) = \mathbf{wx}^s = 0$  and  $\mathbf{x}(\mathbf{c} - \mathbf{wA}) = \mathbf{xw}^s = 0$ , where  $\mathbf{x}^s$  and  $\mathbf{w}^s$  are the slack variables for the primal and dual problems, respectively.

### Why do the KKT conditions hold?

Suppose that the LP  $\min\{\mathbf{c}\mathbf{x} : \mathbf{Ax} \geq \mathbf{b}, \mathbf{x} \geq 0\}$  has an optimal solution  $\mathbf{x}^*$  (the dual is  $\max\{\mathbf{w}\mathbf{b} : \mathbf{wA} \leq \mathbf{c}, \mathbf{w} \geq 0\}$ ).

- Since  $\mathbf{x}^*$  is optimal there is no direction  $\mathbf{d}$  such that  $\mathbf{c}(\mathbf{x}^* + \lambda \mathbf{d}) < \mathbf{c}\mathbf{x}^*, \mathbf{A}(\mathbf{x}^* + \lambda \mathbf{d}) \geq \mathbf{b}$ , and  $\mathbf{x}^* + \lambda \mathbf{d} \geq 0$  for  $\lambda > 0$ .
- Let  $\mathbf{Gx} \geq \mathbf{g}$  be the binding inequalities in  $\mathbf{Ax} \geq \mathbf{b}$  and  $\mathbf{x} \geq 0$  for solution  $\mathbf{x}^*$  that is,  $\mathbf{Gx}^* = \mathbf{g}$ .
- Based on the optimality of  $\mathbf{x}^*$ , there is no direction  $\mathbf{d}$  at  $\mathbf{x}^*$  such that  $\mathbf{cd} < 0$  and  $\mathbf{Gd} \geq 0$  (else we could improve the solution).
- Based on Farka's Lemma, if the system  $\mathbf{cd} < 0, \mathbf{Gd} \geq 0$  does not have a solution, the system  $\mathbf{wG} = \mathbf{c}, \mathbf{w} \geq 0$  must have a solution.
- $\mathbf{G}$  is composed of rows from  $\mathbf{A}$  where  $\mathbf{a}_{i^*}\mathbf{x}^* = b_i$  and vectors  $\mathbf{e}_i$  for any  $x_i^* = 0$ .
- We can divide the  $\mathbf{w}$  into two sets:
  - $\{w_i, i : \mathbf{a}_{i^*}\mathbf{x}^* = b_i\}$  - those corresponding to the binding functional constraints in the primal.
  - $\{w_i^s, j : x_i^* = 0\}$  - those corresponding to the binding non-negativity constraints in the primal.
- Thus  $\mathbf{G}$  has the columns  $\mathbf{a}_{i^*}^T$  for  $w_i$  and  $\mathbf{e}_i^T$  for  $w_i^s$ .
- Since  $\mathbf{wG} = \mathbf{c}, \mathbf{w} \geq 0$  must have a solution, this solution is feasible for  $\mathbf{wA} \leq \mathbf{c}, \mathbf{w} \geq 0$  where  $w_i^s$  are added slacks. Thus,  $\mathbf{G}$  is missing some columns from  $\mathbf{A}$  (and thus some  $w$  variables) and some slack variables if  $\mathbf{wA} \leq \mathbf{c}, \mathbf{w} \geq 0$  were put into standard form, but those are not needed for feasibility based on the result, and thus can be thought of as set to zero, giving us complementary slackness.

**Example:****Example 6.5: Production LP**

Consider a production LP (the primal  $P$ ) where the variables represent the amount of three products to produce, using three resources, represented by the functional constraints. In standard form  $P$  and  $D$  have  $x_4^s, x_5^s, x_6^s$  and  $w_4^s, w_5^s, w_6^s$  as slack variables, respectively.

**Solution.** Decision variables:

$x_i$  : number of units of product  $i$  to produce,  $\forall i = \{1, 2, 3\}$ .

$$(P) : \begin{aligned} & \max z_P = 18x_1 + 16x_2 + 10x_3 \\ & \text{s.t. } 2x_1 + 2x_2 + 1x_3 + x_4^s = 21 \quad (w_1) \\ & \quad 3x_1 + 2x_2 + 2x_3 + x_5^s = 23 \quad (w_2) \\ & \quad 1x_1 + 2x_2 + 1x_3 + x_6^s = 17 \quad (w_3) \\ & \quad x_1, x_2, x_3, x_4^s, x_5^s, x_6^s \geq 0. \end{aligned}$$

$$(D) : \begin{aligned} & \min z_D = 21w_1 + 23w_2 + 17w_3 \\ & \text{s.t. } 2w_1 + 3w_2 + 1w_3 \geq 18 \quad (x_1) \\ & \quad 2w_1 + 2w_2 + 2w_3 \geq 16 \quad (x_2) \\ & \quad 1w_1 + 2w_2 + 1w_3 \geq 10 \quad (x_3) \\ & \quad 1w_1 \geq 0 \\ & \quad 1w_2 \geq 0 \\ & \quad 1w_3 \geq 0 \\ & \quad w_1, w_2, w_3 \text{ urs.} \end{aligned}$$

**Decision variables:**

$w_i$  : unit selling price for resource  $i$ ,  $\forall i = \{1, 2, 3\}$ .

$$(D) : \begin{aligned} & \min z_D = 21w_1 + 23w_2 + 17w_3 : \\ & \quad 2w_1 + 3w_2 + 1w_3 - w_4^s = 18 \quad (x_1) \\ & \quad 2w_1 + 2w_2 + 2w_3 - w_5^s = 16 \quad (x_2) \\ & \quad 1w_1 + 2w_2 + 1w_3 - w_6^s = 10 \quad (x_3) \\ & \quad w_1, w_2, w_3, w_4^s, w_5^s, w_6^s \geq 0. \end{aligned}$$

The initial basic feasible tableau for the primal, i.e., having the slack variables form the basis, follows:

$P : \max$	$z_P$	$x_1$	$x_2$	$x_3$	$x_4^s$	$x_5^s$	$x_6^s$	rhs
$z_P$	1	-18	-16	-10	0	0	0	0
$x_4^s$	0	2	2	1	1	0	0	21
$x_5^s$	0	3	2	2	0	1	0	23
$x_6^s$	0	1	2	1	0	0	1	17

$$x_1, x_2, x_3 = 0, x_4^s = 21, x_5^s = 23, x_6^s = 17 z_P = 0$$

The following dual tableau **conforms with the primal tableau through complementary slackness**.

$D : \min$	$z_D$	$w_1$	$w_2$	$w_3$	$w_4^s$	$w_5^s$	$w_6^s$	rhs
$z_D$	1	-21	-23	-17	0	0	0	0
$w_4^s$	0	-2	-3	-1	1	0	0	-18
$w_5^s$	0	-2	-2	-2	0	1	0	-16
$w_6^s$	0	-1	-2	-1	0	0	1	-10

$$w_1, w_2, w_3 = 0, w_4^s = -18, w_5^s = -16, w_6^s = -10 z_D = 0$$

**Complementary slackness:**  $w_1 x_4^s = 0, w_2 x_5^s = 0, w_3 x_6^s = 0, x_1 w_4^s = 0, x_2 w_5^s = 0, x_3 w_6^s = 0$ .

- If a primal variable is basic, then its corresponding dual variable must be nonbasic, and vice versa.
- The primal is suboptimal, and the dual tableau has a basic infeasible solution.
- Row 0 of the primal tableau has dual variable values in the corresponding primal variable columns.

The primal basis is not optimal, so enter  $x_1$  into the basis, and remove  $x_5^s$ , which yields:

P: Max	$z_P$	$x_1$	$x_2$	$x_3$	$x_4^s$	$x_5^s$	$x_6^s$	rhs
$z_P$	1	0	-4	2	0	6	0	138
$x_4^s$	0	0	2/3	-1/3	1	-2/3	0	17/3
$x_1$	0	1	2/3	2/3	0	1/3	0	23/3
$x_6^s$	0	0	4/3	1/3	0	-1/3	1	28/3

D: Min	$z_D$	$w_1$	$w_2$	$w_3$	$w_4^s$	$w_5^s$	$w_6^s$	rhs
$z_D$	1	-17/3	0	-28/3	-23/3	0	0	138
$w_2$	0	2/3	1	1/3	-1/3	0	0	6
$w_5^s$	0	-2/3	0	-4/3	-2/3	1	0	-4
$w_6^s$	0	1/3	0	-1/3	-2/3	0	1	2

The primal tableau does not represent an optimal basic solution, and the dual tableau does not represent a feasible basic solution.

Using Dantzig's rule, we enter  $x_2$  into the basis, and using the ratio test we find that  $x_6^s$  leaves the basis. This change in basis yields the following tableau:

P: Max	$z_P$	$x_1$	$x_2$	$x_3$	$x_4^s$	$x_5^s$	$x_6^s$	rhs	
	$z_P$	1	0	0	3	0	5	3	166
	$x_4^s$	0	0	0	-1/2	1	-1/2	-1/2	1
	$x_1$	0	1	0	1/2	0	1/2	-1/2	3
	$x_2$	0	0	1	1/4	0	-1/4	3/4	7

D: Min	$z_D$	$w_1$	$w_2$	$w_3$	$w_4^s$	$w_5^s$	$w_6^s$	rhs	
	$z_D$	1	-1	0	0	-3	-7	0	166
	$w_2$	0	1/2	1	0	-1/2	1/4	0	5
	$w_3$	0	1/2	0	1	1/2	-3/4	0	3
	$w_6^s$	0	1/2	0	0	-1/2	-1/4	1	3

Decision variables:

$x_i$  : number of units of product  $i$  to produce,  $\forall i = \{1, 2, 3\}$ .

$$(P) : \max z_P = 18x_1 + 16x_2 + 10x_3 :$$

$$2x_1 + 2x_2 + 1x_3 + x_4^s = 21 \quad (w_1)$$

$$3x_1 + 2x_2 + 2x_3 + x_5^s = 23 \quad (w_2)$$

$$1x_1 + 2x_2 + 1x_3 + x_6^s = 17 \quad (w_3)$$

$$x_1, x_2, x_3, x_4^s, x_5^s, x_6^s \geq 0.$$



The LP  $\max\{\mathbf{c}\mathbf{x} : \mathbf{Ax} \leq \mathbf{b}, \mathbf{x} \geq 0\}$  has an optimal solution  $\mathbf{x}^*$  (the dual is  $\min\{\mathbf{wb} : \mathbf{wA} \geq \mathbf{c}, \mathbf{w} \geq 0\}$ ).

- Since  $\mathbf{x}^*$  is optimal there is no direction  $\mathbf{d}$  such that  $\mathbf{c}(\mathbf{x}^* + \lambda \mathbf{d}) > \mathbf{cx}^*$ ,  $\mathbf{A}(\mathbf{x}^* + \lambda \mathbf{d}) \leq \mathbf{b}$ , and  $\mathbf{x}^* + \lambda \mathbf{d} \geq 0$  for  $\lambda > 0$ .
- Let  $\mathbf{Gx} \leq \mathbf{g}$  be the binding inequalities in  $\mathbf{Ax} \leq \mathbf{b}$  and  $\mathbf{x} \geq 0$  for solution  $\mathbf{x}^*$ , that is,  $\mathbf{Gx}^* = \mathbf{g}$ .

For our example,

$$\mathbf{G|g} = \left[ \begin{array}{ccc|c} 3 & 2 & 2 & 23 \\ 1 & 2 & 1 & 17 \\ 0 & 0 & -1 & 0 \end{array} \right]$$

- Based on the optimality of  $\mathbf{x}^*$ , there is no direction  $\mathbf{d}$  at  $\mathbf{x}^*$  such that  $\mathbf{cd} > 0$  and  $\mathbf{Gd} \leq 0$  (this includes  $\mathbf{d} \leq 0$ ) (else we could improve the solution).
- From Farka's Lemma, if the system  $\mathbf{cd} > 0$ ,  $\mathbf{Gd} \leq 0$  does not have a solution, the system  $\mathbf{wG} = \mathbf{c}$ ,  $\mathbf{w} \geq 0$  must have a solution.

$$3w_2 + 1w_3 = 18 \quad (x_1)$$

$$2w_2 + 2w_3 = 16 \quad (x_2)$$

$$2w_2 + 1w_3 - w_6^s = 10 \quad (x_3)$$

$$w_2, w_3, w_6^s \geq 0.$$

D: Min	$z_D$	$w_1$	$w_2$	$w_3$	$w_4^s$	$w_5^s$	$w_6^s$	rhs	
	1	-1	0	0	-3	-7	0	166	
	$w_2$	0	1/2	1	0	-1/2	1/4	0	5
	$w_3$	0	1/2	0	1	1/2	-3/4	0	3
	$w_6^s$	0	1/2	0	0	-1/2	-1/4	1	3

**Challenge 1:** Solve the following LP (as represented in the tableau), using the given tableau as a starting point. Provide the details of the algorithm to do so, and make it valid for both maximization and minimization problems.

$D : \min$	$z_D$	$w_1$	$w_2$	$w_3$	$w_4^s$	$w_5^s$	$w_6^s$	rhs
	$z_D$	-21	-23	-17	0	0	0	0
	$w_4^s$	0	-2	-3	-1	1	0	-18
	$w_5^s$	0	-2	-2	-2	0	1	-16
	$w_6^s$	0	-1	-2	-1	0	0	-10

**Challenge 2:** Given the following optimal tableau to our production LP, we can buy 12 units of resource 2 for \$4 a unit. Should we, please provide the analysis needed to make this decision.

$P : \max$	$z_P$	$x_1$	$x_2$	$x_3$	$x_4^s$	$x_5^s$	$x_6^s$	rhs	
	$z_P$	1	0	0	3	0	5	3	166
	$x_4^s$	0	0	0	-1/2	1	-1/2	-1/2	1
	$x_1$	0	1	0	1/2	0	1/2	-1/2	3
	$x_2$	0	0	1	1/4	0	-1/4	3/4	7

**Buying more of a resource:** In the example problem, we currently have 23 units of the second resource,  $b_2 = 23$ . We want to know the range of  $b_2$ -values for which the current basis remains feasible. The formula  $\mathbf{B}^{-1}\mathbf{b}$  shows us the impact of changing  $b_2$  (which only changes the rhs column of the tableau). So, we set  $\mathbf{B}^{-1}\mathbf{b} \geq 0$  replacing 23 with unknown  $b_2$ , and solve.

$$\begin{bmatrix} 1 & -1/2 & -1/2 \\ 0 & 1/2 & -1/2 \\ 0 & -1/4 & 3/4 \end{bmatrix} \begin{bmatrix} 21 \\ b_2 \\ 17 \end{bmatrix} \geq 0 \Rightarrow 17 \leq b_2 \leq 25.$$

$P : \max$	$z_P$	$x_1$	$x_2$	$x_3$	$x_4^s$	$x_5^s$	$x_6^s$	rhs	
	$z_P$	1	0	0	3	0	5	3	166
	$x_4^s$	0	0	0	-1/2	1	-1/2	-1/2	1
	$x_1$	0	1	0	1/2	0	1/2	-1/2	3
	$x_2$	0	0	1	1/4	0	-1/4	3/4	7

If  $17 \leq b_2 \leq 25$ , then the current basis remains feasible (and optimal). The *shadow price* for this resource is 5, thus 10 is the break-even price for the two additional units of resource 2. If 4 units of resource 2 were for sale, what is the break-even price?

If we add these additional resources, and recalculate the tableau, we get the following:

$z$	$x_1$	$x_2$	$x_3$	$x_4^s$	$x_5^s$	$x_6^s$	rhs
1	0	0	3	0	5	3	186
0	0	0	-1/2	1	-1/2	-1/2	-1
0	1	0	1/2	0	1/2	-1/2	5
0	0	1	1/4	0	-1/4	3/4	6

This tableau looks optimal (see row zero), but the basis is infeasible. We can find a new basis that is feasible and still looks optimal using the *Dual Simplex Method*.

Here we find the current basic variable with the smallest negative *rhs* coefficient, in this case there is only one negative coefficient, and that is for  $x_4^s$ . This is the leaving variable.

To find the entering variable, use the ratio test and pivot. Note that in this case either  $x_3$  or  $x_6^s$  can enter (they tie in the ratio test), and either route leads to an optimal solution (there are multiple optimal solutions here). If we enter  $x_3$  into the basis we get the following tableau:

$z$	$x_1$	$x_2$	$x_3$	$x_4^s$	$x_5^s$	$x_6^s$	rhs
1	0	0	0	6	2	0	180
0	0	0	1	-2	1	1	2
0	1	0	0	1	0	-1	4
0	0	1	0	1/2	-1/2	1/2	11/2

Thus, the break-even price for the 4 units of resource 2 is 14.

**Adding a new constraint:** Given an optimal basis (i.e., tableau), what does adding a new constraint do? Consider a new resource having the following constraint:

$$3/2x_1 + 3/2x_2 + 3/2x_3 \leq 14.$$

We can enter this into the current optimal tableau:

$z$	$x_1$	$x_2$	$x_3$	$x_4^s$	$x_5^s$	$x_6^s$	$x_7^s$	rhs
1	0	0	3	0	5	3	0	166
0	0	0	-1/2	1	-1/2	-1/2	0	1
0	1	0	1/2	0	1/2	-1/2	0	3
0	0	1	1/4	0	-1/4	3/4	0	7
0	3/2	3/2	3/2	0	0	0	1	14

This tableau no longer looks like one having a basic solution, so using elementary row operations, we get the following:

$z$	$x_1$	$x_2$	$x_3$	$x_4^s$	$x_5^s$	$x_6^s$	$x_7^s$	rhs
1	0	0	3	0	5	3	0	166
0	0	0	-1/2	1	-1/2	-1/2	0	1
0	1	0	1/2	0	1/2	-1/2	0	3
0	0	1	1/4	0	-1/4	3/4	0	7
0	0	0	3/8	0	-3/8	-3/8	1	-1

This tableau is no longer feasible, so using dual simplex we obtain the following ( $x_7^s$  leaves the basis and  $x_6^s$  enters).

$z$	$x_1$	$x_2$	$x_3$	$x_4^s$	$x_5^s$	$x_6^s$	$x_7^s$	rhs
1	0	0	6	0	2	0	8	158
0	0	0	-1	1	0	0	-4/3	7/3
0	1	0	0	0	1	0	-4/3	13/3
0	0	1	1	0	-1	0	2	5
0	0	0	-1	0	1	1	-8/3	8/3

What if the constraint was

$$3/2x_1 + 3/2x_2 + 3/2x_3 = 14.$$

What if the constraint was

$$3/2x_1 + 3/2x_2 + 3/2x_3 = 18.$$

Using  $x_7$  as an artificial variable, we get the following tableau (an additional row labeled  $z_a$  is added for the phase 1 problem).

	$z$	$x_1$	$x_2$	$x_3$	$x_4^s$	$x_5^s$	$x_6^s$	$x_7^a$	RHS
$z$	1	0	0	3	0	5	3	0	166
$z_a$	0	0	0	0	0	0	0	-1	0
$x_4$	0	0	0	-1/2	1	-1/2	-1/2	0	1
$x_1$	0	1	0	1/2	0	1/2	-1/2	0	3
$x_2$	0	0	1	1/4	0	-1/4	3/4	0	7
$x_7$	0	0	0	3/8	0	-3/8	-3/8	1	3

Adjusting this tableau to represent the initial, artificial, basis yields:

	$z$	$x_1$	$x_2$	$x_3$	$x_4^s$	$x_5^s$	$x_6^s$	$x_7^a$	RHS
$z$	1	0	0	3	0	5	3	0	166
$z_a$	0	0	0	3/8	0	-3/8	-3/8	0	3
$x_4$	0	0	0	-1/2	1	-1/2	-1/2	0	1
$x_1$	0	1	0	1/2	0	1/2	-1/2	0	3
$x_2$	0	0	1	1/4	0	-1/4	3/4	0	7
$x_7$	0	0	0	3/8	0	-3/8	-3/8	1	3

Since the phase 1 problem is a minimization, we enter  $x_3$  into the basis, and remove  $x_1$ , this yields an optimal solution to the phase 1 problem having the artificial variable  $x_7$  in the basis, thus with the new constraint, the LP is infeasible.

**Adding a new variable:** Consider the following tableau, to which a new column has been added (corresponding to  $x_7$ ), given this column, please find the original data for this variable, i.e., the values of  $c_7$  and  $a_7$ .

$z$	$x_1$	$x_2$	$x_3$	$x_4^s$	$x_5^s$	$x_6^s$	$x_7$	RHS
1	0	0	3	0	5	3	-1	166
0	0	0	-1/2	1	-1/2	-1/2	0	1
0	1	0	1/2	0	1/2	-1/2	0	3
0	0	1	1/4	0	-1/4	3/4	1/2	7

## 6.2.6. Theory Applications

---

### Cutting Stock - Column Generation

Consider the Cutting Stock Problem, which we use to illustrate column generation:

Given a stock board of length  $q$  and demand  $d_i$  for boards of length  $l_i$  (where  $l_i \leq q$ ), you must cut the stock boards to satisfy this demand, while minimizing waste, i.e., the number of stock boards required to satisfy the demand.

Problem parameters:

$P$  set of cutting pattern indexes,  $\{1, 2, \dots, n\}$ .

$L$  set of board length indexes,  $\{1, 2, \dots, m\}$ .

$d_i$  demand for boards of length  $l_i$ ,  $i = 1, \dots, m$ .

$a_{ij}$  number of boards of length  $l_i$ , obtained when one stock board is cut using pattern  $j$ ,  $i \in L$ ,  $j \in P$ .

Decision variable:

$x_j$  number of stock boards to cut using pattern  $j \in P$ .

$$\begin{aligned} \text{Min } z &= \sum_{j \in P} x_j \\ \text{s.t. } \sum_{j \in P} a_{ij} x_j &\geq d_i, \forall i \in L \\ x_j &\geq 0, \forall j \in P. \end{aligned}$$

It can be difficult to enumerate all possible cutting patterns  $a_j$  (this set can be quite large).

Instead, solve a restricted problem (as follows) where  $P_R$  is a subset of  $P$  that provides a feasible solution:

$$\begin{aligned} \text{Min } z &= \sum_{j \in P_R} x_j \\ \text{s.t. } \sum_{j \in P_R} a_{ij} x_j &\geq d_i, \forall i \in L \\ x_j &\geq 0, \forall j \in P_R. \end{aligned}$$

The optimal solution to the restricted problem is a feasible solution to the full problem. We want to find a new cutting pattern that will allow us to improve the restricted problem solution. Recall that the optimality condition for a minimization is  $\mathbf{c}_B \mathbf{B}^{-1} \mathbf{a}_i - \mathbf{c}_i \leq 0$ , thus to improve the restricted problem we want a column defined by  $\mathbf{a}_i$  such that  $\mathbf{c}_B \mathbf{B}^{-1} \mathbf{a}_i - \mathbf{c}_i > 0$ . For this problem  $c_i = 1, i \in P$ .

To find this vector  $\mathbf{a}_i$  (a column in the simplex tableau and a cutting pattern), we use the optimal solution to the restricted primal, which defines  $\mathbf{c}_B \mathbf{B}^{-1}$ , we can solve the following integer program:

$$\begin{aligned} \text{Max } & \sum_{i \in L} \mathbf{c}_B \mathbf{B}^{-1} a_i \\ \text{s.t. } & \sum_{i \in L} l_i a_i \leq q, \\ & a_i \in \mathbb{Z}^{\geq 0}, \forall i \in L, \end{aligned}$$

where the  $a_i$ 's are the decision variables. This produces a new cutting pattern, which is then added to the restricted problem. This process continues until the sub-problem provides an optimal solution of zero.

### **Revenue Management - Shadow Prices**

Consider an airline with a hub-and-spoke route structure. We define a flight as one take-off and landing of an aircraft at a particular time, flights are usually given flight numbers. Ticket prices are based on the itinerary, where an itinerary is a specific flight or set of (connecting) flights and a booking class (reflated to rules for the ticket, e.g., refundability). The following figure illustrates seven possible combinations that can be used to build itineraries using connections through airport B, the hub (A-B, B-C, B-D, B-E, A-B-C, A-B-D, A-B-E).

The airline forecasts demand for all important itinerary. Here are the problem parameters.

#### Problem Parameters:

$F$  set of all flights (one take-off and landing of an aircraft, at a specific time).

$c_f$  capacity of flight  $f, \forall f \in F$ .

$I$  set of all itineraries (set of flights that customer uses) and booking class.

$I_f$  set of all itineraries on flight  $f, \forall f \in F$ .

$d_i$  demand for itinerary  $i, \forall i \in I$ .

$f_i$  fare for itinerary  $i, \forall i \in I$ .

#### Decision Variables:

$x_i$  # of passengers accepted for itinerary  $i, \forall i \in I$ .

The following linear program maximize the revenue:

$$\begin{aligned} \text{Max } & \sum_{i \in I} f_i x_i \\ \text{s.t.: } & x_i \leq d_i, \quad \forall i \in I \\ & \sum_{i \in I_f} x_i \leq c_f, \quad \forall f \in F \\ & x_i \geq 0, \quad \forall i \in I \end{aligned}$$



## **Part II**

# **Integer Programming**



# 7. Integer Programming Formulations

## Outcomes

- A. Learn classic integer programming formulations.
- B. Demonstrate different uses of binary and integer variables.
- C. Demonstrate the format for modeling an optimization problem with sets, parameters, variables, and the model.

In this section, we will describe classical integer programming formulations.

## Resources

- The AIMMS modeling has many great examples. It can be book found here:[AIMMS Modeling Book](#).
- For many real world examples, see this book *Case Studies in Operations Research Applications of Optimal Decision Making*, edited by Murty, Katta G. Or find it [here](#).

## 7.1 Knapsack Problem

The *knapsack problem*<sup>1</sup> can take different forms depending on if the variables are binary or integer. The binary version means that there is only one item of each item type that can be taken. This is typically illustrated as a backpack (knapsack) and some items to put into it (see Figure 7.1), but has applications in many contexts.

### Binary Knapsack Problem:

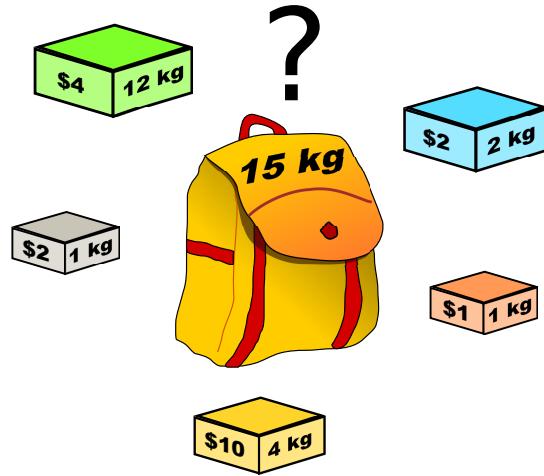
*NP-Complete*

Given a non-negative weight vector  $a \in \mathbb{Q}_+^n$ , a capacity  $b \in \mathbb{Q}_+$ , and objective coefficients  $c \in \mathbb{Q}^n$ ,

$$\begin{aligned} & \max c^\top x \\ & \text{s.t. } a^\top x \leq b \\ & \quad x \in \{0, 1\}^n \end{aligned} \tag{7.1}$$

<sup>1</sup>Video! - Michel Belaire (EPFL) teaching knapsack problem

<sup>2</sup>[wiki/File/knapsack](#), from [wiki/File/knapsack](#). [wiki/File/knapsack](#), [wiki/File/knapsack](#).



© wiki/File/knapsack<sup>2</sup>

**Figure 7.1: Knapsack Problem: which items should we choose take in the knapsack that maximizes the value while respecting the 15kg weight limit?**

### Example: Knapsack

Code

You have a knapsack (bag) that can only hold  $W = 15$  kgs. There are 5 items that you could possibly put into your knapsack. The items (weight, value) are given as: (12 kg, \$4), (2 kg, \$2), (1kg, \$2), (1kg, \$1), (4kg, \$10). Which items should you take to maximize your value in the knapsack? See Figure 7.1.

### Variables:

- let  $x_i = 0$  if item  $i$  is in the bag
- let  $x_i = 1$  if item  $i$  is not in the bag

### Model:

$$\begin{aligned}
 & \text{max } 4x_1 + 2x_2 + 2x_3 + 1x_4 + 10x_5 && \text{(Total value)} \\
 & \text{s.t. } 12x_1 + 2x_2 + 1x_3 + 1x_4 + 4x_5 \leq 15 && \text{(Capacity bound)} \\
 & x_i \in \{0,1\} \text{ for } i = 1, \dots, 5 && \text{(Item taken or not)}
 \end{aligned}$$

In the integer case, we typically require the variables to be non-negative integers, hence we use the notation  $x \in \mathbb{Z}_+^n$ . This setting reflects the fact that instead of single individual items, you have item types of which you can take as many of each type as you like that meets the constraint.

### **Integer Knapsack Problem:**

*NP-Complete*

Given a non-negative weight vector  $a \in \mathbb{Q}_+^n$ , a capacity  $b \in \mathbb{Q}_+$ , and objective coefficients  $c \in \mathbb{Q}^n$ ,

$$\begin{aligned} & \max c^\top x \\ \text{s.t. } & a^\top x \leq b \\ & x \in \mathbb{Z}_+^n \end{aligned} \tag{7.2}$$

We can also consider an equality constrained version

### **Equality Constrained Integer Knapsack Problem:**

*NP-Hard*

Given a non-negative weight vector  $a \in \mathbb{Q}_+^n$ , a capacity  $b \in \mathbb{Q}_+$ , and objective coefficients  $c \in \mathbb{Q}^n$ ,

$$\max c^\top x \tag{7.3}$$

$$\text{s.t. } a^\top x = b \tag{7.4}$$

$$x \in \mathbb{Z}_+^n \tag{7.5}$$

**Example 7.1:**

Using pennies, nickels, dimes, and quarters, how can you minimize the number of coins you need to make up a sum of 83¢?

**Variables:**

- Let  $p$  be the number of pennies used
- Let  $n$  be the number of nickels used
- Let  $d$  be the number of dimes used
- Let  $q$  be the number of quarters used

**Model**

$$\begin{array}{ll} \min & p + n + d + q && \text{total number of coins used} \\ \text{s.t.} & p + 5n + 10d + 25q = 83 && \text{sums to 83¢} \\ & p, d, n, q \in \mathbb{Z}_+ && \text{each is a non-negative integer} \end{array}$$

## 7.2 Capital Budgeting

---

The *capital budgeting* problem is a nice generalization of the knapsack problem. This problem has the same structure as the knapsack problem, except now it has multiple constraints. We will first describe the problem, give a general model, and then look at an explicit example.

**Capital Budgeting:**

A firm has  $n$  projects it could undertake to maximize revenue, but budget limitations require that not all can be completed.

- Project  $j$  expects to produce revenue  $c_j$  dollars overall.
- Project  $j$  requires investment of  $a_{ij}$  dollars in time period  $i$  for  $i = 1, \dots, m$ .
- The capital available to spend in time period  $i$  is  $b_i$ .

Which projects should the firm invest in to maximize its expected return while satisfying its weekly budget constraints?

We will first provide a general formulation for this problem.

### **Capital Budgeting Model:**

#### **Sets:**

- Let  $I = \{1, \dots, m\}$  be the set of time periods.
- Let  $J = \{1, \dots, n\}$  be the set of possible investments.

#### **Parameters:**

- $c_j$  is the expected revenue of investment  $j$  for  $j \in J$
- $b_i$  is the available capital in time period  $i$  for  $i$  in  $I$
- $a_{ij}$  is the resources required for investment  $j$  in time period  $i$ , for  $i$  in  $I$ , for  $j$  in  $J$ .

#### **Variables:**

- let  $x_i = 0$  if investment  $i$  is chosen
- let  $x_i = 1$  if investment  $i$  is not chosen

#### **Model:**

$$\begin{aligned}
 & \max \quad \sum_{j=1}^n c_j x_j && \text{(Total Expected Revenue)} \\
 & s.t. \quad \sum_{j=1}^n a_{ij} x_j \leq b_i, \quad i = 1, \dots, m && \text{(Resource constraint week } i) \\
 & \quad x_j \in \{0, 1\}, \quad j = 1, \dots, n
 \end{aligned}$$

Consider the example given in the following table.

Project	$\mathbb{E}[\text{Revenue}]$	Resources required in week 1	Resources required in week 2
1	10	3	4
2	8	1	2
3	6	2	1
Resources available		5	6

Given this data, we can setup our problem explicitly as follows

### Example: Capital Budgeting

Code

#### Sets:

- Let  $I = \{1, 2\}$  be the set of time periods.
- Let  $J = \{1, 2, 3\}$  be the set of possible investments.

#### Parameters:

- $c_j$  is given in column " $\mathbb{E}[\text{Revenue}]$ ".
- $b_i$  is given in row "Resources available".
- $a_{ij}$  given in row  $j$ , and column for week  $i$ .

#### Variables:

- let  $x_i = 0$  if investment  $i$  is chosen
- let  $x_i = 1$  if investment  $i$  is not chosen

The explicit model is given by

#### Model:

$$\begin{aligned}
 & \max \quad 10x_1 + 8x_2 + 6x_3 && \text{(Total Expected Revenue)} \\
 & s.t. \quad 3x_1 + 1x_2 + 2x_3 \leq 5 && \text{( Resource constraint week 1)} \\
 & \quad \quad \quad 4x_1 + 2x_2 + 1x_3 \leq 6 && \text{( Resource constraint week 2)} \\
 & \quad \quad \quad x_j \in \{0, 1\}, \quad j = 1, 2, 3
 \end{aligned}$$

## 7.3 Set Covering

---

### Resources

*Video! - Michel Belaire (EPFL) explaining set covering problem*

The *set covering* problem can be used for a wide array of problems. We will see several examples in this section.

#### Set Covering:

##### *NP-Complete*

Given a set  $V$  with subsets  $V_1, \dots, V_l$ , determine the smallest subset  $S \subseteq V$  such that  $S \cap V_i \neq \emptyset$  for all  $i = 1, \dots, l$ .

The set cover problem can be modeled as

$$\begin{aligned} & \min \quad 1^\top x \\ & \text{s.t. } \sum_{v \in V_i} x_v \geq 1 \text{ for all } i = 1, \dots, l \\ & \quad x_v \in \{0, 1\} \text{ for all } v \in V \end{aligned} \tag{7.1}$$

where  $x_v$  is a 0/1 variable that takes the value 1 if we include item  $j$  in set  $S$  and 0 if we do not include it in the set  $S$ .

Add flight crew scheduling example and images.

One specific type of set cover problem is the *vertex cover* problem.

#### Example: Vertex Cover:

##### *NP-Complete*

Given a graph  $G = (V, E)$  of vertices and edges, we want to find a smallest size subset  $S \subseteq V$  such that every for every  $e = (v, u) \in E$ , either  $u$  or  $v$  is in  $S$ .

We can write this as a mathematical program in the form:

$$\begin{aligned} & \min \quad 1^\top x \\ & \text{s.t. } x_u + x_v \geq 1 \text{ for all } (u, v) \in E \\ & \quad x_v \in \{0, 1\} \text{ for all } v \in V. \end{aligned} \tag{7.2}$$

**Example: Set cover: Fire station placement**

Code

In the fire station problem, we seek to choose locations for fire stations such that any district either contains a fire station, or neighbors a district that contains a fire station. Figure 7.2 depicts the set of districts and an example placement of locations of fire stations.

**Sets:**

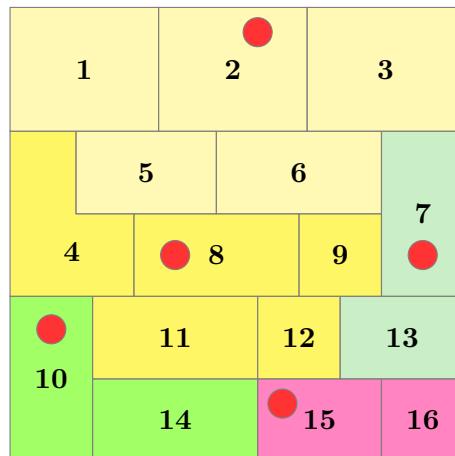
- Let  $V$  be the set of districts ( $V = \{1, \dots, 16\}$ )
- Let  $V_i$  be the set of districts that neighbor district  $i$  (e.g.  $V_1 = \{2, 4, 5\}$ ).

**Variables:**

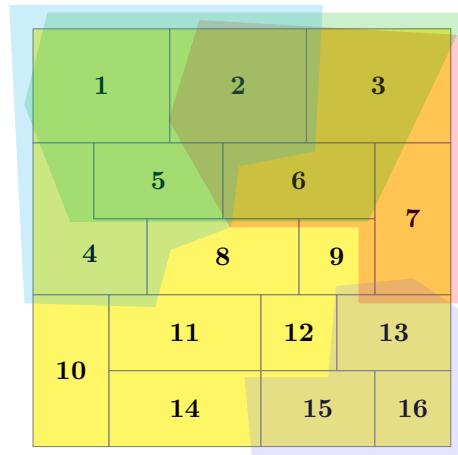
- let  $x_i = 1$  if district  $i$  is chosen to have a fire station.
- let  $x_i = 0$  otherwise.

**Model:**

$$\begin{aligned}
 & \min \sum_{i \in V} x_i && (\# \text{ open fire stations}) \\
 \text{s.t. } & x_i + \sum_{j \in V_i} x_j \geq 1 && \forall i \in V \quad (\text{Station proximity requirement}) \\
 & x_i \in \{0, 1\} && \text{for } i \in V \quad (\text{station either open or closed})
 \end{aligned}$$

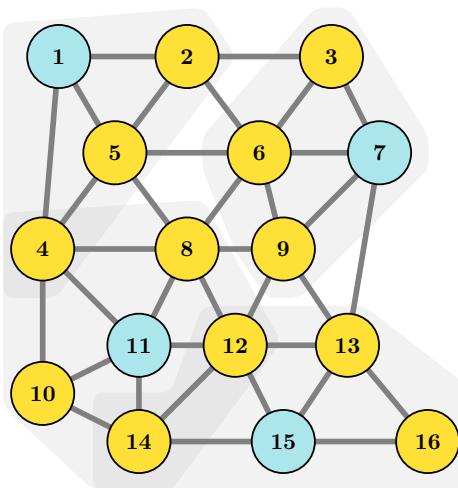
© tikz/Illustration1.pdf<sup>3</sup>

**Figure 7.2: Layout of districts and possible locations of fire stations.**



© tikz/Illustration2.pdf<sup>4</sup>

**Figure 7.3:** Set cover representation of fire station problem. For example, choosing district 16 to have a fire station covers districts 13, 15, and 16.



© tikz/Illustration3.pdf<sup>5</sup>

**Figure 7.4:** Graph representation of fire station problem. Every node is connected to a chosen node by an edge

?? ??

---

<sup>3</sup>tikz/Illustration1.pdf, from tikz/Illustration1.pdf. tikz/Illustration1.pdf, tikz/Illustration1.pdf.

<sup>4</sup>tikz/Illustration2.pdf, from tikz/Illustration2.pdf. tikz/Illustration2.pdf, tikz/Illustration2.pdf.

<sup>5</sup>tikz/Illustration3.pdf, from tikz/Illustration3.pdf. tikz/Illustration3.pdf, tikz/Illustration3.pdf.

**Set Covering - Matrix description:***NP-Complete*

Given a non-negative matrix  $A \in \{0, 1\}^{m \times n}$ , a non-negative vector, and an objective vector  $c \in \mathbb{R}^n$ , the set cover problem is

$$\begin{aligned} & \max c^\top x \\ & \text{s.t.. } Ax \geq 1 \\ & \quad x \in \{0, 1\}^n. \end{aligned} \tag{7.3}$$

**Example: Vertex Cover with matrix**

Code

An alternate way to solve ?? is to define the *adjacency matrix*  $A$  of the graph. The adjacency matrix is a  $|E| \times |V|$  matrix with  $\{0, 1\}$  entries. The each row corresponds to an edge  $e$  and each column corresponds to a node  $v$ . For an edge  $e = (u, v)$ , the corresponding row has a 1 in columns corresponding to the nodes  $u$  and  $v$ , and a 0 everywhere else. Hence, there are exactly two 1's per row. Applying the formulation above in Set Covering - Matrix description models the problem.

**7.3.1. Covering (Generalizing Set Cover)**

We could also allow for a more general type of set covering where we have non-negative integer variables and a right hand side that has values other than 1.

**Covering:***NP-Complete*

Given a non-negative matrix  $A \in \mathbb{Z}_+^{m \times n}$ , a non-negative vector  $b \in \mathbb{Z}^m$ , and an objective vector  $c \in \mathbb{R}^n$ , the set cover problem is

$$\begin{aligned} & \max c^\top x \\ & \text{s.t.. } Ax \geq b \\ & \quad x \in \mathbb{Z}_+^n. \end{aligned} \tag{7.4}$$

## 7.4 Assignment Problem

---

The *assignment problem* (machine/person to job/task assignment) seeks to assign tasks to machines in a way that is most efficient. This problem can be thought of as having a set of machines that can complete various tasks (textile machines that can make t-shirts, pants, socks, etc) that require different amounts of time to complete each task, and given a demand, you need to decide how to allocate your machines to tasks.

Alternatively, you could be an employer with a set of jobs to complete and a list of employees to assign to these jobs. Each employee has various abilities, and hence, can complete jobs in differing amounts of time. And each employee's time might cost a different amount. How should you assign your employees to jobs in order to minimize your total costs?

**Assignment Problem:**

Given  $m$  machines and  $n$  jobs, find a least cost assignment of jobs to machines not exceeding the machine capacities. Each job  $j$  requires  $a_{ij}$  units of machine  $i$ 's capacity  $b_i$ . Cost of assigning job  $j$  to machine  $i$  is  $c_{ij}$ .

Include picture, model, and example data

**Example: Machine Scheduling**

Code

**Sets:**

- 

**Parameters:**

- 

**Variables:**

- 

**Model:**

$$\begin{aligned}
 & \max \quad ? && (?) \\
 & s.t. \quad ? && (?) \\
 & \quad ? && (?) \\
 & \quad ?
 \end{aligned}$$

## 7.5 Facility Location

---

### Resources

- [Wikipedia - Facility Location Problem](#)
- See [GUROBI Modeling Examples - Facility Location](#).

The basic model of the facility location problem is to determine where to place your stores or facilities in order to be close to all of your customers and hence reduce the costs transportation to your customers. Each customer is known to have a certain demand for a product, and each facility has a capacity on how much of that demand it can satisfy. Furthermore, we need to consider the cost of building the facility in a given location.

This basic framework can be applied in many types of problems and there are a number of variants to this problem. We will address two variants: the *capacitated facility location problem* and the *uncapacitated facility location problem*.

### 7.5.1. Capacitated Facility Location

---

#### Capacitated Facility Location:

*NP-Complete*

Given costs connections  $c_{ij}$  and fixed building costs  $f_i$ , demands  $d_j$  and capacities  $u_i$ , the capacitated facility location problem is

$$\begin{aligned}
 \min \quad & \sum_{i=1}^n \sum_{j=1}^m c_{ij} y_{ij} + \sum_{i=1}^n f_i x_i \\
 \text{s.t.} \quad & \sum_{i=1}^n y_{ij} = 1 \text{ for all } j = 1, \dots, m \\
 & \sum_{j=1}^m d_j y_{ij} \leq u_i x_i \text{ for all } i = 1, \dots, n \\
 & y_{ij} \geq 0 \text{ for all } i = 1, \dots, n \text{ and } j = 1, \dots, m \\
 & x_i \in \{0, 1\} \text{ for all } i = 1, \dots, n
 \end{aligned} \tag{7.1}$$

Here  $M$  is a large number and can be chosen as  $M = m$ , but could be refined smaller if more context is known.

## 7.5.2. Uncapacitated Facility Location

---

### Uncapacitated Facility Location:

*NP-Complete*

Given costs connections  $c_{ij}$  and fixed building costs  $f_i$ , the uncapacitated facility location problem is

$$\begin{aligned} \min \quad & \sum_{i=1}^n \sum_{j=1}^m c_{ij} z_{ij} + \sum_{i=1}^n f_i x_i \\ \text{s.t.} \quad & \sum_{i=1}^n z_{ij} = 1 \text{ for all } j = 1, \dots, m \\ & \sum_{j=1}^m z_{ij} \leq M x_i \text{ for all } i = 1, \dots, n \\ & z_{ij} \in \{0, 1\} \text{ for all } i = 1, \dots, n \text{ and } j = 1, \dots, m \\ & x_i \in \{0, 1\} \text{ for all } i = 1, \dots, n \end{aligned} \tag{7.2}$$

Here  $M$  is a large number and can be chosen as  $M = m$ , but could be refined smaller if more context is known.

## 7.6 Network Flow

---

Fix up this section



### 7.6.1. Example - Multicommodity Flow

---

[https://en.wikipedia.org/wiki/Multi-commodity\\_flow\\_problem](https://en.wikipedia.org/wiki/Multi-commodity_flow_problem) The **multi-commodity flow problem** is a network flow problem with multiple commodities (flow demands) between different source and sink nodes.

**PROBLEM DEFINITION** Given a flow network  $G(V, E)$ , where edge  $(u, v) \in E$  has capacity  $c(u, v)$ . There are  $k$  commodities  $K_1, K_2, \dots, K_k$ , defined by  $K_i = (s_i, t_i, d_i)$ , where  $s_i$  and  $t_i$  is the **source** and **sink** of commodity  $i$ , and  $d_i$  is its demand. The variable  $f_i(u, v)$  defines the fraction of flow  $i$  along edge  $(u, v)$ , where  $f_i(u, v) \in [0, 1]$  in case the flow can be split among multiple paths, and  $f_i(u, v) \in \{0, 1\}$  otherwise (i.e. "single path routing"). Find an assignment of all flow variables which satisfies the following four constraints:

**(1) Link capacity:** The sum of all flows routed over a link does not exceed its capacity.

$$\forall (u, v) \in E : \sum_{i=1}^k f_i(u, v) \cdot d_i \leq c(u, v)$$

**(2) Flow conservation on transit nodes:** The amount of a flow entering an intermediate node  $u$  is the same that exits the node.

$$\sum_{w \in V} f_i(u, w) - \sum_{w \in V} f_i(w, u) = 0 \quad \text{when } u \neq s_i, t_i$$

**(3) Flow conservation at the source:** A flow must exit its source node completely.

$$\sum_{w \in V} f_i(s_i, w) - \sum_{w \in V} f_i(w, s_i) = 1$$

**(4) Flow conservation at the destination:** A flow must enter its sink node completely.

$$\sum_{w \in V} f_i(w, t_i) - \sum_{w \in V} f_i(t_i, w) = 1$$

## 7.6.2. Corresponding optimization problems

---

**Load balancing** is the attempt to route flows such that the utilization  $U(u, v)$  of all links  $(u, v) \in E$  is even, where

$$U(u, v) = \frac{\sum_{i=1}^k f_i(u, v) \cdot d_i}{c(u, v)}$$

The problem can be solved e.g. by minimizing  $\sum_{u, v \in V} (U(u, v))^2$ . A common linearization of this problem is the minimization of the maximum utilization  $U_{max}$ , where

$$\forall (u, v) \in E : U_{max} \geq U(u, v)$$

In the **minimum cost multi-commodity flow problem**, there is a cost  $a(u, v) \cdot f(u, v)$  for sending a flow on  $(u, v)$ . You then need to minimize

$$\sum_{(u, v) \in E} \left( a(u, v) \sum_{i=1}^k f_i(u, v) \right)$$

In the **maximum multi-commodity flow problem**, the demand of each commodity is not fixed, and the total throughput is maximized by maximizing the sum of all demands  $\sum_{i=1}^k d_i$

### 7.6.3. Relation to other problems

---

The minimum cost variant of the multi-commodity flow problem is a generalization of the minimum cost flow problem (in which there is merely one source  $s$  and one sink  $t$ ). Variants of the circulation problem are generalizations of all flow problems. That is, any flow problem can be viewed as a particular circulation problem.<sup>6</sup>

### 7.6.4. Usage

---

Routing and wavelength assignment (RWA) in optical burst switching of Optical Network would be approached via multi-commodity flow formulas.

## 7.7 Transportation Problem

---

Add discussion of transportation problem and picture.

Youtube! - TRANSPORTATION PROBLEM with PuLP in PYTHON

Notebook: Solution with Pyomo

## 7.8 Jobshop Scheduling: Makespan Minimization

---

Add discussion of some makespan minimization problems.

Wikipedia: Jobshop Scheduling

## 7.9 Generalized Assignment Problem (GAP)

---

Fix up this section

[https://en.wikipedia.org/wiki/Generalized\\_assignment\\_problem](https://en.wikipedia.org/wiki/Generalized_assignment_problem) In applied mathematics, the maximum **generalized assignment problem** is a problem in combinatorial optimization. This problem is a generalization of the assignment problem in which both tasks and agents have a size. Moreover, the size of each task might vary from one agent to the other.

This problem in its most general form is as follows: There are a number of agents and a number of tasks. Any agent can be assigned to perform any task, incurring some cost and profit that may vary depending on the agent-task assignment. Moreover, each agent has a budget and the sum of the costs of tasks assigned to it cannot exceed this budget. It is required to find an assignment in which all agents do not exceed their budget and total profit of the assignment is maximized.

### 7.9.1. In special cases

---

In the special case in which all the agents' budgets and all tasks' costs are equal to 1, this problem reduces to the assignment problem. When the costs and profits of all tasks do not vary between different agents, this problem reduces to the multiple knapsack problem. If there is a single agent, then, this problem reduces to the knapsack problem.

### 7.9.2. Explanation of definition

---

In the following, we have  $n$  kinds of items,  $a_1$  through  $a_n$  and  $m$  kinds of bins  $b_1$  through  $b_m$ . Each bin  $b_i$  is associated with a budget  $t_i$ . For a bin  $b_i$ , each item  $a_j$  has a profit  $p_{ij}$  and a weight  $w_{ij}$ . A solution is an assignment from items to bins. A feasible solution is a solution in which for each bin  $b_i$  the total weight of assigned items is at most  $t_i$ . The solution's profit is the sum of profits for each item-bin assignment. The goal is to find a maximum profit feasible solution.

Mathematically the generalized assignment problem can be formulated as an integer program:

$$\text{maximize } \sum_{i=1}^m \sum_{j=1}^n p_{ij}x_{ij}. \quad (7.1)$$

$$\text{subject to } \sum_{j=1}^n w_{ij}x_{ij} \leq t_i \quad i = 1, \dots, m; \quad (7.2)$$

$$\sum_{i=1}^m x_{ij} = 1 \quad j = 1, \dots, n; \quad (7.3)$$

$$x_{ij} \in \{0, 1\} \quad i = 1, \dots, m, \quad j = 1, \dots, n; \quad (7.4)$$

## 7.10 Other examples

---

- Sudoku
- AIMMS - Employee Training
- AIMMS - Media Selection
- AIMMS - Diet Problem
- AIMMS - Farm Planning Problem
- AIMMS - Pooling Probem
- INFORMS - Impact
- INFORMS - Success Story - Bus Routing

## 7.11 Modeling Tricks

---

In this section, we describe ways to model a variety of constraints that commonly appear in practice. The goal is changing constraints described in words to constraints defined by math.

### 7.11.1. Either Or Constraints

---

“*At least one of these constraints holds*” is what we would like to model. Equivalently, we can phrase this as an *inclusive or* constraint.

**Either Or:**

$$\text{Either } a^\top x \leq b \text{ or } c^\top x \leq d \text{ holds} \quad (7.1)$$

can be modeled as

$$\begin{aligned} a^\top x - b &\leq M_1 \delta \\ c^\top x - d &\leq M_2(1 - \delta) \\ \delta &\in \{0, 1\}, \end{aligned} \quad (7.2)$$

where  $M_1$  is an upper bound on  $a^\top x - b$  and  $M_2$  is an upper bound on  $c^\top x - d$ .

**Example 7.2**

Either 2 buses or 10 cars are needed shuttle students to the football game.

- Let  $x$  be the number of buses we have and
- let  $y$  be the number of cars that we have.

Suppose that there are at most  $M_1 = 5$  buses that could be rented and at most  $M_2 = 20$  cars that could be available.

This constraint can be modeled as

$$\begin{aligned} x - 2 &\leq 5\delta \\ y - 10 &\leq 20(1 - \delta) \\ \delta &\in \{0, 1\}, \end{aligned} \tag{7.3}$$

**7.11.2. If then implications**

If then implications are extremely useful in models. For instance, if we have more than 5 passengers, then we need to take two cars. Most if then statements can be modeled with by a constraint and an on/off flag. For example

$$\text{If } \delta = 1, \text{ then } a^\top x \leq b. \tag{7.4}$$

By letting  $M$  be an upper bound on the quantity  $a^\top x - b$ , we can model this condition as

$$\begin{aligned} a^\top x - b &\leq M(1 - \delta) \\ \delta &\in \{0, 1\} \end{aligned} \tag{7.5}$$

On the other hand, if we want to model the reverse implication, we have to be slightly more careful. We let  $m$  be a lower bound on the quantity  $a^\top x - b$  and we let  $\varepsilon$  be a tiny number that is an error bound in verifying if an inequality is violated. **If the data  $a, b$  are integer and  $x$  is an integer, then we can take  $\varepsilon = 1$ .**

Now

$$\text{If } a^\top x \leq b \text{ then } \delta = 1 \tag{7.6}$$

can be modeled as

$$a^\top x - b \geq \varepsilon(1 - \delta) + m\delta. \tag{7.7}$$

A simple way to understand this constraint is to consider the *contrapositive* of the if then statement that we want to model. The contrapositive says that

$$\text{If } \delta = 0, \text{ then } a^\top x - b > 0. \tag{7.8}$$

To show the contrapositive, we set  $\delta = 0$ . Then the inequality becomes

$$a^\top x - b \geq \varepsilon(1 - 0) + m0 = \varepsilon > 0.$$

Implication	Constraint
If $\delta = 0$ , then $a^\top x \leq b$	$a^\top x \leq b + M\delta$
If $a^\top x \leq b$ , then $\delta = 1$	$a^\top x \geq m\delta + \varepsilon(1 - \delta)$

**Table 7.1: Short list: If/then models with a constraint and a binary variable.** Here  $M$  and  $m$  are upper and lower bounds on  $a^\top x - b$  and  $\varepsilon$  is a small number such that if  $a^\top x > b$ , then  $a^\top x \geq b + \varepsilon$ .

Thus, the contrapositive holds.

**If instead we wanted a direct proof:**

Case 1: Suppose  $a^\top x \leq b$ . Then  $0 \geq a^\top x - b$ , which implies that

$$\delta(a^\top x - b) \geq a^\top x - b$$

Therefore

$$\delta(a^\top x - b) \geq \varepsilon(1 - \delta) + m\delta$$

After rearranging

$$\delta(a^\top x - b - m) \geq \varepsilon(1 - \delta)$$

Since  $a^\top x - b - m \geq 0$  and  $\varepsilon > 0$ , the only feasible choice is  $\delta = 1$ .

Case 2: Suppose  $a^\top x > b$ . Then  $a^\top x - b \geq \varepsilon$ . Since  $a^\top x - b \geq m$ , both choices  $\delta = 0$  and  $\delta = 1$  are feasible.

By the choice of  $\varepsilon$ , we know that  $a^\top x - b > 0$  implies that  $a^\top x - b \geq \varepsilon$ .

---

Since we don't like strict inequalities, we write the strict inequality as  $a^\top x - b \geq \varepsilon$  where  $\varepsilon$  is a small positive number that is a smallest difference between  $a^\top x - b$  and 0 that we would typically observe. As mentioned above, if  $a, b, x$  are all integer, then we can use  $\varepsilon = 1$ .

Now we want an inequality with left hand side  $a^\top x - b \geq$  and right hand side to take the value

- $\varepsilon$  if  $\delta = 0$ ,
- $m$  if  $\delta = 1$ .

This is accomplished with right hand side  $\varepsilon(1 - \delta) + m\delta$ .

Many other combinations of if then statements are summarized in the following table: These two implications can be used to derive the following longer list of implications.

Lastly, if you insist on having exact correspondance, that is, " $\delta = 0$  if and only if  $a^\top x \leq b$ " you can simply include both constraints for "if  $\delta = 0$ , then  $a^\top x \leq b$ " and if " $a^\top x \leq b$ , then  $\delta = 0$ ". Although many problems may be phrased in a way that suggests you need "if and only if", it is often not necessary to use both constraints due to the objectives in the problem that naturally prevent one of these from happening.

For example, if we want to add a binary variable  $\delta$  that means

$$\begin{cases} \delta = 0 \text{ implies } a^\top x \leq b \\ \delta = 1 \text{ Otherwise} \end{cases}$$

Implication	Constraint
If $\delta = 0$ , then $a^\top x \leq b$	$a^\top x \leq b + M\delta$
If $\delta = 0$ , then $a^\top x \geq b$	$a^\top x \geq b + m\delta$
If $\delta = 1$ , then $a^\top x \leq b$	$a^\top x \leq b + M(1 - \delta)$
If $\delta = 1$ , then $a^\top x \geq b$	$a^\top x \geq b + m(1 - \delta)$
If $a^\top x \leq b$ , then $\delta = 1$	$a^\top x \geq b + m\delta + \varepsilon(1 - \delta)$
If $a^\top x \geq b$ , then $\delta = 1$	$a^\top x \leq b + M\delta - \varepsilon(1 - \delta)$
If $a^\top x \leq b$ , then $\delta = 0$	$a^\top x \geq b + m(1 - \delta) + \varepsilon\delta$
If $a^\top x \geq b$ , then $\delta = 0$	$a^\top x \geq b + m(1 - \delta) - \varepsilon\delta$

**Table 7.2: Long list: If/then models with a constraint and a binary variable. Here  $M$  and  $m$  are upper and lower bounds on  $a^\top x - b$  and  $\varepsilon$  is a small number such that if  $a^\top x > b$ , then  $a^\top x \geq b + \varepsilon$ .**

If  $\delta = 1$  does not effect the rest of the optimization problem, then adding the constraint regarding  $\delta = 1$  is not necessary. Hence, typically, in this scenario, we only need to add the constraint  $a^\top x \leq b + M\delta$ .

### 7.11.3. Binary reformulation of integer variables

---

If an integer variable has small upper and lower bounds, it can sometimes be advantageous to recast it as a sequence of binary variables - for either modeling, the solver, or both. Although there are technically many ways to do this, here are the two most common ways.

**Full reformulation:**

*u* many binary variables

For a non-negative integer variable  $x$  with upper bound  $u$ , modeled as

$$0 \leq x \leq u, \quad x \in \mathbb{Z}, \tag{7.9}$$

this can be reformulated with  $u$  binary variables  $z_1, \dots, z_u$  as

$$\begin{aligned} x &= \sum_{i=1}^u iz_i = z_1 + 2z_2 + \dots + uz_u \\ 1 &\geq \sum_{i=1}^u z_i = z_1 + z_2 + \dots + z_u \\ z_i &\in \{0, 1\} \quad \text{for } i = 1, \dots, u \end{aligned} \tag{7.10}$$

We call this the *full reformulation* because there is a binary variable  $z_i$  associated with every value  $i$  that  $x$  could take. That is, if  $z_3 = 1$ , then the second constraint forces  $z_i = 0$  for all  $i \neq 3$  (that is,  $z_3$  is the

only non-zero binary variable), and hence by the first constraint,  $x = 3$ .

### Binary reformulation:

#### $O(\log u)$ many binary variables

For a non-negative integer variable  $x$  with upper bound  $u$ , modeled as

$$0 \leq x \leq u, \quad x \in \mathbb{Z}, \quad (7.11)$$

this can be reformulated with  $u$  binary variables  $z_1, \dots, z_{\log(\lfloor u \rfloor) + 1}$  as

$$\begin{aligned} x &= \sum_{i=0}^{\log(\lfloor u \rfloor) + 1} 2^i z_i = z_0 + 2z_1 + 4z_2 + 8z_3 + \dots + 2^{\log(\lfloor u \rfloor) + 1} z_{\log(\lfloor u \rfloor) + 1} \\ z_i &\in \{0, 1\} \quad \text{for } i = 1, \dots, \log(\lfloor u \rfloor) + 1 \end{aligned} \quad (7.12)$$

We call this the *log reformulation* because this requires only logarithmically many binary variables in terms of the upper bound  $u$ . This reformulation is particularly better than the full reformulation when the upper bound  $u$  is a “larger” number, although we will leave it ambiguous as to how larger a number need to be in order to be described as a “larger” number.

### 7.11.4. SOS1 Constraints

#### Definition 7.3: A

pecial Ordered Sets of type 1 (SOS1) constraint on a vector indicates that at most one element of the vector can non-zero.

We next give an example of how to use binary variables to model this and then show how much simpler it can be coded using the SOS1 constraint.

#### Example: SOS1 Constraints

Code

Solve the following optimization problem:

$$\begin{aligned} \text{maximize} \quad & 3x_1 + 4x_2 + x_3 + 5x_4 \\ \text{subject to} \quad & 0 \leq x_i \leq 5 \\ & \text{at most one of the } x_i \text{ can be nonzero} \end{aligned}$$

### 7.11.5. SOS2 Constraints

#### Definition 7.4: Special Ordered Sets of Type 2 (SOS2)

A Special Ordered Set of Type 2 (SOS2) constraint on a vector indicates that at most two elements of the vector can non-zero AND the non-zero elements must appear consecutively.

We next give an example of how to use binary variables to model this and then show how much simpler it can be coded using the SOS2 constraint.

#### Example: SOS2

Code

Solve the following optimization problem:

$$\begin{aligned} \text{maximize} \quad & 3x_1 + 4x_2 + x_3 + 5x_4 \\ \text{subject to} \quad & 0 \leq x_i \leq 5 \\ & \text{at most two of the } x_i \text{ can be nonzero} \\ & \text{and the nonzero } x_i \text{ must be consecutive} \end{aligned}$$

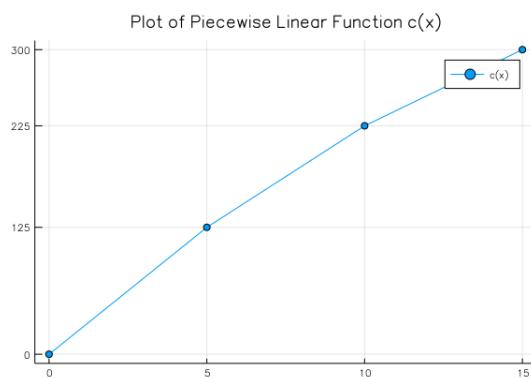
### 7.11.6. Piecewise linear functions with SOS2 constraint

#### Example: Piecewise Linear Function

Code

Consider the piecewise linear function  $c(x)$  given by

$$c(x) = \begin{cases} 25x & \text{if } 0 \leq x \leq 5 \\ 20x + 25 & \text{if } 5 \leq x \leq 10 \\ 15x + 75 & \text{if } 10 \leq x \leq 15 \end{cases}$$



© pwl-plot.png<sup>7</sup>

We will use integer programming to describe this function. We will fix  $x = a$  and then the integer program will set the value  $y$  to  $c(a)$ .

$$\begin{aligned}
 & \min \quad 0 \\
 \text{Subject to} \quad & x - 5z_2 - 10z_3 - 15z_4 = 0 \\
 & y - 125z_2 - 225z_3 - 300z_4 = 0 \\
 & z_1 + z_2 + z_3 + z_4 = 1 \\
 & SOS2 : \{z_1, z_2, z_3, z_4\} \\
 & 0 \leq z_i \leq 1 \quad \forall i \in \{1, 2, 3, 4\} \\
 & x = a
 \end{aligned}$$

### Example: Piecewise Linear Function Application

Code

Consider the following optimization problem where the objective function includes the term  $c(x)$ , where  $c(x)$  is the piecewise linear function described in Example 8:

$$\max z = 12x_{11} + 12x_{21} + 14x_{12} + 14x_{22} - c(x) \quad (7.13)$$

$$\text{s.t. } x_{11} + x_{12} \leq x + 5 \quad (7.14)$$

$$x_{21} + x_{22} \leq 10 \quad (7.15)$$

$$0.5x_{11} - 0.5x_{21} \geq 0 \quad (7.16)$$

$$0.4x_{12} - 0.6x_{22} \geq 0 \quad (7.17)$$

$$x_{ij} \geq 0 \quad (7.18)$$

$$0 \leq x \leq 15 \quad (7.19)$$

Given the piecewise linear, we can model the whole problem explicitly as a mixed-integer linear program.

$$\begin{aligned}
 & \max \quad 12X_{1,1} + 12X_{2,1} + 14X_{1,2} + 14X_{2,2} - y \\
 \text{Subject to} \quad & x - 5z_2 - 10z_3 - 15z_4 = 0 \\
 & y - 125z_2 - 225z_3 - 300z_4 = 0 \\
 & z_1 + z_2 + z_3 + z_4 = 1 \\
 & X_{1,1} + X_{1,2} - x \leq 5 \\
 & X_{2,1} + X_{2,2} \leq 10 \\
 & 0.5X_{1,1} - 0.5X_{2,1} \geq 0 \\
 & 0.4X_{1,2} - 0.6X_{2,2} \geq 0 \\
 & SOS2 : \{z_1, z_2, z_3, z_4\} \\
 & X_{i,j} \geq 0 \quad \forall i \in \{1, 2\}, j \in \{1, 2\} \\
 & 0 \leq z_i \leq 1 \quad \forall i \in \{1, 2, 3, 4\} \\
 & 0 \leq x \leq 15 \\
 & y \text{ free}
 \end{aligned} \quad (7.20)$$

### 7.11.7. Maximizing a minimum

---

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

Consider the problem

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

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

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

### 7.11.8. Relaxing (nonlinear) equality constraints

---

There are a number of scenarios where the constraints can be relaxed without sacrificing optimal solutions to your problem. In a similar vein of the maximizing a minimum, if because of the objective we know that certain constraints will be tight at optimal solutions, we can relax the equality to an inequality. For example,

$$\begin{aligned} & \max \quad x_1 + x_2 + \dots + x_n \\ \text{such that } & x_i = y_i^2 + z_i^2 \text{ for } i = 1, \dots, n \end{aligned}$$

## 7.12 Notes from AIMMS modeling book.

---

- AIMMS - Practical guidelines for solving difficult MILPs
- AIMMS - Linear Programming Tricks
- AIMMS - Formulating Optimization Models
- AIMMS - Practical guidelines for solving difficult linear programs

### 7.12.1. Further Topics

---

- Precedence Constraints

## 7.13 Reference guide for modeling and examples

---

[https://download.aimms.com/aimms/download/manuals/AIMMS3\\_OM.pdf](https://download.aimms.com/aimms/download/manuals/AIMMS3_OM.pdf)

## 7.14 MIP Solvers and Modeling Tools

---

- AMPL
- GAMS
- AIMMS
- Python-MIP
- Pyomo
- PuLP
- JuMP
- GUROBI
- CPLEX (IBM)
- Express
- SAS
- Coin-OR (CBC, CLP, IPOPT)
- SCIP



# 8. Algorithms and Complexity

---

Youtube! - P versus NP

When considering a problem class, we want to know roughly how difficult the problem is. When considering an algorithm, we want to know roughly how fast the algorithm will run. These are questions that we can answer with complexity theory.

## 8.1 Big-O Notation

---

### Definition 8.1

*Big-O]* For two functions  $f(n)$  and  $g(n)$ , we say that  $f(n) = O(g(n))$  if there exist positive constants  $c$  and  $n_0$  such that

$$0 \leq f(n) \leq c g(n) \quad \text{for all } n \geq n_0. \quad (8.1)$$

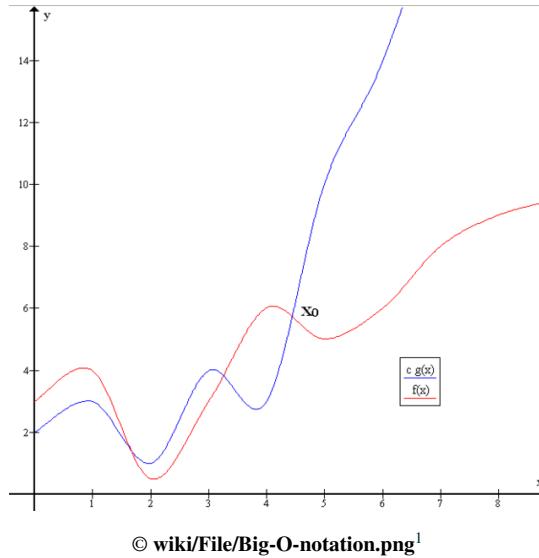
We can also use Big-O to denote a set as

$$O(g(n)) := \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that}$$

$$0 \leq f(n) \leq c g(n), \text{ for all } n \geq n_0\}. \quad (8.2)$$

---

<sup>1</sup>[wiki/File/Big-O-notation.png](#), from [wiki/File/Big-O-notation.png](#). [wiki/File/Big-O-notation.png](#).



**Figure 8.1: Example of Big O notation:**  $f(x) \in O(g(x))$  as there exists  $c > 0$  (e.g.  $c = 1$ ) and  $x_0$  (e.g.  $x_0 = 5$ ) such that  $f(x) \leq cg(x)$  whenever  $x \geq x_0$ .

### Example 8.2:

Consider  $f(n) = 5n^2 + 10n + 7$  and  $g(n) = n^2$ . We want to show that  $f(n) = O(g(n))$ .

Let's try  $c = 22$  and  $n_0 = 1$ . We need to show that Equation 8.1 is satisfied.

Note first that we always have

$$1. n^2 \leq n^2 \text{ and therefore } 5n^2 \leq 5n^2$$

Note that if  $n \geq 1$ , then

$$2. n \leq n^2 \text{ and therefore } 10n \leq 10n^2$$

$$3. 1 \leq n^2 \text{ and therefore } 7 \leq 7n^2$$

Since all inequalities 1,2, and 3 are valid for  $n \geq 1$ , by adding them, we obtain a new inequality that is also valid for  $n \geq 1$ , which is

$$5n^2 + 10n + 7 \leq 5n^2 + 10n^2 + 7n^2 \quad \text{for all } n \geq 1, \quad (8.3)$$

$$\Rightarrow 5n^2 + 10n + 7 \leq 22n^2 \quad \text{for all } n \geq 1. \quad (8.4)$$

Hence, we have shown that Equation 8.1 holds for  $c = 22$  and  $n_0 = 1$ . Hence  $f(n) = O(g(n))$ .

Correct uses:

- $2^n + n^5 + \sin(n) = O(2^n)$
- $2^n = O(n!)$
- $n! + 2^n + 5n = O(n!)$

- $n^2 + n = O(n^3)$

- $n^2 + n = O(n^2)$

- $\log(n) = O(n)$

- $10\log(n) + 5 = O(n)$

Notice that not all examples above give a tight bound on the asymptotic growth. For instance,  $n^2 + n = O(n^3)$  is true, but a tighter bound is  $n^2 + n = O(n^2)$ .

In particular, the goal of big O notation is to give an upper bound on the asymptotic growth of a function. But we would prefer to give a strong upper bound as opposed to a weak upper bound. For instance, if you order a package online, you will typically be given a bound on the latest date that it will arrive. For example, if it will arrive within a week, you might be guaranteed that it will arrive by next Tuesday. This sounds like a reasonable bound. But if instead, they tell you it will arrive before 1 year from today, this may not be as useful information. In the case of big O notation, we would like to give a least upper bound that most simply describes the growth behavior of the function.

In that example,  $n^2 + n = O(n^2)$ , this literally means that there is some number  $c$  and some value  $n_0$  that  $n^2 + n \leq cn^2$  for all  $n \geq n_0$ , that is, for all values of  $n_0$  larger than  $n$ , the function  $cn^2$  dominates  $n^2 + n$ .

For example, a valid choice is  $c = 2$  and  $n_0 = 1$ . Then it is true that  $n^2 + n \leq 2n^2$  for all  $n \geq 1$ .

But it is also true that  $n^2 + n = O(n^3)$ . For example, a valid choice is again  $c = 2$  and  $n_0 = 1$ , then  $n^2 + n \leq 2n^3$  for all  $n \geq 1$ .

In this example,  $O(n^3)$  is the case where the internet tells you the package will arrive before 1 year from today. The bound is true, but it is not as useful information as we would like to have. Let's compare these upper bounds. Let  $f(n) = n^2 + n$ ,  $g(n) = 2n^2$ ,  $h(n) = 2n^3$ .

Then we have

	$n = 10$ .	$n = 100$ .	$n = 1000$ .	$n = .$ 10000
$f(n)$	110,	10100,	1001000,	100010000
$g(n)$	200,	20000,	2000000,	200000000
$h(n)$ .	2000,	2000000,	2000000000,	20000000000000

So, here we see that  $g(n)$  and  $h(n)$  are both upper bounds on  $f(n)$ , but the nice part about  $g(n)$  is that is growing at a similar rate to  $f(n)$ . In particular, it is always within a factor of 2 of  $f(n)$ .

Alternatively, the bound  $h(n)$  is true, but it grows so much faster than  $f(n)$  that is doesn't give a good idea of the asymptotic growth of  $f(n)$ .

Some common classes of functions:

$O(1)$	Constant
$O(\log(n))$	Logarithmic
$O(n)$	Linear
$O(n^c)$ (for $c > 1$ )	Polynomial
$O(c^n)$ (for $c > 1$ )	Exponential

**Exponential Time Algorithms do not currently solve reasonable-sized problems in reasonable time.**

<b>Polynomial</b>	$\log n$	3	4
	$n$	10	20
	$n \log n$	33	86
	$n^2$	100	400
	$n^3$	1,000	8,000
	$n^5$	100,000	3,200,000
	$n^{10}$	10,000,000,000	10,240,000,000,000
<b>Exponential</b>	$n$	10	20
	$n^{\log n}$	2,099	419,718
	$2^n$	1,024	1,048,576
	$5^n$	9,765,625	95,367,431,640,625
	$n!$	3,628,800	2,432,902,008,176,640,000
	$n^n$	10,000,000,000	104,857,600,000,000,000,000,000,000,000

© time-of-algorithms<sup>2</sup>

**Figure 8.2: time-of-algorithms**

## 8.2 Algorithms - Example with Bubble Sort

The following definition comes from Merriam-Webster's dictionary.

### Definition 8.3: A

algorithm is a procedure for solving a mathematical problem in a finite number of steps that frequently involves repetition of an operation; broadly: a step-by-step procedure for solving a problem or accomplishing some end.

<sup>2</sup>time-of-algorithms, from time-of-algorithms. time-of-algorithms, time-of-algorithms.

## 8.2.1. Sorting

---

Wikipedia

Bubble sort algorithm:.....

### Bubble sort algorithm

- Compare two neighboring objects and swap if they are in the wrong order.

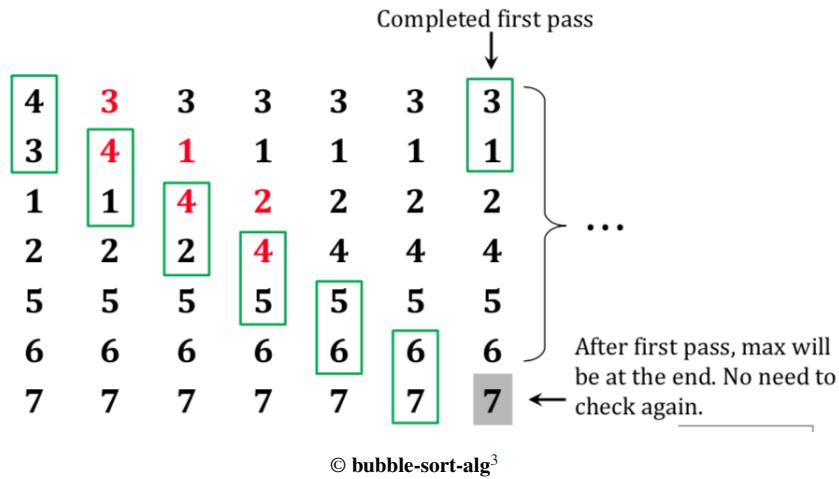


Figure 8.3: bubble-sort-alg

**The algorithm will terminate when a pass is completed with no swaps.**

**How many steps did it take to sort the numbers in this example?**

**15 steps**

**This is around  $2n$  ( $n = 7$ ). Does this mean it is  $O(n)$ ?**

**NO!**

**The worst case must be examined.**

© bubble-sort-complexity-1<sup>4</sup>

Figure 8.4: bubble-sort-complexity-1

<sup>3</sup>bubble-sort-alg, from bubble-sort-alg. bubble-sort-alg, bubble-sort-alg.

<sup>4</sup>bubble-sort-complexity-1, from bubble-sort-complexity-1. bubble-sort-complexity-1, bubble-sort-complexity-1.

<sup>5</sup>bubble-sort-complexity-2, from bubble-sort-complexity-2. bubble-sort-complexity-2, bubble-sort-complexity-2.

**In the worst case, the minimum element will be at the end of the list.**

**The number of steps in this case will be:**

$$(n - 1) + (n - 2) + \dots + 2 + 1 = \frac{n(n - 1)}{2} = O(n^2)$$

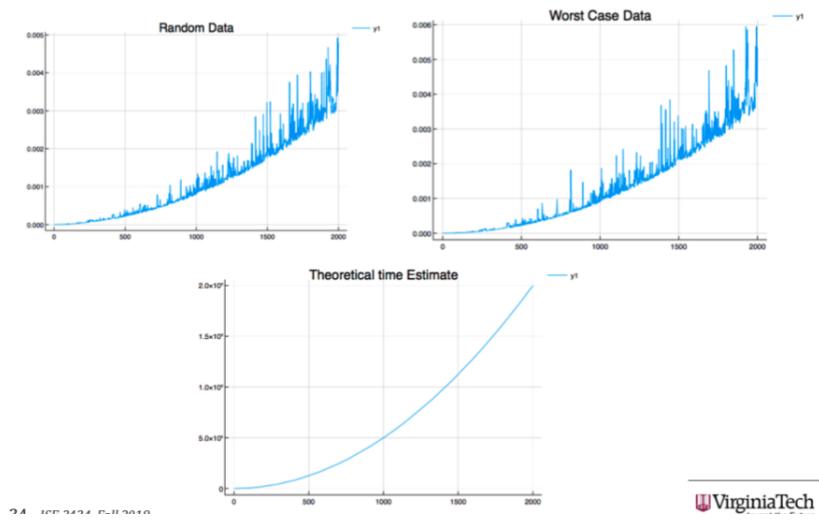
The Bubble Sort is an  $O(n^2)$  algorithm

© bubble-sort-complexity-2<sup>5</sup>

**Figure 8.5: bubble-sort-complexity-2**

These can be verified experimentally as seen in the following plot. The random case grows quadratically just as the worst case does.

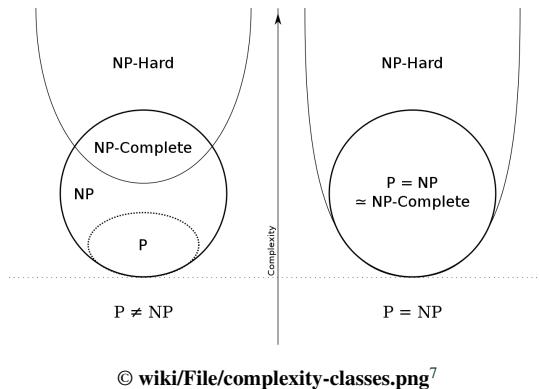
### Time elapsed in computer for bubble sort



© bubble-sort-computational-example<sup>6</sup>

**Figure 8.6: bubble-sort-computational-example**

<sup>6</sup>bubble-sort-computational-example, from bubble-sort-computational-example. bubble-sort-computational-example, bubble-sort-computational-example.



**Figure 8.7: Complexity class possibilities. Most academics agree that the case  $P \neq NP$  is more likely.**

## 8.3 Complexity Classes

---

In this subsection we will discuss the complexity classes P, NP, NP-Complete, and NP-Hard. These classes help measure how difficult a problem is. **Informally**, these classes can be thought of as

- P - the class of efficiently solvable problems
- NP - the class of efficiently checkable problems
- NP-Hard - the class of problems that can solve any problem in NP
- NP-Complete - the class of problems that are in both NP and are NP-Hard.

It is not known if P is the same as NP, but it is conjectured that these are very different classes. This would mean that the NP-Hard problems (and NP-Complete problems) are necessarily much more difficult than the problems in P. See Figure 8.7 .

We will now discuss these classes more formally.

### 8.3.1. P

---

P is the class of polynomially solvable problems. P contains tall problems for which there exists an algorithm that solves the problem in a run time bounded by a polynomial. That is,  $O(n^c)$  for some constant  $c$ .

#### Example 8.4: Complexity Minimum Spanning Tree

*The minimum size spanning tree problem is in P. It can be solved, for instance, by Prim's algorithm, which runs in time  $O(m \log n)$ , where m is the number of edges in the graph and n is the number of nodes in the graph.*

<sup>7</sup>wiki/File/complexity-classes.png, from wiki/File/complexity-classes.png, wiki/File/complexity-classes.png, wiki/File/complexity-classes.png, wiki/File/complexity-classes.png.

**Example 8.5: Complexity Linear Programming**

Linear programming is in P. It can be solved by interior point methods in  $O(n^{3.5}\phi)$  where  $\phi$  represents the number of binary bits that are required to encode the problem. These bits describe the matrix  $A$ , and vectors  $c$  and  $b$  that define the linear program.

**8.3.2. NP**

NP is the class of nondeterministic polynomial problems. NP contains all problems in which membership can be verified in polynomial time from a certificate.

Thus, to show that a problem is in NP, you must do the following:

1. Describe a format for a certificate to the problem.
2. Show that given such a certificate, it is easy to verify the solution to the problem.

**Example 8.6: I**

teger Linear Programming is in NP. More explicitly, the feasibility question of

"Does there exist an integer vector  $x$  that satisfies  $Ax \leq b$ "

is in NP.

Although it turns out to be difficult to find such an  $x$  or even prove that one exists, this problem is in NP for the following reason: if you are given a particular  $x$  and someone claims to you that it is a feasible solution to the problem, then you can easily check if they are correct. In this case, the vector  $x$  that you were given is called a certificate.

Note that it is easy to verify if  $x$  is a solution to the problem because you just have to

1. Check if  $x$  is integer.
2. Use matrix multiplication to check if  $Ax \leq b$  holds.

**8.3.3. NP-Hard**

The class of problems that are called *NP-Hard* are those that can be used to solve any other problem in the NP class. That is, problem A is NP-Hard provided that for any problem B in NP there is a transformation of problem B that preserves the size of the problem, up to a polynomial factor, into a new problem that problem A can be used to solve.

Here we think of "if problem A could be solved efficiently, then all problems in NP could be solved efficiently".

More specifically, we assume that we have an oracle for problem A that runs in polynomial time. An oracle is an algorithm that for the problem that returns the solution of the problem in a time polynomial in the input. This oracle can be thought of as a magic computer that gives us the answer to the problem. Thus, we say that problem A is NP-Complete provided that given an oracle for problem A, one can solve any other problem B in NP in polynomial time.

Note: These problems are not necessarily in NP.

### 8.3.4. NP-Complete

---

The class of problems that are call *NP-Complete* are those which are in NP and also NP-Hard.

We know of many problems that are NP-Complete. For example, binary integer programming feasibility is NP-Complete. One can show that another problem is NP-complete by

1. showing that it can be used to solve binary integer programming feasibility,
2. showing that the problem is in NP.

The first problem proven to be NP-Complete is called *3-SAT* []. 3-SAT is a special case of the *satisfiability problem*. In a satisfiability problem, we have variables  $X_1, \dots, X_n$  and we want to assign them values as either `true` or `false`. The problem is described with *AND* operations, denoted as  $\wedge$ , with *OR* operations, denoted as  $\vee$ , and with *NOT* operations, denoted as  $\neg$ . The *AND* operation  $X_1 \wedge X_2$  returns `true` if BOTH  $X_1$  and  $X_2$  are true. The *OR* operation  $X_1 \vee X_2$  returns `true` if AT LEAST ONE OF  $X_1$  and  $X_2$  are true. Lastly, the *NOT* operation  $\neg X_1$  returns there opposite of the value of  $X_1$ .

These can be described in the following table

$$\text{true} \wedge \text{true} = \text{true} \tag{8.1}$$

$$\text{true} \wedge \text{false} = \text{false} \tag{8.2}$$

$$\text{false} \wedge \text{false} = \text{false} \tag{8.3}$$

$$\text{false} \wedge \text{true} = \text{false} \tag{8.4}$$

$$\text{true} \vee \text{true} = \text{true} \tag{8.5}$$

$$\text{true} \vee \text{false} = \text{true} \tag{8.6}$$

$$\text{false} \vee \text{false} = \text{false} \tag{8.7}$$

$$\text{false} \vee \text{true} = \text{true} \tag{8.8}$$

$$\neg \text{true} = \text{false} \tag{8.9}$$

$$\neg \text{false} = \text{true} \tag{8.10}$$

For example, **Missing code here** A *logical expression* is a sequence of logical operations on variables  $X_1, \dots, X_n$ , such that

$$(X_1 \wedge \neg X_2 \vee X_3) \wedge (X_1 \vee \neg X_3) \vee (X_1 \wedge X_2 \wedge X_3). \quad (8.11)$$

A *clause* is a logical expression that only contains the operations  $\vee$  and  $\neg$  and is not nested (with parentheses), such as

$$X_1 \vee \neg X_2 \vee X_3 \vee \neg X_4. \quad (8.12)$$

A fundamental result about logical expressions is that they can always be reduced to a sequence of clauses that are joined by  $\wedge$  operations, such as

$$(X_1 \vee \neg X_2 \vee X_3 \vee \neg X_4) \wedge (X_1 \vee X_2 \vee X_3) \wedge (X_2 \vee \neg X_3 \vee \neg X_4 \vee X_5). \quad (8.13)$$

The satisfiability problem takes as input a logical expression in this format and asks if there is an assignment of `true` or `false` to each variable  $X_i$  that makes the expression `true`. The 3-SAT problem is a special case where the clauses have only three variables in them.

### 3-SAT:

#### *NP-Complete*

Given a logical expression in  $n$  variables where each clause has only 3 variables, decide if there is an assignment to the variables that makes the expression `true`.

### Binary Integer Programming:

#### *NP-Complete*

Binary Integer Programming can easily be shown to be in NP, since verifying solutions to BIP can be done by checking a linear system of inequalities.

Furthermore, it can be shown to be NP-Complete since it can be used to solve 3-SAT. That is, given an oracle for BIP, since 3-SAT can be modeled as a BIP, the 3-SAT could be solved in oracle-polynomial time.

## 8.4 Relevant Terminology

---

We will discuss the following concepts:

- Feasible solutions
- Optimal solutions

- Approximate solutions
- Heuristics
- Exact Algorithms
- Approximation Algorithms
- Complexity class relations

## 8.5 Matching Problem

---

### Definition 8.7: G

iven a graph  $G = (V, E)$ , a matching is a subset  $E' \subseteq E$  such that no vertex  $v \in V$  is contained in more than one edge in  $E'$ .

A perfect matching is a matching where every vertex is connected to an edge in  $E'$ .

A maximal matching is a matching  $E'$  such that there is no matching  $E''$  that strictly contains it.

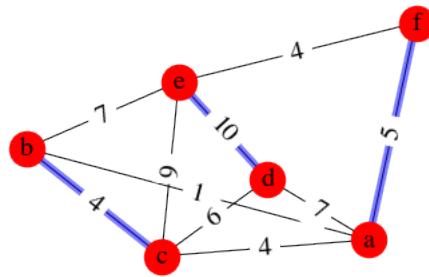
INCLUDE PICTURES OF MATCHINGS

**Figure 8.8:** Two possible matchings. On the left, we have a perfect matching (all nodes are matched). On the right, a feasible matching, but not a perfect matching since not all nodes are matched.

### Definition 8.8: Maximum Weight Matching

iven a graph  $G = (V, E)$ , with associated weights  $w_e \geq 0$  for all  $e \in E$ , a maximum weight matching is a matching that maximizes the sum of the weights in the matching.

<sup>8</sup>graph-for-matching-maximal, from graph-for-matching-maximal. graph-for-matching-maximal, graph-for-matching-maximal.

© graph-for-matching-maximal<sup>8</sup>**Figure 8.9: graph-for-matching-maximal**

### 8.5.1. Greedy Algorithm for Maximal Matching

---

The greedy algorithm iteratively adds the edge with largest weight that is feasible to add.

**Greedy Algorithm for Maximal Matching:**

Complexity:  $O(|E| \log(|V|))$

1. Begin with an empty graph ( $M = \emptyset$ )
2. Label the edges in the graph such that  $w(e_1) \geq w(e_2) \geq \dots \geq w(e_m)$
3. For  $i = 1, \dots, m$   
If  $M \cup \{e_i\}$  is a valid matching (i.e., no vertex is incident with two edges), then set  $M \leftarrow M \cup \{e_i\}$   
(i.e., add edge  $e_i$  to the graph  $M$ )
4. Return  $M$

**Theorem 8.9**

*[[Avis83]] The greedy algorithm finds a 2-approximation of the maximum weighted matching problem. That is,  $w(M_{greedy}) \geq \frac{1}{2}w(M^*)$ .*

### 8.5.2. Other algorithms to look at

1. Improved Algorithm [DRAKE2003211]

2. Blossom Algorithm Wikipedia

## 8.6 Minimum Spanning Tree

### Definition 8.10: G

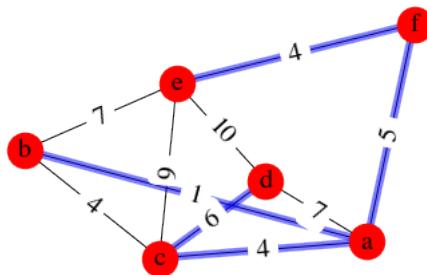
Given a graph  $G = (V, E)$ , a spanning tree is a connected, acyclic subgraph  $T$  that contains every node in  $V$ .

© spanning-tree<sup>9</sup>

**Figure 8.10:** spanning-tree

### Definition 8.11: G

Given a graph  $G = (V, E)$ , with associated weights  $w_e \geq 0$  for all  $e \in E$ , a maximum weight spanning tree is a spanning tree that maximizes the sum of the edge weights.



© spanning-tree-MST<sup>10</sup>

**Figure 8.11:** spanning-tree-MST

<sup>9</sup>spanning-tree, from spanning-tree. spanning-tree, spanning-tree.

<sup>10</sup>spanning-tree-MST, from spanning-tree-MST. spanning-tree-MST, spanning-tree-MST.

**Lemma 8.12: L**

$t$   $G$  be a connected graph with  $n$  vertices.

1.  $T$  is a spanning tree of  $G$  if and only if  $T$  has  $n - 1$  edges and is connected.
2. Any subgraph  $S$  of  $G$  with more than  $n - 1$  edges contains a cycle.

See Section 9.2 for integer programming formulations of this problem.

### 8.6.1. Kruskal's algorithm

---

[Wikipedia](#)

**Kruskal - for Minimum Spanning tree:**

Complexity:  $O(|E| \log(|V|))$

1. Begin with an empty tree ( $T = \emptyset$ )
2. Label the edges in the graph such that  $w(e_1) \leq w(e_2) \leq \dots \leq w(e_m)$
3. For  $i = 1, \dots, m$   
If  $T \cup \{e_i\}$  is acyclic, then set  $T \leftarrow T \cup \{e_i\}$
4. Return  $T$

### 8.6.2. Prim's Algorithm

---

[Wikipedia](#)

[TeXample - Figure for min spannig tree](#)

## 8.7 Traveling Salesman Problem

---

See Section 9.3 for integer programming formulations of this problem. Also, hill climbing algorithms for this problem such as 2-Opt, simulated annealing, and tabu search will be discussed in Section ??.

### 8.7.1. Nearest Neighbor - Construction Heuristic

---

We will discuss heuristics more later in this book. For now, present this construction heuristic as a simple algorithmic example.

[https://en.wikipedia.org/wiki/Nearest\\_neighbour\\_algorithm](https://en.wikipedia.org/wiki/Nearest_neighbour_algorithm) Starting from any node, add the edge to the next closest node. Continue this process.

**Nearest Neighbor:**

Complexity:  $O(n^2)$

1. Start from any node (lets call this node 1) and label this as your current node.
2. Pick the next current node as the one that is closest to the current node that has not yet been visited.
3. Repeat step 2 until all nodes are in the tour.

### 8.7.2. Double Spanning Tree - 2-Apx

---

We can use a minimum spanning tree algorithm to find a provably okay solution to the TSP, provided certain properties of the graph are satisfied.

Graphs with nice properties are often easier to handle and typically graphs found in the real world have some nice properties. The *triangle inequality* comes from the idea of a triangle that the sum of the lengths of two sides always are longer than the length of the third side. See Figure 8.12

**Definition 8.13: Triangle Inequality on a Graph**

A complete, weighted graph  $G$  (i.e., a graph that has all possible edges and a weight assigned to each edge) satisfies the triangle inequality provided that for ever triple of vertices  $a, b, c$  and edges  $e_{ab}, e_{bc}, e_{ac}$ , we have that

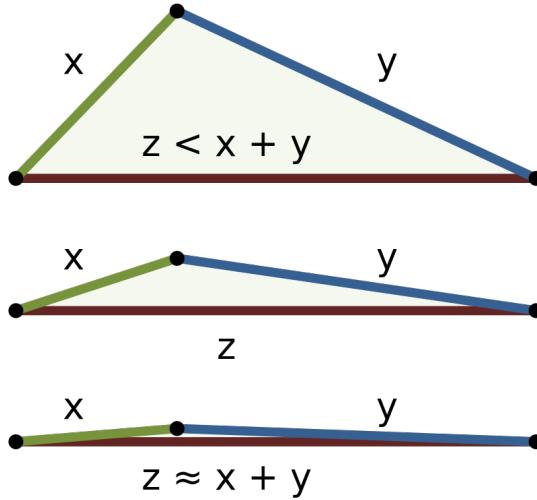
$$w(e_{ab}) + w(e_{bc}) \geq w(e_{ac}).$$

Let  $S$  be the resulting tour and let  $S^*$  be an optimal tour. Since the resulting tour is feasible, it will satisfy

$$w(S^*) \leq w(S).$$

---

<sup>11</sup> [wiki/File:triangle\\_inequality.png](https://en.wikipedia.org/wiki/File:triangle_inequality.png), from [https://en.wikipedia.org/wiki/File:triangle\\_inequality.png](https://en.wikipedia.org/wiki/File:triangle_inequality.png).



© wiki/File/triangle\_inequality.png<sup>11</sup>

**Figure 8.12:** wiki/File/triangle\_inequality.png

### Algorithm 1 Double Spanning Tree

**Require:** A graph  $G = (V, E)$  that satisfies the triangle inequality

**Ensure:** A tour that is a 2-Apx of the optimal solution

- 1: Compute a minimum spanning tree  $T$  of  $G$ .
- 2: Double each edge in the minimum spanning tree (i.e., if edge  $e_{ab}$  is in  $T$ , add edge  $e_{ba}$ .
- 3: Compute an Eulerian Tour using these edges.
- 4: Return tour that visits vertices in the order the Eulerian Tour visits them, but without repeating any vertices.

But we also know that the weight of a minimum spanning tree  $T$  is less than that of the optimal tour, hence

$$w(T) \leq w(S^*).$$

Lastly, due to the triangle inequality we know that

$$w(S) \leq 2w(T),$$

since replacing any edge in the Eulerian tour with a more direct edge only reduces the total weight.

Putting this together, we have

$$w(S) \leq 2w(T) \leq 2w(S^*)$$

and hence,  $S$  is a 2-approximation of the optimal solution.

### 8.7.3. Christofides - Approximation Algorithm - (3/2)-Apx

---

If we combine algorithms for minimum spanning tree and matching, we can find a better approximation algorithm. This is given by Christofides. Again, this is in the case where the graph satisfies the triangle inequality. See Wikipedia - Christofides Algorithm or Ola Svensson Lecture Slides for more detail.



# 9. Exponential Size Formulations

---

Although typically models need to be a reasonable size in order for us to code them and send them to a solver, there are some ways that we can allow having models of exponential size. The first example here is the cutting stock problem, where we will model with exponentially many variables. The second example is the traveling salesman problem, where we will model with exponentially many constraints. We will also look at some other models for the traveling salesman problem.

## 9.1 Cutting Stock

---

This is a classic problem that works excellent for a technique called *column generation*. We will discuss two versions of the model and then show how we can use column generation to solve the second version more efficiently. First, let's describe the problem.

### Cutting Stock:

You run a company that sells of pipes of different lengths. These lengths are  $L_1, \dots, L_k$ . To produce these pipes, you have one machine that produces pipes of length  $L$ , and then cuts them into a collection of shorter pipes as needed.

Image to add

You have an order come in for  $d_i$  pipes of length  $i$  for  $i = 1, \dots, k$ . How can you fill the order while cutting up the fewest number of pipes?

### Example 9.1: Cutting stock with pipes

A plumber stocks standard lengths of pipe, all of length 19 m. An order arrives for:

- 12 lengths of 4m
- 15 lengths of 5m
- 22 lengths of 6m

How should these lengths be cut from standard stock pipes so as to minimize the number of standard pipes used?

An initial model for this problem could be constructed as follows:

- Let  $N$  be an upper bound on the number of pipes that we may need.
- Let  $z_j = 1$  if we use pipe  $i$  and  $z_j = 0$  if we do not use pipe  $j$ , for  $j = 1, \dots, N$ .
- Let  $x_{ij}$  be the number of cuts of length  $L_i$  in pipe  $j$  that we use.

Then we have the following model

$$\begin{aligned}
 & \min \sum_{j=1}^N z_j \\
 \text{s.t. } & \sum_{i=1}^k L_i x_{ij} \leq L z_j \text{ for } j = 1, \dots, N \\
 & \sum_{j=1}^N x_{ij} \geq d_i \text{ for } i = 1, \dots, k \\
 & z_j \in \{0, 1\} \text{ for } j = 1, \dots, N \\
 & x_{ij} \in \mathbb{Z}_+ \text{ for } i = 1, \dots, k, j = 1, \dots, N
 \end{aligned} \tag{9.1}$$

### Exercise 9.2: Show Bound

In the example above, show that we can choose  $N = 16$ .

For our example above, using  $N = 16$ , we have

$$\begin{aligned}
 & \min \sum_{j=1}^{16} z_j \\
 \text{s.t. } & 4x_{1j} + 5x_{2j} + 6x_{3j} \leq 19z_j \\
 & \sum_{j=1}^{16} x_{1j} \geq 12 \\
 & \sum_{j=1}^{16} x_{2j} \geq 15 \\
 & \sum_{j=1}^{16} x_{3j} \geq 22 \\
 & z_j \in \{0, 1\} \text{ for } j = 1, \dots, 16 \\
 & x_{ij} \in \mathbb{Z}_+ \text{ for } i = 1, \dots, 3, j = 1, \dots, 16
 \end{aligned} \tag{9.2}$$

Additionally, we could break the symmetry in the problem. That is, suppose the solution uses 10 of the 16 pipes. The current formulation does not restrict which 10 pipes are used. Thus, there are many possible

solutions. To reduce this complexity, we can state that we only use the first 10 pipes. We can write a constraint that says *if we don't use pipe j, then we also will not use any subsequent pipes*. Hence, by not using pipe 11, we enforce that pipes 11, 12, 13, 14, 15, 16 are not used. This can be done by adding the constraints

$$z_1 \geq z_2 \geq z_3 \geq \cdots \geq z_N. \quad (9.3)$$

See ?? for code for this formulation.

Unfortunately, this formulation is slow and does not scale well with demand. In particular, the number of variables is  $N + kN$  and the number of constraints is  $N$  (plus integrality and non-negativity constraints on the variables). The solution times for this model are summarized in the following table:

### INPUT TABLE OF COMPUTATIONS

#### 9.1.1. Pattern formulation

---

We could instead list all patterns that are possible to cut each pipe. A pattern is a vector  $a \in \mathbb{Z}_+^k$  such that for each  $i$ ,  $a_i$  lengths of  $L_i$  can be cut from a pipe of length  $L$ . That is

$$\begin{aligned} \sum_{i=1}^k L_i a_i &\leq L \\ a_i &\in \mathbb{Z}_+ \text{ for all } i = 1, \dots, k \end{aligned} \quad (9.4)$$

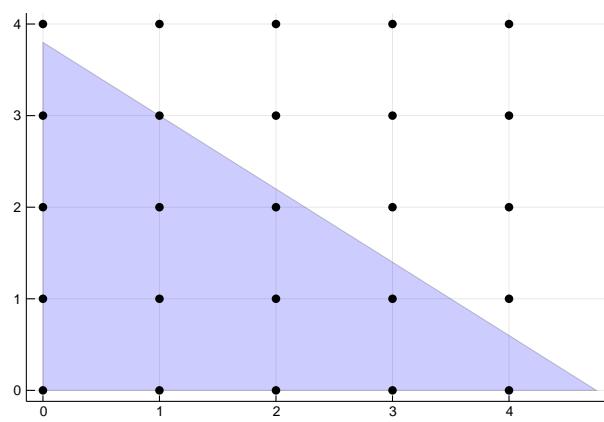
In our running example, we have

$$\begin{aligned} 4a_1 + 5a_2 + 6a_3 &\leq 19 \\ a_i &\in \mathbb{Z}_+ \text{ for all } i = 1, \dots, 3 \end{aligned} \quad (9.5)$$

For visualization purposes, consider the patterns where  $a_3 = 0$ . That is, only patterns with cuts of length 4m or 5m. All patterns of this type are represented by an integer point in the polytope

$$P = \{(a_1, a_2) : 4a_1 + 5a_2 \leq 19, a_1 \geq 0, a_2 \geq 0\} \quad (9.6)$$

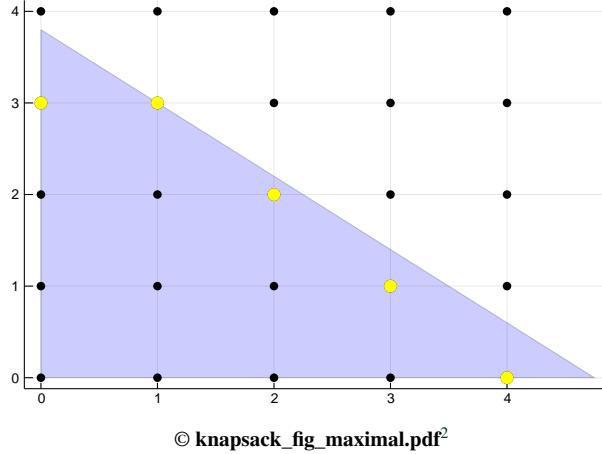
which we can see here:



© knapsack\_fig.pdf<sup>1</sup>

<sup>1</sup>knapsack\_fig.pdf, from knapsack\_fig.pdf. knapsack\_fig.pdf, knapsack\_fig.pdf.

where  $P$  is the blue triangle and each integer point represents a pattern. Feasible patterns lie inside the polytope  $P$ . Note that we only need patterns that are maximal with respect to number of each type we cut. Pictorially, we only need the patterns that are integer points represented as yellow dots in the picture below.



For example, the pattern  $[3, 0, 0]$  is not needed (only cut 3 of length 4m) since we could also use the pattern  $[4, 0, 0]$  (cut 4 of length 4m) or we could even use the pattern  $[3, 1, 0]$  (cut 3 of length 4m and 1 of length 5m).

### 9.1.2. Column Generation

---

Consider the (4.8), but in this case we are instead minimizing. Thus we can write it as

$$\begin{aligned} \min \quad & (c_N - c_B B^{-1} N)x_N + c_B B^{-1} b \\ \text{s.t.} \quad & x_B + B^{-1} N)x_N = B^{-1} b \\ & x \geq 0 \end{aligned} \tag{9.7}$$

In our LP we have  $c = 1$ , that is,  $c_i = 1$  for all  $i = 1, \dots, k$ . Hence, we can write it as

$$\begin{aligned} \min \quad & (1_N - 1_B B^{-1} N)x_N + 1_B B^{-1} b \\ \text{s.t.} \quad & x_B + B^{-1} N)x_N = B^{-1} b \\ & x \geq 0 \end{aligned} \tag{9.8}$$

Now, if there exists a non-basic variable that could enter the basis and improve the objective, then there is one with a reduced cost that is negative. For a particular non-basic variable, the coefficient on it is

$$(1 - 1_B B^{-1} N^i)_i \tag{9.9}$$

where  $N^i$  is the  $i$ -th column of the matrix  $N$ . Thus, we want to look for a column  $a$  of  $N$  such that

$$1 - 1_B B^{-1} a < 0 \Rightarrow 1 < 1_B B^{-1} a \tag{9.10}$$

---

<sup>2</sup>knapsack\_fig\_maximal.pdf,  
knapsack\_fig\_maximal.pdf.

from knapsack\_fig\_maximal.pdf.

knapsack\_fig\_maximal.pdf,

**Pricing Problem:**

(knapsack problem!)

Given a current basis  $B$  of the *master* linear program, there exists a new column to add to the basis that improves the LP objective if and only if the following problem has an objective value strictly larger than 1.

$$\begin{aligned} \max \quad & 1_B B^{-1} a \\ \text{s.t.} \quad & \sum_{i=1}^k L_i a_i \leq L \\ & a_i \in \mathbb{Z}_+ \text{ for } i = 1, \dots, k \end{aligned} \tag{9.11}$$

**Example 9.3: Pricing Problem**

*After choosing the initial columns ....  
we can find the objective function....*

$$\begin{aligned} \max \quad & \dots \dots a \\ \text{s.t.} \quad & 4a_1 + 5a_2 + 6a_3 \leq 19 \\ & a_i \in \mathbb{Z}_+ \text{ for } i = 1, \dots, k \end{aligned} \tag{9.12}$$

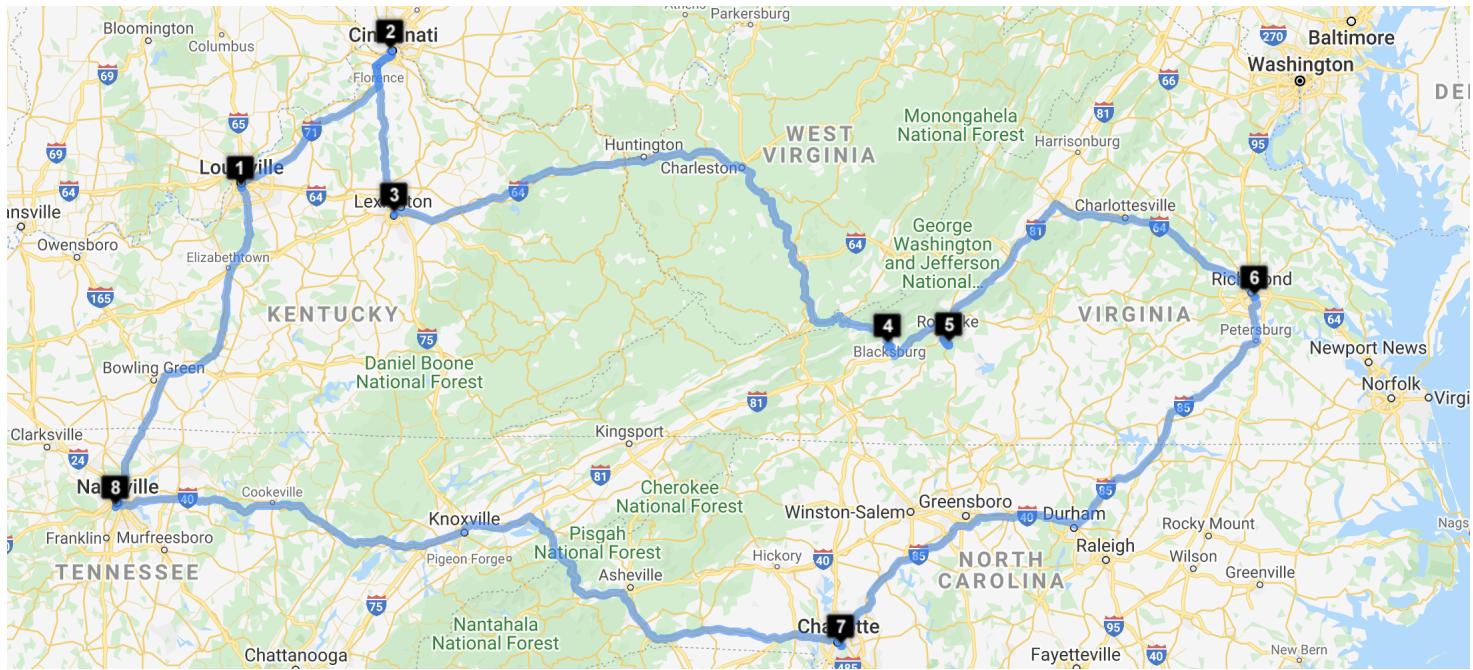
**9.1.3. Cutting Stock - Multiple widths****Resources**

Gurobi has an excellent demonstration application to look at: [Gurobi - Cutting Stock Demo Gurobi - Multiple Master Rolls](#)

Here are some solutions:

- <https://github.com/fzsun/cutstock-gurobi>.
- <http://www.dcc.fc.up.pt/~jpp/mpa/cutstock.py>

Here is an AIMMS description of the problem: [AIMMS Cutting Stock](#)



**Figure 9.1:** Optimal tour through 8 cities. Generated by Gebweb - Optimap. See it also on Google Maps!.

## 9.2 Spanning Trees

### Resources

See [Abdelmaguid2018] for a list of 11 models for the minimum spanning tree and a comparison using CPLEX.

## 9.3 Traveling Salesman Problem

### Resources

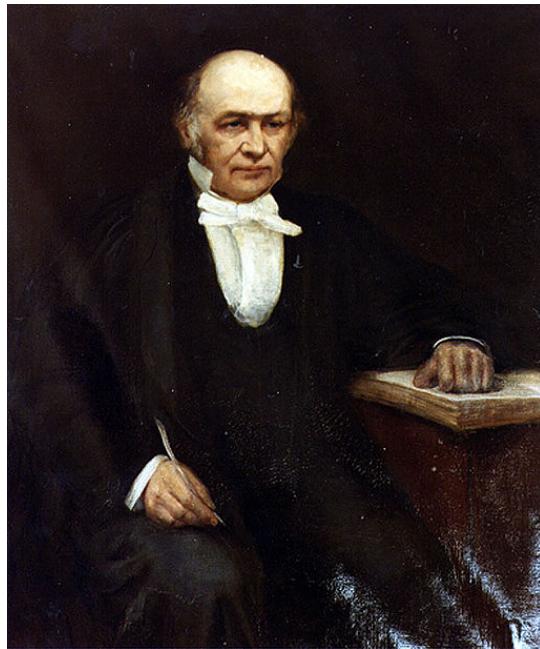
See [math.watloo.ca](http://math.watloo.ca) for excellent material on the TSP.

See also this chapter *A Practical Guide to Discrete Optimization*.

Also, watch this excellent talk by Bill Cook "Postcards from the Edge of Possibility": [Youtube!](https://www.youtube.com/watch?v=JyfXzqjwvIY)

Google Maps!

We consider a directed graph, graph  $G = (N, A)$  of nodes  $N$  and arcs  $A$ . Arcs are directed edges. Hence the arc  $(i, j)$  is the directed path  $i \rightarrow j$ .



© wiki/File/William\_Rowan\_Hamilton\_painting.jpg<sup>3</sup>

**Figure 9.2: wiki/File/William\_Rowan\_Hamilton\_painting.jpg**

A *tour*, or Hamiltonian cycle (see Figure 9.2 ), is a cycle that visits all the nodes in  $N$  exactly once and returns back to the starting node.

Given costs  $c_{ij}$  for each arc  $(i, j) \in A$ , the goal is to find a minimum cost tour.

**Traveling Salesman Problem:**

**NP-Hard**

Given a directed graph  $G = (N, A)$  and costs  $c_{ij}$  for all  $(i, j) \in A$ , find a tour of minimum cost.

**ADD TSP FIGURE**

In the figure, the nodes  $N$  are the cities and the arcs  $A$  are the directed paths city $i \rightarrow$  city $j$ .

---

<sup>3</sup>wiki/File/William\_Rowan\_Hamilton\_painting.jpg, from wiki/File/William\_Rowan\_Hamilton\_painting.jpg, wiki/File/William\_Rowan\_Hamilton\_painting.jpg, wiki/File/William\_Rowan\_Hamilton\_painting.jpg.

**MODELS** When constructing an integer programming model for TSP, we define variables  $x_{ij}$  for all  $(i, j) \in A$  as

$$x_{ij} = 1 \text{ if the arc } (i, j) \text{ is used and } x_{ij} = 0 \text{ otherwise.}$$

We want the model to satisfy the fact that each node should have exactly one incoming arc and one leaving arc. Furthermore, we want to prevent self loops. Thus, we need the constraints:

$$\sum_{j \in N} x_{ij} = 1 \quad \text{for all } i \in N \quad [\text{outgoing arc}] \quad (9.1)$$

$$\sum_{i \in N} x_{ij} = 1 \quad \text{for all } j \in N \quad [\text{incoming arc}] \quad (9.2)$$

$$x_{ii} = 0 \quad \text{for all } i \in N \quad [\text{no self loops}] \quad (9.3)$$

Unfortunately, these constraints are not enough to completely describe the problem. The issue is that *subtours* may arise. For instance

### ADD SUBTOURS FIGURE

#### 9.3.1. MTZ Model

---

The Miller-Tucker-Zemlin (MTZ) model for the TSP uses variables to mark the order for which cities are visited. This model introduce general integer variables to do so, but in the process, creates a formulation that has few inequalities to describe.

Some feature of this model:

- This model adds variables  $u_i \in \mathbb{Z}$  with  $1 \leq u_i \leq n$  that decide the order in which nodes are visited.
- We set  $u_1 = 1$  to set a starting place.
- Crucially, this model relies on the following fact

*Let  $x$  be a solution to (9.1)-(9.3) with  $x_{ij} \in \{0, 1\}$ . If there exists a subtour in this solution that contains the node 1, then there also exists a subtour that does not contain the node 1.*

The following model adds constraints

$$\text{If } x_{ij} = 1, \text{ then } u_i + 1 \leq u_j. \quad (9.4)$$

This if-then statement can be modeled with a big-M, choosing  $M = n$  is a sufficient upper bound. Thus, it can be written as

$$u_i + 1 \leq u_j + n(1 - x_{ij}) \quad (9.5)$$

Setting these constraints to be active enforces the order  $u_i < u_j$ .

Consider a subtour now  $2 \rightarrow 5 \rightarrow 3 \rightarrow 2$ . Thus,  $x_{25} = x_{53} = x_{32} = 1$ . Then using the constraints from (9.5), we have that

$$u_2 < u_5 < u_3 < u_2, \quad (9.6)$$

but this is infeasible since we cannot have  $u_2 < u_2$ .

As stated above, if there is a subtour containing the node 1, then there is also a subtour not containing the node 1. Thus, we can enforce these constraints to only prevent subtours that don't contain the node 1. Thus, the full tour that contains the node 1 will still be feasible.

This is summarized in the following model:

### Traveling Salesman Problem - MTZ Model:

$$\min \sum_{i,j \in N} c_{ij} x_{ij} \quad (9.7)$$

$$\sum_{j \in N} x_{ij} = 1 \quad \text{for all } i \in N \quad [\text{outgoing arc}] \quad (9.8)$$

$$\sum_{i \in N} x_{ij} = 1 \quad \text{for all } j \in N \quad [\text{incoming arc}] \quad (9.9)$$

$$x_{ii} = 0 \quad \text{for all } i \in N \quad [\text{no self loops}] \quad (9.10)$$

$$u_i + 1 \leq u_j + n(1 - x_{ij}) \quad \text{for all } i, j \in N, i, j \neq 1 \quad [\text{prevents subtours}] \quad (9.11)$$

$$u_1 = 1 \quad (9.12)$$

$$2 \leq u_i \leq n \quad \text{for all } i \in N, i \neq 1 \quad (9.13)$$

$$u_i \in \mathbb{Z} \quad \text{for all } i \in N \quad (9.14)$$

$$x_{ij} \in \{0, 1\} \quad \text{for all } i, j \in N \quad (9.15)$$

### Example 9.4: TSP with 4 nodes

*Distance Matrix:*

A \ B	1	2	3	4
1	0	1	2	3
2	1	0	1	2
3	2	1	0	4
4	3	2	4	0

**Solution.** Here is the full MTZ model:

$$\begin{aligned} \min \quad & x_{1,2} + 2x_{1,3} + 3x_{1,4} + x_{2,1} + x_{2,3} + 2x_{2,4} + \\ & 2x_{3,1} + x_{3,2} + 4x_{3,4} + 3x_{4,1} + 2x_{4,2} + 4x_{4,3} \end{aligned}$$

Subject to

$$\begin{array}{ll} x_{1,1} + x_{1,2} + x_{1,3} + x_{1,4} = 1 & \text{outgoing from node 1} \\ x_{2,1} + x_{2,2} + x_{2,3} + x_{2,4} = 1 & \text{outgoing from node 2} \\ x_{3,1} + x_{3,2} + x_{3,3} + x_{3,4} = 1 & \text{outgoing from node 3} \\ x_{4,1} + x_{4,2} + x_{4,3} + x_{4,4} = 1 & \text{outgoing from node 4} \end{array}$$

$$\begin{array}{ll} x_{1,1} + x_{2,1} + x_{3,1} + x_{4,1} = 1 & \text{incoming to node 1} \\ x_{1,2} + x_{2,2} + x_{3,2} + x_{4,2} = 1 & \text{incoming to node 2} \\ x_{1,3} + x_{2,3} + x_{3,3} + x_{4,3} = 1 & \text{incoming to node 3} \\ x_{1,4} + x_{2,4} + x_{3,4} + x_{4,4} = 1 & \text{incoming to node 4} \end{array}$$

$$\begin{array}{ll} x_{1,1} = 0 & \text{No self loop with node 1} \\ x_{2,2} = 0 & \text{No self loop with node 2} \\ x_{3,3} = 0 & \text{No self loop with node 3} \\ x_{4,4} = 0 & \text{No self loop with node 4} \end{array}$$

$$\begin{array}{ll} u_1 = 1 & \text{Start at node 1} \\ 2 \leq u_i \leq 4, \quad \forall i \in \{2, 3, 4\} & \\ u_2 + 1 \leq u_3 + 4(1 - x_{2,3}) & \\ u_2 + 1 \leq u_4 + 4(1 - x_{2,4}) \leq 3 & \\ u_3 + 1 \leq u_2 + 4(1 - x_{3,2}) \leq 3 & \\ u_3 + 1 \leq u_4 + 4(1 - x_{3,4}) \leq 3 & \\ u_4 + 1 \leq u_2 + 4(1 - x_{4,2}) \leq 3 & \\ u_4 + 1 \leq u_3 + 4(1 - x_{4,3}) \leq 3 & \\ x_{i,j} \in \{0, 1\} \quad \forall i \in \{1, 2, 3, 4\}, j \in \{1, 2, 3, 4\} & \\ u_i \in \mathbb{Z}, \quad \forall i \in \{1, 2, 3, 4\} & \end{array}$$



**Example 9.5: TSP with 5 nodes**

$$\min \quad x_{1,2} + 2x_{1,3} + 3x_{1,4} + 4x_{1,5} + x_{2,1} + x_{2,3} + 2x_{2,4} + 2x_{2,5} + 2x_{3,1} + \\ x_{3,2} + 4x_{3,4} + x_{3,5} + 3x_{4,1} + 2x_{4,2} + 4x_{4,3} + 2x_{4,5} + \\ 4x_{5,1} + 2x_{5,2} + x_{5,3} + 2x_{5,4}$$

*Subject to*

$$\begin{aligned} x_{1,1} + x_{1,2} + x_{1,3} + x_{1,4} + x_{1,5} &= 1 \\ x_{2,1} + x_{2,2} + x_{2,3} + x_{2,4} + x_{2,5} &= 1 \\ x_{3,1} + x_{3,2} + x_{3,3} + x_{3,4} + x_{3,5} &= 1 \\ x_{4,1} + x_{4,2} + x_{4,3} + x_{4,4} + x_{4,5} &= 1 \\ x_{5,1} + x_{5,2} + x_{5,3} + x_{5,4} + x_{5,5} &= 1 \end{aligned}$$

$$\begin{aligned} x_{1,1} + x_{2,1} + x_{3,1} + x_{4,1} + x_{5,1} &= 1 \\ x_{1,2} + x_{2,2} + x_{3,2} + x_{4,2} + x_{5,2} &= 1 \\ x_{1,3} + x_{2,3} + x_{3,3} + x_{4,3} + x_{5,3} &= 1 \\ x_{1,4} + x_{2,4} + x_{3,4} + x_{4,4} + x_{5,4} &= 1 \\ x_{1,5} + x_{2,5} + x_{3,5} + x_{4,5} + x_{5,5} &= 1 \end{aligned}$$

$$\begin{aligned} x_{1,1} &= 0 \\ x_{2,2} &= 0 \\ x_{3,3} &= 0 \\ x_{4,4} &= 0 \\ x_{5,5} &= 0 \end{aligned}$$

$$\begin{aligned} u_1 &= 1 \\ 2 \leq u_i \leq 5 &\quad \forall i \in \{1, 2, 3, 4, 5\} \\ u_2 + 1 &\leq u_3 + 5(1 - x_{2,3}) \\ u_2 + 1 &\leq u_4 + 5(1 - x_{2,4}) \\ u_2 + 1 &\leq u_5 + 5(1 - x_{2,5}) \\ u_3 + 1 &\leq u_2 + 5(1 - x_{3,2}) \\ u_3 + 1 &\leq u_4 + 5(1 - x_{3,4}) \\ u_4 + 1 &\leq u_2 + 5(1 - x_{4,2}) \\ u_4 + 1 &\leq u_3 + 5(1 - x_{4,3}) \\ u_3 + 1 &\leq u_5 + 5(1 - x_{3,5}) \\ u_4 + 1 &\leq u_5 + 5(1 - x_{4,5}) \\ u_5 + 1 &\leq u_2 + 5(1 - x_{5,2}) \\ u_5 + 1 &\leq u_3 + 5(1 - x_{5,3}) \\ u_5 + 1 &\leq u_4 + 5(1 - x_{5,4}) \\ x_{i,j} &\in \{0, 1\} \quad \forall i \in \{1, 2, 3, 4, 5\}, j \in \{1, 2, 3, 4, 5\} \\ u_i &\in \mathbb{Z}, \quad \forall i \in \{1, 2, 3, 4, 5\} \end{aligned}$$

## PROS OF THIS MODEL

- Small description
- Easy to implement

## CONS OF THIS MODEL

- Linear relaxation is not very tight. Thus, the solver may be slow when given this model.

**9.3.2. Dantzig-Fulkerson-Johnson Model**

This model does not add new variables. Instead, it adds constraints that conflict with the subtours. For instance, consider a subtour

$$2 \rightarrow 5 \rightarrow 3 \rightarrow 2. \quad (9.16)$$

We can prevent this subtour by adding the constraint

$$x_{25} + x_{53} + x_{32} \leq 2 \quad (9.17)$$

meaning that at most 2 of those arcs are allowed to happen at the same time. In general, for any subtour  $S$ , we can have the *subtour elimination constraint*

$$\sum_{(i,j) \in S} x_{ij} \leq |S| - 1 \quad \text{Subtour Elimination Constraint.} \quad (9.18)$$

In the previous example with  $S = \{(2,5), (5,3), (3,2)\}$  we have  $|S| = 3$ , where  $|S|$  denotes the size of the set  $S$ .

This model suggests that we just add all of these subtour elimination constraints.

**Traveling Salesman Problem - DFJ Model:**

$$\min \sum_{i,j \in N} c_{ij} x_{ij} \quad (9.19)$$

$$\sum_{j \in N} x_{ij} = 1 \quad \text{for all } i \in N \quad [\text{outgoing arc}] \quad (9.20)$$

$$\sum_{i \in N} x_{ij} = 1 \quad \text{for all } j \in N \quad [\text{incoming arc}] \quad (9.21)$$

$$x_{ii} = 0 \quad \text{for all } i \in N \quad [\text{no self loops}] \quad (9.22)$$

$$\sum_{(i,j) \in S} x_{ij} \leq |S| - 1 \quad \text{for all subtours } S \quad [\text{prevents subtours}] \quad (9.23)$$

$$x_{ij} \in \{0, 1\} \quad \text{for all } i, j \in N \quad (9.24)$$

**Example 9.6: DFJ Model for  $n = 4$  nodes***Distance Matrix:*

A \ B	1	2	3	4
1	0	1	2	3
2	1	0	1	2
3	2	1	0	4
4	3	2	4	0

$$\begin{aligned} \text{1 min } & x_{1,2} + 2x_{1,3} + 3x_{1,4} + x_{2,1} + x_{2,3} + 2x_{2,4} \\ & + 2x_{3,1} + x_{3,2} + 4x_{3,4} + 3x_{4,1} + 2x_{4,2} + 4x_{4,3} \end{aligned}$$

*Subject to*

$$\begin{array}{ll} x_{1,1} + x_{1,2} + x_{1,3} + x_{1,4} = 1 & \text{outgoing from node 1} \\ x_{2,1} + x_{2,2} + x_{2,3} + x_{2,4} = 1 & \text{outgoing from node 2} \\ x_{3,1} + x_{3,2} + x_{3,3} + x_{3,4} = 1 & \text{outgoing from node 3} \\ x_{4,1} + x_{4,2} + x_{4,3} + x_{4,4} = 1 & \text{outgoing from node 4} \end{array}$$

$$\begin{array}{ll} x_{1,1} + x_{2,1} + x_{3,1} + x_{4,1} = 1 & \text{incoming to node 1} \\ x_{1,2} + x_{2,2} + x_{3,2} + x_{4,2} = 1 & \text{incoming to node 2} \\ x_{1,3} + x_{2,3} + x_{3,3} + x_{4,3} = 1 & \text{incoming to node 3} \\ x_{1,4} + x_{2,4} + x_{3,4} + x_{4,4} = 1 & \text{incoming to node 4} \end{array}$$

$$\begin{array}{ll} x_{1,1} = 0 & \text{No self loop with node 1} \\ x_{2,2} = 0 & \text{No self loop with node 2} \\ x_{3,3} = 0 & \text{No self loop with node 3} \\ x_{4,4} = 0 & \text{No self loop with node 4} \end{array}$$

$$\begin{array}{ll} x_{1,2} + x_{2,1} \leq 1 & S = [(1,2), (2,1)] \\ x_{1,3} + x_{3,1} \leq 1 & S = [(1,3), (3,1)] \\ x_{1,4} + x_{4,1} \leq 1 & S = [(1,4), (4,1)] \\ x_{2,3} + x_{3,2} \leq 1 & S = [(2,3), (3,2)] \\ x_{2,4} + x_{4,2} \leq 1 & S = [(2,4), (4,2)] \\ x_{3,4} + x_{4,3} \leq 1 & S = [(3,4), (4,3)] \\ x_{2,1} + x_{1,3} + x_{3,2} \leq 2 & S = [(2,1), (1,3), (3,2)] \\ x_{1,2} + x_{2,3} + x_{3,1} \leq 2 & S = [(1,2), (2,3), (3,1)] \\ x_{3,1} + x_{1,4} + x_{4,3} \leq 2 & S = [(3,1), (1,4), (4,3)] \\ x_{1,3} + x_{3,4} + x_{4,1} \leq 2 & S = [(1,3), (3,4), (4,1)] \\ x_{2,1} + x_{1,4} + x_{4,2} \leq 2 & S = [(2,1), (1,4), (4,2)] \end{array}$$

**Example 9.7**

Consider a graph on 5 nodes.

Here are all the subtours of length at least 3 and also including the full length tours.

Hence, there are many subtours to consider.

**PROS OF THIS MODEL**

- Very tight linear relaxation

**CONS OF THIS MODEL**

- Exponentially many subtours  $S$  possible, hence this model is too large to write down.

**SOLUTION:** ADD SUBTOUR ELIMINATION CONSTRAINTS AS NEEDED. WE WILL DISCUSS THIS IN A FUTURE SECTION ON *cutting planes* .

### 9.3.3. Traveling Salesman Problem - Branching Solution

---

We will see in the next section

1. That the constraint (9.1)-(9.3) always produce integer solutions as solutions to the linear relaxation.
2. A way to use branch and bound (the topic of the next section) in order to avoid subtours.

## 9.4 Vehicle Routing Problem (VRP)

---

The VRP is a generalization of the TSP and comes in many many forms. The major difference is now we may consider multiple vehicles visiting the around cities. Obvious examples are creating bus schedules and mail delivery routes.

Variations of this problem include

- Time windows (for when a city needs to be visited)
- Prize collecting (possibly not all cities need to be visited, but you gain a prize for visiting each city)
- Multi-depot vehicle routing problem (fueling or drop off stations)
- Vehicle rescheduling problem (When delays have been encountered, how do you adjust the routes)
- Inhomogeneous vehicles (vehicles have different abilities (speed, distance, capacity, etc.).

To read about the many variants, see: Vehicle Routing: Problems, Methods, and Applications, Second Edition. Editor(s): Paolo Toth and Daniele Vigo. MOS-SIAM Series on Optimization.

For one example of a VRP model, see GUROBI Modeling Examples - technician routing scheduling.

### 9.4.1. Case Study: Bus Routing in Boston

---

Review this case study after studying algorithms and heuristics for integer programming.

<https://www.informs.org/Impact/O.R.-Analytics-Success-Stories/Optimized-school-bus-routing>

<https://pubsonline.informs.org/doi/abs/10.1287/inte.2019.1015>

<https://www.informs.org/Resource-Center/Video-Library/Edelman-Competition-Videos/2019-Edelman-Competition-Videos/2019-Edelman-Finalist-Boston-Public-Schools>

<https://www.youtube.com/watch?v=LFeeaNPrbY>

## 9.5 Literature and other notes

---

- Gilmore-Gomory Cutting Stock [[Gilmore-Gomory](#)]
- A Column Generation Algorithm for Vehicle Scheduling and Routing Problems
- The Integrated Last-Mile Transportation Problem
- [http://www.optimization-online.org/DB\\_FILE/2017/11/6331.pdf](http://www.optimization-online.org/DB_FILE/2017/11/6331.pdf) A BRANCH-AND-PRICE ALGORITHM FOR CAPACITATED HYPERGRAPH VERTEX SEPARATION

### 9.5.1. Google maps data

---

Blog - Python | Calculate distance and duration between two places using google distance matrix API

### 9.5.2. TSP In Excel

---

TSP with excel solver



# 10. Algorithms to Solve Integer Programs

## 10.1 LP to solve IP

Recall that the linear relaxation of an integer program is the linear programming problem after removing the integrality constraints

Integer Program:

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

Linear Relaxation:

$$\begin{aligned} \max \quad & z_{LP} = c^\top x \\ \text{s.t.} \quad & Ax \leq b \\ & x \in \mathbb{R}^n \end{aligned}$$

### Theorem 10.1: LP Bounds

It always holds that

$$z_{IP}^* \leq z_{LP}^*. \quad (10.1)$$

Furthermore, if  $x_{LP}^*$  is integral (feasible for the integer program), then

$$x_{LP}^* = x_{IP}^* \quad \text{and} \quad z_{LP}^* = z_{IP}^*. \quad (10.2)$$

### Example 10.2

Consider the problem

$$\begin{aligned} \max z = & 3x_1 + 2x_2 \\ \text{s.t.} \quad & 2x_1 + x_2 \leq 6 \\ & x_1, x_2 \geq 0; x_1, x_2 \text{ integer} \end{aligned}$$

### 10.1.1. Rounding LP Solution can be bad!

---

#### Resources

*Video! - Michel Belaire (EPFL) looking at rounding the LP solution to an IP solution*

Consider the two variable knapsack problem

$$\max 3x_1 + 100x_2 \quad (10.3)$$

$$x_1 + 100x_2 \leq 100 \quad (10.4)$$

$$x_i \in \{0, 1\} \text{ for } i = 1, 2. \quad (10.5)$$

Then  $x_{LP}^* = (1, 0.99)$  and  $z_{LP}^* = 1 \cdot 3 + 0.99 \cdot 100 = 3 + 99 = 102$ .

But  $x_{IP}^* = (0, 1)$  with  $z_{IP}^* = 0 \cdot 3 + 1 \cdot 100 = 100$ .

Suppose that we rounded the LP solution.

$x_{LP-Rounded-Down}^* = (1, 0)$ . Then  $z_{LP-Rounded-Down}^* = 1 \cdot 3 = 3$ . Which is a terrible solution!

How can we avoid this issue?

Cool trick! Using two different strategies gives you at least a 1/2 approximation to the optimal solution.

### 10.1.2. Rounding LP solution can be infeasible!

---

Now only could it produce a poor solution, it is not always clear how to round to a feasible solution.

### 10.1.3. Fractional Knapsack

---

The fractional knapsack problem has an exact greedy algorithm.

#### Resources

- *Youtube!*
- *Blog*

## 10.2 Branch and Bound

---

### Resources

[Video! - Michel Belaire \(EPFL\) Teaching Branch and Bound Theory Video!](#) - Michel Belaire (EPFL) Teaching Branch and Bound with Example

See Module by Miguel Casquilho for some nice notes on branch and bound.

### 10.2.1. Algorithm

---

#### Algorithm 2 Branch and Bound - Maximization

---

**Require:** Integer Linear Problem with max objective

**Ensure:** Exact Optimal Solution  $x^*$

- 1: Set  $LB = -\infty$ .
  - 2: Solve LP relaxation.
    - a: If  $x^*$  is integer, stop!
    - b: Otherwise, choose fractional entry  $x_i^*$  and branch onto subproblems:
      - (i)  $x_i \leq \lfloor x_i^* \rfloor$
      - and (ii)  $x_i \geq \lceil x_i^* \rceil$ .
  - 3: Solve LP relaxation of any subproblem.
    - a: If LP relaxation is infeasible, prune this node as "Infeasible"
    - b: If  $z^* < LB$ , prune this node as "Suboptimal"
    - c:  $x^*$  is integer, prune this nodes as "Integer" and update  $LB = \max(LB, z^*)$ .
    - d: Otherwise, choose fractional entry  $x_i^*$  and branch onto subproblems:
      - (i)  $x_i \leq \lfloor x_i^* \rfloor$
      - and (ii)  $x_i \geq \lceil x_i^* \rceil$ . Return to step 2 until all subproblems are pruned.
  - 4: Return best integer solution found.
- 

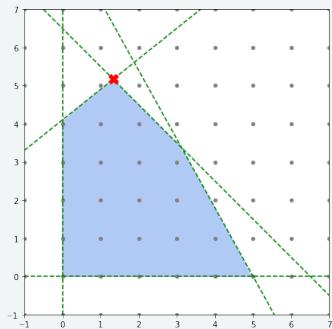
Here is an example of branching on general integer variables.

**Example 10.3**

Consider the two variable example with

$$\begin{aligned} \max & -3x_1 + 4x_2 \\ 2x_1 + 2x_2 &\leq 13 \\ -8x_1 + 10x_2 &\leq 41 \\ 9x_1 + 5x_2 &\leq 45 \\ 0 \leq x_1 &\leq 10, \text{ integer} \\ 0 \leq x_2 &\leq 10, \text{ integer} \end{aligned}$$

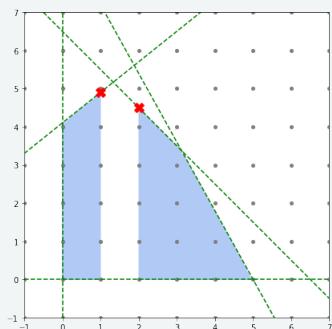
$$x = [1.33, 5.167] \text{ obj} = 16.664$$



© branch-and-bound1<sup>1</sup>

$$x = [1, 4.9] \text{ obj} = 16.5998$$

$$x = [2, 4.5] \text{ obj} = 12.0$$

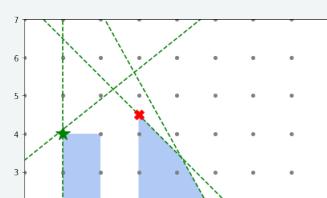


© branch-and-bound2<sup>2</sup>

*Infeasible Region*

$$x = [0.4] \text{ obj} = 16.0$$

$$x = [2.4.5] \text{ obj} = 12.0$$



### 10.2.2. Knapsack Problem and 0/1 branching

---

Consider the problem

$$\begin{aligned} \max \quad & 16x_1 + 22x_2 + 12x_3 + 8x_4 \\ \text{s.t. } & 5x_1 + 7x_2 + 4x_3 + 3x_4 \leq 14 \\ & 0 \leq x_i \leq 1 \quad i = 1, 2, 3, 4 \\ & x_i \in \{0, 1\} \quad i = 1, 2, 3, 4 \end{aligned}$$

**Question:** What is the optimal solution if we remove the binary constraints?

$$\begin{aligned} \max \quad & c_1x_1 + c_2x_2 + c_3x_3 + c_4x_4 \\ \text{s.t. } & a_1x_1 + a_2x_2 + a_3x_3 + a_4x_4 \leq b \\ & 0 \leq x_i \leq 1 \quad i = 1, 2, 3, 4 \end{aligned}$$

**Question:** How do I find the solution to this problem?

$$\begin{aligned} \max \quad & c_1x_1 + c_2x_2 + c_3x_3 + c_4x_4 \\ \text{s.t. } & (a_1 - A)x_1 + (a_2 - A)x_2 + (a_3 - A)x_3 + (a_4 - A)x_4 \leq 0 \\ & 0 \leq x_i \leq m_i \quad i = 1, 2, 3, 4 \end{aligned}$$

**Question:** How do I find the solution to this problem?

Consider the problem

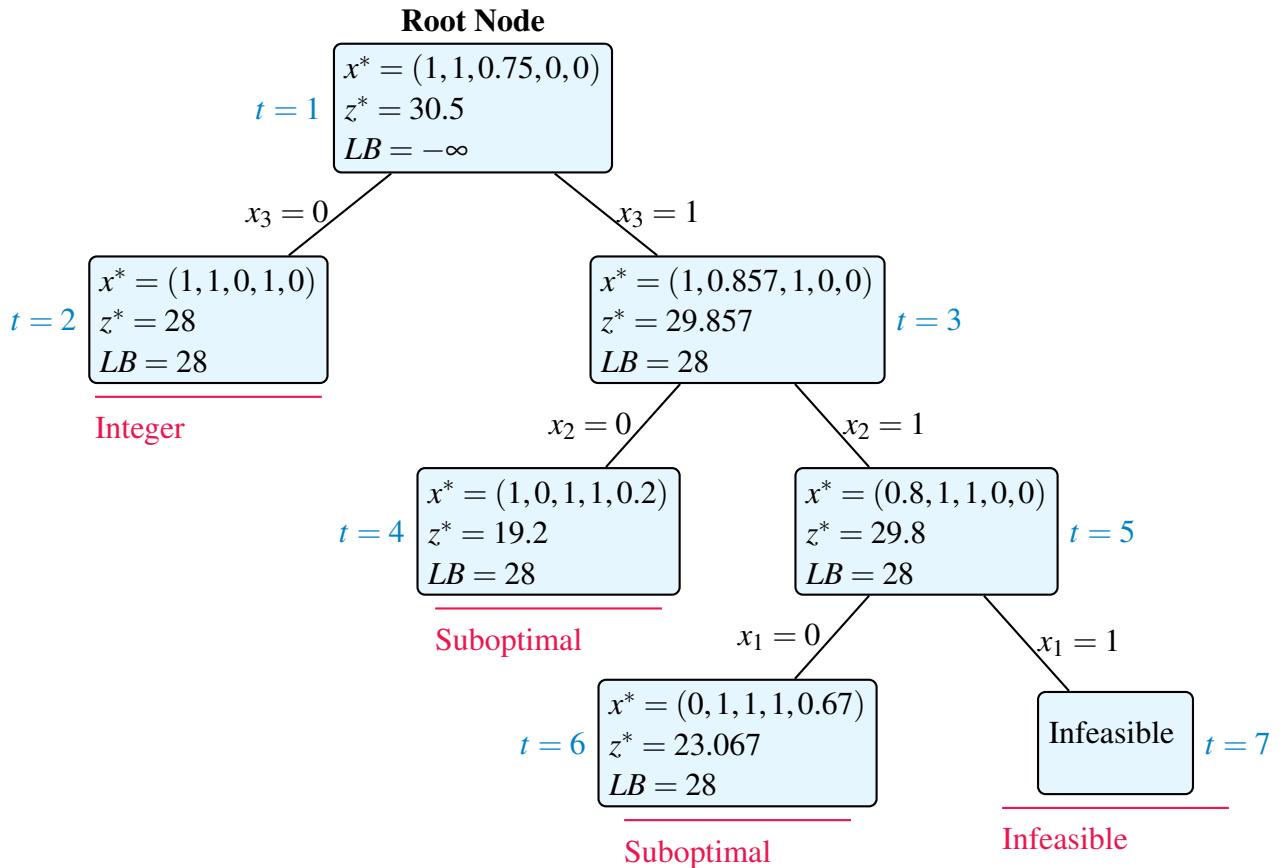
$$\begin{aligned} \max \quad & 16x_1 + 22x_2 + 12x_3 + 8x_4 \\ \text{s.t. } & 5x_1 + 7x_2 + 4x_3 + 3x_4 \leq 14 \\ & 0 \leq x_i \leq 1 \quad i = 1, 2, 3, 4 \\ & x_i \in \{0, 1\} \quad i = 1, 2, 3, 4 \end{aligned}$$

We can solve this problem with branch and bound.

The optimal solution was found at  $t = 5$  at subproblem 6 to be  $x^* = (0, 1, 1, 1)$ ,  $z^* = 42$ .

**Example: Binary Knapsack** Solve the following problem with branch and bound.

$$\begin{aligned} \max \quad & z = 11x_1 + 15x_2 + 6x_3 + 2x_4 + x_5 \\ \text{Subject to: } & 5x_1 + 7x_2 + 4x_3 + 3x_4 + 15x_5 \leq 15 \\ & x_i \text{ binary}, i = 1, \dots, 4 \end{aligned}$$



### 10.2.3. Traveling Salesman Problem solution via Branching

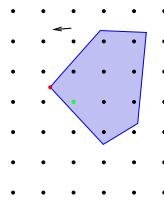
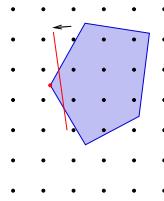
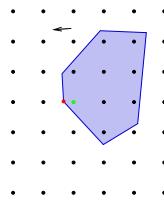
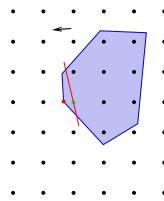
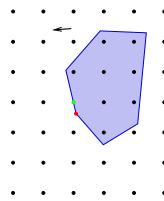
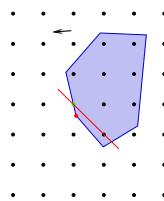
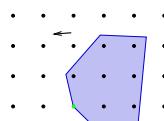
Explain solving TSP via a generalized branching method that removes subtours (instead of adding constraints).

## 10.3 Cutting Planes

Cutting planes are inequalities  $\pi^\top x \leq \pi_0$  that are valid for the feasible integer solutions that the cut off part of the LP relaxation. Cutting planes can create a tighter description of the feasible region that allows for the optimal solution to be obtained by simply solving a strengthened linear relaxation.

The cutting plane procedure, as demonstrated in Figure 10.1. The procedure is as follows:

1. Solve the current LP relaxation.
2. If solution is integral, then return that solution. STOP
3. Add a cutting plane (or many cutting planes) that cut off the LP-optimal solution.
4. Return to Step 1.

© figureCuttingPlane1<sup>4</sup>© figureCuttingPlane2<sup>5</sup>© figureCuttingPlane3<sup>6</sup>© figureCuttingPlane4<sup>7</sup>© figureCuttingPlane5<sup>8</sup>© figureCuttingPlane6<sup>9</sup>

In practice, this procedure is integrated in some with branch and bound and also other primal heuristics.

### 10.3.1. Chvátal Cuts

---

Chvátal Cuts are a general technique to produce new inequalities that are valid for feasible integer points.

**Chvátal Cuts:**

Suppose

$$a_1x_1 + \cdots + a_nx_n \leq d \quad (10.1)$$

is a valid inequality for the polyhedron  $P = \{x \in \mathbb{R}^n : Ax \leq b, x \geq 0\}$ , then

$$\lfloor a_1 \rfloor x_1 + \cdots + \lfloor a_n \rfloor x_n \leq \lfloor d \rfloor \quad (10.2)$$

is valid for the integer points in  $P$ , that is, it is valid for the set  $P \cap \mathbb{Z}^n$ . Equation (10.2) is called a Chvátal Cut.

We will illustrate this idea with an example.

**Example 10.4**

Recall example ?? . The model was

**Model**

$$\begin{array}{ll} \min & p + n + d + q \\ \text{s.t.} & p + 5n + 10d + 25q = 83 \\ & p, d, n, q \in \mathbb{Z}_+ \end{array} \quad \begin{array}{l} \text{total number of coins used} \\ \text{sums to } 83\text{¢} \\ \text{each is a non-negative integer} \end{array}$$

From the equality constraint we can derive several inequalities.

1. Divide by 25 and round down both sides:

$$\frac{p + 5n + 10d + 25q}{25} = 83/25 \Rightarrow q \leq 3$$

2. Divide by 10 and round down both sides:

$$\frac{p + 5n + 10d + 25q}{10} = 83/10 \Rightarrow d + 2q \leq 8$$

3. Divide by 5 and round down both sides:

$$\frac{p + 5n + 10d + 25q}{10} = 83/5 \Rightarrow n + 2d + 5q \leq 16$$

4. Multiply by 0.12 and round down both sides:

$$0.12(p + 5n + 10d + 25q) = 0.12(83) \Rightarrow d + 3q \leq 9$$

These new inequalities are all valid for the integer solutions. Consider the new model:

**New Model**

$$\begin{array}{ll} \min & p + n + d + q \\ \text{s.t.} & p + 5n + 10d + 25q = 83 \\ & q \leq 3 \\ & d + 2q \leq 8 \\ & n + 2d + 5q \leq 16 \\ & d + 3q \leq 9 \\ & p, d, n, q \in \mathbb{Z}_+ \end{array} \quad \begin{array}{l} \text{total number of coins used} \\ \text{sums to } 83\text{¢} \\ \text{each is a non-negative integer} \end{array}$$

The solution to the LP relaxation is exactly  $q = 3, d = 0, n = 1, p = 3$ , which is an integral feasible solution, and hence it is an optimal solution.

### 10.3.2. Gomory Cuts

#### Resources

*Michel Bierlaire (EPFL) Teaching Gomory Cuts*

Gomory cuts are a type of Chvátal cut that is derived from the simplex tableau. Specifically, suppose that

$$x_i + \sum_{i \in N} \tilde{a}_i x_i = \tilde{b}_i \quad (10.3)$$

is an equation in the optimal simplex tableau.

#### Gomory Cut:

The Gomory cut corresponding to the tableau row (10.3) is

$$\sum_{i \in N} (\tilde{a}_i - \lfloor \tilde{a}_i \rfloor) x_i \geq \tilde{b}_i - \lfloor \tilde{b}_i \rfloor \quad (10.4)$$

We will solve the following problem using only Gomory Cuts.

$$\begin{array}{lll} \min & x_1 - 2x_2 \\ \text{s.t.} & -4x_1 + 6x_2 & \leq 9 \\ & x_1 + x_2 & \leq 4 \\ & x \geq 0 & , \quad x_1, x_2 \in \mathbb{Z} \end{array}$$

**Step 1:** The first thing to do is to put this into standard form by appending slack variables.

$$\begin{array}{lll} \min & x_1 - 2x_2 \\ \text{s.t.} & -4x_1 + 6x_2 + s_1 & = 9 \\ & x_1 + x_2 + s_2 & = 4 \\ & x \geq 0 & , \quad x_1, x_2 \in \mathbb{Z} \end{array} \quad (10.5)$$

We can apply the simplex method to solve the LP relaxation.

	Basis	RHS	$x_1$	$x_2$	$s_1$	$s_2$
Initial Basis	$z$	0.0	1.0	-2.0	0.0	0.0
	$s_1$	9.0	-4.0	6.0	1.0	0.0
	$s_2$	4.0	1.0	1.0	0.0	1.0
	:		⋮			
Optimal Basis	$Basis$	$RHS$	$x_1$	$x_2$	$s_1$	$s_2$
	$z$	-3.5	0.0	0.0	0.3	0.2
	$x_1$	1.5	1.0	0.0	-0.1	0.6
	$x_2$	2.5	0.0	1.0	0.1	0.4

This LP relaxation produces the fractional basic solution  $x_{LP} = (1.5, 2.5)$ .

### Example 10.5

**(Gomory cut removes LP solution)** We now identify an integer variable  $x_i$  that has a fractional basic solution. Since both variables have fractional values, we can choose either row to make a cut. Let's focus on the row corresponding to  $x_1$ .

The row from the tableau expresses the equation

$$x_1 - 0.1s_1 + -0.6s_2 = 1.5. \quad (10.6)$$

Applying the Gomory Cut (10.4), we have the inequality

$$0.9s_1 + 0.4s_2 \geq 0.5. \quad (10.7)$$

The current LP solution is  $(x_{LP}, s_{LP}) = (1.5, 2.5, 0, 0)$ . Trivially, since  $s_1, s_2 = 0$ , the inequality is violated.

### Example 10.6: (Gomory Cut in Original Space)

The Gomory Cut (10.7) can be rewritten in the original variables using the equations from (10.5). That is, we can use the equations

$$\begin{aligned} s_1 &= 9 + 4x_1 - 6x_2 \\ s_2 &= 4 - x_1 - x_2, \end{aligned} \quad (10.8)$$

which transforms the Gomory cut into the original variables to create the inequality

$$0.9(9 + 4x_1 - 6x_2) + 0.4(4 - x_1 - x_2) \geq 0.5.$$

or equivalently

$$-3.2x_1 + 5.8x_2 \leq 9.2. \quad (10.9)$$

As you can see, this inequality does cut off the current LP relaxation.

### Example 10.7: (Gomory cuts plus new tableau)

Now we add the slack variable  $s_3 \geq 0$  to make the equation

$$0.9s_1 + 0.4s_2 - s_3 = 0.5. \quad (10.10)$$

Next, we need to solve the linear programming relaxation (where we assume the variables are continuous).

## 10.4 Branching Rules

---

There is a few clever ideas out there on how to choose which variables to branch on. We will not go into this here. But for the interested reader, look into

- Strong Branching
- Pseudo-cost Branching

## 10.5 Lagrangian Relaxation for Branch and Bound

---

At each note in the branch and bound tree, we want to bound the objective value. One way to get a good bound can be using the Lagrangian.

### Resources

See [Fisher2004] ([link](#)) for a description of this.

## 10.6 Benders Decomposition

---

### Resources

*Benders Decomposition - Julia Opt*  
*Youtube! SCIP lecture*

## 10.7 Literature

---

## 10.8 Other material for Integer Linear Programming

---

Recall the problem on lemonade and lemon juice from Chapter 5.1:

**Problem.** Say you are a vendor of lemonade and lemon juice. Each unit of lemonade requires 1 lemon and 2 litres of water. Each unit of lemon juice requires 3 lemons and 1 litre of water. Each unit of lemonade gives a profit of \$3. Each unit of lemon juice gives a profit of \$2. You have 6 lemons and 4 litres of water available. How many units of lemonade and lemon juice should you make to maximize profit?

Letting  $x$  denote the number of units of lemonade to be made and letting  $y$  denote the number of units of lemon juice to be made, the problem could be formulated as the following linear programming problem:

$$\begin{aligned} \max \quad & 3x + 2y \\ \text{s.t.} \quad & x + 3y \leq 6 \\ & 2x + y \leq 4 \\ & x \geq 0 \\ & y \geq 0. \end{aligned}$$

The problem has a unique optimal solution at  $\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1.2 \\ 1.6 \end{bmatrix}$  for a profit of 6.8. But this solution requires us to make fractional units of lemonade and lemon juice. What if we require the number of units to be integers? In other words, we want to solve

$$\begin{aligned} \max \quad & 3x + 2y \\ \text{s.t.} \quad & x + 3y \leq 6 \\ & 2x + y \leq 4 \\ & x \geq 0 \\ & y \geq 0 \\ & x, y \in \mathbb{Z}. \end{aligned}$$

This problem is no longer a linear programming problem. But rather, it is an integer linear programming problem.

A **mixed-integer linear programming problem** is a problem of minimizing or maximizing a linear function subject to finitely many linear constraints such that the number of variables are finite and at least one of which is required to take on integer values.

If all the variables are required to take on integer values, the problem is called a **pure integer linear programming problem** or simply an **integer linear programming problem**. Normally, we assume the problem data to be rational numbers to rule out some pathological cases.

Mixed-integer linear programming problems are in general difficult to solve yet they are too important to ignore because they have a wide range of applications (e.g. transportation planning, crew scheduling, circuit design, resource management etc.) Many solution methods for these problems have been devised and some of them first solve the **linear programming relaxation** of the original problem, which is the problem obtained from the original problem by dropping all the integer requirements on the variables.

**Example 10.8**

Let (MP) denote the following mixed-integer linear programming problem:

$$\begin{array}{lllll} \min & x_1 & + & x_3 \\ \text{s.t.} & -x_1 & + & x_2 & + & x_3 \geq 1 \\ & -x_1 & - & x_2 & + & 2x_3 \geq 0 \\ & -x_1 & + & 5x_2 & - & x_3 = 3 \\ & x_1, & x_2, & x_3 & \geq 0 \\ & & & x_3 & \in \mathbb{Z}. \end{array}$$

The linear programming relaxation of (MP) is:

$$\begin{array}{lllll} \min & x_1 & + & x_3 \\ \text{s.t.} & -x_1 & + & x_2 & + & x_3 \geq 1 \\ & -x_1 & - & x_2 & + & 2x_3 \geq 0 \\ & -x_1 & + & 5x_2 & - & x_3 = 3 \\ & x_1, & x_2, & x_3 & \geq 0. \end{array}$$

Let (P1) denote the linear programming relaxation of (MP). Observe that the optimal value of (P1) is a lower bound for the optimal value of (MP) since the feasible region of (P1) contains all the feasible solutions to (MP), thus making it possible to find a feasible solution to (P1) with objective function value better than the optimal value of (MP). Hence, if an optimal solution to the linear programming relaxation happens to be a feasible solution to the original problem, then it is also an optimal solution to the original problem. Otherwise, there is an integer variable having a nonintegral value  $v$ . What we then do is to create two new subproblems as follows: one requiring the variable to be at most the greatest integer less than  $v$ , the other requiring the variable to be at least the smallest integer greater than  $v$ . This is the basic idea behind the **branch-and-bound method**. We now illustrate these ideas on (MP).

Solving the linear programming relaxation (P1), we find that  $\mathbf{x}' = \begin{bmatrix} 0 \\ \frac{2}{3} \\ \frac{1}{3} \end{bmatrix}$  is an optimal solution to (P1).

Note that  $\mathbf{x}'$  is not a feasible solution to (MP) because  $x'_3$  is not an integer. We now create two subproblems (P2) and (P3) such that (P2) is obtained from (P1) by adding the constraint  $x_3 \leq \lfloor x'_3 \rfloor$  and (P3) is obtained from (P1) by adding the constraint  $x_3 \geq \lceil x'_3 \rceil$ . (For a number  $a$ ,  $\lfloor a \rfloor$  denotes the greatest integer at most  $a$  and  $\lceil a \rceil$  denotes the smallest integer at least  $a$ .) Hence, (P2) is the problem

$$\begin{array}{lllll} \min & x_1 & + & x_3 \\ \text{s.t.} & -x_1 & + & x_2 & + & x_3 \geq 1 \\ & -x_1 & - & x_2 & + & 2x_3 \geq 0 \\ & -x_1 & + & 5x_2 & - & x_3 = 3 \\ & & & x_3 & \leq 0 \\ & x_1, & x_2, & x_3 & \geq 0, \end{array}$$

and (P3) is the problem

$$\begin{array}{lllll} \min & x_1 & + & x_3 \\ \text{s.t.} & -x_1 + x_2 + x_3 & \geq & 1 \\ & -x_1 - x_2 + 2x_3 & \geq & 0 \\ & -x_1 + 5x_2 - x_3 & = & 3 \\ & & x_3 & \geq & 1 \\ & x_1, x_2, x_3 & \geq & 0. \end{array}$$

Note that any feasible solution to (MP) must be a feasible solution to either (P2) or (P3). Using the help

of a solver, one sees that (P2) is infeasible. The problem (P3) has an optimal solution at  $\mathbf{x}^* = \begin{bmatrix} 0 \\ \frac{4}{5} \\ 1 \end{bmatrix}$ , which

is also feasible to (MP). Hence,  $\mathbf{x}^* = \begin{bmatrix} 0 \\ \frac{4}{5} \\ 1 \end{bmatrix}$  is an optimal solution to (MP).

We now give a description of the method for a general mixed-integer linear programming problem (MIP). Suppose that (MIP) is a minimization problem and has  $n$  variables  $x_1, \dots, x_n$ . Let  $\mathcal{I} \subseteq \{1, \dots, n\}$  denote the set of indices  $i$  such that  $x_i$  is required to be an integer in (MIP).

### Branch-and-bound method

**Input:** The problem (MIP).

**Steps:**

1. Set  $\text{bestbound} := \infty$ ,  $\mathbf{x}_{\text{best}}^* := \text{N/A}$ ,  $\text{activeproblems} := \{(LP)\}$  where  $(LP)$  denotes the linear programming relaxation of (MIP).
2. If there is no problem in  $\text{activeproblems}$ , then stop; if  $\mathbf{x}_{\text{best}}^* \neq \text{N/A}$ , then  $\mathbf{x}_{\text{best}}^*$  is an optimal solution; otherwise, (MIP) has no optimal solution.
3. Select a problem  $P$  from  $\text{activeproblems}$  and remove it from  $\text{activeproblems}$ .
4. Solve  $P$ .
  - If  $P$  is unbounded, then stop and conclude that (MIP) does not have an optimal solution.
  - If  $P$  is infeasible, go to step 2.
  - If  $P$  has an optimal solution  $\mathbf{x}^*$ , then let  $z$  denote the objective function value of  $\mathbf{x}^*$ .
    5. If  $z \geq \text{bestbound}$ , go to step 2.
    6. If  $x_i^*$  is not an integer for some  $i \in \mathcal{I}$ , then create two subproblems  $P_1$  and  $P_2$  such that  $P_1$  is the problem obtained from  $P$  by adding the constraint  $x_i \leq \lfloor x_i^* \rfloor$  and  $P_2$  is the problem obtained from  $P$  by adding the constraint  $x_i \geq \lceil x_i^* \rceil$ . Add the problems  $P_1$  and  $P_2$  to  $\text{activeproblems}$  and go to step 2.
    7. Set  $\mathbf{x}_{\text{best}}^* = \mathbf{x}^*$ ,  $\text{bestbound} = z$  and go to step 2.

**Remarks.**

- Throughout the algorithm, `activeproblems` is a set of subproblems remained to be solved. Note that for each problem  $P$  in `activeproblems`,  $P$  is a linear programming problem and that any feasible solution to  $P$  satisfying the integrality requirements is a feasible solution to (MIP).
- $x_{\text{best}}^*$  is the feasible solution to (MIP) that has the best objective function value found so far and `bestbound` is its objective function value. It is often called an **incumbent**.
- In practice, how a problem from `activeproblems` is selected in step 3 has an impact on the overall performance. However, there is no general rule for selection that guarantees good performance all the time.
- In step 5, the problem  $P$  is discarded since it cannot contain any feasible solution to (MIP) having a better objective function value than  $x_{\text{best}}^*$ .
- If step 7 is reached, then  $x^*$  is a feasible solution to (MIP) having objective function value better than `bestbound`. So it becomes the current best solution.
- It is possible for the algorithm to never terminate. Below is an example for which the algorithm will never stop:

$$\begin{aligned} \min \quad & x_1 \\ \text{s.t. } & x_1 + 2x_2 - 2x_3 = 1 \\ & x_1, x_2, x_3 \geq 0 \\ & x_1, x_2, x_3 \in \mathbb{Z}. \end{aligned}$$

However, it is easy to see that  $\mathbf{x}^* = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$  is an optimal solution because there is no feasible solution with  $x_1 = 0$ .

One way to keep track of the progress of the computations is to set up a progress chart with the following headings:

Iter	solved	status	branching	activeproblems	$\mathbf{x}_{\text{best}}^*$	bestbound
------	--------	--------	-----------	----------------	------------------------------	-----------

In a given iteration, the entry in the **solved** column denotes the subproblem that has been solved with the result in the **status** column. The **branching** column indicates the subproblems created from the solved subproblem with an optimal solution not feasible to (MIP). The entries in the remaining columns contain the latest information in the given iteration. For the example (MP) above, the chart could look like the following:

Iter	solved	status	branching	active problems	$\mathbf{x}^*_{\text{best}}$	bestbound
1	(P1)	optimal $\mathbf{x}^* = \begin{bmatrix} 0 \\ \frac{2}{3} \\ \frac{1}{3} \end{bmatrix}$	(P2): $x_3 \leq 0$ , (P3): $x_3 \geq 1$	(P2), (P3)	N/A	$\infty$
2	(P2)	infeasible	—	(P3)	N/A	$\infty$
3	(P3)	optimal $\mathbf{x}^* = \begin{bmatrix} 0 \\ \frac{4}{5} \\ 1 \end{bmatrix}$	—	—	$\begin{bmatrix} 0 \\ \frac{4}{5} \\ 1 \end{bmatrix}$	1

## Exercises

---

- Suppose that (MP) in Example 10.8 above has  $x_2$  required to be an integer as well. Continue with the computations and determine an optimal solution to the modified problem.
- With the help of a solver, determine the optimal value of

$$\begin{aligned} \max \quad & 3x + 2y \\ \text{s.t.} \quad & x + 3y \leq 6 \\ & 2x + y \leq 4 \\ & x, y \geq 0 \\ & x, y \in \mathbb{Z}. \end{aligned}$$

- Let  $\mathbf{A} \in \mathbb{Q}^{m \times n}$  and  $\mathbf{b} \in \mathbb{Q}^m$ . Let  $S$  denote the system

$$\begin{aligned} \mathbf{Ax} &\geq \mathbf{b} \\ \mathbf{x} &\in \mathbb{Z}^n \end{aligned}$$

- Suppose that  $\mathbf{d} \in \mathbb{Q}^m$  satisfies  $\mathbf{d} \geq 0$  and  $\mathbf{d}^\top \mathbf{A} \in \mathbb{Z}^n$ . Prove that every  $\mathbf{x}$  satisfying  $S$  also satisfies  $\mathbf{d}^\top \mathbf{Ax} \geq \lceil \mathbf{d}^\top \mathbf{b} \rceil$ . (This inequality is known as a **Chvátal-Gomory cutting plane**.)
- Suppose that  $\mathbf{A} = \begin{bmatrix} 2 & 3 \\ 5 & 3 \\ 7 & 6 \end{bmatrix}$  and  $\mathbf{b} = \begin{bmatrix} 2 \\ 1 \\ 8 \end{bmatrix}$ . Show that every  $\mathbf{x}$  satisfying  $S$  also satisfies  $x_1 + x_2 \geq 2$ .

## Solutions

---

1. An optimal solution to the modified problem is given by  $x^* = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$ .
2. An optimal solution is  $\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 2 \\ 0 \end{bmatrix}$ . Thus, the optimal value is 6.
3. a. Since  $\mathbf{d} \geq 0$  and  $\mathbf{Ax} \geq \mathbf{b}$ , we have  $\mathbf{d}^T \mathbf{Ax} \geq \mathbf{d}^T \mathbf{b}$ . If  $\mathbf{d}^T \mathbf{b}$  is an integer, the result follows immediately. Otherwise, note that  $\mathbf{d}^T \mathbf{A} \in \mathbb{Z}^n$  and  $\mathbf{x} \in \mathbb{Z}^n$  imply that  $\mathbf{d}^T \mathbf{Ax}$  is an integer. Thus,  $\mathbf{d}^T \mathbf{Ax}$  must be greater than or equal to the least integer greater than  $\mathbf{d}^T \mathbf{b}$ .
   
b. Take  $\mathbf{d} = \begin{bmatrix} \frac{1}{9} \\ 0 \\ \frac{1}{9} \end{bmatrix}$  and apply the result in the previous part.

### 10.8.1. Other discrete problems

---

### 10.8.2. Assignment Problem and the Hungarian Algorithm

---

**Assignment Problem:**

*Polynomial time (P)*

$$\begin{aligned}
 & \min \langle C, X \rangle \\
 \text{s.t. } & \sum_i X_{ij} = 1 \text{ for all } j \\
 & \sum_j X_{ij} = 1 \text{ for all } i \\
 & X_{ij} \in \{0, 1\} \text{ for all } i = 1, \dots, n, j = 1, \dots, m
 \end{aligned} \tag{10.1}$$

This problem is efficiently solved by the Hungarian Algorithm.

### 10.8.3. History of Computation in Combinatorial Optimization

---

Book: Computing in Combinatorial Optimization by William Cook, 2019

Model	LP Solution
$\begin{array}{ll} \max & x_1 + x_2 \\ \text{subject to} & -2x_1 + x_2 \leq 0.5 \\ & x_1 + 2x_2 \leq 10.5 \\ & x_1 - x_2 \leq 0.5 \\ & -2x_1 - x_2 \leq -2 \end{array}$	
$\begin{array}{ll} \max & x_1 + x_2 \\ \text{subject to} & -2x_1 + x_2 \leq 0.5 \\ & x_1 + 2x_2 \leq 10.5 \\ & x_1 - x_2 \leq 0.5 \\ & -2x_1 - x_2 \leq -2 \\ & x_1 \leq 3 \end{array}$	
$\begin{array}{ll} \max & x_1 + x_2 \\ \text{subject to} & -2x_1 + x_2 \leq 0.5 \\ & x_1 + 2x_2 \leq 10.5 \\ & x_1 - x_2 \leq 0.5 \\ & -2x_1 - x_2 \leq -2 \\ & x_1 \leq 3 \\ & x_1 + x_2 \leq 6 \end{array}$	

© cutting-plane-1-picture<sup>11</sup>© cutting-plane-2-picture<sup>12</sup>© cutting-plane-3-picture<sup>13</sup>

# 11. Heuristics for TSP

---



In this section we will show how different heuristics can find good solutions to TSP. For convenience, we will focus on the *symmetric TSP* problem. That is, the distance  $d_{ij}$  from node  $i$  to node  $j$  is the same as the distance  $d_{ji}$  from node  $j$  to node  $i$ .

There are two general types of heuristics: construction heuristics and improvement heuristics. We will first discuss a few construction heuristics for TSP.

Then we will demonstrate three types of metaheuristics for improvement- Hill Climbing, Tabu Search, and Simulated Annealing. These are called *metaheuristics* because they are a general framework for a heuristic that can be applied to many different types of settings. Furthermore, Tabu Search and Simulated Annealing have parameters that can be adjusted to try to find better solutions.

## 11.1 Construction Heuristics

---

### 11.1.1. Random Solution

---

TSP is convenient in that choosing a random ordering of the nodes creates a feasible solution. It may not be a very good one, but it is at least a solution.

**Random Construction:**Complexity:  $O(n)$ For  $i = 1, \dots, n$ , randomly choose a node not yet in the tour and place it at the end of the tour.

### 11.1.2. Nearest Neighbor

---

Starting from any node, add the edge to the next closest node. Continue this process.

**Nearest Neighbor:**Complexity:  $O(n^2)$ 

1. Start from any node (lets call this node 1) and label this as your current node.
2. Pick the next current node as the one that is closest to the current node that has not yet been visited.
3. Repeat step 2 until all nodes are in the tour.

### 11.1.3. Insertion Method

---

**Insertion Method:**Complexity:  $O(n^2)$ 

1. Start from any 3 nodes (lets call this node 1) and label this as your current node.
2. Pick the next current node as the one that is closest to the current node that has not yet been visited.
3. Repeat step 2 until all nodes are in the tour.

## 11.2 Improvement Heuristics

---

There are many ways to generate improving steps. The key features of improving step to consider are

- What is the complexity of computing this improving step?
- How good this this improving step?

We will mention ways to find neighbors of a current solution for TSP. If the neighbor has a better objective value, the moving to this neighbor will be an improving step.

### 11.2.1. 2-Opt (Subtour Reversal)

---

We will assume that all tours start and end with then node 1.

**2-Opt (Subtour reversal):**

Input a tour  $1 \rightarrow \dots \rightarrow 1$ .

1. Pick distinct nodes  $i, j \neq 1$ .
2. Let  $s, t$  and  $x_1, \dots, x_k$  be nodes in the tour such that it can be written as

$$1 \rightarrow \dots \rightarrow s \rightarrow i \rightarrow x_1 \rightarrow \dots \rightarrow x_k \rightarrow j \rightarrow t \rightarrow \dots \rightarrow 1.$$

3. Consider the subtour reversal

$$1 \rightarrow \dots \rightarrow s \rightarrow j \rightarrow x_k \rightarrow \dots \rightarrow x_1 \rightarrow i \rightarrow t \rightarrow \dots \rightarrow 1.$$

Thus, we reverse the order of  $i \rightarrow x_1 \rightarrow \dots \rightarrow x_k \rightarrow j$ .

4. In this process, we

- deleted the edges  $(s, i)$  and  $(j, t)$
- added the edges  $(s, j)$  and  $(i, t)$

Pictorially, this looks like the following

Add 2-opt figures

Computationally, we need to consider the costs on the edges of a graph.....

See [Englert2014] for an analysis of performance of this improvement.

## 11.2.2. 3-Opt

---

## 11.2.3. $k$ -Opt

---

This is a generalization of 2-Opt and 3-Opt.

# 11.3 Meta-Heuristics

---

## 11.3.1. Hill Climbing (2-Opt for TSP)

---

The *Hill Climbing* algorithm finds an improving neighboring solution and climbs in that direction. It continues this process until there is no other neighbor that is improving.

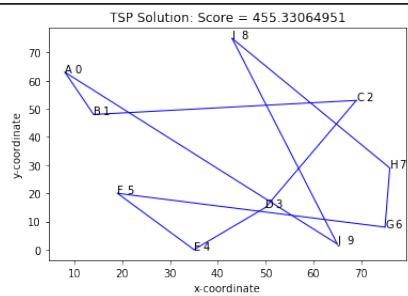
In the context of TSP, we will consider 2-Opt improving moves and the Hill Climbing algorithm for TSP in this case is referred to as the 2-Opt algorithm (also known as the Subtour Reversal Algorithm).

### Hill Climbing:

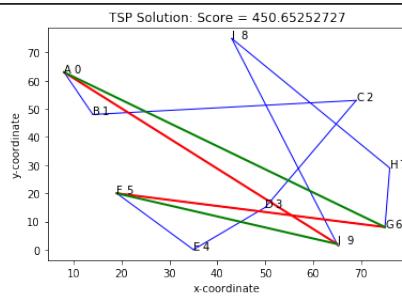
1. Start with an initial feasible solution, label it as the current solution.
2. List all neighbors of the current solution.
3. If no neighbor has a better solution, then stop.
4. Otherwise, move to the best neighbor and go to Step 2.

Here is an example on the TSP problem with 2-Opt swaps:

**Current solution**

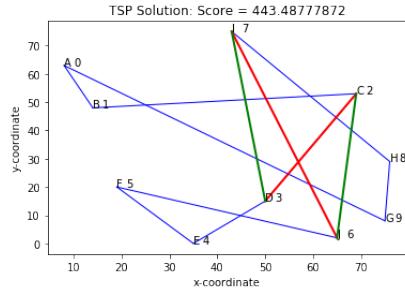
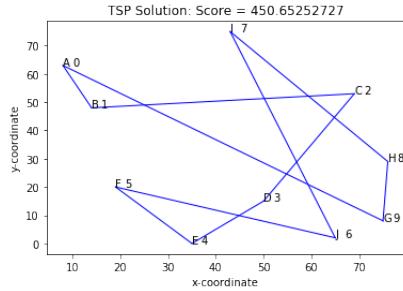


**Improvement Step**

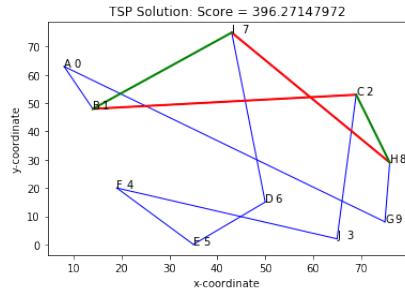
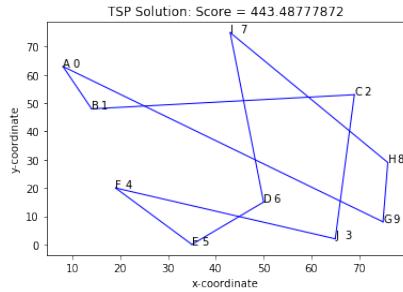


**Reversal**

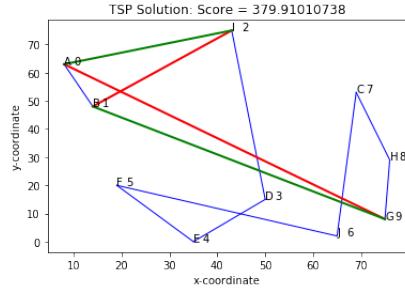
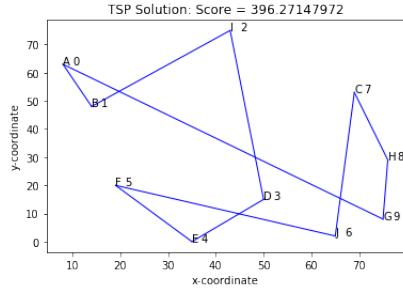
[A, B, C, D, E, F, G, H, I, J]



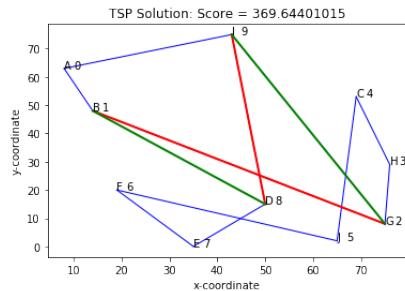
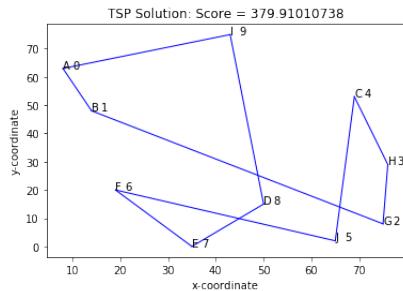
[A, B, C, D, E, F, J, I, H, G]



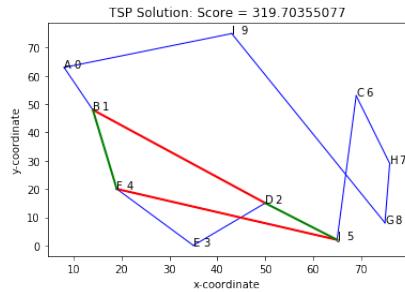
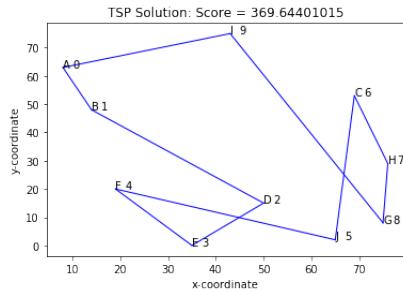
[A, B, C, J, F, E, D, I, H, G]



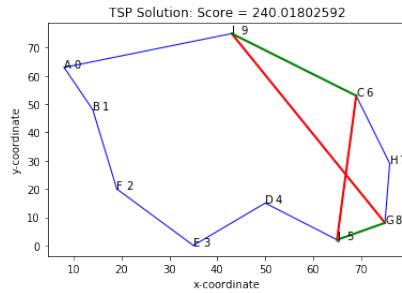
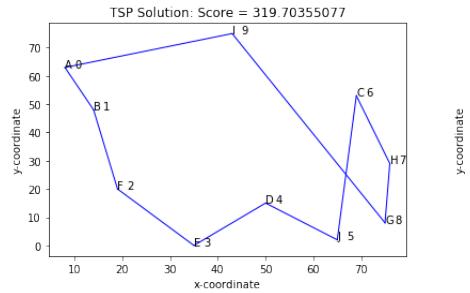
[A, B, I, D, E, F, J, C, H, G]



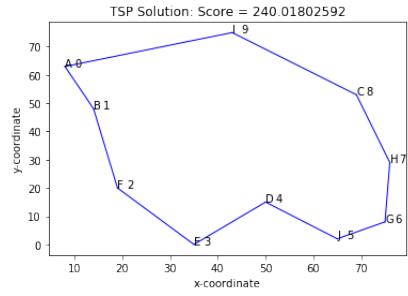
[A, B, G, H, C, J, F, E, D, I]



[A, B, D, E, F, J, C, H, G, I]



[A, B, F, E, D, J, C, H, G, I]



No Improvement

[A, B, F, E, D, J, G, H, C, I]

### 11.3.2. Simulated Annealing

Here is a great python package for TSP Simulated annealing: <https://pypi.org/project/satsp/>.

Simulated annealing is a randomized heuristic that randomly decides when to accept non-improving moves. For this, we use what is referred to as a *temperature schedule*. The temperature schedule guides a parameter  $T$  that goes into deciding the probability of accepting a non-improving move.

A typical temperature schedule starts at a value  $T = 0.2 \times Z_c$ , where  $Z_c$  is the objective value of an initial feasible solution. Then the temperature is decreased over time to smaller values.

**Temperature schedule example:**

- $T_1 = 0.2Z_c$
- $T_2 = 0.8T_1$
- $T_3 = 0.8T_2$
- $T_4 = 0.8T_3$
- $T_5 = 0.8T_4$

For instance, we could choose to run the algorithm for 10 iterations at each temperature value. The Simulated Annealing algorithm is the following:

**Simulated Annealing Outline:**

[minimization version]

1. Start with an initial feasible solution, label it as the current solution, and compute its object value  $Z_c$ .
2. Select a neighbor of the current solution and compute its objective value  $Z_n$ .
3. Decide whether or not to move to the neighbor:
  - (a) If the neighbor is an improvement ( $Z_n < Z_c$ ), **accept the move** and set the neighbor as the current solution.
  - (b) Otherwise,  $Z_c < Z_n$  and thus  $Z_c - Z_n < 0$ . Now with probability  $e^{\frac{Z_c - Z_n}{T}}$  accept the move. In detail:
    - Compute the number  $p = e^{\frac{Z_c - Z_n}{T}}$
    - Generate a random number  $x \in [0, 1]$  from the computer.
    - If  $x < p$ , then **accept the move**.
    - Otherwise, if  $x \geq p$ , **reject the move** and stay at the current solution.
4. While still iterations left in the schedule, update the temperature  $T$  and go to Step 2.
5. Return the best found solution during the algorithm.

### 11.3.3. Tabu Search

---

#### Tabu Search Outline:

[minimization version]

1. Initialize a *Tabu List* as an empty set:  $\text{Tabu} = \{\}$ .
2. Start with an initial feasible solution, label it as the current solution.
3. List all neighbors of the current solution.
4. Choose the best neighbor that is not tabu to move too (the move should not be restricted by the set  $\text{Tabu}$ .)
5. Add moves to the Tabu List.
6. If the Tabu List is longer than its designated maximal size  $S$ , then remove old moves in the list until it reaches the size  $S$ .

7. If no object improvement has been seen for  $K$  steps, then Stop.
8. Otherwise, Go to Step 3 and continue.

### 11.3.4. Genetic Algorithms

---

Genetic algorithms start with a set of possible solutions, and then slowly mutate them to better solutions. See Scikit-opt for an implementation for the TSP.

### 11.3.5. Greedy randomized adaptive search procedure (GRASP)

---

We currently do not cover this topic.

[Wikipedia - GRASP](#)

For an in depth (and recent) book, check out Optimization by GRASP Greedy Randomized Adaptive Search Procedures Authors: Resende, Mauricio, Ribeiro, Celso C..

### 11.3.6. Ant Colony Optimization

---

[Wikipedia - Ant Colony Optimization](#)

## 11.4 Computational Comparisons

---

Notice how the heuristics are generally faster and provide reasonable solutions, but the solvers provide the best solutions. This is a trade off to consider when deciding how fast you need a solution and how good of a solution it is that you actually need.

On an instance with 5 nodes:

```
Nearest Neighbor
494
0.000065 seconds (58 allocations: 2.172 KiB)
```

```
Farthest Insertion
494
0.000057 seconds (49 allocations: 1.781 KiB)
```

Simulated Annealing

494  
0.000600 seconds (7.81 k allocations: 162.156 KiB)

Math Programming Cbc  
494.0  
0.091290 seconds (26.29 k allocations: 1.460 MiB)

Math Programming Gurobi  
Academic license - for non-commercial use only  
494.0  
0.006610 seconds (780 allocations: 78.797 KiB)

One instance on 20 nodes.

Nearest Neighbor  
790  
0.000162 seconds (103 allocations: 6.406 KiB)

Farthest Insertion  
791  
0.000128 seconds (58 allocations: 2.734 KiB)

Simulated Annealing  
777  
0.007818 seconds (130.31 k allocations: 2.601 MiB)

Math Programming Cbc  
773.0  
2.738521 seconds (5.76 k allocations: 607.961 KiB)

Math Programming Gurobi  
Academic license - for non-commercial use only  
773.0  
0.238488 seconds (5.68 k allocations: 717.133 KiB)

Nearest Neighbor  
1216  
0.000288 seconds (142 allocations: 15.141 KiB)

Farthest Insertion  
1281  
0.000286 seconds (60 allocations: 3.969 KiB)

```

Simulated Annealing
1227
0.047512 seconds (520.51 k allocations: 10.387 MiB, 19.12% gc time)

Math Programming Cbc
1088.0
6.292632 seconds (20.30 k allocations: 2.111 MiB)

Math Programming Gurobi
Academic license - for non-commercial use only
1088.0
1.349253 seconds (20.16 k allocations: 2.520 MiB)

```

## 11.5 Polyhedra

---

### 11.5.1. Convex Hull

---

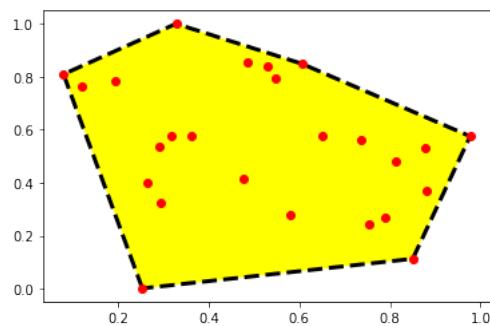
```

[[0.78913 0.2673 ]
 [0.5462  0.79165]
 [0.29012 0.53835]
 [0.97705 0.57501]
 [0.32814 0.99928]
 [0.87902 0.53254]
 [0.36117 0.57705]
 [0.48521 0.85347]
 [0.75269 0.24459]
 [0.81315 0.47965]
 [0.60529 0.84873]
 [0.64951 0.57425]
 [0.57901 0.27927]
 [0.53027 0.83896]
 [0.85029 0.11324]
 [0.19383 0.7854 ]
 [0.07971 0.80899]
 [0.4783  0.41661]
 [0.88042 0.37067]
 [0.25369 0.00183]
 [0.73571 0.56145]
```

```
[0.29421 0.32336]  
[0.11878 0.76099]  
[0.2648 0.40023]  
[0.31803 0.57827]]
```

Single polytope

```
[[ 0.96433 -0.26471] | [[ 0.78998]  
[-0.60809 0.79387] | [ 0.59376]  
[ 0.18357 -0.98301] x <= [ 0.04477]  
[-0.97755 -0.2107 ] | [-0.24838]  
[ 0.5929 0.80528] | [ 1.04234]  
[ 0.47734 0.87872]] | [ 1.03472]]
```





# 12. Dynamic Programming

---

Repository of Dynamic Programming Examples

**Lab Objective:** Sequential decision making problems are a class of problems in which the current choice depends on future choices. They are a subset of Markov decision processes, an important class of problems with applications in business, robotics, and economics. Dynamic programming is a method of solving these problems that optimizes the solution by breaking the problem down into steps and optimizing the decision at each time period. In this lab we use dynamic programming to solve two classic dynamic optimization problems.

## The Marriage Problem

---

Many dynamic optimization problems can be classified as *optimal stopping* problems, where the goal is to determine at what time to take an action to maximize the expected reward. For example, when hiring a secretary, how many people should you interview before hiring the current interviewer? Or how many people should you date before you get married? These problems try to determine at what person  $t$  to stop in order to maximize the chance of getting the best candidate.

For instance, let  $N$  be the number of people you could date. After dating each person, you can either marry them or move on; you can't resume a relationship once it ends. In addition, you can rank your current relationship to all of the previous options, but not to future ones. The goal is to find the policy that maximizes the probability of choosing the best marriage partner. That policy may not always choose the best candidate, but it should get an almost-best candidate most of the time.

Let  $V(t - 1)$  be the probability that we choose the best partner when we have passed over the first  $t - 1$  candidates with an optimal policy. In other words, we have dated  $t - 1$  people and want to know the probability that the  $t^{th}$  person is the one we should marry. Note that the probability that the  $t^{th}$  person is not the best candidate is  $\frac{t-1}{t}$  and the probability that they are is  $\frac{1}{t}$ . If the  $t^{th}$  person is not the best out of the first  $t$ , then probability they are the best overall is 0 and the probability they are not is  $V(t)$ . If the  $t^{th}$  person is the best out of the first  $t$ , then the probability they are the best overall is  $\frac{t}{N}$  and the probability they are not is  $V(t)$ .

By Bellman's optimality equations,

$$V(t - 1) = \frac{t - 1}{t} \max \{0, V(t)\} + \frac{1}{t} \max \left\{ \frac{t}{N}, V(t) \right\} = \max \left\{ \frac{t - 1}{t} V(t) + \frac{1}{N}, V(t) \right\}. \quad (12.1)$$

Notice that (12.1) implies that  $V(t - 1) \geq V(t)$  for all  $t \leq N$ . Hence, the probability of selecting the best match  $V(t)$  is non-increasing. Conversely,  $P(\text{t is best overall} | t \text{ is best out of the first } t) = \frac{t}{N}$  is strictly

increasing. Therefore, there is some  $t_0$ , called the *optimal stopping point*, such that  $V(t) \leq \frac{t}{N}$  for all  $t \geq t_0$ . After  $t_0$  relationships, we choose the next partner who is better than all of the previous ones. We can write (12.1) as

$$V(t-1) = \begin{cases} V(t_0) & t < t_0, \\ \frac{t-1}{t}V(t) + \frac{1}{N} & t \geq t_0. \end{cases}$$

The goal of an optimal stopping problem is to find  $t_0$ , which we can do by backwards induction. We start at the final candidate, who always has probability 0 of being the best overall if they are not the best so far, and work our way backwards, computing the expected value  $V(t)$ , for  $t = N, N-1, \dots, 1$ .

If  $N = 4$ , we have

$$\begin{aligned} V(4) &= 0, \\ V(3) &= \max \left\{ \frac{3}{4}V(4) + \frac{1}{4}, 0 \right\} = .25, \\ V(2) &= \max \left\{ \frac{2}{3}V(3) + \frac{1}{4}, .25 \right\} = .4166, \\ V(1) &= \max \left\{ \frac{1}{4}, .4166 \right\} = .4166. \end{aligned}$$

In this case, the maximum expected value is .4166 and the stopping point is  $t = 2$ . It is also useful to look at the optimal stopping percentage of people to date before getting married. In this case, it is  $2/4 = .5$ .

### Problem 12.1: W

ite a function that accepts a number of candidates  $N$ . Calculate the expected values of choosing candidate  $t$  for  $t = 0, 1, \dots, N-1$ .

Return the highest expected value  $V(t_0)$  and the optimal stopping point  $t_0$ .

(Hint: Since Python starts indices at 0, the first candidate is  $t = 0$ .)

Check your answer for  $N = 4$  with the example detailed above.

### Problem 12.2: W

ite a function that takes in an integer  $M$  and runs your function from Problem 12 for each  $N = 3, 4, \dots, M$ . Graph the optimal stopping percentage of candidates ( $t_0/N$ ) to interview and the maximum probability  $V(t_0)$  against  $N$ . Return the optimal stopping percentage for  $M$ .

The optimal stopping percentage for  $M = 1000$  is .367.

Both the stopping time and the probability of choosing the best person converge to  $\frac{1}{e} \approx .36788$ . Then to maximize the chance of having the best marriage, you should date at least  $\frac{N}{e}$  people before choosing the next best person. This famous problem is also known as the *secretary problem*, the *sultan's dowry problem*, and the *best choice problem*. For more information, see [https://en.wikipedia.org/wiki/Secretary\\_problem](https://en.wikipedia.org/wiki/Secretary_problem).

# The Cake Eating Problem

---

Imagine you are given a cake. How do you eat it to maximize your enjoyment? Some people may prefer to eat all of their cake at once and not save any for later. Others may prefer to eat a little bit at a time. If we are to consume a cake of size  $W$  over  $T + 1$  time periods, then our consumption at each step is represented as a vector

$$\mathbf{c} = [c_0 \ c_1 \ \cdots \ c_T]^\top,$$

where

$$\sum_{i=0}^T c_i = W.$$

This vector is called a *policy vector* and describes how much cake is eaten at each time period. The enjoyment of eating a slice of cake is represented by a utility function. For some amount of consumption  $c_i \in [0, W]$ , the utility gained is given by  $u(c_i)$ .

For this lab, we assume the utility function satisfies  $u(0) = 0$ , that  $W = 1$ , and that  $W$  is cut into  $N$  equally-sized pieces so that each  $c_i$  must be of the form  $\frac{i}{N}$  for some integer  $0 \leq i \leq N$ .

## Discount Factors

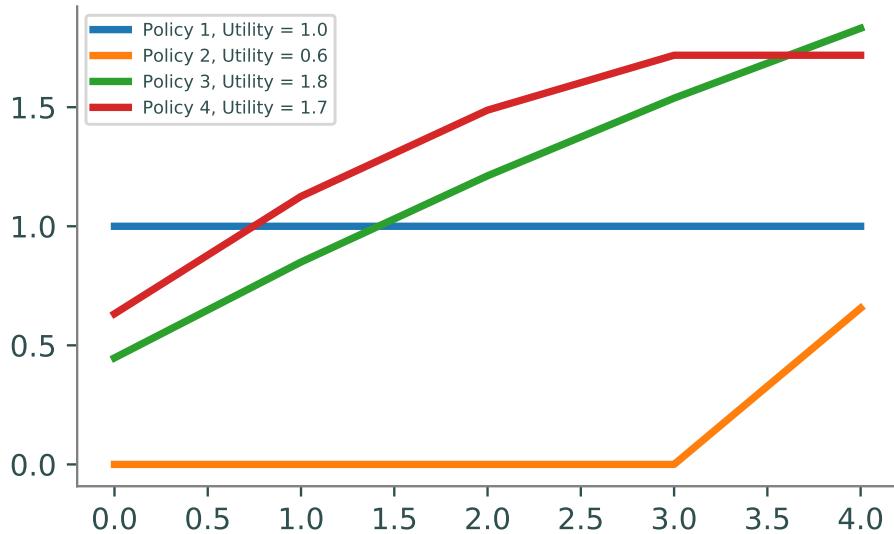
---

A person or firm typically has a time preference for saving or consuming. For example, a dollar today can be invested and yield interest, whereas a dollar received next year does not include the accrued interest. Since cake gets stale as it gets older, we assume that cake in the present yields more utility than cake in the future. We can model this by multiplying future utility by a discount factor  $\beta \in (0, 1)$ . For example, if we were to consume  $c_0$  cake at time 0 and  $c_1$  cake at time 1, with  $c_0 = c_1$  then the utility gained at time 0 is larger than the utility at time 1:

$$u(c_0) > \beta u(c_1).$$

The total utility for eating the cake is

$$\sum_{t=0}^T \beta^t u(c_t).$$



**Figure 12.1:** Plots for various policies with  $u(x) = \sqrt{x}$  and  $\beta = 0.9$ . Policy 1 eats all of the cake in the first step while policy 2 eats all of the cake in the last step. Their difference in utility demonstrate the effect of the discount factor on waiting to eat. Policy 3 eats the same amount of cake at each step, while policy 4 begins by eating .4 of the cake, then .3, .2, and .1.

## The Value Function

---

The cake eating problem is an optimization problem where we maximize utility.

$$\begin{aligned}
 & \max_{\mathbf{c}} \sum_{t=0}^T \beta^t u(c_t) \\
 & \text{subject to } \sum_{t=0}^T c_t = W \\
 & \quad c_t \geq 0.
 \end{aligned} \tag{12.2}$$

One way to solve it is with the value function. The value function  $V(a, b, W)$  gives the utility gained from following an optimal policy from time  $a$  to time  $b$ .

$$\begin{aligned}
V(a, b, W) &= \max_{\mathbf{c}} \sum_{t=a}^b \beta^t u(c_t) \\
\text{subject to } &\sum_{t=a}^b c_t = W \\
&c_t \geq 0.
\end{aligned}$$

$V(0, T, W)$  gives how much utility we gain in  $T$  days and is the same as Equation 12.2.

Let  $W_t$  represent the total amount of cake left at time  $t$ . Observe that  $W_{t+1} \leq W_t$  for all  $t$ , because our problem does not allow for the creation of more cake. Notice that  $V(t+1, T, W_{t+1})$  can be represented by  $\beta V(t, T-1, W_{t+1})$ , which is the value of eating  $W_{t+1}$  cake later. Then we can express the value function as the sum of the utility of eating  $W_t - W_{t+1}$  cake now and  $W_{t+1}$  cake later.

$$V(t, T, W_t) = \max_{W_{t+1}} (u(W_t - W_{t+1}) + \beta V(t, T-1, W_{t+1})) \quad (12.3)$$

where  $u(W_t - W_{t+1})$  is the value gained from eating  $W_t - W_{t+1}$  cake at time  $t$ .

Let  $\mathbf{w} = [0 \quad \frac{1}{N} \quad \dots \quad \frac{N-1}{N} \quad 1]^T$ . We define the *consumption matrix*  $C$  by  $C_{ij} = u(w_i - w_j)$ . Note that  $C$  is an  $(N+1) \times (N+1)$  lower triangular matrix since we assume  $j \leq i$ ; we can't consume more cake than we have. The consumption matrix will help solve the value function by calculating all possible value of  $u(W_t - W_{t+1})$  at once. At each time  $t$ ,  $W_t$  can only have  $N+1$  values, which will be represented as  $w_i = \frac{i}{N}$ , which is  $i$  pieces of cake remaining. For example, if  $N = 4$ , then  $\mathbf{w} = [0, .25, .5, .75, 1]^T$ , and  $w_3 = 0.75$  represents having three pieces of cake left. In this case, we get the following consumption matrix.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ u(0.25) & 0 & 0 & 0 & 0 \\ u(0.5) & u(0.25) & 0 & 0 & 0 \\ u(0.75) & u(0.5) & u(0.25) & 0 & 0 \\ u(1) & u(0.75) & u(0.5) & u(0.25) & 0 \end{bmatrix}.$$

### Problem 12.3

Write a function that accepts the number of equal sized pieces  $N$  that divides the cake and a utility function  $u(x)$ . Assume  $W = 1$ . Create a partition vector  $\mathbf{w}$  whose entries correspond to possible amounts of cake. Return the consumption matrix.

## Solving the Optimization Problem

---

Initially we do not know how much cake to eat at  $t = 0$ : should we eat one piece of cake ( $w_1$ ), or perhaps all of the cake ( $w_N$ )? It may not be obvious which option is best and that option may change depending on the discount factor  $\beta$ . Instead of asking how much cake to eat at some time  $t$ , we ask how valuable  $w_i$  cake is at time  $t$ . As mentioned above,  $V(t, T - 1, W_{t+1})$  in 12.3 is a new value function problem with  $a = t, b = T - 1$ , and  $W = W_{t+1}$ , making 12.3 a recursion formula. By using the optimal value of the value function in the future,  $V(t, T - 1, W_{t+1})$ , we can determine the optimal value for the present,  $V(t, T, W_t)$ .  $V(t, T, W_t)$  can be solved by trying each possible  $W_{t+1}$  and choosing the one that gives the highest utility.

The  $(N + 1) \times (T + 1)$  matrix  $A$  that solves the value function is called the *value function matrix*.  $A_{ij}$  is the value of having  $w_i$  cake at time  $j$ .  $A_{0j} = 0$  because there is never any value in having  $w_0$  cake, i.e.  $u(w_0) = u(0) = 0$ .

We start at the last time period. Since there is no value in having any cake left over when time runs out, the decision at time  $T$  is obvious: eat the rest of the cake. The amount of utility gained from having  $w_i$  cake at time  $T$  is given by  $u(w_i)$ . So  $A_{iT} = u(w_i)$ . Written in the form of (12.3),

$$A_{iT} = V(0, 0, w_i) = \max_{w_j} (u(w_i - w_j) + \beta V(0, -1, w_j)) = u(w_i). \quad (12.4)$$

This happens because  $V(0, -1, w_j) = 0$ . As mentioned, there is no value in saving cake so this equation is maximized when  $w_j = 0$ . All possible values of  $w_i$  are calculated so that the value of having  $w_i$  cake at time  $T$  is known.

### ACHTUNG!

Given a time interval from  $t = 0$  to  $t = T$  the utility of waiting until time  $T$  to eat  $w_i$  cake is actually  $\beta^T u(w_i)$ . However, through backwards induction, the problem is solved backwards by beginning with  $t = T$  as an isolated state and calculating its value. This is why the value function above is  $V(0, 0, W_i)$  and not  $V(T, T, W_i)$ .

For example, the following matrix results with  $T = 3, N = 4$ , and  $\beta = 0.9$ .

$$\begin{bmatrix} 0 & 0 & 0 & u(0) \\ 0 & 0 & 0 & u(0.25) \\ 0 & 0 & 0 & u(0.5) \\ 0 & 0 & 0 & u(0.75) \\ 0 & 0 & 0 & u(1) \end{bmatrix}.$$

**Problem 12.4: W**

ite a function that accepts a stopping time  $T$ , a number of equal sized pieces  $N$  that divides the cake, a discount factor  $\beta$ , and a utility function  $u(x)$ . Return the value function matrix  $A$  for  $t = T$  (the matrix should have zeros everywhere except the last column). Return a matrix of zeros for the policy matrix  $P$ .

Next, we use the fact that  $A_{jT} = V(0, 0, w_j)$  to evaluate the  $T - 1$  column of the value function matrix,  $A_{i(T-1)}$ , by modifying (12.4) as follows,

$$A_{i(T-1)} = V(0, 1, w_i) = \max_{w_j} (u(w_i - w_j) + \beta V(0, 0, w_j)) = \max_{w_j} (u(w_i - w_j) + \beta A_{jT}). \quad (12.5)$$

Remember that there is a limited set of possibilities for  $w_j$ , and we only need to consider options such that  $w_j \leq w_i$ . Instead of doing these one by one for each  $w_i$ , we can compute the options for each  $w_i$  simultaneously by creating a matrix. This information is stored in an  $(N+1) \times (N+1)$  matrix known as the *current value matrix*, or  $CV^t$ , where the  $(ij)$ th entry is the value of eating  $w_i - w_j$  pieces of cake at time  $t$  and saving  $j$  pieces of cake until the next period. For  $t = T - 1$ ,

$$CV_{ij}^{T-1} = u(w_i - w_j) + \beta A_{jT}. \quad (12.6)$$

The largest entry in the  $i$ th row of  $CV^{T-1}$  is the optimal value that the value function can attain at  $T - 1$ , given that we start with  $w_i$  cake. The maximal values of each row of  $CV^{T-1}$  become the column of the value function matrix,  $A$ , at time  $T - 1$ .

**ACHTUNG!**

The notation  $CV^t$  does not mean raising the matrix to the  $t$ th power; rather, it indicates what time period we are in. All of the  $CV^t$  could be grouped together into a three-dimensional matrix,  $CV$ , that has dimensions  $(N+1) \times (N+1) \times (T+1)$ . Although this is possible, we will not use  $CV$  in this lab, and will instead only consider  $CV^t$  for any given time  $t$ .

The following matrix is  $CV^2$  where  $T = 3$ ,  $\beta = .9$ ,  $N = 4$ , and  $u(x) = \sqrt{x}$ . The maximum value of each row, circled in red, is used in the  $3^{rd}$  column of  $A$ . Remember that  $A$ 's column index begins at 0, so the  $3^{rd}$  column represents  $j = 2$ .

$$CV^2 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0.5 & 0.45 & 0 & 0 & 0 \\ 0.707 & 0.95 & 0.636 & 0 & 0 \\ 0.866 & 1.157 & 1.136 & 0.779 & 0 \\ 1 & 1.316 & 1.343 & 1.279 & 0.9 \end{bmatrix}$$

Now that the column of A corresponding to  $t = T - 1$  has been calculated, we repeat the process for  $T - 2$  and so on until we have calculated each column of A. In summary, at each time step  $t$ , find  $CV^t$  and then set  $A_{it}$  as the maximum value of the  $i$ th row of  $CV^t$ . Generalizing (12.5) and (12.6) shows

$$CV_{ij}^t = u(w_i - w_j) + \beta A_{j(t+1)}. \quad A_{it} = \max_j (CV_{ij}^t). \quad (12.7)$$

The full value function matrix corresponding to the example is below. The maximum value in the value function matrix is the maximum possible utility to be gained.

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0.5 & 0.5 & 0.5 & 0.5 \\ 0.95 & 0.95 & 0.95 & 0.707 \\ 1.355 & 1.355 & 1.157 & 0.866 \\ 1.7195 & 1.562 & 1.343 & 1 \end{bmatrix}.$$

**Figure 12.2: The value function matrix where  $T = 3$ ,  $\beta = .9$ ,  $N = 4$ , and  $u(x) = \sqrt{x}$ . The bottom left entry indicates the highest utility that can be achieved is 1.7195.**

### Problem 12.5: C

Complete your function from Problem 12 so it returns the entire value function matrix. Starting from the next to last column, iterate backwards by

- calculating the current value matrix for time  $t$  using (12.7),
- finding the largest value in each row of the current value matrix, and
- filling in the corresponding column of  $A$  with these values.

(Hint: Use `axis` arguments.)

## Solving for the Optimal Policy

With the value function matrix constructed, the optimization problem is solved in some sense. The value function matrix contains the maximum possible utility to be gained. However, it is not immediately apparent what policy should be followed by only inspecting the value function matrix  $A$ . The  $(N + 1) \times (T + 1)$  policy matrix,  $P$ , is used to find the optimal policy. The  $(ij)$ th entry of the policy matrix indicates how much cake to eat at time  $j$  if we have  $i$  pieces of cake. Like  $A$  and  $CV$ ,  $i$  and  $j$  begin at 0.

The last column of  $P$  is calculated similarly to last column of  $A$ .  $P_{iT} = w_i$ , because at time  $T$  we know that the remainder of the cake should be eaten. Recall that the column of  $A$  corresponding to  $t$  was calculated by the maximum values of  $CV^t$ . The column of  $P$  for time  $t$  is calculated by taking  $w_i - w_j$ ,

where  $j$  is the smallest index corresponding to the maximum value of  $CV^t$ ,

$$P_{it} = w_i - w_j.$$

$$\text{where } j = \{\min\{j\} \mid CV_{ij}^t \geq CV_{ik}^t \forall k \in [0, 1, \dots, N]\}$$

Recall  $CV^2$  in our example with  $T = 3$ ,  $\beta = .9$ ,  $N = 4$ , and  $u(x) = \sqrt{x}$  above.

$$CV^2 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0.5 & 0.45 & 0 & 0 & 0 \\ 0.707 & 0.95 & 0.636 & 0 & 0 \\ 0.866 & 1.157 & 1.136 & 0.779 & 0 \\ 1 & 1.316 & 1.343 & 1.279 & 0.9 \end{bmatrix}$$

To calculate  $P_{12}$ , we look at the second row ( $i = 1$ ) in  $CV^2$ . The maximum, .5, occurs at  $CV_{10}^2$ , so  $j = 0$  and  $P_{12} = w_1 - w_0 = .25 - 0 = .25$ . Similarly,  $P_{42} = w_4 - w_2 = 1 - .5 = .5$ . Continuing in this manner,

$$P = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0.25 & 0.25 & 0.25 & 0.25 \\ 0.25 & 0.25 & 0.25 & 0.5 \\ 0.25 & 0.25 & 0.5 & 0.75 \\ 0.25 & 0.5 & 0.5 & 1 \end{bmatrix}$$

Given that the rows of  $P$  are the slices of cake available and the columns are the time intervals, we find the policy by starting in the bottom left corner,  $P_{N0}$ , where there are  $N$  slices of cake available and  $t = 0$ . This entry tells us what percentage of the  $N$  slices of cake we should eat. In the example, this entry is .25, telling us we should eat 1 slice of cake at  $t = 0$ . Thus, when  $t = 1$  we have  $N - 1$  slices of cake available, since we ate 1 slice of cake. We look at the entry at  $P_{(N-1)1}$ , which has value .25. So we eat 1 slice of cake at  $t = 1$ . We continue this pattern to find the optimal policy  $\mathbf{c} = [.25 \quad .25 \quad .25 \quad .25]$ .

### ACHTUNG!

The optimal policy will not always be a straight diagonal in the example above. For example, if the bottom left corner had value .5, then we should eat 2 pieces of cake instead of 1. Then the next entry we should evaluate would be  $P_{(N-2)1}$  in order to determine the optimal policy.

To verify the optimal policy found with  $P$ , we can use the value function matrix  $A$ . By expanding the entires of  $A$ , we can see that the optimal policy does give the maximum value.

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 \\ \sqrt{0.25} & \sqrt{0.25} & \sqrt{0.25} & \sqrt{0.25} \\ \sqrt{0.25} + \beta\sqrt{0.25} & \sqrt{0.25} + \beta\sqrt{0.25} & \sqrt{0.25} + \beta\sqrt{0.25} & \sqrt{0.5} \\ \sqrt{0.25} + \beta\sqrt{0.25} + \beta^2\sqrt{0.25} & \sqrt{0.25} + \beta\sqrt{0.25} + \beta^2\sqrt{0.25} & \sqrt{0.5} + \beta\sqrt{0.25} & \sqrt{0.75} \\ \cancel{\sqrt{0.25} + \beta\sqrt{0.25} + \beta^2\sqrt{0.25} + \beta^3\sqrt{0.25}} & \sqrt{0.5} + \beta\sqrt{0.25} + \beta^2\sqrt{0.25} & \sqrt{0.5} + \beta\sqrt{0.5} & \sqrt{1} \end{bmatrix}$$

### Problem 12.6: M

Modify your function from Problem 12 to determine the policy matrix. Initialize the matrix as zeros and fill it in starting from the last column at the same time that you calculate the value function matrix.

(Hint: You may find `np.argmax()` useful.)

### Problem 12.7: W

Create a function `find_policy()` that will find the optimal policy for the stopping time  $T$ , a cake of size 1 split into  $N$  pieces, a discount factor  $\beta$ , and the utility function  $u$ .

# 13. Policy Function Iteration

---

**Lab Objective:** *Iterative methods can be powerful ways to solve dynamic optimization problems without computing the exact solution. Often we can iterate very quickly to the true solution, or at least within some  $\epsilon$  error of the solution. These methods are significantly faster than computing the exact solution using dynamic programming. We demonstrate two iterative methods, value iteration (VI) and policy iteration (PI), and use them to solve a deterministic Markov decision process.*

## Dynamic Optimization

---

Many dynamic optimization problem take the form of a *Markov decision process*. A Markov decision process is similar to that of a Markov chain, but rather than determining state movement using only probabilities, state movement is determined based on probabilities, actions, and rewards. They are formulated as follows.

$\mathbb{T}$  is a set of discrete time periods. In this lab,  $\mathbb{T} = 0, 1, \dots, T$ .  $S$  is the set of possible states. The set of allowable actions for each state  $s$  is  $A_s$ .  $s_{t+1} = g(s_t, a_t)$  is a transition function that determines the state  $s_{t+1}$  at time  $t + 1$  based on the previous state  $s_t$  and action  $a_t$ . The reward  $u(s_t, a_t, s_{t+1})$  is the reward for taking action  $a$  while in state  $s$  at time  $t$  and the next state being state  $s_{t+1}$ . The time discount factor  $\beta \in [0, 1]$  determines how much less a reward is worth in the future. Let  $N_{s,a}$  be the set of all possible next states when taking action  $a$  in state  $s$ .  $p(s_t, a_t, s_{t+1})$  is probability of taking action  $a$  at time  $t$  while in state  $s$  and arriving at state  $s_{t+1} \in N_{s,a}$ .

A deterministic Markov process has  $p(s_t, a_t, s_{t+1}) = 1 \forall s, a$ . This means that  $N_{s,a}$  has one element  $\forall s, a$ . A stochastic Markov process has  $p(s_t, a_t, s_{t+1}) \leq 1$ , given that there can be multiple possible next states for taking a given action in a given state.

The dynamic optimization problem is

$$\max_{\mathbf{a}} \sum_{t=0}^T \beta^t u(s_t, a_t) \quad (13.1)$$

$$\text{subject to } s_{t+1} = g(s_t, a_t) \quad \forall t. \quad (13.2)$$

The cake eating problem described in the previous lab follows this format where  $S$  consists of the possible amounts of remaining cake ( $\frac{i}{W}$ ),  $c_t$  is the amount of cake we can eat, and the amount of cake remaining  $s_{t+1} = g(s_t, a_t)$  is  $w_t - c_t$ , where  $w_t$  is the amount of cake we have left and  $c_t$  is the amount of cake we eat at time  $t$ . This is an example of a deterministic Markov process.

For this lab we define a dictionary  $P$  to represent the decision process. This dictionary contains all of the information about the states, actions, probabilities, and rewards. Each dictionary key is a state-action combination and each dictionary value is a list of tuples.

$$P[s][a] = [(p(s, a, \bar{s}), \bar{s}, u(s, a, \bar{s}), is\_terminal), \dots]$$

There is a tuple for each  $\bar{s} \in N_{s,a}$  in the list. The final entry in the tuple, `is_terminal`, indicates if the  $\bar{s}$  is a stopping point.

## Moving on a Grid

---

Now consider an  $N \times N$  grid. Assume that a robot moves around the grid, one space at a time, until it reaches the lower right hand corner and stops. Each square is a state,  $S = \{0, 1, \dots, N^2 - 1\}$ , and the set of actions is  $\{Left, Down, Right, Up\}$ . For this lab,  $Left = 0$ ,  $Down = 1$ ,  $Right = 2$ , and  $Up = 3$ .

### ACHTUNG!

It is important to remember that the actions do not correspond to the states the robot is in after the action. When the robot is in state 0 and takes action 1, he is then in state 2.

$A_s$  is the set of actions that keep the robot on the grid. If the robot is in the top left hand corner, the only allowed actions are *Down* and *Right* so  $A_0 = \{1, 2\}$ . The transition function  $g(s_t, a_t) = s_{t+1}$  can be explicitly defined for each  $s, a$  where  $s_{t+1}$  is the new state after moving.

Let  $N = 2$  and label the squares as displayed below. In this example, we define the reward to be  $-1$  if the robot moves into 2,  $-1$  if the robot moves into 0 from 1, and 1 when it reaches the end, 3. We define the reward function to be  $u(s_t, a_t, s_{t+1}) = u(s_{t+1})$ . Since this is a deterministic model,  $p(s_t, a_t, s_{t+1}) = p(s_{t+1}) = 1, \forall s, a$ .

0	1
2	3

All of this information is encapsulated in  $P$ . We define  $P[s][a]$  for all states and actions, even if they are not possible. This simplifies coding the algorithm but is not necessary.

```

P[0][0] = [(0, 0, 0, False)]   P[2][0] = [(0, 2, -1, False)]
P[0][1] = [(1, 2, -1, False)]  P[2][1] = [(0, 2, -1, False)]
P[0][2] = [(1, 1, 0, False)]   P[2][2] = [(1, 3, 1, True)]
P[0][3] = [(0, 0, 0, False)]  P[2][3] = [(1, 0, 0, False)]
P[1][0] = [(1, 0, -1, False)] P[3][0] = [(0, 0, 0, True)]
P[1][1] = [(1, 3, 1, True)]   P[3][1] = [(0, 0, 0, True)]
P[1][2] = [(0, 0, 0, False)]  P[3][2] = [(0, 0, 0, True)]
P[1][3] = [(0, 0, 0, False)]  P[3][3] = [(0, 0, 1, True)]

```

We define the *value function*  $V(s)$  to be the maximum possible reward of starting in state  $s$ . Then using Bellman's optimality equation,

$$V(s) = \max_{a \in A_s} \left\{ \sum_{\bar{s} \in N_{s,a}} p(\bar{s}) * (u(\bar{s}) + \beta V(\bar{s})) \right\}. \quad (13.3)$$

The summation occurs when it is a stochastic Markov process. For example, if the robot is in the top left corner and moves right, we could have that the probability the robot actually moves right is .5. In this case,  $P[0][2] = [(.5, 1, 0, False), (.5, 2, -1, False)]$ . This will occur later in the lab.

## Value Iteration

---

In the previous lab, we used dynamic programming to solve for the value function. This was a recursive method where we calculated all possible values for each state and time period. *Value iteration* is another algorithm that solves the value function by taking an initial value function and calculating a new value function iteratively. Since we are not calculating all possible values, it is typically faster than dynamic programming.

### Convergence of Value Iteration

---

A function  $f$  that is a contraction mapping has a *fixed point*  $p$  such that  $f(p) = p$ . Blackwell's contraction theorem can be used to show that Bellman's equation is a “fixed point” (it actually acts more like a fixed function in this case) for an operator  $T : L^\infty(X; \mathbb{R}) \rightarrow L^\infty(X; \mathbb{R})$  where  $L^\infty(X; \mathbb{R})$  is the set of all bounded functions:

$$[T(f)](s) = \max_{a \in A_s} \left\{ \sum_{\bar{s} \in N_{s,a}} p(\bar{s}) * (u(\bar{s}) + \beta f(\bar{s})) \right\} \quad (13.4)$$

It can be shown that 13.1 is the fixed “point” of our operator  $T$ . A result of contraction mappings is that there exists a unique solution to 13.4.

$$V_{k+1}(s_i) = [T(V_k)](s_i) = \max_{a \in A_s} \left\{ \sum_{\bar{s} \in N_{s,a}} p(\bar{s}) * (u(\bar{s}) + \beta V_k(\bar{s})) \right\} \quad (13.5)$$

where an initial guess for  $V_0(s)$  is used. As  $k \rightarrow \infty$ , it is guaranteed that  $(V_k(s)) \rightarrow V^*(s)$ . Because of the contraction mapping, if  $V_{k+1}(s) = V_k(s) \forall s$ , we have found the true value function,  $V^*(s)$ .

As an example, let  $V_0 = [0, 0, 0, 0]$  and  $\beta = 1$ , where each entry of  $V_0$  represents the maximum value at

that state. We calculate  $V_1(s)$  from the robot example above.

$$\begin{aligned}
 V_1(0) &= \max_{a \in A_0} \left\{ \sum_{\bar{s} \in N_{s,a}} p(\bar{s}) * (u(\bar{s}) + V_0(\bar{s})) \right\} \\
 &= \max\{p(1) * (u(1) + V_0(1)), p(2) * (u(2) + V_0(2))\} \\
 &= \max\{1(-1+0), 1(0+0)\} \\
 &= \max\{-1, 0\} \\
 &= 0 \\
 V_1(1) &= \max\{p(0) * (u(0) + V_0(0)), p(2) * (u(2) + V_0(2))\} \\
 &= \max\{1(0+0), 1(1+0)\} \\
 &= 1
 \end{aligned}$$

Calculating  $V_1(2)$  and  $V_1(3)$  gives  $V_1 = [0, 1, 1, 0]$ . Repeating the process,  $V_2 = [1, 1, 1, 0]$ , which is the solution. It means that maximum reward the robot can achieve by starting on square  $i$  is  $V_2(i)$ .

### Problem 13.1:

*Write a function called `value_iteration()` that will accept a dictionary  $P$  representing the decision process, the number of states, the number of actions, a discount factor  $\beta \in (0, 1)$ , the tolerance amount  $\epsilon$ , and the maximum number of iterations `maxiter`. Perform value iteration until  $\|V_{k+1} - V_k\| < \epsilon$  or  $k > \text{maxiter}$ . Return the final vector representing  $V^*$  and the number of iterations. Test your code on the example given above.*

## Calculating the Policy

---

While knowing the maximum expected value is helpful, it is usually more important to know the policy that generates the most value. Value Iteration tells the robot what reward he can expect, but not how to get it. The policy vector,  $\pi$ , is found by using the policy function:  $\pi : \mathbb{R} \rightarrow \mathbb{R}$ .  $\pi(s)$  is the action we should take while in state  $s$  to maximize reward. We can modify the Bellman equation using  $V^*(s)$  to find  $\pi$ :

$$\pi(s) = \operatorname{argmax}_{a \in A_s} \left\{ \sum_{\bar{s} \in N_{s,a}} p(\bar{s}) * (u(\bar{s}) + \beta * V^*(\bar{s})) \right\} \quad (13.6)$$

Using value iteration, we found  $V^* = [1, 1, 1, 0]$  in the example above. We find  $\pi(0)$  by looking at actions 1 and 2 (since actions 0 and 3 have probability 0).

$$\begin{aligned}
 \pi(0) &= \operatorname{argmax}_{1,2} \left\{ p(1) * (u(1) + V^*(1)), p(2) * (u(2) + V^*(2)) \right\} \\
 &= \operatorname{argmax}\{1 * (-1+1), 1 * (0+1)\} \\
 &= \operatorname{argmax}\{0, 1\} \\
 &= 2.
 \end{aligned}$$

So when the robot is in state 0, he should take action 2, moving *Right*. This avoids the  $-1$  penalty for moving *Down* into square 2. Similarly,

$$\pi(1) = \operatorname{argmax}_{0,1} \left\{ 1 * (0+1), 1 * (1+1) \right\} = \operatorname{argmax}\{1, 2\} = 1.$$

The policy corresponding to the optimal reward is  $[2, 1, 2, 0]$ . The robot should move to square 3 if possible, avoiding 2 because it has a negative reward. Since 3 is terminal, it does not matter what  $\pi(3)$  is. We set it to 0 for convenience.

### NOTE

Note that  $\pi$  gives the optimal action  $a$  to take at each state  $s$ . It does not give a sequence of actions to take in order to maximize the policy.

### Problem 13.2:

Write a function called `extract_policy()` that will accept a dictionary  $P$  representing the decision process, the number of states, the number of actions, an array representing the value function, and a discount factor  $\beta \in (0, 1)$ , defaulting to 1. Return the policy vector corresponding to  $V^*$ . Test your code on the example with  $\beta = 1$ .

## Policy Iteration

---

For dynamic programming problems, it can be shown that value function iteration converges relative to the discount factor  $\beta$ . As  $\beta \rightarrow 1$ , the number of iterations increases dramatically. As mentioned earlier  $\beta$  is usually close to 1, which means this algorithm can converge slowly. In value iteration, we used an initial guess for the value function,  $V_0$  and used (13.1) to iterate towards the true value function. Once we achieved a good enough approximation for  $V^*$ , we recovered the true policy function  $\pi^*$ . Instead of iterating on our value function, we can instead make an initial guess for the policy function,  $\pi_0$ , and use this to iterate toward the true policy function. We do so by taking advantage of the definition of the value function, where we assume that our policy function yields the most optimal result. This is policy iteration.

That is, given a specific policy function  $\pi_k$ , we can modify (13.1) by assuming that the policy function is the optimal choice. This process, called *policy evaluation*, evaluates the value function for a given policy.

$$V_{k+1}(s) = \max_{a \in [A_s]} \{ \sum_{\bar{s} \in N_{s,a}} (p(\bar{s}) * (u(\bar{s}) + \beta * V_k(\bar{s})) \} = \sum_{\bar{s} \in N_{s,\pi(s)}} (p(\bar{s}) * (u(\bar{s}) + \beta * V_k(\bar{s})) \quad (13.7)$$

The last equality occurs because in state  $s$ , the robot should choose the action that maximizes reward, which is  $\pi(s)$  by definition.

**Problem 13.3:**

Write a function called `compute_policy_v()` that accepts a dictionary  $P$  representing the decision process, the number of states, the number of actions, an array representing a policy, a discount factor  $\beta \in (0, 1)$ , and a tolerance amount  $\epsilon$ . Return the value function corresponding to the policy.

Test your code on the policy vector generated from `extract_policy()` for the example. The result should be the same value function array from `value_iteration()`.

Now that we have the value function for our policy, we can take the value function and find a better policy. Called *policy improvement*, this step is the same method used in value iteration to find the policy.

Given an initial guess for our policy function,  $\pi_0$ , we calculate the corresponding value function using (13.7), and then use (13.6) to improve our policy function. The algorithm for policy function iteration can be summarized as follows:

**Algorithm 3** Policy Iteration

---

```

1: procedure POLICY ITERATION FUNCTION( $P, nS, nA, \beta, tol, \text{maxiter}$ )
2:    $\pi_0 \leftarrow [\pi_0(w_0), \pi_0(w_1), \dots, \pi_0(w_N)]$             $\triangleright$  Common choice is  $\pi_0(w_i) = w_{i-1}$  with  $\pi_0(0) = 0$ 
3:   for  $k = 1, 2, \dots, \text{maxiter}$  do                                 $\triangleright$  Iterate only maxiter times at most.
4:     for  $s \in S$  do                                          $\triangleright$  Policy evaluation
5:        $V_{k+1}(s) = \sum_{\bar{s} \in N_{s,\pi(s)}} (p(\bar{s}) * (u(\bar{s}) + \beta * V_k(\bar{s})))$        $\triangleright$  compute_policy_v.
6:     for  $s \in S$  do                                          $\triangleright$  Policy improvement.
7:        $\pi_{k+1}(s) = \operatorname{argmax}_{a \in A_s} \{\sum_{\bar{s} \in N_{s,a}} p(\bar{s}) * (u(\bar{s}) + \beta * V_{k+1}(\bar{s}))\}$      $\triangleright$  extract_policy.
8:     if  $\|\pi_{k+1} - \pi_k\| < \epsilon$  then                                 $\triangleright$  Stop iterating if the policy doesn't change enough.
9:       break
10:  return  $V_{k+1}, \pi_{k+1}$ 

```

---

**Problem 13.4:**

Write a function called `policy_iteration()` that will accept a dictionary  $P$  representing the decision process, the number of states, the number of actions, a discount factor  $\beta \in (0, 1)$ , the tolerance amount  $\epsilon$ , and the maximum number of iterations `maxiter`. Perform policy iteration until  $\|\pi_{k+1} - \pi_k\| < \epsilon$  or  $k > \text{maxiter}$ . Return the final vector representing  $V_k$ , the optimal policy  $\pi_k$ , and the number of iterations. Test your code on the example given above and compare your answers to the results from ?? and ??.

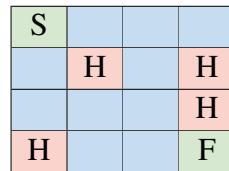
# The Frozen Lake Problem

---

For the rest of the lab, we will be using OpenAi Gym environments. They can be installed using conda or pip.

```
$ pip install gym[all]
```

In the Frozen Lake problem, you and your friends tossed a frisbee onto a mostly frozen lake. The lake is divided into an  $N \times N$  grid where the top left hand corner is the start, the bottom right hand corner is the end, and the other squares are either frozen or holes. To retrieve the frisbee, you must successfully navigate around the melted ice without falling. The possible actions are left, right, up, and down. Since ice is slippery, you won't always move in the intended direction. Hence, this is a stochastic Markov process, i.e.  $p(s_t, a_t, s_{t+1}) < 1$ . If you fall, your reward is 0. If you succeed, your reward is 1. There are two scenarios:  $N = 4$  and  $N = 8$ .



**Figure 13.1: Diagram of the 4x4 scenario. The green  $S$  represents the starting point and the green  $F$  represents the frisbee. Red squares marked  $H$  are holes. Blue squares are pieces of the frozen lake.**

This problem can be found in two environments in OpenAi Gym. To run the  $4 \times 4$  scenario, use `env_name='FrozenLake-v0'`. For the  $8 \times 8$  scenario, use `env_name='FrozenLake8x8-v0'`.

## Using Gym

---

To use gym, we import it and create an environment based on the built-in gym environment. The Frozen-Lake environment has 3 important attributes,  $P$ ,  $nS$ , and  $nA$ .  $P$  is the same dictionary we used in the previous problems.  $nS$  and  $nA$  are the number of states and actions respectively. We can calculate the optimal policy with value iteration or policy iteration using these attributes. Since the ice is slippery, this policy will not always result in a reward of 1.

```
import gym
from gym import wrappers
# Make environment for 4x4 scenario
env_name = 'FrozenLake-v0'
env = gym.make(env_name).env
# Find number of states and actions
number_of_states = env.nS
```

```
number_of_actions = env.nA
```

**Problem 13.5:**

Write a function that runs `value_iteration` and `policy_iteration` on `FrozenLake`. It should accept a boolean `basic_case` defaulting to `True` and an integer `n` defaulting to 1000 that indicates how many times to run the simulation. If `basic_case` is `True`, run the 4x4 scenario. If not, run the 8x8 scenario. Calculate the value function and policy for the environment using both value iteration and policy iteration. Return the policies generated by value iteration and the policy and value function generated by policy iteration. Set the mean total rewards to 0 and return them as well.

The gym environments have built-in functions that allow us to simulate each step of the environment. Before running a scenario in gym, always put it in the starting state by calling `env.reset()`. To simulate moving, call `env.step`.

```
import gym
from gym import wrappers
# Make environment for 4x4 scenario
env_name = 'FrozenLake-v0'
env = gym.make(env_name).env
# Put environment in starting state
obs = env.reset()
# Take a step in the optimal direction and update variables
obs, reward, done, _ = env.step(int(policy[obs]))
```

The `step` function returns four values: observation, reward, done, info. The observation is an environment-specific object representing the observation of the environment. For `FrozenLake`, this is the current state. When we step, or take an action, we get a new observation, or state, as well as the reward for taking that action. If we fall into a hole or reach the frisbee, the simulation is over so we are done. When we are done, the boolean `done` is `True`. The `info` value is a dictionary of diagnostic information. It will not be used in this lab.

**Problem 13.6:**

Write a function `run_simulation()` that takes in the environment `env`, a policy `policy`, a boolean `render`, and a discount factor  $\beta$ . Calculate the total reward for the policy for one simulation using `env.reset` and `env.step()`. Stop the simulation when `done` is `True`. (Hint: When calculating reward, use  $\beta^k$ .)

Modify `frozen_lake()` to call `run_simulation()` for both the value iteration policy and the policy iteration policy  $M$  times. Return the policy generated by value iteration, the mean total reward for the policy generated by value iteration, the value function generated by policy iteration, the policy generated by policy iteration, and the mean total reward for the policy generated by policy iteration.

# 14. Graph Algorithms

---

Youtube! Video of many graph algorithms by Google engineer (6+ hours)



# **Part III**

# **Nonlinear Programming**



# 15. Non-linear Programming (NLP)

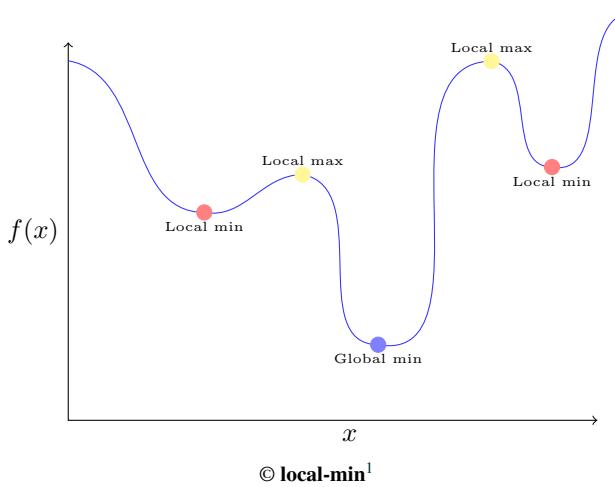
$\min_{x \in \mathbb{R}^n} f(x)$	$\min_{x \in \mathbb{R}^n} f(x)$ $f_i(x) \leq 0 \text{ for } i = 1, \dots, m$
Unconstrained Minimization	Constrained Minimization

- **objective function**  $f: \mathbb{R}^n \rightarrow \mathbb{R}$
- may **maximize**  $f$  by minimizing the function  $g(x) := -f(x)$

## Definition 15.1

The vector  $x^*$  is a

- global minimizer if  $f(x^*) \leq f(x)$  for all  $x \in \mathbb{R}^n$ .
- local minimizer if  $f(x^*) \leq f(x)$  for all  $x$  satisfying  $\|x - x^*\| \leq \varepsilon$  for some  $\varepsilon > 0$ .
- strict local minimizer if  $f(x^*) < f(x)$  for all  $x \neq x^*$  satisfying  $\|x - x^*\| \leq \varepsilon$  for some  $\varepsilon > 0$ .



## Theorem 15.2: L

Let  $S$  be a nonempty set that is closed and bounded. Suppose that  $f: S \rightarrow \mathbb{R}$  is continuous. Then the problem  $\min\{f(x) : x \in S\}$  attains its minimum.

<sup>1</sup>local-min, from local-min. local-min, local-min.

**Definition 15.3: Critical Point**

A critical point is a point  $\bar{x}$  where  $\nabla f(\bar{x}) = 0$ .

**Theorem 15.4**

Suppose that  $f: \mathbb{R}^n \rightarrow \mathbb{R}$  is differentiable. If  $\min\{f(x) : x \in \mathbb{R}^n\}$  has an optimizer  $x^*$ , then  $x^*$  is a critical point of  $f$  (i.e.,  $\nabla f(x^*) = 0$ ).

## 15.1 Convex Sets

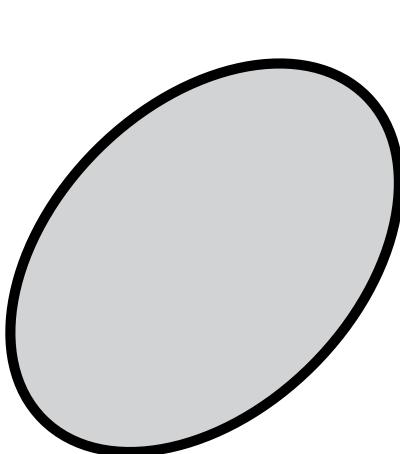
---

**Definition 15.5: Convex Combination**

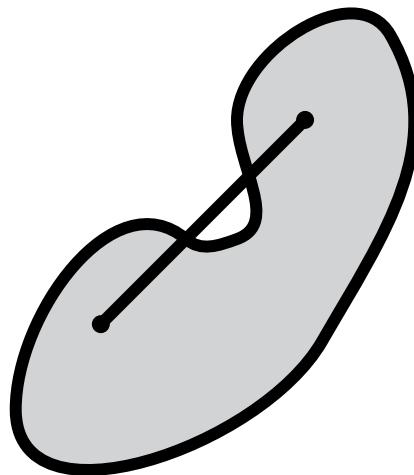
Given two points  $x, y$ , a convex combination is any point  $z$  that lies on the line between  $x$  and  $y$ . Algebraically, a convex combination is any point  $z$  that can be represented as  $z = \lambda x + (1 - \lambda)y$  for some multiplier  $\lambda \in [0, 1]$ .

**Definition 15.6: Convex Set**

A set  $C$  is convex if it contains all convex combinations of points in  $C$ . That is, for any  $x, y \in C$ , it holds that  $\lambda x + (1 - \lambda)y \in C$  for all  $\lambda \in [0, 1]$ .



(a) Convex Set



(b) Non-convex set

**Figure 15.1: Examples of convex and non-convex sets.**

**Definition 15.7: Convex Sets**

set  $S$  is convex if for any two points in  $S$ , the entire line segment between them is also contained in  $S$ . That is, for any  $x, y \in S$

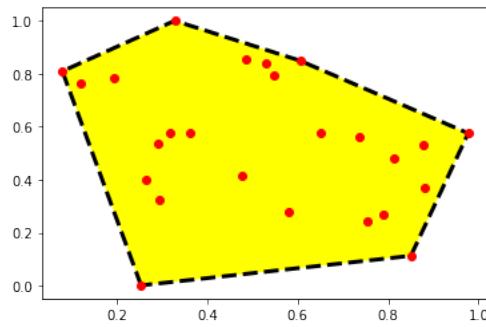
$$\lambda x + (1 - \lambda)y \in S \quad \text{for all } \lambda \in [0, 1].$$

**Examples Convex Sets**

1. Hyperplane  $H = \{x \in \mathbb{R}^n : a^\top x = b\}$
2. Halfspace  $H = \{x \in \mathbb{R}^n : a^\top x \leq b\}$
3. Polyhedron  $P = \{x \in \mathbb{R}^n : Ax \leq b\}$
4. Second Order Cone  $S = \{(x, t) \in \mathbb{R}^n \times \mathbb{R} : \sum_{i=1}^n x_i^2 \leq t^2\}$

**Definition 15.8: Convex Hull**

Let  $S \subseteq \mathbb{R}^n$ . The convex hull  $\text{conv}(S)$  is the smallest convex set containing  $S$ .



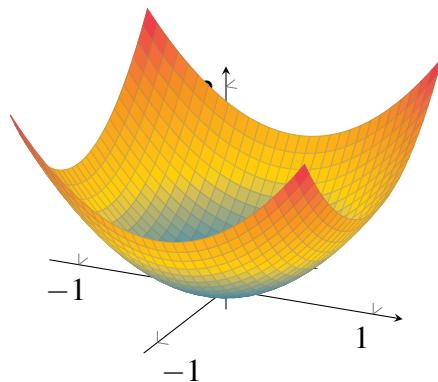
© convex-hull-random<sup>2</sup>

**Theorem 15.9**

[Caratheodory's Theorem] Let  $x \in \text{conv}(S)$  and  $S \subseteq \mathbb{R}^n$ . Then there exist  $x^1, \dots, x^k \in S$  such that  $x \in \text{conv}(\{x^1, \dots, x^k\})$  and  $k \leq n + 1$ .

## 15.2 Convex Functions

Convex functions are "nice" functions that "open up". They represent an extremely important class of functions in optimization and typically can be optimized over efficiently.



**Figure 15.2: Convex Function**  $f(x, y) = x^2 + y^2$ .

### Definition 15.10

*Convex Functions]* A function  $f: \mathbb{R}^n \rightarrow \mathbb{R}$  is convex if for all  $x, y \in \mathbb{R}^n$  and  $\lambda \in [0, 1]$  we have

$$\lambda f(x) + (1 - \lambda)f(y) \geq f(\lambda x + (1 - \lambda)y). \quad (15.1)$$

An equivalent definition of convex function are through the epigraph.

### Definition 15.11

*Epigraph]* The epigraph of  $f$  is the set  $\{(x, y) : y \geq f(x)\}$ . This is the set of all points "above" the function.

### Theorem 15.12

$f(x)$  is a convex function if and only if the epigraph of  $f$  is a convex set.

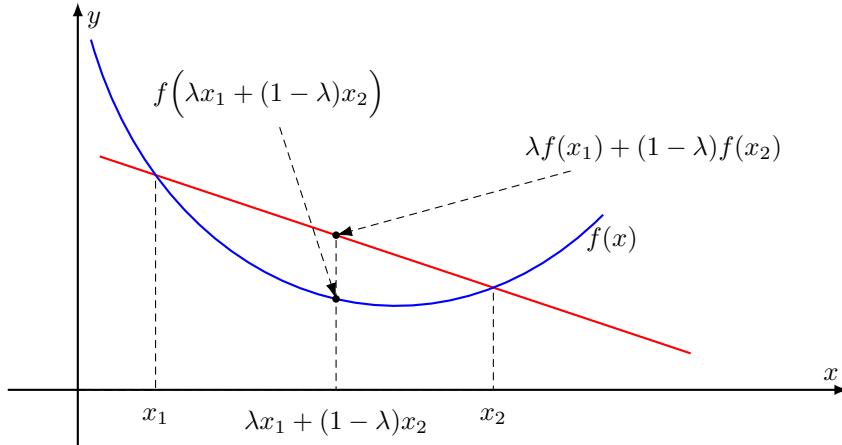
<sup>3</sup>tikz/convexity-definition.pdf,  
tikz/convexity-definition.pdf.

from

tikz/convexity-definition.pdf.

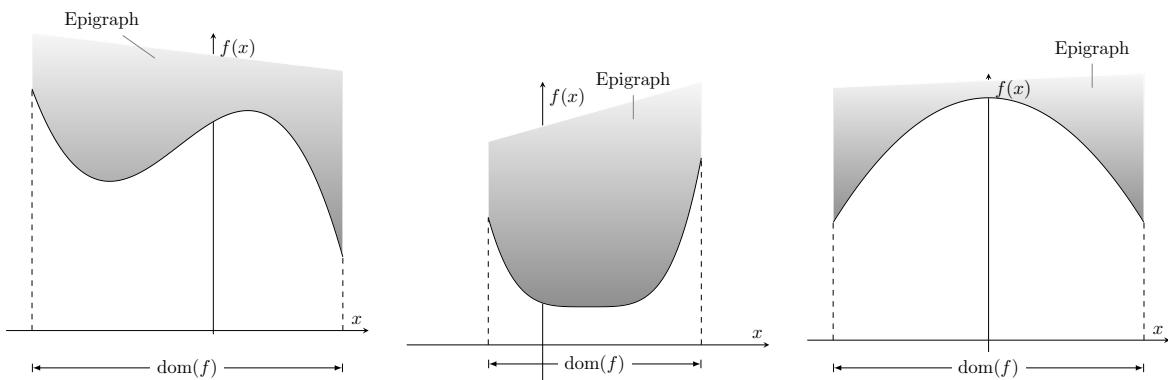
tikz/convexity-definition.pdf,

<sup>3</sup><https://tex.stackexchange.com/questions/394923/how-one-can-draw-a-convex-function>



© tikz/convexity-definition.pdf<sup>3</sup>

**Figure 15.3: Illustration explaining the definition of a convex function.**



© epigraph.pdf<sup>4</sup>

5

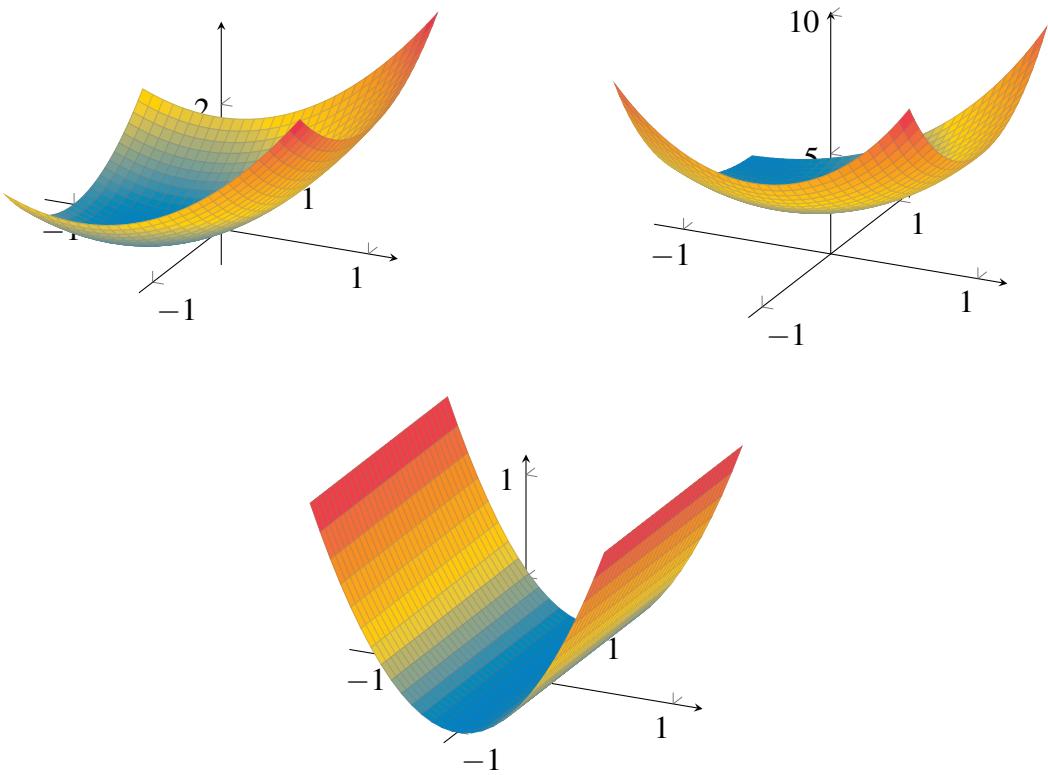
<sup>4</sup>epigraph.pdf, from epigraph.pdf, epigraph.pdf, epigraph.pdf.

<sup>5</sup><https://tex.stackexchange.com/questions/261501/function-epigraph-possibly-using-fillbetween>

**Example 15.13: Examples of Convex functions**

Some examples are

- $f(x) = ax + b$
- $f(x) = x^2$
- $f(x) = x^4$
- $f(x) = |x|$
- $f(x) = e^x$
- $f(x) = -\sqrt{x}$  on the domain  $[0, \infty)$ .
- $f(x) = x^3$  on the domain  $[0, \infty)$ .
- $f(x, y) = \sqrt{x^2 + y^2}$
- $f(x, y) = x^2 + y^2 + x$
- $f(x, y) = e^{x+y}$
- $f(x, y) = e^x + e^y + x^2 + (3x + 4y)^6$



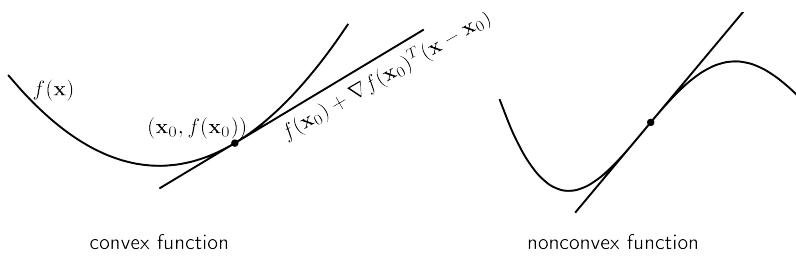
**Figure 15.4: Convex Functions**  $f(x,y) = x^2 + y^2 + x$ ,  $f(x,y) = e^{x+y} + e^{x-y} + e^{-x-y}$ , and  $f(x,y) = x^2$ .

### 15.2.1. Proving Convexity - Characterizations

#### Theorem 15.14

[Convexity: First order characterization - linear underestimates] Suppose that  $f: \mathbb{R}^n \rightarrow \mathbb{R}$  is differentiable. Then  $f$  is convex if and only if for all  $\bar{x} \in \mathbb{R}^n$ , the linear tangent is an underestimator to the function, that is,

$$f(\bar{x}) + (x - \bar{x})^\top \nabla f(\bar{x}) \leq f(x).$$

© first-order-convexity<sup>6</sup>

7

**Theorem 15.15**

*Convexity: Second order characterization - positive curvature]* We give statements for uni-variate functions and multi-variate functions.

- Suppose  $f: \mathbb{R} \rightarrow \mathbb{R}$  is twice differentiable. Then  $f$  is convex if and only if  $f''(x) \geq 0$  for all  $x \in \mathbb{R}$ .
- Suppose  $f: \mathbb{R}^n \rightarrow \mathbb{R}$  is twice differentiable. Then  $f$  is convex if and only if  $\nabla^2 f(x) \preceq 0$  for all  $x \in \mathbb{R}^n$ .

**15.2.2. Proving Convexity - Composition Tricks****Positive Scaling of Convex Function is Convex:**

If  $f$  is convex and  $\alpha > 0$ , then  $\alpha f$  is convex.

**Example:**  $f(x) = e^x$  is convex. Therefore,  $25e^x$  is also convex.

**Sum of Convex Functions is Convex:**

If  $f$  and  $g$  are both convex, then  $f + g$  is also convex.

**Example:**  $f(x) = e^x, g(x) = x^4$  are convex. Therefore,  $e^x + x^4$  is also convex.

<sup>7</sup><https://machinelearningcoban.com/2017/03/12/convexity/>

**Composition with affine function:**

If  $f(x)$  is convex, then  $f(a^\top x + b)$  is also convex.

**Example:**  $f(x) = x^4$  are convex. Therefore,  $(3x + 5y + 10z)^4$  is also convex.

**Pointwise maximum:**

If  $f_i$  are convex for  $i = 1, \dots, t$ , then  $f(x) = \max_{i=1, \dots, t} f_i(x)$  is convex.

**Example:**  $f_1(x) = e^{-x}$ ,  $f_2(x) = e^x$  are convex. Therefore,  $f(x) = \max(e^x, e^{-x})$  is also convex.

**Other compositions:**

Suppose

$$f(x) = h(g(x)).$$

1. If  $g$  is convex,  $h$  is convex and **non-decreasing**, then  $f$  is convex.
2. If  $g$  is concave,  $h$  is convex and **non-increasing**, then  $f$  is convex.

**Example 1:**  $g(x) = x^4$  is convex,  $h(x) = e^x$  is convex and non-decreasing. Therefore,  $f(x) = e^{x^4}$  is also convex.

**Example 2:**  $g(x) = \sqrt{x}$  is concave (on  $[0, \infty)$ ),  $h(x) = e^{-x}$  is convex and non-increasing. Therefore,  $f(x) = e^{-\sqrt{x}}$  is convex on  $x \in [0, \infty)$ .

## 15.3 Convex Optimization Examples

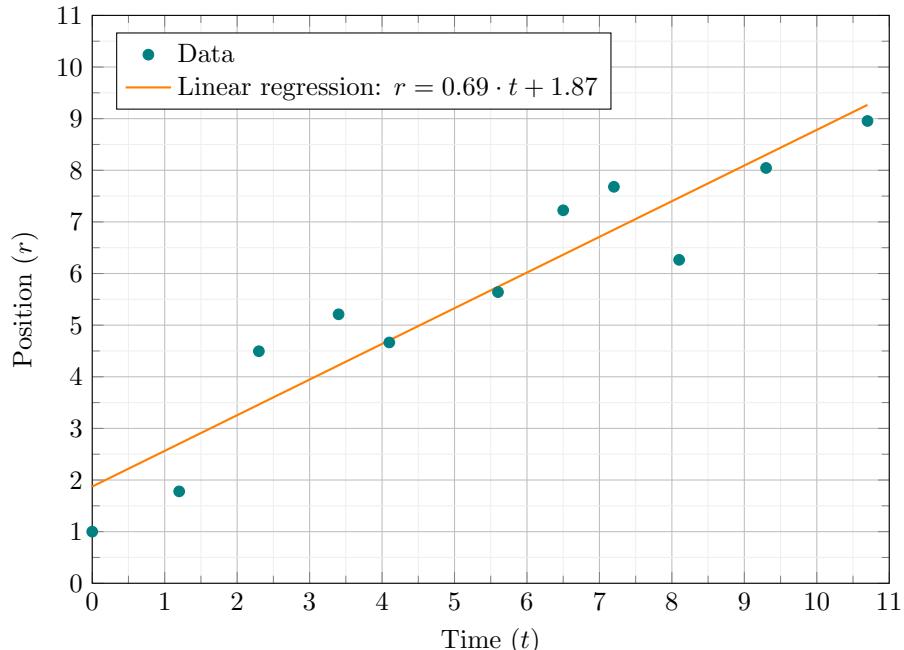
---

### 15.3.1. Unconstrained Optimization: Linear Regression

---

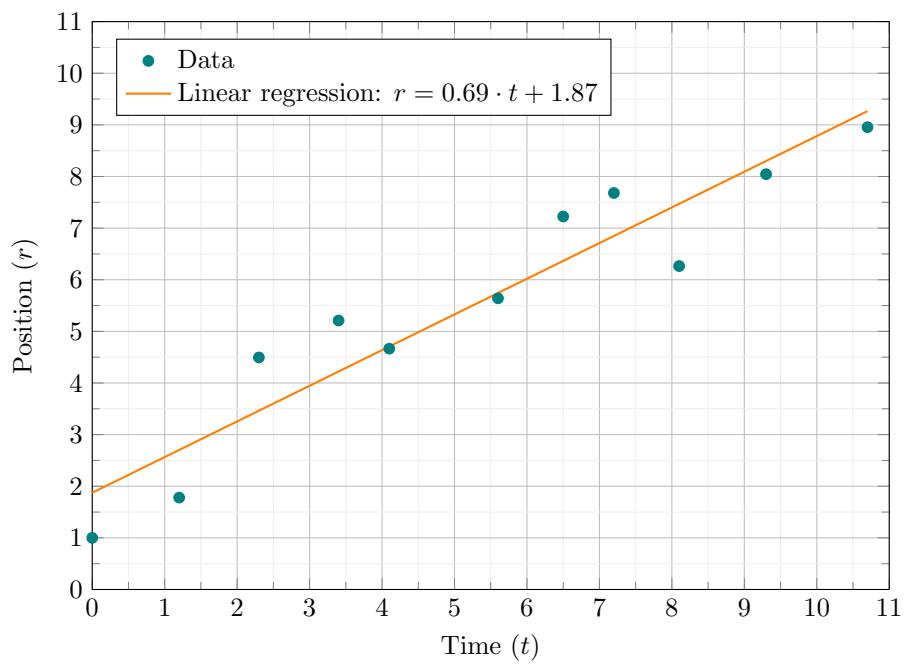
Given data points  $x^1, \dots, x^N \in \mathbb{R}^d$  and out values  $y^i \in \mathbb{R}$ , we want to find a linear function  $y = \beta \cdot x$  that best approximates  $x^i \cdot \beta \approx y^i$ . For example, the data could  $x$  = (time) and the output could be  $y$  = position.

<sup>8</sup>tikz/linear-regression.pdf, from tikz/linear-regression.pdf. tikz/linear-regression.pdf, tikz/linear-regression.pdf.



© tikz/linear-regression.pdf<sup>8</sup>

**Figure 15.5:** Line derived through linear regression.



© LinearRegression.pdf<sup>9</sup>

10

As is standard, we choose the error (or "loss") from each data point as the squared error. Hence, we

---

<sup>10</sup><https://latexdraw.com/linear-regression-in-latex-using-tikz/>

can model this as the optimization problem:

$$\min_{\beta \in \mathbb{R}^d} \sum_{i=1}^N (x^i \cdot \beta - y^i)^2 \quad (15.1)$$

This problem has a nice closed form solution. We will derive this solution, and then in a later section discuss why using this solution might be too slow to compute on large data sets. In particular, the solution comes as a system of linear equations. But when  $N$  is really large, we may not have time to solve this system, so an alternative is to use decent methods, discussed later in this chapter.

### Theorem 15.16: Linear Regression Solution

The solution to (15.1) is

$$\beta = (X^\top X)^{-1} X^\top Y, \quad (15.2)$$

where

$$X = \begin{bmatrix} x^1 \\ \vdots \\ x^N \end{bmatrix} = \begin{bmatrix} x_1^1 & x_2^1 & \dots & x_d^1 \\ \vdots & & & \\ x_1^N & x_2^N & \dots & x_d^N \end{bmatrix}$$

**Proof.** Solve for  $\nabla f(\beta) = 0$ .

To be completed....



<https://www.youtube.com/watch?v=E5RjzSK0fvY>

## 15.4 Machine Learning - SVM

*Support Vector Machine* (SVM) is a tool used in machine learning for classifying data points. For instance, if there are red and black data points, how can we find a good line that separates them? The input data that you are given is as follows:

INPUT:

- $d$ -dimensional data points  $x^1, \dots, x^N$
- 1-dimensional labels  $z^1, \dots, z^N$  (typically we will use  $z_i$  is either 1 or -1)

The output to the problem should be a hyperplane  $w^\top x + b = 0$  that separates the two data types (either exact separation or approximate separation).

OUTPUT:

- A  $d$ -dimensional vector  $w$
- A 1-dimensional value  $b$

Given this output, we can construct a classification function  $f(x)$  as

$$f(x) = \begin{cases} 1 & \text{if } w^\top x + b \geq 0, \\ -1 & \text{if } w^\top x + b < 0. \end{cases} \quad (15.1)$$

There are three versions to consider:

#### 15.4.0.1. Feasible separation

---

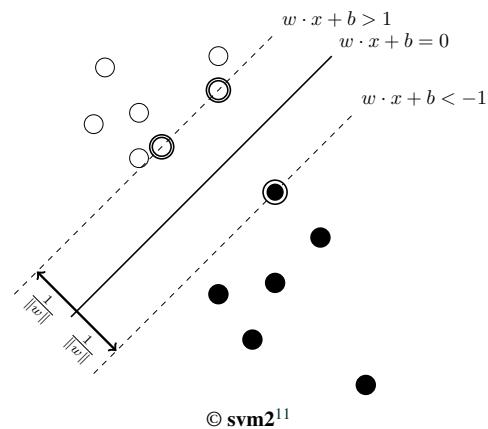
If we only want to a line that separates the data points, we can use the following optimization model.

$$\begin{aligned} \min \quad & 0 \\ \text{such that} \quad & z^i(w^\top x^i + b) \geq 1 \quad \text{for all } i = 1, \dots, N \\ & w \in \mathbb{R}^d \\ & b \in \mathbb{R} \end{aligned}$$

#### 15.4.0.2. SVM

---

We can modify the objective function to find a best separation between the points. This can be done in the following way



<sup>¹¹</sup>svm2, from **svm2**. **svm2**, **svm2**.

$$\begin{aligned}
 \min \quad & \|w\|_2^2 \\
 \text{such that} \quad & z^i(w^\top x^i + b) \geq 1 \quad \text{for all } i = 1, \dots, N \\
 & w \in \mathbb{R}^d \\
 & b \in \mathbb{R}
 \end{aligned}$$

Here,  $\|w\|_2^2 = \sum_{i=1}^d w_i^2 = w_1^2 + \dots + w_d^2$ .

#### 15.4.0.3. Approximate SVM

---

We can modify the objective function and the constraints to allow for approximate separation. This would be the case when you want to ignore outliers that don't fit well with the data set, or when exact SVM is not possible. This is done by changing the constraints to be

$$z^i(w^\top x^i + b) \geq 1 - \delta_i$$

where  $\delta_i \geq 0$  is the error in the constraint for datapoint  $i$ . In order to reduce these errors, we add a penalty term in the objective function that encourages these errors to be small. For this, we can pick some number  $C$  and write the objective as

$$\min \|w\|_2^2 + C \sum_{i=1}^N \delta_i.$$

This creates the following optimization problem

$$\begin{aligned}
 \min \quad & \|w\|_2^2 + C \sum_{i=1}^N \delta_i \\
 \text{such that} \quad & z^i(w^\top x^i + b) \geq 1 - \delta_i \quad \text{for all } i = 1, \dots, N \\
 & w \in \mathbb{R}^d \\
 & b \in \mathbb{R} \\
 & \delta_i \geq 0 \text{ for all } i = 1, \dots, N
 \end{aligned}$$

See information about the scikit-learn module for svm here: <https://scikit-learn.org/stable/modules/svm.html>.

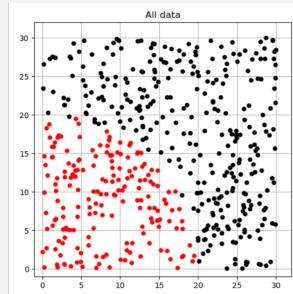
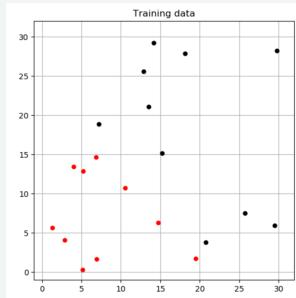
#### 15.4.1. SVM with non-linear separators

---

[https://www.youtube.com/watch?time\\_continue=6&v=N1v0golbjSc](https://www.youtube.com/watch?time_continue=6&v=N1v0golbjSc)

Suppose for instance you are given data  $x^1, \dots, x^N \in \mathbb{R}^2$  (2-dimensional data) and given labels are dependent on the distance from the origin, that is, all data points  $x$  with  $x_1^2 + x_2^2 > r$  are given a label  $+1$  and all data points with  $x_1^2 + x_2^2 \leq r$  are given a label  $-1$ . That is, we want to learn the function

$$f(x) = \begin{cases} 1 & \text{if } x_1^2 + x_2^2 > r, \\ -1 & \text{if } x_1^2 + x_2^2 \leq r. \end{cases} \quad (15.2)$$

**Example 15.17**© svm-nonlinear<sup>12</sup>© svm-nonlinear-training<sup>13</sup>

Here we have a classification problem where the data cannot be separated by a hyperplane. On the left, we have all of the data given to use. On the right, we have a subset of the data that we could try using for training and then test our learned function on the remaining data. As we saw in class, this amount of data was not sufficient to properly classify most of the data.

---

**svm-nonlinear**, from **svm-nonlinear**. **svm-nonlinear**, **svm-nonlinear**.

**svm-nonlinear-training**, from **svm-nonlinear-training**. **svm-nonlinear-training**, **svm-nonlinear-training**.

---

We cannot learn this classifier from the data directly using the hyperplane separation with SVM in the last section. But, if we modify the data set, then we can do this.

For each data point  $x$ , we transform it into a data point  $X$  by adding a third coordinate equal to  $x_1^2 + x_2^2$ . That is

$$x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \rightarrow X = \begin{pmatrix} x_1 \\ x_2 \\ x_1^2 + x_2^2 \end{pmatrix}. \quad (15.3)$$

In this way, we convert the data  $x^1, \dots, x^N$  into data  $X^1, \dots, X^N$  that lives in a higher-dimensional space. But with this new dataset, we can apply the hyperplane separation technique in the last section to properly classify the data.

This can be done with other nonlinear classification functions.

### 15.4.2. Support Vector Machines

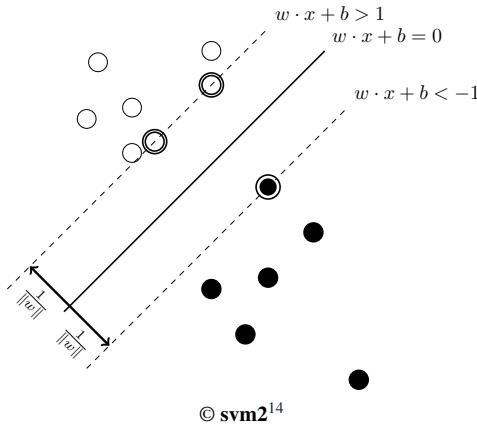
#### Support Vector Machine - Exact Classification:

Given labeled data  $(x^i, y_i)$  for  $i = 1, \dots, N$ , where  $x^i \in \mathbb{R}^d$  and  $y^i \in \{-1, 1\}$ , find a vector  $w \in \mathbb{R}^d$  and a number  $b \in \mathbb{R}$  such that

$$x^i \cdot w + b > 0 \quad \text{if } y^i = 1 \quad (15.4)$$

$$x^i \cdot w + b < 0 \quad \text{if } y^i = -1 \quad (15.5)$$

There may exist many solutions to this problem. Thus, we are interested in the "best" solution. Such a solution will maximize the separation between the two sets of points. To consider an equal margin on either side, we set the right hand sides to 1 and -1 and then compute the margin from the hyperplane. Notice that it is sufficient to use 1 and -1 on the right hand sides since any scaling can happen in  $w$  and  $b$ .



We will show that the margin under this model can be computed as  $\frac{2}{\|w\|}$  where  $\|w\| = \sqrt{w_1^2 + \dots + w_d^2}$ . Hence, maximizing the margin is equivalent to minimizing  $w_1^2 + \dots + w_d^2$ . We arrive at the model

$$\min \sum_{i=1}^d w_i^2 \quad (15.6)$$

$$x^i \cdot w + b \geq 1 \quad \text{if } y^i = 1 \quad (15.7)$$

$$x^i \cdot w + b \leq -1 \quad \text{if } y^i = -1 \quad (15.8)$$

<sup>14</sup>**svm2**, from **svm2**. **svm2**, **svm2**.

Or even more compactly written as

$$\min \sum_{i=1}^d w_i^2 \quad (15.9)$$

$$y^i(x^i \cdot w + b) \geq 1 \quad \text{for } i = 1, \dots, N \quad (15.10)$$

## 15.5 Classification

---

### 15.5.1. Machine Learning

---

[https://www.youtube.com/watch?v=bwZ3Qiuj3i8&list=PL9ooVrP1hQOHUfd-g8GUpKI3hH0wM\\_9Dn&index=13](https://www.youtube.com/watch?v=bwZ3Qiuj3i8&list=PL9ooVrP1hQOHUfd-g8GUpKI3hH0wM_9Dn&index=13)

<https://towardsdatascience.com/solving-a-simple-classification-problem-with-python-fruitful-iteration-10133a2a2a>

### 15.5.2. Neural Networks

---

<https://www.youtube.com/watch?v=bVQUSndD11U>

<https://www.youtube.com/watch?v=8bNIkfRJZpo>

<https://www.youtube.com/watch?v=Dws9Zveu9ug>

## 15.6 Box Volume Optimization in Scipy.Minimize

---

<https://www.youtube.com/watch?v=iSnTtV6b0Gw>

## 15.7 Modeling

---

We will discuss a few models and mention important changes to the models that will make them solvable.

## IMPORTANT TIPS

1. **Find a convex formulation.** It may be that the most obvious model for your problem is actually non-convex. Try to reformulate your model into one that is convex and hence easier for solvers to handle.
2. **Intelligent formulation.** Understanding the problem structure may help reduce the complexity of the problem. Try to deduce something about the solution to the problem that might make the problem easier to solve. This may work for special cases of the problem.
3. **Identify problem type and select solver.** Based on your formulation, identify which type of problem it is and which solver is best to use for that type of problem. For instance,  Gurobi can handle some convex quadratic problems, but not all. Ipopt is a more general solver, but may be slower due to the types of algorithms that it uses.
4. **Add bounds on the variables.** Many solvers perform much better if they are provided bounds to the variables. This is because it reduces the search region where the variables live. Adding good bounds could be the difference in the solver finding an optimal solution and not finding any solution at all.
5. **Warm start.** If you know good possible solutions to the problem (or even just a feasible solution), you can help the solver by telling it this solution. This will reduce the amount of work the solver needs to do. In  Jupyter this can be done by using the command `setvalue(x,[2 4 6])`, where here it sets the value of vector  $x$  to [2 4 6]. It may be necessary to specify values for all variables in the problem for it to start at.
6. **Rescaling variables.** It sometimes is useful to have all variables on the same rough scale. For instance, if minimizing  $x^2 + 100^2y^2$ , it may be useful to define a new variable  $\bar{y} = 100y$  and instead minimize  $x^2 + \bar{y}^2$ .
7. **Provide derivatives.** Working out gradient and hessian information by hand can save the solver time. Particularly when these are sparse (many zeros). These can often be provided directly to the solver.

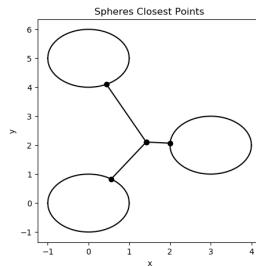
See <http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=4982C26EC5F25564BCC239FD3785E2Ddoi=10.1.1.210.3547&rep=rep1&type=pdf> for many other helpful tips on using Ipopt.

### 15.7.1. Minimum distance to circles

The problem we will consider here is: Given  $n$  circles, find a center point that minimizes the sum of the distances to all of the circles.

#### Minimize distance to circles:

Given circles described by center points  $(a_i, b_i)$  and radius  $r_i$  for  $i = 1, \dots, n$ , find a point  $c = (c_x, c_y)$  that minimizes the sum of the distances to the circles.



© circles-figure<sup>15</sup>

#### Minimize distance to circles - Model attempt #1:

Non-convex

Let  $(x_i, y_i)$  be a point in circle  $i$ . Let  $w_i$  be the distance from  $(x_i, y_i)$  to  $c$ . Then we can model the problem as follows:

$$\begin{array}{ll} \min & \sum_{i=1}^3 w_i & \text{Sum of distances} \\ \text{s.t.} & \sqrt{(x_i - a_i)^2 + (y_i - b_i)^2} = r, & (x_i, y_i) \text{ is in circle } i \\ & \sqrt{(x_i - c_x)^2 + (y_i - c_y)^2} = w_i & w_i \text{ is distance from } (x_i, y_i) \text{ to } c \end{array} \quad (15.1)$$

This model has several issues:

1. If the center  $c$  lies inside one of the circles, then the constraint  $\sqrt{(x_i - a_i)^2 + (y_i - b_i)^2} = r$  may not be valid. This is because the optimal choice for  $(x_i, y_i)$  in this case would be inside the circle, that is, satisfying  $\sqrt{(x_i - a_i)^2 + (y_i - b_i)^2} \leq r$ .
2. This model is **nonconvex**. In particular the equality constraints make the problem nonconvex.

<sup>15</sup>circles-figure, from circles-figure. circles-figure, circles-figure.

Fortunately, we can relax the problem to make it convex and still model the correct solution. In particular, consider the constraint

$$\sqrt{(x_i - c_x)^2 + (y_i - c_y)^2} = w_i.$$

Since we are minimizing  $\sum w_i$ , it is equivalent to have the constraint

$$\sqrt{(x_i - c_x)^2 + (y_i - c_y)^2} \leq w_i.$$

This is equivalent because any optimal solution makes  $w_i$  the smallest it can, and hence will meet that constraint at equality.

What is great about this change, is that it makes the constraint **convex!**. To see this we can write  $f(z) = \|z\|_2^2$ ,  $z = (x_i - c_x, y_i - c_y)$ . Since  $f(z)$  is convex and the transformation into variables  $x_i, c_x, y_i, c_y$  is linear, we have that  $f(x_i - c_x, y_i - c_y)$  is convex. Then since  $-w_i$  is linear, we have that

$$f(x_i - c_x, y_i - c_y) - w_i$$

is a convex function. Thus, the constraint

$$f(x_i - c_x, y_i - c_y) - w_i \leq 0$$

is a convex constraint.

This brings us to our second model.

### Minimize distance to circles - Model attempt #2:

Convex

Let  $(x_i, y_i)$  be a point in circle  $i$ . Let  $w_i$  be the distance from  $(x_i, y_i)$  to  $c$ . Then we can model the problem as follows:

$\min \quad \sum_{i=1}^3 w_i$	Sum of distances	(15.2)
s.t. $\sqrt{(x_i - a_i)^2 + (y_i - b_i)^2} \leq r, \quad i = 1, \dots, n$	$(x_i, y_i)$ is in circle $i$	
$\sqrt{(x_i - c_x)^2 + (y_i - c_y)^2} \leq w_i \quad i = 1, \dots, n$	$w_i$ is distance from $(x_i, y_i)$ to $c$	

Lastly, we would like to make this model better for a solver. For this we will

1. Add bounds on all the variables
2. Change format of non-linear inequalities

**Minimize distance to circles - Model attempt #3:**

Convex

Let  $(x_i, y_i)$  be a point in circle  $i$ . Let  $w_i$  be the distance from  $(x_i, y_i)$  to  $c$ . Then we can model the problem as follows:

$$\begin{aligned}
 \min \quad & \sum_{i=1}^3 w_i && \text{Sum of distances} \\
 \text{s.t.} \quad & (x_i - a_i)^2 + (y_i - b_i)^2 \leq r^2, \quad i = 1, \dots, n && (x_i, y_i) \text{ is in circle } i \\
 & (x_i - c_x)^2 + (y_i - c_y)^2 \leq w_i^2 \quad i = 1, \dots, n && w_i \text{ is distance from } (x_i, y_i) \text{ to } c \\
 & 0 \leq w_i \leq u_i \\
 & a_i - r \leq x_i \leq a_i + r \\
 & b_i - r \leq y_i \leq b_i + r
 \end{aligned} \tag{15.3}$$

**Example: Minimize distance to circles**

Code

Here we minimize the distance of three circles of radius 1 centered at  $(0,0)$ ,  $(3,2)$ , and  $(0,5)$ .

Note: The bounds on the variables here are not chosen optimally.

$$\begin{aligned}
 \min \quad & w_1 + w_2 + w_3 \\
 \text{Subject to} \quad & (x_1 - 0)^2 + (y_1 - 0)^2 \leq 1 \\
 & (x_2 - 3)^2 + (y_2 - 2)^2 \leq 1 \\
 & (x_3 - 0)^2 + (y_3 - 5)^2 \leq 1 \\
 & (x_1 - c_x)^2 + (y_1 - c_y)^2 \leq w_1^2 \\
 & (x_2 - c_x)^2 + (y_2 - c_y)^2 \leq w_2^2 \\
 & (x_3 - c_x)^2 + (y_3 - c_y)^2 \leq w_3^2 \\
 & -1 \leq x_i \leq 10 \quad \forall i \in \{1, 2, 3\} \\
 & -1 \leq y_i \leq 10 \quad \forall i \in \{1, 2, 3\} \\
 & 0 \leq w_i \leq 40 \quad \forall i \in \{1, 2, 3\} \\
 & -1 \leq c_x \leq 10 \\
 & -1 \leq c_y \leq 10
 \end{aligned}$$

## 15.8 Machine Learning

---

There are two main fields of machine learning:

- Supervised Machine Learning,
- Unsupervised Machine Learning.

Supervised machine learning is composed of *Regression* and *Classification*. This area is thought of as being given labeled data that you are then trying to understand the trends of this labeled data.

Unsupervised machine learning is where you are given unlabeled data and then need to decide how to label this data. For instance, how can you optimally partition the people in a room into 5 groups that share the most commonalities?

## 15.9 Machine Learning - Supervised Learning - Regression

---

See the video lecture information.

## 15.10 Machine learning - Supervised Learning - Classification

---

The problem of data *classification* begins with *data* and *labels*. The goal is *classification* of future data based on sample data that you have by constructing a function to understand future data.

**Goal:** *Classification - create a function  $f(x)$  that takes in a data point  $x$  and outputs the correct label.*

These functions can take many forms. In binary classification, the label set is  $\{+1, -1\}$ , and we want to correctly (as often as we can) determine the correct label for a future data point.

There are many ways to determine such a function  $f(x)$ . In the next section, we will learn about SVM that determines the function by computing a hyperplane that separates the data labeled  $+1$  from the data labeled  $-1$ .

Later, we will learn about *neural networks* that describe much more complicated functions.

Another method is to create a *decision tree*. These are typically more interpretable functions (neural networks are often a bit mysterious) and thus sometimes preferred in settings where the classification should be easily understood, such as a medical diagnosis. We will not discuss this method here since it fits less well with the theme of nonlinear programming.

### 15.10.1. Python SGD implementation and video

---

[https://github.com/l1Sourcell/Classifying\\_Data\\_Using\\_a\\_Support\\_Vector\\_Machine/blob/master/support\\_vector\\_machine\\_lesson.ipynb](https://github.com/l1Sourcell/Classifying_Data_Using_a_Support_Vector_Machine/blob/master/support_vector_machine_lesson.ipynb)

# 16. NLP Algorithms

---

## 16.1 Algorithms Introduction

---

We will begin with unconstrained optimization and consider several different algorithms based on what is known about the objective function. In particular, we will consider the cases where we use

- Only function evaluations (also known as *derivative free optimization*),
- Function and gradient evaluations,
- Function, gradient, and hessian evaluations.

We will first look at these algorithms and their convergence rates in the 1-dimensional setting and then extend these results to higher dimensions.

## 16.2 1-Dimensional Algorithms

---

We suppose that we solve the problem

$$\min f(x) \tag{16.1}$$

$$x \in [a, b]. \tag{16.2}$$

That is, we minimize the univariate function  $f(x)$  on the interval  $[l, u]$ .

For example,

$$\min(x^2 - 2)^2 \tag{16.3}$$

$$0 \leq x \leq 10. \tag{16.4}$$

Note, the optimal solution lies at  $x^* = \sqrt{2}$ , which is an irrational number. Since we will consider algorithms using floating point precision, we will look to return a solution  $\bar{x}$  such that  $\|x^* - \bar{x}\| < \varepsilon$  for some small  $\varepsilon > 0$ , for instance,  $\varepsilon = 10^{-6}$ .

### 16.2.1. Golden Search Method - Derivative Free Algorithm

---

<https://www.youtube.com/watch?v=hLm8xfwWYPw>

Suppose that  $f(x)$  is unimodal on the interval  $[a, b]$ , that is, it is a continuous function that has a single minimizer on the interval.

Without any extra information, our best guess for the optimizer is  $\bar{x} = \frac{a+b}{2}$  with a maximum error of  $\epsilon = \frac{b-a}{2}$ . Our goal is to reduce the size of the interval where we know  $x^*$  to be, and hence improve our best guess and the maximum error of our guess.

Now we want to choose points in the interior of the interval to help us decide where the minimizer is. Let  $x_1, x_2$  such that

$$a < x_2 < x_1 < b.$$

Next, we evaluate the function at these four points. Using this information, we would like to argue a smaller interval in which  $x^*$  is contained. In particular, since  $f$  is unimodal, it must hold that

1.  $x^* \in [a, x_2]$  if  $f(x_1) \leq f(x_2)$ ,
2.  $x^* \in [x_1, b]$  if  $f(x_2) < f(x_1)$ ,

After comparing these function values, we can reduce the size of the interval and hence reduce the region where we think  $x^*$  is.

We will now discuss how to chose  $x_1, x_2$  in a way that we can

1. Reuse function evaluations,
2. Have a constant multiplicative reduction in the size of the interval.

We consider the picture:

To determine the best  $d$ , we want to decrease by a constant factor. Hence, we decrease be a factor  $\gamma$ , which we will see is the golden ration (GR). To see this, we assume that  $(b - a) = 1$ , and ask that  $d = \gamma$ . Thus,  $x_1 - a = \gamma$  and  $b - x_2 = \gamma$ . If we are in case 1, then we cut off  $b - x_1 = 1 - \gamma$ . Now, if we iterate and do this again, we will have an initial length of  $\gamma$  and we want to cut off the interval  $x_2 - x_1$  with this being a proportion of  $(1 - \gamma)$  of the remaining length. Hence, the second time we will cut of  $(1 - \gamma)\gamma$ , which we set as the length between  $x_1$  and  $x_2$ .

Considering the geometry, we have

$$\text{length } a \text{ to } x_1 + \text{length } x_2 \text{ to } b = \text{total length} + \text{length } x_2 \text{ to } x_1$$

hence

$$\gamma + \gamma = 1 + (1 - \gamma)\gamma.$$

Simplifying, we have

$$\gamma^2 + \gamma - 1 = 0.$$

Applying the quadratic formula, we see

$$\gamma = \frac{-1 \pm \sqrt{5}}{2}.$$

Since we want  $\gamma > 0$ , we take

$$\gamma = \frac{-1 + \sqrt{5}}{2} \approx 0.618$$

This is exactly the Golden Ratio (or, depending on the definition, the golden ratio minus 1).

### 16.2.1.1. Example:

---

We can conclude that the optimal solution is in  $[1.4, 3.8]$ , so we would guess the midpoint  $\bar{x} = 2.6$  as our approximate solution with a maximum error of  $\epsilon = 1.2$ .

**Convergence Analysis of Golden Search Method:**

After  $t$  steps of the Golden Search Method, the interval in question will be of length

$$(b - a)(GR)^t \approx (b - a)(0.618)^t$$

Hence, by guessing the midpoint, our worst error could be

$$\frac{1}{2}(b - a)(0.618)^t.$$

## 16.2.2. Bisection Method - 1st Order Method (using Derivative)

---

### 16.2.2.1. Minimization Interpretation

---

**Assumptions:**  $f$  is convex, differentiable

We can look for a minimizer of the function  $f(x)$  on the interval  $[a, b]$ .

### 16.2.2.2. Root finding Interpretation

---

Instead of minimizing, we can look for a root of  $f'(x)$ . That is, find  $x$  such that  $f'(x) = 0$ .

**Assumptions:**  $f'(a) < 0 < f'(b)$ , OR,  $f'(b) < 0 < f'(a)$ .  $f'$  is continuous

The goal is to find a root of the function  $f'(x)$  on the interval  $[a, b]$ . If  $f$  is convex, then we know that this root is indeed a global minimizer.

Note that if  $f$  is convex, it only makes sense to have the assumption  $f'(a) < 0 < f'(b)$ .

#### Convergence Analysis of Bisection Method:

After  $t$  steps of the Bisection Method, the interval in question will be of length

$$(b-a) \left(\frac{1}{2}\right)^t.$$

Hence, by guessing the midpoint, our worst error could be

$$\frac{1}{2}(b-a) \left(\frac{1}{2}\right)^t.$$

## 16.2.3. Gradient Descent - 1st Order Method (using Derivative)

---

**Input:**  $f(x)$ ,  $\nabla f(x)$ , initial guess  $x^0$ , learning rate  $\alpha$ , tolerance  $\varepsilon$

**Output:** An approximate solution  $x$

1. Set  $t = 0$
2. While  $\|f(x^t)\|_2 > \varepsilon$ :
  - (a) Set  $x^{t+1} \leftarrow x^t - \alpha \nabla f(x^t)$ .

- (b) Set  $t \leftarrow t + 1$ .
3. Return  $x^t$ .

#### 16.2.4. Newton's Method - 2nd Order Method (using Derivative and Hessian)

---

**Input:**  $f(x)$ ,  $\nabla f(x)$ ,  $\nabla^2 f(x)$ , initial guess  $x^0$ , learning rate  $\alpha$ , tolerance  $\varepsilon$

**Output:** An approximate solution  $x$

1. Set  $t = 0$
2. While  $\|f(x^t)\|_2 > \varepsilon$ :
  - (a) Set  $x^{t+1} \leftarrow x^t - \alpha[\nabla^2 f(x^t)]^{-1}\nabla f(x^t)$ .
  - (b) Set  $t \leftarrow t + 1$ .
3. Return  $x^t$ .

## 16.3 Multi-Variate Unconstrained Optimizaiton

---

We will now use the techniques for 1-Dimensional optimization and extend them to multi-variate case. We will begin with unconstrained versions (or at least, constrained to a large box) and then show how we can apply these techniques to constrained optimization.

#### 16.3.1. Descent Methods - Unconstrained Optimization - Gradient, Newton

---

##### Outline for Descent Method for Unconstrained Optimization:

###### **Input:**

- A function  $f(x)$
- Initial solution  $x^0$
- Method for computing step direction  $d_t$
- Method for computing length  $t$  of step
- Number of iterations  $T$

**Output:**

- A point  $x_T$  (hopefully an approximate minimizer)

**Algorithm**

1. For  $t = 1, \dots, T$ ,

$$\text{set } x_{t+1} = x_t + \alpha_t d_t$$

**16.3.1.1. Choice of  $\alpha_t$** 

There are many different ways to choose the step length  $\alpha_t$ . Some choices have proofs that the algorithm will converge quickly. An easy choice is to have a constant step length  $\alpha_t = \alpha$ , but this may depend on the specific problem.

**16.3.1.2. Choice of  $d_t$  using  $\nabla f(x)$** 

Choice of descent methods using  $\nabla f(x)$  are known as *first order methods*. Here are some choices:

1. **Gradient Descent:**  $d_t = -\nabla f(x_t)$
2. **Nesterov Accelerated Descent:**  $d_t = \mu(x_t - x_{t-1}) - \gamma \nabla f(x_t + \mu(x_t - x_{t-1}))$

Here,  $\mu, \gamma$  are some numbers. The number  $\mu$  is called the momentum.

**16.3.2. Stochastic Gradient Descent - The mother of all algorithms.**

A popular method is called *stochastic gradient descent* (SGD). This has been described as "The mother of all algorithms". This is a method to **approximate the gradient** typically used in machine learning or stochastic programming settings.

**Stochastic Gradient Descent:**

Suppose we want to solve

$$\min_{x \in \mathbb{R}^n} F(x) = \sum_{i=1}^N f_i(x). \quad (16.1)$$

We could use *gradient descent* and have to compute the gradient  $\nabla F(x)$  at each iteration. But! We see that in the **cost to compute the gradient** is roughly  $O(nN)$ , that is, it is very dependent on the number of function  $N$ , and hence each iteration will take time dependent on  $N$ .

**Instead!** Let  $i$  be a uniformly random sample from  $\{1, \dots, N\}$ . Then we will use  $\nabla f_i(x)$  as an approximation of  $\nabla F(x)$ . Although we lose a bit by using a guess of the gradient, this approximation only takes  $O(n)$  time to compute. And in fact, in expectation, we are doing the same thing. That is,

$$N \cdot \mathbb{E}(\nabla f_i(x)) = N \sum_{i=1}^N \frac{1}{N} \nabla f_i(x) = \sum_{i=1}^N \nabla f_i(x) = \nabla \left( \sum_{i=1}^N f_i(x) \right) = \nabla F(x).$$

Hence, the SGD algorithm is:

1. Set  $t = 0$
2. While ... (some stopping criterion)
  - (a) Choose  $i$  uniformly at random in  $\{1, \dots, N\}$ .
  - (b) Set  $d_t = \nabla f_i(x_t)$
  - (c) Set  $x_{t+1} = x_t - \alpha d_t$

There can be many variations on how to decide which functions  $f_i$  to evaluate gradient information on. Above is just one example.

Linear regression is an excellent example of this.

### Example 16.1: Linear Regression with SGD

Given data points  $x^1, \dots, x^N \in \mathbb{R}^d$  and output  $y^1, \dots, y^N \in \mathbb{R}$ , find  $a \in \mathbb{R}^d$  and  $b \in \mathbb{R}$  such that  $a^\top x^i + b \approx y^i$ . This can be written as the optimization problem

$$\min_{a,b} \sum_{i=1}^N g_i(a,b) \quad (16.2)$$

where  $g_i(a,b) = (a^\top x^i + b)^2$ .

Notice that the objective function  $G(a,b) = \sum_{i=1}^N g_i(a,b)$  is a convex quadratic function. The gradient of the objective function is

$$\nabla G(a,b) = \sum_{i=1}^N \nabla g_i(a,b) = \sum_{i=1}^N 2x^i(a^\top x^i + b)$$

Hence, if we want to use gradient descent, we must compute this large sum (think of  $N \approx 10,000$ ). Instead, we can **approximate the gradient!**. Let  $\tilde{\nabla}G(a,b)$  be our approximate gradient. We will compute this by randomly choosing a value  $r \in \{1, \dots, N\}$  (with uniform probability). Then set

$$\tilde{\nabla}G(a,b) = \nabla g_r(a,b).$$

It holds that the expected value is the same as the gradient, that is,

$$\mathbb{E}(\tilde{\nabla}G(a,b)) = G(a,b).$$

Hence, we can make probabilistic arguments that these two will have the same (or similar) convergence properties (in expectation).

#### 16.3.2.1. Choice of $\Delta_k$ using the hessian $\nabla^2 f(x)$

These choices are called *second order methods*

1. **Newton's Method:**  $\Delta_k = -(\nabla^2 f(x_k))^{-1} \nabla f(x_k)$
2. **BFGS (Quasi-Newton):**  $\Delta_k = -(B_k)^{-1} \nabla f(x_k)$

Here

$$\begin{aligned} s_k &= x_{k+1} - x_k \\ y_k &= \nabla f(x_{k+1}) - \nabla f(x_k) \end{aligned}$$

and

$$B_{k+1} = B_k - \frac{(B_k s_k)(B_k s_k)^\top}{s_k^\top B_k s_k} + \frac{y_k y_k^\top}{y_k^\top s_k}.$$

This serves as an approximation of the hessian and can be efficiently computed. Furthermore, the inverse can be easily computed using certain updating rules. This makes for a fast way to approximate the hessian.

## 16.4 Constrained Convex Nonlinear Programming

---

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

$$\begin{aligned} \min \quad & f(x) \\ \text{s.t.} \quad & f_i(x) \leq 0 \quad \text{for } i = 1, \dots, m \\ & x \in \mathbb{R}^d \end{aligned} \tag{16.1}$$

### 16.4.1. Barrier Method

---

**Constrained Convex Programming via Barrier Method:**

We convert (16.1) into the unconstrained minimization problem:

$$\begin{aligned} \min \quad & f(x) - \phi \sum_{i=1}^m \log(-f_i(x)) \\ & x \in \mathbb{R}^d \end{aligned} \tag{16.2}$$

Here  $\phi > 0$  is some number that we choose. As  $\phi \rightarrow 0$ , the optimal solution  $x(\phi)$  to (16.2) tends to the optimal solution of (16.1). That is  $x(\phi) \rightarrow x^*$  as  $\phi \rightarrow 0$ .

**Constrained Convex Programming via Barrier Method - Initial solution:**

Define a variable  $s \in \mathbb{R}$  and add that to the right hand side of the inequalities and then minimize it in the objective function.

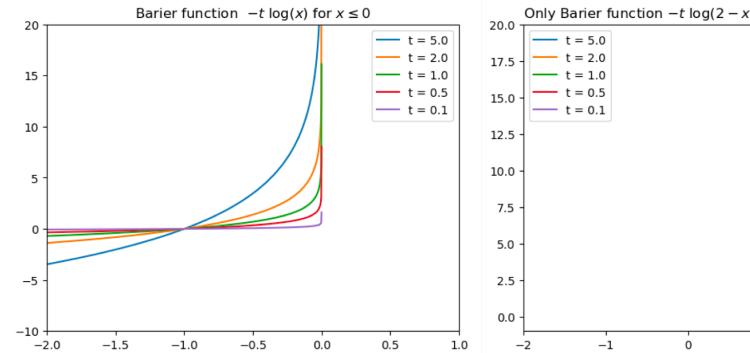
$$\begin{aligned} \min \quad & s \\ \text{s.t.} \quad & f_i(x) \leq s \quad \text{for } i = 1, \dots, m \\ & x \in \mathbb{R}^d, s \in \mathbb{R} \end{aligned} \tag{16.3}$$

Note that this problem is feasible for all  $x$  values since  $s$  can always be made larger. If there exists a solution with  $s \leq 0$ , then we can use the corresponding  $x$  solution as an initial feasible solution. Otherwise, the problem is infeasible.

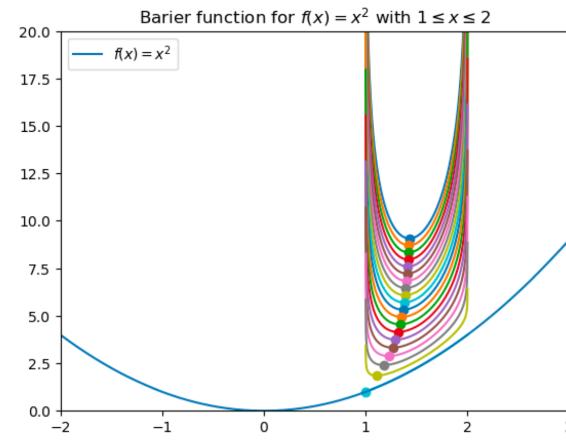
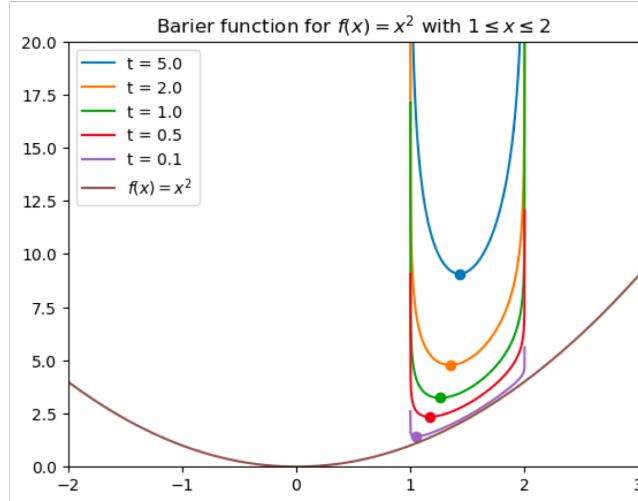
Now, convert this problem into the unconstrained minimization problem:

$$\begin{aligned} \min \quad & f(x) - \phi \sum_{i=1}^m \log(-(f_i(x) - s)) \\ & x \in \mathbb{R}^d, s \in \mathbb{R} \end{aligned} \tag{16.4}$$

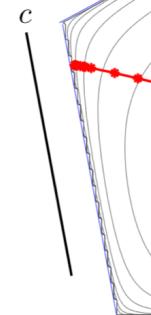
This problem has an easy time of finding an initial feasible solution. For instance, let  $x = 0$ , and then  $s = \max_i f_i(x) + 1$ .



**Images below: the value  $t$  is the value  $\phi$  discussed above**



Minimizing  $c^T x$  subject to



<sup>1</sup>Image taken from unknown source.

# 17. Computational Issues with NLP

---

We mention a few computational issues to consider with nonlinear programs.

## 17.1 Irrational Solutions

---

Consider nonlinear problem (this is even convex)

$$\begin{aligned} \min \quad & -x \\ \text{s.t.} \quad & x^2 \leq 2. \end{aligned} \tag{17.1}$$

The optimal solution is  $x^* = \sqrt{2}$ , which cannot be easily represented. Hence, we would settle for an **approximate solution** such as  $\bar{x} = 1.41421$ , which is feasible since  $\bar{x}^2 \leq 2$ , and it is close to optimal.

## 17.2 Discrete Solutions

---

Consider nonlinear problem (not convex)

$$\begin{aligned} \min \quad & -x \\ \text{s.t.} \quad & x^2 = 2. \end{aligned} \tag{17.1}$$

Just as before, the optimal solution is  $x^* = \sqrt{2}$ , which cannot be easily represented. Furthermore, the only two feasible solutions are  $\sqrt{2}$  and  $-\sqrt{2}$ . Thus, there is no chance to write down a feasible rational approximation.

## 17.3 Convex NLP Harder than LP

---

Convex NLP is typically polynomially solvable. It is a generalization of linear programming.

**Convex Programming:**

*Polynomial time ( $P$ )* (typically)

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

$$\begin{aligned} \min \quad & f(x) \\ \text{s.t.} \quad & f_i(x) \leq 0 \quad \text{for } i = 1, \dots, m \\ & x \in \mathbb{R}^d \end{aligned} \tag{17.1}$$

### Example 17.1: C

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

## 17.4 NLP is harder than IP

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

$$\begin{aligned} \max \quad & c^\top x \\ \text{s.t.} \quad & Ax \leq b \\ & x \in \{0, 1\}^n \\ & x_i(1 - x_i) = 0 \quad \text{for } i = 1, \dots, n \end{aligned} \tag{17.1}$$

## 17.5 Karush-Huhn-Tucker (KKT) Conditions

---

The KKT conditions use the augmented Lagrangian problem to describe sufficient conditions for optimality of a convex program.

### KKT Conditions for Optimality:

Given a convex function  $f(x): \mathbb{R}^d \rightarrow \mathbb{R}$  and convex functions  $g_i(x): \mathbb{R}^d \rightarrow \mathbb{R}$  for  $i = 1, \dots, m$ , the *convex programming* problem is

$$\begin{aligned} \min \quad & f(x) \\ \text{s.t.} \quad & g_i(x) \leq 0 \quad \text{for } i = 1, \dots, m \\ & x \in \mathbb{R}^d \end{aligned} \tag{17.1}$$

Given  $(\bar{x}, \bar{\lambda})$  with  $\bar{x} \in \mathbb{R}^d$  and  $\bar{\lambda} \in \mathbb{R}^m$ , if the KKT conditions hold, then  $\bar{x}$  is optimal for the convex programming problem.

The KKT conditions are

1. (Stationary).

$$-\nabla f(\bar{x}) = \sum_{i=1}^m \bar{\lambda}_i \nabla g_i(\bar{x}) \tag{17.2}$$

2. (Complimentary Slackness).

$$\bar{\lambda}_i g_i(\bar{x}) = 0 \text{ for } i = 1, \dots, m \tag{17.3}$$

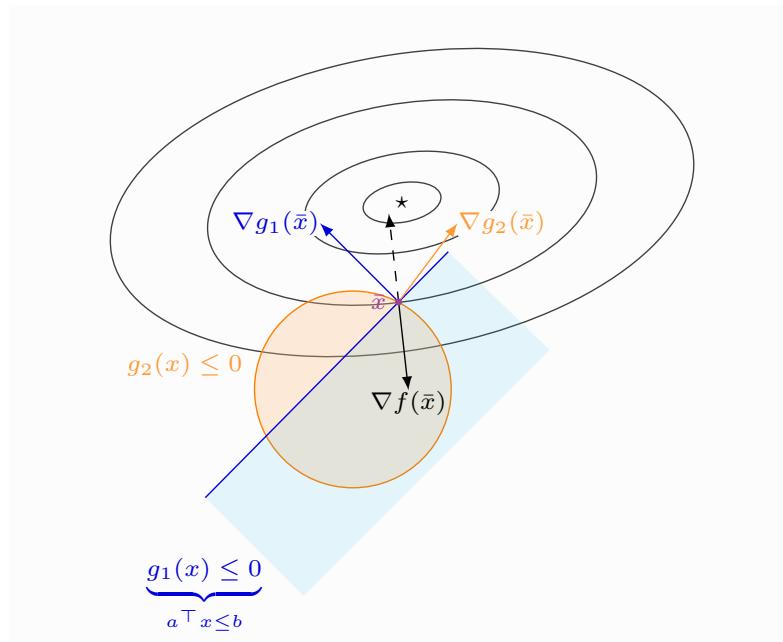
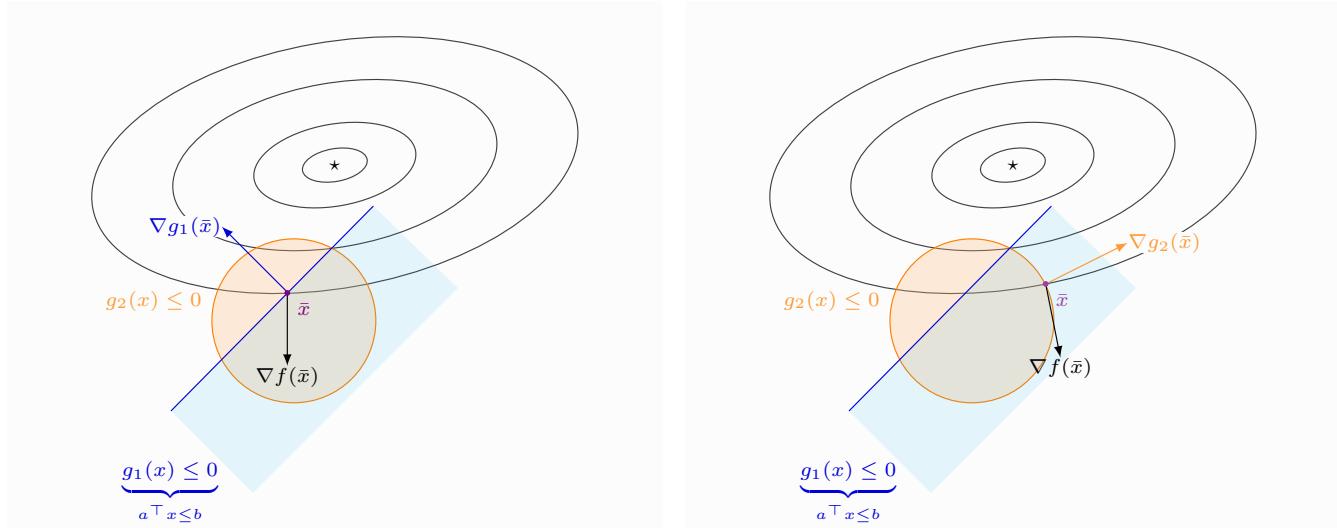
3. (Primal Feasibility).

$$g_i(\bar{x}) \leq 0 \text{ for } i = 1, \dots, m \tag{17.4}$$

4. (Dual Feasibility).

$$\bar{\lambda}_i \geq 0 \text{ for } i = 1, \dots, m \tag{17.5}$$

If certain properties are true of the convex program, then every optimizer has these properties. In particular, this holds for Linear Programming.

© tikz/kkt-optimal<sup>1</sup>© tikz/kkt-non-optimal1<sup>2</sup>


---

tikz/kkt-non-optimal1, from tikz/kkt-non-optimal1.  
tikz/kkt-non-optimal1, tikz/kkt-non-optimal1.

© tikz/kkt-non-optimal2<sup>3</sup>


---

tikz/kkt-non-optimal2, from tikz/kkt-non-optimal2.  
tikz/kkt-non-optimal2, tikz/kkt-non-optimal2.

---

<sup>1</sup>tikz/kkt-optimal, from tikz/kkt-optimal. tikz/kkt-optimal, tikz/kkt-optimal.

## 17.6 Gradient Free Algorithms

---

### 17.6.1. Nelder-Mead

---

[Wikipedia](#)

[Youtube](#)



# 18. Material to add...

---

## 18.0.1. Bisection Method and Newton's Method

---

See section 4 of the following nodes: <http://www.seas.ucla.edu/~vandenbe/133A/133A-notes.pdf>

## 18.1 Gradient Descent

---

Recap Gradient and Directional Derivatives: <https://www.youtube.com/watch?v=tIpKfDc295M>

[https://www.youtube.com/watch?v=\\_-02ze7tf08](https://www.youtube.com/watch?v=_-02ze7tf08)

[https://www.youtube.com/watch?v=N\\_ZRcLheNv0](https://www.youtube.com/watch?v=N_ZRcLheNv0)

<https://www.youtube.com/watch?v=4RBkIJPG6Yo>

Idea of Gradient descent:

<https://youtu.be/IHZwWFHwa-w?t=323>

Vectors:

[https://www.youtube.com/watch?v=fNk\\_zzaMoSs&list=PLZHQBObOWTQDPD3MizzM2xVFitgF8hE\\_ab&index=2&t=0s](https://www.youtube.com/watch?v=fNk_zzaMoSs&list=PLZHQBObOWTQDPD3MizzM2xVFitgF8hE_ab&index=2&t=0s)

## 18.2 Quadratic Programming

---

Quadratic programming is similar to linear programming, but the objective function is quadratic rather than linear. The constraints, if there are any, are still of the same form. Thus,  $G, \mathbf{h}, A$ , and  $\mathbf{b}$  are optional. The formulation that we will use is

$$\begin{aligned} &\text{minimize} && \frac{1}{2} \mathbf{x}^T Q \mathbf{x} + \mathbf{r}^T \mathbf{x} \\ &\text{subject to} && G \mathbf{x} \leq \mathbf{h} \\ & && A \mathbf{x} = \mathbf{b}, \end{aligned}$$

where  $Q$  is a positive semidefinite symmetric matrix. In this formulation, we require again that  $A$  has full row rank and that the block matrix  $[Q \quad G \quad A]^T$  has full column rank.

As an example, consider the quadratic function

$$f(x_1, x_2) = 2x_1^2 + 2x_1x_2 + x_2^2 + x_1 - x_2.$$

There are no constraints, so we only need to initialize the matrix  $Q$  and the vector  $\mathbf{r}$ . To find these, we first rewrite our function to match the formulation given above. If we let

$$Q = \begin{bmatrix} a & b \\ b & c \end{bmatrix}, \quad \mathbf{r} = \begin{bmatrix} d \\ e \end{bmatrix}, \quad \text{and} \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix},$$

then

$$\begin{aligned} \frac{1}{2}\mathbf{x}^T Q \mathbf{x} + \mathbf{r}^T \mathbf{x} &= \frac{1}{2} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} a & b \\ b & c \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} d \\ e \end{bmatrix}^T \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \\ &= \frac{1}{2}ax_1^2 + bx_1x_2 + \frac{1}{2}cx_2^2 + dx_1 + ex_2 \end{aligned}$$

Thus, we see that the proper values to initialize our matrix  $Q$  and vector  $\mathbf{r}$  are:

$$\begin{array}{ll} a = 4 & d = 1 \\ b = 2 & e = -1 \\ c = 2 & \end{array}$$

Now that we have the matrix  $Q$  and vector  $\mathbf{r}$ , we are ready to use the CVXOPT function for quadratic programming `solvers.qp()`.

```
>>> Q = matrix(np.array([[4., 2.], [2., 2.]]))
>>> r = matrix([1., -1.])
>>> sol=solvers.qp(Q, r)
>>> print(sol['x'])
[-1.00e+00]
[ 1.50e+00]
>>> print sol['primal objective']
-1.25
```

### Problem 18.1: Quadratic Minimization

Find the minimizer and minimum of

$$g(x_1, x_2, x_3) = \frac{3}{2}x_1^2 + 2x_1x_2 + x_1x_3 + 2x_2^2 + 2x_2x_3 + \frac{3}{2}x_3^2 + 3x_1 + x_3$$

(Hint: Write the function  $g$  to match the formulation given above before coding.)

**Problem 18.2:  $l_2$  minimization**

The  $l_2$  minimization problem is to

$$\begin{array}{ll} \text{minimize} & \|\mathbf{x}\|_2 \\ \text{subject to} & A\mathbf{x} = \mathbf{b}. \end{array}$$

This problem is equivalent to a quadratic program, since  $\|\mathbf{x}\|_2 = \mathbf{x}^\top \mathbf{x}$ . Write a function that accepts a matrix  $A$  and vector  $\mathbf{b}$  and solves the  $l_2$  minimization problem. Return the minimizer  $\mathbf{x}$  and the primal objective value.

To test your function, use the matrix  $A$  and vector  $\mathbf{b}$  from Problem 3. The minimizer is approximately  $\mathbf{x} = [0.966, 2.169, 0.809, 0.888]^\top$  and the minimum primal objective value is approximately 7.079.



# 19. Least Squares and Computing Eigenvalues

---

**Lab Objective:** Because of its numerical stability and convenient structure, the QR decomposition is the basis of many important and practical algorithms. In this lab we introduce linear least squares problems, tools in Python for computing least squares solutions, and two fundamental algorithms for computing eigenvalue. The QR decomposition makes solving several of these problems quick and numerically stable.

## Least Squares

---

A linear system  $A\mathbf{x} = \mathbf{b}$  is *overdetermined* if it has more equations than unknowns. In this situation, there is no true solution, and  $\mathbf{x}$  can only be approximated.

The *least squares solution* of  $A\mathbf{x} = \mathbf{b}$ , denoted  $\hat{\mathbf{x}}$ , is the “closest” vector to a solution, meaning it minimizes the quantity  $\|A\hat{\mathbf{x}} - \mathbf{b}\|_2$ . In other words,  $\hat{\mathbf{x}}$  is the vector such that  $A\hat{\mathbf{x}}$  is the projection of  $\mathbf{b}$  onto the range of  $A$ , and can be calculated by solving the *normal equations*,<sup>1</sup>

$$A^\top A\hat{\mathbf{x}} = A^\top \mathbf{b}.$$

If  $A$  is full rank (which it usually is in applications) its QR decomposition provides an efficient way to solve the normal equations. Let  $A = \widehat{Q}\widehat{R}$  be the reduced QR decomposition of  $A$ , so  $\widehat{Q}$  is  $m \times n$  with orthonormal columns and  $\widehat{R}$  is  $n \times n$ , invertible, and upper triangular. Since  $\widehat{Q}^\top \widehat{Q} = I$ , and since  $\widehat{R}^\top$  is invertible, the normal equations can be reduced as follows (we omit the hats on  $\widehat{Q}$  and  $\widehat{R}$  for clarity).

$$\begin{aligned} A^\top A\hat{\mathbf{x}} &= A^\top \mathbf{b} \\ (QR)^\top QR\hat{\mathbf{x}} &= (QR)^\top \mathbf{b} \\ R^\top Q^\top QR\hat{\mathbf{x}} &= R^\top Q^\top \mathbf{b} \\ R^\top R\hat{\mathbf{x}} &= R^\top Q^\top \mathbf{b} \\ R\hat{\mathbf{x}} &= Q^\top \mathbf{b} \end{aligned} \tag{19.1}$$

Thus  $\hat{\mathbf{x}}$  is the least squares solution to  $A\mathbf{x} = \mathbf{b}$  if and only if  $R\hat{\mathbf{x}} = Q^\top \mathbf{b}$ . Since  $R$  is upper triangular, this equation can be solved quickly with back substitution.

---

<sup>1</sup>See Volume 1 for a formal derivation of the normal equations.

**Problem 19.1: Solve the normal equations with QR**

Write a function that accepts an  $m \times n$  matrix  $A$  of rank  $n$  and a vector  $\mathbf{b}$  of length  $m$ . Use the reduced QR decomposition of  $A$  and (19.1) to solve the normal equations corresponding to  $\mathbf{Ax} = \mathbf{b}$ .

You may use either SciPy's reduced QR routine (`la.qr()` with `mode="economic"`) or one of your own reduced QR routines. In addition, you may use `la.solve_triangular()`, SciPy's optimized routine for solving triangular systems.

**Fitting a Line**

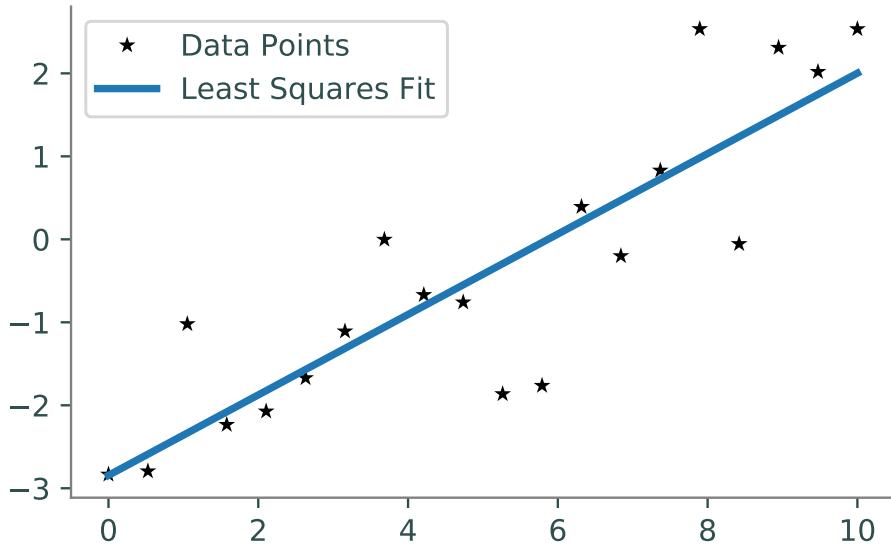
The least squares solution can be used to find the best fit curve of a chosen type to a set of points. Consider the problem of finding the line  $y = ax + b$  that best fits a set of  $m$  points  $\{(x_k, y_k)\}_{k=1}^m$ . Ideally, we seek  $a$  and  $b$  such that  $y_k = ax_k + b$  for all  $k$ . These equations can be simultaneously represented by the linear system

$$\mathbf{Ax} = \begin{bmatrix} x_1 & 1 \\ x_2 & 1 \\ x_3 & 1 \\ \vdots & \vdots \\ x_m & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_m \end{bmatrix} = \mathbf{b}. \quad (19.2)$$

Note that  $A$  has full column rank as long as not all of the  $x_k$  values are the same.

Because this system has two unknowns, it is guaranteed to have a solution if it has two or fewer equations. However, if there are more than two data points, the system is overdetermined if any set of three points is not collinear. We therefore seek a least squares solution, which in this case means finding the slope  $\hat{a}$  and y-intercept  $\hat{b}$  such that the line  $y = \hat{a}x + \hat{b}$  best fits the data.

Figure 19.1 is a typical example of this idea where  $\hat{a} \approx \frac{1}{2}$  and  $\hat{b} \approx -3$ .



**Figure 19.1:** A linear least squares fit.

### Problem 19.2: T

The file `housing.npy` contains the purchase-only housing price index, a measure of how housing prices are changing, for the United States from 2000 to 2010.<sup>a</sup> Each row in the array is a separate measurement; the columns are the year and the price index, in that order. To avoid large numerical computations, the year measurements start at 0 instead of 2000.

Find the least squares line that relates the year to the housing price index (i.e., let year be the  $x$ -axis and index the  $y$ -axis).

1. Construct the matrix  $A$  and the vector  $\mathbf{b}$  described by (19.2).  
(Hint: `np.vstack()`, `np.column_stack()`, and/or `np.ones()` may be helpful.)
2. Use your function from Problem 19 to find the least squares solution.
3. Plot the data points as a scatter plot.
4. Plot the least squares line with the scatter plot.

<sup>a</sup>See <http://www.fhfa.gov/DataTools/Downloads/Pages/House-Price-Index.aspx>.

### NOTE

The least squares problem of fitting a line to a set of points is often called *linear regression*, and the resulting line is called the *linear regression line*. SciPy's specialized tool for linear regression is `scipy.stats.linregress()`. This function takes in an array of  $x$ -coordinates and a corresponding

array of  $y$ -coordinates, and returns the slope and intercept of the regression line, along with a few other statistical measurements.

For example, the following code produces Figure 19.1.

```
>>> import numpy as np
>>> from scipy.stats import linregress

# Generate some random data close to the line y = .5x - 3.
>>> x = np.linspace(0, 10, 20)
>>> y = .5*x - 3 + np.random.randn(20)

# Use linregress() to calculate m and b, as well as the correlation
# coefficient, p-value, and standard error. See the documentation for
# details on each of these extra return values.
>>> a, b, rvalue, pvalue, stderr = linregress(x, y)

>>> plt.plot(x, y, 'k*', label="Data Points")
>>> plt.plot(x, a*x + b, label="Least Squares Fit")
>>> plt.legend(loc="upper left")
>>> plt.show()
```

## Fitting a Polynomial

---

Least squares can also be used to fit a set of data to the best fit polynomial of a specified degree. Let  $\{(x_k, y_k)\}_{k=1}^m$  be the set of  $m$  data points in question. The general form for a polynomial of degree  $n$  is

$$p_n(x) = c_n x^n + c_{n-1} x^{n-1} + \cdots + c_2 x^2 + c_1 x + c_0 = \sum_{i=0}^n c_i x^i.$$

Note that the polynomial is uniquely determined by its  $n+1$  coefficients  $\{c_i\}_{i=0}^n$ . Ideally, then, we seek the set of coefficients  $\{c_i\}_{i=0}^n$  such that

$$y_k = c_n x_k^n + c_{n-1} x_k^{n-1} + \cdots + c_2 x_k^2 + c_1 x_k + c_0$$

for all values of  $k$ . These  $m$  linear equations yield the linear system

$$A\mathbf{x} = \begin{bmatrix} x_1^n & x_1^{n-1} & \cdots & x_1^2 & x_1 & 1 \\ x_2^n & x_2^{n-1} & \cdots & x_2^2 & x_2 & 1 \\ x_3^n & x_3^{n-1} & \cdots & x_3^2 & x_3 & 1 \\ \vdots & \vdots & & \vdots & \vdots & \vdots \\ x_m^n & x_m^{n-1} & \cdots & x_m^2 & x_m & 1 \end{bmatrix} \begin{bmatrix} c_n \\ c_{n-1} \\ \vdots \\ c_2 \\ c_1 \\ c_0 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_m \end{bmatrix} = \mathbf{b}. \quad (19.3)$$

If  $m > n+1$  this system is overdetermined, requiring a least squares solution.

## Working with Polynomials in NumPy

The  $m \times (n + 1)$  matrix  $A$  of (19.3) is called a *Vandermonde matrix*.<sup>2</sup> NumPy's `np.vander()` is a convenient tool for quickly constructing a Vandermonde matrix, given the values  $\{x_k\}_{k=1}^m$  and the number of desired columns.

```
>>> print(np.vander([2, 3, 5], 2))
[[2 1]                      # [[2**1, 2**0]
 [3 1]                      # [3**1, 3**0]
 [5 1]]                     # [5**1, 5**0]

>>> print(np.vander([2, 3, 5, 4], 3))
[[ 4  2  1]                  # [[2**2, 2**1, 2**0]
 [ 9  3  1]                  # [3**2, 3**1, 3**0]
 [25  5  1]                  # [5**2, 5**1, 5**0]
 [16  4  1]]                 # [4**2, 4**1, 4**0]
```

NumPy also has powerful tools for working efficiently with polynomials. The class `np.poly1d` represents a 1-dimensional polynomial. Instances of this class are callable like a function.<sup>3</sup> The constructor accepts the polynomial's coefficients, from largest degree to smallest.

Table 19.1 lists some attributes and methods of the `np.poly1d` class.

Attribute	Description
<code>coeffs</code>	The $n + 1$ coefficients, from greatest degree to least.
<code>order</code>	The polynomial degree ( $n$ ).
<code>roots</code>	The $n - 1$ roots.
Method	Returns
<code>deriv()</code>	The coefficients of the polynomial after being differentiated.
<code>integ()</code>	The coefficients of the polynomial after being integrated (with $c_0 = 0$ ).

**Table 19.1: Attributes and methods of the `np.poly1d` class.**

```
# Create a callable object for the polynomial f(x) = (x-1)(x-2) = x^2 - 3x + 2.
>>> f = np.poly1d([1, -3, 2])
>>> print(f)
2
1 x - 3 x + 2
```

<sup>2</sup>Vandermonde matrices have many special properties and are useful for many applications, including polynomial interpolation and discrete Fourier analysis.

<sup>3</sup>Class instances can be made callable by implementing the `__call__()` magic method.

```
# Evaluate f(x) for several values of x in a single function call.
>>> f([1, 2, 3, 4])
array([0, 0, 2, 6])
```

**Problem 19.3: T**

The data in `housing.npy` is nonlinear, and might be better fit by a polynomial than a line. Write a function that uses (19.3) to calculate the polynomials of degree 3, 6, 9, and 12 that best fit the data. Plot the original data points and each least squares polynomial together in individual subplots.

(Hint: define a separate, refined domain with `np.linspace()` and use this domain to smoothly plot the polynomials.)

Instead of using Problem 19 to solve the normal equations, you may use SciPy's least squares routine, `scipy.linalg.lstsq()`.

```
>>> from scipy import linalg as la

# Define A and b appropriately.

# Solve the normal equations using SciPy's least squares routine.
# The least squares solution is the first of four return values.
>>> x = la.lstsq(A, b)[0]
```

Compare your results to `np.polyfit()`. This function receives an array of  $x$  values, an array of  $y$  values, and an integer for the polynomial degree, and returns the coefficients of the best fit polynomial of that degree.

**ACHTUNG!**

Having more parameters in a least squares model is not always better. For a set of  $m$  points, the best fit polynomial of degree  $m - 1$  *interpolates* the data set, meaning that  $p(x_k) = y_k$  exactly for each  $k$ . In this case there are enough unknowns that the system is no longer overdetermined. However, such polynomials are highly subject to numerical errors and are unlikely to accurately represent true patterns in the data.

Choosing to have too many unknowns in a fitting problem is (fittingly) called *overfitting*, and is an important issue to avoid in any statistical model.

## Fitting a Circle

---

Suppose the set of  $m$  points  $\{(x_k, y_k)\}_{k=1}^m$  are arranged in a nearly circular pattern. The general equation of a circle with radius  $r$  and center  $(c_1, c_2)$  is

$$(x - c_1)^2 + (y - c_2)^2 = r^2. \quad (19.4)$$

The circle is uniquely determined by  $r$ ,  $c_1$ , and  $c_2$ , so these are the parameters that should be solved for in a least squares formulation of the problem. However, (19.4) is not linear in any of these variables.

$$\begin{aligned} (x - c_1)^2 + (y - c_2)^2 &= r^2 \\ x^2 - 2c_1x + c_1^2 + y^2 - 2c_2y + c_2^2 &= r^2 \\ x^2 + y^2 &= 2c_1x + 2c_2y + r^2 - c_1^2 - c_2^2 \end{aligned} \quad (19.5)$$

The quadratic terms  $x^2$  and  $y^2$  are acceptable because the points  $\{(x_k, y_k)\}_{k=1}^m$  are given. To eliminate the nonlinear terms in the unknown parameters  $r$ ,  $c_1$ , and  $c_2$ , define a new variable  $c_3 = r^2 - c_1^2 - c_2^2$ . Then for each point  $(x_k, y_k)$ , (19.5) becomes

$$2c_1x_k + 2c_2y_k + c_3 = x_k^2 + y_k^2.$$

These  $m$  equations are linear in  $c_1$ ,  $c_2$ , and  $c_3$ , and can be written as the linear system

$$\begin{bmatrix} 2x_1 & 2y_1 & 1 \\ 2x_2 & 2y_2 & 1 \\ \vdots & \vdots & \vdots \\ 2x_m & 2y_m & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} x_1^2 + y_1^2 \\ x_2^2 + y_2^2 \\ \vdots \\ x_m^2 + y_m^2 \end{bmatrix}. \quad (19.6)$$

After solving for the least squares solution,  $r$  can be recovered with the relation  $r = \sqrt{c_1^2 + c_2^2 + c_3}$ . Finally, plotting a circle is best done with polar coordinates. Using the same variables as before, the circle can be represented in polar coordinates by setting

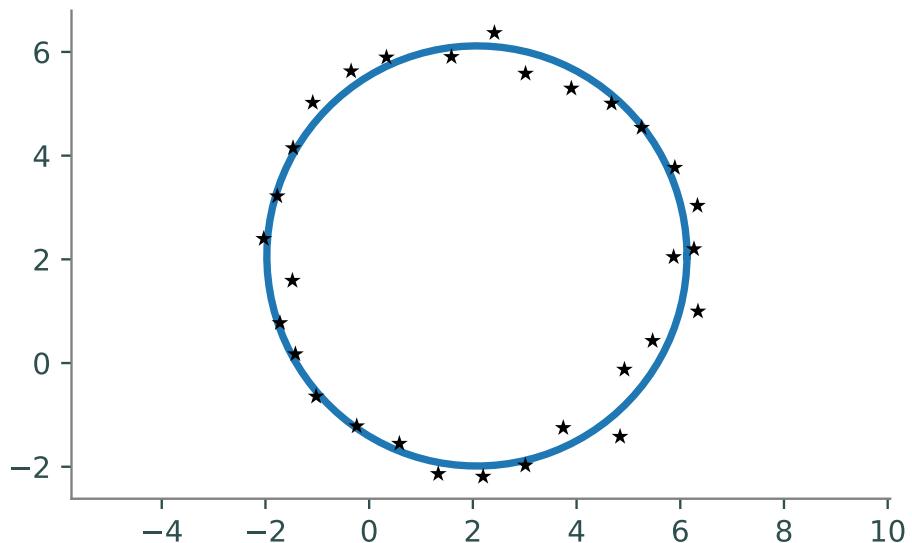
$$x = r\cos(\theta) + c_1, \quad y = r\sin(\theta) + c_2, \quad \theta \in [0, 2\pi]. \quad (19.7)$$

To plot the circle, solve the least squares system for  $c_1$ ,  $c_2$ , and  $r$ , define an array for  $\theta$ , then use (19.7) to calculate the coordinates of the points the circle.

```
# Load some data and construct the matrix A and the vector b.
>>> xk, yk = np.load("circle.npy").T
>>> A = np.column_stack((2*xk, 2*yk, np.ones_like(xk)))
>>> b = xk**2 + yk**2

# Calculate the least squares solution and solve for the radius.
>>> c1, c2, c3 = la.lstsq(A, b)[0]
>>> r = np.sqrt(c1**2 + c2**2 + c3)
```

```
# Plot the circle using polar coordinates.  
>>> theta = np.linspace(0, 2*np.pi, 200)  
>>> x = r*np.cos(theta) + c1  
>>> y = r*np.sin(theta) + c2  
>>> plt.plot(x, y)                      # Plot the circle.  
>>> plt.plot(xk, yk, 'k*')            # Plot the data points.  
>>> plt.axis("equal")
```



**Problem 19.4:**

The general equation for an ellipse is

$$ax^2 + bx + cxy + dy + ey^2 = 1.$$

Write a function that calculates the parameters for the ellipse that best fits the data in the file `ellipse.npy`. Plot the original data points and the ellipse together, using the following function to plot the ellipse.

```
def plot_ellipse(a, b, c, d, e):
    """Plot an ellipse of the form ax^2 + bx + cxy + dy + ey^2 = 1."""
    theta = np.linspace(0, 2*np.pi, 200)
    cos_t, sin_t = np.cos(theta), np.sin(theta)
    A = a*(cos_t**2) + c*cos_t*sin_t + e*(sin_t**2)
    B = b*cos_t + d*sin_t
    r = (-B + np.sqrt(B**2 + 4*A)) / (2*A)
    plt.plot(r*cos_t, r*sin_t, lw=2)
    plt.gca().set_aspect("equal", "datalim")
```

## Computing Eigenvalues

---

The eigenvalues of an  $n \times n$  matrix  $A$  are the roots of its characteristic polynomial  $\det(A - \lambda I)$ . Thus, finding the eigenvalues of  $A$  amounts to computing the roots of a polynomial of degree  $n$ . However, for  $n \geq 5$ , it is provably impossible to find an algebraic closed-form solution to this problem.<sup>4</sup> In addition, numerically computing the roots of a polynomial is a famously ill-conditioned problem, meaning that small changes in the coefficients of the polynomial (brought about by small changes in the entries of  $A$ ) may yield wildly different results. Instead, eigenvalues must be computed with iterative methods.

### The Power Method

---

The *dominant eigenvalue* of the  $n \times n$  matrix  $A$  is the unique eigenvalue of greatest magnitude, if such an eigenvalue exists. The *power method* iteratively computes the dominant eigenvalue of  $A$  and its corresponding eigenvector.

Begin by choosing a vector  $\mathbf{x}_0$  such that  $\|\mathbf{x}_0\|_2 = 1$ , and define

$$\mathbf{x}_{k+1} = \frac{A\mathbf{x}_k}{\|A\mathbf{x}_k\|_2}.$$

---

<sup>4</sup>This result, called *Abel's impossibility theorem*, was first proven by Niels Heinrik Abel in 1824.

If  $A$  has a dominant eigenvalue  $\lambda$ , and if the projection of  $\mathbf{x}_0$  onto the subspace spanned by the eigenvectors corresponding to  $\lambda$  is nonzero, then the sequence of vectors  $(\mathbf{x}_k)_{k=0}^{\infty}$  converges to an eigenvector  $\mathbf{x}$  of  $A$  corresponding to  $\lambda$ .

Since  $\mathbf{x}$  is an eigenvector of  $A$ ,  $A\mathbf{x} = \lambda\mathbf{x}$ . Left multiplying by  $\mathbf{x}^T$  on each side results in  $\mathbf{x}^T A \mathbf{x} = \lambda \mathbf{x}^T \mathbf{x}$ , and hence  $\lambda = \frac{\mathbf{x}^T A \mathbf{x}}{\mathbf{x}^T \mathbf{x}}$ . This ratio is called the *Rayleigh quotient*. However, since each  $\mathbf{x}_k$  is normalized,  $\mathbf{x}^T \mathbf{x} = \|\mathbf{x}\|_2^2 = 1$ , so  $\lambda = \mathbf{x}^T A \mathbf{x}$ .

The entire algorithm is summarized below.

---

**Algorithm 4**


---

```

1: procedure POWERMETHOD( $A$ )
2:    $m, n \leftarrow \text{shape}(A)$                                       $\triangleright A$  is square so  $m = n$ .
3:    $\mathbf{x}_0 \leftarrow \text{random}(n)$                                  $\triangleright$  A random vector of length  $n$ 
4:    $\mathbf{x}_0 \leftarrow \mathbf{x}_0 / \|\mathbf{x}_0\|_2$                           $\triangleright$  Normalize  $\mathbf{x}_0$ 
5:   for  $k = 0, 1, \dots, N - 1$  do
6:      $\mathbf{x}_{k+1} \leftarrow A\mathbf{x}_k$ 
7:      $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_{k+1} / \|\mathbf{x}_{k+1}\|_2$ 
8:   return  $\mathbf{x}_N^T A \mathbf{x}_N, \mathbf{x}_N$ 

```

---

The power method is limited by a few assumptions. First, not all square matrices  $A$  have a dominant eigenvalue. However, the Perron-Frobenius theorem guarantees that if all entries of  $A$  are positive, then  $A$  has a dominant eigenvalue. Second, there is no way to choose an  $\mathbf{x}_0$  that is guaranteed to have a nonzero projection onto the span of the eigenvectors corresponding to  $\lambda$ , though a random  $\mathbf{x}_0$  will almost surely satisfy this condition. Even with these assumptions, a rigorous proof that the power method converges is most convenient with tools from spectral calculus.

### Problem 19.5: Implement the power method.

Write a function that accepts an  $n \times n$  matrix  $A$ , a maximum number of iterations  $N$ , and a stopping tolerance  $\text{tol}$ . Use Algorithm 4 to compute the dominant eigenvalue of  $A$  and a corresponding eigenvector. Continue the loop in step 5 until either  $\|\mathbf{x}_{k+1} - \mathbf{x}_k\|_2$  is less than the tolerance  $\text{tol}$ , or until iterating the maximum number of times  $N$ .

Test your function on square matrices with all positive entries, verifying that  $A\mathbf{x} = \lambda\mathbf{x}$ . Use SciPy's eigenvalue solver, `scipy.linalg.eig()`, to compute all of the eigenvalues and corresponding eigenvectors of  $A$  and check that  $\lambda$  is the dominant eigenvalue of  $A$ .

```
# Construct a random matrix with positive entries.
>>> A = np.random.random((10,10))

# Compute the eigenvalues and eigenvectors of A via SciPy.
>>> eigs, vecs = la.eig(A)

# Get the dominant eigenvalue and eigenvector of A.
# The eigenvector of the kth eigenvalue is the kth column of 'vecs'.
>>> loc = np.argmax(eigs)
>>> lamb, x = eigs[loc], vecs[:,loc]

# Verify that Ax = lambda x.
>>> np.allclose(A @ x, lamb * x)
True
```

## Additional Material

---

### Variations on the Linear Least Squares Problem

---

If  $W$  is an  $n \times n$  is symmetric positive-definite matrix, then the function  $\|\cdot\|_{W^2} : \mathbb{R}^n \rightarrow \mathbb{R}$  given by

$$\|\mathbf{x}\|_{W^2} = \|W\mathbf{x}\|_2 = \sqrt{\mathbf{x}^\top W^\top W \mathbf{x}}$$

defines a norm and is called a *weighted 2-norm*. Given the overdetermined system  $A\mathbf{x} = \mathbf{b}$ , the problem of choosing  $\hat{\mathbf{x}}$  to minimize  $\|A\hat{\mathbf{x}} - \mathbf{b}\|_{W^2}$  is called a *weighted least squares* (WLS) problem. This problem has a slightly different set of normal equations,

$$A^\top W^\top W A \hat{\mathbf{x}} = A^\top W^\top W \mathbf{b}.$$

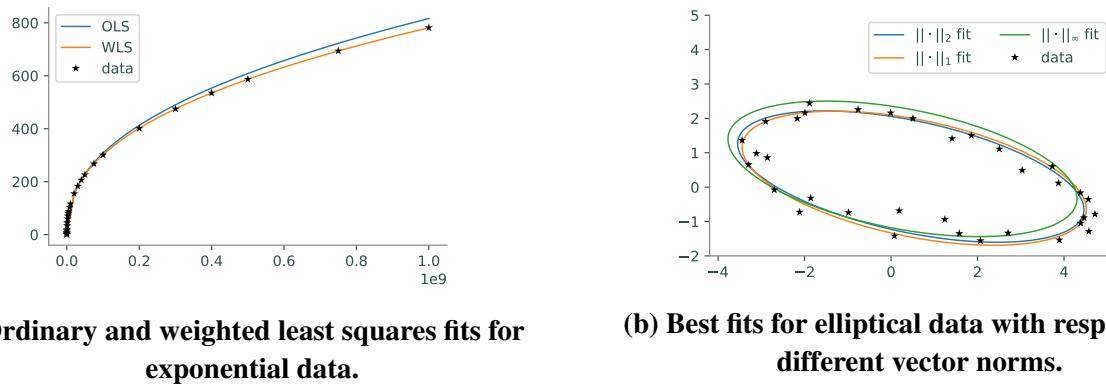
However, letting  $C = WA$  and  $\mathbf{z} = W\mathbf{b}$ , this equation reduces to the usual normal equations,

$$C^\top C \hat{\mathbf{x}} = C^\top \mathbf{z},$$

so a WLS problem can be solved in the same way as an ordinary least squares (OLS) problem.

Weighted least squares is useful when some points in a data set are more important than others. Typically  $W$  is chosen to be a diagonal matrix, and each positive diagonal entry  $W_{i,i}$  indicate how much weight should be given to the  $i$ th data point. For example, Figure 19.2a shows OLS and WLS fits of an exponential curve  $y = ae^{kx}$  to data that gets more sparse as  $x$  increases, where the matrix  $W$  is chosen to give more weight to the data with larger  $x$  values.

Alternatively, the least squares problem can be formulated with other common vector norms, but such problems cannot be solved via the normal equations. For example, minimizing  $\|Ax - b\|_1$  or  $\|Ax - b\|_\infty$  is usually done by solving an equivalent *linear program*, a type of constrained optimization problem. These norms may be better suited to a particular application than the regular 2-norm. Figure 19.2b illustrates how different norms give slightly different results in the context of Problem ??.



**Figure 19.2: Variations on the ordinary least squares problem.**

## The Inverse Power Method

The major drawback of the power method is that it only computes a single eigenvector-eigenvalue pair, and it is always the eigenvalue of largest magnitude. The *inverse power method*, sometimes simply called the *inverse iteration*, is a way of computing an eigenvalue that is closest in magnitude to an initial guess. The key observation is that if  $\lambda$  is an eigenvalue of  $A$ , then  $1/(\lambda - \mu)$  is an eigenvalue of  $(A - \mu I)^{-1}$ , so applying the power method to  $(A - \mu I)^{-1}$  yields the eigenvalue of  $A$  that is closest in magnitude to  $\mu$ .

The inverse power method is more expensive than the regular power method because at each iteration, instead of a matrix-vector multiplication (step 6 of Algorithm 4), a system of the form  $(A - \mu I)\mathbf{x} = \mathbf{b}$  must be solved. To speed this step up, start by taking the LU or QR factorization of  $A - \mu I$  before the loop, then use the factorization and back substitution to solve the system quickly within the loop. For instance, if  $QR = A - \mu I$ , then since  $Q^{-1} = Q^\top$ ,

$$\mathbf{b} = (A - \mu I)\mathbf{x} = QR\mathbf{x} \Leftrightarrow R\mathbf{x} = Q^\top \mathbf{b},$$

which is a triangular system. This version of the algorithm is described below.

---

**Algorithm 5**

---

```

1: procedure INVERSEPOWERMETHOD( $A, \mu$ )
2:    $m, n \leftarrow \text{shape}(A)$ 
3:    $\mathbf{x}_0 \leftarrow \text{random}(n)$ 
4:    $\mathbf{x}_0 \leftarrow \mathbf{x}_0 / \|\mathbf{x}_0\|$ 
5:    $Q, R \leftarrow A - \mu I$                                  $\triangleright$  Factor  $A - \mu I$  with la.qr().
6:   for  $k = 0, 1, 2, \dots, N - 1$  do
7:     Solve  $R\mathbf{x}_{k+1} = Q^\top \mathbf{x}_k$                    $\triangleright$  Use la.solve_triangular().
8:      $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_{k+1} / \|\mathbf{x}_{k+1}\|$ 
9:   return  $\mathbf{x}_N^\top A \mathbf{x}_N, \mathbf{x}_N$ 

```

---

It is worth noting that the QR algorithm can be improved with a similar technique: instead of computing the QR factorization of  $A_k$ , factor the shifted matrix  $A_k - \mu_k I$ , where  $\mu_k$  is a guess for an eigenvalue of  $A$ , and unshift the recombined factorization accordingly. That is, compute

$$\begin{aligned} Q_k R_k &= A_k - \mu_k I, \\ A_{k+1} &= R_k Q_k + \mu_k I. \end{aligned}$$

This technique yields the *single-shift QR algorithm*. Another variant, the *practical QR algorithm*, uses intelligent shifts and recursively operates on smaller blocks of  $A_{k+1}$  where possible. See [[quarteroni2010numerical](#), [Trefethen1997](#)] for further discussion.



# 20. Newton's Method

---

**Lab Objective:** *Newton's method, the classical method for finding the zeros of a function, is one of the most important algorithms of all time. In this lab we implement Newton's method in arbitrary dimensions and use it to solve a few interesting problems. We also explore in some detail the convergence (or lack of convergence) of the method under various circumstances.*

## Iterative Methods

---

An *iterative method* is an algorithm that must be applied repeatedly to obtain a result. The general idea behind any iterative method is to make an initial guess at the solution to a problem, apply a few easy computations to better approximate the solution, use that approximation as the new initial guess, and repeat until done. More precisely, let  $F$  be some function used to approximate the solution to a problem. Starting with an initial guess of  $x_0$ , compute

$$x_{k+1} = F(x_k) \tag{20.1}$$

for successive values of  $k$  to generate a sequence  $(x_k)_{k=0}^{\infty}$  that hopefully converges to the true solution. If the terms of the sequence are vectors, they are denoted by  $\mathbf{x}_k$ .

In the best case, the iteration converges to the true solution  $x$ , written  $\lim_{k \rightarrow \infty} x_k = x$  or  $x_k \rightarrow x$ . In the worst case, the iteration continues forever without approaching the solution. In practice, iterative methods require carefully chosen *stopping criteria* to guarantee that the algorithm terminates at some point. The general approach is to continue iterating until the difference between two consecutive approximations is sufficiently small, and to iterate no more than a specific number of times. That is, choose a very small  $\varepsilon > 0$  and an integer  $N \in \mathbb{N}$ , and update the approximation using (20.1) until either

$$|x_k - x_{k-1}| < \varepsilon \quad \text{or} \quad k > N. \tag{20.2}$$

The choices for  $\varepsilon$  and  $N$  are significant: a “large”  $\varepsilon$  (such as  $10^{-6}$ ) produces a less accurate result than a “small”  $\varepsilon$  (such  $10^{-16}$ ), but demands less computations; a small  $N$  (10) also potentially lowers accuracy, but detects and halts nonconvergent iterations sooner than a large  $N$  (10,000). In code,  $\varepsilon$  and  $N$  are often named `tol` and `maxiter`, respectively (or similar).

While there are many ways to structure the code for an iterative method, probably the cleanest way is to combine a `for` loop with a `break` statement. As a very simple example, let  $F(x) = \frac{x}{2}$ . This method converges to  $x = 0$  independent of starting point.

```
>>> F = lambda x: x / 2
>>> x0, tol, maxiter = 10, 1e-9, 8
```

```
>>> for k in range(maxiter):           # Iterate at most N times.
...     print(x0, end=' ')
...     x1 = F(x0)                     # Compute the next iteration.
...     if abs(x1 - x0) < tol:         # Check for convergence.
...         break                       # Upon convergence, stop iterating.
...     x0 = x1                         # Otherwise, continue iterating.
...
10  5.0  2.5  1.25  0.625  0.3125  0.15625  0.078125
```

In this example, the algorithm terminates after  $N = 8$  iterations (the maximum number of allowed iterations) because the tolerance condition  $|x_k - x_{k-1}| < 10^{-9}$  is not met fast enough. If  $N$  had been larger (say 40), the iteration would have quit early due to the tolerance condition.

## Newton's Method in One Dimension

---

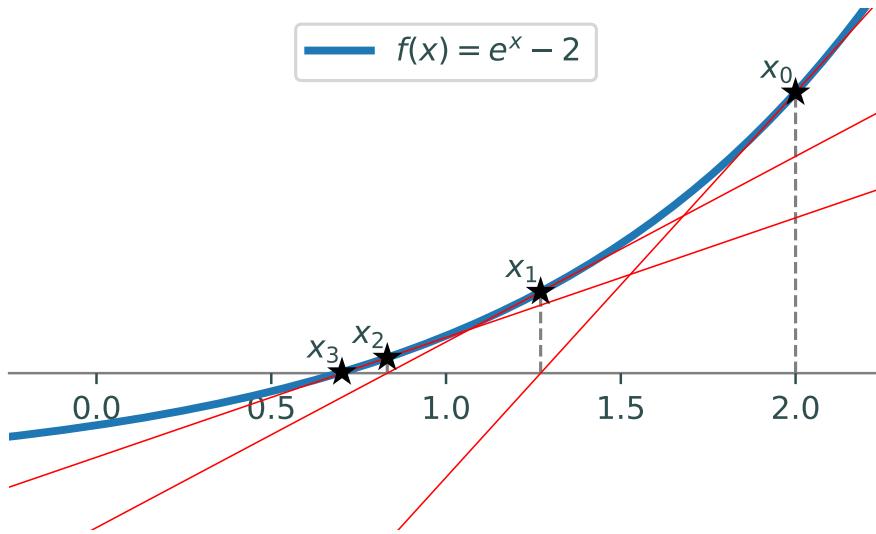
*Newton's method* is an iterative method for finding the zeros of a function. That is, if  $f : \mathbb{R} \rightarrow \mathbb{R}$ , the method attempts to find a  $\bar{x}$  such that  $f(\bar{x}) = 0$ . Beginning with an initial guess  $x_0$ , calculate successive approximations for  $\bar{x}$  with the recursive sequence

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}. \quad (20.3)$$

The sequence converges to the zero  $\bar{x}$  of  $f$  if three conditions hold:

1.  $f$  and  $f'$  exist and are continuous,
2.  $f'(\bar{x}) \neq 0$ , and
3.  $x_0$  is “sufficiently close” to  $\bar{x}$ .

In applications, the first two conditions usually hold. If  $\bar{x}$  and  $x_0$  are not “sufficiently close,” Newton's method may converge very slowly, or it may not converge at all. However, when all three conditions hold, Newton's method converges quadratically, meaning that the maximum error is squared at every iteration. This is very quick convergence, making Newton's method as powerful as it is simple.



**Figure 20.1:** Newton’s method approximates the zero of a function (blue) by choosing as the next approximation the  $x$ -intercept of the tangent line (red) that goes through the point  $(x_k, f(x_k))$ . In this example,  $f(x) = e^x - 2$ , which has a zero at  $\bar{x} = \log(2)$ . Setting  $x_0 = 2$  and using (20.3) to iterate, we have  $x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} = 2 - \frac{e^2 - 2}{e^2} \approx 1.2707$ . Similarly,  $x_2 \approx 0.8320$ ,  $x_3 \approx .7024$ , and  $x_4 \approx 0.6932$ . After only a few iterations, the zero  $\log(2) \approx 0.6931$  is already computed to several digits of accuracy.

### Problem 20.1:

Write a function that accepts a function  $f$ , an initial guess  $x_0$ , the derivative  $f'$ , a stopping tolerance defaulting to  $10^{-5}$ , and a maximum number of iterations defaulting to 15. Use Newton’s method as described in (20.3) to compute a zero  $\bar{x}$  of  $f$ . Terminate the algorithm when  $|x_k - x_{k-1}|$  is less than the stopping tolerance or after iterating the maximum number of allowed times. Return the last computed approximation to  $\bar{x}$ , a boolean value indicating whether or not the algorithm converged, and the number of iterations completed.

Test your function against functions like  $f(x) = e^x - 2$  (see Figure 20.1) or  $f(x) = x^4 - 3$ . Check that the computed zero  $\bar{x}$  satisfies  $f(\bar{x}) \approx 0$ . Also consider comparing your function to `scipy.optimize.newton()`, which accepts similar arguments.

### NOTE

Newton’s method can be used to find zeros of functions that are hard to solve for analytically. For example, the function  $f(x) = \frac{\sin(x)}{x} - x$  is not continuous on any interval containing 0, but it can be made continuous by defining  $f(0) = 1$ . Newton’s method can then be used to compute the zeros of this function.

**Problem 20.2:**

Suppose that an amount of  $P_1$  dollars is put into an account at the beginning of years  $1, 2, \dots, N_1$  and that the account accumulates interest at a fractional rate  $r$  (so  $r = .05$  corresponds to 5% interest). In addition, at the beginning of years  $N_1 + 1, N_1 + 2, \dots, N_1 + N_2$ , an amount of  $P_2$  dollars is withdrawn from the account and that the account balance is exactly zero after the withdrawal at year  $N_1 + N_2$ . Then the variables satisfy

$$P_1[(1+r)^{N_1} - 1] = P_2[1 - (1+r)^{-N_2}].$$

Write a function that, given  $N_1, N_2, P_1$ , and  $P_2$ , uses Newton's method to determine  $r$ . For the initial guess, use  $r_0 = 0.1$ .

(Hint: Construct  $f(r)$  such that when  $f(r) = 0$ , the equation is satisfied. Also compute  $f'(r)$ .) To test your function, if  $N_1 = 30, N_2 = 20, P_1 = 2000$ , and  $P_2 = 8000$ , then  $r \approx 0.03878$ . (From Atkinson, page 118).

## Backtracking

Newton's method may not converge for a variety of reasons. One potential problem occurs when the step from  $x_k$  to  $x_{k+1}$  is so large that the zero is stepped over completely. *Backtracking* is a strategy that combats the problem of overstepping by moving only a fraction of the full step from  $x_k$  to  $x_{k+1}$ . This suggests a slight modification to (20.3),

$$x_{k+1} = x_k - \alpha \frac{f(x_k)}{f'(x_k)}, \quad \alpha \in (0, 1]. \quad (20.4)$$

Note that setting  $\alpha = 1$  results in the exact same method defined in (20.3), but for  $\alpha \in (0, 1)$ , only a fraction of the step is taken at each iteration.

**Problem 20.3: M**

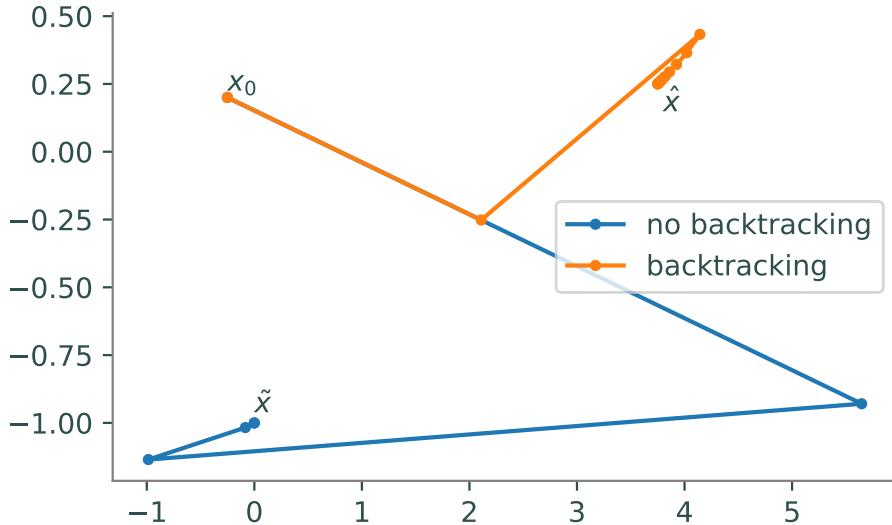
Modify your function from Problem ?? so that it accepts a parameter  $\alpha$  that defaults to 1. Incorporate (20.4) to allow for backtracking.

To test your modified function, consider  $f(x) = x^{1/3}$ . The command `x**(1/3.)` fails when  $x$  is negative, so the function can be defined with NumPy as follows.

```
import numpy as np
f = lambda x: np.sign(x) * np.power(np.abs(x), 1./3)
```

With  $x_0 = .01$  and  $\alpha = 1$ , the iteration should **not** converge. However, setting  $\alpha = .4$ , the iteration should converge to a zero that is close to 0.

The backtracking constant  $\alpha$  is significant, as it can result in faster convergence or convergence to a different zero (see Figure 20.2). However, it is not immediately obvious how to choose an optimal value for  $\alpha$ .



**Figure 20.2:** Starting at the same initial value but using different backtracking constants can result in convergence to two different solutions. The blue line converges to  $\tilde{\mathbf{x}} = (0, -1)$  with  $\alpha = 1$  in 5 iterations of Newton's method while the orange line converges to  $\hat{\mathbf{x}} = (3.75, .25)$  with  $\alpha = 0.4$  in 15 iterations. Note that the points in this example are 2-dimensional, which is discussed in the next section.

#### Problem 20.4:

Write a function that accepts the same arguments as your function from Problem 20 except for  $\alpha$ . Use Newton's method to find a zero of  $f$  using various values of  $\alpha$  in the interval  $(0, 1]$ . Plot the values of  $\alpha$  against the number of iterations performed by Newton's method. Return a value for  $\alpha$  that results in the lowest number of iterations.

A good test case for this problem is the function  $f(x) = x^{1/3}$  discussed in Problem 20. In this case, your plot should show that the optimal value for  $\alpha$  is actually closer to .3 than to .4.

## Newton's Method in Higher Dimensions

Newton's method can be generalized to work on functions with a multivariate domain and range. Let  $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$  be given by  $f(\mathbf{x}) = [f_1(\mathbf{x}) \ f_2(\mathbf{x}) \ \dots \ f_k(\mathbf{x})]^\top$ , with  $f_i : \mathbb{R}^n \rightarrow \mathbb{R}$  for each  $i$ . The derivative  $Df : \mathbb{R}^n \rightarrow \mathbb{R}^{n \times n}$  is the  $n \times n$  Jacobian matrix of  $f$ .

$$Df = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \dots & \frac{\partial f_1}{\partial x_k} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \dots & \frac{\partial f_n}{\partial x_k} \end{bmatrix}$$

In this setting, Newton's method seeks a vector  $\bar{\mathbf{x}}$  such that  $f(\bar{\mathbf{x}}) = 0$ , the vector of  $n$  zeros. With backtracking incorporated, (20.4) becomes

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha Df(\mathbf{x}_k)^{-1} f(\mathbf{x}_k). \quad (20.5)$$

Note that if  $n = 1$ , (20.5) is exactly (20.4) because in that case,  $Df(x)^{-1} = 1/f'(x)$ .

This vector version of Newton's method terminates when the maximum number of iterations is reached or the difference between successive approximations is less than a predetermined tolerance  $\varepsilon$  with respect to a vector norm, that is,  $\|\mathbf{x}_k - \mathbf{x}_{k-1}\| < \varepsilon$ .

### Problem 20.5: M

Modify your function from Problems ?? and 20 so that it can compute a zero of a function  $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$  for any  $n \in \mathbb{N}$ . Take the following tips into consideration.

- If  $n > 1$ ,  $f$  should be a function that accepts a 1-D NumPy array with  $n$  entries and returns another NumPy array with  $n$  entries. Similarly,  $Df$  should be a function that accepts a 1-D array with  $n$  entries and returns a  $n \times n$  array. In other words,  $f$  and  $Df$  are callable functions, but  $f(\mathbf{x})$  is a vector and  $Df(\mathbf{x})$  is a matrix.
- `np.isscalar()` may be useful for determining whether or not  $n > 1$ .
- Instead of computing  $Df(\mathbf{x}_k)^{-1}$  directly at each step, solve the system  $Df(\mathbf{x}_k)\mathbf{y}_k = f(\mathbf{x}_k)$  and set  $\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha\mathbf{y}_k$ . In other words, use `la.solve()` instead of `la.inv()`.
- The stopping criterion now requires using a norm function instead of `abs()`.

After your modifications, carefully verify that your function still works in the case that  $n = 1$ , and that your functions from Problems ?? and ?? also still work correctly. In addition, your function from Problem ?? should also work for any  $n \in \mathbb{N}$ .

### Problem 20.6: B

Bioremediation involves the use of bacteria to consume toxic wastes. At a steady state, the bacterial density  $x$  and the nutrient concentration  $y$  satisfy the system of nonlinear equations

$$\begin{aligned} \gamma xy - x(1+y) &= 0 \\ -xy + (\delta - y)(1+y) &= 0, \end{aligned}$$

where  $\gamma$  and  $\delta$  are parameters that depend on various physical features of the system.<sup>a</sup>

For this problem, assume the typical values  $\gamma = 5$  and  $\delta = 1$ , for which the system has solutions at  $(x, y) = (0, 1), (0, -1)$ , and  $(3.75, .25)$ . Write a function that finds an initial point  $\mathbf{x}_0 = (x_0, y_0)$  such that Newton's method converges to either  $(0, 1)$  or  $(0, -1)$  with  $\alpha = 1$ , and to  $(3.75, .25)$  with  $\alpha = 0.55$ . As soon as a valid  $\mathbf{x}_0$  is found, return it (stop searching).

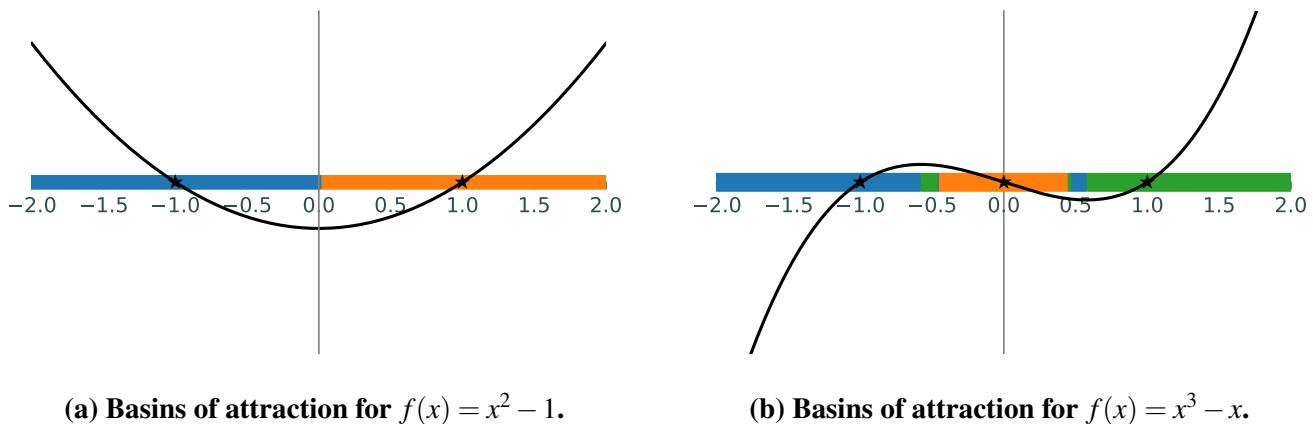
(Hint: search within the rectangle  $[-\frac{1}{4}, 0] \times [0, \frac{1}{4}]$ .)

<sup>a</sup>This problem is adapted from exercise 5.19 of [heath2002scientific] and the notes of Homer Walker).

## Basins of Attraction

When a function  $f$  has many zeros, the zero that Newton's method converges to depends on the initial guess  $x_0$ . For example, the function  $f(x) = x^2 - 1$  has zeros at  $-1$  and  $1$ . If  $x_0 < 0$ , then Newton's method converges to  $-1$ ; if  $x_0 > 0$  then it converges to  $1$  (see Figure 20.3a). The regions  $(-\infty, 0)$  and  $(0, \infty)$  are called the *basins of attraction* of  $f$ . Starting in one basin of attraction leads to finding one zero, while starting in another basin yields a different zero.

When  $f$  is a polynomial of degree greater than 2, the basins of attraction are much more interesting. For example, the basis of attraction for  $f(x) = x^3 - x$  are shown in Figure 20.3b. The basin for the zero at the origin is connected, but the other two basins are disconnected and share a kind of symmetry.



**Figure 20.3: Basins of attraction with  $\alpha = 1$ . Since choosing a different value for  $\alpha$  can change which zero Newton's method converges to, the basins of attraction may change for other values of  $\alpha$ .**

It can be shown that Newton's method converges in any Banach space with only slightly stronger hypotheses than those discussed previously. In particular, Newton's method can be performed over the complex plane  $\mathbb{C}$  to find imaginary zeros of functions. Plotting the basins of attraction over  $\mathbb{C}$  yields some interesting results.

The zeros of  $f(x) = x^3 - 1$  are  $1$ , and  $-\frac{1}{2} \pm \frac{\sqrt{3}}{2}i$ . To plot the basins of attraction for  $f(x) = x^3 - 1$  on the square complex domain  $X = \{a + bi \mid a \in [-\frac{3}{2}, \frac{3}{2}], b \in [-\frac{3}{2}, \frac{3}{2}]\}$ , create an initial grid of complex points in this domain using `np.meshgrid()`.

```
>>> x_real = np.linspace(-1.5, 1.5, 500)      # Real parts.
>>> x_imag = np.linspace(-1.5, 1.5, 500)      # Imaginary parts.
>>> X_real, X_imag = np.meshgrid(x_real, x_imag)
>>> X_0 = X_real + 1j*X_imag                  # Combine real and imaginary parts←
.
```

The grid  $X_0$  is a  $500 \times 500$  array of complex values to use as initial points for Newton's method. Array broadcasting makes it easy to compute an iteration of Newton's method at every grid point.

```
>>> f = lambda x: x**3 - 1
>>> Df = lambda x: 3*x**2
>>> X_1 = X_0 - f(X_0)/Df(X_0)
```

After enough iterations, the  $(i, j)$ th element of the grid  $X_k$  corresponds to the zero of  $f$  that results from using the  $(i, j)$ th element of  $X_0$  as the initial point. For example, with  $f(x) = x^3 - 1$ , each entry of  $X_k$  should be close to 1,  $-\frac{1}{2} + \frac{\sqrt{3}}{2}i$ , or  $-\frac{1}{2} - \frac{\sqrt{3}}{2}i$ . Each entry of  $X_k$  can then be assigned a value indicating which zero it corresponds to. Some results of this process are displayed below.

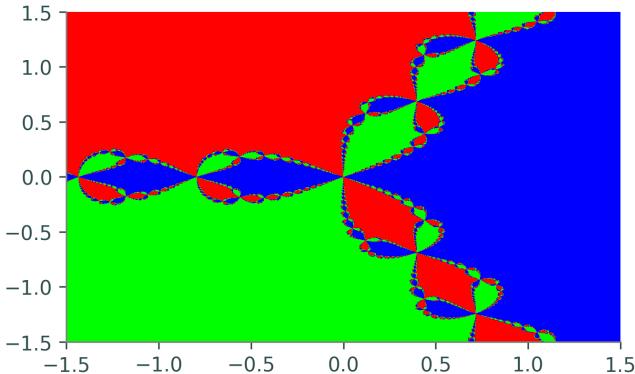
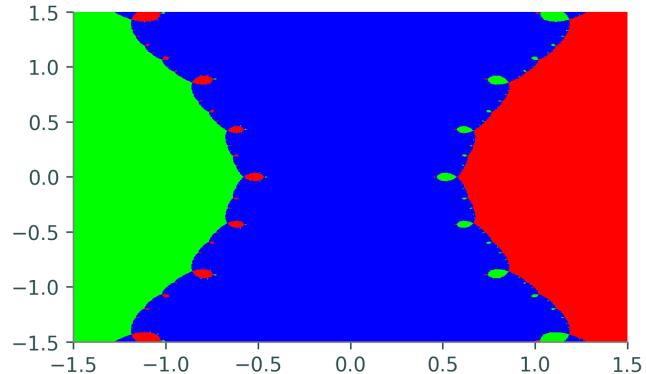
(a) Basins of attraction for  $f(x) = x^3 - 1$ .(b) Basins of attraction for  $f(x) = x^3 - x$ .

Figure 20.4

### NOTE

Notice that in some portions of Figure 20.4a, whenever red and blue try to come together, a patch of green appears in between. This behavior repeats on an infinitely small scale, producing a fractal. Because it arises from Newton's method, this kind of fractal is called a *Newton fractal*.

Newton fractals show that the long-term behavior of Newton's method is **extremely** sensitive to the initial guess  $x_0$ . Changing  $x_0$  by a small amount can change the output of Newton's method in a seemingly random way. This phenomenon is called *chaos* in mathematics.

### Problem 20.7: W

ite a function that accepts a function  $f : \mathbb{C} \rightarrow \mathbb{C}$ , its derivative  $f' : \mathbb{C} \rightarrow \mathbb{C}$ , an array `zeros` of the zeros of  $f$ , bounds  $[r_{\min}, r_{\max}, i_{\min}, i_{\max}]$  for the domain of the plot, an integer `res` that determines the resolution of the plot, and number of iterations `iters` to run the iteration. Compute and plot the basins of attraction of  $f$  in the complex plane over the specified domain in the following steps.

1. Construct a  $\text{res} \times \text{res}$  grid  $X_0$  over the domain  $\{a + bi \mid a \in [r_{\min}, r_{\max}], b \in [i_{\min}, i_{\max}]\}$ .
2. Run Newton's method (without backtracking) on  $X_0$  `iters` times, obtaining the  $\text{res} \times \text{res}$  array  $x_k$ . To avoid the additional computation of checking for convergence at each step, do not use your function from Problem 20.
3.  $X_k$  cannot be directly visualized directly because its values are complex. Solve this issue by creating another  $\text{res} \times \text{res}$  array  $Y$ . To compute the  $(i, j)$ th entry  $Y_{i,j}$ , determine which zero of  $f$  is closest to the  $(i, j)$ th entry of  $X_k$ . Set  $Y_{i,j}$  to the index of this zero in the array `zeros`. If there are  $R$  distinct zeros, each  $Y_{i,j}$  should be one of  $0, 1, \dots, R - 1$ .  
(Hint: `np.argmin()` may be useful.)
4. Use `plt.pcolormesh()` to visualize the basins. Recall that this function accepts three array arguments: the  $x$ -coordinates (in this case, the real components of the initial grid), the  $y$ -coordinates (the imaginary components of the grid), and an array indicating color values ( $Y$ ). Set `cmap="brg"` to get the same color scheme as in Figure 20.4.

Test your function using  $f(x) = x^3 - 1$  and  $f(x) = x^3 - x$ . The resulting plots should resemble Figures 20.4a and 20.4b, respectively (perhaps with the colors permuted).



# 21. One-dimensional Optimization

---

**Lab Objective:** *Most mathematical optimization problems involve estimating the minimizer(s) of a scalar-valued function. Many algorithms for optimizing functions with a high-dimensional domain depend on routines for optimizing functions of a single variable. There are many techniques for optimization in one dimension, each with varying degrees of precision and speed. In this lab, we implement the golden section search method, Newton's method, and the secant method, then apply them to the backtracking problem.*

## Golden Section Search

---

A function  $f : [a, b] \rightarrow \mathbb{R}$  satisfies the *unimodal property* if it has exactly one local minimum and is monotonic on either side of the minimizer. In other words,  $f$  decreases from  $a$  to its minimizer  $x^*$ , then increases up to  $b$  (see Figure 21.1). The *golden section search* method optimizes a unimodal function  $f$  by iteratively defining smaller and smaller intervals containing the unique minimizer  $x^*$ . This approach is especially useful if the function's derivative does not exist, is unknown, or is very costly to compute.

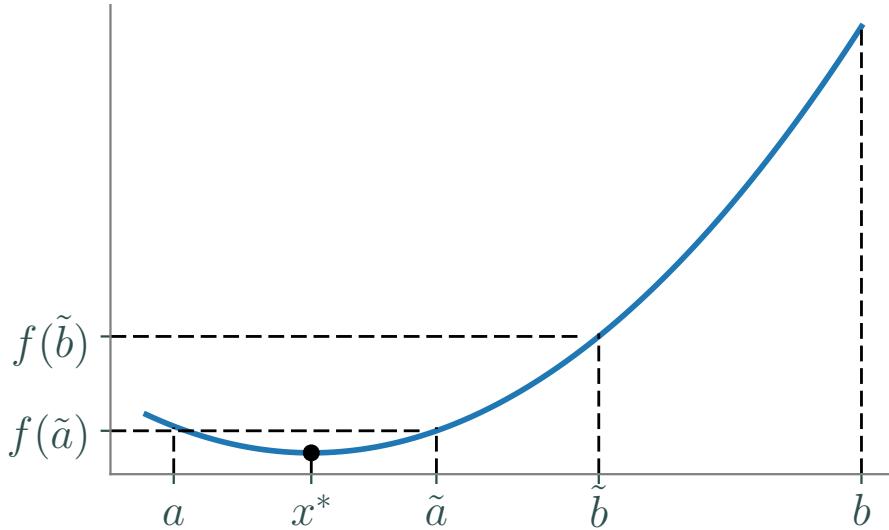
By definition, the minimizer  $x^*$  of  $f$  must lie in the interval  $[a, b]$ . To shrink the interval around  $x^*$ , we test the following strategically chosen points.

$$\tilde{a} = b - \frac{b-a}{\varphi} \quad \tilde{b} = a + \frac{b-a}{\varphi}$$

Here  $\varphi = \frac{1+\sqrt{5}}{2}$  is the *golden ratio*. At each step of the search,  $[a, b]$  is refined to either  $[a, \tilde{b}]$  or  $[\tilde{a}, b]$ , called the *golden sections*, depending on the following criteria.

If  $f(\tilde{a}) < f(\tilde{b})$ , then since  $f$  is unimodal, it must be increasing in a neighborhood of  $\tilde{b}$ . The unimodal property also guarantees that  $f$  must be increasing on  $[\tilde{b}, b]$  as well, so  $x^* \in [a, \tilde{b}]$  and we set  $b = \tilde{b}$ . By similar reasoning, if  $f(\tilde{a}) > f(\tilde{b})$ , then  $x^* \in [\tilde{a}, b]$  and we set  $a = \tilde{a}$ . If, however,  $f(\tilde{a}) = f(\tilde{b})$ , then the unimodality of  $f$  does not guarantee anything about where the minimizer lies. Assuming either  $x^* \in [a, \tilde{b}]$  or  $x^* \in [\tilde{a}, b]$  allows the iteration to continue, but the method is no longer guaranteed to converge to the local minimum.

At each iteration, the length of the search interval is divided by  $\varphi$ . The method therefore converges linearly, which is somewhat slow. However, the idea is simple and each step is computationally inexpensive.



**Figure 21.1:** The unimodal  $f : [a, b] \rightarrow \mathbb{R}$  can be minimized with a golden section search. For the first iteration,  $f(\tilde{a}) < f(\tilde{b})$ , so  $x^* \in [a, \tilde{b}]$ . New values of  $\tilde{a}$  and  $\tilde{b}$  are then calculated from this new, smaller interval.

---

**Algorithm 6** The Golden Section Search

---

```

1: procedure GOLDEN_SECTION( $f, a, b, \text{tol}, \text{maxiter}$ )
2:    $x_0 \leftarrow (a + b)/2$                                  $\triangleright$  Set the initial minimizer approximation as the interval midpoint.
3:    $\varphi = (1 + \sqrt{5})/2$ 
4:   for  $i = 1, 2, \dots, \text{maxiter}$  do                       $\triangleright$  Iterate only  $\text{maxiter}$  times at most.
5:      $c \leftarrow (b - a)/\varphi$ 
6:      $\tilde{a} \leftarrow b - c$ 
7:      $\tilde{b} \leftarrow a + c$ 
8:     if  $f(\tilde{a}) \leq f(\tilde{b})$  then                   $\triangleright$  Get new boundaries for the search interval.
9:        $b \leftarrow \tilde{b}$ 
10:    else
11:       $a \leftarrow \tilde{a}$ 
12:       $x_1 \leftarrow (a + b)/2$                            $\triangleright$  Set the minimizer approximation as the interval midpoint.
13:      if  $|x_0 - x_1| < \text{tol}$  then
14:        break                                      $\triangleright$  Stop iterating if the approximation stops changing enough.
15:       $x_0 \leftarrow x_1$ 
16:   return  $x_1$ 

```

---

### Problem 21.1: W

ite a function that accepts a function  $f : \mathbb{R} \rightarrow \mathbb{R}$ , interval limits  $a$  and  $b$ , a stopping tolerance  $tol$ , and a maximum number of iterations  $maxiter$ . Use Algorithm 6 to implement the golden section search. Return the approximate minimizer  $x^*$ , whether or not the algorithm converged (true or false), and the number of iterations computed.

Test your function by minimizing  $f(x) = e^x - 4x$  on the interval  $[0, 3]$ , then plotting the function and the computed minimizer together. Also compare your results to SciPy's golden section search, `scipy.optimize.golden()`.

```
>>> from scipy import optimize as opt
>>> import numpy as np

>>> f = lambda x : np.exp(x) - 4*x
>>> opt.golden(f, brack=(0,3), tol=.001)
1.3862578679031485 # ln(4) is the minimizer.
```

## Newton's Method

---

*Newton's method* is an important root-finding algorithm that can also be used for optimization. Given  $f : \mathbb{R} \rightarrow \mathbb{R}$  and a good initial guess  $x_0$ , the sequence  $(x_k)_{k=1}^\infty$  generated by the recursive rule

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

converges to a point  $\bar{x}$  satisfying  $f(\bar{x}) = 0$ . The first-order necessary conditions from elementary calculus state that if  $f$  is differentiable, then its derivative evaluates to zero at each of its local minima and maxima. Therefore using Newton's method to find the zeros of  $f'$  is a way to identify potential minima or maxima of  $f$ . Specifically, starting with an initial guess  $x_0$ , set

$$x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)} \tag{21.1}$$

and iterate until  $|x_k - x_{k-1}|$  is satisfactorily small. Note that this procedure does not use the actual function  $f$  at all, but it requires many evaluations of its first and second derivatives. As a result, Newton's method converges in few iterations, but it can be computationally expensive.

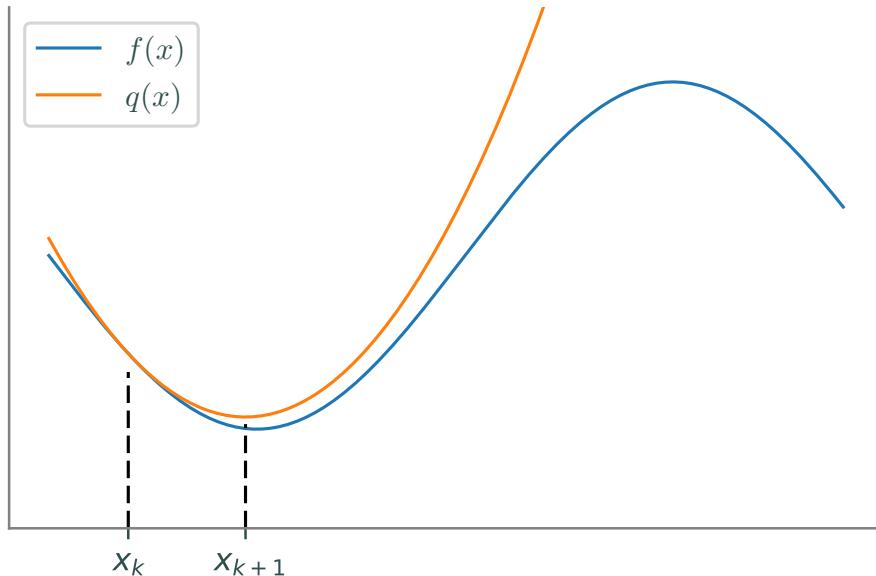
Each step of (21.1) can be thought of approximating the objective function  $f$  by a quadratic function  $q$  and finding its unique extrema. That is, we first approximate  $f$  with its second-degree Taylor polynomial centered at  $x_k$ .

$$q(x) = f(x_k) + f'(x_k)(x - x_k) + \frac{1}{2}f''(x_k)(x - x_k)^2$$

This quadratic function satisfies  $q(x_k) = f(x_k)$  and matches  $f$  fairly well close to  $x_k$ . Thus the optimizer of  $q$  is a reasonable guess for an optimizer of  $f$ . To compute that optimizer, solve  $q'(x) = 0$ .

$$0 = q'(x) = f'(x_k) + f''(x_k)(x - x_k) \implies x = x_k - \frac{f'(x_k)}{f''(x_k)}$$

This agrees with (21.1) using  $x_{k+1}$  for  $x$ . See Figure 21.2.



**Figure 21.2: A quadratic approximation of  $f$  at  $x_k$ . The minimizer  $x_{k+1}$  of  $q$  is close to the minimizer of  $f$ .**

Newton's method for optimization works well to locate minima when  $f''(x) > 0$  on the entire domain. However, it may fail to converge to a minimizer if  $f''(x) \leq 0$  for some portion of the domain. If  $f$  is not unimodal, the initial guess  $x_0$  must be sufficiently close to a local minimizer  $x^*$  in order to converge.

### Problem 21.2: L

Let  $f : \mathbb{R} \rightarrow \mathbb{R}$ . Write a function that accepts  $f'$ ,  $f''$ , a starting point  $x_0$ , a stopping tolerance  $tol$ , and a maximum number of iterations  $maxiter$ . Implement Newton's method using (21.1) to locate a local optimizer. Return the approximate optimizer, whether or not the algorithm converged, and the number of iterations computed.

Test your function by minimizing  $f(x) = x^2 + \sin(5x)$  with an initial guess of  $x_0 = 0$ . Compare your results to `scipy.optimize.newton()`, which implements the root-finding version of Newton's method.

```
>>> df = lambda x : 2*x + 5*np.cos(5*x)
>>> d2f = lambda x : 2 - 25*np.sin(5*x)
>>> opt.newton(df, x0=0, fprime=d2f, tol=1e-10, maxiter=500)
-1.4473142236328096
```

Note that other initial guesses can yield different minima for this function.

## The Secant Method

---

The second derivative of an objective function is not always known or may be prohibitively expensive to evaluate. The *secant method* solves this problem by numerically approximating the second derivative with a difference quotient.

$$f''(x) \approx \frac{f'(x+h) - f'(x)}{h}$$

Selecting  $x = x_k$  and  $h = x_{k-1} - x_k$  gives the following approximation.

$$f''(x_k) \approx \frac{f'(x_k + x_{k-1} - x_k) - f'(x_k)}{x_{k-1} - x_k} = \frac{f(x_k) - f'(x_{k-1})}{x_k - x_{k-1}} \quad (21.2)$$

Inserting (21.2) into (21.1) results in the complete secant method formula.

$$x_{k+1} = x_k - \frac{x_k - x_{k-1}}{f'(x_k) - f'(x_{k-1})} f'(x_k) = \frac{x_{k-1}f'(x_k) - x_kf'(x_{k-1})}{f'(x_k) - f'(x_{k-1})} \quad (21.3)$$

Notice that this recurrence relation requires two previous points (both  $x_k$  and  $x_{k-1}$ ) to calculate the next estimate. This method converges superlinearly—slower than Newton's method, but faster than the golden section search—with convergence criteria similar to Newton's method.

**Problem 21.3: W**

ite a function that accepts a first derivative  $f'$ , starting points  $x_0$  and  $x_1$ , a stopping tolerance  $tol$ , and a maximum of iterations  $maxiter$ . Use (21.3) to implement the Secant method. Try to make as few computations as possible by only computing  $f'(x_k)$  once for each  $k$ . Return the minimizer approximation, whether or not the algorithm converged, and the number of iterations computed. Test your code with the function  $f(x) = x^2 + \sin(x) + \sin(10x)$  and with initial guesses of  $x_0 = 0$  and  $x_1 = -1$ . Plot your answer with the graph of the function. Also compare your results to `scipy.optimize.newton()`; without providing the `fprime` argument, this function uses the secant method. However, it still only takes in one initial condition, so it may converge to a different local minimum than your function.

```
>>> df = lambda x: 2*x + np.cos(x) + 10*np.cos(10*x)
>>> opt.newton(df, x0=0, tol=1e-10, maxiter=500)
-3.2149595174761636
```

## Descent Methods

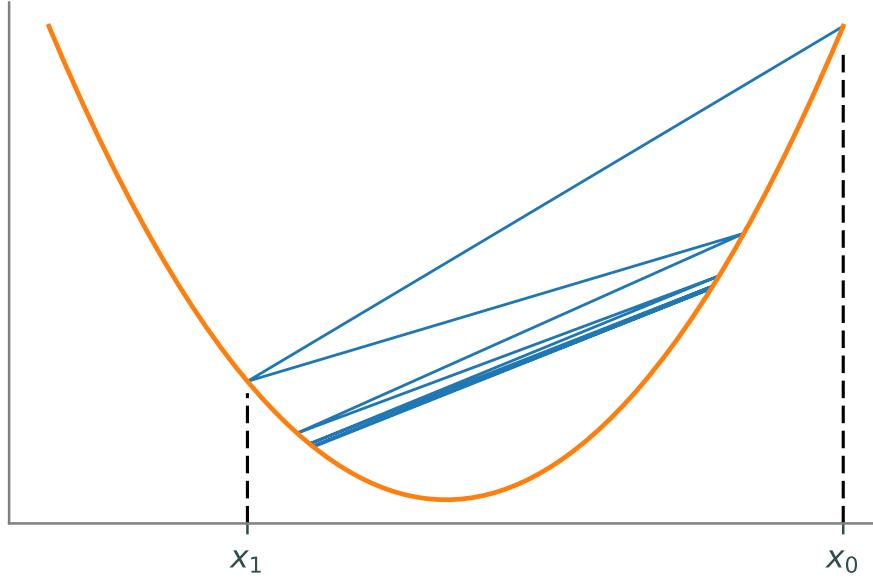
---

Consider now a function  $f : \mathbb{R}^n \rightarrow \mathbb{R}$ . *Descent methods*, also called *line search methods*, are optimization algorithms that create a convergent sequence  $(x_k)_{k=1}^\infty$  by the following rule.

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k \quad (21.4)$$

Here  $\alpha_k \in \mathbb{R}$  is called the *step size* and  $\mathbf{p}_k \in \mathbb{R}^n$  is called the *search direction*. The choice of  $\mathbf{p}_k$  is usually what distinguishes an algorithm; in the one-dimensional case ( $n = 1$ ),  $p_k = f'(x_k)/f''(x_k)$  results in Newton's method, and using the approximation in (21.2) results in the secant method.

To be effective, a descent method must also use a good step size  $\alpha_k$ . If  $\alpha_k$  is too large, the method may repeatedly overstep the minimum; if  $\alpha_k$  is too small, the method may converge extremely slowly. See Figure 21.3.



**Figure 21.3: If the step size  $\alpha_k$  is too large, a descent method may repeatedly overstep the minimizer.**

Given a search direction  $\mathbf{p}_k$ , the best step size  $\alpha_k$  minimizes the function  $\phi_k(\alpha) = f(\mathbf{x}_k + \alpha\mathbf{p}_k)$ . Since  $f$  is scalar-valued,  $\phi_k : \mathbb{R} \rightarrow \mathbb{R}$ , so any of the optimization methods discussed previously can be used to minimize  $\phi_k$ . However, computing the best  $\alpha_k$  at every iteration is not always practical. Instead, some methods use a cheap routine to compute a step size that may not be optimal, but which is good enough. The most common approach is to find an  $\alpha_k$  that satisfies the *Wolfe conditions*:

$$f(\mathbf{x}_k + \alpha_k \mathbf{p}_k) \leq f(\mathbf{x}_k) + c_1 \alpha_k Df(\mathbf{x}_k)^T \mathbf{p}_k \quad (21.5)$$

$$-Df(\mathbf{x}_k + \alpha_k \mathbf{p}_k)^T \mathbf{p}_k \leq -c_2 Df(\mathbf{x}_k)^T \mathbf{p}_k \quad (21.6)$$

where  $0 < c_1 < c_2 < 1$  (for the best results, choose  $c_1 \ll c_2$ ). The condition (21.5) is also called the *Armijo rule* and ensures that the step decreases  $f$ . However, this condition is not enough on its own. By Taylor's theorem,

$$f(\mathbf{x}_k + \alpha_k \mathbf{p}_k) = f(\mathbf{x}_k) + \alpha_k Df(\mathbf{x}_k)^T \mathbf{p}_k + \mathcal{O}(\alpha_k^2).$$

Thus, a very small  $\alpha_k$  will always satisfy (21.5) since  $Df(\mathbf{x}_k)^T \mathbf{p}_k < 0$  (as  $\mathbf{p}_k$  is a descent direction). The condition (21.6), called the *curvature condition*, ensures that the  $\alpha_k$  is large enough for the algorithm to make significant progress.

It is possible to find an  $\alpha_k$  that satisfies the Wolfe conditions, but that is far from the minimizer of  $\phi_k(\alpha)$ . The *strong Wolfe conditions* modify (21.6) to ensure that  $\alpha_k$  is near the minimizer.

$$|Df(\mathbf{x}_k + \alpha_k \mathbf{p}_k)^T \mathbf{p}_k| \leq c_2 |Df(\mathbf{x}_k)^T \mathbf{p}_k|$$

The *Armijo–Goldstein conditions* provide another alternative to (21.6):

$$f(\mathbf{x}_k) + (1 - c) \alpha_k Df(\mathbf{x}_k)^T \mathbf{p}_k \leq f(\mathbf{x}_k + \alpha_k \mathbf{p}_k) \leq f(\mathbf{x}_k) + c \alpha_k Df(\mathbf{x}_k)^T \mathbf{p}_k,$$

where  $0 < c < 1$ . These conditions are very similar to the Wolfe conditions (the right inequality is (21.5)), but they do not require the calculation of the directional derivative  $Df(\mathbf{x}_k + \alpha_k \mathbf{p}_k)^T \mathbf{p}_k$ .

## Backtracking

---

A *backtracking line search* is a simple strategy for choosing an acceptable step size  $\alpha_k$ : start with a fairly large initial step size  $\alpha$ , then repeatedly scale it down by a factor  $\rho$  until the desired conditions are satisfied. The following algorithm only requires  $\alpha$  to satisfy (21.5). This is usually sufficient, but if it finds  $\alpha$ 's that are too small, the algorithm can be modified to satisfy (21.6) or one of its variants.

---

### Algorithm 7 Backtracking using the Armijo Rule

---

```

1: procedure BACKTRACKING( $f$ ,  $Df$ ,  $\mathbf{x}_k$ ,  $\mathbf{p}_k$ ,  $\alpha$ ,  $\rho$ ,  $c$ )
2:    $Dfp \leftarrow Df(\mathbf{x}_k)^T \mathbf{p}_k$                                  $\triangleright$  Compute these values only once.
3:    $fx \leftarrow f(\mathbf{x}_k)$ 
4:   while  $(f(\mathbf{x}_k + \alpha \mathbf{p}_k) > fx + c\alpha Dfp)$  do
5:      $\alpha \leftarrow \rho \alpha$ 
return  $\alpha$ 
```

---

#### Problem 21.4: W

ite a function that accepts a function  $f : \mathbb{R}^n \rightarrow \mathbb{R}$ , its derivative  $Df : \mathbb{R}^n \rightarrow \mathbb{R}^n$ , an approximate minimizer  $\mathbf{x}_k$ , a search direction  $\mathbf{p}_k$ , an initial step length  $\alpha$ , and parameters  $\rho$  and  $c$ . Implement the backtracking method of Algorithm 7. Return the computed step size.

The functions  $f$  and  $Df$  should both accept 1-D NumPy arrays of length  $n$ . For example, if  $f(x,y,z) = x^2 + y^2 + z^2$ , then  $f$  and  $Df$  could be defined as follows.

```
>>> f = lambda x: x[0]**2 + x[1]**2 + x[2]**2
>>> Df = lambda x: np.array([2*x[0], 2*x[1], 2*x[2]])
```

SciPy's `scipy.optimize.linesearch.scalar_search_armijo()` finds an acceptable step size using the Armijo rule. It may not give the exact answer as your implementation since it decreases  $\alpha$  differently, but the answers should be similar.

```
>>> from scipy.optimize import linesearch
>>> from autograd import numpy as np
>>> from autograd import grad

# Get a step size for  $f(x,y,z) = x^2 + y^2 + z^2$ .
>>> f = lambda x: x[0]**2 + x[1]**2 + x[2]**2
>>> x = np.array([150., .03, 40.])           # Current minimizer guesss.
>>> p = np.array([-5, -100., -4.5])         # Current search direction.
>>> phi = lambda alpha: f(x + alpha*p)        # Define phi(alpha).
>>> dphi = grad(phi)
>>> alpha, _ = linesearch.scalar_search_armijo(phi, phi(0.), dphi(0.))
```

# 22. Gradient Descent Methods

---

**Lab Objective:** Iterative optimization methods choose a search direction and a step size at each iteration. One simple choice for the search direction is the negative gradient, resulting in the method of steepest descent. While theoretically foundational, in practice this method is often slow to converge. An alternative method, the conjugate gradient algorithm, uses a similar idea that results in much faster convergence in some situations. In this lab we implement a method of steepest descent and two conjugate gradient methods, then apply them to regression problems.

## The Method of Steepest Descent

---

Let  $f : \mathbb{R}^n \rightarrow \mathbb{R}$  with first derivative  $Df : \mathbb{R}^n \rightarrow \mathbb{R}^n$ . The following iterative technique is a common template for methods that aim to compute a local minimizer  $\mathbf{x}^*$  of  $f$ .

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k \quad (22.1)$$

Here  $\mathbf{x}_k$  is the  $k$ th approximation to  $\mathbf{x}^*$ ,  $\alpha_k$  is the *step size*, and  $\mathbf{p}_k$  is the *search direction*. Newton's method and its relatives follow this pattern, but they require the calculation (or approximation) of the inverse Hessian matrix  $Df^2(\mathbf{x}_k)^{-1}$  at each step. The following idea is a simpler and less computationally intensive approach than Newton and quasi-Newton methods.

The derivative  $Df(\mathbf{x})^\top$  (often called the *gradient* of  $f$  at  $\mathbf{x}$ , sometimes notated  $\nabla f(\mathbf{x})$ ) is a vector that points in the direction of greatest **increase** of  $f$  at  $\mathbf{x}$ . It follows that the negative derivative  $-Df(\mathbf{x})^\top$  points in the direction of steepest **decrease** at  $\mathbf{x}$ . The *method of steepest descent* chooses the search direction  $\mathbf{p}_k = -Df(\mathbf{x}_k)^\top$  at each step of (22.1), resulting in the following algorithm.

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k Df(\mathbf{x}_k)^\top \quad (22.2)$$

Setting  $\alpha_k = 1$  for each  $k$  is often sufficient for Newton and quasi-Newton methods. However, a constant choice for the step size in (22.2) can result in oscillating approximations or even cause the sequence  $(\mathbf{x}_k)_{k=1}^\infty$  to travel away from the minimizer  $\mathbf{x}^*$ . To avoid this problem, the step size  $\alpha_k$  can be chosen in a few ways.

- Start with  $\alpha_k = 1$ , then set  $\alpha_k = \frac{\alpha_k}{2}$  until  $f(\mathbf{x}_k - \alpha_k Df(\mathbf{x}_k)^\top) < f(\mathbf{x}_k)$ , terminating the iteration if  $\alpha_k$  gets too small. This guarantees that the method actually descends at each step and that  $\alpha_k$  satisfies the Armijo rule, without endangering convergence.
- At each step, solve the following one-dimensional optimization problem.

$$\alpha_k = \operatorname{argmin}_\alpha f(\mathbf{x}_k - \alpha Df(\mathbf{x}_k)^\top)$$

Using this choice is called *exact steepest descent*. This option is more expensive per iteration than the above strategy, but it results in fewer iterations before convergence.

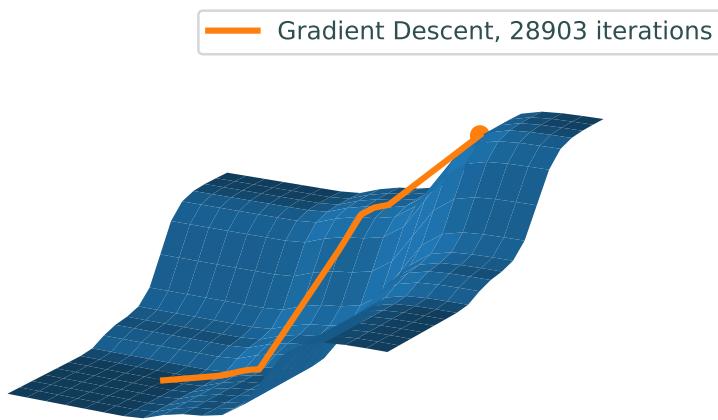
### Problem 22.1: W

ite a function that accepts an objective function  $f : \mathbb{R}^n \rightarrow \mathbb{R}$ , its derivative  $Df : \mathbb{R}^n \rightarrow \mathbb{R}^n$ , an initial guess  $\mathbf{x}_0 \in \mathbb{R}^n$ , a convergence tolerance  $\text{tol}$  defaulting to  $1e^{-5}$ , and a maximum number of iterations  $\text{maxiter}$  defaulting to 100. Implement the exact method of steepest descent, using a one-dimensional optimization method to choose the step size (use `opt.minimize_scalar()` or your own 1-D minimizer). Iterate until  $\|Df(\mathbf{x}_k)\|_\infty < \text{tol}$  or  $k > \text{maxiter}$ . Return the approximate minimizer  $\mathbf{x}^*$ , whether or not the algorithm converged (`True` or `False`), and the number of iterations computed.

Test your function on  $f(x, y, z) = x^4 + y^4 + z^4$  (easy) and the Rosenbrock function (hard). It should take many iterations to minimize the Rosenbrock function, but it should converge eventually with a large enough choice of `maxiter`.

## The Conjugate Gradient Method

Unfortunately, the method of steepest descent can be very inefficient for certain problems. Depending on the nature of the objective function, the sequence of points can zig-zag back and forth or get stuck on flat areas without making significant progress toward the true minimizer.



**Figure 22.1:** On this surface, gradient descent takes an extreme number of iterations to converge to the minimum because it gets stuck in the flat basins of the surface.

Unlike the method of steepest descent, the *conjugate gradient algorithm* chooses a search direction that is guaranteed to be a descent direction, though not the direction of greatest descent. These directions are using a generalized form of orthogonality called *conjugacy*.

Let  $Q$  be a square, positive definite matrix. A set of vectors  $\{\mathbf{x}_0, \mathbf{x}_1, \dots, \mathbf{x}_m\}$  is called  $Q$ -*conjugate* if each distinct pair of vectors  $\mathbf{x}_i, \mathbf{x}_j$  satisfy  $\mathbf{x}_i^\top Q \mathbf{x}_j = 0$ . A  $Q$ -conjugate set of vectors is linearly independent and can form a basis that diagonalizes the matrix  $Q$ . This guarantees that an iterative method to solve  $Q\mathbf{x} = \mathbf{b}$  only require as many steps as there are basis vectors.

Solve a positive definite system  $Q\mathbf{x} = \mathbf{b}$  is valuable in and of itself for certain problems, but it is also equivalent to minimizing certain functions. Specifically, consider the quadratic function

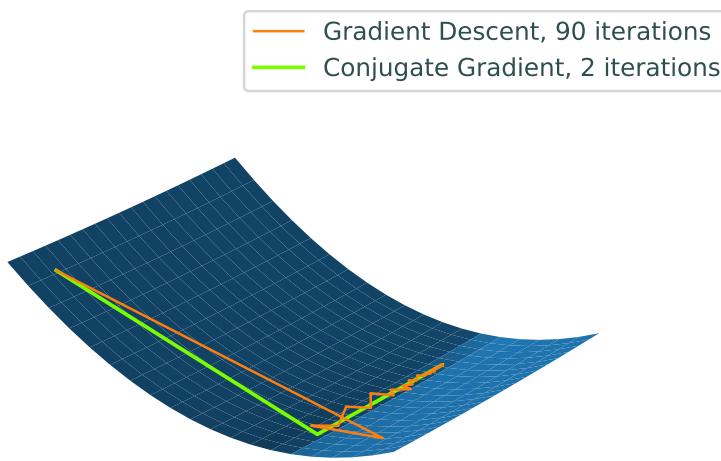
$$f(\mathbf{x}) = \frac{1}{2} \mathbf{x}^\top Q \mathbf{x} - \mathbf{b}^\top \mathbf{x} + c.$$

Because  $Df(\mathbf{x})^\top = Q\mathbf{x} - \mathbf{b}$ , minimizing  $f$  is the same as solving the equation

$$0 = Df(\mathbf{x})^\top = Q\mathbf{x} - \mathbf{b} \Rightarrow Q\mathbf{x} = \mathbf{b},$$

which is the original linear system. Note that the constant  $c$  does not affect the minimizer, since if  $\mathbf{x}^*$  minimizes  $f(\mathbf{x})$  it also minimizes  $f(\mathbf{x}) + c$ .

Using the conjugate directions guarantees an iterative method to converge on the minimizer because each iteration minimizes the objective function over a subspace of dimension equal to the iteration number. Thus, after  $n$  steps, where  $n$  is the number of conjugate basis vectors, the algorithm has found a minimizer over the entire space. In certain situations, this has a great advantage over gradient descent, which can bounce back and forth. This comparison is illustrated in Figure 22.2. Additionally, because the method utilizes a basis of conjugate vectors, the previous search direction can be used to find a conjugate projection onto the next subspace, saving computational time.



**Figure 22.2:** Paths traced by Gradient Descent (orange) and Conjugate Gradient (red) on a quadratic surface. Notice the zig-zagging nature of the Gradient Descent path, as opposed to the Conjugate Gradient path, which finds the minimizer in 2 steps.

**Algorithm 8**


---

```

1: procedure CONJUGATE GRADIENT( $\mathbf{x}_0, Q, \mathbf{b}, \text{tol}$ )
2:    $\mathbf{r}_0 \leftarrow Q\mathbf{x}_0 - \mathbf{b}$ 
3:    $\mathbf{d}_0 \leftarrow -\mathbf{r}_0$ 
4:    $k \leftarrow 0$ 
5:   while  $\|\mathbf{r}_k\| \geq \text{tol}, k < n$  do
6:      $\alpha_k \leftarrow \mathbf{r}_k^\top \mathbf{r}_k / \mathbf{d}_k^\top Q \mathbf{d}_k$ 
7:      $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \alpha_k \mathbf{d}_k$ 
8:      $\mathbf{r}_{k+1} \leftarrow \mathbf{r}_k + \alpha_k Q \mathbf{d}_k$ 
9:      $\beta_{k+1} \leftarrow \mathbf{r}_{k+1}^\top \mathbf{r}_{k+1} / \mathbf{r}_k^\top \mathbf{r}_k$ 
10:     $\mathbf{d}_{k+1} \leftarrow -\mathbf{r}_{k+1} + \beta_{k+1} \mathbf{d}_k$ 
11:     $k \leftarrow k + 1.$ 
return  $\mathbf{x}_{k+1}$ 

```

---

The points  $\mathbf{x}_k$  are the successive approximations to the minimizer, the vectors  $\mathbf{d}_k$  are the conjugate descent directions, and the vectors  $\mathbf{r}_k$  (which actually correspond to the steepest descent directions) are used in determining the conjugate directions. The constants  $\alpha_k$  and  $\beta_k$  are used, respectively, in the line search, and in ensuring the  $Q$ -conjugacy of the descent directions.

### Problem 22.2: W

ite a function that accepts an  $n \times n$  positive definite matrix  $Q$ , a vector  $\mathbf{b} \in \mathbb{R}^n$ , an initial guess  $\mathbf{x}_0 \in \mathbb{R}^n$ , and a stopping tolerance. Use Algorithm 8 to solve the system  $Q\mathbf{x} = \mathbf{b}$ . Continue the algorithm until  $\|\mathbf{r}_k\|$  is less than the tolerance, iterating no more than  $n$  times. Return the solution  $\mathbf{x}$ , whether or not the algorithm converged in  $n$  iterations or less, and the number of iterations computed.

Test your function on the simple system

$$Q = \begin{bmatrix} 2 & 0 \\ 0 & 4 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 1 \\ 8 \end{bmatrix},$$

which has solution  $\mathbf{x}^* = [\frac{1}{2}, 2]^\top$ . This is equivalent to minimizing the quadratic function  $f(x, y) = x^2 + 2y^2 - x - 8y$ ; check that your function from Problem 22 gets the same solution.

More generally, you can generate a random positive definite matrix  $Q$  for testing by setting setting  $Q = A^\top A$  for any  $A$  of full rank.

```
>>> import numpy as np
>>> from scipy import linalg as la

# Generate Q, b, and the initial guess x0.
>>> n = 10
>>> A = np.random.random((n,n))
>>> Q = A.T @ A
>>> b, x0 = np.random.random((2,n))

>>> x = la.solve(Q, b)      # Use your function here.
>>> np.allclose(Q @ x, b)
True
```

## Non-linear Conjugate Gradient

The algorithm presented above is only valid for certain linear systems and quadratic functions, but the basic strategy may be adapted to minimize more general convex or non-linear functions. Though the non-linear version does not have guaranteed convergence as the linear formulation does, it can still converge in less iterations than the method of steepest descent. Modifying the algorithm for more general functions requires new formulas for  $\alpha_k$ ,  $\mathbf{r}_k$ , and  $\beta_k$ .

- The scalar  $\alpha_k$  is simply the result of performing a line-search in the given direction  $\mathbf{d}_k$  and is thus defined  $\alpha_k = \underset{\alpha}{\operatorname{argmin}} f(\mathbf{x}_k + \alpha \mathbf{d}_k)$ .
- The vector  $\mathbf{r}_k$  in the original algorithm was really just the gradient of the objective function, so now

define  $\mathbf{r}_k = Df(\mathbf{x}_k)^\top$ .

- The constants  $\beta_k$  can be defined in various ways, and the most correct choice depends on the nature of the objective function. A well-known formula, attributed to Fletcher and Reeves, is  $\beta_k = Df(\mathbf{x}_k)Df(\mathbf{x}_k)^\top / Df(\mathbf{x}_{k-1})Df(\mathbf{x}_{k-1})^\top$ .

**Algorithm 9**

```

1: procedure NON-LINEAR CONJUGATE GRADIENT( $f, Df, \mathbf{x}_0, \text{tol}, \text{maxiter}$ )
2:    $\mathbf{r}_0 \leftarrow -Df(\mathbf{x}_0)^\top$ 
3:    $\mathbf{d}_0 \leftarrow \mathbf{r}_0$ 
4:    $\alpha_0 \leftarrow \underset{\alpha}{\operatorname{argmin}} f(\mathbf{x}_0 + \alpha \mathbf{d}_0)$ 
5:    $\mathbf{x}_1 \leftarrow \mathbf{x}_0 + \alpha_0 \mathbf{d}_0$ 
6:    $k \leftarrow 1$ 
7:   while  $\|\mathbf{r}_k\| \geq \text{tol}, k < \text{maxiter}$  do
8:      $\mathbf{r}_k \leftarrow -Df(\mathbf{x}_k)^\top$ 
9:      $\beta_k = \mathbf{r}_k^\top \mathbf{r}_k / \mathbf{r}_{k-1}^\top \mathbf{r}_{k-1}$ 
10:     $\mathbf{d}_k \leftarrow \mathbf{r}_k + \beta_k \mathbf{d}_{k-1}$ .
11:     $\alpha_k \leftarrow \underset{\alpha}{\operatorname{argmin}} f(\mathbf{x}_k + \alpha \mathbf{d}_k).$ 
12:     $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \alpha_k \mathbf{d}_k.$ 
13:     $k \leftarrow k + 1$ .

```

**Problem 22.3: W**

ite a function that accepts a convex objective function  $f$ , its derivative  $Df$ , an initial guess  $\mathbf{x}_0$ , a convergence tolerance defaultin to  $1e^{-5}$ , and a maximum number of iterations defaultin to 100. Use Algorithm 9 to compute the minimizer  $\mathbf{x}^*$  of  $f$ . Return the approximate minimizer, whether or not the algorithm converged, and the number of iterations computed.

Compare your function to SciPy's `opt.fmin_cg()`.

```

>>> opt.fmin_cg(opt.rosen, np.array([10, 10]), fprime=opt.rosen_der)
Optimization terminated successfully.
    Current function value: 0.000000
    Iterations: 44
    Function evaluations: 102 # Much faster than steepest descent!
    Gradient evaluations: 102
    array([ 1.00000007,  1.00000015])

```

# Regression Problems

---

A major use of the conjugate gradient method is solving linear least squares problems. Recall that a least squares problem can be formulated as an optimization problem:

$$\mathbf{x}^* = \min_{\mathbf{x}} \|\mathbf{Ax} - \mathbf{b}\|_2,$$

where  $A$  is an  $m \times n$  matrix with full column rank,  $\mathbf{x} \in \mathbb{R}^n$ , and  $\mathbf{b} \in \mathbb{R}^m$ . The solution can be calculated analytically, and is given by

$$\mathbf{x}^* = (A^T A)^{-1} A^T \mathbf{b}.$$

In other words, the minimizer solves the linear system

$$A^T A \mathbf{x} = A^T \mathbf{b}. \quad (22.3)$$

Since  $A$  has full column rank, it is invertible,  $A^T A$  is positive definite, and for any non-zero vector  $\mathbf{z}$ ,  $A\mathbf{z} \neq 0$ . Therefore,  $\mathbf{z}^T A^T A \mathbf{z} = \|A\mathbf{z}\|^2 > 0$ . As  $A^T A$  is positive definite, conjugate gradient can be used to solve Equation 22.3.

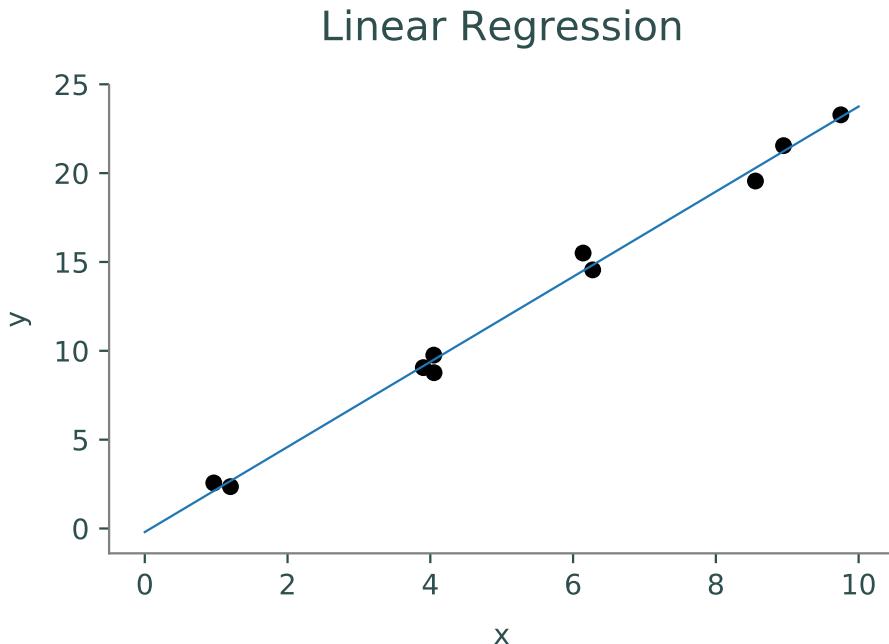
Linear least squares is the mathematical underpinning of *linear regression*. Linear regression involves a set of real-valued data points  $\{y_1, \dots, y_m\}$ , where each  $y_i$  is paired with a corresponding set of predictor variables  $\{x_{i,1}, x_{i,2}, \dots, x_{i,n}\}$  with  $n < m$ . The linear regression model posits that

$$y_i = \beta_0 + \beta_1 x_{i,1} + \beta_2 x_{i,2} + \dots + \beta_n x_{i,n} + \varepsilon_i$$

for  $i = 1, 2, \dots, m$ . The real numbers  $\beta_0, \dots, \beta_n$  are known as the parameters of the model, and the  $\varepsilon_i$  are independent, normally-distributed error terms. The goal of linear regression is to calculate the parameters that best fit the data. This can be accomplished by posing the problem in terms of linear least squares. Define

$$\mathbf{b} = \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix}, \quad A = \begin{bmatrix} 1 & x_{1,1} & x_{1,2} & \cdots & x_{1,n} \\ 1 & x_{2,1} & x_{2,2} & \cdots & x_{2,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{m,1} & x_{m,2} & \cdots & x_{m,n} \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_n \end{bmatrix}.$$

The solution  $\mathbf{x}^* = [\beta_0^*, \beta_1^*, \dots, \beta_n^*]^T$  to the system  $A^T A \mathbf{x} = A^T \mathbf{b}$  gives the parameters that best fit the data. These values can be understood as defining the hyperplane that best fits the data.



**Figure 22.3:** Solving the linear regression problem results in a best-fit hyperplane.

#### Problem 22.4: U

ing your function from Problem 22, solve the linear regression problem specified by the data contained in the file<sup>a</sup> `linregression.txt`. This is a whitespace-delimited text file formatted so that the  $i$ -th row consists of  $y_i, x_{i,1}, \dots, x_{i,n}$ . Use `np.loadtxt()` to load in the data and return the solution to the normal equations.

<sup>a</sup>Source: Statistical Reference Datasets website at <http://www.itl.nist.gov/div898/strd/lls/data/LINKS/v-Longley.shtml>.

## Logistic Regression

*Logistic regression* is another important technique in statistical analysis and machine learning that builds off of the concepts of linear regression. As in linear regression, there is a set of predictor variables  $\{x_{i,1}, x_{i,2}, \dots, x_{i,n}\}_{i=1}^m$  with corresponding outcome variables  $\{y_i\}_{i=1}^m$ . In logistic regression, the outcome variables  $y_i$  are binary and can be modeled by a *sigmoidal* relationship. The value of the predicted  $y_i$  can be thought of as the probability that  $y_i = 1$ . In mathematical terms,

$$\mathbb{P}(y_i = 1 | x_{i,1}, \dots, x_{i,n}) = p_i,$$

where

$$p_i = \frac{1}{1 + \exp(-(\beta_0 + \beta_1 x_{i,1} + \dots + \beta_n x_{i,n}))}.$$

The parameters of the model are the real numbers  $\beta_0, \beta_1, \dots, \beta_n$ . Note that  $p_i \in (0, 1)$  regardless of the values of the predictor variables and parameters.

The probability of observing the outcome variables  $y_i$  under this model, assuming they are independent, is given by the *likelihood function*  $\mathcal{L} : \mathbb{R}^{n+1} \rightarrow \mathbb{R}$

$$\mathcal{L}(\beta_0, \dots, \beta_n) = \prod_{i=1}^m p_i^{y_i} (1 - p_i)^{1-y_i}.$$

The goal of logistic regression is to find the parameters  $\beta_0, \dots, \beta_k$  that maximize this likelihood function. Thus, the problem can be written as:

$$\max_{(\beta_0, \dots, \beta_n)} \mathcal{L}(\beta_0, \dots, \beta_n).$$

Maximizing this function is often a numerically unstable calculation. Thus, to make the objective function more suitable, the logarithm of the objective function may be maximized because the logarithmic function is strictly monotone increasing. Taking the log and turning the problem into a minimization problem, the final problem is formulated as:

$$\min_{(\beta_0, \dots, \beta_n)} -\log \mathcal{L}(\beta_0, \dots, \beta_n).$$

A few lines of calculation reveal that this objective function can also be rewritten as

$$\begin{aligned} -\log \mathcal{L}(\beta_0, \dots, \beta_n) &= \sum_{i=1}^m \log(1 + \exp(-(\beta_0 + \beta_1 x_{i,1} + \dots + \beta_n x_{i,n}))) + \\ &\quad \sum_{i=1}^m (1 - y_i)(\beta_0 + \beta_1 x_{i,1} + \dots + \beta_n x_{i,n}). \end{aligned}$$

The values for the parameters  $\{\beta_i\}_{i=1}^n$  that we obtain are known as the *maximum likelihood estimate* (MLE). To find the MLE, conjugate gradient can be used to minimize the objective function.

For a one-dimensional binary logistic regression problem, we have predictor data  $\{x_i\}_{i=1}^m$  with labels  $\{y_i\}_{i=1}^m$  where each  $y_i \in \{0, 1\}$ . The negative log likelihood then becomes the following.

$$-\log \mathcal{L}(\beta_0, \beta_1) = \sum_{i=1}^m \log(1 + e^{-(\beta_0 + \beta_1 x_i)}) + (1 - y_i)(\beta_0 + \beta_1 x_i) \tag{22.4}$$

**Problem 22.5: W**

ite a class for doing binary logistic regression in one dimension that implement the following methods.

1. `fit()`: accept an array  $\mathbf{x} \in \mathbb{R}^n$  of data, an array  $\mathbf{y} \in \mathbb{R}^n$  of labels (0s and 1s), and an initial guess  $\beta_0 \in \mathbb{R}^2$ . Define the negative log likelihood function as given in (22.4), then minimize it (with respect to  $\beta$ ) with your function from Problem 22 or `opt.fmin_cg()`. Store the resulting parameters  $\beta_0$  and  $\beta_1$  as attributes.
2. `predict()`: accept a float  $x \in \mathbb{R}$  and calculate

$$\sigma(x) = \frac{1}{1 + \exp(-(\beta_0 + \beta_1 x))},$$

where  $\beta_0$  and  $\beta_1$  are the optimal values calculated in `fit()`. The value  $\sigma(x)$  is the probability that the observation  $x$  should be assigned the label  $y = 1$ .

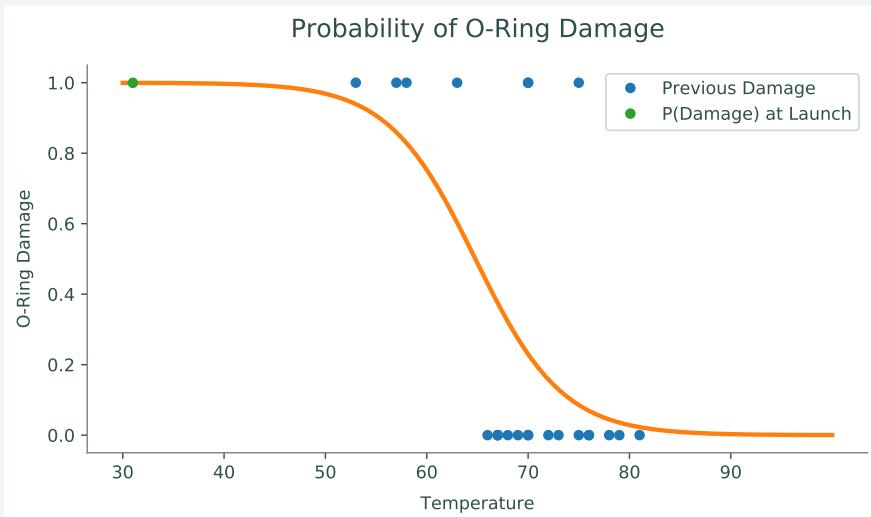
This class does not need an explicit constructor. You may assume that `predict()` will be called after `fit()`.

### Problem 22.6: O

January 28, 1986, less than two minutes into the Challenger space shuttle's 10th mission, there was a large explosion that originated from the spacecraft, killing all seven crew members and destroying the shuttle. The investigation that followed concluded that the malfunction was caused by damage to O-rings that are used as seals for parts of the rocket engines. There were 24 space shuttle missions before this disaster, some of which had noted some O-ring damage. Given the data, could this disaster have been predicted?

The file `challenger.npy` contains data for 23 missions (during one of the 24 missions, the engine was lost at sea). The first column ( $x$ ) contains the ambient temperature, in Fahrenheit, of the shuttle launch. The second column ( $y$ ) contains a binary indicator of the presence of O-ring damage (1 if O-ring damage was present, 0 otherwise).

Instantiate your class from Problem 22 and fit it to the data, using an initial guess of  $\beta_0 = [20, -1]^T$ . Plot the resulting curve  $\sigma(x)$  for  $x \in [30, 100]$ , along with the raw data. Return the predicted probability (according to this model) of O-ring damage on the day the shuttle was launched, given that it was 31°F.





# 23. Interior Point 1: Linear Programs

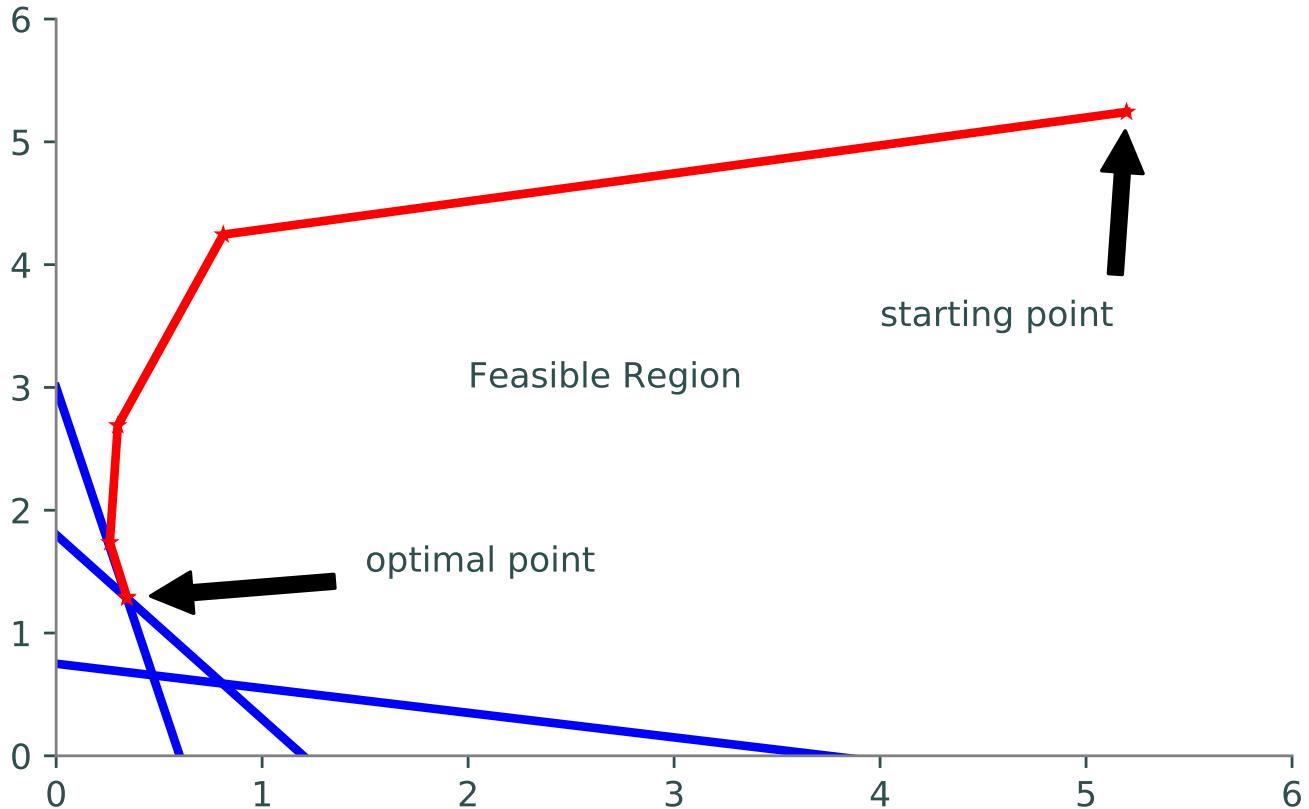
---

**Lab Objective:** *For decades after its invention, the Simplex algorithm was the only competitive method for linear programming. The past 30 years, however, have seen the discovery and widespread adoption of a new family of algorithms that rival—and in some cases outperform—the Simplex algorithm, collectively called Interior Point methods. One of the major shortcomings of the Simplex algorithm is that the number of steps required to solve the problem can grow exponentially with the size of the linear system. Thus, for certain large linear programs, the Simplex algorithm is simply not viable. Interior Point methods offer an alternative approach and enjoy much better theoretical convergence properties. In this lab we implement an Interior Point method for linear programs, and in the next lab we will turn to the problem of solving quadratic programs.*

## Introduction

---

Recall that a linear program is a constrained optimization problem with a linear objective function and linear constraints. The linear constraints define a set of allowable points called the *feasible region*, the boundary of which forms a geometric object known as a *polytope*. The theory of convex optimization ensures that the optimal point for the objective function can be found among the vertices of the feasible polytope. The Simplex Method tests a sequence of such vertices until it finds the optimal point. Provided the linear program is neither unbounded nor infeasible, the algorithm is certain to produce the correct answer after a finite number of steps, but it does not guarantee an efficient path along the polytope toward the minimizer. Interior point methods do away with the feasible polytope and instead generate a sequence of points that cut through the interior (or exterior) of the feasible region and converge iteratively to the optimal point. Although it is computationally more expensive to compute such interior points, each step results in significant progress toward the minimizer. See Figure 23.1 for an example of a path using an Interior Point algorithm. In general, the Simplex Method requires many more iterations (though each iteration is less expensive computationally).



**Figure 23.1:** A path traced by an Interior Point algorithm.

## Primal-Dual Interior Point Methods

---

Some of the most popular and successful types of Interior Point methods are known as Primal-Dual Interior Point methods. Consider the following linear program:

$$\begin{array}{ll} \text{minimize} & \mathbf{c}^T \mathbf{x} \\ \text{subject to} & A\mathbf{x} = \mathbf{b} \\ & \mathbf{x} \succeq 0. \end{array}$$

Here,  $\mathbf{x}, \mathbf{c} \in \mathbb{R}^n$ ,  $\mathbf{b} \in \mathbb{R}^m$ , and  $A \in \mathbb{R}^{m \times n}$  with full row rank. This is the *primal* problem, and its *dual* takes the form:

$$\begin{array}{ll} \text{maximize} & \mathbf{b}^T \boldsymbol{\lambda} \\ \text{subject to} & A^T \boldsymbol{\lambda} + \boldsymbol{\mu} = \mathbf{c} \\ & \boldsymbol{\mu}, \boldsymbol{\lambda} \succeq 0, \end{array}$$

where  $\boldsymbol{\lambda} \in \mathbb{R}^m$  and  $\boldsymbol{\mu} \in \mathbb{R}^n$ .

## KKT Conditions

---

The theory of convex optimization gives us necessary and sufficient conditions for the solutions to the primal and dual problems via the Karush-Kuhn-Tucker (KKT) conditions. The Lagrangian for the primal problem is as follows:

$$\mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) = \mathbf{c}^\top \mathbf{x} + \boldsymbol{\lambda}^\top (\mathbf{b} - A\mathbf{x}) - \boldsymbol{\mu}^\top \mathbf{x}$$

The KKT conditions are

$$\begin{aligned} A^\top \boldsymbol{\lambda} + \boldsymbol{\mu} &= \mathbf{c} \\ A\mathbf{x} &= \mathbf{b} \\ x_i \mu_i &= 0, \quad i = 1, 2, \dots, n, \\ \mathbf{x}, \boldsymbol{\mu} &\succeq 0. \end{aligned}$$

It is convenient to write these conditions in a more compact manner, by defining an almost-linear function  $F$  and setting it equal to zero:

$$F(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\mu}) := \begin{bmatrix} A^\top \boldsymbol{\lambda} + \boldsymbol{\mu} - \mathbf{c} \\ A\mathbf{x} - \mathbf{b} \\ M\mathbf{x} \end{bmatrix} = 0, \quad (\mathbf{x}, \boldsymbol{\mu} \succeq 0),$$

where  $M = \text{diag}(\mu_1, \mu_2, \dots, \mu_n)$ . Note that the first row of  $F$  is the KKT condition for dual feasibility, the second row of  $F$  is the KKT condition for the primal problem, and the last row of  $F$  accounts for complementary slackness.

### Problem 23.1: D

Define a function `interiorPoint()` that will be used to solve the complete interior point problem. This function should accept  $A$ ,  $\mathbf{b}$ , and  $\mathbf{c}$  as parameters, along with the keyword arguments `niter=20` and `tol=1e-16`. The keyword arguments will be used in a later problem. For this problem, within the `interiorPoint()` function, write a function for the vector-valued function  $F$  described above. This function should accept  $\mathbf{x}$ ,  $\boldsymbol{\lambda}$ , and  $\boldsymbol{\mu}$  as parameters and return a 1-dimensional NumPy array with  $2n + m$  entries.

## Search Direction

---

A Primal-Dual Interior Point method is a line search method that starts with an initial guess  $(\mathbf{x}_0^\top, \boldsymbol{\lambda}_0^\top, \boldsymbol{\mu}_0^\top)$  and produces a sequence of points that converge to  $(\mathbf{x}^*{}^\top, \boldsymbol{\lambda}^*{}^\top, \boldsymbol{\mu}^*{}^\top)$ , the solution to the KKT equations and hence the solution to the original linear program. The constraints on the problem make finding a search direction and step length a little more complicated than for the unconstrained line search we have studied previously.

In the spirit of Newton's Method, we can form a linear approximation of the system  $F(\mathbf{x}, \lambda, \mu) = 0$  centered around our current point  $(\mathbf{x}, \lambda, \mu)$ , and calculate the direction  $(\Delta\mathbf{x}^T, \Delta\lambda^T, \Delta\mu^T)$  in which to step to set the linear approximation equal to 0. This equates to solving the linear system:

$$DF(\mathbf{x}, \lambda, \mu) \begin{bmatrix} \Delta\mathbf{x} \\ \Delta\lambda \\ \Delta\mu \end{bmatrix} = -F(\mathbf{x}, \lambda, \mu) \quad (23.1)$$

Here  $DF(\mathbf{x}, \lambda, \mu)$  denotes the total derivative matrix of  $F$ . We can calculate this matrix block-wise by obtaining the partial derivatives of each block entry of  $F(\mathbf{x}, \lambda, \mu)$  with respect to  $\mathbf{x}$ ,  $\lambda$ , and  $\mu$ , respectively. We thus obtain:

$$DF(\mathbf{x}, \lambda, \mu) = \begin{bmatrix} 0 & A^T & I \\ A & 0 & 0 \\ M & 0 & X \end{bmatrix}$$

where  $X = \text{diag}(x_1, x_2, \dots, x_n)$ .

Unfortunately, solving Equation 23.1 often leads to a search direction that is too greedy. Even small steps in this direction may lead the iteration out of the feasible region by violating one of the constraints. To remedy this, we define the *duality measure*  $v^1$  of the problem:

$$v = \frac{\mathbf{x}^T \boldsymbol{\mu}}{n}$$

The idea is to use Newton's method to identify a direction that strictly decreases  $v$ . Thus instead of solving Equation 23.1, we solve:

$$DF(\mathbf{x}, \lambda, \mu) \begin{bmatrix} \Delta\mathbf{x} \\ \Delta\lambda \\ \Delta\mu \end{bmatrix} = -F(\mathbf{x}, \lambda, \mu) + \begin{bmatrix} 0 \\ 0 \\ \sigma v \mathbf{e} \end{bmatrix} \quad (23.2)$$

where  $\mathbf{e} = (1, 1, \dots, 1)^T$  and  $\sigma \in [0, 1]$  is called the *centering parameter*. The closer  $\sigma$  is to 0, the more similar the resulting direction will be to the plain Newton direction. The closer  $\sigma$  is to 1, the more the direction points inward to the interior of the feasible region.

### Problem 23.2: W

thin `interiorPoint()`, write a subroutine to compute the search direction  $(\Delta\mathbf{x}^T, \Delta\lambda^T, \Delta\mu^T)$  by solving Equation 23.2. Use  $\sigma = \frac{1}{10}$  for the centering parameter.

Note that only the last block row of  $DF$  will need to be changed at each iteration (since  $M$  and  $X$  depend on  $\mu$  and  $\mathbf{x}$ , respectively). Use the functions `lu_factor()` and `lu_solve()` from the `scipy.linalg` module to solving the system of equations efficiently.

<sup>1</sup> $v$  is the Greek letter for  $n$ , pronounced “nu.”

## Step Length

---

Now that we have our search direction, it remains to choose our step length. We wish to step nearly as far as possible without violating the problem's constraints, thus remaining in the interior of the feasible region. First, we calculate the maximum allowable step lengths for  $\mathbf{x}$  and  $\mu$ , respectively:

$$\begin{aligned}\alpha_{\max} &= \min\{-\mu_i/\Delta\mu_i \mid \Delta\mu_i < 0\} \\ \delta_{\max} &= \min\{-x_i/\Delta x_i \mid \Delta x_i < 0\}\end{aligned}$$

If all values of  $\Delta\mu$  are nonnegative, let  $\alpha_{\max} = 1$ . Likewise, if all values of  $\Delta x$  are nonnegative, let  $\delta_{\max} = 1$ . Next, we back off from these maximum step lengths slightly:

$$\begin{aligned}\alpha &= \min(1, 0.95\alpha_{\max}) \\ \delta &= \min(1, 0.95\delta_{\max}).\end{aligned}$$

These are our final step lengths. Thus, the next point in the iteration is given by:

$$\begin{aligned}\mathbf{x}_{k+1} &= \mathbf{x}_k + \delta \Delta \mathbf{x}_k \\ (\lambda_{k+1}, \mu_{k+1}) &= (\lambda_k, \mu_k) + \alpha(\Delta\lambda_k, \Delta\mu_k).\end{aligned}$$

### Problem 23.3: W

*thin interiorPoint(), write a subroutine to compute the step size after the search direction has been computed. Avoid using loops when computing  $\alpha_{\max}$  and  $\beta_{\max}$  (use masking and NumPy functions instead).*

## Initial Point

---

Finally, the choice of initial point  $(\mathbf{x}_0, \lambda_0, \mu_0)$  is an important, nontrivial one. A naïvely or randomly chosen initial point may cause the algorithm to fail to converge. The following function will calculate an appropriate initial point.

```
def starting_point(A, b, c):
    """Calculate an initial guess to the solution of the linear program
    min c\trp x, Ax = b, x>=0.
    Reference: Nocedal and Wright, p. 410.
    """
    # Calculate x, lam, mu of minimal norm satisfying both
    # the primal and dual constraints.
    B = la.inv(A @ A.T)
    x = A.T @ B @ b
    lam = B @ A @ c
```

```
mu = c - (A.T @ lam)

# Perturb x and s so they are nonnegative.
dx = max((-3./2)*x.min(), 0)
dmu = max((-3./2)*mu.min(), 0)
x += dx*np.ones_like(x)
mu += dmu*np.ones_like(mu)

# Perturb x and mu so they are not too small and not too dissimilar.
dx = .5*(x*mu).sum()/mu.sum()
dmu = .5*(x*mu).sum()/x.sum()
x += dx*np.ones_like(x)
mu += dmu*np.ones_like(mu)

return x, lam, mu
```

### Problem 23.4: C

Complete the implementation of `interiorPoint()`.

Use the function `starting_point()` provided above to select an initial point, then run the iteration `niter` times, or until the duality measure is less than `tol`. Return the optimal point  $\mathbf{x}^*$  and the optimal value  $\mathbf{c}^\top \mathbf{x}^*$ .

The duality measure  $v$  tells us in some sense how close our current point is to the minimizer. The closer  $v$  is to 0, the closer we are to the optimal point. Thus, by printing the value of  $v$  at each iteration, you can track how your algorithm is progressing and detect when you have converged.

To test your implementation, use the following code to generate a random linear program, along with the optimal solution.

```
def randomLP():
    """Generate a linear program min c\trp x s.t. Ax = b, x>=0.
    First generate m feasible constraints, then add
    slack variables to convert it into the above form.

    Inputs:
        m (int >= n): number of desired constraints.
        n (int): dimension of space in which to optimize.

    Outputs:
        A ((m,n+m) ndarray): Constraint matrix.
        b ((m,) ndarray): Constraint vector.
        c ((n+m,), ndarray): Objective function with m trailing 0s.
        x ((n,) ndarray): The first 'n' terms of the solution to the LP.

    """
    A = np.random.random((m,n))*20 - 10
    A[A[:, -1] < 0] *= -1
    x = np.random.random(n)*10
    b = np.zeros(m)
    b[:n] = A[:, :n] @ x
    b[n:] = A[:, n:] @ x + np.random.random(m-n)*10
    c = np.zeros(n+m)
    c[:n] = A[:, :n].sum(axis=0)/n
    A = np.hstack((A, np.eye(m)))
    return A, b, -c, x
```

```
>>> m, n = 7, 5
>>> A, b, c, x = randomLP(m, n)
>>> point, value = interiorPoint(A, b, c)
>>> np.allclose(x, point[:n])
True
```

## Least Absolute Deviations (LAD)

---

We now return to the familiar problem of fitting a line (or hyperplane) to a set of data. We have previously approached this problem by minimizing the sum of the squares of the errors between the data points and the line, an approach known as *least squares*. The least squares solution can be obtained analytically when fitting a linear function, or through a number of optimization methods (such as Conjugate Gradient) when fitting a nonlinear function.

The method of *least absolute deviations* (LAD) also seeks to find a best fit line to a set of data, but the error between the data and the line is measured differently. In particular, suppose we have a set of data points  $(y_1, \mathbf{x}_1), (y_2, \mathbf{x}_2), \dots, (y_m, \mathbf{x}_m)$ , where  $y_i \in \mathbb{R}$ ,  $\mathbf{x}_i \in \mathbb{R}^n$  for  $i = 1, 2, \dots, m$ . Here, the  $\mathbf{x}_i$  vectors are the *explanatory variables* and the  $y_i$  values are the *response variables*, and we assume the following linear model:

$$y_i = \beta^\top \mathbf{x}_i + b, \quad i = 1, 2, \dots, m,$$

where  $\beta \in \mathbb{R}^n$  and  $b \in \mathbb{R}$ . The error between the data and the proposed linear model is given by

$$\sum_{i=1}^n |\beta^\top \mathbf{x}_i + b - y_i|,$$

and we seek to choose the parameters  $\beta, b$  so as to minimize this error.

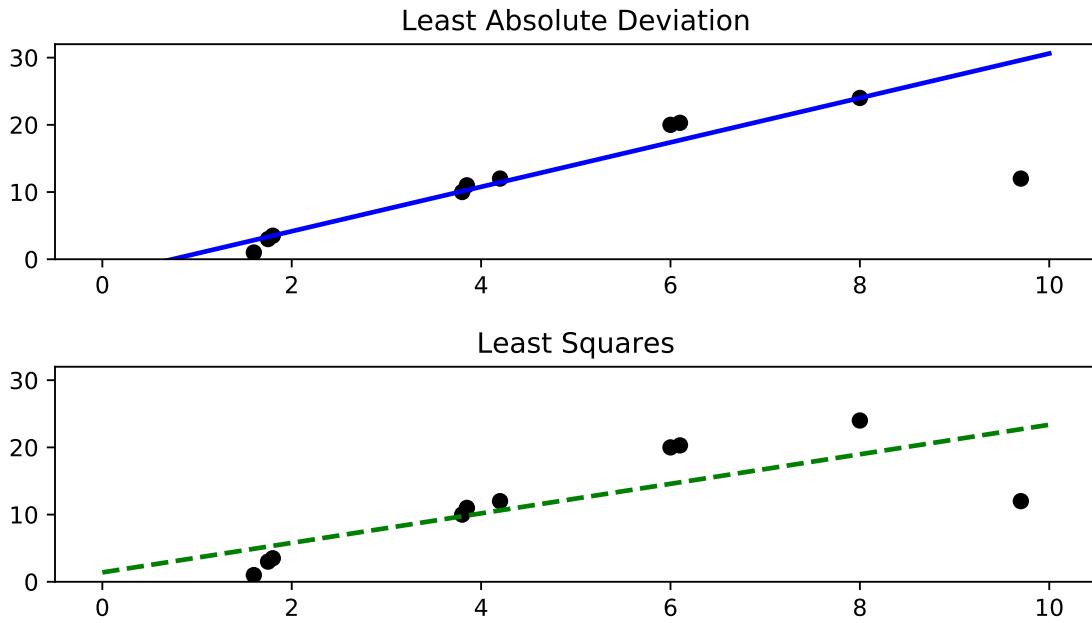
### Advantages of LAD

---

The most prominent difference between this approach and least squares is how they respond to outliers in the data. Least absolute deviations is robust in the presence of outliers, meaning that one (or a few) errant data points won't severely affect the fitted line. Indeed, in most cases, the best fit line is guaranteed to pass through at least two of the data points. This is a desirable property when the outliers may be ignored (perhaps because they are due to measurement error or corrupted data). Least squares, on the other hand, is much more sensitive to outliers, and so is the better choice when outliers cannot be dismissed. See Figure 23.2.

While least absolute deviations is robust with respect to outliers, small horizontal perturbations of the data points can lead to very different fitted lines. Hence, the least absolute deviations solution is less stable than the least squares solution. In some cases there are even infinitely many lines that minimize the least absolute deviations error term. However, one can expect a unique solution in most cases.

The least absolute deviations solution arises naturally when we assume that the residual terms  $\beta^\top \mathbf{x}_i + b - y_i$  have a particular statistical distribution (the Laplace distribution). Ultimately, however, the choice between least absolute deviations and least squares depends on the nature of the data at hand, as well as your own good judgment.



**Figure 23.2:** Fitted lines produced by least absolute deviations (top) and least squares (bottom). The presence of an outlier accounts for the stark difference between the two lines.

## LAD as a Linear Program

---

We can formulate the least absolute deviations problem as a linear program, and then solve it using our interior point method. For  $i = 1, 2, \dots, m$  we introduce the artificial variable  $u_i$  to take the place of the error term  $|\beta^\top \mathbf{x}_i + b - y_i|$ , and we require this variable to satisfy  $u_i \geq |\beta^\top \mathbf{x}_i + b - y_i|$ . This constraint is not yet linear, but we can split it into an equivalent set of two linear constraints:

$$\begin{aligned} u_i &\geq \beta^\top \mathbf{x}_i + b - y_i, \\ u_i &\geq y_i - \beta^\top \mathbf{x}_i - b. \end{aligned}$$

The  $u_i$  are implicitly constrained to be nonnegative.

Our linear program can now be stated as follows:

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^m u_i \\ \text{subject to} \quad & u_i \geq \beta^\top \mathbf{x}_i + b - y_i, \\ & u_i \geq y_i - \beta^\top \mathbf{x}_i - b. \end{aligned}$$

Now for each inequality constraint, we bring all variables  $(u_i, \beta, b)$  to the left hand side and introduce a nonnegative slack variable to transform the constraint into an equality:

$$\begin{aligned} u_i - \beta^\top \mathbf{x}_i - b - s_{2i-1} &= -y_i, \\ u_i + \beta^\top \mathbf{x}_i + b - s_{2i} &= y_i, \\ s_{2i-1}, s_{2i} &\geq 0. \end{aligned}$$

Notice that the variables  $\beta, b$  are not assumed to be nonnegative, but in our interior point method, all variables are assumed to be nonnegative. We can fix this situation by writing these variables as the difference of nonnegative variables:

$$\begin{aligned}\beta &= \beta_1 - \beta_2, \\ b &= b_1 - b_2, \\ \beta_1, \beta_2 &\succeq 0; b_1, b_2 \geq 0.\end{aligned}$$

Substituting these values into our constraints, we have the following system of constraints:

$$\begin{aligned}u_i - \beta_1^\top \mathbf{x}_i + \beta_2^\top \mathbf{x}_i - b_1 + b_2 - s_{2i-1} &= -y_i, \\ u_i + \beta_1^\top \mathbf{x}_i - \beta_2^\top \mathbf{x}_i + b_1 - b_2 - s_{2i} &= y_i, \\ \beta_1, \beta_2 &\succeq 0; u_i, b_1, b_2, s_{2i-1}, s_{2i} \geq 0.\end{aligned}$$

Writing  $\mathbf{y} = (-y_1, y_1, -y_2, y_2, \dots, -y_m, y_m)^\top$  and  $\beta_i = (\beta_{i,1}, \dots, \beta_{i,n})^\top$  for  $i = \{1, 2\}$ , we can aggregate all of our variables into one vector as follows:

$$\mathbf{v} = (u_1, \dots, u_m, \beta_{1,1}, \dots, \beta_{1,n}, \beta_{2,1}, \dots, \beta_{2,n}, b_1, b_2, s_1, \dots, s_{2m})^\top.$$

Defining  $\mathbf{c} = (1, 1, \dots, 1, 0, \dots, 0)^\top$  (where only the first  $m$  entries are equal to 1), we can write our objective function as

$$\sum_{i=1}^m u_i = \mathbf{c}^\top \mathbf{v}.$$

Hence, the final form of our linear program is:

$$\begin{aligned}&\text{minimize} && \mathbf{c}^\top \mathbf{v} \\ &\text{subject to} && A\mathbf{v} = \mathbf{y}, \\ & && \mathbf{v} \succeq 0,\end{aligned}$$

where  $A$  is a matrix containing the coefficients of the constraints. Our constraints are now equalities, and the variables are all nonnegative, so we are ready to use our interior point method to obtain the solution.

## LAD Example

---

Consider the following example. We start with an array  $\text{data}$ , each row of which consists of the values  $y_i, x_{i,1}, \dots, x_{i,n}$ , where  $\mathbf{x}_i = (x_{i,1}, x_{i,2}, \dots, x_{i,n})^\top$ . We will have  $3m + 2(n+1)$  variables in our linear program. Below, we initialize the vectors  $\mathbf{c}$  and  $\mathbf{y}$ .

```
>>> m = data.shape[0]
>>> n = data.shape[1] - 1
>>> c = np.zeros(3*m + 2*(n + 1))
>>> c[:m] = 1
>>> y = np.empty(2*m)
>>> y[::2] = -data[:, 0]
>>> y[1::2] = data[:, 0]
>>> x = data[:, 1:]
```

The hardest part is initializing the constraint matrix correctly. It has  $2m$  rows and  $3m + 2(n+1)$  columns. Try writing out the constraint matrix by hand for small  $m, n$ , and make sure you understand why the code below is correct.

```
>>> A = np.ones((2*m, 3*m + 2*(n + 1)))
>>> A[::2, :m] = np.eye(m)
>>> A[1::2, :m] = np.eye(m)
>>> A[::2, m:m+n] = -x
>>> A[1::2, m:m+n] = x
>>> A[::2, m+n:m+2*n] = x
>>> A[1::2, m+n:m+2*n] = -x
>>> A[::2, m+2*n] = -1
>>> A[1::2, m+2*n+1] = -1
>>> A[:, m+2*n+2:] = -np.eye(2*m, 2*m)
```

Now we can calculate the solution by calling our interior point function.

```
>>> sol = interiorPoint(A, y, c, niter=10)[0]
```

However, the variable `sol` holds the value for the vector

$$\mathbf{v} = (u_1, \dots, u_m, \beta_{1,1}, \dots, \beta_{1,n}, \beta_{2,1}, \dots, \beta_{2,n}, b_1, b_2, s_1, \dots, s_{2m+1})^T.$$

We extract values of  $\beta = \beta_1 - \beta_2$  and  $b = b_1 - b_2$  with the following code:

```
>>> beta = sol[m:m+n] - sol[m+n:m+2*n]
>>> b = sol[m+2*n] - sol[m+2*n+1]
```

### Problem 23.5: T

The file `simdata.txt` contains two columns of data. The first gives the values of the response variables ( $y_i$ ), and the second column gives the values of the explanatory variables ( $x_i$ ). Find the least absolute deviations line for this data set, and plot it together with the data. Plot the least squares solution as well to compare the results.

```
>>> from scipy.stats import linregress
>>> slope, intercept = linregress(data[:,1], data[:,0])[:2]
>>> domain = np.linspace(0,10,200)
>>> plt.plot(domain, domain*slope + intercept)
```



# 24. Interior Point 2: Quadratic Programs

---

**Lab Objective:** *Interior point methods originated as an alternative to the Simplex method for solving linear optimization problems. However, they can also be adapted to treat convex optimization problems in general. In this lab we implement a primal-dual Interior Point method for convex quadratic constrained optimization and explore applications in elastic membrane theory and finance.*

## Quadratic Optimization Problems

---

A *quadratic constrained optimization problem* differs from a linear constrained optimization problem only in that the objective function is quadratic rather than linear. We can pose such a problem as follows:

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2} \mathbf{x}^T Q \mathbf{x} + \mathbf{c}^T \mathbf{x} \\ \text{subject to} \quad & A \mathbf{x} \succeq \mathbf{b}, \\ & G \mathbf{x} = \mathbf{h}. \end{aligned}$$

We will restrict our attention to quadratic programs involving positive semidefinite quadratic terms (in general, indefinite quadratic objective functions admit many local minima, complicating matters considerably). Such problems are called *convex*, since the objective function is convex. To simplify the exposition, we will also only allow inequality constraints (generalizing to include equality constraints is not difficult). Thus, we have the problem

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2} \mathbf{x}^T Q \mathbf{x} + \mathbf{c}^T \mathbf{x} \\ \text{subject to} \quad & A \mathbf{x} \succeq \mathbf{b} \end{aligned}$$

where  $Q \in \mathbb{R}^{n \times n}$  is a positive semidefinite matrix,  $A \in \mathbb{R}^{m \times n}$ ,  $\mathbf{x}, \mathbf{c} \in \mathbb{R}^n$ , and  $\mathbf{b} \in \mathbb{R}^m$ .

The Lagrangian function for this problem is:

$$\mathcal{L}(\mathbf{x}, \boldsymbol{\mu}) = \frac{1}{2} \mathbf{x}^T Q \mathbf{x} + \mathbf{c}^T \mathbf{x} - \boldsymbol{\mu}^T (A \mathbf{x} - \mathbf{b}), \quad (24.1)$$

where  $\boldsymbol{\mu} \in \mathbb{R}^m$  is the Lagrange multiplier.

We also introduce a nonnegative slack vector  $\mathbf{y} \in \mathbb{R}^m$  to change the inequality  $A \mathbf{x} - \mathbf{b} \succeq 0$  into the equality  $A \mathbf{x} - \mathbf{b} - \mathbf{y} = 0$ .

Then the complete set of KKT conditions are:

$$\begin{aligned} Q\mathbf{x} - A^T \boldsymbol{\mu} + \mathbf{c} &= 0, \\ A\mathbf{x} - \mathbf{b} - \mathbf{y} &= 0, \\ y_i \mu_i &= 0, \quad i = 1, 2, \dots, m, \\ \mathbf{y}, \boldsymbol{\mu} &\succeq 0. \end{aligned}$$

## Quadratic Interior Point Method

---

The Interior Point method we describe here is an adaptation of the method we used with linear programming. Define  $Y = \text{diag}(y_1, y_2, \dots, y_m)$ ,  $M = \text{diag}(\mu_1, \mu_2, \dots, \mu_m)$ , and let  $\mathbf{e} \in \mathbb{R}^m$  be a vector of all ones. Then the roots of the function

$$F(\mathbf{x}, \mathbf{y}, \boldsymbol{\mu}) = \begin{bmatrix} Q\mathbf{x} - A^T \boldsymbol{\mu} + \mathbf{c} \\ A\mathbf{x} - \mathbf{b} - \mathbf{y} \\ YM\mathbf{e} \end{bmatrix} = 0,$$

$$(\mathbf{y}, \boldsymbol{\mu}) \succeq 0$$

satisfy the KKT conditions. The derivative matrix of this function is given by

$$DF(\mathbf{x}, \mathbf{y}, \boldsymbol{\mu}) = \begin{bmatrix} Q & 0 & -A^T \\ A & -I & 0 \\ 0 & M & Y \end{bmatrix},$$

and the duality measure  $v$  for this problem is

$$v = \frac{\mathbf{y}^T \boldsymbol{\mu}}{m}.$$

## Search Direction

---

We calculate the search direction for this algorithm in the spirit of Newton's Method; this is the same way that we did in the linear programming case. That is, we solve the system:

$$DF(\mathbf{x}, \mathbf{y}, \boldsymbol{\mu}) \begin{bmatrix} \Delta \mathbf{x} \\ \Delta \mathbf{y} \\ \Delta \boldsymbol{\mu} \end{bmatrix} = -F(\mathbf{x}, \mathbf{y}, \boldsymbol{\mu}) + \begin{bmatrix} 0 \\ 0 \\ \sigma v \mathbf{e} \end{bmatrix}, \quad (24.2)$$

where  $\sigma \in [0, 1]$  is the centering parameter.

**Problem 24.1: C**

Create a function `qInteriorPoint()`. It should accept the arrays  $Q, \mathbf{c}, A$ , and  $\mathbf{b}$ , a tuple of arrays `guess` giving initial estimates for  $\mathbf{x}, \mathbf{y}$ , and  $\mu$  (this will be explained later), along with the keyword arguments `niter=20` and `tol=1e-16`.

In this function, calculate the search direction. Create  $F$  and  $DF$  as described above, and calculate the search direction  $(\Delta\mathbf{x}^T, \Delta\mathbf{y}^T, \Delta\mu^T)$  by solving Equation 24.2. Use  $\sigma = \frac{1}{10}$  for the centering parameter.

(Hint: What are the dimensions of  $F$  and  $DF$ ?)

## Step Length

---

Now that we have our search direction, we select a step length. We want to step nearly as far as possible without violating the nonnegativity constraints. However, we back off slightly from the maximum allowed step length because an overly greedy step at one iteration may prevent a descent step at the next iteration. Thus, we choose our step size

$$\alpha = \max\{\alpha \in (0, 1] \mid \tau(\mathbf{y}, \mu) + \alpha(\Delta\mathbf{y}, \Delta\mu) \succeq 0\},$$

where  $\tau \in (0, 1)$  controls how much we back off from the maximal step length. For now, choose  $\tau = 0.95$ . In general,  $\tau$  can be made to approach 1 at each successive iteration. This may speed up convergence in some cases.

We wish to step nearly as far as possible without violating the problem's constraints, as to remain in the interior of the feasible region. First, we calculate the maximum allowable step lengths for  $\mu$  and  $\mathbf{y}$ .

$$\begin{aligned}\beta_{\max} &= \min\{-\mu_i/\Delta\mu_i \mid \Delta\mu_i < 0\} \\ \delta_{\max} &= \min\{-y_i/\Delta y_i \mid \Delta y_i < 0\}\end{aligned}$$

If all of the entries of  $\Delta\mu$  are nonnegative, we let  $\beta_{\max} = 1$ . Likewise, if all the entries of  $\Delta\mathbf{y}$  are nonnegative, let  $\delta_{\max} = 1$ . Next, we back off from these maximum step lengths slightly:

$$\begin{aligned}\beta &= \min(1, \tau\beta_{\max}) \\ \delta &= \min(1, \tau\delta_{\max}) \\ \alpha &= \min(\beta, \delta)\end{aligned}$$

This  $\alpha$  is our final step length. Thus, the next point in the iteration is given by:

$$(\mathbf{x}_{k+1}, \mathbf{y}_{k+1}, \mu_{k+1}) = (\mathbf{x}_k, \mathbf{y}_k, \mu_k) + \alpha(\Delta\mathbf{x}_k, \Delta\mathbf{y}_k, \Delta\mu_k).$$

This completes one iteration of the algorithm.

## Initial Point

---

The starting point  $(\mathbf{x}_0, \mathbf{y}_0, \mu_0)$  has an important effect on the convergence of the algorithm. The code listed below will calculate an appropriate starting point:

```

def startingPoint(G, c, A, b, guess):
    """
    Obtain an appropriate initial point for solving the QP
    .5 x\trp Gx + x\trp c s.t. Ax >= b.

    Parameters:
        G -- symmetric positive semidefinite matrix shape (n,n)
        c -- array of length n
        A -- constraint matrix shape (m,n)
        b -- array of length m
        guess -- a tuple of arrays (x, y, l) of lengths n, m, and m, resp.

    Returns:
        a tuple of arrays (x0, y0, l0) of lengths n, m, and m, resp.
    """
    m,n = A.shape
    x0, y0, l0 = guess

    # initialize linear system
    N = np.zeros((n+m+m, n+m+m))
    N[:n,:n] = G
    N[:n, n+m:] = -A.T
    N[n:n+m, :n] = A
    N[n:n+m, n:n+m] = -np.eye(m)
    N[n+m:, n:n+m] = np.diag(l0)
    N[n+m:, n+m:] = np.diag(y0)
    rhs = np.empty(n+m+m)
    rhs[:n] = -(G.dot(x0) - A.T.dot(l0)+c)
    rhs[n:n+m] = -(A.dot(x0) - y0 - b)
    rhs[n+m:] = -(y0*l0)

    sol = la.solve(N, rhs)
    dx = sol[:n]
    dy = sol[n:n+m]
    dl = sol[n+m:]

    y0 = np.maximum(1, np.abs(y0 + dy))
    l0 = np.maximum(1, np.abs(l0+dl))

    return x0, y0, l0

```

Notice that we still need to provide a tuple of arrays `guess` as an argument. Do your best to provide a reasonable guess for the array `x`, and we suggest setting `y` and `mu` equal to arrays of ones. We summarize the entire algorithm below.

---

```

1: procedure INTERIOR POINT METHOD FOR QP
2:   Choose initial point  $(\mathbf{x}_0, \mathbf{y}_0, \boldsymbol{\mu}_0)$ .
3:   while  $k < \text{nitors}$  and  $v < \text{tol}$ : do
4:     Calculate the duality measure  $v$ .
5:     Solve 24.2 for the search direction  $(\Delta \mathbf{x}_k, \Delta \mathbf{y}_k, \Delta \boldsymbol{\mu}_k)$ .
6:     Calculate the step length  $\alpha$ .
7:      $(\mathbf{x}_{k+1}, \mathbf{y}_{k+1}, \boldsymbol{\mu}_{k+1}) = (\mathbf{x}_k, \mathbf{y}_k, \boldsymbol{\mu}_k) + \alpha(\Delta \mathbf{x}_k, \Delta \mathbf{y}_k, \Delta \boldsymbol{\mu}_k)$ .

```

---

### Problem 24.2: C

Complete the implementation of `qInteriorPoint()`. Return the optimal point `x` as well as the final objective function value.

Test your algorithm on the simple problem

$$\begin{aligned}
 & \text{minimize} && \frac{1}{2}x_1^2 + x_2^2 - x_1x_2 - 2x_1 - 6x_2 \\
 & \text{subject to} && -x_1 - x_2 \geq -2, \\
 & && x_1 - 2x_2 \geq -2, \\
 & && -2x_1 - x_2 \geq -3, \\
 & && x_1, x_2 \geq 0.
 \end{aligned}$$

In this case, we have for the objective function matrix  $Q$  and vector  $\mathbf{c}$ ,

$$Q = \begin{bmatrix} 1 & -1 \\ -1 & 2 \end{bmatrix}, \quad \mathbf{c} = \begin{bmatrix} -2 \\ -6 \end{bmatrix}.$$

The constraint matrix  $A$  and vector  $\mathbf{b}$  are given by:

$$A = \begin{bmatrix} -1 & -1 \\ 1 & -2 \\ -2 & -1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} -2 \\ -2 \\ -3 \\ 0 \\ 0 \end{bmatrix}.$$

Use  $\mathbf{x} = [.5, .5]$  as the initial guess. The correct minimizer is  $\left[\frac{2}{3}, \frac{4}{3}\right]$ .

(Hint: You may want to print out the duality measure  $v$  to check the progress of the iteration).

**NOTE**

The Interior Point methods presented in this and the preceding labs are only special cases of the more general Interior Point algorithm. The general version can be used to solve many convex optimization problems, provided that one can derive the corresponding KKT conditions and duality measure  $v$ .

## Application: Optimal Elastic Membranes

---

The properties of elastic membranes (stretchy materials like a thin rubber sheet) are of interest in certain fields of mathematics and various sciences. A mathematical model for such materials can be used by biologists to study interfaces in cellular regions of an organism or by engineers to design tensile structures. Often we can describe configurations of elastic membranes as a solution to an optimization problem. As a simple example, we will find the shape of a large circus tent by solving a quadratic constrained optimization problem using our Interior Point method.

Imagine a large circus tent held up by a few poles. We can model the tent by a square two-dimensional grid, where each grid point has an associated number that gives the height of the tent at that point. At each grid point containing a tent pole, the tent height is constrained to be at least as large as the height of the tent pole. At all other grid points, the tent height is simply constrained to be greater than zero (ground height). In Python, we can store a two-dimensional grid of values as a simple two-dimensional array. We can then flatten this array to give a one-dimensional vector representation of the grid. If we let  $\mathbf{x}$  be a one-dimensional array giving the tent height at each grid point, and  $L$  be the one-dimensional array giving the underlying tent pole structure (consisting mainly of zeros, except at the grid points that contain a tent pole), we have the linear constraint:

$$\mathbf{x} \succeq L.$$

The theory of elastic membranes claims that such materials tend to naturally minimize a quantity known as the *Dirichlet energy*. This quantity can be expressed as a quadratic function of the membrane. Since we have modeled our tent with a discrete grid of values, this energy function has the form

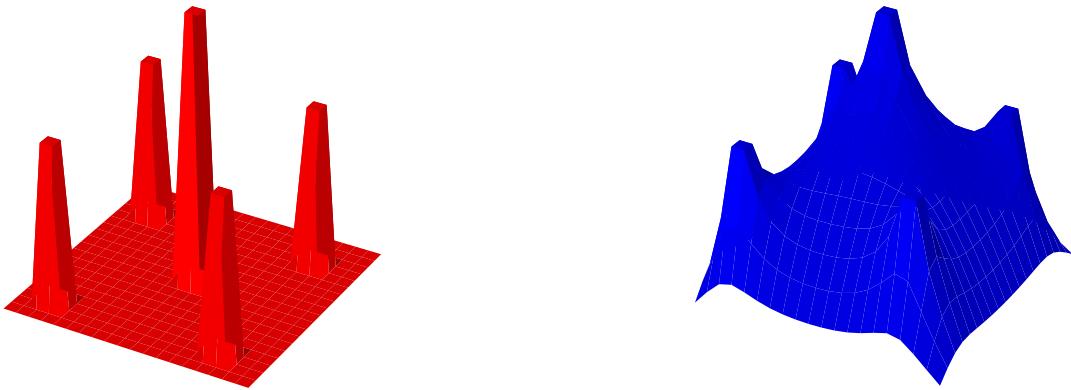
$$\frac{1}{2} \mathbf{x}^T H \mathbf{x} + \mathbf{c}^T \mathbf{x},$$

where  $H$  is a particular positive semidefinite matrix closely related to Laplace's Equation,  $\mathbf{c}$  is a vector whose entries are all equal to  $-(n-1)^{-2}$ , and  $n$  is the side length of the grid. Our circus tent is therefore given by the solution to the quadratic constrained optimization problem:

$$\begin{aligned} & \text{minimize} && \frac{1}{2} \mathbf{x}^T H \mathbf{x} + \mathbf{c}^T \mathbf{x} \\ & \text{subject to} && \mathbf{x} \succeq L. \end{aligned}$$

See Figure 24.1 for an example of a tent pole configuration and the corresponding tent.

We provide the following function for producing the Dirichlet energy matrix  $H$ .



**Figure 24.1: Tent pole configuration (left) and optimal elastic tent (right).**

```
from scipy.sparse import spdiags
def laplacian(n):
    """Construct the discrete Dirichlet energy matrix H for an n x n grid."""
    data = -1*np.ones((5, n**2))
    data[2,:] = 4
    data[1, n-1::n] = 0
    data[3, ::n] = 0
    diags = np.array([-n, -1, 0, 1, n])
    return spdiags(data, diags, n**2, n**2).toarray()
```

Now we initialize the tent pole configuration for a grid of side length  $n$ , as well as initial guesses for  $\mathbf{x}$ ,  $\mathbf{y}$ , and  $\mu$ .

```
# Create the tent pole configuration.
>>> L = np.zeros((n,n))
>>> L[n//2-1:n//2+1,n//2-1:n//2+1] = .5
>>> m = [n//6-1, n//6, int(5*(n/6))-1, int(5*(n/6))]
>>> mask1, mask2 = np.meshgrid(m, m)
>>> L[mask1, mask2] = .3
>>> L = L.ravel()

# Set initial guesses.
>>> x = np.ones((n,n)).ravel()
>>> y = np.ones(n**2)
```

```
>>> mu = np.ones(n**2)
```

We leave it to you to initialize the vector  $\mathbf{c}$ , the constraint matrix  $A$ , and to initialize the matrix  $H$  with the `laplacian()` function. We can solve and plot the tent with the following code:

```
>>> from matplotlib import pyplot as plt
>>> from mpl_toolkits.mplot3d import axes3d

# Calculate the solution.
>>> z = qInteriorPoint(H, c, A, L, (x,y,mu))[0].reshape((n,n))

# Plot the solution.
>>> domain = np.arange(n)
>>> X, Y = np.meshgrid(domain, domain)
>>> fig = plt.figure()
>>> ax1 = fig.add_subplot(111, projection='3d')
>>> ax1.plot_surface(X, Y, z, rstride=1, cstride=1, color='r')
>>> plt.show()
```

### Problem 24.3: S

Solve the circus tent problem with the tent pole configuration given above, for grid side length  $n = 15$ .  
Plot your solution.

## Application: Markowitz Portfolio Optimization

Suppose you have a certain amount of money saved up, with no intention of consuming it any time soon. What will you do with this money? If you hide it somewhere in your living quarters or on your person, it will lose value over time due to inflation, not to mention you run the risk of burglary or accidental loss. A safer choice might be to put the money into a bank account. That way, there is less risk of losing the money, plus you may even add to your savings through interest payments from the bank. You could also consider purchasing bonds from the government or stocks from various companies, which come with their own sets of risks and returns. Given all of these possibilities, how can you invest your money in such a way that maximizes the return (i.e. the wealth that you gain over the course of the investment) while still exercising caution and avoiding excessive risk? Economist and Nobel laureate Harry Markowitz developed the mathematical underpinnings and answer to this question in his work on modern portfolio theory.

A *portfolio* is a set of investments over a period of time. Each investment is characterized by a financial asset (such as a stock or bond) together with the proportion of wealth allocated to the asset. An asset is a random variable, and can be described as a sequence of values over time. The variance or spread of these

values is associated with the risk of the asset, and the percent change of the values over each time period is related to the return of the asset. For our purposes, we will assume that each asset has a positive risk, i.e. there are no *riskless* assets available.

Stated more precisely, our portfolio consists of  $n$  risky assets together with an allocation vector  $\mathbf{x} = (x_1, \dots, x_n)^\top$ , where  $x_i$  indicates the proportion of wealth we invest in asset  $i$ . By definition, the vector  $\mathbf{x}$  must satisfy

$$\sum_{i=1}^n x_i = 1.$$

Suppose the  $i$ th asset has an expected rate of return  $\mu_i$  and a standard deviation  $\sigma_i$ . The total return on our portfolio, i.e. the expected percent change in our invested wealth over the investment period, is given by

$$\sum_{i=1}^n \mu_i x_i.$$

We define the risk of this portfolio in terms of the covariance matrix  $Q$  of the  $n$  assets:

$$\sqrt{\mathbf{x}^\top Q \mathbf{x}}.$$

The covariance matrix  $Q$  is always positive semidefinite and captures the variance and correlations of the assets.

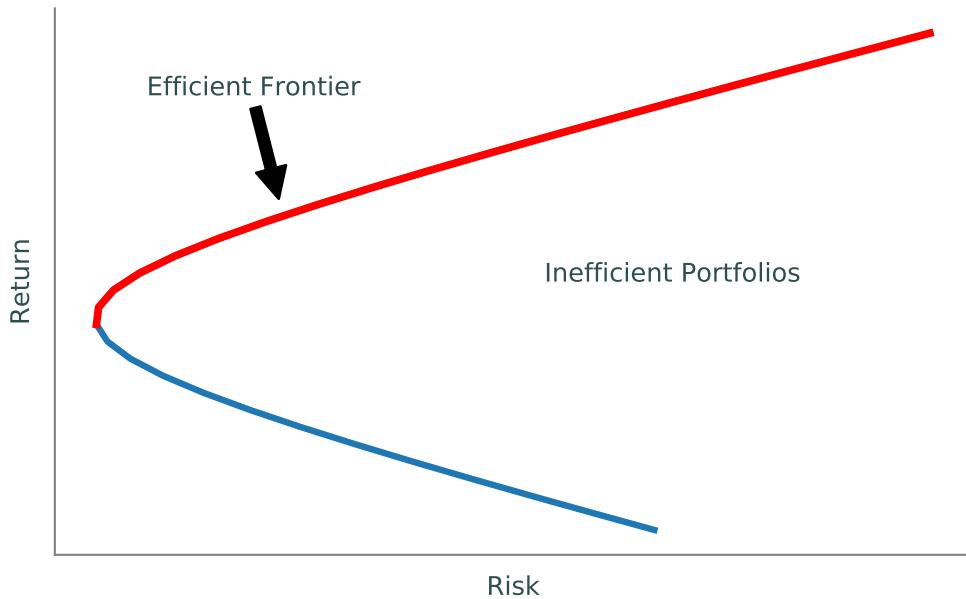
Given that we want our portfolio to have a prescribed return  $R$ , there are many possible allocation vectors  $\mathbf{x}$  that make this possible. It would be wise to choose the vector minimizing the risk. We can state this as a quadratic program:

$$\begin{aligned} & \text{minimize} && \frac{1}{2} \mathbf{x}^\top Q \mathbf{x} \\ & \text{subject to} && \sum_{i=1}^n x_i = 1 \\ & && \sum_{i=1}^n \mu_i x_i = R. \end{aligned}$$

Note that we have slightly altered our objective function for convenience, as minimizing  $\frac{1}{2} \mathbf{x}^\top Q \mathbf{x}$  is equivalent to minimizing  $\sqrt{\mathbf{x}^\top Q \mathbf{x}}$ . The solution to this problem will give the portfolio with least risk having a return  $R$ . Because the components of  $\mathbf{x}$  are not constrained to be nonnegative, the solution may have some negative entries. This indicates short selling those particular assets. If we want to disallow short selling, we simply include nonnegativity constraints, stated in the following problem:

$$\begin{aligned} & \text{minimize} && \frac{1}{2} \mathbf{x}^\top Q \mathbf{x} \\ & \text{subject to} && \sum_{i=1}^n x_i = 1 \\ & && \sum_{i=1}^n \mu_i x_i = R \\ & && \mathbf{x} \succeq 0. \end{aligned}$$

Each return value  $R$  can be paired with its corresponding minimal risk  $\sigma$ . If we plot these risk-return pairs on the risk-return plane, we obtain a hyperbola. In general, the risk-return pair of any portfolio, optimal or not, will be found in the region bounded on the left by the hyperbola. The positively-sloped portion of the hyperbola is known as the *efficient frontier*, since the points there correspond to optimal portfolios. Portfolios with risk-return pairs that lie to the right of the efficient frontier are inefficient portfolios, since we could either increase the return while keeping the risk constant, or we could decrease the risk while keeping the return constant. See Figure 24.2.



**Figure 24.2: Efficient frontier on the risk-return plane.**

One weakness of this model is that the risk and return of each asset is in general unknown. After all, no one can predict the stock market with complete certainty. There are various ways of estimating these values given past stock prices, and we take a very straightforward approach. Suppose for each asset, we have  $k$  previous return values of the asset. That is, for asset  $i$ , we have the data vector

$$\mathbf{y}^i = [y_1^i, \dots, y_k^i]^T.$$

We estimate the expected rate of return for asset  $i$  by simply taking the average of  $y_1, \dots, y_k$ , and we estimate the variance of asset  $i$  by taking the variance of the data. We can estimate the covariance matrix for all assets by taking the covariance matrix of the vectors  $\mathbf{y}^1, \dots, \mathbf{y}^n$ . In this way, we obtain estimated values for each  $\mu_i$  and  $Q$ .

**Problem 24.4: T**

The text file `portfolio.txt` contains historical stock data for several assets (U.S. bonds, gold, S&P 500, etc). In particular, the first column gives the years corresponding to the data, and the remaining eight columns give the historical returns of eight assets over the course of these years. Use this data to estimate the covariance matrix  $Q$  as well as the expected rates of return  $\mu_i$  for each asset. Assuming that we want to guarantee an expected return of  $R = 1.13$  for our portfolio, find the optimal portfolio both with and without short selling.

Since the problem contains both equality and inequality constraints, use the QP solver in CVXOPT rather than your `qInteriorPoint()` function.

Hint: Use `numpy.cov()` to compute  $Q$ .



# 25. OpenGym AI

---

**Lab Objective:** *OpenGym AI is a module designed to learn and apply reinforcement learning. The purpose of this lab is to learn the variety of functionalities available in OpenGym AI and to implement them in various environments. Applying reinforcement learning techniques into OpenGym AI will take place in future labs.*

OpenGym AI is a module used to perform reinforcement learning. It contains a collection of environments where reinforcement learning can be used to accomplish various tasks. These environments include performing computer functions such as copy and paste, playing Atari video games, and controlling robots. To install OpenGym AI, run the following code.

```
>>> pip install gym
```

## Environments

---

Each environment in OpenGym AI can be thought of as a different scenario where reinforcement learning can be applied. A catalog of all available environments can be found using the following code:

```
>>> from gym import envs
>>> print(envs.registry.all())
dict_values([EnvSpec(Copy-v0), EnvSpec(RepeatCopy-v0), EnvSpec(<-
    ReversedAddition-v0), EnvSpec(ReversedAddition3-v0), EnvSpec(<-
    DuplicatedInput-v0), EnvSpec(Reverse-v0), EnvSpec(CartPole-v0), ...])
```

Note that some of these environments require additional software. To learn more about each environment, its scenario and necessary software, visit [gym.openai.com/envs](https://gym.openai.com/envs).

To begin working in an environment, identify the name of the environment and initialize the environment. Once initialized, make sure to reset the environment. Resetting the environment is necessary to begin using the environment by putting everything in the correct starting position. For example, the environment "NChain-v0" presents a scenario where a player is traversing a chain with  $n$  states. Restarting the environment puts the player at the beginning of the chain. Once the environment is complete, make sure to close the environment. Closing the environment tells the computer to stop running the environment as to not run the environment in the background.

```
>>> import gym
```

```
>>> # Get NChain-v0 environment
env = gym.make('NChain-v0')

>>> # Reset the environment
>>> env.reset()
0

>>> # Close the environment
>>> env.close()
```

## Action Space

---

Once reset, the player in the environment can then perform actions from the action space. In "NChain-v0", the action space has 2 actions; move forward one state or return to the beginning of the chain. To perform an action, use the function `step`, which accepts the action as a parameter and returns an observation (more on those later). If the action space is discrete, then actions are defined as integers 0 through  $n$ , where  $n$  is the number of actions. The action each integer represents can be found in the documentation of each environment.

```
>>> # Determine the number of actions available
>>> env.action_space
Discrete(2)

>>> # Reset environment and perform a random action
>>> env.step(env.action_space.sample())
(1, 0, False, {})
```

However, not all action spaces are discrete. Consider the environment "GuessingGame-v0". The purpose of this game is to guess within 1% of a random number in the interval  $[-1000, 1000]$  in 200 guesses. Since the number is not required to be an integer, and each action is guessing a number, it does not make sense for the action space to be discrete. Rather this action space should be an interval. In OpenGym AI, this action space is described as an  $n$ -dimensional array `Box(n, )`. This means an feasible action is an  $n$ -dimensional vector. To identify the range of the box, use the attributes `high` and `low`. Thus, in the environment "GuessingGame-v0", the action space will be a 1-dimensional box with range  $[-1000, 1000]$ .

```
>>> # Get Guessing Game environment
env = gym.make("GuessingGame-v0")

>>> # Check size of action space
>>> env.action_space
Box(1, )
```

```
>>> # Check range of action space
>>> env.action_space.high
array([10000.], dtype=float32)

>>> env.action_space.low
array([-10000.], dtype=float32)
```

## Observation Space

---

The observation space contains all possible observations given an action. For example, in "NChain-v0", an observation would be the position of the player on the chain and in "GuessingGame-v0", the observation would be whether the guess is higher than, equal to, or lower than the target. The observation from each action can be found in the tuple returned by `step`. This tuple tells us the following information:

1. `observation`: The current state of the environment. For example, in "GuessingGame-v0", 0 indicates the guess is too high, 1 indicates the guess is on target, and 2 indicates the guess is too low.
2. `reward`: The reward given from the observation. In most environments, maximizing the total reward increases performance. For example, the reward for each observation is 0 in "GuessingGame-v0" unless the observation is within 1% of the target.
3. `done`: A boolean indicating if the observation terminates the environment.
4. `info`: Various information that may be helpful when debugging.

Consider the code below.

```
>>> env = gym.make("GuessingGame-v0")

>>> # Make a random guess
>>> env.step(env.action_space.sample())
(1, 0, False, {'guesses': 1, 'number': 524.50509074})
```

This tuple can be interpreted as follows:

1. The guess was too high.
2. The guess was not within 1% of the target.
3. The environment is not terminated.
4. Information that may help debugging (the number of guesses made so far and the target number).

**Problem 25.1: T**

The game Blackjack<sup>a</sup> is a card game where the player receives two cards from a facecard deck. The goal of the player is to get cards whose sum is as close to 21 as possible without exceeding 21. In this version of Blackjack, an ace is considered 1 or 11 and any facecard is considered 10. At each turn, the player may choose to take another card or stop drawing cards. If their card sum does not exceed 21, they may take another card. If it does, they lose. After the player stops drawing cards, the computer may play the same game. If the computer gets closer to 21 than the player, the player loses.

The environment "Blackjack-v0" is an OpenGym AI environment that plays blackjack. The actions in the action space are 0 to stop drawing and 1 to draw another card. The observation (first entry in the tuple returned by `step`) is a tuple containing the total sum of the players hand, the first card of the computer's hand, and whether the player has an ace. The reward (second entry in the tuple returned by `step`) is 1 if the player wins, -1 if the player loses, and 0 if there is a draw.

Write a function `random_blackjack()` that accepts an integer `n`. Initialize "Blackjack-v0" `n` times and each time take random actions until the game is terminated. Return the percentage of games the player wins.

<sup>a</sup>For more on how to play Blackjack, see <https://en.wikipedia.org/wiki/Blackjack>.

## Understanding Environments

---

Because each action and observation space is made up of numbers, good documentation is imperative to understanding any given environment. Fortunately, most environments in OpenAI Gym are very well documented. Documentation for any given environment can be found through [gym.openai.com/envs](https://gym.openai.com/envs) by clicking on the github link in the environment.

Most documentation follows the same pattern. There is a docstring which includes a description of the environment, a detailed action space, a detailed observation space, and explanation of rewards. It is always helpful to refer to this documentation when working in an OpenGym AI environment.

In addition to documentation, certain environments can be understood better through visualization. For example, the environment "Acrobot-v1" displays an inverted pendulum. Visualizing the environment allows the user to see the movement of the inverted pendulum as forces are applied to it. This can be done with the function `render()`. When using `render()`, ALWAYS use a try-finally block to close the environment. This ensures that the video rendering ends no matter what.

```
>>> # Get environment
>>> env = gym.make("Acrobot-v1")

>>> # Take random actions and visualize each action
>>> try:
>>>     env.reset()
>>>     done = False
>>>     while not done:
```

```
>>> env.render()
>>> obs, reward, done, info = env.step(env.action_space.sample())
>>> if done:
>>>     break
>>> finally:
>>>     env.close()
```



**Figure 25.1: Rendering of "Acrobot-v1"**

### Problem 25.2: W

ite a function `blackjack()` which runs a naive algorithm to win blackjack. The function should receive an integer  $n$ . If the players hand is less than or equal to  $n$ , the player should draw another card. If the players hand is more than  $n$ , they should stop playing. Run the algorithm 10000 times and return the average reward. What value of  $n$  wins most on average?

## Solving An Environment

---

One way to solve an environment is to use information from the current observation to choose our next action. For example, consider "["GuessingGame-v0"](#)". Each observation tells us whether the guess was too high or too low. After each observation, the interval where the target lies continues to get smaller. By choosing the midpoint of the current interval where the target lies, the true target can be identified much faster.

**Problem 25.3: T**

The environment "*CartPole-v0*" presents a cart with a vertical pole. The goal of the environment is to keep the pole vertical as long as possible. Write a function `cartpole()` which initializes the environment and keeps the pole vertical as long as possible based on the velocity of the tip of the pole. Render the environment at each step and return the time before the environment terminates. The time should be at least 2 seconds and on average be about 3 seconds.

(Hint: Use the documentation of the environment to determine the meaning of each action and observation. It can be found at [https://github.com/openai/gym/wiki/CartPole-v0.0](https://github.com/openai/gym/wiki/CartPole-v0.).)

**Problem 25.4: T**

The environment "*MountainCar-v0*" shows a car in a valley. The goal of the environment is to get the car to the top of the right mountain. The car can be driven forward (toward the goal) with the action 2, can be driven backward with the action 0, and will be put in neutral with the action 1. Note that the car cannot immediately get up the hill because of gravity. In order to move the car to goal, momentum will need to be gained by going back and forth between both sides of the valley. Each observation is a 2-dimensional array, containing the (x,y) position of the car. Using the position of the car, write a function `car()` that solves the "*MountainCar-v0*" environment. Render the environment at each step and return the time before the environment terminates. The time should be less than 3 seconds.

## Q-Learning

---

While naive methods like the ones above can be useful, reinforcement is a much better approach for using OpenAI Gym. Reinforcement learning is a subfield of machine learning where a problem is attempted over and over again. Each time a method is used to solve the problem, the method adapts based on the information gained from the previous attempt. Information can be gained from the sequence of observations and the total reward earned.

A simple reinforcement method is called *Q-learning*. While the details of Q-learning will not be explained in detail, the main idea is that the next action is not only based on the reward of the current action, but also of the next action. Q-learning creates a Q-table, which is an  $n \times m$  dimensional array, where  $n$  is the number of observations and  $m$  is the number of actions. For each state, the optimal action is the action that maximizes the value in the Q-table. In other words, if I am at observation  $i$ , the best action is the argmax of row  $i$  in the Q-table.

Q-learning requires 3 hyperparameters:

1. alpha: the learning rate. This determines whether to accept new values into the q-table.
2. gamma: the discount factor. The discount factor determines how important the reward of the current action is compared to the following action.

3. `epsilon`: the maximum value. This is the max reward that can be earned from a future action (not the current).

These hyperparameters can be changed to created different Q-tables.

**Problem 25.5: W**

ite a function `taxi()` which initializes the environment "`Taxi-v2`". The goal of this environment is the pick up a passenger in a taxi and drop them off at their destination as fast as possible (see <https://gym.openai.com/envs/Taxi-v2/>). First, randomly act until the environment is done and calculate the reward. Then use `find_qtable()` to get the optimal Q-table of the environment. Set `alpha=.1`, `gamma=.6`, and `epsilon=.1`. Use the qtable to move through the environment and calculate the reward. Return the average reward of the random moves and the average reward of the Q-learning over 10000 iterations.

(Hint: Use the documentation found at [https://github.com/openai/gym/blob/master/gym/envs/toy\\_text/taxi.py](https://github.com/openai/gym/blob/master/gym/envs/toy_text/taxi.py) to understand the environment better).



# 26. K-Means Clustering

---

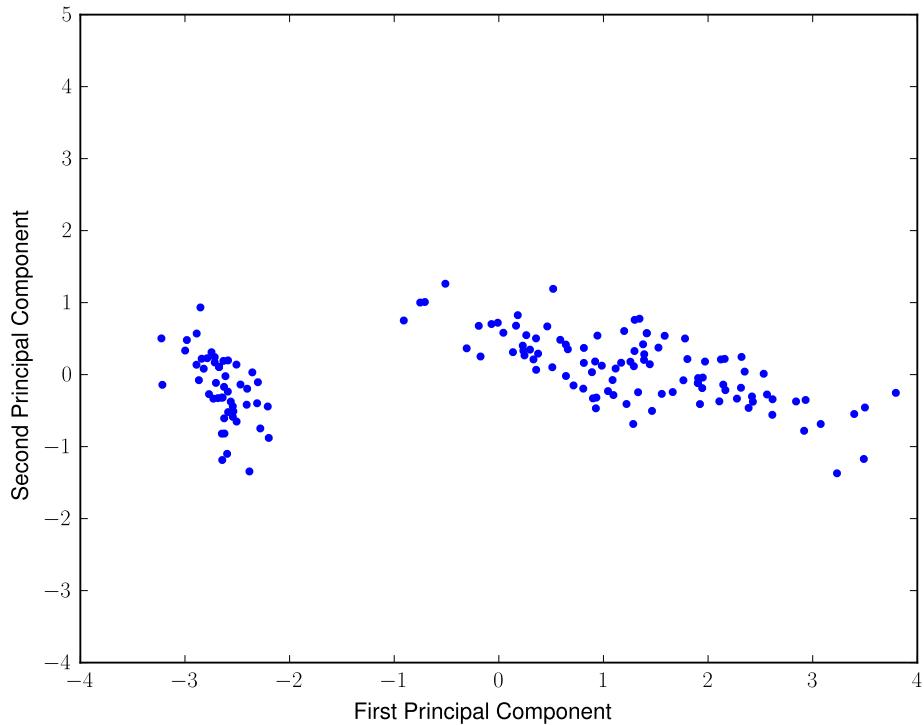
**Lab Objective:** *Clustering is the one of the main tools in unsupervised learning—machine learning problems where the data comes without labels. In this lab we implement the k-means algorithm, a simple and popular clustering method, and apply it to geographic clustering and color quantization.*

## Clustering

---

Previously, we analyzed the iris dataset from `sklearn` using PCA; we have reproduced the first two principal components of the iris data in Figure 26.1. Upon inspection, a human can easily see that there are two very distinct groups of irises. Can we create an algorithm to identify these groups without human supervision? This task is called *clustering*, an instance of *unsupervised learning*.

The objective of clustering is to find a partition of the data such that points in the same subset will be “close” according to some metric. The metric used will likely depend on the data, but some obvious choices include Euclidean distance and angular distance. Throughout this lab we will use the metric



**Figure 26.1: The first two principal components of the iris dataset.**

$d(x, y) = \|x - y\|_2$ , the Euclidean distance between  $x$  and  $y$ .

More formally, suppose we have a collection of  $\mathbb{R}^K$ -valued observations  $X = \{x_1, x_2, \dots, x_n\}$ . Let  $N \in \mathbb{N}$  and let  $\mathcal{S}$  be the set of all  $N$ -partitions of  $X$ , where an  $N$ -partition is a partition with exactly  $N$  nonempty elements. We can represent a typical partition in  $\mathcal{S}$  as  $S = \{S_1, S_2, \dots, S_N\}$ , where

$$X = \bigcup_{i=1}^N S_i$$

and

$$|S_i| > 0, \quad i = 1, 2, \dots, N.$$

We seek the  $N$ -partition  $S^*$  that minimizes the within-cluster sum of squares, i.e.

$$S^* = \arg \min_{S \in \mathcal{S}} \sum_{i=1}^N \sum_{x_j \in S_i} \|x_j - \mu_i\|_2^2,$$

where  $\mu_i$  is the mean of the elements in  $S_i$ , i.e.

$$\mu_i = \frac{1}{|S_i|} \sum_{x_j \in S_i} x_j.$$

## The K-Means Algorithm

---

Finding the global minimizing partition  $S^*$  is generally intractable since the set of partitions can be very large indeed, but the *k-means* algorithm is a heuristic approach that can often provide reasonably accurate results.

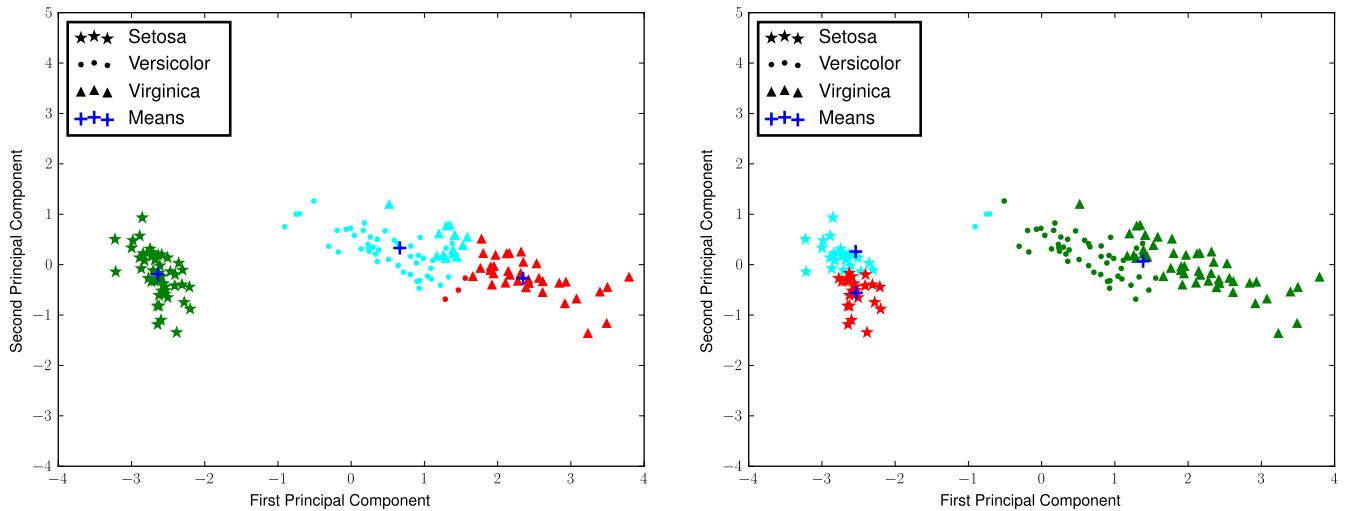
We begin by specifying an initial cluster mean  $\mu_i^{(1)}$  for each  $i = 1, \dots, N$  (this can be done by random initialization, or according to some heuristic). For each iteration, we adopt the following procedure. Given a current set of cluster means  $\mu^{(t)}$ , we find a partition  $S^{(t)}$  of the observations such that

$$S_i^{(t)} = \{x_j : \|x_j - \mu_i^{(t)}\|_2^2 \leq \|x_j - \mu_l^{(t)}\|_2^2, l = 1, \dots, N\}.$$

We then update our cluster means by computing for each  $i = 1, \dots, N$ . We continue to iterate in this manner until the partition ceases to change.

Figure 26.2 shows two different clusterings of the iris data produced by the *k-means* algorithm. Note that the quality of the clustering can depend heavily on the initial cluster means. We can use the within-cluster sum of squares as a measure of the quality of a clustering (a lower sum of squares is better). Where possible, it is advisable to run the clustering algorithm several times, each with a different initialization of the means, and keep the best clustering. Note also that it is possible to have very slow convergence. Thus, when implementing the algorithm, it is a good idea to terminate after some specified maximum number of iterations. The algorithm can be summarized as follows.

1. Choose  $k$  initial cluster centers.
2. For  $i = 0, \dots, \text{max\_iter}$ ,



**Figure 26.2:** Two different K-Means clusterings for the iris dataset. Notice that the clustering on the left predicts the flower species to a high degree of accuracy, while the clustering on the right is less effective.

- Assign each data point to the cluster center that is closest, forming  $k$  clusters.
- Recompute the cluster centers as the means of the new clusters.
- If the old cluster centers and the new cluster centers are sufficiently close, terminate early.

### Problem 26.1: W

Write a `KMeans` class for doing basic  $k$ -means clustering. Implement the following methods, following `sklearn` class conventions.

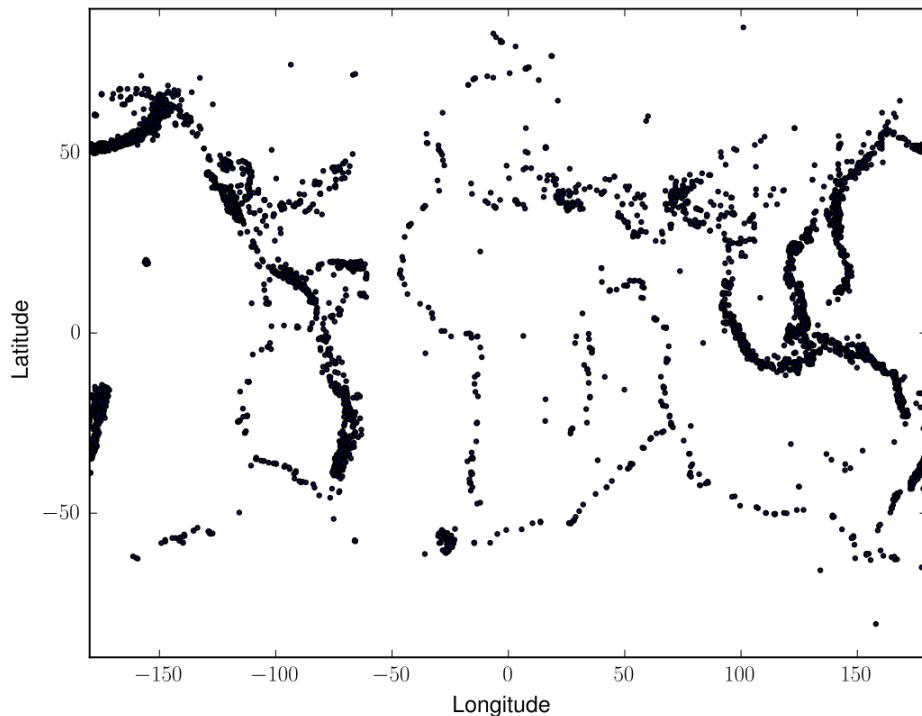
1. `__init__()`: Accept a number of clusters  $k$ , a maximum number of iterations, and a convergence tolerance. Store these as attributes.
2. `fit()`: Accept an  $m \times n$  matrix  $X$  of  $m$  data points with  $n$  features. Choose  $k$  random rows of  $X$  as the initial cluster centers. Run the  $k$ -means iteration until consecutive centers are within the convergence tolerance, or until iterating the maximum number of times. Save the cluster centers as attributes.  
If a cluster is empty, reassign the cluster center as a random row of  $X$ .
3. `predict()`: Accept an  $l \times n$  matrix  $X$  of data. Return an array of  $l$  integers where the  $i$ th entry indicates which cluster center the  $i$ th row of  $X$  is closest to.

Test your class on the `iris` data set after reducing the data to two principal components. Plot the data, coloring by cluster.

## Detecting Active Earthquake Regions

---

Suppose we are interested in learning about which regions are prone to experience frequent earthquake activity. We could make a map of all earthquakes over a given period of time and examine it ourselves, but this, as an unsupervised learning problem, can be solved using our  $k$ -means clustering tool.



**Figure 26.3: Earthquake epicenters over a 6 month period.**

The file `earthquake_coordinates.npy` contains earthquake data throughout the world from January 2010 through June 2010. Each row represents a different earthquake; the columns are scaled longitude and latitude measurements. We want to cluster this data into active earthquake regions. For this task, we might think that we can regard any epicenter as a point in  $\mathbb{R}^2$  with coordinates being their latitude and longitude. This, however, would be incorrect, because the earth is not flat. Instead, latitude and longitude should be viewed in *spherical coordinates* in  $\mathbb{R}^3$ , which could then be clustered.

A simple way to accomplish this transformation is to first transform the latitude and longitude values to spherical coordinates, and then to Euclidean coordinates. Recall that a spherical coordinate in  $\mathbb{R}^3$  is a triple  $(r, \theta, \varphi)$ , where  $r$  is the distance from the origin,  $\theta$  is the radial angle in the  $xy$ -plane from the  $x$ -axis, and  $\varphi$  is the angle from the  $z$ -axis. In our earthquake data, once the longitude is converted to radians it is an appropriate  $\theta$  value; the latitude needs to be offset by  $90^\circ$  degrees, then converted to radians to obtain  $\varphi$ . For simplicity, we can take  $r = 1$ , since the earth is roughly a sphere. We can then transform to Euclidean coordinates using the following relationships.

$$\theta = \frac{\pi}{180} (\text{longitude}) \quad \varphi = \frac{\pi}{180} (90 - \text{latitude})$$

$$\begin{array}{ll}
 r = \sqrt{x^2 + y^2 + z^2} & x = r \sin \varphi \cos \theta \\
 \varphi = \arccos \frac{z}{r} & y = r \sin \varphi \sin \theta \\
 \theta = \arctan \frac{y}{x} & z = r \cos \varphi
 \end{array}$$

There is one last issue to solve before clustering. Each earthquake data point has norm 1 in Euclidean coordinates, since it lies on the surface of a sphere of radius 1. Therefore, the cluster centers should also have norm 1. Otherwise, the means can't be interpreted as locations on the surface of the earth, and the *k-means* algorithm will struggle to find good clusters. A solution to this problem is to normalize the mean vectors at each iteration, so that they are always unit vectors.

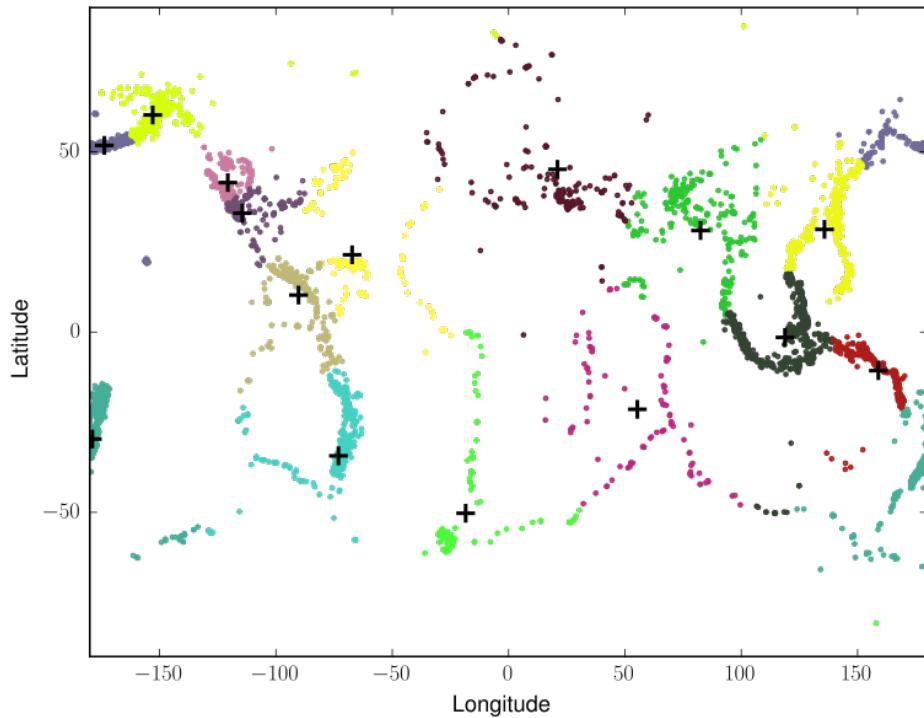
### Problem 26.2: A

*d* a keyword argument `normalize=False` to your `KMeans` constructor. Modify `fit()` so that if `normalize` is `True`, the cluster centers are normalized at each iteration.

Cluster the earthquake data in three dimensions by converting the data from raw data to spherical coordinates to euclidean coordinates on the sphere.

1. Convert longitude and latitude to radians, then to spherical coordinates.  
*(Hint: np.deg2rad() may be helpful.)*
2. Convert the spherical coordinates to euclidean coordinates in  $\mathbb{R}^3$ .
3. Use your `KMeans` class with normalization to cluster the euclidean coordinates.
4. Translate the cluster center coordinates back to spherical coordinates, then to degrees. Transform the cluster means back to latitude and longitude coordinates.  
*(Hint: use numpy.arctan2() for arctan, so that that correct quadrant is chosen).*
5. Plot the data, coloring by cluster. Also mark the cluster centers.

With 15 clusters, your plot should resemble the Figure 26.4.



**Figure 26.4: Earthquake epicenter clusters with  $k = 15$ .**

## Color Quantization

---

The  $k$ -means algorithm uses the euclidean metric, so it is natural to cluster geographic data. However, clustering can be done in any abstract vector space. The following application is one example.

Images are usually represented on computers as 3-dimensional arrays. Each 2-dimensional layer represents the red, green, and blue color values, so each pixel on the image is really a vector in  $\mathbb{R}^3$ . Clustering the pixels in  $RGB$  space leads a one kind of image segmentation that facilitate memory reduction.

Reading: [https://en.wikipedia.org/wiki/Color\\_quantization](https://en.wikipedia.org/wiki/Color_quantization)

### Problem 26.3: W

Write a function that accepts an image array (of shape  $(m, n, 3)$ ), an integer number of clusters  $k$ , and an integer number of samples  $S$ . Reshape the image so that each row represents a single pixel. Choose  $S$  pixels to train a  $k$ -means model on with  $k$  clusters. Make a copy of the original picture where each pixel has the same color as its cluster center. Return the new image. For this problem, you may use `sklearn.cluster.KMeans` instead of your `KMeans` class from Problem 1.

Test your function on some of the provided NASA images.

# Additional Material

---

## Spectral Clustering

---

We now turn to another method for solving a clustering problem, namely that of Spectral Clustering. As you can see in Figure ???, it can cluster data not just by its location on a graph, but can even separate shapes that overlap others into distinct clusters. It does so by utilizing the spectral properties of a Laplacian matrix. Different types of Laplacian matrices can be used. In order to construct a Laplacian matrix, we first need to create a graph of vertices and edges from our data points. This graph can be represented as a symmetric matrix  $W$  where  $w_{ij}$  represents the edge from  $x_i$  to  $x_j$ . In the simplest approach, we can set  $w_{ij} = 1$  if there exists an edge and  $w_{ij} = 0$  otherwise. However, we are interested in the similarity of points, so we will weight the edges by using a *similarity measure*. Points that are similar to one another are assigned a high similarity measure value, and dissimilar points a low value. One possible measure is the *Gaussian similarity function*, which defines the similarity between distinct points  $x_i$  and  $x_j$  as

$$s(x_i, x_j) = e^{-\frac{\|x_i - x_j\|^2}{2\sigma^2}}$$

for some set value  $\sigma$ .

Note that some similarity functions can yield extremely small values for dissimilar points. We have several options for dealing with this possibility. One is simply to set all values which are less than some  $\varepsilon$  to be zero, entirely erasing the edge between these two points. Another option is to keep only the  $T$  largest-valued edges for each vertex. Whichever method we choose to use, we will end up with a weighted *similarity matrix*  $W$ . Using this we can find the diagonal *degree matrix*  $D$ , which gives the number of edges found at each vertex. If we have the original fully-connected graph, then  $D_{ii} = n - 1$  for each  $i$ . If we keep the  $T$  highest-valued edges,  $D_{ii} = T$  for each  $i$ .

As mentioned before, we may use different types of Laplacian matrices. Three such possibilities are:

1. The *unnormalized Laplacian*,  $L = D - W$
2. The *symmetric normalized Laplacian*,  $L_{\text{sym}} = I - D^{-1/2}WD^{-1/2}$
3. The *random walk normalized Laplacian*,  $L_{\text{rw}} = I - D^{-1}W$ .

Given a similarity measure, which type of Laplacian to use, and the desired number of clusters  $k$ , we can now proceed with the Spectral Clustering algorithm as follows:

- Compute  $W$ ,  $D$ , and the appropriate Laplacian matrix.
- Compute the first  $k$  eigenvectors  $u_1, \dots, u_k$  of the Laplacian matrix.
- Set  $U = [u_1, \dots, u_k]$ , and if using  $L_{\text{sym}}$  or  $L_{\text{rw}}$  normalize  $U$  so that each row is a unit vector in the Euclidean norm.

- Perform  $k$ -means clustering on the  $n$  rows of  $U$ .
- The  $n$  labels returned from your `kmeans` function correspond to the label assignments for  $x_1, \dots, x_n$ .

As before, we need to run through our  $k$ -means function multiple times to find the best measure when we use random initialization. Also, if you normalize the rows of  $U$ , then you will need to set the argument `normalize = True`.

### Problem 26.4: I

Implement the Spectral Clustering Algorithm by calling your `kmeans` function, using the following function declaration:

```
def specClus(measure,Laplacian,args,arg1=None,kiters=10):
    """
    Cluster a dataset using the k-means algorithm.

    Parameters
    -----
    measure : function
        The function used to calculate the similarity measure.
    Laplacian : int in {1,2,3}
        Which Laplacian matrix to use. 1 corresponds to the unnormalized,
        2 to the symmetric normalized, 3 to the random walk normalized.
    args : tuple
        The arguments as they were passed into your k-means function,
        consisting of (data, n_clusters, init, max_iter, normalize). Note
        that you will not pass 'data' into your k-means function.
    arg1 : None, float, or int
        If Laplacian==1, it should remain as None
        If Laplacian==2, the cut-off value, epsilon.
        If Laplacian==3, the number of edges to retain, T.
    kiters : int
        How many times to call your kmeans function to get the best
        measure.

    Returns
    -----
    labels : ndarray of shape (n,)
        The i-th entry is an integer in [0,n_clusters-1] indicating
        which cluster the i-th row of data belongs to.
    """
    pass
```

We now need a way to test our code. The website <http://cs.joensuu.fi/sipu/datasets/> contains many free data sets that will be of use to us. Scroll down to the “Shape sets” heading, and download some of the datasets found there to use for trial datasets.

### Problem 26.5: C

Create a function that will return the accuracy of your spectral clustering implementation, as follows:

```
def test_specClus(location,measure,Laplacian,args,arg1=None,kiters=10):
    """
    Cluster a dataset using the k-means algorithm.

    Parameters
    -----
    location : string
        The location of the dataset to be tested.
    measure : function
        The function used to calculate the similarity measure.
    Laplacian : int in {1,2,3}
        Which Laplacian matrix to use. 1 corresponds to the unnormalized,
        2 to the symmetric normalized, 3 to the random walk normalized.
    args : tuple
        The arguments as they were passed into your k-means function,
        consisting of (data, n_clusters, init, max_iter, normalize). Note
        that you will not pass 'data' into your k-means function.
    arg1 : None, float, or int
        If Laplacian==1, it should remain as None
        If Laplacian==2, the cut-off value, epsilon.
        If Laplacian==3, the number of edges to retain, T.
    kiters : int
        How many times to call your kmeans function to get the best
        measure.

    Returns
    -----
    accuracy : float
        The percent of labels correctly predicted by your spectral
        clustering function with the given arguments (the number
        correctly predicted divided by the total number of points).
    """

    pass
```



# **Part IV**

# **Advanced Optimization**



# 27. Integral polyhedra, TU matrices, TDI systems

---

## 27.1 Integral polyhedra

---

### 27.1.1. Basics

---

**Definition 27.1: Integral polyhedron**

*polyhedron  $P$  is called integral if every minimal face of  $P$  contains an integral vector.*

**Remark 27.2**

*If  $P$  has vertices, then  $P$  is integral if and only if every vertex is an integral vector.*

### 27.1.2. Properties

---

**Theorem 27.3**

*Let  $P \subseteq \mathbb{R}^n$  be a polyhedron. Then the following are equivalent:*

1.  $P = \text{conv}(P \cap \mathbb{Z}^n)$
2.  $P$  is integral
3.  $\max\{c^\top x : x \in P\}$  has an integral optimal solution for all  $c \in \mathbb{R}^n$  such that the optimal value is finite.
4.  $\max\{c^\top x : x \in P\}$  has an integral optimal solution for all  $c \in \mathbb{Z}^n$  such that the optimal value is finite.
5.  $\max\{c^\top x : x \in P\}$  is an integer for all  $c \in \mathbb{Z}^n$  such that the optimal value is finite.

## 27.2 Unimodular and totally unimodular matrices

---

### 27.2.1. Unimodular matrices

---

**Definition 27.4: Unimodular matrix**

*matrix  $A \in \mathbb{R}^{m \times n}$  is called unimodular if: (1) All entries are integers. (2)  $A$  has full rank. (3) Every  $m \times m$  square submatrix of  $A$  has determinant  $-1, 0, 1$ .*

**Theorem 27.5**

*Let  $A \in \mathbb{Z}^{m \times n}$  be a full row rank matrix. Then the polyhedron  $P = \{x \in \mathbb{R}^n : Ax = b, x \geq 0\}$  is integral for all  $b \in \mathbb{Z}^m$  if and only if  $A$  is unimodular.*

### 27.2.2. Totally unimodular matrices

---

**Definition 27.6: Totally unimodular matrix**

*The matrix  $A \in \mathbb{R}^{m \times n}$  is called totally unimodular if every square submatrix of  $A$  has determinant  $-1, 0, 1$ .*

**Theorem 27.7**

*Let  $A \in \mathbb{Z}^{m \times n}$ . Then the polyhedron  $P = \{x \in \mathbb{R}^n : Ax \leq b, x \geq 0\}$  is integral for all  $b \in \mathbb{Z}^m$  if and only if  $A$  is totally unimodular.*

**Theorem 27.8**

*Let  $A \in \mathbb{Z}^{m \times n}$  be a totally unimodular matrix. Then the polyhedron  $P = \{x \in \mathbb{R}^n : Ax \leq b, x \geq 0\}$  is integral for all  $b \in \mathbb{Z}^m$ .*

### 27.2.3. How to detect unimodularity and totally unimodularity

#### Theorem 27.9

*Basic properties]* Let  $A \in \mathbb{Z}^{n \times m}$ . Then the following are equivalent:

1.  $A$  is totally unimodular
2.  $A^T$  is totally unimodular
3.  $[A \ I]$  is totally unimodular (where  $I \in \mathbb{R}^n$  denotes the identity matrix)
4.  $[A \ I]$  is unimodular

#### Theorem 27.10

Let  $A \in \mathbb{Z}^{m \times n}$ . Then  $A$  is totally unimodular if and only if for all  $J \subseteq \{1, \dots, m\}$  there exists  $J_1, J_2$  such that

1.  $J_1 \cap J_2 = \emptyset$  and  $J = J_1 \cup J_2$
2. For all  $i = 1, \dots, n$  we have

$$\left| \sum_{j \in J_1} a_{ji} - \sum_{j \in J_2} a_{ji} \right| \leq 1$$

#### Remark 27.11: A

analogous result can be written in terms of the columns instead of the rows of  $A$ .

### 27.2.4. Examples of totally unimodular matrices

Classical examples of matrices that are totally unimodular are: network flow matrices, the node-incidence matrix for a bipartite graph, interval matrices.

## 27.3 Totally dual integral systems

### 27.3.1. Basics

---

#### Definition 27.12

**Totally dual integral system]** Let  $A \in \mathbb{Q}^{m \times n}$  and  $b \in \mathbb{Q}^m$ . The system  $Ax \leq b$  is totally dual integral system (TDI) if for each integral vector  $c \in \mathbb{Z}^n$  such that

$$\max\{c^T x : Ax \leq b\}$$

is finite, then the dual

$$\min\{b^T y : A^T y = c, y \geq 0\}$$

has an integral optimal solution.

### 27.3.2. Properties

---

#### Theorem 27.13

Let  $A \in \mathbb{Q}^{m \times n}$  and  $b \in \mathbb{Z}^m$ . If  $Ax \leq b$  is TDI then  $P = \{x \in \mathbb{R}^n : Ax \leq b\}$  is an integral polyhedron.

#### Remark 27.14: T

The condition  $b \in \mathbb{Z}^n$  is crucial in the proof of the theorem above.

### 27.3.3. Totally unimodularity and TDI systems

---

#### Theorem 27.15

Let  $A \in \mathbb{Q}^{m \times n}$  be a totally unimodular matrix. Then the system  $Ax \leq b$  is TDI for all  $b \in \mathbb{R}^n$ .

### 27.3.4. Examples of TDI systems

---

Classical examples of TDI systems are: the independent set formulation for matroids, matchings.

# 28. Cutting Planes

## 28.1 Introduction

### 28.1.1. Cutting planes

#### Definition 28.1

*Cutting plane for IP]* Let  $P \subseteq \mathbb{R}^n$  be a polyhedron. An inequality  $a^T x \leq b$  is called a cutting plane if

$$P \cap \mathbb{Z}^n \subseteq \{x \in \mathbb{R}^n : a^T x \leq b\}.$$

#### Definition 28.2

*Cutting plane for MIP]* Let  $P \subseteq \mathbb{R}^n$  be a polyhedron. An inequality  $a^T x \leq b$  is called a cutting plane if

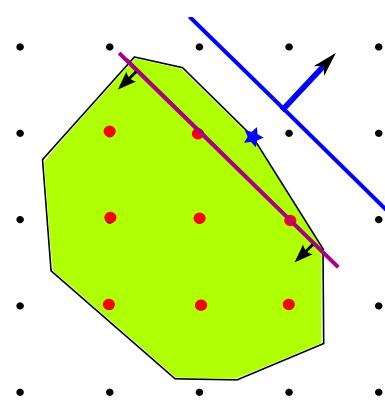
$$P \cap (\mathbb{Z}^{n_1} \times \mathbb{R}^{n_2}) \subseteq \{x \in \mathbb{R}^n : a^T x \leq b\},$$

where we are assuming that in the MIP only the first  $n_1$  variables must be integers ( $n = n_1 + n_2$ ).

### 28.1.2. Cutting plane algorithm

#### Generic cutting plane algorithm

1. Solve LP (continuous relaxation of MILP).
2. If solution of LP is **fractional**: add cutting plane and go to (1.)
3. If solution of LP is **integral**: STOP.



© CPalgorithm6.pdf<sup>1</sup>

<sup>1</sup>CPalgorithm6.pdf, from CPalgorithm6.pdf. CPalgorithm6.pdf, CPalgorithm6.pdf.

### 28.1.3. How to compute cutting planes

---

Two approaches:

1. Computing cutting planes for general IPs.
  - From “Algebraic” properties: CG cuts, MIR inequalities, functional cuts, etc.
  - From “Geometric” properties: lattice-free cuts, etc.
2. Computing cutting planes for specific IPs.
  - Knapsack problem, Node packing, etc. (many many other examples...)

## 28.2 Computing cutting planes for general IPs

---

### 28.2.1. Chvátal-Gomory cuts (for pure integer programs)

---

**Definition 28.3: Chvátal-Gomory cut for  $P$**

Let  $P \subseteq \mathbb{R}^n$  be a polyhedron. Let  $a \in \mathbb{Z}^n$ ,  $b \in \mathbb{R}$  and let  $a^T x \leq b$  be a valid inequality for  $P$ . Then the inequality

$$a^T x \leq \lfloor b \rfloor$$

is called a Chvátal-Gomory cut.

**Remark 28.4**

Some examples of CG cuts are: blossom inequalities for the matching problem, clique inequalities for the independent set problem, Gomory’s fractional cut.

### 28.2.1.1. A nice property of CG cuts

#### Definition 28.5: Chvátal-Gomory closure of $P$

Let  $P \subseteq \mathbb{R}^n$  be a polyhedron. Then the set

$$P' = P \cap \bigcap_{\substack{\alpha^T x \leq \beta \\ \text{is a CG cut for } P}} \{x \in \mathbb{R}^n : \alpha^T x \leq \beta\}$$

is called a Chvátal-Gomory closure.

#### Theorem 28.6: Finiteness of the CG cuts procedure

Let  $P_0$  be a rational polyhedron and for  $k \in \mathbb{Z}_+$  define  $P^{k+1} = (P^k)'$ . Then

1. For all  $k \in \mathbb{Z}_+$ ,  $P^k$  is again a rational polyhedron.
2. There exists  $t \in \mathbb{Z}_+$  such that  $P^t = P_t$ .

### 28.2.2. Cutting planes from the Simplex tableau

Assume  $P = \{x \in \mathbb{R}^n : Ax = b, x \geq 0\}$  where  $A \in \mathbb{R}^{m \times n}$  is a full-row rank matrix. Let  $B, N$  denote the basic and nonbasic variables defining a vertex  $(\hat{x}_B, \hat{x}_N)$  of  $P$  (where  $\hat{x}_N = 0$ ). You can write the constraints defining  $P$  in terms of the basis  $B$ :

$$\begin{aligned} x_B &= \bar{b} - \bar{A}_N x_N \\ x_B, x_N &\geq 0, \end{aligned}$$

where  $\bar{b} = A_B^{-1}b$  and  $\bar{A}_N = A_B^{-1}A_N$ .

Denote  $\bar{b} = (\bar{b}_i)_{i \in B}$  and  $\bar{A}_N = (\bar{a}_{ij})_{i \in B, j \in N}$ . Assume that  $\bar{b}_i \notin \mathbb{Z}$ , so the vertex is fractional (that is,  $(\hat{x}_B, \hat{x}_N) \notin \mathbb{Z}^n$ ), and therefore, we would want to cut off that LP solution.

#### Remark 28.7

Recall that the vertex  $(\hat{x}_B, \hat{x}_N)$  is the only feasible point in  $P$  satisfying  $x_N = 0$ . We will use this fact in order to derive some cutting planes.

### 28.2.2.1. A simple inequality

---

The following is a valid inequality that cuts off the fractional vertex:

$$\sum_{j \in N} x_j \geq 1.$$

### 28.2.3. A stronger inequality

---

Let  $N_f = \{j \in N : \bar{a}_{ij} \text{ is fractional}\}$ . Then the following is a valid inequality that cuts off the fractional vertex:

$$\sum_{j \in N_f} x_j \geq 1.$$

#### 28.2.3.1. Gomory's fractional cut

---

The following inequality can be derived as a CG cut:

$$\sum_{j \in N} (\bar{a}_{ij} - \lfloor \bar{a}_{ij} \rfloor) x_j \geq (\bar{b}_i - \lfloor \bar{b}_i \rfloor). \quad (28.1)$$

Let  $f = \bar{b}_i - \lfloor \bar{b}_i \rfloor$ . Then this can be equivalently written as

$$\sum_{j \in N} \left( \frac{\bar{a}_{ij} - \lfloor \bar{a}_{ij} \rfloor}{f} \right) x_j \geq 1. \quad (28.2)$$

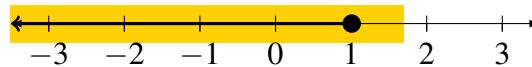
It can be verified that this valid inequality cuts off the fractional vertex.

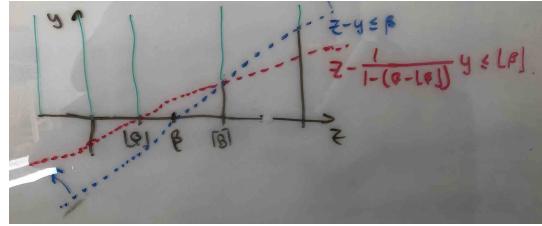
Notice that the key in deriving this inequality is looking at the set

$$T_b^0 := \{z \in \mathbb{Z} : z \leq b\} \quad (28.3)$$

where

$$\text{conv}(T_b^0) = \{z \in \mathbb{R} : z \leq \lfloor b \rfloor\} \quad (28.4)$$



© MIR-Cut<sup>2</sup>**Figure 28.1: MIR-Cut**

## 28.3 Mixed-integer rounding cuts (MIR)

### 28.3.1. Basic MIR inequality

Consider the set

$$T_b^1 = \{(z, y) \in \mathbb{Z} \times \mathbb{R}_+: z - y \leq b\} \quad (28.1)$$

Then the inequality

$$z - \frac{1}{1 - (b - \lfloor b \rfloor)}y \leq \lfloor b \rfloor. \quad (28.2)$$

We will rewrite this as

$$z - \frac{1}{1-f}y \leq \lfloor b \rfloor. \quad (28.3)$$

### 28.3.2. GMIC Derivation from MIR cut

#### Theorem 28.8: Gomory Mixed Integer Cut - Pure Integer Case

Suppose we have  $x \in \mathbb{Z}_+^n$  with  $x_i \in \mathbb{Z}$ , along with the equation

$$\sum_{i \in I} a_i x_i = b. \quad (28.4)$$

Then

$$\sum_{i \in I^-} \frac{\{a_i\}}{f} x_i + \sum_{i \in I^+} \frac{1 - \{a_i\}}{1-f} x_i \geq 1 \quad (28.5)$$

is valid for this set.

### Proof.

<sup>2</sup>MIR-Cut, from MIR-Cut. MIR-Cut, MIR-Cut.

Let  $I^- = \{i \in I : \{a_i\} \leq f\}$ ,  $I^+ = \{i \in I : \{a_i\} > f\}$ .

$$\sum_{i \in I^-} a_i x_i + \sum_{i \in I^+} a_i x_i = b \quad (28.6)$$

Now we rewrite this as

$$\sum_{i \in I^-} (\lfloor a_i \rfloor + \{a_i\}) x_i + \sum_{i \in I^+} (\lceil a_i \rceil - (1 - \{a_i\})) x_i = b \quad (28.7)$$

Next, we remove fractional parts that are non-negative and keep ones that have negative coefficients.

$$\underbrace{\sum_{i \in I^-} \lfloor a_i \rfloor x_i + \sum_{i \in I^+} \lceil a_i \rceil x_i}_{z} - \underbrace{\left( \sum_{i \in I^+} (1 - \{a_i\}) x_i \right)}_{y \geq 0} \leq b \quad (28.8)$$

Applying the MIR cut, we obtain

$$\sum_{i \in I^-} \lfloor a_i \rfloor x_i + \sum_{i \in I^+} \lceil a_i \rceil x_i - \sum_{i \in I^+} \frac{1 - \{a_i\}}{1 - f} x_i \leq \lfloor b \rfloor \quad (28.9)$$

Now we subtract (28.2.2) from (29.2.2) to obtain

$$\sum_{i \in I^-} \underbrace{(a_i - \lfloor a_i \rfloor)}_{\{a_i\}} x_i + \sum_{i \in I^+} \underbrace{(a_i - \lceil a_i \rceil)}_{\{a_i\}-1} x_i + \sum_{i \in I^+} \frac{1 - \{a_i\}}{1 - f} x_i \geq b - \lfloor b \rfloor \quad (28.10)$$

Which rearranges to

$$\sum_{i \in I^-} \{a_i\} x_i + \sum_{i \in I^+} (1 - \{a_i\}) \left( -1 + \frac{1}{1 - f} \right) x_i \geq f \quad (28.11)$$

Lastly, we divide through by  $f$ . Note that

$$\frac{1}{f} \left( -1 + \frac{1}{1 - f} \right) = \frac{1}{f} \left( \frac{-(1-f)+1}{1-f} \right) = \frac{1}{f} \left( \frac{f}{1-f} \right) = \frac{1}{1-f}$$

Hence, we obtain

$$\sum_{i \in I^-} \frac{\{a_i\}}{f} x_i + \sum_{i \in I^+} \frac{1 - \{a_i\}}{1 - f} x_i \geq 1$$



We now repeat the proof and show the mixed integer case to showcase the more general version.

**Theorem 28.9: Gomory Mixed Integer Cut - Mixed integer case**

Suppose we have  $x \in \mathbb{R}_+^n$  with  $x_i \in \mathbb{Z}$  for all  $i \in I$ , along with the equation

$$\sum_{i \in I} a_i x_i + \sum_{i \in C} a_i x_i = b. \quad (28.12)$$

Then

$$\sum_{i \in I^-} \frac{\{a_i\}}{f} x_i + \sum_{i \in I^+} \frac{1 - \{a_i\}}{1-f} x_i + \sum_{i \in C^+} \frac{a_i}{f} x_i + \sum_{i \in C^-} \frac{-a_i}{1-f} x_i \geq 1 \quad (28.13)$$

is valid for this set.

**Proof.**

Let  $I^- = \{i \in I : \{a_i\} \leq f\}$ ,  $I^+ = \{i \in I : \{a_i\} > f\}$  and let  $C^- = \{i \in C : a_i < 0\}$  and  $C^+ = \{i \in C : a_i \geq 0\}$ .

$$\sum_{i \in I^-} a_i x_i + \sum_{i \in I^+} a_i x_i - \sum_{i \in C^-} (-a_i) x_i + \sum_{i \in C^+} a_i x_i = b \quad (28.14)$$

Now we rewrite this as

$$\sum_{i \in I^-} (\lfloor a_i \rfloor + \{a_i\}) x_i + \sum_{i \in I^+} (\lceil a_i \rceil - (1 - \{a_i\})) x_i - \sum_{i \in C^-} (-a_i) x_i + \sum_{i \in C^+} a_i x_i = b \quad (28.15)$$

Next, we remove fractional parts that are non-negative and keep ones that have negative coefficients.

$$\underbrace{\sum_{i \in I^-} \lfloor a_i \rfloor x_i + \sum_{i \in I^+} \lceil a_i \rceil x_i}_{z} - \underbrace{\left( \sum_{i \in I^+} (1 - \{a_i\}) x_i + \sum_{i \in C^-} (-a_i) x_i \right)}_{y \geq 0} \leq b \quad (28.16)$$

Applying the MIR cut, we obtain

$$\sum_{i \in I^-} \lfloor a_i \rfloor x_i + \sum_{i \in I^+} \lceil a_i \rceil x_i - \sum_{i \in I^+} \frac{1 - \{a_i\}}{1-f} x_i + \sum_{i \in C^-} \frac{a_i}{1-f} x_i \leq \lfloor b \rfloor \quad (28.17)$$

Now we subtract (28.17) from (28.16) to obtain

$$\sum_{i \in I^-} \underbrace{(a_i - \lfloor a_i \rfloor)}_{\{a_i\}} x_i + \sum_{i \in I^+} \underbrace{(a_i - \lceil a_i \rceil)}_{\{a_i\}-1} x_i + \sum_{i \in I^+} \frac{1 - \{a_i\}}{1-f} x_i + \sum_{i \in C^+} a_i x_i + \sum_{i \in C^-} a_i x_i + \sum_{i \in C^-} \frac{a_i}{1-f} x_i \geq \underbrace{b - \lfloor b \rfloor}_f \quad (28.18)$$

Which rearranges to

$$\sum_{i \in I^-} \{a_i\} x_i + \sum_{i \in I^+} (1 - \{a_i\}) \left( -1 + \frac{1}{1-f} \right) x_i + \sum_{i \in C^+} a_i x_i + \sum_{i \in C^-} (-a_i) \left( -1 + \frac{1}{1-f} \right) x_i \geq f \quad (28.19)$$

Lastly, we divide through by  $f$ . Note that

$$\frac{1}{f} \left( -1 + \frac{1}{1-f} \right) = \frac{1}{f} \left( \frac{-(1-f)+1}{1-f} \right) = \frac{1}{f} \left( \frac{f}{1-f} \right) = \frac{1}{1-f}$$

Hence, we obtain

$$\sum_{i \in I^-} \frac{\{a_i\}}{f} x_i + \sum_{i \in I^+} \frac{1 - \{a_i\}}{1-f} x_i + \sum_{i \in C^+} \frac{a_i}{f} x_i + \sum_{i \in C^-} \frac{-a_i}{1-f} x_i \geq 1$$



If we let  $\pi: \mathbb{R} \rightarrow \mathbb{R}_+$  map coefficients on integer variables to coefficients in the inequality and let  $\psi: \mathbb{R} \rightarrow \mathbb{R}_+$  map coefficients on the continuous variables to coefficients on the inequalities, then these functions look like

Picture!!

**Figure 28.2:** Gomory mixed integer cut (GMIC) described by the pair of functions  $(\pi, \psi)$ .

## 28.4 Cutting planes from lattice free sets

### 28.4.1. The general case

#### Definition 28.10: Lattice-free sets

A set  $L \subseteq \mathbb{R}^n$  is a lattice-free set if it does not contain any integral vector in its (topological) interior, that is,  $\text{int}(L) \cap \mathbb{Z}^n = \emptyset$ .

Let  $P$  be a polyhedron and let  $L$  be a lattice-free convex set. Then, we can derive cutting planes from  $L$  by using the following fact:

$$P \cap \mathbb{Z}^n \subseteq P \setminus \text{int}(L).$$

Such a cutting plane is called a cutting plane derived from a lattice-free set.

#### Remark 28.11

It suffice to consider only the cutting planes defining facets of  $\text{conv}(P \setminus \text{int}(L))$  as all the cuts not defining these facets are redundant.

#### Definition 28.12: Maximal lattice-free convex sets

A maximal lattice-free is a lattice-free convex set that is not strictly contained in any other lattice-free convex set.

Maximal lattice-free convex sets are important since they give stronger cuts, since  $L' \subseteq L$  implies  $P \setminus \text{int}(L) \subseteq P \setminus \text{int}(L')$ . The following is a nice property of such sets:

### Theorem 28.13

*L is a maximal lattice-free convex set if and only if L is a polyhedron satisfying certain “simple characterization”.*

### Remark 28.14: T

*The fact that all maximal lattice-free convex sets are polyhedra is useful because if L is a polyhedron, then cutting planes for the set  $P \setminus \text{int}(L)$  are likely ‘easy’ to obtain.*

## 28.4.2. Split cuts

---

A special case of a maximal lattice-free convex set is the case of split sets.

### Definition 28.15: Split set, split cuts

*Let  $\pi \in \mathbb{Z}^n$  and let  $\pi_0 \in \mathbb{Z}$ . Then, a split set is a set of the form*

$$\{x \in \mathbb{R}^n : \pi_0 < \pi^T x < \pi_0 + 1\}.$$

*A split cut is any cutting plane valid for  $P \setminus S$ , where S is some split set.*

### Remark 28.16: T

*The set  $P \setminus S$  can be seen as a disjunction. Let  $S = \{x \in \mathbb{R}^n : \pi_0 < \pi^T x < \pi_0 + 1\}$ , then*

$$P \setminus S = \{x \in P : \pi^T x \leq \pi_0\} \cup \{x \in P : \pi_0 + 1 \leq \pi^T x\}.$$

*In general, disjunctions as the one given by a split set or more general ones are very useful to derive cutting planes for integer programs.*

### 28.4.3. MIR inequalities from one-row relaxations

---

Let  $P$  be a polyhedron. We want to find valid inequalities for  $P \cap (\mathbb{Z}^{|I|} \times \mathbb{R}^{|J|})$ .

#### 28.4.3.1. One-row relaxation

---

A set of the form

$$Q = \left\{ (x, y) \in \mathbb{R}^{|I|} \times \mathbb{R}^{|J|} : \sum_{i \in I} a_i x_i + \sum_{j \in J} c_j y_j \geq b, x, y \geq 0 \right\}$$

is a one-row relaxation of  $P$  if  $P \subseteq Q$ .

**Remark 28.17: O**

e-row relaxations can be constructed by using any valid inequality for  $P$ . In particular one could obtain a valid inequality by combining rows of the matrix and vector defining  $P$ .

#### 28.4.3.2. Applying the basic MIR inequality

---

We first relax the inequality defining  $Q$ . Let  $I' \subseteq I$  and consider the following mixed-integer set:

$$B = \left\{ (x, y) \in \mathbb{Z}^{|I|} \times \mathbb{R}^{|J|} : \left( \sum_{i \in I \setminus I'} x_i + \sum_{i \in I} \lfloor a_i \rfloor x_i \right) + \left( \sum_{i \in I'} (a_i - \lfloor a_i \rfloor) x_i + \sum_{j \in J} \max\{0, c_j\} y_j \right) \geq b \right\}.$$

Since the first part of the l.h.s. of the inequality is integral and the second part is nonnegative, we can apply the procedure described in Section 29.2.1. We obtain the following valid inequality for  $B$ :

$$\left( \sum_{i \in I'} (a_i - \lfloor a_i \rfloor) x_i + \sum_{j \in J} \max\{0, c_j\} y_j \right) \geq (b - \lfloor b \rfloor) \left( \lceil b \rceil - \left( \sum_{i \in I \setminus I'} x_i + \sum_{i \in I} \lfloor a_i \rfloor x_i \right) \right).$$

**Remark 28.18: T**

The above inequality is valid for  $P \cap (\mathbb{Z}^{|I|} \times \mathbb{R}^{|J|})$ , for all  $I' \subseteq I$ . The set  $I' = \{i \in I : (a_i - \lfloor a_i \rfloor) < (b - \lfloor b \rfloor)\}$  gives the strongest inequality of this form.

## 28.5 Gomory Mixed-integer cut (GMI)

---

Consider the one-row relaxation  $Q = \{(x, y) \in \mathbb{Z}^{|I|} \times \mathbb{R}^{|J|} : \sum_{i \in I} a_i x_i + \sum_{j \in J} c_j y_j = b, x, y \geq 0\}$ . Denote  $f_0 = b - \lfloor b \rfloor$  and for  $i \in I$  denote  $f_i = a_i - \lfloor a_i \rfloor$ .

We will assume that  $0 < f_0 < 1$ . In this case, the Gomory mixed-integer cut (GMI) is given by

$$\sum_{\substack{i \in I \\ f_i \leq f_0}} \frac{f_i}{f_0} x_i + \sum_{\substack{i \in I \\ f_i > f_0}} \frac{1-f_i}{1-f_0} x_i + \sum_{\substack{j \in J \\ c_j > 0}} \frac{c_j}{f_0} y_j + \sum_{\substack{j \in J \\ c_j < 0}} \frac{c_j}{1-f_0} y_j \geq 1.$$

**Remark 28.19:**

- The validity of GMI cuts follows from the fact that they are also split cuts.
- In the pure integer programming case (that is,  $J = \emptyset$ ), GMI gives a cut that is stronger than the Gomory's fractional cut.



# 29. Cutting Planes

---

## 29.1 Introduction

---

### 29.1.1. Cutting planes

---

#### Definition 29.1: Cutting plane for IP

Let  $P \subseteq \mathbb{R}^n$  be a polyhedron. An inequality  $a^T x \leq b$  is called a cutting plane if

$$P \cap \mathbb{Z}^n \subseteq \{x \in \mathbb{R}^n : a^T x \leq b\}.$$

#### Definition 29.2: Cutting plane for MIP

Let  $P \subseteq \mathbb{R}^n$  be a polyhedron. An inequality  $a^T x \leq b$  is called a cutting plane if

$$P \cap (\mathbb{Z}^{n_1} \times \mathbb{R}^{n_2}) \subseteq \{x \in \mathbb{R}^n : a^T x \leq b\},$$

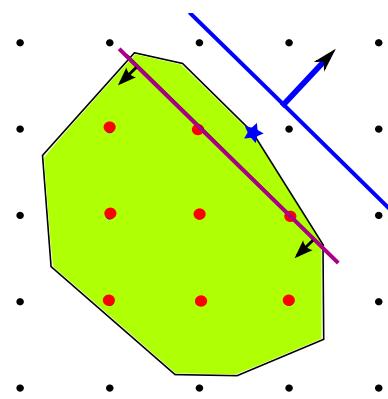
where we are assuming that in the MIP only the first  $n_1$  variables must be integers ( $n = n_1 + n_2$ ).

### 29.1.2. Cutting plane algorithm

---

#### Generic cutting plane algorithm

1. Solve LP (continuous relaxation of MILP).
2. If solution of LP is **fractional**: add cutting plane and go to (1.)
3. If solution of LP is **integral**: STOP.



### 29.1.3. How to compute cutting planes

---

Two approaches:

1. Computing cutting planes for general IPs.
  - From “Algebraic” properties: CG cuts, MIR inequalities, functional cuts, etc.
  - From “Geometric” properties: lattice-free cuts, etc.
2. Computing cutting planes for specific IPs.
  - Knapsack problem, Node packing, etc. (many many other examples...)

### 29.1.4. Cutting planes from the Simplex tableau

---

Assume  $P = \{x \in \mathbb{R}^n : Ax = b, x \geq 0\}$  where  $A \in \mathbb{R}^{m \times n}$  is a full-row rank matrix. Let  $B, N$  denote the basic and nonbasic variables defining a vertex  $(\hat{x}_B, \hat{x}_N)$  of  $P$  (where  $\hat{x}_N = 0$ ). You can write the constraints defining  $P$  in terms of the basis  $B$ :

$$\begin{aligned} x_B &= \bar{b} - \bar{A}_N x_N \\ x_B, x_N &\geq 0, \end{aligned}$$

where  $\bar{b} = A_B^{-1}b$  and  $\bar{A}_N = A_B^{-1}A_N$ .

Denote  $\bar{b} = (\bar{b}_i)_{i \in B}$  and  $\bar{A}_N = (\bar{a}_{ij})_{i \in B, j \in N}$ . Assume that  $\bar{b}_i \notin \mathbb{Z}$ , so the vertex is fractional (that is,  $(\hat{x}_B, \hat{x}_N) \notin \mathbb{Z}^n$ ), and therefore, we would want to cut off that LP solution.

#### Remark 29.3

Recall that the vertex  $(\hat{x}_B, \hat{x}_N)$  is the only feasible point in  $P$  satisfying  $x_N = 0$ . We will use this fact in order to derive some cutting planes.

#### 29.1.4.1. Gomory’s fractional cut

---

Suppose we have  $x \in \mathbb{Z}_+^n$  with  $x_i \in \mathbb{Z}$ , along with the equation

$$\sum_{i \in I} a_i x_i = b. \tag{29.1}$$

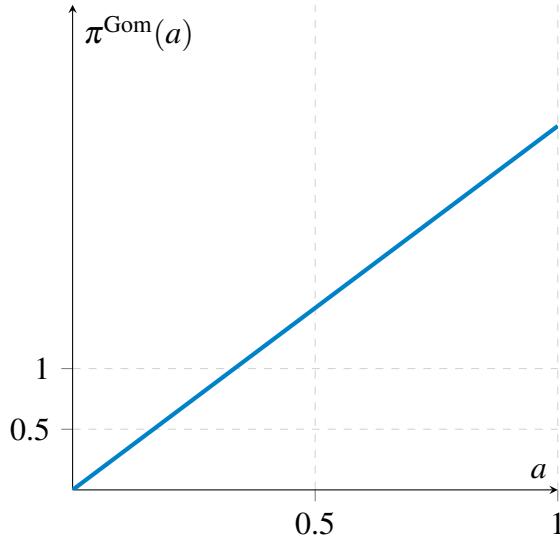
Then the following inequality can be derived as a Chvatal cut:

$$\sum_{j \in N} (\bar{a}_{ij} - \lfloor \bar{a}_{ij} \rfloor) x_j \geq (\bar{b}_i - \lfloor \bar{b}_i \rfloor). \tag{29.2}$$

Let  $f = \bar{b}_i - \lfloor \bar{b}_i \rfloor$ . Then this can be equivalently written as

$$\sum_{j \in N} \left( \frac{\bar{a}_{ij} - \lfloor \bar{a}_{ij} \rfloor}{f} \right) x_j \geq 1. \quad (29.3)$$

It can be verified that this valid inequality cuts off the fractional vertex.



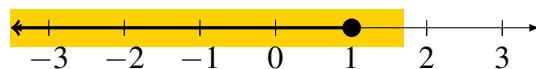
**Figure 29.1: Gomory cut described by the function  $\pi$  for  $f = \frac{1}{3}$ .**

Notice that the key in deriving this inequality is looking at the set

$$T_b^0 := \{z \in \mathbb{Z} : z \leq b\} \quad (29.4)$$

where

$$\text{conv}(T_b^0) = \{z \in \mathbb{R} : z \leq \lfloor b \rfloor\} \quad (29.5)$$



## 29.2 Mixed-integer rounding cuts (MIR)

---

### 29.2.1. Basic MIR inequality

---

Consider the set

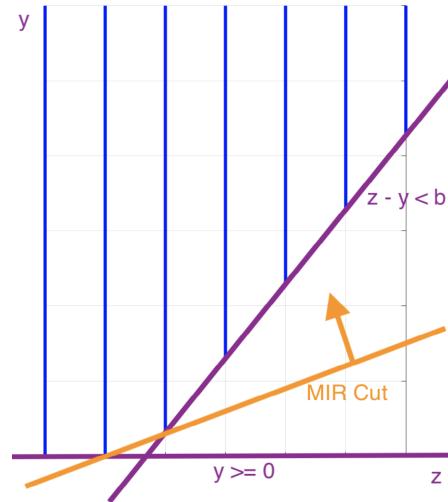
$$T_b^1 = \{(z, y) \in \mathbb{Z} \times \mathbb{R}_+: z - y \leq b\} \quad (29.1)$$

Then the inequality

$$z - \frac{1}{1 - (b - \lfloor b \rfloor)}y \leq \lfloor b \rfloor. \quad (29.2)$$

We will rewrite *MIR Cut* as

$$z - \frac{1}{1-f}y \leq \lfloor b \rfloor. \quad (29.3)$$



### 29.2.2. GMIC Derivation from MIR cut

---

#### Theorem 29.4: Gomory Mixed Integer Cut - Pure Integer Case

Suppose we have  $x \in \mathbb{Z}_+^n$  with  $x_i \in \mathbb{Z}$ , along with the equation

$$\sum_{i \in I} a_i x_i = b. \quad (29.4)$$

Then

$$\sum_{i \in I^-} \frac{\{a_i\}}{f} x_i + \sum_{i \in I^+} \frac{1 - \{a_i\}}{1-f} x_i \geq 1 \quad (29.5)$$

is valid for this set.

**Proof.** Let  $I^- = \{i \in I : \{a_i\} \leq f\}$ ,  $I^+ = \{i \in I : \{a_i\} > f\}$ .

$$\sum_{i \in I^-} a_i x_i + \sum_{i \in I^+} a_i x_i = b \quad (29.6)$$

Now we rewrite this as

$$\sum_{i \in I^-} (\lfloor a_i \rfloor + \{a_i\}) x_i + \sum_{i \in I^+} (\lceil a_i \rceil - (1 - \{a_i\})) x_i = b \quad (29.7)$$

Next, we remove fractional parts that are non-negative and keep ones that have negative coefficients.

$$\underbrace{\sum_{i \in I^-} \lfloor a_i \rfloor x_i + \sum_{i \in I^+} \lceil a_i \rceil x_i}_{z} - \underbrace{\left( \sum_{i \in I^+} (1 - \{a_i\}) x_i \right)}_{y \geq 0} \leq b \quad (29.8)$$

Applying the MIR cut, we obtain

$$\sum_{i \in I^-} \lfloor a_i \rfloor x_i + \sum_{i \in I^+} \lceil a_i \rceil x_i - \sum_{i \in I^+} \frac{1 - \{a_i\}}{1-f} x_i \leq \lfloor b \rfloor \quad (29.9)$$

Now we subtract (29.2.2) from (29.2.2) to obtain

$$\sum_{i \in I^-} \underbrace{(a_i - \lfloor a_i \rfloor)}_{\{a_i\}} x_i + \sum_{i \in I^+} \underbrace{(a_i - \lceil a_i \rceil)}_{\{a_i\}-1} x_i + \sum_{i \in I^+} \frac{1 - \{a_i\}}{1-f} x_i \geq \underbrace{b - \lfloor b \rfloor}_f \quad (29.10)$$

Which rearranges to

$$\sum_{i \in I^-} \{a_i\} x_i + \sum_{i \in I^+} (1 - \{a_i\}) \left( -1 + \frac{1}{1-f} \right) x_i \geq f \quad (29.11)$$

Lastly, we divide through by  $f$ . Note that

$$\frac{1}{f} \left( -1 + \frac{1}{1-f} \right) = \frac{1}{f} \left( \frac{-(1-f)+1}{1-f} \right) = \frac{1}{f} \left( \frac{f}{1-f} \right) = \frac{1}{1-f}$$

Hence, we obtain

$$\sum_{i \in I^-} \frac{\{a_i\}}{f} x_i + \sum_{i \in I^+} \frac{1 - \{a_i\}}{1-f} x_i \geq 1$$



We now repeat the proof and show the mixed integer case to showcase the more general version.

### Theorem 29.5: Gomory Mixed Integer Cut - Mixed integer case

Suppose we have  $x \in \mathbb{R}_+^n$  with  $x_i \in \mathbb{Z}$  for all  $i \in I$ , along with the equation

$$\sum_{i \in I} a_i x_i + \sum_{i \in C} a_i x_i = b. \quad (29.12)$$

Then

$$\sum_{i \in I^-} \frac{\{a_i\}}{f} x_i + \sum_{i \in I^+} \frac{1 - \{a_i\}}{1-f} x_i + \sum_{i \in C^+} \frac{a_i}{f} x_i + \sum_{i \in C^-} \frac{-a_i}{1-f} x_i \geq 1 \quad (29.13)$$

is valid for this set.

**Proof.** Let  $I^- = \{i \in I : \{a_i\} \leq f\}$ ,  $I^+ = \{i \in I : \{a_i\} > f\}$  and let  $C^- = \{i \in C : a_i < 0\}$  and  $C^+ = \{i \in C : a_i \geq 0\}$ .

$$\sum_{i \in I^-} a_i x_i + \sum_{i \in I^+} a_i x_i - \sum_{i \in C^-} (-a_i) x_i + \sum_{i \in C^+} a_i x_i = b$$

Now we rewrite this as

$$\sum_{i \in I^-} (\lfloor a_i \rfloor + \{a_i\}) x_i + \sum_{i \in I^+} (\lceil a_i \rceil - (1 - \{a_i\})) x_i - \sum_{i \in C^-} (-a_i) x_i + \sum_{i \in C^+} a_i x_i = b$$

Next, we remove fractional parts that are non-negative and keep ones that have negative coefficients.

$$\underbrace{\sum_{i \in I^-} \lfloor a_i \rfloor x_i + \sum_{i \in I^+} \lceil a_i \rceil x_i}_{z} - \underbrace{\left( \sum_{i \in I^+} (1 - \{a_i\}) x_i + \sum_{i \in C^-} (-a_i) x_i \right)}_{y \geq 0} \leq b \quad (29.14)$$

Applying the MIR cut, we obtain

$$\sum_{i \in I^-} \lfloor a_i \rfloor x_i + \sum_{i \in I^+} \lceil a_i \rceil x_i - \sum_{i \in I^+} \frac{1 - \{a_i\}}{1-f} x_i + \sum_{i \in C^-} \frac{a_i}{1-f} x_i \leq \lfloor b \rfloor$$

Now we subtract (29.2.2) from (29.2.2) to obtain

$$\sum_{i \in I^-} \underbrace{(a_i - \lfloor a_i \rfloor)}_{\{a_i\}} x_i + \sum_{i \in I^+} \left( \underbrace{(a_i - \lceil a_i \rceil)}_{\{a_i\}-1} x_i + \frac{1 - \{a_i\}}{1-f} x_i \right) + \sum_{i \in C^+} a_i x_i + \sum_{i \in C^-} \left( a_i x_i + \frac{a_i}{1-f} x_i \right) \geq \underbrace{b - \lfloor b \rfloor}_f$$

Which rearranges to

$$\sum_{i \in I^-} \{a_i\} x_i + \sum_{i \in I^+} (1 - \{a_i\}) \left( -1 + \frac{1}{1-f} \right) x_i + \sum_{i \in C^+} a_i x_i + \sum_{i \in C^-} (-a_i) \left( -1 + \frac{1}{1-f} \right) x_i \geq f \quad (29.15)$$

Lastly, we divide through by  $f$ . Note that

$$\frac{1}{f} \left( -1 + \frac{1}{1-f} \right) = \frac{1}{f} \left( \frac{-(1-f)+1}{1-f} \right) = \frac{1}{f} \left( \frac{f}{1-f} \right) = \frac{1}{1-f}$$

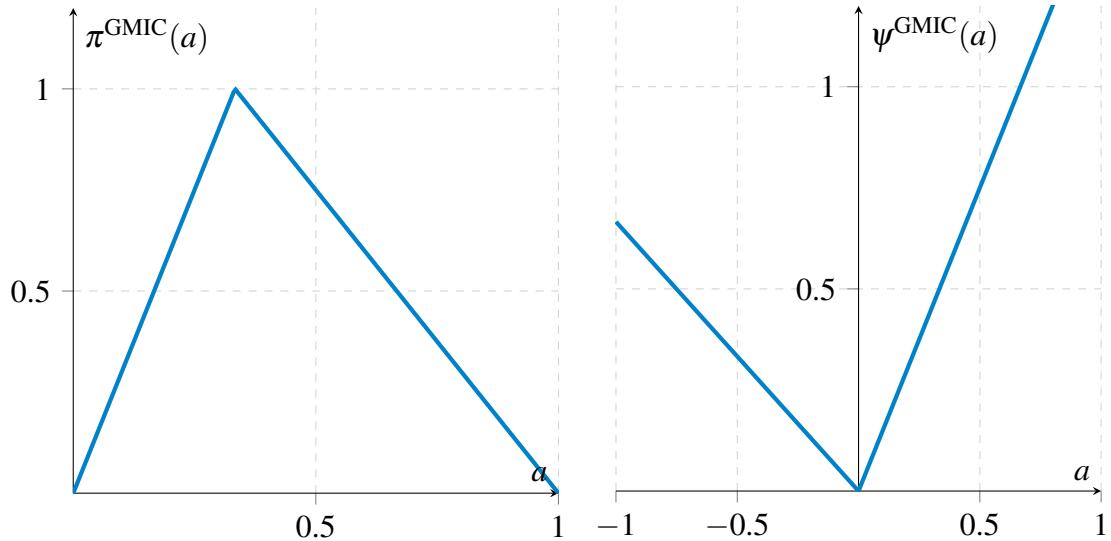
Hence, we obtain

$$\sum_{i \in I^-} \frac{\{a_i\}}{f} x_i + \sum_{i \in I^+} \frac{1 - \{a_i\}}{1-f} x_i + \sum_{i \in C^+} \frac{a_i}{f} x_i + \sum_{i \in C^-} \frac{-a_i}{1-f} x_i \geq 1$$

$$\sum_{i \in I^-} \frac{\{a_i\}}{f} x_i + \sum_{i \in I^+} \frac{1 - \{a_i\}}{1-f} x_i + \sum_{i \in C^+} \frac{a_i}{f} x_i + \sum_{i \in C^-} \frac{-a_i}{1-f} x_i \geq 1$$



If we let  $\pi: \mathbb{R} \rightarrow \mathbb{R}_+$  map coefficients on integer variables to coefficients in the inequality and let  $\psi: \mathbb{R} \rightarrow \mathbb{R}_+$  map coefficients on the continuous variables to coefficients on the inequalities, then these functions look like



**Figure 29.2:** Gomory mixed integer cut (GMIC) described by the pair of functions  $(\pi, \psi)$  for  $f = \frac{1}{3}$ .

## 29.3 Split Cuts

---

[http://www.princeton.edu/~aaa/Public/Teaching/ORF523/ORF523\\_Lec17\\_guest.pdf](http://www.princeton.edu/~aaa/Public/Teaching/ORF523/ORF523_Lec17_guest.pdf)

Split cuts are any inequality valid for  $P \setminus \text{intr}(S)$ .

### Theorem 29.6: Containment in $P^{\pi, \pi_0}$ (Bonami, 2012)

Suppose  $\bar{x} \in P \cap \text{intr}(S)$  where  $\text{intr}(S) := \{x : \pi_0 < \pi^\top x < \pi_0 + 1\}$ . Then  $\bar{x} \in P^{\pi, \pi_0}$  if and only if there exists an  $x^2 \in \Pi_2$  such that

$$b - Ax^2 \leq \frac{b - A\bar{x}}{\pi_0 - \pi^\top \bar{x}}. \quad (29.1)$$

#### Proof. ( $\Rightarrow$ )

Suppose  $\bar{x} \in P^{\pi, \pi_0}$ . Then there exist  $x^1 \in \Pi_1, x^2 \in \Pi_2$  such that

$$\bar{x} = \lambda x^1 + (1 - \lambda)x^2$$

for some  $\lambda \in (0, 1)$ . That is,

$$\begin{cases} Ax^1 \leq b, & \pi^\top x^1 \leq \pi_0, \\ Ax^2 \leq b, & \pi^\top x^2 \geq \pi_0 + 1, \\ x = \lambda x^1 + (1 - \lambda)x^2. \end{cases} \quad (\text{Convex Combination System})$$

Equivalently, we can write

$$x^1 = \frac{1}{\lambda}(\bar{x} - (1 - \lambda)x^2)$$

and hence

$$\begin{aligned} & A\left(\frac{1}{\lambda}(\bar{x} - (1 - \lambda)x^2)\right) \leq b \\ \iff & A\bar{x} - (1 - \lambda)Ax^2 \leq \lambda b = b - (1 - \lambda)b \\ \iff & (1 - \lambda)(b - Ax^2) \leq b - A\bar{x} \\ \iff & b - Ax^2 \leq \frac{b - A\bar{x}}{1 - \lambda} \end{aligned}$$

Rearranging yields the equivalent system

$$\begin{cases} b - Ax^2 \leq \frac{b - A\bar{x}}{1 - \lambda}, & \pi^\top x^1 \leq \pi_0, \\ Ax^2 \leq b, & \pi^\top x^2 \geq \pi_0 + 1, \\ \bar{x} = \lambda x^1 + (1 - \lambda)x^2. \end{cases} \quad (\text{Equivalent System})$$

Now, since the interval  $[\pi^\top x^1, \pi^\top x^2]$  contains the interval  $[\pi_0, \pi_0 + 1]$ , which strictly contains  $\pi^\top \bar{x}$ , we can assume **without loss of generality** that  $\pi^\top x^1 = \pi_0$  and  $\pi^\top x^2 = \pi_0 + 1$ .

Therefore,

$$\pi^\top \bar{x} = \pi^\top(\lambda x^1 + (1 - \lambda)x^2) = \lambda \pi_0 + (1 - \lambda)(\pi_0 + 1) = \lambda + (\pi_0 + 1).$$

Thus

$$\lambda = \pi^\top \bar{x} - (\pi_0 + 1) \quad \text{and} \quad (1 - \lambda) = \pi_0 - \pi^\top \bar{x}. \quad (29.2)$$

Therefore, combining with (Equivalent System), we obtain the desired inequality (29.1).

( $\Leftarrow$ )

Suppose instead there exist  $x^2 \in \Pi^2$  such that (29.1) holds. Then by choosing  $\lambda = \pi^\top \bar{x} - (\pi_0 + 1)$  (according to (29.2)), we see that (Equivalent System) is satisfied. Since (Equivalent System) is equivalent to (Convex Combination System), we see that  $x^1$  defined by this system is feasible for  $\Pi^1$ . Hence,  $\bar{x} \in P^{\pi, \pi_0}$ .



Consider the cut generating linear program (or LP to generate the  $\tilde{x}$  point).

$$\max_x \pi x \quad (29.3)$$

$$b - \frac{b - A\bar{x}}{\pi\bar{x} - \pi_0} \leq Ax \quad (29.4)$$

$$Ax \leq b \quad (29.5)$$

And the point  $\bar{x}$  is cut off if  $\pi x^* \geq \pi_0 + 1$ .

Let's write the dual of the LP.

$$\min(u^1 - u^2)b + u^2 \frac{b - A\bar{x}}{\pi\bar{x} - \pi_0} \quad (29.6)$$

$$(u^1 - u^2)A = \pi \quad (29.7)$$

$$u^1, u^2 \geq 0. \quad (29.8)$$

### Theorem 29.7: All split inequalities

Let  $P = \{x \in \mathbb{R}^n : Ax \leq b\}$ . Suppose that  $u \in \mathbb{R}^m$  satisfies  $uA_I = \pi \in \mathbb{Z}^I$ ,  $uA_C = 0$ . Let  $u = u^+ - u^-$  where  $u_i^+ = \max\{u_i, 0\}$  and  $u_i^- = \max\{-u_i, 0\}$ . Then

$$\frac{u^+(b - Ax)}{f} + \frac{u^-(b - Ax)}{1-f} \geq 1 \quad (29.9)$$

is a valid inequality for  $P^{\pi, \pi_0}$ , where  $f = \{ub\}$  and  $\pi_0 = \lfloor ub \rfloor$ .

**Proof.** Suppose that  $x \in P \cap (\mathbb{Z}^I \times \mathbb{R}^C)$ . Then

$$\begin{aligned} ub &= u^+b - u^-b \\ &\geq u^+Ax - u^-b \\ &= uAx + u^-Ax - u^-b \\ &= uAx - u^-(b - Ax) \\ &= \underbrace{\pi^\top x_I}_z - \underbrace{u^-(b - Ax)}_{y \geq 0} \end{aligned}$$

Applying the MIR inequality and rearranging yields (29.9). ♠

Let  $B_\pi$  be the set of all basic solutions to  $uA_I = \pi$ ,  $uA_C = 0$ .

Then the  $P^{\pi, \pi_0} = \cap(\text{split inequalities with } \pi_0 < ub < \pi_0 + 1)$ .

## 30. Closures

---

<https://archive.siam.org/meetings/op08/Weissmantel.pdf>



## 31. Dantzig Wolfe

---

0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1
0	1	0	0	1	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	1	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	1	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	-2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	-3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	-4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	-5	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	-1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	-1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	-1	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	-4	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	-3	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	-1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	-2



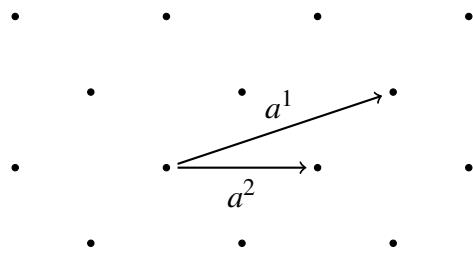
## 32. Lattices, IP in fixed dimensions

---

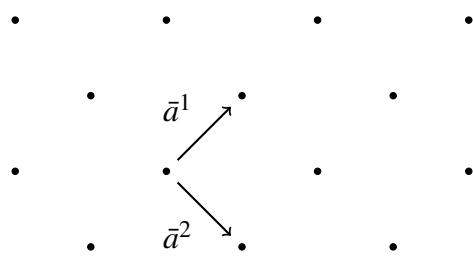
A lattice is a rational linear transformation of the set of integer points. That is, for a rational matrix  $A \in \mathbb{Q}^{m \times n}$ , the a lattice is the set  $\Lambda = A\mathbb{Z}^n = \{x : x = \sum_{i=1}^n a^i z_i, z \in \mathbb{Z}^n\}$  where  $a^i$  are the columns of the matrix  $A$ . The dimension of the lattice  $\Lambda$  is the dimension of the affine hull of  $\Lambda$ . If  $d = \dim(\text{aff}(\Lambda))$ , then there exists a rational matrix  $A' \in \mathbb{Q}^{d \times n}$  such that  $\Lambda = A'\mathbb{Z}^n$ . For this reason, and the fact that we can project lattices into a more natural dimension, we typically assume that  $A \in \mathbb{Q}^{n \times n}$  is invertible and that  $\Lambda = A\mathbb{Z}^n$  has dimension  $n$ .

As mentioned in class, there are many possible choices of  $A$  that describe the same lattice  $\Lambda$ . The columns of  $A$  are  $n$ -dimensional vectors that make up the lattice basis, and hence there are many choices of lattice bases.

Here, is a lattice generated by the basis vectors  $a^1 = (3, 1)$ ,  $a^2 = (2, 0)$ .



The LLL algorithm, named for Lenstra, Lenstra, and Lovasz, is a polynomial time algorithm that uses Gram-Schmidt orthogonalization to take a basis matrix  $A$  and make a better basis that is more orthogonal. This makes various properties of the lattice easier to compute such as approximations to the shortest vector problem and the closest vector problem. Some call the produced basis the LLL basis. For Instance, the LLL algorithm reduces the above lattice basis to the new lattice basis  $\bar{a}^1 = (1, 1)$  and  $\bar{a}^2 = (1, -1)$ . This scenario is ideal since the basis vectors are orthogonal.



There are other bases such as the HKZ (or sometimes called the KZ) basis that is composed of the  $n$ -shortest vectors in the lattices (that is, the first vector in the basis the shortest vector, the second is the shortest vector in the lattice created by the last  $n - 1$  basis vectors, and so on). This basis allows the computation of many problems exactly, but it takes exponential time to construct this basis. There are also bases

called BKZ (Block KZ) bases that for any fixed size of block  $k$ , it can be computed in polynomial time. These bases interpolate between LLL and HKZ by if  $k = 1$ , you get back LLL and if  $k = n$  you obtain HKZ.

# 33. Introduction to computational complexity

---

## 33.1 Introduction

---

**Motivation:** We want to understand *what* is a problem and *when* a problem is *easy/hard*.

**Our strategy:** An intuitive review of the basic ideas in Computational Complexity theory.

**Key concepts:**

- Problem types: optimization problems, decision problems, feasibility problems.
- Instance (of a problem)
- A problem is a collection of instances.
- Size of an instance
- Algorithm
- Running time of an algorithm; worst-case time complexity
- Complexity classes:  $P$  and  $NP$

## 33.2 Problem, instance, size

---

### 33.2.1. Problem, instance

---

**Definition 33.1: Problem**

*Is a generic question/task that needs to be answered/solved.*

*A problem is a “collection of instances” (see below).*

A *particular* realization of a problem is define next.

**Definition 33.2: Instance**

*Is a specific case of a problem. In other words, we can say “ $\text{Instance} \in \text{Problem}$ ”.*

**33.2.2. Format and examples of problems/instances**

A problem is an abstract concept. We will write problems in the following format:

- **INPUT:** Generic data/instance.
- **OUTPUT:** Question to be answered and/or task to be performed with the data.

**Examples of problems/instances:**

- Typical problems: optimization problems, decision problems, feasibility problems.
- LP and IP feasibility, TSP, IP minimization, Maximum cardinality independent set.

**33.2.3. Size of an instance**

The size of an instance is the *amount of information* required to represent the instance (in the computer).

**Definition 33.3: Binary size/length**

*Is the number of bits that are needed in order to give the problem to a computer.*

**Examples of sizes:**

- Size of an integer/rational number.
- Size of a rational matrix.
- Size of a graph (node-edge matrix representation).

**33.3 Algorithms, running time, Big-O notation****33.3.1. Basics****Definition 33.4: Algorithm**

*List of instructions to solve a problem.*

**Definition 33.5: Running time of an algorithm**

*Is the number of steps (as a function of the size) that the algorithm takes in order to solve an instance.*

**33.3.2. Worst-time complexity**

Given an algorithm to solve a problem  $P$ , the *running time of*, as a function of the size  $\sigma \in \mathbb{Z}_+$  will be defined as follows:

- The (generic) running time will be a function  $f : \mathbb{Z}_+ \rightarrow \mathbb{Z}_+$ .
- Given  $\sigma$ , the function  $f$  is defined as follows:

$$f(\sigma) = \max \{\text{running time of } z \text{ for instance } z, \text{ where } \text{size}(z) \leq \sigma\}.$$

**Remark:** This is a very conservative/pessimistic measure of running time.

**33.3.3. Big-O notation**

A function  $f : \mathbb{Z}_+ \rightarrow \mathbb{Z}_+$  belongs to the class of functions  $O(g(n))$  (that is,  $f \in O(g(n))$ ) if there exists  $c > 0, n_0 \in \mathbb{Z}_+$  such that

$$f(n) \leq cg(n), \quad \text{for all } n \geq n_0.$$

We usually say “ $f$  is  $O(g(n))$ ” or “ $f$  is order  $O(g(n))$ ”.

**33.3.4. Examples**

- Basic examples of Big-O notation:  $O(1)$ ,  $O(n^k)$ ,  $O(c^n)$ ,  $O(\log(n))$ , etc.
- An illustration of the fact that the running time depends on the size of the instance: the algorithm for the binary knapsack problem that is  $O(nb)$  is not polynomial, since the size of the instance is  $\log(b)$ .

**33.4 Basics****Definition 33.6: Polynomial time algorithm**

*An algorithm is said to be a polynomial time algorithm if its running time is  $O(n^k)$  for some  $k \geq 1$  (where  $n$  represents the size of a generic instance).*

**Remark:** *Polynomial time algorithms* are also known as *Polytime algorithms*.

### Definition 33.7: Decision problem

A decision problem is any problem whose only acceptable answers are either YES or NO (but not both at the same time).

**Some examples:** feasibility problems, decision version of optimization problems, etc.

## 33.5 Complexity classes

---

We will introduce 3 complexity classes (For see at least 495 more classes, please see [http://complexityzoo.uwaterloo.ca/Complexity\\_Zoo](http://complexityzoo.uwaterloo.ca/Complexity_Zoo)).

### 33.5.1. Polynomial time problems

---

#### Definition 33.8: The class $\mathcal{P}$

Is the set of all decision problems for which a YES or NO answer for a particular instance can be obtained in polytime.

**Remark:** For a particular problem  $P$ , there are 3 possibilities: (1)  $P \in \mathcal{P}$ , (2)  $P \notin \mathcal{P}$ , and  $P ? \mathcal{P}$  (i.e., we don't know).

**Examples:**

- Shortest path
- Max flow
- Min cut
- Matroid optimization
- Matchings
- Linear programming

### 33.5.2. Non-deterministic polynomial time problems

---

#### Definition 33.9: The class NP

Is the set of all decision problems for which a YES answer for a particular instance can be verified in polytime.

**Examples:**

- All problems in  $\mathcal{P}$
- Integer programming
- TSP
- Binary knapsack
- Maximum independent set
- Subset sum
- Partition
- SAT,  $k$ -SAT
- Clique

### 33.5.3. Complements of problems in NP

---

**Definition 33.10: The class NP**

*Is the set of all decision problems for which a NO answer for a particular instance can be verified in polytime.*

**Examples:**

- All problems in  $\mathcal{P}$
- PRIMES
- Every “complement” of an NP problem

**Remark:** Actually,  $\text{PRIMES} \in \text{NP}$  (see *Pratt's certificates*), even better, it was recently proven that  $\text{PRIMES} \in \mathcal{P}$  (by Manindra Agrawal, Neeraj Kayal, Nitin Saxena in 2004).

## 33.6 Relationship between the classes

---

### 33.6.1. A basic result

---

**Theorem 33.11**

*The following relationship holds:*

$$\mathcal{P} \subseteq \text{NP} \cap \text{coNP}.$$

### 33.6.2. An \$1,000,000 open question

---

The question “Is  $\mathcal{P} = \text{NP?}$ ” is one of the most important problems in mathematics and computer science. A correct answer is worth 1 Million dollars! Most people believe that  $\mathcal{P} \neq \text{NP}$ .

## 33.7 Comparing problems, Polynomial time reductions

---

**Motivation:** We would like to solve problem  $P_1$  by efficiently *reducing* it to another problem  $P_2$  (why? Perhaps we know how to solve  $P_2$ !!!).

### Definition 33.12: Polynomial time reductions

Let  $P_1, P_2$  be decision problems. We say that  $P_1$  is polynomially reducible to  $P_2$  (denoted  $P_1 \leq_{\mathcal{P}} P_2$ ) if there exists a function

$$f : P_1 \rightarrow P_2$$

such that

1. For all  $w \in P_1$  the answer to  $w$  is YES if and only if the answer to  $f(w)$  is YES.
2. For all  $w \in P_1$   $f(w)$  can be computed in polynomial time w.r.t to  $\text{size}(w)$ . In particular, we must have that  $\text{size}(f(w))$  is polynomially bounded by  $\text{size}(w)$ .

### Remarks:

1. In the above definition “ $f$  efficiently transforms an instance of  $P_1$  into an instance of  $P_2$ ”. In particular, if we know how to solve problem  $P_2$ , then we can solve problem  $P_1$ .
2. Therefore, the notation  $P_1 \leq_{\mathcal{P}} P_2$  makes sense: we are saying that  $P_1$  is “easier” to solve than  $P_2$ , as any algorithm for  $P_2$  would work for  $P_1$ .

## 33.8 Comparing problems, Polynomial time reductions

---

### 33.8.1. Definition

---

**Motivation:** We would like to solve problem  $P_1$  by efficiently *reducing* it to another problem  $P_2$  (why? Perhaps we know how to solve  $P_2$ !!!).

**Definition 33.13: Polynomial time reductions**

Let  $P_1, P_2$  be decision problems. We say that  $P_1$  is polynomially reducible to  $P_2$  (denoted  $P_1 \leq_{\mathcal{P}} P_2$ ) if there exists a function

$$f : P_1 \rightarrow P_2$$

such that

1. For all  $w \in P_1$  the answer to  $w$  is YES if and only if the answer to  $f(w)$  is YES.
2. For all  $w \in P_1$   $f(w)$  can be computed in polynomial time w.r.t to  $\text{size}(w)$ . In particular, we must have that  $\text{size}(f(w))$  is polynomially bounded by  $\text{size}(w)$ .

**Remarks:**

1. In the above definition “ $f$  efficiently transforms an instance of  $P_1$  into an instance of  $P_2$ ”. In particular, if we know how to solve problem  $P_2$ , then we can solve problem  $P_1$ .
2. Therefore, the notation  $P_1 \leq_{\mathcal{P}} P_2$  makes sense: we are saying that  $P_1$  is “easier” to solve than  $P_2$ , as any algorithm for  $P_2$  would work for  $P_1$ .

**33.8.2. Basic properties****Proposition 33.14: L**

*t*  $P_1, P_2$  be two problems such that  $P_1 \leq_{\mathcal{P}} P_2$  and assume that  $P_2 \in \mathcal{P}$ . Then  $P_1 \in \mathcal{P}$ .

**Proposition 33.15: L**

*t*  $P_1, P_2$  be two problems such that  $P_1 \leq_{\mathcal{P}} P_2$  and assume that  $P_2 \in NP$ . Then  $P_1 \in NP$ .

**Proposition 33.16: L**

*t*  $P_1, P_2, P_3$  be three problems assume that  $P_1 \leq_{\mathcal{P}} P_2$  and  $P_2 \leq_{\mathcal{P}} P_3$ . Then  $P_1 \leq_{\mathcal{P}} P_3$ .

## 33.9 NP-Completeness

---

### 33.9.1. The basics

---

#### Definition 33.17: NP-Completeness

*A decision problem  $P$  is said to be NP-complete if:*

1.  $P \in NP$
2.  $Q \leq_{\mathcal{P}} P$  for all  $Q \in NP$  (that is, every problem  $Q$  in  $NP$  can be polynomially reduced to  $P$ ).

#### Proposition 33.18

*If  $P$  is NP-complete and  $P \in \mathcal{P}$  then  $\mathcal{P} = NP$ .*

### 33.9.2. Do NP-complete problems exist?

---

#### Theorem 33.19: S. Cook, 1971

*SAT is NP-complete.*

## 33.10 NP-Hardness

---

#### Definition 33.20: NP-Completeness

*A problem  $P$  is said to be NP-hard if there exists a NP-complete decision problem that can be reduced to it.*

#### Remarks:

- NP-complete problems are NP-hard.
- Problems in NP-hard not need to be decision problems.
- Optimization versions of NP-complete decision problems are NP-hard.
- If  $P$  is NP-hard and  $P \in \mathcal{P}$  then  $\mathcal{P} = NP$ .

## 33.11 Exercises

---

1. Let  $P, Q$  be decision problems such that every instance of  $Q$  is an instance of  $P$  (that is  $\{\text{instances in } Q\} \subseteq \{\text{instances in } P\}$ ).
  - (a) Give an example of  $P, Q$  such that  $Q \in \mathcal{P}$  and  $P \in \text{NP - complete}$ .
  - (b) Give an example of  $P, Q$  such that  $Q, P \in \text{NP - complete}$ .

**Note:** You must prove that your problem belongs to the corresponding class unless we have proved or sketched the proof of that fact in class.

### Solution:

- (a) • Let  $P$  be the *Knapsack problem*. Then  $P$  is NP-complete (see Problem 5).  
 • Let  $Q$  be the special case of the *Knapsack problem* where all weights are 1, that is,  $a_1, \dots, a_n = 1$ . The following algorithm decides this special case:  
**ALGORITHM:**
  1. List the objects  $1, \dots, n$  in decreasing order according to  $c_1, \dots, c_n$ .
  2. Select the first  $b$  objects from that list. Call this set  $S$ .
  3. If  $\sum_{i \in S} c_i \leq k$ , then the output of the algorithm is YES. Else, the output is NO.
 Clearly, this algorithm is correct and runs in polynomial time w.r.t. to the instance. Hence,  $Q \in \mathcal{P}$ .
- (b) • Let  $P$  be SAT.  
 • Let  $Q$  be 3-SAT.

We showed in class that both  $P$  and  $Q$  are NP-complete problems.

2. A *Hamiltonian cycle* in a graph  $G = (V, E)$  is a simple cycle that contains all the vertices. A *Hamiltonian  $s - t$  path* in a graph is a simple path from  $s$  to  $t$  that contains all of the vertices. The associated decision problems are:

- **Hamiltonian Cycle Input:**  $G = (V, E)$ . **Question:** Does there exist a hamiltonian cycle in  $G$ ?
  - **Hamiltonian Path Input:**  $G = (V, E)$ ,  $s, t \in V, s \neq t$ . **Question:** Does there exist a hamiltonian  $s-t$  path in  $G$ ?
- (a) Given that *Hamiltonian Cycle* is NP-complete, prove that *Hamiltonian Path* is NP-complete.
- (b) Given that *Hamiltonian Cycle* is NP-complete, prove that the optimization version of the TSP problem is NP-hard.

## Solution:

- (a) **Step 1: Hamiltonian Path is in NP.**

It is clear that *Hamiltonian Path* is in NP, the certificate to a YES answer is the path itself. Given a Hamiltonian path from  $s$  to  $t$ . We only need to travel along it to check that in fact it visits every vertex once and that starts in  $s$  and ends in  $t$ . This takes  $O(|E|)$  time.

**Step 2: Hamiltonian Cycle  $\leq_P$  Hamiltonian Path.**

Given an instance  $G = (V, E)$  of *Hamiltonian Cycle*, we construct an instance of *Hamiltonian Path* as follows:

- Let  $v \in V$ , then we construct the graph  $G' = (V', E')$ , where:
  - $V' = V \setminus \{v\} \cup \{v_1, v_2\}$  (this takes  $O(1)$ ).
  - $E' = (E \setminus \{\{v, u\} : \{v, u\} \in E\}) \cup \{\{v_1, u\} : \{v, u\} \in E\} \cup \{\{v_2, u\} : \{v, u\} \in E\}$  (this takes  $O(|V|)$ ).
- The instance is:  $G' = (V', E')$ ,  $s = v_1$ ,  $t = v_2$ .

Notice the construction takes polynomial time w.r.t. the size of the instance.

Finally, the fact that a YES to an instance of *Hamiltonian Cycle* is equivalent to a YES to the associated *Hamiltonian Path* instance follows by noticing that there exists a Hamiltonian cycle of the form

$$vu_1u_2 \dots u_{n-1}v$$

in  $G$  if and only if there exists a Hamiltonian  $v_1-v_2$  path in  $G'$  of the form

$$v_1u_1u_2 \dots u_{n-1}v_2$$

in  $G'$ .

**Note:** we are denoting  $n = |V|$  and the notation  $u_0u_1 \dots u_k$  represents the path/cycle that uses the edges  $\{u_0, u_1\} \{u_1, u_2\} \dots \{u_{k-1}, u_k\}$  (in that order).

**Conclusion:** *Hamiltonian Path* is NP-complete.

- (b) Given an instance  $G = (V, E)$  of *Hamiltonian Cycle*, the following algorithm decides whether the answer to this instance is yes or no:

**ALGORITHM:**

1. Given  $V = \{v_1, \dots, v_n\}$ , consider the cities  $\{1, \dots, n\}$ .
2. Construct the objective function  $c$  given by:

$$c_{ij} = \begin{cases} 0, & \{v_i, v_j\} \in E \\ 1, & \text{else.} \end{cases}$$

3. Solve the TSP instance given above. Let  $\alpha$  be the cost of the optimal tour.
4. If  $\alpha = 0$ , then the output of the algorithm is YES. Else, the output is NO.

The algorithm is correct since the definition of the objective function implies that the optimal tour has cost equals to zero if and only if the tour only travels between pairs of cities associated to edges in  $E$ .

Notice that the algorithm only need to solve the TSP problem once and that every other step takes polynomial time. Therefore, this is a valid polynomial time reduction since if we were able to solve the optimization version of the TSP in polynomial time, we would also be able to decide *Hamiltonian Cycle* in polynomial time.

**Conclusion:** the optimization version of the TSP problem is NP-hard.

3. Given that the *node packing problem* is NP – complete, show that the following problems are also NP – complete:

- (a) *Node cover*: **Input:**  $G = (V, E)$ ,  $k \in \mathbb{Z}_+$ . **Question:** Does there exist a set  $S \subseteq V$  of size at most  $k$  such that every edge of  $G$  is incident to a node of  $S$ ?
- (b) *Uncapacitated facility location*. **Input:** sets  $M, N$  and integers  $k, c_{ij}, f_j$  for  $i \in M, j \in N$ . **Question:** Is there a set  $S \subseteq N$  such that  $\sum_{i \in M} \min_{j \in S} c_{ij} + \sum_{j \in S} f_j \leq k$ ?

Recall that *Node packing problem* is: **Input:**  $G = (V, E)$ ,  $k \in \mathbb{Z}_+$ . **Question:** Does there exist an independent set of size at least  $k$  in  $G$ ?

## Solution:

- (a) **Step 1: Node cover is in NP.**

It is clear that *Node cover* is in NP, the certificate is the node cover itself. Verifying that the set of nodes is a cover can be done by checking that every edge is connected to a node in the given set. This takes  $O(|E||V|)$  time.

**Step 2: Node packing  $\leq_P$  Node cover.**

Given an instance  $G = (V, E)$ ,  $l \in \mathbb{Z}_+$  of *Node Packing*, we construct an instance of *Node cover* as follows:

- We construct the graph  $G' = (V', E')$ , where:

- $V' = V$  (this takes  $O(1)$ ).
- $E' = E$  (this takes  $O(1)$ ).
- We take  $k = |V| - l$  (this takes  $O(1)$ ).
- The instance is:  $G' = (V', E'), k \in \mathbb{Z}_+$ .

Notice the construction takes polynomial time w.r.t. the size of the instance.

Finally, the fact that a YES to an instance of *Node packing* is equivalent to a YES to the associated *Node cover* instance follows by noticing that a set  $U \subseteq V$  is a node packing in  $G$  if and only if no edge in  $E$  has both end points in  $U$ , which is equivalent to say that every edge in  $E$  has at least one end point in  $V \setminus U$ . Equivalently, this is saying that the set  $V \setminus U$  is a node cover in  $G'$ .

**Conclusion:** *Node cover* is NP-complete.

(b) **Step 1: Uncapacitated facility location is in NP.**

It is clear that *Uncapacitated facility location* is in NP, because given  $S \subseteq N$ , we can verify in  $O(|M||N| + |N|)$  if

$$\sum_{i \in M} \min_{j \in S} c_{ij} + \sum_{j \in S} f_j \leq k$$

**Step 2: Node packing  $\leq_P$  Uncapacitated facility location.**

Given an instance  $G = (V, E)$ ,  $l \in \mathbb{Z}_+$  of *Node Packing*, we construct an instance of *Uncapacitated facility location* as follows:

- $M = E$ ,  $N = V$  (this takes  $O(|E|)$ ).
- The objective

$$c_{ij} = \begin{cases} |V| + 1, & \text{if } j = uv, \text{ where } u, v \neq i \\ 0, & \text{otherwise.} \end{cases}$$

(This takes  $O(|E|^2)$ .)

- $k = |V| - l$  (this takes  $O(1)$ ).
- $f_j = 1$ , for all  $j \in N$  (this takes  $O(|V|)$ ).

Notice the construction takes polynomial time w.r.t. the size of the instance.

- **YES to Node Packing  $\Rightarrow$  YES to Uncapacitated facility location:**

If  $U$  is a node packing, with  $|U| \geq l$ , then  $S = V \setminus U$  is a vertex cover, hence for all  $i \in M$ :

$$\min\{c_{ij} : j \in S\} = 0.$$

And

$$\sum_{j \in S} f_j = |S| = |V| - |U| \leq |V| - l \leq k.$$

This implies

$$\sum_{i \in M} \min_{j \in S} c_{ij} + \sum_{j \in S} f_j \leq k.$$

Thus, the set  $S \subseteq N$  gives a YES answer to *Uncapacitated facility location*.

- **YES to Uncapacitated facility location  $\Rightarrow$  YES to Node Packing:**

There exists  $S \subseteq N$  such that

$$\sum_{i \in M} \min_{j \in S} c_{ij} + \sum_{j \in S} f_j \leq k.$$

By definition of the reduction from node packing, we have

- (i) For all  $i \in M$ ,  $\min\{c_{ij} : j \in S\} \leq |V|$ , which implies  $\min\{c_{ij} : j \in S\} = 0$ .
- (ii) Let  $U = V \setminus S$ . By (i),  $\sum_{j \in S} f_j = |S| = |V| - |U|$ .
- (iii) By (i), if  $u, v \in U$ , then we must have  $\{u, v\} \notin E$ .
- (iv) By (ii), we have  $|U| \geq l$ .

This implies that the set  $U \subseteq V$  is a node packing with  $|U| \geq l$ , which gives a YES answer to *Node Packing*.

**Conclusion:** *Uncapacitated facility location* is NP-complete.



# 34. Reformulation and Decomposition Techniques

## 34.1 Lagrangean relaxation

Consider the MIP problem

$$z_{IP} = \max\{c^T x : A^1 x \leq b^1, A^2 x \leq b^2, x \in \mathbb{Z}^n\}, \quad (1)$$

where  $A^i \in \mathbb{R}^{m_i \times n}$  and  $b^i \in \mathbb{R}^{m_i}$ , for  $i = 1, 2$ . Denote  $X = \{x \in \mathbb{R}^n : A^2 x \leq b^2, x \in \mathbb{Z}^n\}$ .

### Definition 34.1: Lagrangean relaxation

Given  $u \in \mathbb{R}_+^{m_1}$ , the Lagrangean relaxation is the following MIP

$$v(u) = \max\{c^T x + u^T (b^1 - A^1 x) : x \in X\}.$$

### Remark 34.2

Observe that we have  $v(u) \geq z_{IP}$  for all  $u \in \mathbb{R}_+^{m_1}$ .

### Definition 34.3: Lagrangean dual

The Lagrangean dual is the following optimization problem

$$v_{LD} = \min\{v(u) : u \in \mathbb{R}_+^{m_1}\}.$$

### Remark 34.4

$v_{LD}$  is the ‘best possible value’ for the upper bound  $v(u)$ .

**Theorem 34.5**

Denote  $z_{LP}$  the optimal value of the continuous relaxation of the MIP problem (1). Then

1. Consider the following linear program

$$w_{LD} = \max\{c^T x : A^1 x \leq b^1, x \in \text{conv}(X)\}.$$

Then  $v_{LD} = w_{LD}$ .

2. We have that  $z_{IP} \leq v_{LD} \leq z_{LP}$ .

Describe subgradient algorithm of optimization the Lagrangian Dual.

Add example and connect this to code. Show that using the lagrangian dual is faster than solving the original problem. Example might be chosen to have an efficiently solvable lagrangian dual, such as TU constraints.

## 34.2 Column generation

---

### 34.2.1. The master problem and the pricing subproblem

---

Let  $A \in \mathbb{R}^{m \times n}$ ,  $b \in \mathbb{R}^m$  and  $c \in \mathbb{R}^n$ . Consider the following linear program

$$z_{LP} = \min \left\{ c^T x : \sum_{i=1}^n A^i x_i = b, x \geq 0 \right\} \quad (LP).$$

**Definition 34.6: Master problem**

Given  $I \subseteq \{1, \dots, n\}$ , consider the following restriction of the above LP:

$$z_{LP}(I) = \min \left\{ c^T x : \sum_{i \in I} A^i x_i = b x_i \geq 0, \text{ for all } i \in I \right\} \quad (MLP).$$

Let  $\bar{x}_i, i \in I$  be an optimal solution to (MLP) and let  $\bar{y} \in \mathbb{R}^m$  be an optimal solution to its dual. Then  $\bar{x}_i, i \in I$ ,  $\bar{x}_i := 0, i \notin I$  is an optimal solution to (LP) if and only if

$$c_i - \bar{y}^T A^i \geq 0, \text{ for all } i = 1, \dots, n.$$

(that is, if and only if  $\bar{y}$  is also a feasible solution to the dual of (LP).)

**Definition 34.7: The pricing subproblem**

Let  $\bar{y} \in \mathbb{R}^m$  be an optimal solution to the dual of (MLP). Consider the following optimization problem

$$w_{SP} = \min\{c_i - \bar{y}^T A^i : i = 1, \dots, n\}.$$

**Theorem 34.8**

Let  $\bar{x}_i, i \in I$  be an optimal solution to (MLP) and let  $\bar{y} \in \mathbb{R}^m$  be an optimal solution to its dual.

1. If  $w_{SP} \geq 0$ , then  $\bar{x}_i, i \in I, \bar{x}_i := 0, i \notin I$  is an optimal solution to (LP). Thus, we can solve (LP) by solving the restricted problem (MLP).
2. If  $w_{SP} < 0$ , in order to solve (LP) by solving the restricted problem (MLP), we need to add more columns to (MLP).

Connect this to example in first part of book (or rewrite this). Create or find code example to connect this to and show column generation version is faster.

**34.2.2. Dantzig-Wolf decomposition**

Consider the following MIP

$$z_{IP} = \max \left\{ \sum_{k=1}^K (c^k)^T x^k : A^1 + \dots + A^K = b, x^k \in X^k, \text{ for all } k = 1, \dots, K \right\}, \quad (2)$$

where  $X^k = \{x \in \mathbb{Z}^{n_k} : D^k x^k \leq d^k\}$ , for  $k = 1, \dots, K$ .

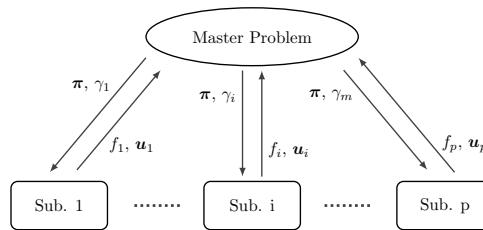
Assume that the sets  $X^k, k = 1, \dots, K$  are bounded. Consequently, for each  $k = 1, \dots, K$ , we can write

$$X^k = \{x^{k,t} : t = 1, \dots, T_k\}.$$

**Definition 34.9: Dantzig-Wolf reformulation**

The following MIP is the Dantzig-Wolf reformulation of (2)

$$\begin{aligned} z_{DW} = \max & \sum_{k=1}^K \sum_{t=1}^{T_k} (c^k)^T x^{k,t} \\ & \sum_{k=1}^K \sum_{t=1}^{T_k} A^k x^{k,t} \lambda_{kt} = b \\ & \sum_{t=1}^{T_k} \lambda_{kt} = 1, \quad \text{for all } k = 1, \dots, K \\ & \lambda_{kt} \in \{0, 1\} \quad \text{for all } k = 1, \dots, K, t = 1, \dots, T_k. \end{aligned}$$



© tikz/dantzig-wolfe-decomposition/dantzig-wolfe-decomposition.pdf<sup>1</sup>

**Figure 34.1: tikz/dantzig-wolfe-decomposition/dantzig-wolfe-decomposition.pdf**

### Remark 34.10: C

early,  $z_{IP} = z_{DW}$ .

The continuous relaxation of the above MIP can be solved by using the column generation approach.

Elaborate and connect this to example in first part of book (or rewrite this). Create or find code example to connect this to and show decomposition version is faster.

## 34.3 Extended formulations

Let  $P = \{x \in \mathbb{R}^n : Ax \leq b\}$  be a polyhedron.

### Definition 34.11: Extended formulation

An extended formulation for  $P$  is a polyhedron

$$Q = \{(x, y) \in \mathbb{R}^n \times \mathbb{R}^m : Ex + Fy = g, y \geq 0\},$$

with the property that  $x \in P$  if and only if exists  $y \in \mathbb{R}^m$  such that  $(x, y) \in Q$ .

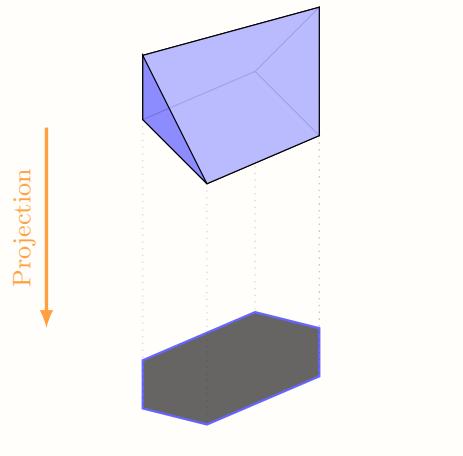
### Definition 34.12: Extension complexity

The of  $P$  is the minimum  $m$  such that there exists an extended formulation  $Q \subseteq \mathbb{R}^n \times \mathbb{R}^m$  of  $P$ .

### Theorem 34.13

Let  $P$  be the TSP polytope (that is,  $P$  is the convex hull of the 0-1 vectors that represent valid tours). Then the extension complexity of  $P$  is at least  $2^{\Omega(n^{1/2})}$ .

<sup>1</sup>tikz/dantzig-wolfe-decomposition/dantzig-wolfe-decomposition.pdf, from tikz/dantzig-wolfe-decomposition/dantzig-wolfe-decomposition.pdf, tikz/dantzig-wolfe-decomposition/dantzig-wolfe-decomposition.pdf, tikz/dantzig-wolfe-decomposition/dantzig-wolfe-decomposition.pdf



© tikz/extended-formulation<sup>2</sup>

**Figure 34.2: tikz/extended-formulation**

#### Remark 34.14

This result implies that there is no **ideal** formulation for the TSP problem such that the formulation is of polynomial size. Conversely, if we have a formulation for the TSP problem that is of polynomial size, the this formulation cannot be ideal.

Add bib references for these figures (Created by Robert Hildebrand).

SI: 4OR Surveys Published: 22 September 2015 Deriving compact extended formulations via LP-based separation techniques Giuseppe Lancia & Paolo Serafini Survey

## 34.4 Benders decomposition

In , we try projecting out some variales. This creates new inequalities that we must generate on the fly in a cutting plane scheme.

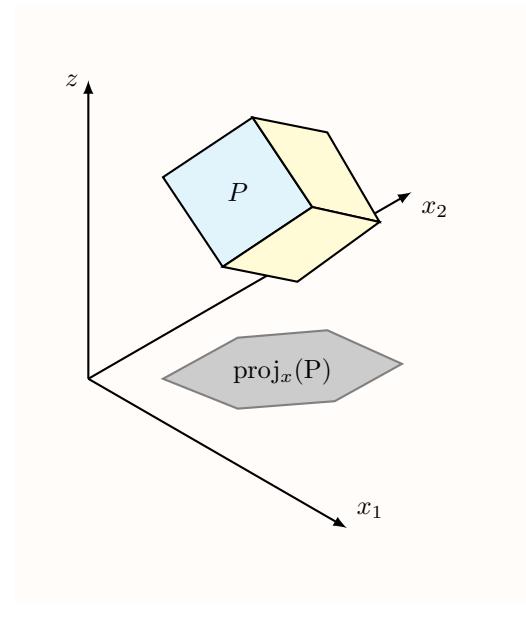
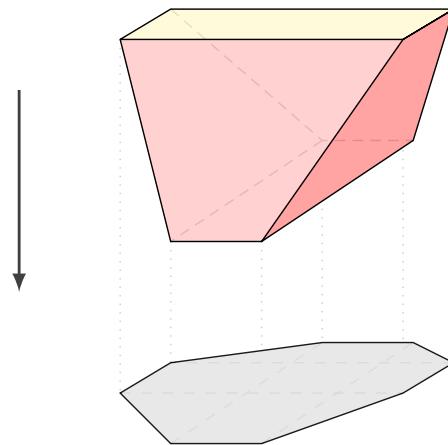
From Wikipedia, the free encyclopedia:

**Benders' decomposition** (alternatively, Benders's decomposition; named after Jacques F. Benders) is a technique in mathematical programming that allows the solution of very large linear programming

<sup>2</sup>tikz/extended-formulation, from tikz/extended-formulation. tikz/extended-formulation, tikz/extended-formulation.

<sup>3</sup>tikz/extended-formulation2, from tikz/extended-formulation2. tikz/extended-formulation2, tikz/extended-formulation2.

<sup>4</sup>tikz/extended-formulation3, from tikz/extended-formulation3. tikz/extended-formulation3, tikz/extended-formulation3.

© tikz/extended-formulation2<sup>3</sup>**Figure 34.3: tikz/extended-formulation2**© tikz/extended-formulation3<sup>4</sup>**Figure 34.4: tikz/extended-formulation3**

problems that have a special block structure. This structure often occurs in applications such as stochastic programming.

As it progresses towards a solution, Benders' decomposition adds new constraints, so the approach is called “row generation”. In contrast, Dantzig-Wolfe decomposition uses “column generation”.

See more information in the next section.

## 35. Stochastic Programming

---

Argonne DSP solver for two stage programs



## **Part V**

### **Appendix - Linear Algebra Background**



# A. Linear Transformations

---

**Lab Objective:** *Linear transformations are the most basic and essential operators in vector space theory. In this lab we visually explore how linear transformations alter points in the Cartesian plane. We also empirically explore the computational cost of applying linear transformations via matrix multiplication.*

## Linear Transformations

---

A *linear transformation* is a mapping between vector spaces that preserves addition and scalar multiplication. More precisely, let  $V$  and  $W$  be vector spaces over a common field  $\mathbb{F}$ . A map  $L : V \rightarrow W$  is a linear transformation from  $V$  into  $W$  if

$$L(a\mathbf{x}_1 + b\mathbf{x}_2) = aL\mathbf{x}_1 + bL\mathbf{x}_2$$

for all vectors  $\mathbf{x}_1, \mathbf{x}_2 \in V$  and scalars  $a, b \in \mathbb{F}$ .

Every linear transformation  $L$  from an  $m$ -dimensional vector space into an  $n$ -dimensional vector space can be represented by an  $m \times n$  matrix  $A$ , called the *matrix representation* of  $L$ . To apply  $L$  to a vector  $\mathbf{x}$ , left multiply by its matrix representation. This results in a new vector  $\mathbf{x}'$ , where each component is some linear combination of the elements of  $\mathbf{x}$ . For linear transformations from  $\mathbb{R}^2$  to  $\mathbb{R}^2$ , this process has the form

$$A\mathbf{x} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} ax+by \\ cx+dy \end{bmatrix} = \begin{bmatrix} x' \\ y' \end{bmatrix} = \mathbf{x}'.$$

Linear transformations can be interpreted geometrically. To demonstrate this, consider the array of points  $H$  that collectively form a picture of a horse, stored in the file `horse.npy`. The coordinate pairs  $\mathbf{x}_i$  are organized by column, so the array has two rows: one for  $x$ -coordinates, and one for  $y$ -coordinates. Matrix multiplication on the left transforms each coordinate pair, resulting in another matrix  $H'$  whose columns are the transformed coordinate pairs:

$$\begin{aligned} AH &= A \begin{bmatrix} x_1 & x_2 & x_3 & \dots \\ y_1 & y_2 & y_3 & \dots \end{bmatrix} = A \begin{bmatrix} \mathbf{x}_1 & | & \mathbf{x}_2 & | & \mathbf{x}_3 & | & \dots \end{bmatrix} = \begin{bmatrix} A\mathbf{x}_1 & | & A\mathbf{x}_2 & | & A\mathbf{x}_3 & | & \dots \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{x}'_1 & | & \mathbf{x}'_2 & | & \mathbf{x}'_3 & | & \dots \end{bmatrix} = \begin{bmatrix} x'_1 & x'_2 & x'_3 & \dots \\ y'_1 & y'_2 & y'_3 & \dots \end{bmatrix} = H'. \end{aligned}$$

To begin, use `np.load()` to extract the array from the `npy` file, then plot the unaltered points as individual pixels. See Figure A.1 for the result.

```

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

# Load the array from the .npy file.
>>> data = np.load("horse.npy")

# Plot the x row against the y row with black pixels.
>>> plt.plot(data[0], data[1], 'k,')

# Set the window limits to [-1, 1] by [-1, 1] and make the window square.
>>> plt.axis([-1,1,-1,1])
>>> plt.gca().set_aspect("equal")
>>> plt.show()

```

## Types of Linear Transformations

---

Linear transformations from  $\mathbb{R}^2$  into  $\mathbb{R}^2$  can be classified in a few ways.

- **Stretch:** Stretches or compresses the vector along each axis. The matrix representation is diagonal:

$$\begin{bmatrix} a & 0 \\ 0 & b \end{bmatrix}.$$

If  $a = b$ , the transformation is called a *dilation*. The stretch in Figure A.1 uses  $a = \frac{1}{2}$  and  $b = \frac{6}{5}$  to compress the  $x$ -axis and stretch the  $y$ -axis.

- **Shear:** Slants the vector by a scalar factor horizontally or vertically (or both simultaneously). The matrix representation is

$$\begin{bmatrix} 1 & a \\ b & 1 \end{bmatrix}.$$

Pure horizontal shears ( $b = 0$ ) skew the  $x$ -coordinate of the vector while pure vertical shears ( $a = 0$ ) skew the  $y$ -coordinate. Figure A.1 has a horizontal shear with  $a = \frac{1}{2}, b = 0$ .

- **Reflection:** Reflects the vector about a line that passes through the origin. The reflection about the line spanned by the vector  $[a,b]^T$  has the matrix representation

$$\frac{1}{a^2 + b^2} \begin{bmatrix} a^2 - b^2 & 2ab \\ 2ab & b^2 - a^2 \end{bmatrix}.$$

The reflection in Figure A.1 reflects the image about the  $y$ -axis ( $a = 0, b = 1$ ).

- **Rotation:** Rotates the vector around the origin. A counterclockwise rotation of  $\theta$  radians has the following matrix representation:

$$\begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix}$$

A negative value of  $\theta$  performs a clockwise rotation. Choosing  $\theta = \frac{\pi}{2}$  produces the rotation in Figure A.1.

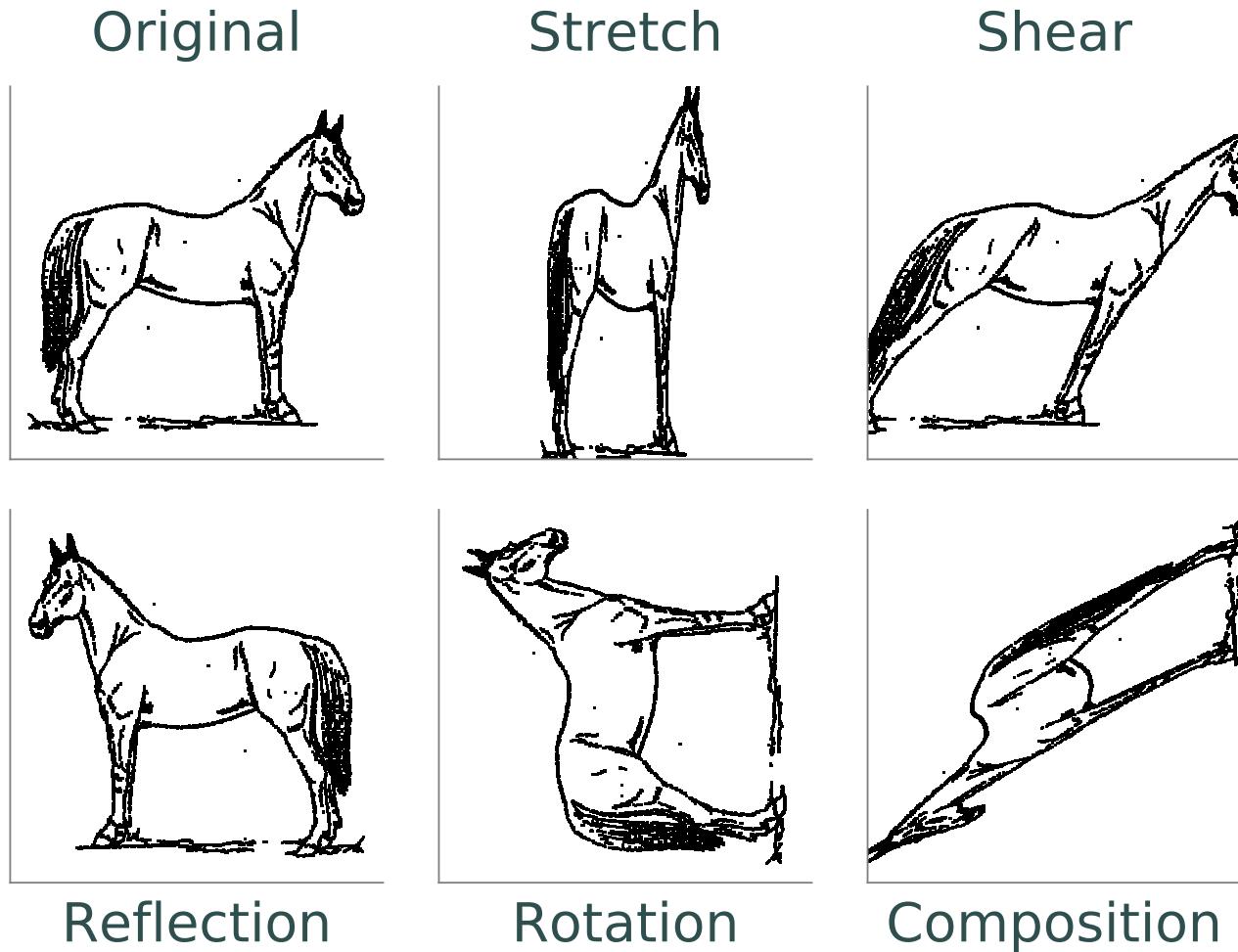


Figure A.1: The points stored in `horse.npy` under various linear transformations.

### Problem A.1: Implement linear transformations.

Write a function for each type of linear transformation. Each function should accept an array to transform and the scalars that define the transformation ( $a$  and  $b$  for stretch, shear, and reflection, and  $\theta$  for rotation). Construct the matrix representation, left multiply it with the input array, and return the transformed array.

To test these functions, write a function to plot the original points in `horse.npy` together with the transformed points in subplots for a side-by-side comparison. Compare your results to Figure A.1.

## Compositions of Linear Transformations

---

Let  $V$ ,  $W$ , and  $Z$  be finite-dimensional vector spaces. If  $L : V \rightarrow W$  and  $K : W \rightarrow Z$  are linear transformations with matrix representations  $A$  and  $B$ , respectively, then the *composition* function  $KL : V \rightarrow Z$  is also a linear transformation, and its matrix representation is the matrix product  $BA$ .

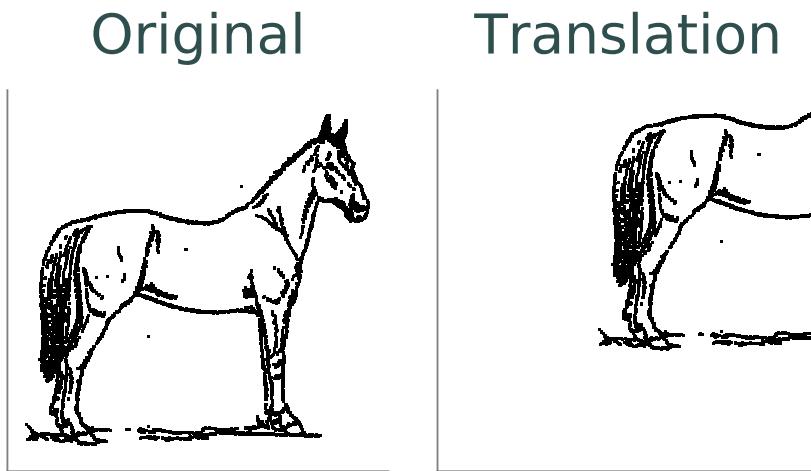
For example, if  $S$  is a matrix representing a shear and  $R$  is a matrix representing a rotation, then  $RS$  represents a shear followed by a rotation. In fact, any linear transformation  $L : \mathbb{R}^2 \rightarrow \mathbb{R}^2$  is a composition of the four transformations discussed above. Figure A.1 displays the composition of all four previous transformations, applied in order (stretch, shear, reflection, then rotation).

## Affine Transformations

---

All linear transformations map the origin to itself. An *affine transformation* is a mapping between vector spaces that preserves the relationships between points and lines, but that may not preserve the origin. Every affine transformation  $T$  can be represented by a matrix  $A$  and a vector  $b$ . To apply  $T$  to a vector  $x$ , calculate  $Ax + b$ . If  $b = 0$  then the transformation is linear, and if  $A = I$  but  $b \neq 0$  then it is called a *translation*.

For example, if  $T$  is the translation with  $\mathbf{b} = \begin{bmatrix} \frac{3}{4}, \frac{1}{2} \end{bmatrix}^\top$ , then applying  $T$  to an image will shift it right by  $\frac{3}{4}$  and up by  $\frac{1}{2}$ . This translation is illustrated below.

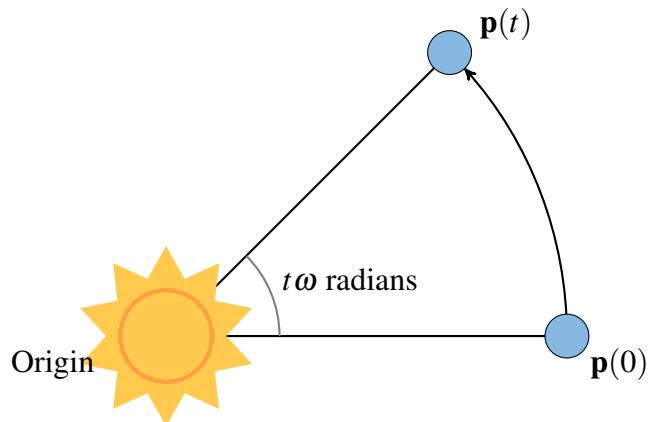


Affine transformations include all compositions of stretches, shears, rotations, reflections, and translations. For example, if  $S$  represents a shear and  $R$  a rotation, and if  $b$  is a vector, then  $RS\mathbf{x} + b$  shears, then rotates, then translates  $\mathbf{x}$ .

## Modeling Motion with Affine Transformations

---

Affine transformations can be used to model particle motion, such as a planet rotating around the sun. Let the sun be the origin, the planet's location at time  $t$  be given by the vector  $\mathbf{p}(t)$ , and suppose the planet has angular velocity  $\omega$  (a measure of how fast the planet goes around the sun). To find the planet's position at time  $t$  given the planet's initial position  $\mathbf{p}(0)$ , rotate the vector  $\mathbf{p}(0)$  around the origin by  $t\omega$  radians. Thus if  $R(\theta)$  is the matrix representation of the linear transformation that rotates a vector around the origin by  $\theta$  radians, then  $\mathbf{p}(t) = R(t\omega)\mathbf{p}(0)$ .



Composing the rotation with a translation shifts the center of rotation away from the origin, yielding more complicated motion.

**Problem A.2: Moon orbiting the earth orbiting the sun.**

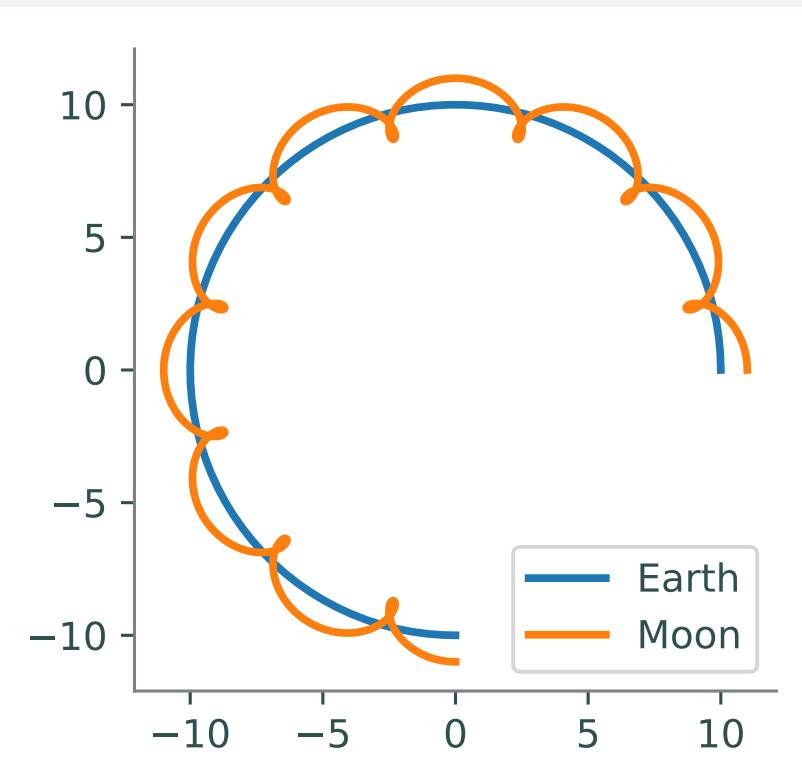
The moon orbits the earth while the earth orbits the sun. Assuming circular orbits, we can compute the trajectories of both the earth and the moon using only linear and affine transformations.

Assume an orientation where both the earth and moon travel counterclockwise, with the sun at the origin. Let  $\mathbf{p}_e(t)$  and  $\mathbf{p}_m(t)$  be the positions of the earth and the moon at time  $t$ , respectively, and let  $\omega_e$  and  $\omega_m$  be each celestial body's angular velocity. For a particular time  $t$ , we calculate  $\mathbf{p}_e(t)$  and  $\mathbf{p}_m(t)$  with the following steps.

1. Compute  $\mathbf{p}_e(t)$  by rotating the initial vector  $\mathbf{p}_e(0)$  counterclockwise about the origin by  $t\omega_e$  radians.
2. Calculate the position of the moon relative to the earth at time  $t$  by rotating the vector  $\mathbf{p}_m(0) - \mathbf{p}_e(0)$  counterclockwise about the origin by  $t\omega_m$  radians.
3. To compute  $\mathbf{p}_m(t)$ , translate the vector resulting from the previous step by  $\mathbf{p}_e(t)$ .

Write a function that accepts a final time  $T$ , initial positions  $x_e$  and  $x_m$ , and the angular momenta  $\omega_e$  and  $\omega_m$ . Assuming initial positions  $\mathbf{p}_e(0) = (x_e, 0)$  and  $\mathbf{p}_m(0) = (x_m, 0)$ , plot  $\mathbf{p}_e(t)$  and  $\mathbf{p}_m(t)$  over the time interval  $t \in [0, T]$ .

Setting  $T = \frac{3\pi}{2}$ ,  $x_e = 10$ ,  $x_m = 11$ ,  $\omega_e = 1$ , and  $\omega_m = 13$ , your plot should resemble the following figure (fix the aspect ratio with `ax.set_aspect("equal")`). Note that a more celestially accurate figure would use  $x_e = 400$ ,  $x_m = 401$  (the interested reader should see <http://www.math.nus.edu.sg/aslaksen/teaching/convex.html>).



# Timing Matrix Operations

Linear transformations are easy to perform via matrix multiplication. However, performing matrix multiplication with very large matrices can strain a machine's time and memory constraints. For the remainder of this lab we take an empirical approach in exploring how much time and memory different matrix operations require.

## Timing Code

Recall that the `time` module's `time()` function measures the number of seconds since the Epoch. To measure how long it takes for code to run, record the time just before and just after the code in question, then subtract the first measurement from the second to get the number of seconds that have passed. Additionally, in IPython, the quick command `%timeit` uses the `timeit` module to quickly time a single line of code.

```
In [1]: import time

In [2]: def for_loop():
....:     """Go through ten million iterations of nothing."""
....:     for _ in range(int(1e7)):
....:         pass

In [3]: def time_for_loop():
....:     """Time for_loop() with time.time()."""
....:     start = time.time()           # Clock the starting time.
....:     for_loop()
....:     return time.time() - start   # Return the elapsed time.

In [4]: time_for_loop()
0.24458789825439453

In [5]: %timeit for_loop()
248 ms +- 5.35 ms per loop (mean +- std. dev. of 7 runs, 1 loop each)
```

## Timing an Algorithm

---

Most algorithms have at least one input that dictates the size of the problem to be solved. For example, the following functions take in a single integer  $n$  and produce a random vector of length  $n$  as a list or a random  $n \times n$  matrix as a list of lists.

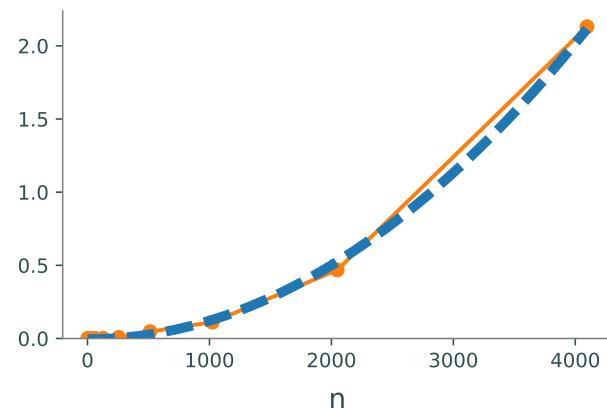
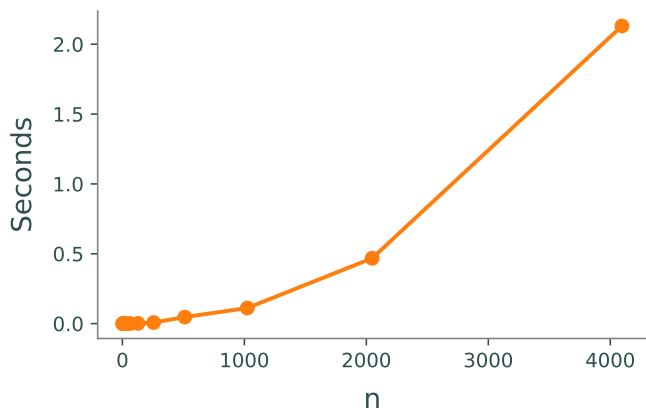
```
from random import random
def random_vector(n):      # Equivalent to np.random.random(n).tolist()
    """Generate a random vector of length n as a list."""
    return [random() for i in range(n)]

def random_matrix(n):      # Equivalent to np.random.random((n,n)).tolist()
    """Generate a random nxn matrix as a list of lists."""
    return [[random() for j in range(n)] for i in range(n)]
```

Executing `random_vector(n)` calls `random()`  $n$  times, so doubling  $n$  should about double the amount of time `random_vector(n)` takes to execute. By contrast, executing `random_matrix(n)` calls `random()`  $n^2$  times ( $n$  times per row with  $n$  rows). Therefore doubling  $n$  will likely more than double the amount of time `random_matrix(n)` takes to execute, especially if  $n$  is large.

To visualize this phenomenon, we time `random_matrix()` for  $n = 2^1, 2^2, \dots, 2^{12}$  and plot  $n$  against the execution time. The result is displayed below on the left.

```
>>> domain = 2**np.arange(1,13)
>>> times = []
>>> for n in domain:
...     start = time.time()
...     random_matrix(n)
...     times.append(time.time() - start)
...
>>> plt.plot(domain, times, 'g.-', linewidth=2, markersize=15)
>>> plt.xlabel("n", fontsize=14)
>>> plt.ylabel("Seconds", fontsize=14)
>>> plt.show()
```



The figure on the left shows that the execution time for `random_matrix(n)` increases quadratically in  $n$ . In fact, the blue dotted line in the figure on the right is the parabola  $y = an^2$ , which fits nicely over the timed observations. Here  $a$  is a small constant, but it is much less significant than the exponent on the  $n$ . To represent this algorithm's growth, we ignore  $a$  altogether and write  $\text{random\_matrix}(n) \sim n^2$ .

### NOTE

An algorithm like `random_matrix(n)` whose execution time increases quadratically with  $n$  is called  $O(n^2)$ , denoted by  $\text{random\_matrix}(n) \in O(n^2)$ . Big-oh notation is common for indicating both the *temporal complexity* of an algorithm (how the execution time grows with  $n$ ) and the *spatial complexity* (how the memory usage grows with  $n$ ).

**Problem A.3: Time Matrix-Vector and Matrix-Matrix Multiplication**

Let  $A$  be an  $m \times n$  matrix with entries  $a_{ij}$ ,  $\mathbf{x}$  be an  $n \times 1$  vector with entries  $x_k$ , and  $B$  be an  $n \times p$  matrix with entries  $b_{ij}$ . The matrix-vector product  $A\mathbf{x} = \mathbf{y}$  is a new  $m \times 1$  vector and the matrix-matrix product  $AB = C$  is a new  $m \times p$  matrix. The entries  $y_i$  of  $\mathbf{y}$  and  $c_{ij}$  of  $C$  are determined by the following formulas:

$$y_i = \sum_{k=1}^n a_{ik}x_k \quad c_{ij} = \sum_{k=1}^n a_{ik}b_{kj}$$

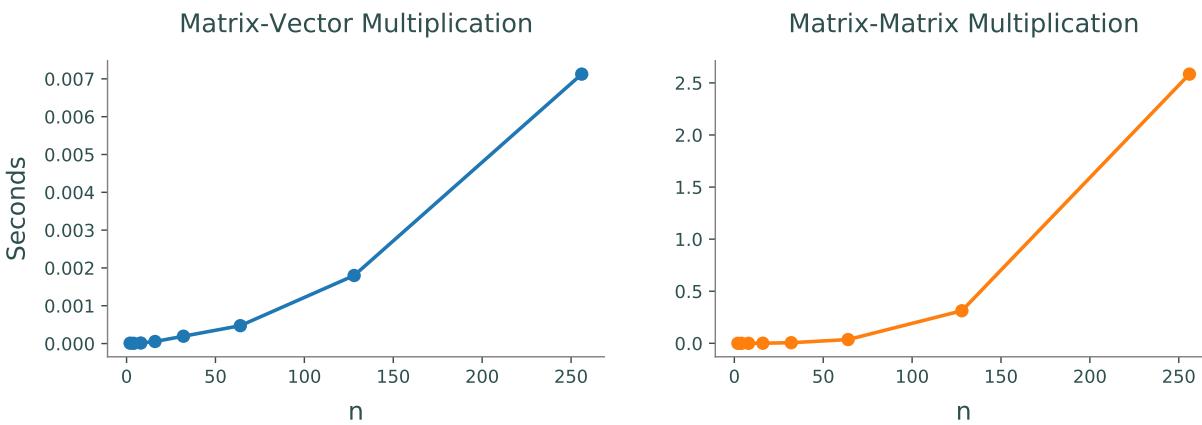
These formulas are implemented below **without** using NumPy arrays or operations.

```
def matrix_vector_product(A, x):      # Equivalent to np.dot(A,x).tolist()
    """Compute the matrix-vector product Ax as a list."""
    m, n = len(A), len(x)
    return [sum([A[i][k] * x[k] for k in range(n)]) for i in range(m)]

def matrix_matrix_product(A, B):        # Equivalent to np.dot(A,B).tolist()
    """Compute the matrix-matrix product AB as a list of lists."""
    m, n, p = len(A), len(B), len(B[0])
    return [[sum([A[i][k] * B[k][j] for k in range(n)])
            for j in range(p)] for i in range(m)]
```

Time each of these functions with increasingly large inputs. Generate the inputs  $A$ ,  $\mathbf{x}$ , and  $B$  with `random_matrix()` and `random_vector()` (so each input will be  $n \times n$  or  $n \times 1$ ). Only time the multiplication functions, not the generating functions.

Report your findings in a single figure with two subplots: one with matrix-vector times, and one with matrix-matrix times. Choose a domain for  $n$  so that your figure accurately describes the growth, but avoid values of  $n$  that lead to execution times of more than 1 minute. Your figure should resemble the following plots.



## Logarithmic Plots

---

Though the two plots from Problem A look similar, the scales on the  $y$ -axes show that the actual execution times differ greatly. To be compared correctly, the results need to be viewed differently.

A *logarithmic plot* uses a logarithmic scale—with values that increase exponentially, such as  $10^1$ ,  $10^2$ ,  $10^3$ , ...—on one or both of its axes. The three kinds of log plots are listed below.

- **log-lin:** the  $x$ -axis uses a logarithmic scale but the  $y$ -axis uses a linear scale.  
Use `plt.semilogx()` instead of `plt.plot()`.
- **lin-log:** the  $x$ -axis is uses a linear scale but the  $y$ -axis uses a log scale.  
Use `plt.semilogy()` instead of `plt.plot()`.
- **log-log:** both the  $x$  and  $y$ -axis use a logarithmic scale.  
Use `plt.loglog()` instead of `plt.plot()`.

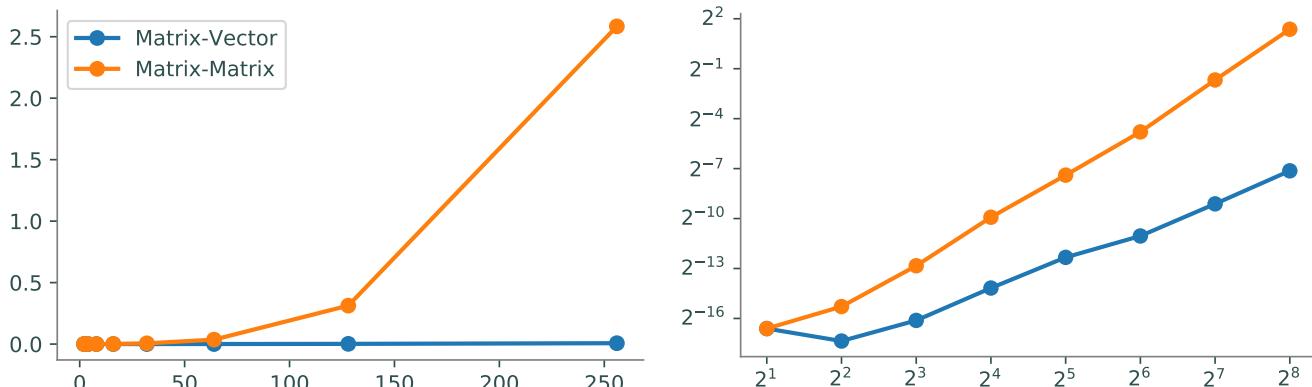
Since the domain  $n = 2^1, 2^2, \dots$  is a logarithmic scale and the execution times increase quadratically, we visualize the results of the previous problem with a log-log plot. The default base for the logarithmic scales on logarithmic plots in Matplotlib is 10. To change the base to 2 on each axis, specify the keyword arguments `basex=2` and `basey=2`.

Suppose the domain of  $n$  values are stored in `domain` and the corresponding execution times for `matrix_vector_product()` and `matrix_matrix_product()` are stored in `vector_times` and `matrix_times`, respectively. Then the following code produces Figure A.5.

```
>>> ax1 = plt.subplot(121) # Plot both curves on a regular lin-lin plot.
>>> ax1.plot(domain, vector_times, 'b.-', lw=2, ms=15, label="Matrix-Vector")
>>> ax1.plot(domain, matrix_times, 'g.-', lw=2, ms=15, label="Matrix-Matrix")
>>> ax1.legend(loc="upper left")

>>> ax2 = plt.subplot(122) # Plot both curves on a base 2 log-log plot.
>>> ax2.loglog(domain, vector_times, 'b.-', basex=2, basey=2, lw=2)
>>> ax2.loglog(domain, matrix_times, 'g.-', basex=2, basey=2, lw=2)

>>> plt.show()
```

**Figure A.5**

In the log-log plot, the slope of the `matrix_matrix_product()` line is about 3 and the slope of the `matrix_vector_product()` line is about 2. This reflects the fact that matrix-matrix multiplication (which uses 3 loops) is  $O(n^3)$ , while matrix-vector multiplication (which only has 2 loops) is only  $O(n^2)$ .

### Exercise A.4: N

*mPy is built specifically for fast numerical computations. Repeat the experiment of Problem A, timing the following operations:*

- *matrix-vector multiplication with `matrix_vector_product()`.*
- *matrix-matrix multiplication with `matrix_matrix_product()`.*
- *matrix-vector multiplication with `np.dot()` or `@`.*
- *matrix-matrix multiplication with `np.dot()` or `@`.*

*Create a single figure with two subplots: one with all four sets of execution times on a regular linear scale, and one with all four sets of execution times on a log-log scale. Compare your results to Figure A.5.*

### NOTE

Problem A shows that **matrix operations are significantly faster in NumPy than in plain Python**. Matrix-matrix multiplication grows cubically regardless of the implementation; however, with lists the times grows at a rate of  $an^3$  while with NumPy the times grow at a rate of  $bn^3$ , where  $a$  is much larger than  $b$ . NumPy is more efficient for several reasons:

1. Iterating through loops is very expensive. Many of NumPy's operations are implemented in C, which are much faster than Python loops.
2. Arrays are designed specifically for matrix operations, while Python lists are general purpose.

3. NumPy carefully takes advantage of computer hardware, efficiently using different levels of computer memory.

However, in Problem A, the execution times for matrix multiplication with NumPy seem to increase somewhat inconsistently. This is because the fastest layer of computer memory can only handle so much information before the computer has to begin using a larger, slower layer of memory.



# B. Linear Systems

---

**Lab Objective:** *The fundamental problem of linear algebra is solving the linear system  $\mathbf{Ax} = \mathbf{b}$ , given that a solution exists. There are many approaches to solving this problem, each with different pros and cons. In this lab we implement the LU decomposition and use it to solve square linear systems. We also introduce SciPy, together with its libraries for linear algebra and working with sparse matrices.*

## Gaussian Elimination

---

The standard approach for solving the linear system  $\mathbf{Ax} = \mathbf{b}$  on paper is reducing the augmented matrix  $[A | b]$  to row-echelon form (REF) via *Gaussian elimination*, then using back substitution. The matrix is in REF when the leading non-zero term in each row is the diagonal term, so the matrix is upper triangular.

At each step of Gaussian elimination, there are three possible operations: swapping two rows, multiplying one row by a scalar value, or adding a scalar multiple of one row to another. Many systems, like the one displayed below, can be reduced to REF using only the third type of operation. First, use multiples of the first row to get zeros below the diagonal in the first column, then use a multiple of the second row to get zeros below the diagonal in the second column.

$$\left[ \begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 1 & 4 & 2 & 3 \\ 4 & 7 & 8 & 9 \end{array} \right] \rightarrow \left[ \begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ \textcolor{red}{0} & 3 & 1 & 2 \\ 4 & 7 & 8 & 9 \end{array} \right] \rightarrow \left[ \begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 0 & 3 & 1 & 2 \\ \textcolor{red}{0} & 3 & 4 & 5 \end{array} \right] \rightarrow \left[ \begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 0 & 3 & 1 & 2 \\ 0 & \textcolor{red}{0} & 3 & 3 \end{array} \right]$$

Each of these operations is equivalent to left-multiplying by a *type III elementary matrix*, the identity with a single non-zero non-diagonal term. If row operation  $k$  corresponds to matrix  $E_k$ , the following equation is  $E_3 E_2 E_1 A = U$ .

$$\left[ \begin{array}{ccc} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 1 \end{array} \right] \left[ \begin{array}{ccc} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -4 & 0 & 1 \end{array} \right] \left[ \begin{array}{ccc} 1 & 0 & 0 \\ -1 & 1 & 0 \\ 0 & 0 & 1 \end{array} \right] \left[ \begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 1 & 4 & 2 & 3 \\ 4 & 7 & 8 & 9 \end{array} \right] = \left[ \begin{array}{ccc|c} 1 & 1 & 1 & 1 \\ 0 & 3 & 1 & 2 \\ 0 & 0 & 3 & 3 \end{array} \right]$$

However, matrix multiplication is an inefficient way to implement row reduction. Instead, modify the matrix in place (without making a copy), changing only those entries that are affected by each row operation.

```
>>> import numpy as np  
  
>>> A = np.array([[1, 1, 1, 1],  
...                 [1, 4, 2, 3],  
...                 [4, 7, 8, 9],  
...                 [0, 0, 0, 1]])  
  
>>> E1 = np.array([[1, 0, 0],  
...                 [0, 1, 0],  
...                 [0, -1, 1]])  
  
>>> E2 = np.array([[1, 0, 0],  
...                 [0, 1, 0],  
...                 [4, 0, 1]])  
  
>>> E3 = np.array([[1, 0, 0],  
...                 [-1, 1, 0],  
...                 [0, 0, 1]])  
  
>>> E1 @ E2 @ E3 @ A
```

```

... [4, 7, 8, 9]], dtype=np.float)

# Reduce the 0th column to zeros below the diagonal.
>>> A[1,0:] -= (A[1,0] / A[0,0]) * A[0]
>>> A[2,0:] -= (A[2,0] / A[0,0]) * A[0]

# Reduce the 1st column to zeros below the diagonal.
>>> A[2,1:] -= (A[2,1] / A[1,1]) * A[1,1:]
>>> print(A)
[[ 1.  1.  1.  1.]
 [ 0.  3.  1.  2.]
 [ 0.  0.  3.  3.]]

```

Note that the final row operation modifies only part of the third row to avoid spending the computation time of adding 0 to 0.

If a 0 appears on the main diagonal during any part of row reduction, the approach given above tries to divide by 0. Swapping the current row with one below it that does not have a 0 in the same column solves this problem. This is equivalent to left-multiplying by a type II elementary matrix, also called a *permutation matrix*.

### ACHTUNG!

Gaussian elimination is not always numerically stable. In other words, it is susceptible to rounding error that may result in an incorrect final matrix. Suppose that, due to roundoff error, the matrix  $A$  has a very small entry on the diagonal.

$$A = \begin{bmatrix} 10^{-15} & 1 \\ -1 & 0 \end{bmatrix}$$

Though  $10^{-15}$  is essentially zero, instead of swapping the first and second rows to put  $A$  in REF, a computer might multiply the first row by  $10^{15}$  and add it to the second row to eliminate the  $-1$ . The resulting matrix is far from what it would be if the  $10^{-15}$  were actually 0.

$$\begin{bmatrix} 10^{-15} & 1 \\ -1 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 10^{-15} & 1 \\ 0 & 10^{15} \end{bmatrix}$$

Round-off error can propagate through many steps in a calculation. The NumPy routines that employ row reduction use several tricks to minimize the impact of round-off error, but these tricks cannot fix every matrix.

### Problem B.1: Program simple row reduction to REF.

Write a function that reduces an arbitrary square matrix  $A$  to REF. You may assume that  $A$  is invertible and that a 0 will never appear on the main diagonal (so only use type III row reductions, not type II). Avoid operating on entries that you know will be 0 before and after a row operation. Use at most two nested loops.

Test your function with small test cases that you can check by hand. Consider using `np.random.randint()` to generate a few manageable tests cases.

## The LU Decomposition

---

The *LU decomposition* of a square matrix  $A$  is a factorization  $A = LU$  where  $U$  is the **upper** triangular REF of  $A$  and  $L$  is the **lower** triangular product of the type III elementary matrices whose inverses reduce  $A$  to  $U$ . The LU decomposition of  $A$  exists when  $A$  can be reduced to REF using only type III elementary matrices (without any row swaps). However, the rows of  $A$  can always be permuted in a way such that the decomposition exists. If  $P$  is a permutation matrix encoding the appropriate row swaps, then the decomposition  $PA = LU$  always exists.

Suppose  $A$  has an LU decomposition (not requiring row swaps). Then  $A$  can be reduced to REF with  $k$  row operations, corresponding to left-multiplying the type III elementary matrices  $E_1, \dots, E_k$ . Because there were no row swaps, each  $E_i$  is lower triangular, so each inverse  $E_i^{-1}$  is also lower triangular. Furthermore, since the product of lower triangular matrices is lower triangular,  $L$  is lower triangular:

$$\begin{aligned} E_k \dots E_2 E_1 A &= U \quad \longrightarrow \quad A = (E_k \dots E_2 E_1)^{-1} U \\ &= E_1^{-1} E_2^{-1} \dots E_k^{-1} U \\ &= LU. \end{aligned}$$

Thus,  $L$  can be computed by right-multiplying the identity by the matrices used to reduce  $U$ . However, in this special situation, each right-multiplication only changes one entry of  $L$ , matrix multiplication can be avoided altogether. The entire process, only slightly different than row reduction, is summarized below.

---

### Algorithm 10

```

1: procedure LU DECOMPOSITION( $A$ )
2:    $m, n \leftarrow \text{shape}(A)$                                  $\triangleright$  Store the dimensions of  $A$ .
3:    $U \leftarrow \text{copy}(A)$                                   $\triangleright$  Make a copy of  $A$  with np.copy().
4:    $L \leftarrow I_m$                                           $\triangleright$  The  $m \times m$  identity matrix.
5:   for  $j = 0 \dots n - 1$  do
6:     for  $i = j + 1 \dots m - 1$  do
7:        $L_{i,j} \leftarrow U_{i,j} / U_{j,j}$ 
8:        $U_{i,j:} \leftarrow U_{i,j:} - L_{i,j} U_{j,j:}$ 
9:   return  $L, U$ 

```

---

**Problem B.2: LU Decomposition**

Write a function that finds the LU decomposition of a square matrix. You may assume that the decomposition exists and requires no row swaps.

**Forward and Backward Substitution**

If  $PA = LU$  and  $A\mathbf{x} = b$ , then  $L\mathbf{U}\mathbf{x} = P\mathbf{A}\mathbf{x} = Pb$ . This system can be solved by first solving  $L\mathbf{y} = Pb$ , then  $\mathbf{U}\mathbf{x} = \mathbf{y}$ . Since  $L$  and  $U$  are both triangular, these systems can be solved with backward and forward substitution. We can thus compute the LU factorization of  $A$  once, then use substitution to efficiently solve  $A\mathbf{x} = b$  for various values of  $b$ .

Since the diagonal entries of  $L$  are all 1, the triangular system  $L\mathbf{y} = b$  has the form

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ l_{21} & 1 & 0 & \cdots & 0 \\ l_{31} & l_{32} & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & l_{n3} & \cdots & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \end{bmatrix}.$$

Matrix multiplication yields the equations

$$\begin{aligned} y_1 &= b_1, & y_1 &= b_1, \\ l_{21}y_1 + y_2 &= b_2, & y_2 &= b_2 - l_{21}y_1, \\ &\vdots &&\vdots \\ \sum_{j=1}^{k-1} l_{kj}y_j + y_k &= b_k, & y_k &= b_k - \sum_{j=1}^{k-1} l_{kj}y_j. \end{aligned} \tag{2.1}$$

The triangular system  $U\mathbf{x} = \mathbf{y}$  yields similar equations, but in reverse order:

$$\begin{aligned} \begin{bmatrix} u_{11} & u_{12} & u_{13} & \cdots & u_{1n} \\ 0 & u_{22} & u_{23} & \cdots & u_{2n} \\ 0 & 0 & u_{33} & \cdots & u_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & u_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} &= \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{bmatrix}, \\ u_{nn}x_n &= y_n, & x_n &= \frac{1}{u_{nn}}y_n, \\ u_{n-1,n-1}x_{n-1} + u_{n-1,n}x_n &= y_{n-1}, & x_{n-1} &= \frac{1}{u_{n-1,n-1}}(y_{n-1} - u_{n-1,n}x_n), \\ &\vdots &&\vdots \\ \sum_{j=k}^n u_{kj}x_j &= y_k, & x_k &= \frac{1}{u_{kk}} \left( y_k - \sum_{j=k+1}^n u_{kj}x_j \right). \end{aligned} \tag{2.2}$$

**Problem B.3: Program back and forward substitution.**

Write a function that, given  $A$  and  $b$ , solves the square linear system  $A\mathbf{x} = b$ . Use the function from Problem B to compute  $L$  and  $U$ , then use (2.1) and (2.2) to solve for  $\mathbf{y}$ , then  $\mathbf{x}$ . You may again assume that no row swaps are required ( $P = I$  in this case).

## SciPy

---

SciPy [scipy] is a powerful scientific computing library built upon NumPy. It includes high-level tools for linear algebra, statistics, signal processing, integration, optimization, machine learning, and more.

SciPy is typically imported with the convention `import scipy as sp`. However, SciPy is set up in a way that requires its submodules to be imported individually.<sup>1</sup>

```
>>> import scipy as sp
>>> hasattr(sp, "stats")           # The stats module isn't loaded yet.
False

>>> from scipy import stats        # Import stats explicitly. Access it
>>> hasattr(sp, "stats")           # with 'stats' or 'sp.stats'.
True
```

## Linear Algebra

---

NumPy and SciPy both have a linear algebra module, each called `linalg`, but SciPy's module is the larger of the two. Some of SciPy's common `linalg` functions are listed below.

Function	Returns
<code>det()</code>	The determinant of a square matrix.
<code>eig()</code>	The eigenvalues and eigenvectors of a square matrix.
<code>inv()</code>	The inverse of an invertible matrix.
<code>norm()</code>	The norm of a vector or matrix norm of a matrix.
<code>solve()</code>	The solution to $A\mathbf{x} = b$ (the system need not be square).

This library also includes routines for computing matrix decompositions.

```
>>> from scipy import linalg as la

# Make a random matrix and a random vector.
```

<sup>1</sup>SciPy modules like `linalg` are really *packages*, which are not initialized when SciPy is imported alone.

```

>>> A = np.random.random((1000,1000))
>>> b = np.random.random(1000)

# Compute the LU decomposition of A, including pivots.
>>> L, P = la.lu_factor(A)

# Use the LU decomposition to solve Ax = b.
>>> x = la.lu_solve((L,P), b)

# Check that the solution is legitimate.
>>> np.allclose(A @ x, b)
True

```

As with NumPy, SciPy's routines are all highly optimized. However, some algorithms are, by nature, faster than others.

#### Problem B.4: Time ways to solve $Ax = b$ with `scipy.linalg`.

*Write a function that times different `scipy.linalg` functions for solving square linear systems.*

*For various values of  $n$ , generate a random  $n \times n$  matrix  $A$  and a random  $n$ -vector  $b$  using `np.random.random()`. Time how long it takes to solve the system  $Ax = b$  with each of the following approaches:*

1. Invert  $A$  with `la.inv()` and left-multiply the inverse to  $b$ .
2. Use `la.solve()`.
3. Use `la.lu_factor()` and `la.lu_solve()` to solve the system with the LU decomposition.
4. Use `la.lu_factor()` and `la.lu_solve()`, but only time `la.lu_solve()` (not the time it takes to do the factorization with `la.lu_factor()`).

*Plot the system size  $n$  versus the execution times. Use log scales if needed.*

#### ACHTUNG!

Problem B demonstrates that computing a matrix inverse is computationally expensive. In fact, numerically inverting matrices is so costly that there is hardly ever a good reason to do it. Use a specific solver like `la.lu_solve()` whenever possible instead of using `la.inv()`.

## Sparse Matrices

---

Large linear systems can have tens of thousands of entries. Storing the corresponding matrices in memory can be difficult: a  $10^5 \times 10^5$  system requires around 40 GB to store in a NumPy array (4 bytes per entry  $\times 10^{10}$  entries). This is well beyond the amount of RAM in a normal laptop.

In applications where systems of this size arise, it is often the case that the system is *sparse*, meaning that most of the entries of the matrix are 0. SciPy's `sparse` module provides tools for efficiently constructing and manipulating 1- and 2-D sparse matrices. A sparse matrix only stores the nonzero values and the positions of these values. For sufficiently sparse matrices, storing the matrix as a sparse matrix may only take megabytes, rather than gigabytes.

For example, diagonal matrices are sparse. Storing an  $n \times n$  diagonal matrix in the naïve way means storing  $n^2$  values in memory. It is more efficient to instead store the diagonal entries in a 1-D array of  $n$  values. In addition to using less storage space, this allows for much faster matrix operations: the standard algorithm to multiply a matrix by a diagonal matrix involves  $n^3$  steps, but most of these are multiplying by or adding 0. A smarter algorithm can accomplish the same task much faster.

SciPy has seven sparse matrix types. Each type is optimized either for storing sparse matrices whose nonzero entries follow certain patterns, or for performing certain computations.

Name	Description	Advantages
<code>bsr_matrix</code>	Block Sparse Row	Specialized structure.
<code>coo_matrix</code>	Coordinate Format	Conversion among sparse formats.
<code>csc_matrix</code>	Compressed Sparse Column	Column-based operations and slicing.
<code>csr_matrix</code>	Compressed Sparse Row	Row-based operations and slicing.
<code>dia_matrix</code>	Diagonal Storage	Specialized structure.
<code>dok_matrix</code>	Dictionary of Keys	Element access, incremental construction.
<code>lil_matrix</code>	Row-based Linked List	Incremental construction.

### Creating Sparse Matrices

---

A regular, non-sparse matrix is called *full* or *dense*. Full matrices can be converted to each of the sparse matrix formats listed above. However, it is more memory efficient to never create the full matrix in the first place. There are three main approaches for creating sparse matrices from scratch.

- **Coordinate Format:** When all of the nonzero values and their positions are known, create the entire sparse matrix at once as a `coo_matrix`. All nonzero values are stored as a coordinate and a value. This format also converts quickly to other sparse matrix types.

```
>>> from scipy import sparse

# Define the rows, columns, and values separately.
>>> rows = np.array([0, 1, 0])
>>> cols = np.array([0, 1, 1])
```

```
>>> vals = np.array([3, 5, 2])
>>> A = sparse.coo_matrix((vals, (rows,cols)), shape=(3,3))
>>> print(A)
(0, 0)    3
(1, 1)    5
(0, 1)    2

# The toarray() method casts the sparse matrix as a NumPy array.
>>> print(A.toarray())           # Note that this method forfeits
[[3 2 0]           # all sparsity-related optimizations.
 [0 5 0]
 [0 0 0]]
```

- **DOK and LIL Formats:** If the matrix values and their locations are not known beforehand, construct the matrix incrementally with a `dok_matrix` or a `lil_matrix`. Indicate the size of the matrix, then change individual values with regular slicing syntax.

```
>>> B = sparse.lil_matrix((2,6))
>>> B[0,2] = 4
>>> B[1,3:] = 9

>>> print(B.toarray())
[[ 0.  0.  4.  0.  0.  0.]
 [ 0.  0.  0.  9.  9.  9.]]
```

- **DIA Format:** Use a `dia_matrix` to store matrices that have nonzero entries on only certain diagonals. The function `sparse.diags()` is one convenient way to create a `dia_matrix` from scratch. Additionally, every sparse matrix has a `setdiags()` method for modifying specified diagonals.

```
# Use sparse.diags() to create a matrix with diagonal entries.
>>> diagonals = [[1,2],[3,4,5],[6]]      # List the diagonal entries.
>>> offsets = [-1,0,3]                  # Specify the diagonal they go on↔
.
>>> print(sparse.diags(diagonals, offsets, shape=(3,4)).toarray())
[[ 3.  0.  0.  6.]
 [ 1.  4.  0.  0.]
 [ 0.  2.  5.  0.]]
```

# If all of the diagonals have the same entry, specify the entry alone.

```
>>> A = sparse.diags([1,3,6], offsets, shape=(3,4))
>>> print(A.toarray())
[[ 3.  0.  0.  6.]
 [ 1.  3.  0.  0.]]
```

```
[ 0.  1.  3.  0.]]  
  
# Modify a diagonal with the setdiag() method.  
>>> A.setdiag([4,4,4], 0)  
>>> print(A.toarray())  
[[ 4.  0.  0.  6.]  
 [ 1.  4.  0.  0.]  
 [ 0.  1.  4.  0.]]
```

- **BSR Format:** Many sparse matrices can be formulated as block matrices, and a block matrix can be stored efficiently as a `bsr_matrix`. Use `sparse.bmat()` or `sparse.block_diag()` to create a block matrix quickly.

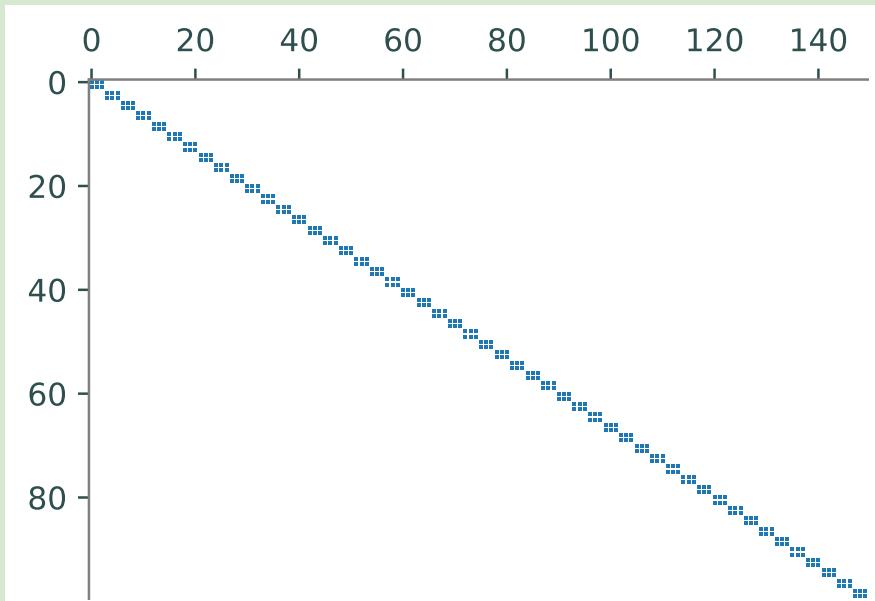
```
# Use sparse.bmat() to create a block matrix. Use 'None' for zero blocks.  
>>> A = sparse.coo_matrix(np.ones((2,2)))  
>>> B = sparse.coo_matrix(np.full((2,2), 2.))  
>>> print(sparse.bmat([[ A , None , A ],  
                      [None,  B , None]], format='bsr').toarray())  
[[ 1.  1.  0.  0.  1.  1.]  
 [ 1.  1.  0.  0.  1.  1.]  
 [ 0.  0.  2.  2.  0.  0.]  
 [ 0.  0.  2.  2.  0.  0.]]  
  
# Use sparse.block_diag() to construct a block diagonal matrix.  
>>> print(sparse.block_diag((A,B)).toarray())  
[[ 1.  1.  0.  0.]  
 [ 1.  1.  0.  0.]  
 [ 0.  0.  2.  2.]  
 [ 0.  0.  2.  2.]]
```

## NOTE

If a sparse matrix is too large to fit in memory as an array, it can still be visualized with Matplotlib's `plt.spy()`, which colors in the locations of the non-zero entries of the matrix.

```
>>> from matplotlib import pyplot as plt  
  
# Construct and show a matrix with 50 2x3 diagonal blocks.  
>>> B = sparse.coo_matrix([[1,3,5],[7,9,11]])  
>>> A = sparse.block_diag([B]*50)  
>>> plt.spy(A, markersize=1)
```

```
>>> plt.show()
```



### Problem B.5: Construct a large sparse matrix.

Let  $I$  be the  $n \times n$  identity matrix, and define

$$A = \begin{bmatrix} B & I & & \\ I & B & I & \\ & I & \ddots & \ddots & \\ & & \ddots & \ddots & I \\ & & & I & B \end{bmatrix}, \quad B = \begin{bmatrix} -4 & 1 & & \\ 1 & -4 & 1 & \\ & 1 & \ddots & \ddots & \\ & & \ddots & \ddots & 1 \\ & & & 1 & -4 \end{bmatrix},$$

where  $A$  is  $n^2 \times n^2$  and each block  $B$  is  $n \times n$ . The large matrix  $A$  is used in finite difference methods for solving Laplace's equation in two dimensions,  $\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$ .

Write a function that accepts an integer  $n$  and constructs and returns  $A$  as a sparse matrix. Use `plt.spy()` to check that your matrix has nonzero values in the correct places.

## Sparse Matrix Operations

Once a sparse matrix has been constructed, it should be converted to a `csr_matrix` or a `csc_matrix` with the matrix's `tocsr()` or `tocsc()` method. The CSR and CSC formats are optimized for row or column operations, respectively. To choose the correct format to use, determine what direction the matrix will be traversed.

For example, in the matrix-matrix multiplication  $AB$ ,  $A$  is traversed row-wise, but  $B$  is traversed column-wise. Thus  $A$  should be converted to a `csr_matrix` and  $B$  should be converted to a `csc_matrix`.

```
# Initialize a sparse matrix incrementally as a lil_matrix.
>>> A = sparse.lil_matrix((10000,10000))
>>> for k in range(10000):
...     A[np.random.randint(0,9999), np.random.randint(0,9999)] = k
...
>>> A
<10000x10000 sparse matrix of type '<type 'numpy.float64'>' with 9999 stored elements in LInked List format>

# Convert A to CSR and CSC formats to compute the matrix product AA.
>>> Acsr = A.tocsr()
>>> Acsc = A.tocsc()
>>> Acsr.dot(Acsc)
<10000x10000 sparse matrix of type '<type 'numpy.float64'>' with 10142 stored elements in Compressed Sparse Row format>
```

Beware that row-based operations on a `csc_matrix` are very slow, and similarly, column-based operations on a `csr_matrix` are very slow.

### ACHTUNG!

Many familiar NumPy operations have analogous routines in the `sparse` module. These methods take advantage of the sparse structure of the matrices and are, therefore, usually significantly faster. However, SciPy's sparse matrices behave a little differently than NumPy arrays.

Operation	<code>numpy</code>	<code>scipy.sparse</code>
Component-wise Addition	<code>A + B</code>	<code>A + B</code>
Scalar Multiplication	<code>2 * A</code>	<code>2 * A</code>
Component-wise Multiplication	<code>A * B</code>	<code>A.multiply(B)</code>
Matrix Multiplication	<code>A.dot(B), A @ B</code>	<code>A * B, A.dot(B), A @ B</code>

Note in particular the difference between `A * B` for NumPy arrays and SciPy sparse matrices. Do **not** use `np.dot()` to try to multiply sparse matrices, as it may treat the inputs incorrectly. The syntax

`A.dot(B)` is safest in most cases.

SciPy's sparse module has its own linear algebra library, `scipy.sparse.linalg`, designed for operating on sparse matrices. Like other SciPy modules, it must be imported explicitly.

```
>>> from scipy.sparse import linalg as spla
```

### Problem B.6: Time `scipy.sparse.linalg.spsolve()` against `sp.linalg.solve()`.

Write a function that times regular and sparse linear system solvers.

For various values of  $n$ , generate the  $n^2 \times n^2$  matrix  $A$  described in Problem B and a random vector  $b$  with  $n^2$  entries. Time how long it takes to solve the system  $Ax = b$  with each of the following approaches:

1. Convert  $A$  to CSR format and use `scipy.sparse.linalg.spsolve()` (`spla.spsolve()`).
2. Convert  $A$  to a NumPy array and use `scipy.linalg.solve()` (`la.solve()`).

In each experiment, only time how long it takes to solve the system (not how long it takes to convert  $A$  to the appropriate format).

Plot the system size  $n^2$  versus the execution times. As always, use log scales where appropriate and use a legend to label each line.

### ACHTUNG!

Even though there are fast algorithms for solving certain sparse linear systems, it is still very computationally difficult to invert sparse matrices. In fact, the inverse of a sparse matrix is usually not sparse. There is rarely a good reason to invert a matrix, sparse or dense.

See <http://docs.scipy.org/doc/scipy/reference/sparse.html> for additional details on SciPy's sparse module.

# Additional Material

---

## Improvements on the LU Decomposition

---

### Vectorization

---

Algorithm 10 uses two loops to compute the LU decomposition. With a little vectorization, the process can be reduced to a single loop.

---

### Algorithm 11

---

```

1: procedure FAST LU DECOMPOSITION( $A$ )
2:    $m, n \leftarrow \text{shape}(A)$ 
3:    $U \leftarrow \text{copy}(A)$ 
4:    $L \leftarrow I_m$ 
5:   for  $k = 0 \dots n - 1$  do
6:      $L_{k+1:k} \leftarrow U_{k+1:k} / U_{k,k}$ 
7:      $U_{k+1:k} \leftarrow U_{k+1:k} - L_{k+1:k} U_{k,k}^T$ 
8:   return  $L, U$ 
```

---

Note that step 7 is an *outer product*, not the regular dot product ( $\mathbf{x}\mathbf{y}^T$  instead of the usual  $\mathbf{x}^T\mathbf{y}$ ). Use `np.outer()` instead of `np.dot()` or `@` to get the desired result.

### Pivoting

---

Gaussian elimination iterates through the rows of a matrix, using the diagonal entry  $x_{k,k}$  of the matrix at the  $k$ th iteration to zero out all of the entries in the column below  $x_{k,k}$  ( $x_{i,k}$  for  $i \geq k$ ). This diagonal entry is called the *pivot*. Unfortunately, Gaussian elimination, and hence the LU decomposition, can be very numerically unstable if at any step the pivot is a very small number. Most professional row reduction algorithms avoid this problem via *partial pivoting*.

The idea is to choose the largest number (in magnitude) possible to be the pivot by swapping the pivot row<sup>2</sup> with another row before operating on the matrix. For example, the second and fourth rows of the following matrix are exchanged so that the pivot is  $-6$  instead of  $2$ .

$$\begin{bmatrix} \times & \times & \times & \times \\ 0 & 2 & \times & \times \\ 0 & 4 & \times & \times \\ 0 & -6 & \times & \times \end{bmatrix} \rightarrow \begin{bmatrix} \times & \times & \times & \times \\ 0 & -6 & \times & \times \\ 0 & 4 & \times & \times \\ 0 & 2 & \times & \times \end{bmatrix} \rightarrow \begin{bmatrix} \times & \times & \times & \times \\ 0 & -6 & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & \times & \times \end{bmatrix}$$

<sup>2</sup>Complete pivoting involves row and column swaps, but doing both operations is usually considered overkill.

A row swap is equivalent to left-multiplying by a type II elementary matrix, also called a *permutation matrix*.

$$\left[ \begin{array}{cccc} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{array} \right] \left[ \begin{array}{cccc} \times & \times & \times & \times \\ 0 & 2 & \times & \times \\ 0 & 4 & \times & \times \\ 0 & -6 & \times & \times \end{array} \right] = \left[ \begin{array}{cccc} \times & \times & \times & \times \\ 0 & -6 & \times & \times \\ 0 & 4 & \times & \times \\ 0 & 2 & \times & \times \end{array} \right]$$

For the LU decomposition, if the permutation matrix at step  $k$  is  $P_k$ , then  $P = P_k \dots P_2 P_1$  yields  $PA = LU$ . The complete algorithm is given below.

---

**Algorithm 12**


---

```

1: procedure LU DECOMPOSITION WITH PARTIAL PIVOTING( $A$ )
2:    $m, n \leftarrow \text{shape}(A)$ 
3:    $U \leftarrow \text{copy}(A)$ 
4:    $L \leftarrow I_m$ 
5:    $P \leftarrow [0, 1, \dots, n-1]$                                  $\triangleright$  See tip 2 below.
6:   for  $k = 0 \dots n-1$  do
7:     Select  $i \geq k$  that maximizes  $|U_{i,k}|$ 
8:      $U_{k,:} \leftrightarrow U_{i,:}$                                       $\triangleright$  Swap the two rows.
9:      $L_{k,:} \leftrightarrow L_{i,:}$                                       $\triangleright$  Swap the two rows.
10:     $P_k \leftrightarrow P_i$                                           $\triangleright$  Swap the two entries.
11:     $L_{k+1:,k} \leftarrow U_{k+1:,k} / U_{k,k}$ 
12:     $U_{k+1:,k} \leftarrow U_{k+1:,k} - L_{k+1:,k} U_{k,k}^T$ 
13:   return  $L, U, P$ 

```

---

The following tips may be helpful for implementing this algorithm:

1. Since NumPy arrays are mutable, use `np.copy()` to reassign the rows of an array simultaneously.
2. Instead of storing  $P$  as an  $n \times n$  array, fancy indexing allows us to encode row swaps in a 1-D array of length  $n$ . Initialize  $P$  as the array  $[0, 1, \dots, n]$ . After performing a row swap on  $A$ , perform the same operations on  $P$ . Then the matrix product  $PA$  will be the same as  $A[P]$ .

```

>>> A = np.zeros(3) + np.vstack(np.arange(3))
>>> P = np.arange(3)
>>> print(A)
[[ 0.  0.  0.]
 [ 1.  1.  1.]
 [ 2.  2.  2.]]

# Swap rows 1 and 2.
>>> A[1], A[2] = np.copy(A[2]), np.copy(A[1])
>>> P[1], P[2] = P[2], P[1]

```

```
>>> print(A)                                     # A with the new row arrangement.
[[ 0.  0.  0.]
 [ 2.  2.  2.]
 [ 1.  1.  1.]]
>>> print(P)                                    # The permutation of the rows.
[0 2 1]
>>> print(A[P])                                # A with the original row arrangement.
[[ 0.  0.  0.]
 [ 1.  1.  1.]
 [ 2.  2.  2.]]
```

There are potential cases where even partial pivoting does not eliminate catastrophic numerical errors in Gaussian elimination, but the odds of having such an amazingly poor matrix are essentially zero. The numerical analyst J.H. Wilkinson captured the likelihood of encountering such a matrix in a natural application when he said, “Anyone that unlucky has already been run over by a bus!”

### In Place

---

The LU decomposition can be performed in place (overwriting the original matrix  $A$ ) by storing  $U$  on and above the main diagonal of the array and storing  $L$  below it. The main diagonal of  $L$  does not need to be stored since all of its entries are 1. This format saves an entire array of memory, and is how `scipy.linalg.lu_factor()` returns the factorization.

## More Applications of the LU Decomposition

---

The LU decomposition can also be used to compute inverses and determinants with relative efficiency.

- **Inverse:**  $(PA)^{-1} = (LU)^{-1} \implies A^{-1}P^{-1} = U^{-1}L^{-1} \implies LUA^{-1} = P$ . Solve  $LUA_i = \mathbf{p}_i$  with forward and backward substitution (as in Problem B) for every column  $\mathbf{p}_i$  of  $P$ . Then

$$A^{-1} = \left[ \begin{array}{c|c|c|c} \mathbf{a}_1 & \mathbf{a}_2 & \cdots & \mathbf{a}_n \end{array} \right],$$

the matrix where  $\mathbf{a}_k$  is the  $k$ th column.

- **Determinant:**  $\det(A) = \det(P^{-1}LU) = \frac{\det(L)\det(U)}{\det(P)}$ . The determinant of a triangular matrix is the product of its diagonal entries. Since every diagonal entry of  $L$  is 1,  $\det(L) = 1$ . Also,  $P$  is just a row permutation of the identity matrix (which has determinant 1), and a single row swap negates the determinant. Then if  $S$  is the number of row swaps, the determinant is

$$\det(A) = (-1)^S \prod_{i=1}^n u_{ii}.$$

## The Cholesky Decomposition

---

A square matrix  $A$  is called *positive definite* if  $\mathbf{z}^T A \mathbf{z} > 0$  for all nonzero vectors  $\mathbf{z}$ . In addition,  $A$  is called *Hermitian* if  $A = A^H = \overline{A^T}$ . If  $A$  is Hermitian positive definite, it has a *Cholesky Decomposition*  $A = U^H U$  where  $U$  is upper triangular with real, positive entries on the diagonal. This is the matrix equivalent to taking the square root of a positive real number.

The Cholesky decomposition takes advantage of the conjugate symmetry of  $A$  to simultaneously reduce the columns *and* rows of  $A$  to zeros (except for the diagonal). It thus requires only half of the calculations and memory of the LU decomposition. Furthermore, the algorithm is *numerically stable*, which means, roughly speaking, that round-off errors do not propagate throughout the computation.

---

### Algorithm 13

---

```

1: procedure CHOLESKY DECOMPOSITION( $A$ )
2:    $n, n \leftarrow \text{shape}(A)$ 
3:    $U \leftarrow \text{np.triu}(A)$                                  $\triangleright$  Get the upper-triangular part of  $A$ .
4:   for  $i = 0 \dots n - 1$  do
5:     for  $j = i + 1 \dots n - 1$  do
6:        $U_{j,j} \leftarrow U_{j,j} - U_{i,j} \overline{U_{ij}} / U_{ii}$ 
7:        $U_{i,i} \leftarrow U_{i,i} / \sqrt{U_{ii}}$ 
8:   return  $U$ 

```

---

As with the LU decomposition, SciPy's `linalg` module has optimized routines, `la.cho_factor()` and `la.cho_solve()`, for using the Cholesky decomposition.

# C. Systems of Equations

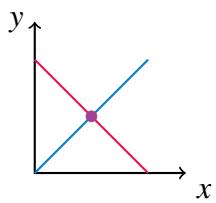
## C.1 Systems of Equations, Geometry

### Outcomes

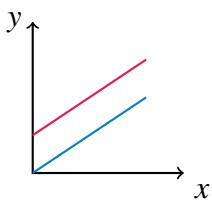
- A. Relate the types of solution sets of a system of two (three) variables to the intersections of lines in a plane (the intersection of planes in three space)

As you may remember, linear equations like  $2x + 3y = 6$  can be graphed as straight lines in the coordinate plane. We say that this equation is in two variables, in this case  $x$  and  $y$ . Suppose you have two such equations, each of which can be graphed as a straight line, and consider the resulting graph of two lines. What would it mean if there exists a point of intersection between the two lines? This point, which lies on *both* graphs, gives  $x$  and  $y$  values for which both equations are true. In other words, this point gives the ordered pair  $(x, y)$  that satisfy both equations. If the point  $(x, y)$  is a point of intersection, we say that  $(x, y)$  is a **solution** to the two equations. In linear algebra, we often are concerned with finding the solution(s) to a system of equations, if such solutions exist. First, we consider graphical representations of solutions and later we will consider the algebraic methods for finding solutions.

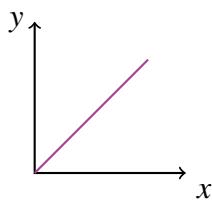
When looking for the intersection of two lines in a graph, several situations may arise. The following picture demonstrates the possible situations when considering two equations (two lines in the graph) involving two variables.



One Solution



No Solutions



Infinitely Many Solutions

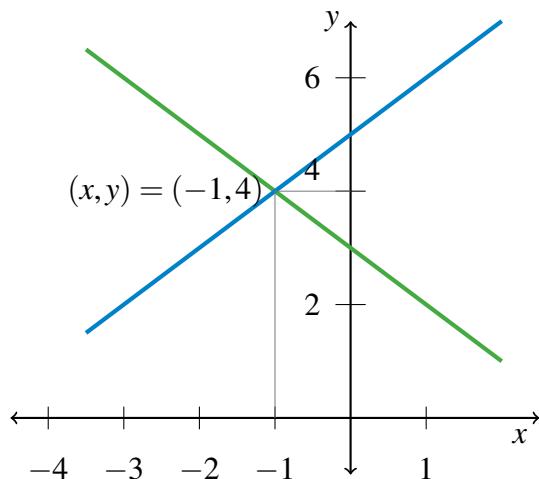
In the first diagram, there is a unique point of intersection, which means that there is only one (unique) solution to the two equations. In the second, there are no points of intersection and no solution. When no solution exists, this means that the two lines are parallel and they never intersect. The third situation which can occur, as demonstrated in diagram three, is that the two lines are really the same line. For example,  $x + y = 1$  and  $2x + 2y = 2$  are equations which when graphed yield the same line. In this case there are infinitely many points which are solutions of these two equations, as every ordered pair which is on the graph of the line satisfies both equations. When considering linear systems of equations, there are always three types of solutions possible; exactly one (unique) solution, infinitely many solutions, or no solution.

**Example C.1: A Graphical Solution**

Use a graph to find the solution to the following system of equations

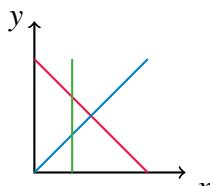
$$\begin{aligned}x + y &= 3 \\y - x &= 5\end{aligned}$$

**Solution.** Through graphing the above equations and identifying the point of intersection, we can find the solution(s). Remember that we must have either one solution, infinitely many, or no solutions at all. The following graph shows the two equations, as well as the intersection. Remember, the point of intersection represents the solution of the two equations, or the  $(x, y)$  which satisfy both equations. In this case, there is one point of intersection at  $(-1, 4)$  which means we have one unique solution,  $x = -1, y = 4$ .

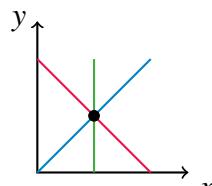


In the above example, we investigated the intersection point of two equations in two variables,  $x$  and  $y$ . Now we will consider the graphical solutions of three equations in two variables.

Consider a system of three equations in two variables. Again, these equations can be graphed as straight lines in the plane, so that the resulting graph contains three straight lines. Recall the three possible types of solutions; no solution, one solution, and infinitely many solutions. There are now more complex ways of achieving these situations, due to the presence of the third line. For example, you can imagine the case of three intersecting lines having no common point of intersection. Perhaps you can also imagine three intersecting lines which do intersect at a single point. These two situations are illustrated below.



No Solution

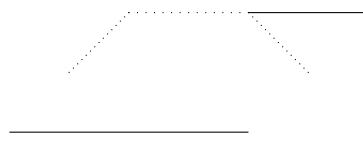


One Solution

Consider the first picture above. While all three lines intersect with one another, there is no common point of intersection where all three lines meet at one point. Hence, there is no solution to the three equations. Remember, a solution is a point  $(x,y)$  which satisfies **all** three equations. In the case of the second picture, the lines intersect at a common point. This means that there is one solution to the three equations whose graphs are the given lines. You should take a moment now to draw the graph of a system which results in three parallel lines. Next, try the graph of three identical lines. Which type of solution is represented in each of these graphs?

We have now considered the graphical solutions of systems of two equations in two variables, as well as three equations in two variables. However, there is no reason to limit our investigation to equations in two variables. We will now consider equations in three variables.

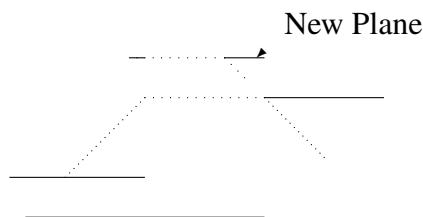
You may recall that equations in three variables, such as  $2x + 4y - 5z = 8$ , form a plane. Above, we were looking for intersections of lines in order to identify any possible solutions. When graphically solving systems of equations in three variables, we look for intersections of planes. These points of intersection give the  $(x,y,z)$  that satisfy all the equations in the system. What types of solutions are possible when working with three variables? Consider the following picture involving two planes, which are given by two equations in three variables.



Notice how these two planes intersect in a line. This means that the points  $(x,y,z)$  on this line satisfy both equations in the system. Since the line contains infinitely many points, this system has infinitely many solutions.

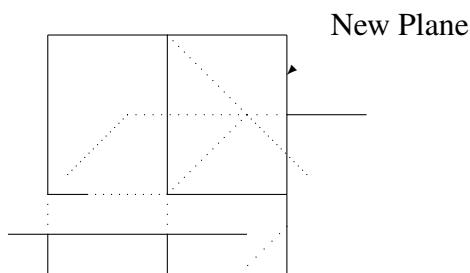
It could also happen that the two planes fail to intersect. However, is it possible to have two planes intersect at a single point? Take a moment to attempt drawing this situation, and convince yourself that it is not possible! This means that when we have only two equations in three variables, there is no way to have a unique solution! Hence, the types of solutions possible for two equations in three variables are no solution or infinitely many solutions.

Now imagine adding a third plane. In other words, consider three equations in three variables. What types of solutions are now possible? Consider the following diagram.

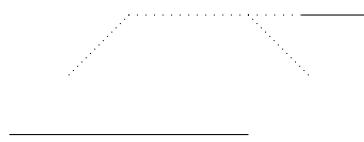


In this diagram, there is no point which lies in all three planes. There is no intersection between **all** planes so there is no solution. The picture illustrates the situation in which the line of intersection of the new plane with one of the original planes forms a line parallel to the line of intersection of the first two planes. However, in three dimensions, it is possible for two lines to fail to intersect even though they are not parallel. Such lines are called **skew lines**.

Recall that when working with two equations in three variables, it was not possible to have a unique solution. Is it possible when considering three equations in three variables? In fact, it is possible, and we demonstrate this situation in the following picture.



In this case, the three planes have a single point of intersection. Can you think of other types of solutions possible? Another is that the three planes could intersect in a line, resulting in infinitely many solutions, as in the following diagram.



We have now seen how three equations in three variables can have no solution, a unique solution, or intersect in a line resulting in infinitely many solutions. It is also possible that the three equations graph the same plane, which also leads to infinitely many solutions.

You can see that when working with equations in three variables, there are many more ways to achieve the different types of solutions than when working with two variables. It may prove enlightening to spend time imagining (and drawing) many possible scenarios, and you should take some time to try a few.

You should also take some time to imagine (and draw) graphs of systems in more than three variables. Equations like  $x + y - 2z + 4w = 8$  with more than three variables are often called **hyper-planes**. You may soon realize that it is tricky to draw the graphs of hyper-planes! Through the tools of linear algebra, we can algebraically examine these types of systems which are difficult to graph. In the following section, we will consider these algebraic tools.

## C.2 Systems Of Equations, Algebraic Procedures

### Outcomes

- A. Use elementary operations to find the solution to a linear system of equations.
- B. Find the row-echelon form and reduced row-echelon form of a matrix.
- C. Determine whether a system of linear equations has no solution, a unique solution or an infinite number of solutions from its row-echelon form.
- D. Solve a system of equations using Gaussian Elimination and Gauss-Jordan Elimination.
- E. Model a physical system with linear equations and then solve.

We have taken an in depth look at graphical representations of systems of equations, as well as how to find possible solutions graphically. Our attention now turns to working with systems algebraically.

### Definition C.2: System of Linear Equations

A **system of linear equations** is a list of equations,

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ &\vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &= b_m \end{aligned}$$

where  $a_{ij}$  and  $b_j$  are real numbers. The above is a system of  $m$  equations in the  $n$  variables,  $x_1, x_2, \dots, x_n$ . Written more simply in terms of summation notation, the above can be written in the form

$$\sum_{j=1}^n a_{ij}x_j = b_i, \quad i = 1, 2, 3, \dots, m$$

The relative size of  $m$  and  $n$  is not important here. Notice that we have allowed  $a_{ij}$  and  $b_j$  to be any real number. We can also call these numbers **scalars**. We will use this term throughout the text, so keep in mind that the term **scalar** just means that we are working with real numbers.

Now, suppose we have a system where  $b_i = 0$  for all  $i$ . In other words every equation equals 0. This is a special type of system.

**Definition C.3: Homogeneous System of Equations**

A system of equations is called **homogeneous** if each equation in the system is equal to 0. A homogeneous system has the form

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= 0 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= 0 \\ &\vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &= 0 \end{aligned}$$

where  $a_{ij}$  are scalars and  $x_i$  are variables.

Recall from the previous section that our goal when working with systems of linear equations was to find the point of intersection of the equations when graphed. In other words, we looked for the solutions to the system. We now wish to find these solutions algebraically. We want to find values for  $x_1, \dots, x_n$  which solve all of the equations. If such a set of values exists, we call  $(x_1, \dots, x_n)$  the **solution set**.

Recall the above discussions about the types of solutions possible. We will see that systems of linear equations will have one unique solution, infinitely many solutions, or no solution. Consider the following definition.

**Definition C.4: Consistent and Inconsistent Systems**

A system of linear equations is called **consistent** if there exists at least one solution. It is called **inconsistent** if there is no solution.

If you think of each equation as a condition which must be satisfied by the variables, consistent would mean there is some choice of variables which can satisfy **all** the conditions. Inconsistent would mean there is no choice of the variables which can satisfy all of the conditions.

The following sections provide methods for determining if a system is consistent or inconsistent, and finding solutions if they exist.

### C.2.1. Elementary Operations

---

We begin this section with an example. Recall from Example C.1 that the solution to the given system was  $(x, y) = (-1, 4)$ .

**Example C.5: Verifying an Ordered Pair is a Solution**

Algebraically verify that  $(x, y) = (-1, 4)$  is a solution to the following system of equations.

$$\begin{aligned} x + y &= 3 \\ y - x &= 5 \end{aligned}$$

**Solution.** By graphing these two equations and identifying the point of intersection, we previously found that  $(x, y) = (-1, 4)$  is the unique solution.

We can verify algebraically by substituting these values into the original equations, and ensuring that the equations hold. First, we substitute the values into the first equation and check that it equals 3.

$$x + y = (-1) + (4) = 3$$

This equals 3 as needed, so we see that  $(-1, 4)$  is a solution to the first equation. Substituting the values into the second equation yields

$$y - x = (4) - (-1) = 4 + 1 = 5$$

which is true. For  $(x, y) = (-1, 4)$  each equation is true and therefore, this is a solution to the system. ♠

Now, the interesting question is this: If you were not given these numbers to verify, how could you algebraically determine the solution? Linear algebra gives us the tools needed to answer this question. The following basic operations are important tools that we will utilize.

### Definition C.6: Elementary Operations

**Elementary operations** are those operations consisting of the following.

1. Interchange the order in which the equations are listed.
2. Multiply any equation by a nonzero number.
3. Replace any equation with itself added to a multiple of another equation.

It is important to note that none of these operations will change the set of solutions of the system of equations. In fact, elementary operations are the *key tool* we use in linear algebra to find solutions to systems of equations.

Consider the following example.

### Example C.7: Effects of an Elementary Operation

Show that the system

$$\begin{aligned} x + y &= 7 \\ 2x - y &= 8 \end{aligned}$$

has the same solution as the system

$$\begin{aligned} x + y &= 7 \\ -3y &= -6 \end{aligned}$$

**Solution.** Notice that the second system has been obtained by taking the second equation of the first system and adding -2 times the first equation, as follows:

$$2x - y + (-2)(x + y) = 8 + (-2)(7)$$

By simplifying, we obtain

$$-3y = -6$$

which is the second equation in the second system. Now, from here we can solve for  $y$  and see that  $y = 2$ . Next, we substitute this value into the first equation as follows

$$x + y = x + 2 = 7$$

Hence  $x = 5$  and so  $(x, y) = (5, 2)$  is a solution to the second system. We want to check if  $(5, 2)$  is also a solution to the first system. We check this by substituting  $(x, y) = (5, 2)$  into the system and ensuring the equations are true.

$$\begin{aligned} x + y &= (5) + (2) = 7 \\ 2x - y &= 2(5) - (2) = 8 \end{aligned}$$

Hence,  $(5, 2)$  is also a solution to the first system. ♠

This example illustrates how an elementary operation applied to a system of two equations in two variables does not affect the solution set. However, a linear system may involve many equations and many variables and there is no reason to limit our study to small systems. For any size of system in any number of variables, the solution set is still the collection of solutions to the equations. In every case, the above operations of Definition C.6 do not change the set of solutions to the system of linear equations.

In the following theorem, we use the notation  $E_i$  to represent an equation, while  $b_i$  denotes a constant.

### Theorem C.8: Elementary Operations and Solutions

Suppose you have a system of two linear equations

$$\begin{aligned} E_1 &= b_1 \\ E_2 &= b_2 \end{aligned} \tag{3.1}$$

Then the following systems have the same solution set as 3.1:

1.

$$\begin{aligned} E_2 &= b_2 \\ E_1 &= b_1 \end{aligned} \tag{3.2}$$

2.

$$\begin{aligned} E_1 &= b_1 \\ kE_2 &= kb_2 \end{aligned} \tag{3.3}$$

for any scalar  $k$ , provided  $k \neq 0$ .

3.

$$\begin{aligned} E_1 &= b_1 \\ E_2 + kE_1 &= b_2 + kb_1 \end{aligned} \tag{3.4}$$

for any scalar  $k$  (including  $k = 0$ ).

Before we proceed with the proof of Theorem C.8, let us consider this theorem in context of Example

C.7. Then,

$$\begin{aligned} E_1 &= x + y, \quad b_1 = 7 \\ E_2 &= 2x - y, \quad b_2 = 8 \end{aligned}$$

Recall the elementary operations that we used to modify the system in the solution to the example. First, we added  $(-2)$  times the first equation to the second equation. In terms of Theorem C.8, this action is given by

$$E_2 + (-2)E_1 = b_2 + (-2)b_1$$

or

$$2x - y + (-2)(x + y) = 8 + (-2)7$$

This gave us the second system in Example C.7, given by

$$\begin{aligned} E_1 &= b_1 \\ E_2 + (-2)E_1 &= b_2 + (-2)b_1 \end{aligned}$$

From this point, we were able to find the solution to the system. Theorem C.8 tells us that the solution we found is in fact a solution to the original system.

Stated simply, the above theorem shows that the elementary operations do not change the solution set of a system of equations.

We will now look at an example of a system of three equations and three variables. Similarly to the previous examples, the goal is to find values for  $x, y, z$  such that each of the given equations are satisfied when these values are substituted in.

### Example C.9: Solving a System of Equations with Elementary Operations

*Find the solutions to the system,*

$$\begin{aligned} x + 3y + 6z &= 25 \\ 2x + 7y + 14z &= 58 \\ 2y + 5z &= 19 \end{aligned} \tag{3.5}$$

**Solution.** We can relate this system to Theorem C.8 above. In this case, we have

$$\begin{aligned} E_1 &= x + 3y + 6z, \quad b_1 = 25 \\ E_2 &= 2x + 7y + 14z, \quad b_2 = 58 \\ E_3 &= 2y + 5z, \quad b_3 = 19 \end{aligned}$$

Theorem C.8 claims that if we do elementary operations on this system, we will not change the solution set. Therefore, we can solve this system using the elementary operations given in Definition C.6. First, replace the second equation by  $(-2)$  times the first equation added to the second. This yields the system

$$\begin{aligned} x + 3y + 6z &= 25 \\ y + 2z &= 8 \\ 2y + 5z &= 19 \end{aligned} \tag{3.6}$$

Now, replace the third equation with  $(-2)$  times the second added to the third. This yields the system

$$\begin{aligned}x + 3y + 6z &= 25 \\y + 2z &= 8 \\z &= 3\end{aligned}\tag{3.7}$$

At this point, we can easily find the solution. Simply take  $z = 3$  and substitute this back into the previous equation to solve for  $y$ , and similarly to solve for  $x$ .

$$\begin{aligned}x + 3y + 6(3) &= x + 3y + 18 = 25 \\y + 2(3) &= y + 6 = 8 \\z &= 3\end{aligned}$$

The second equation is now

$$y + 6 = 8$$

You can see from this equation that  $y = 2$ . Therefore, we can substitute this value into the first equation as follows:

$$x + 3(2) + 18 = 25$$

By simplifying this equation, we find that  $x = 1$ . Hence, the solution to this system is  $(x, y, z) = (1, 2, 3)$ . This process is called **back substitution**.

Alternatively, in 3.7 you could have continued as follows. Add  $(-2)$  times the third equation to the second and then add  $(-6)$  times the second to the first. This yields

$$\begin{aligned}x + 3y &= 7 \\y &= 2 \\z &= 3\end{aligned}$$

Now add  $(-3)$  times the second to the first. This yields

$$\begin{aligned}x &= 1 \\y &= 2 \\z &= 3\end{aligned}$$

a system which has the same solution set as the original system. This avoided back substitution and led to the same solution set. It is your decision which you prefer to use, as both methods lead to the correct solution,  $(x, y, z) = (1, 2, 3)$ . ♠

### C.2.2. Gaussian Elimination

---

The work we did in the previous section will always find the solution to the system. In this section, we will explore a less cumbersome way to find the solutions. First, we will represent a linear system with an **augmented matrix**. A **matrix** is simply a rectangular array of numbers. The size or dimension of a matrix is defined as  $m \times n$  where  $m$  is the number of rows and  $n$  is the number of columns. In order to construct an augmented matrix from a linear system, we create a **coefficient matrix** from the coefficients

of the variables in the system, as well as a **constant matrix** from the constants. The coefficients from one equation of the system create one row of the augmented matrix.

For example, consider the linear system in Example C.9

$$\begin{aligned}x + 3y + 6z &= 25 \\2x + 7y + 14z &= 58 \\2y + 5z &= 19\end{aligned}$$

This system can be written as an augmented matrix, as follows

$$\left[ \begin{array}{ccc|c} 1 & 3 & 6 & 25 \\ 2 & 7 & 14 & 58 \\ 0 & 2 & 5 & 19 \end{array} \right]$$

Notice that it has exactly the same information as the original system. Here it is understood that the first column contains the coefficients from  $x$  in each equation, in order,  $\begin{bmatrix} 1 \\ 2 \\ 0 \end{bmatrix}$ . Similarly, we create a column from the coefficients on  $y$  in each equation,  $\begin{bmatrix} 3 \\ 7 \\ 2 \end{bmatrix}$  and a column from the coefficients on  $z$  in each equation,  $\begin{bmatrix} 6 \\ 14 \\ 5 \end{bmatrix}$ . For a system of more than three variables, we would continue in this way constructing a column for each variable. Similarly, for a system of less than three variables, we simply construct a column for each variable.

Finally, we construct a column from the constants of the equations,  $\begin{bmatrix} 25 \\ 58 \\ 19 \end{bmatrix}$ .

The rows of the augmented matrix correspond to the equations in the system. For example, the top row in the augmented matrix,  $\begin{bmatrix} 1 & 3 & 6 & | & 25 \end{bmatrix}$  corresponds to the equation

$$x + 3y + 6z = 25.$$

Consider the following definition.

**Definition C.10: Augmented Matrix of a Linear System**

For a linear system of the form

$$a_{11}x_1 + \cdots + a_{1n}x_n = b_1$$

⋮

$$a_{m1}x_1 + \cdots + a_{mn}x_n = b_m$$

where the  $x_i$  are variables and the  $a_{ij}$  and  $b_i$  are constants, the augmented matrix of this system is given by

$$\left[ \begin{array}{ccc|c} a_{11} & \cdots & a_{1n} & b_1 \\ \vdots & & \vdots & \vdots \\ a_{m1} & \cdots & a_{mn} & b_m \end{array} \right]$$

Now, consider elementary operations in the context of the augmented matrix. The elementary operations in Definition C.6 can be used on the rows just as we used them on equations previously. Changes to a system of equations in as a result of an elementary operation are equivalent to changes in the augmented matrix resulting from the corresponding row operation. Note that Theorem C.8 implies that any elementary row operations used on an augmented matrix will not change the solution to the corresponding system of equations. We now formally define elementary row operations. These are the *key tool* we will use to find solutions to systems of equations.

**Definition C.11: Elementary Row Operations**

The **elementary row operations** (also known as **row operations**) consist of the following

1. Switch two rows.
2. Multiply a row by a nonzero number.
3. Replace a row by any multiple of another row added to it.

Recall how we solved Example C.9. We can do the exact same steps as above, except now in the context of an augmented matrix and using row operations. The augmented matrix of this system is

$$\left[ \begin{array}{ccc|c} 1 & 3 & 6 & 25 \\ 2 & 7 & 14 & 58 \\ 0 & 2 & 5 & 19 \end{array} \right]$$

Thus the first step in solving the system given by 3.5 would be to take  $(-2)$  times the first row of the augmented matrix and add it to the second row,

$$\left[ \begin{array}{ccc|c} 1 & 3 & 6 & 25 \\ 0 & 1 & 2 & 8 \\ 0 & 2 & 5 & 19 \end{array} \right]$$

Note how this corresponds to 3.6. Next take  $(-2)$  times the second row and add to the third,

$$\left[ \begin{array}{ccc|c} 1 & 3 & 6 & 25 \\ 0 & 1 & 2 & 8 \\ 0 & 0 & 1 & 3 \end{array} \right]$$

This augmented matrix corresponds to the system

$$\begin{aligned} x + 3y + 6z &= 25 \\ y + 2z &= 8 \\ z &= 3 \end{aligned}$$

which is the same as 3.7. By back substitution you obtain the solution  $x = 1$ ,  $y = 2$ , and  $z = 3$ .

Through a systematic procedure of row operations, we can simplify an augmented matrix and carry it to **row-echelon form** or **reduced row-echelon form**, which we define next. These forms are used to find the solutions of the system of equations corresponding to the augmented matrix.

In the following definitions, the term **leading entry** refers to the first nonzero entry of a row when scanning the row from left to right.

### Definition C.12: Row-Echelon Form

An augmented matrix is in **row-echelon form** if

1. All nonzero rows are above any rows of zeros.
2. Each leading entry of a row is in a column to the right of the leading entries of any row above it.
3. Each leading entry of a row is equal to 1.

We also consider another reduced form of the augmented matrix which has one further condition.

### Definition C.13: Reduced Row-Echelon Form

An augmented matrix is in **reduced row-echelon form** if

1. All nonzero rows are above any rows of zeros.
2. Each leading entry of a row is in a column to the right of the leading entries of any rows above it.
3. Each leading entry of a row is equal to 1.
4. All entries in a column above and below a leading entry are zero.

Notice that the first three conditions on a reduced row-echelon form matrix are the same as those for row-echelon form.

Hence, every reduced row-echelon form matrix is also in row-echelon form. The converse is not necessarily true; we cannot assume that every matrix in row-echelon form is also in reduced row-echelon form. However, it often happens that the row-echelon form is sufficient to provide information about the solution of a system.

The following examples describe matrices in these various forms. As an exercise, take the time to carefully verify that they are in the specified form.

### Example C.14: Not in Row-Echelon Form

*The following augmented matrices are not in row-echelon form (and therefore also not in reduced row-echelon form).*

$$\left[ \begin{array}{ccc|c} 0 & 0 & 0 & 0 \\ 1 & 2 & 3 & 3 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{array} \right], \left[ \begin{array}{cc|c} 1 & 2 & 3 \\ 2 & 4 & -6 \\ 4 & 0 & 7 \end{array} \right], \left[ \begin{array}{ccc|c} 0 & 2 & 3 & 3 \\ 1 & 5 & 0 & 2 \\ 7 & 5 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{array} \right]$$

### Example C.15: Matrices in Row-Echelon Form

*The following augmented matrices are in row-echelon form, but not in reduced row-echelon form.*

$$\left[ \begin{array}{ccccc|c} 1 & 0 & 6 & 5 & 8 & 2 \\ 0 & 0 & 1 & 2 & 7 & 3 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{array} \right], \left[ \begin{array}{ccc|c} 1 & 3 & 5 & 4 \\ 0 & 1 & 0 & 7 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{array} \right], \left[ \begin{array}{ccc|c} 1 & 0 & 6 & 0 \\ 0 & 1 & 4 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{array} \right]$$

Notice that we could apply further row operations to these matrices to carry them to reduced row-echelon form. Take the time to try that on your own. Consider the following matrices, which are in reduced row-echelon form.

### Example C.16: Matrices in Reduced Row-Echelon Form

*The following augmented matrices are in reduced row-echelon form.*

$$\left[ \begin{array}{ccccc|c} 1 & 0 & 0 & 5 & 0 & 0 \\ 0 & 0 & 1 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{array} \right], \left[ \begin{array}{ccc|c} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{array} \right], \left[ \begin{array}{ccc|c} 1 & 0 & 0 & 4 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 0 \end{array} \right]$$

One way in which the row-echelon form of a matrix is useful is in identifying the pivot positions and pivot columns of the matrix.

**Definition C.17: Pivot Position and Pivot Column**

A **pivot position** in a matrix is the location of a leading entry in the row-echelon form of a matrix.  
A **pivot column** is a column that contains a pivot position.

For example consider the following.

**Example C.18: Pivot Position**

Let

$$A = \left[ \begin{array}{ccc|c} 1 & 2 & 3 & 4 \\ 3 & 2 & 1 & 6 \\ 4 & 4 & 4 & 10 \end{array} \right]$$

Where are the pivot positions and pivot columns of the augmented matrix A?

**Solution.** The row-echelon form of this matrix is

$$\left[ \begin{array}{ccc|c} 1 & 2 & 3 & 4 \\ 0 & 1 & 2 & \frac{3}{2} \\ 0 & 0 & 0 & 0 \end{array} \right]$$

This is all we need in this example, but note that this matrix is not in reduced row-echelon form.

In order to identify the pivot positions in the original matrix, we look for the leading entries in the row-echelon form of the matrix. Here, the entry in the first row and first column, as well as the entry in the second row and second column are the leading entries. Hence, these locations are the pivot positions. We identify the pivot positions in the original matrix, as in the following:

$$\left[ \begin{array}{ccc|c} \boxed{1} & 2 & 3 & 4 \\ 3 & \boxed{2} & 1 & 6 \\ 4 & 4 & 4 & 10 \end{array} \right]$$

Thus the pivot columns in the matrix are the first two columns. ♠

The following is an algorithm for carrying a matrix to row-echelon form and reduced row-echelon form. You may wish to use this algorithm to carry the above matrix to row-echelon form or reduced row-echelon form yourself for practice.

Most often we will apply this algorithm to an augmented matrix in order to find the solution to a system of linear equations. However, we can use this algorithm to compute the reduced row-echelon form of any matrix which could be useful in other applications.

Consider the following example of Algorithm ??.

---

### Reduced Row-Echelon Form Algorithm

This algorithm provides a method for using row operations to take a matrix to its reduced row-echelon form. We begin with the matrix in its original form.

1. Starting from the left, find the first nonzero column. This is the first pivot column, and the position at the top of this column is the first pivot position. Switch rows if necessary to place a nonzero number in the first pivot position.
2. Use row operations to make the entries below the first pivot position (in the first pivot column) equal to zero.
3. Ignoring the row containing the first pivot position, repeat steps 1 and 2 with the remaining rows. Repeat the process until there are no more rows to modify.
4. Divide each nonzero row by the value of the leading entry, so that the leading entry becomes 1. The matrix will then be in row-echelon form.

The following step will carry the matrix from row-echelon form to reduced row-echelon form.

5. Moving from right to left, use row operations to create zeros in the entries of the pivot columns which are above the pivot positions. The result will be a matrix in reduced row-echelon form.
- 

**Example C.19: Finding Row-Echelon Form and Reduced Row-Echelon Form of a Matrix**

Let

$$A = \begin{bmatrix} 0 & -5 & -4 \\ 1 & 4 & 3 \\ 5 & 10 & 7 \end{bmatrix}$$

*Find the row-echelon form of A. Then complete the process until A is in reduced row-echelon form.*

**Solution.** In working through this example, we will use the steps outlined in Algorithm ??.

1. The first pivot column is the first column of the matrix, as this is the first nonzero column from the left. Hence the first pivot position is the one in the first row and first column. Switch the first two rows to obtain a nonzero entry in the first pivot position, outlined in a box below.

$$\left[ \begin{array}{ccc} \boxed{1} & 4 & 3 \\ 0 & -5 & -4 \\ 5 & 10 & 7 \end{array} \right]$$

2. Step two involves creating zeros in the entries below the first pivot position. The first entry of the second row is already a zero. All we need to do is subtract 5 times the first row from the third row.

The resulting matrix is

$$\begin{bmatrix} 1 & 4 & 3 \\ 0 & -5 & -4 \\ 0 & 10 & 8 \end{bmatrix}$$

3. Now ignore the top row. Apply steps 1 and 2 to the smaller matrix

$$\begin{bmatrix} -5 & -4 \\ 10 & 8 \end{bmatrix}$$

In this matrix, the first column is a pivot column, and  $-5$  is in the first pivot position. Therefore, we need to create a zero below it. To do this, add 2 times the first row (of this matrix) to the second. The resulting matrix is

$$\begin{bmatrix} -5 & -4 \\ 0 & 0 \end{bmatrix}$$

Our original matrix now looks like

$$\begin{bmatrix} 1 & 4 & 3 \\ 0 & -5 & -4 \\ 0 & 0 & 0 \end{bmatrix}$$

We can see that there are no more rows to modify.

4. Now, we need to create leading 1s in each row. The first row already has a leading 1 so no work is needed here. Divide the second row by  $-5$  to create a leading 1. The resulting matrix is

$$\begin{bmatrix} 1 & 4 & 3 \\ 0 & 1 & \frac{4}{5} \\ 0 & 0 & 0 \end{bmatrix}$$

This matrix is now in row-echelon form.

5. Now create zeros in the entries above pivot positions in each column, in order to carry this matrix all the way to reduced row-echelon form. Notice that there is no pivot position in the third column so we do not need to create any zeros in this column! The column in which we need to create zeros is the second. To do so, subtract 4 times the second row from the first row. The resulting matrix is

$$\begin{bmatrix} 1 & 0 & -\frac{1}{5} \\ 0 & 1 & \frac{4}{5} \\ 0 & 0 & 0 \end{bmatrix}$$

This matrix is now in reduced row-echelon form. ♠

The above algorithm gives you a simple way to obtain the row-echelon form and reduced row-echelon form of a matrix. The main idea is to do row operations in such a way as to end up with a matrix in row-echelon form or reduced row-echelon form. This process is important because the resulting matrix will allow you to describe the solutions to the corresponding linear system of equations in a meaningful way.

In the next example, we look at how to solve a system of equations using the corresponding augmented matrix.

### Example C.20: Finding the Solution to a System

*Give the complete solution to the following system of equations*

$$\begin{aligned} 2x + 4y - 3z &= -1 \\ 5x + 10y - 7z &= -2 \\ 3x + 6y + 5z &= 9 \end{aligned}$$

**Solution.** The augmented matrix for this system is

$$\left[ \begin{array}{ccc|c} 2 & 4 & -3 & -1 \\ 5 & 10 & -7 & -2 \\ 3 & 6 & 5 & 9 \end{array} \right]$$

In order to find the solution to this system, we wish to carry the augmented matrix to reduced row-echelon form. We will do so using Algorithm ???. Notice that the first column is nonzero, so this is our first pivot column. The first entry in the first row, 2, is the first leading entry and it is in the first pivot position. We will use row operations to create zeros in the entries below the 2. First, replace the second row with  $-5$  times the first row plus 2 times the second row. This yields

$$\left[ \begin{array}{ccc|c} 2 & 4 & -3 & -1 \\ 0 & 0 & 1 & 1 \\ 3 & 6 & 5 & 9 \end{array} \right]$$

Now, replace the third row with  $-3$  times the first row plus to 2 times the third row. This yields

$$\left[ \begin{array}{ccc|c} 2 & 4 & -3 & -1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 21 \end{array} \right]$$

Now the entries in the first column below the pivot position are zeros. We now look for the second pivot column, which in this case is column three. Here, the 1 in the second row and third column is in the pivot position. We need to do just one row operation to create a zero below the 1.

Taking  $-1$  times the second row and adding it to the third row yields

$$\left[ \begin{array}{ccc|c} 2 & 4 & -3 & -1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 20 \end{array} \right]$$

We could proceed with the algorithm to carry this matrix to row-echelon form or reduced row-echelon form. However, remember that we are looking for the solutions to the system of equations. Take another look at the third row of the matrix. Notice that it corresponds to the equation

$$0x + 0y + 0z = 20$$

There is no solution to this equation because for all  $x, y, z$ , the left side will equal 0 and  $0 \neq 20$ . This shows there is no solution to the given system of equations. In other words, this system is inconsistent. ♠

The following is another example of how to find the solution to a system of equations by carrying the corresponding augmented matrix to reduced row-echelon form.

### Example C.21: An Infinite Set of Solutions

*Give the complete solution to the system of equations*

$$\begin{aligned} 3x - y - 5z &= 9 \\ y - 10z &= 0 \\ -2x + y &= -6 \end{aligned} \tag{3.8}$$

**Solution.** The augmented matrix of this system is

$$\left[ \begin{array}{ccc|c} 3 & -1 & -5 & 9 \\ 0 & 1 & -10 & 0 \\ -2 & 1 & 0 & -6 \end{array} \right]$$

In order to find the solution to this system, we will carry the augmented matrix to reduced row-echelon form, using Algorithm ???. The first column is the first pivot column. We want to use row operations to create zeros beneath the first entry in this column, which is in the first pivot position. Replace the third row with 2 times the first row added to 3 times the third row. This gives

$$\left[ \begin{array}{ccc|c} 3 & -1 & -5 & 9 \\ 0 & 1 & -10 & 0 \\ 0 & 1 & -10 & 0 \end{array} \right]$$

Now, we have created zeros beneath the 3 in the first column, so we move on to the second pivot column (which is the second column) and repeat the procedure. Take  $-1$  times the second row and add to the third row.

$$\left[ \begin{array}{ccc|c} 3 & -1 & -5 & 9 \\ 0 & 1 & -10 & 0 \\ 0 & 0 & 0 & 0 \end{array} \right]$$

The entry below the pivot position in the second column is now a zero. Notice that we have no more pivot columns because we have only two leading entries.

At this stage, we also want the leading entries to be equal to one. To do so, divide the first row by 3.

$$\left[ \begin{array}{ccc|c} 1 & -\frac{1}{3} & -\frac{5}{3} & 3 \\ 0 & 1 & -10 & 0 \\ 0 & 0 & 0 & 0 \end{array} \right]$$

This matrix is now in row-echelon form.

Let's continue with row operations until the matrix is in reduced row-echelon form. This involves creating zeros above the pivot positions in each pivot column. This requires only one step, which is to add  $\frac{1}{3}$  times the second row to the first row.

$$\left[ \begin{array}{ccc|c} 1 & 0 & -5 & 3 \\ 0 & 1 & -10 & 0 \\ 0 & 0 & 0 & 0 \end{array} \right]$$

This is in reduced row-echelon form, which you should verify using Definition C.13. The equations corresponding to this reduced row-echelon form are

$$\begin{aligned} x - 5z &= 3 \\ y - 10z &= 0 \end{aligned}$$

or

$$\begin{aligned} x &= 3 + 5z \\ y &= 10z \end{aligned}$$

Observe that  $z$  is not restrained by any equation. In fact,  $z$  can equal any number. For example, we can let  $z = t$ , where we can choose  $t$  to be any number. In this context  $t$  is called a **parameter**. Therefore, the solution set of this system is

$$\begin{aligned} x &= 3 + 5t \\ y &= 10t \\ z &= t \end{aligned}$$

where  $t$  is arbitrary. The system has an infinite set of solutions which are given by these equations. For any value of  $t$  we select,  $x, y$ , and  $z$  will be given by the above equations. For example, if we choose  $t = 4$  then the corresponding solution would be

$$\begin{aligned} x &= 3 + 5(4) = 23 \\ y &= 10(4) = 40 \\ z &= 4 \end{aligned}$$



In Example C.21 the solution involved one parameter. It may happen that the solution to a system involves more than one parameter, as shown in the following example.

### Example C.22: A Two Parameter Set of Solutions

*Find the solution to the system*

$$\begin{aligned} x + 2y - z + w &= 3 \\ x + y - z + w &= 1 \\ x + 3y - z + w &= 5 \end{aligned}$$

**Solution.** The augmented matrix is

$$\left[ \begin{array}{cccc|c} 1 & 2 & -1 & 1 & 3 \\ 1 & 1 & -1 & 1 & 1 \\ 1 & 3 & -1 & 1 & 5 \end{array} \right]$$

We wish to carry this matrix to row-echelon form. Here, we will outline the row operations used. However, make sure that you understand the steps in terms of Algorithm ??.

Take  $-1$  times the first row and add to the second. Then take  $-1$  times the first row and add to the third. This yields

$$\left[ \begin{array}{cccc|c} 1 & 2 & -1 & 1 & 3 \\ 0 & -1 & 0 & 0 & -2 \\ 0 & 1 & 0 & 0 & 2 \end{array} \right]$$

Now add the second row to the third row and divide the second row by  $-1$ .

$$\left[ \begin{array}{cccc|c} 1 & 2 & -1 & 1 & 3 \\ 0 & 1 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 \end{array} \right] \quad (3.9)$$

This matrix is in row-echelon form and we can see that  $x$  and  $y$  correspond to pivot columns, while  $z$  and  $w$  do not. Therefore, we will assign parameters to the variables  $z$  and  $w$ . Assign the parameter  $s$  to  $z$  and the parameter  $t$  to  $w$ . Then the first row yields the equation  $x + 2y - s + t = 3$ , while the second row yields the equation  $y = 2$ . Since  $y = 2$ , the first equation becomes  $x + 4 - s + t = 3$  showing that the solution is given by

$$\begin{aligned} x &= -1 + s - t \\ y &= 2 \\ z &= s \\ w &= t \end{aligned}$$

It is customary to write this solution in the form

$$\left[ \begin{array}{c} x \\ y \\ z \\ w \end{array} \right] = \left[ \begin{array}{c} -1 + s - t \\ 2 \\ s \\ t \end{array} \right] \quad (3.10)$$



This example shows a system of equations with an infinite solution set which depends on two parameters. It can be less confusing in the case of an infinite solution set to first place the augmented matrix in reduced row-echelon form rather than just row-echelon form before seeking to write down the description of the solution.

In the above steps, this means we don't stop with the row-echelon form in equation 3.9. Instead we first place it in reduced row-echelon form as follows.

$$\left[ \begin{array}{cccc|c} 1 & 0 & -1 & 1 & -1 \\ 0 & 1 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 \end{array} \right]$$

Then the solution is  $y = 2$  from the second row and  $x = -1 + z - w$  from the first. Thus letting  $z = s$  and  $w = t$ , the solution is given by 3.10.

You can see here that there are two paths to the correct answer, which both yield the same answer. Hence, either approach may be used. The process which we first used in the above solution is called **Gaussian Elimination**. This process involves carrying the matrix to row-echelon form, converting back to equations, and using back substitution to find the solution. When you do row operations until you obtain reduced row-echelon form, the process is called **Gauss-Jordan Elimination**.

We have now found solutions for systems of equations with no solution and infinitely many solutions, with one parameter as well as two parameters. Recall the three types of solution sets which we discussed in the previous section; no solution, one solution, and infinitely many solutions. Each of these types of solutions could be identified from the graph of the system. It turns out that we can also identify the type of solution from the reduced row-echelon form of the augmented matrix.

- *No Solution:* In the case where the system of equations has no solution, the row-echelon form of the augmented matrix will have a row of the form

$$\left[ \begin{array}{ccc|c} 0 & 0 & 0 & 1 \end{array} \right]$$

This row indicates that the system is inconsistent and has no solution.

- *One Solution:* In the case where the system of equations has one solution, every column of the coefficient matrix is a pivot column. The following is an example of an augmented matrix in reduced row-echelon form for a system of equations with one solution.

$$\left[ \begin{array}{ccc|c} 1 & 0 & 0 & 5 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 2 \end{array} \right]$$

- *Infinitely Many Solutions:* In the case where the system of equations has infinitely many solutions, the solution contains parameters. There will be columns of the coefficient matrix which are not pivot columns. The following are examples of augmented matrices in reduced row-echelon form for systems of equations with infinitely many solutions.

$$\left[ \begin{array}{ccc|c} 1 & 0 & 0 & 5 \\ 0 & 1 & 2 & -3 \\ 0 & 0 & 0 & 0 \end{array} \right]$$

or

$$\left[ \begin{array}{ccc|c} 1 & 0 & 0 & 5 \\ 0 & 1 & 0 & -3 \end{array} \right]$$

### C.2.3. Uniqueness of the Reduced Row-Echelon Form

As we have seen in earlier sections, we know that every matrix can be brought into reduced row-echelon form by a sequence of elementary row operations. Here we will prove that the resulting matrix is unique; in other words, the resulting matrix in reduced row-echelon form does not depend upon the particular sequence of elementary row operations or the order in which they were performed.

Let  $A$  be the augmented matrix of a homogeneous system of linear equations in the variables  $x_1, x_2, \dots, x_n$  which is also in reduced row-echelon form. The matrix  $A$  divides the set of variables in two different types. We say that  $x_i$  is a *basic variable* whenever  $A$  has a leading 1 in column number  $i$ , in other words, when column  $i$  is a pivot column. Otherwise we say that  $x_i$  is a *free variable*.

Recall Example C.22.

#### Example C.23: Basic and Free Variables

*Find the basic and free variables in the system*

$$\begin{aligned}x + 2y - z + w &= 3 \\x + y - z + w &= 1 \\x + 3y - z + w &= 5\end{aligned}$$

**Solution.** Recall from the solution of Example C.22 that the row-echelon form of the augmented matrix of this system is given by

$$\left[ \begin{array}{cccc|c} 1 & 2 & -1 & 1 & 3 \\ 0 & 1 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 \end{array} \right]$$

You can see that columns 1 and 2 are pivot columns. These columns correspond to variables  $x$  and  $y$ , making these the basic variables. Columns 3 and 4 are not pivot columns, which means that  $z$  and  $w$  are free variables.

We can write the solution to this system as

$$\begin{aligned}x &= -1 + s - t \\y &= 2 \\z &= s \\w &= t\end{aligned}$$

Here the free variables are written as parameters, and the basic variables are given by linear functions of these parameters. ♠

In general, all solutions can be written in terms of the free variables. In such a description, the free variables can take any values (they become parameters), while the basic variables become simple linear functions of these parameters. Indeed, a basic variable  $x_i$  is a linear function of *only* those free variables  $x_j$  with  $j > i$ . This leads to the following observation.

**Proposition C.24: Basic and Free Variables**

*If  $x_i$  is a basic variable of a homogeneous system of linear equations, then any solution of the system with  $x_j = 0$  for all those free variables  $x_j$  with  $j > i$  must also have  $x_i = 0$ .*

Using this proposition, we prove a lemma which will be used in the proof of the main result of this section below.

**Lemma C.25: Solutions and the Reduced Row-Echelon Form of a Matrix**

*Let  $A$  and  $B$  be two distinct augmented matrices for two homogeneous systems of  $m$  equations in  $n$  variables, such that  $A$  and  $B$  are each in reduced row-echelon form. Then, the two systems do not have exactly the same solutions.*

Now, we say that the matrix  $B$  is **equivalent** to the matrix  $A$  provided that  $B$  can be obtained from  $A$  by performing a sequence of elementary row operations beginning with  $A$ . The importance of this concept lies in the following result.

**Theorem C.26: Equivalent Matrices**

*The two linear systems of equations corresponding to two equivalent augmented matrices have exactly the same solutions.*

The proof of this theorem is left as an exercise.

Now, we can use Lemma C.25 and Theorem C.26 to prove the main result of this section.

**Theorem C.27: Uniqueness of the Reduced Row-Echelon Form**

*Every matrix  $A$  is equivalent to a unique matrix in reduced row-echelon form.*

According to this theorem we can say that each matrix  $A$  has a unique reduced row-echelon form.

## C.2.4. Rank and Homogeneous Systems

---

There is a special type of system which requires additional study. This type of system is called a homogeneous system of equations, which we defined above in Definition C.3. Our focus in this section is to consider what types of solutions are possible for a homogeneous system of equations.

Consider the following definition.

**Definition C.28: Trivial Solution**

Consider the homogeneous system of equations given by

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= 0 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= 0 \\ &\vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &= 0 \end{aligned}$$

Then,  $x_1 = 0, x_2 = 0, \dots, x_n = 0$  is always a solution to this system. We call this the **trivial solution**.

If the system has a solution in which not all of the  $x_1, \dots, x_n$  are equal to zero, then we call this solution **nontrivial**. The trivial solution does not tell us much about the system, as it says that  $0 = 0!$  Therefore, when working with homogeneous systems of equations, we want to know when the system has a nontrivial solution.

Suppose we have a homogeneous system of  $m$  equations, using  $n$  variables, and suppose that  $n > m$ . In other words, there are more variables than equations. Then, it turns out that this system always has a nontrivial solution. Not only will the system have a nontrivial solution, but it also will have infinitely many solutions. It is also possible, but not required, to have a nontrivial solution if  $n = m$  and  $n < m$ .

Consider the following example.

**Example C.29: Solutions to a Homogeneous System of Equations**

Find the nontrivial solutions to the following homogeneous system of equations

$$\begin{aligned} 2x + y - z &= 0 \\ x + 2y - 2z &= 0 \end{aligned}$$

**Solution.** Notice that this system has  $m = 2$  equations and  $n = 3$  variables, so  $n > m$ . Therefore by our previous discussion, we expect this system to have infinitely many solutions.

The process we use to find the solutions for a homogeneous system of equations is the same process we used in the previous section. First, we construct the augmented matrix, given by

$$\left[ \begin{array}{ccc|c} 2 & 1 & -1 & 0 \\ 1 & 2 & -2 & 0 \end{array} \right]$$

Then, we carry this matrix to its reduced row-echelon form, given below.

$$\left[ \begin{array}{ccc|c} 1 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 \end{array} \right]$$

The corresponding system of equations is

$$\begin{aligned} x &= 0 \\ y - z &= 0 \end{aligned}$$

Since  $z$  is not restrained by any equation, we know that this variable will become our parameter. Let  $z = t$  where  $t$  is any number. Therefore, our solution has the form

$$\begin{aligned}x &= 0 \\y &= z = t \\z &= t\end{aligned}$$

Hence this system has infinitely many solutions, with one parameter  $t$ . ♠

Suppose we were to write the solution to the previous example in another form. Specifically,

$$\begin{aligned}x &= 0 \\y &= 0 + t \\z &= 0 + t\end{aligned}$$

can be written as

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} + t \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}$$

Notice that we have constructed a column from the constants in the solution (all equal to 0), as well as a column corresponding to the coefficients on  $t$  in each equation. While we will discuss this form of solution more in further chapters, for now consider the column of coefficients of the parameter  $t$ . In this case, this is the column  $\begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}$ .

There is a special name for this column, which is **basic solution**. The basic solutions of a system are columns constructed from the coefficients on parameters in the solution. We often denote basic solutions by  $X_1, X_2$  etc., depending on how many solutions occur. Therefore, Example C.29 has the basic solution

$$X_1 = \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}.$$

We explore this further in the following example.

### Example C.30: Basic Solutions of a Homogeneous System

Consider the following homogeneous system of equations.

$$\begin{aligned}x + 4y + 3z &= 0 \\3x + 12y + 9z &= 0\end{aligned}$$

Find the basic solutions to this system.

**Solution.** The augmented matrix of this system and the resulting reduced row-echelon form are

$$\left[ \begin{array}{ccc|c} 1 & 4 & 3 & 0 \\ 3 & 12 & 9 & 0 \end{array} \right] \rightarrow \dots \rightarrow \left[ \begin{array}{ccc|c} 1 & 4 & 3 & 0 \\ 0 & 0 & 0 & 0 \end{array} \right]$$

When written in equations, this system is given by

$$x + 4y + 3z = 0$$

Notice that only  $x$  corresponds to a pivot column. In this case, we will have two parameters, one for  $y$  and one for  $z$ . Let  $y = s$  and  $z = t$  for any numbers  $s$  and  $t$ . Then, our solution becomes

$$x = -4s - 3t$$

$$y = s$$

$$z = t$$

which can be written as

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} + s \begin{bmatrix} -4 \\ 1 \\ 0 \end{bmatrix} + t \begin{bmatrix} -3 \\ 0 \\ 1 \end{bmatrix}$$

You can see here that we have two columns of coefficients corresponding to parameters, specifically one for  $s$  and one for  $t$ . Therefore, this system has two basic solutions! These are

$$X_1 = \begin{bmatrix} -4 \\ 1 \\ 0 \end{bmatrix}, X_2 = \begin{bmatrix} -3 \\ 0 \\ 1 \end{bmatrix}$$



We now present a new definition.

### Definition C.31: Linear Combination

Let  $X_1, \dots, X_n, V$  be column matrices. Then  $V$  is said to be a **linear combination** of the columns  $X_1, \dots, X_n$  if there exist scalars,  $a_1, \dots, a_n$  such that

$$V = a_1X_1 + \dots + a_nX_n$$

A remarkable result of this section is that a linear combination of the basic solutions is again a solution to the system. Even more remarkable is that every solution can be written as a linear combination of these solutions. Therefore, if we take a linear combination of the two solutions to Example C.30, this would also be a solution. For example, we could take the following linear combination

$$3 \begin{bmatrix} -4 \\ 1 \\ 0 \end{bmatrix} + 2 \begin{bmatrix} -3 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} -18 \\ 3 \\ 2 \end{bmatrix}$$

You should take a moment to verify that

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} -18 \\ 3 \\ 2 \end{bmatrix}$$

is in fact a solution to the system in Example C.30.

Another way in which we can find out more information about the solutions of a homogeneous system is to consider the **rank** of the associated coefficient matrix. We now define what is meant by the rank of a matrix.

### Definition C.32: Rank of a Matrix

*Let  $A$  be a matrix and consider any row-echelon form of  $A$ . Then, the number  $r$  of leading entries of  $A$  does not depend on the row-echelon form you choose, and is called the **rank** of  $A$ . We denote it by  $\text{rank}(A)$ .*

Similarly, we could count the number of pivot positions (or pivot columns) to determine the rank of  $A$ .

### Example C.33: Finding the Rank of a Matrix

Consider the matrix

$$\begin{bmatrix} 1 & 2 & 3 \\ 1 & 5 & 9 \\ 2 & 4 & 6 \end{bmatrix}$$

What is its rank?

**Solution.** First, we need to find the reduced row-echelon form of  $A$ . Through the usual algorithm, we find that this is

$$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & 2 \\ 0 & 0 & 0 \end{bmatrix}$$

Here we have two leading entries, or two pivot positions, shown above in boxes. The rank of  $A$  is  $r = 2$ .



Notice that we would have achieved the same answer if we had found the row-echelon form of  $A$  instead of the reduced row-echelon form.

Suppose we have a homogeneous system of  $m$  equations in  $n$  variables, and suppose that  $n > m$ . From our above discussion, we know that this system will have infinitely many solutions. If we consider the rank of the coefficient matrix of this system, we can find out even more about the solution. Note that we are looking at just the coefficient matrix, not the entire augmented matrix.

### Theorem C.34: Rank and Solutions to a Homogeneous System

*Let  $A$  be the  $m \times n$  coefficient matrix corresponding to a homogeneous system of equations, and suppose  $A$  has rank  $r$ . Then, the solution to the corresponding system has  $n - r$  parameters.*

Consider our above Example C.30 in the context of this theorem. The system in this example has  $m = 2$  equations in  $n = 3$  variables. First, because  $n > m$ , we know that the system has a nontrivial solution, and therefore infinitely many solutions. This tells us that the solution will contain at least one parameter. The

rank of the coefficient matrix can tell us even more about the solution! The rank of the coefficient matrix of the system is 1, as it has one leading entry in row-echelon form. Theorem C.34 tells us that the solution will have  $n - r = 3 - 1 = 2$  parameters. You can check that this is true in the solution to Example C.30.

Notice that if  $n = m$  or  $n < m$ , it is possible to have either a unique solution (which will be the trivial solution) or infinitely many solutions.

We are not limited to homogeneous systems of equations here. The rank of a matrix can be used to learn about the solutions of any system of linear equations. In the previous section, we discussed that a system of equations can have no solution, a unique solution, or infinitely many solutions. Suppose the system is consistent, whether it is homogeneous or not. The following theorem tells us how we can use the rank to learn about the type of solution we have.

### Theorem C.35: Rank and Solutions to a Consistent System of Equations

*Let  $A$  be the  $m \times (n + 1)$  augmented matrix corresponding to a consistent system of equations in  $n$  variables, and suppose  $A$  has rank  $r$ . Then*

1. *the system has a unique solution if  $r = n$*
2. *the system has infinitely many solutions if  $r < n$*

We will not present a formal proof of this, but consider the following discussions.

1. *No Solution* The above theorem assumes that the system is consistent, that is, that it has a solution. It turns out that it is possible for the augmented matrix of a system with no solution to have any rank  $r$  as long as  $r > 1$ . Therefore, we must know that the system is consistent in order to use this theorem!
2. *Unique Solution* Suppose  $r = n$ . Then, there is a pivot position in every column of the coefficient matrix of  $A$ . Hence, there is a unique solution.
3. *Infinitely Many Solutions* Suppose  $r < n$ . Then there are infinitely many solutions. There are less pivot positions (and hence less leading entries) than columns, meaning that not every column is a pivot column. The columns which are *not* pivot columns correspond to parameters. In fact, in this case we have  $n - r$  parameters.



# D. Matrices

## D.1 Matrix Arithmetic

### Outcomes

- A. Perform the matrix operations of matrix addition, scalar multiplication, transposition and matrix multiplication. Identify when these operations are not defined. Represent these operations in terms of the entries of a matrix.
- B. Prove algebraic properties for matrix addition, scalar multiplication, transposition, and matrix multiplication. Apply these properties to manipulate an algebraic expression involving matrices.
- C. Compute the inverse of a matrix using row operations, and prove identities involving matrix inverses.
- E. Solve a linear system using matrix algebra.
- F. Use multiplication by an elementary matrix to apply row operations.
- G. Write a matrix as a product of elementary matrices.

You have now solved systems of equations by writing them in terms of an augmented matrix and then doing row operations on this augmented matrix. It turns out that matrices are important not only for systems of equations but also in many applications.

Recall that a **matrix** is a rectangular array of numbers. Several of them are referred to as **matrices**. For example, here is a matrix.

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 2 & 8 & 7 \\ 6 & -9 & 1 & 2 \end{bmatrix} \quad (4.1)$$

Recall that the size or dimension of a matrix is defined as  $m \times n$  where  $m$  is the number of rows and  $n$  is the number of columns. The above matrix is a  $3 \times 4$  matrix because there are three rows and four columns. You can remember the columns are like columns in a Greek temple. They stand upright while the rows lay flat like rows made by a tractor in a plowed field.

When specifying the size of a matrix, you always list the number of rows before the number of columns. You might remember that you always list the rows before the columns by using the phrase **Rowman Catholic**.

Consider the following definition.

### Definition D.1: Square Matrix

A matrix  $A$  which has size  $n \times n$  is called a **square matrix**. In other words,  $A$  is a square matrix if it has the same number of rows and columns.

There is some notation specific to matrices which we now introduce. We denote the columns of a matrix  $A$  by  $A_j$  as follows

$$A = [ A_1 \ A_2 \ \cdots \ A_n ]$$

Therefore,  $A_j$  is the  $j^{\text{th}}$  column of  $A$ , when counted from left to right.

The individual elements of the matrix are called **entries** or **components** of  $A$ . Elements of the matrix are identified according to their position. The **(i,j)-entry** of a matrix is the entry in the  $i^{\text{th}}$  row and  $j^{\text{th}}$  column. For example, in the matrix 4.1 above, 8 is in position  $(2,3)$  (and is called the  $(2,3)$ -entry) because it is in the second row and the third column.

In order to remember which matrix we are speaking of, we will denote the entry in the  $i^{\text{th}}$  row and the  $j^{\text{th}}$  column of matrix  $A$  by  $a_{ij}$ . Then, we can write  $A$  in terms of its entries, as  $A = [a_{ij}]$ . Using this notation on the matrix in 4.1,  $a_{23} = 8, a_{32} = -9, a_{12} = 2$ , etc.

There are various operations which are done on matrices of appropriate sizes. Matrices can be added to and subtracted from other matrices, multiplied by a scalar, and multiplied by other matrices. We will never divide a matrix by another matrix, but we will see later how matrix inverses play a similar role.

In doing arithmetic with matrices, we often define the action by what happens in terms of the entries (or components) of the matrices. Before looking at these operations in depth, consider a few general definitions.

### Definition D.2: The Zero Matrix

The  $m \times n$  **zero matrix** is the  $m \times n$  matrix having every entry equal to zero. It is denoted by 0.

One possible zero matrix is shown in the following example.

### Example D.3: The Zero Matrix

The  $2 \times 3$  zero matrix is  $0 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$ .

Note there is a  $2 \times 3$  zero matrix, a  $3 \times 4$  zero matrix, etc. In fact there is a zero matrix for every size!

### Definition D.4: Equality of Matrices

Let  $A$  and  $B$  be two  $m \times n$  matrices. Then  $A = B$  means that for  $A = [a_{ij}]$  and  $B = [b_{ij}]$ ,  $a_{ij} = b_{ij}$  for all  $1 \leq i \leq m$  and  $1 \leq j \leq n$ .

In other words, two matrices are equal exactly when they are the same size and the corresponding entries are identical. Thus

$$\begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \neq \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

because they are different sizes. Also,

$$\begin{bmatrix} 0 & 1 \\ 3 & 2 \end{bmatrix} \neq \begin{bmatrix} 1 & 0 \\ 2 & 3 \end{bmatrix}$$

because, although they are the same size, their corresponding entries are not identical.

In the following section, we explore addition of matrices.

### D.1.1. Addition of Matrices

---

When adding matrices, all matrices in the sum need have the same size. For example,

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 2 \end{bmatrix}$$

and

$$\begin{bmatrix} -1 & 4 & 8 \\ 2 & 8 & 5 \end{bmatrix}$$

cannot be added, as one has size  $3 \times 2$  while the other has size  $2 \times 3$ .

However, the addition

$$\begin{bmatrix} 4 & 6 & 3 \\ 5 & 0 & 4 \\ 11 & -2 & 3 \end{bmatrix} + \begin{bmatrix} 0 & 5 & 0 \\ 4 & -4 & 14 \\ 1 & 2 & 6 \end{bmatrix}$$

is possible.

The formal definition is as follows.

#### Definition D.5: Addition of Matrices

Let  $A = [a_{ij}]$  and  $B = [b_{ij}]$  be two  $m \times n$  matrices. Then  $A + B = C$  where  $C$  is the  $m \times n$  matrix  $C = [c_{ij}]$  defined by

$$c_{ij} = a_{ij} + b_{ij}$$

This definition tells us that when adding matrices, we simply add corresponding entries of the matrices. This is demonstrated in the next example.

**Example D.6: Addition of Matrices of Same Size**

Add the following matrices, if possible.

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 0 & 4 \end{bmatrix}, B = \begin{bmatrix} 5 & 2 & 3 \\ -6 & 2 & 1 \end{bmatrix}$$

**Solution.** Notice that both  $A$  and  $B$  are of size  $2 \times 3$ . Since  $A$  and  $B$  are of the same size, the addition is possible. Using Definition D.5, the addition is done as follows.

$$A + B = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 0 & 4 \end{bmatrix} + \begin{bmatrix} 5 & 2 & 3 \\ -6 & 2 & 1 \end{bmatrix} = \begin{bmatrix} 1+5 & 2+2 & 3+3 \\ 1+(-6) & 0+2 & 4+1 \end{bmatrix} = \begin{bmatrix} 6 & 4 & 6 \\ -5 & 2 & 5 \end{bmatrix}$$



Addition of matrices obeys very much the same properties as normal addition with numbers. Note that when we write for example  $A + B$  then we assume that both matrices are of equal size so that the operation is indeed possible.

**Proposition D.7: Properties of Matrix Addition**

Let  $A, B$  and  $C$  be matrices. Then, the following properties hold.

- Commutative Law of Addition

$$A + B = B + A \quad (4.2)$$

- Associative Law of Addition

$$(A + B) + C = A + (B + C) \quad (4.3)$$

- Existence of an Additive Identity

There exists a zero matrix  $0$  such that  
 $A + 0 = A$  (4.4)

- Existence of an Additive Inverse

There exists a matrix  $-A$  such that  
 $A + (-A) = 0$  (4.5)

We call the zero matrix in 4.4 the **additive identity**. Similarly, we call the matrix  $-A$  in 4.5 the **additive inverse**.  $-A$  is defined to equal  $(-1)A = [-a_{ij}]$ . In other words, every entry of  $A$  is multiplied by  $-1$ . In the next section we will study scalar multiplication in more depth to understand what is meant by  $(-1)A$ .

## D.1.2. Scalar Multiplication of Matrices

Recall that we use the word *scalar* when referring to numbers. Therefore, *scalar multiplication of a matrix* is the multiplication of a matrix by a number. To illustrate this concept, consider the following example in which a matrix is multiplied by the scalar 3.

$$3 \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 2 & 8 & 7 \\ 6 & -9 & 1 & 2 \end{bmatrix} = \begin{bmatrix} 3 & 6 & 9 & 12 \\ 15 & 6 & 24 & 21 \\ 18 & -27 & 3 & 6 \end{bmatrix}$$

The new matrix is obtained by multiplying every entry of the original matrix by the given scalar.  
The formal definition of scalar multiplication is as follows.

### Definition D.8: Scalar Multiplication of Matrices

If  $A = [a_{ij}]$  and  $k$  is a scalar, then  $kA = [ka_{ij}]$ .

Consider the following example.

### Example D.9: Effect of Multiplication by a Scalar

Find the result of multiplying the following matrix  $A$  by 7.

$$A = \begin{bmatrix} 2 & 0 \\ 1 & -4 \end{bmatrix}$$

**Solution.** By Definition D.8, we multiply each element of  $A$  by 7. Therefore,

$$7A = 7 \begin{bmatrix} 2 & 0 \\ 1 & -4 \end{bmatrix} = \begin{bmatrix} 7(2) & 7(0) \\ 7(1) & 7(-4) \end{bmatrix} = \begin{bmatrix} 14 & 0 \\ 7 & -28 \end{bmatrix}$$



Similarly to addition of matrices, there are several properties of scalar multiplication which hold.

**Proposition D.10: Properties of Scalar Multiplication**

Let  $A, B$  be matrices, and  $k, p$  be scalars. Then, the following properties hold.

- Distributive Law over Matrix Addition

$$k(A + B) = kA + kB$$

- Distributive Law over Scalar Addition

$$(k + p)A = kA + pA$$

- Associative Law for Scalar Multiplication

$$k(pA) = (kp)A$$

- Rule for Multiplication by 1

$$1A = A$$

The proof of this proposition is similar to the proof of Proposition D.7 and is left an exercise to the reader.

### D.1.3. Multiplication of Matrices

The next important matrix operation we will explore is multiplication of matrices. The operation of matrix multiplication is one of the most important and useful of the matrix operations. Throughout this section, we will also demonstrate how matrix multiplication relates to linear systems of equations.

First, we provide a formal definition of row and column vectors.

**Definition D.11: Row and Column Vectors**

Matrices of size  $n \times 1$  or  $1 \times n$  are called **vectors**. If  $X$  is such a matrix, then we write  $x_i$  to denote the entry of  $X$  in the  $i^{\text{th}}$  row of a column matrix, or the  $i^{\text{th}}$  column of a row matrix.

The  $n \times 1$  matrix

$$X = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$$

is called a **column vector**. The  $1 \times n$  matrix

$$X = [x_1 \ \cdots \ x_n]$$

is called a **row vector**.

We may simply use the term **vector** throughout this text to refer to either a column or row vector. If

we do so, the context will make it clear which we are referring to.

In this chapter, we will again use the notion of linear combination of vectors as in Definition F.7. In this context, a linear combination is a sum consisting of vectors multiplied by scalars. For example,

$$\begin{bmatrix} 50 \\ 122 \end{bmatrix} = 7 \begin{bmatrix} 1 \\ 4 \end{bmatrix} + 8 \begin{bmatrix} 2 \\ 5 \end{bmatrix} + 9 \begin{bmatrix} 3 \\ 6 \end{bmatrix}$$

is a linear combination of three vectors.

It turns out that we can express any system of linear equations as a linear combination of vectors. In fact, the vectors that we will use are just the columns of the corresponding augmented matrix!

### Definition D.12: The Vector Form of a System of Linear Equations

Suppose we have a system of equations given by

$$\begin{aligned} a_{11}x_1 + \cdots + a_{1n}x_n &= b_1 \\ &\vdots \\ a_{m1}x_1 + \cdots + a_{mn}x_n &= b_m \end{aligned}$$

We can express this system in **vector form** which is as follows:

$$x_1 \begin{bmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{m1} \end{bmatrix} + x_2 \begin{bmatrix} a_{12} \\ a_{22} \\ \vdots \\ a_{m2} \end{bmatrix} + \cdots + x_n \begin{bmatrix} a_{1n} \\ a_{2n} \\ \vdots \\ a_{nn} \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

Notice that each vector used here is one column from the corresponding augmented matrix. There is one vector for each variable in the system, along with the constant vector.

The first important form of matrix multiplication is multiplying a matrix by a vector. Consider the product given by

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \begin{bmatrix} 7 \\ 8 \\ 9 \end{bmatrix}$$

We will soon see that this equals

$$7 \begin{bmatrix} 1 \\ 4 \end{bmatrix} + 8 \begin{bmatrix} 2 \\ 5 \end{bmatrix} + 9 \begin{bmatrix} 3 \\ 6 \end{bmatrix} = \begin{bmatrix} 50 \\ 122 \end{bmatrix}$$

In general terms,

$$\begin{aligned} \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} &= x_1 \begin{bmatrix} a_{11} \\ a_{21} \end{bmatrix} + x_2 \begin{bmatrix} a_{12} \\ a_{22} \end{bmatrix} + x_3 \begin{bmatrix} a_{13} \\ a_{23} \end{bmatrix} \\ &= \begin{bmatrix} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 \end{bmatrix} \end{aligned}$$

Thus you take  $x_1$  times the first column, add to  $x_2$  times the second column, and finally  $x_3$  times the third column. The above sum is a linear combination of the columns of the matrix. When you multiply a matrix on the left by a vector on the right, the numbers making up the vector are just the scalars to be used in the linear combination of the columns as illustrated above.

Here is the formal definition of how to multiply an  $m \times n$  matrix by an  $n \times 1$  column vector.

### Definition D.13: Multiplication of Vector by Matrix

Let  $A = [a_{ij}]$  be an  $m \times n$  matrix and let  $X$  be an  $n \times 1$  matrix given by

$$A = [A_1 \cdots A_n], X = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$$

Then the product  $AX$  is the  $m \times 1$  column vector which equals the following linear combination of the columns of  $A$ :

$$x_1 A_1 + x_2 A_2 + \cdots + x_n A_n = \sum_{j=1}^n x_j A_j$$

If we write the columns of  $A$  in terms of their entries, they are of the form

$$A_j = \begin{bmatrix} a_{1j} \\ a_{2j} \\ \vdots \\ a_{mj} \end{bmatrix}$$

Then, we can write the product  $AX$  as

$$AX = x_1 \begin{bmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{m1} \end{bmatrix} + x_2 \begin{bmatrix} a_{12} \\ a_{22} \\ \vdots \\ a_{m2} \end{bmatrix} + \cdots + x_n \begin{bmatrix} a_{1n} \\ a_{2n} \\ \vdots \\ a_{mn} \end{bmatrix}$$

Note that multiplication of an  $m \times n$  matrix and an  $n \times 1$  vector produces an  $m \times 1$  vector.

Here is an example.

### Example D.14: A Vector Multiplied by a Matrix

Compute the product  $AX$  for

$$A = \begin{bmatrix} 1 & 2 & 1 & 3 \\ 0 & 2 & 1 & -2 \\ 2 & 1 & 4 & 1 \end{bmatrix}, X = \begin{bmatrix} 1 \\ 2 \\ 0 \\ 1 \end{bmatrix}$$

**Solution.** We will use Definition D.13 to compute the product. Therefore, we compute the product  $AX$  as follows.

$$\begin{aligned} & 1 \begin{bmatrix} 1 \\ 0 \\ 2 \end{bmatrix} + 2 \begin{bmatrix} 2 \\ 2 \\ 1 \end{bmatrix} + 0 \begin{bmatrix} 1 \\ 1 \\ 4 \end{bmatrix} + 1 \begin{bmatrix} -2 \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} 1 \\ 0 \\ 2 \end{bmatrix} + \begin{bmatrix} 4 \\ 4 \\ 2 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 3 \\ -2 \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} 8 \\ 2 \\ 5 \end{bmatrix} \end{aligned}$$



Using the above operation, we can also write a system of linear equations in **matrix form**. In this form, we express the system as a matrix multiplied by a vector. Consider the following definition.

### Definition D.15: The Matrix Form of a System of Linear Equations

Suppose we have a system of equations given by

$$\begin{aligned} a_{11}x_1 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + \cdots + a_{2n}x_n &= b_2 \\ &\vdots \\ a_{m1}x_1 + \cdots + a_{mn}x_n &= b_m \end{aligned}$$

Then we can express this system in **matrix form** as follows.

$$\left[ \begin{array}{cccc} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{array} \right] \left[ \begin{array}{c} x_1 \\ x_2 \\ \vdots \\ x_n \end{array} \right] = \left[ \begin{array}{c} b_1 \\ b_2 \\ \vdots \\ b_m \end{array} \right]$$

The expression  $AX = B$  is also known as the **Matrix Form** of the corresponding system of linear equations. The matrix  $A$  is simply the coefficient matrix of the system, the vector  $X$  is the column vector constructed from the variables of the system, and finally the vector  $B$  is the column vector constructed from the constants of the system. It is important to note that any system of linear equations can be written in this form.

Notice that if we write a homogeneous system of equations in matrix form, it would have the form  $AX = 0$ , for the zero vector  $0$ .

You can see from this definition that a vector

$$X = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

will satisfy the equation  $AX = B$  only when the entries  $x_1, x_2, \dots, x_n$  of the vector  $X$  are solutions to the original system.

Now that we have examined how to multiply a matrix by a vector, we wish to consider the case where we multiply two matrices of more general sizes, although these sizes still need to be appropriate as we will see. For example, in Example D.14, we multiplied a  $3 \times 4$  matrix by a  $4 \times 1$  vector. We want to investigate how to multiply other sizes of matrices.

We have not yet given any conditions on when matrix multiplication is possible! For matrices  $A$  and  $B$ , in order to form the product  $AB$ , the number of columns of  $A$  must equal the number of rows of  $B$ . Consider a product  $AB$  where  $A$  has size  $m \times n$  and  $B$  has size  $n \times p$ . Then, the product in terms of size of matrices is given by

$$(m \times \overbrace{n}^{\text{these must match!}}) (\overbrace{n \times p}^{}) = m \times p$$

Note the two outside numbers give the size of the product. One of the most important rules regarding matrix multiplication is the following. If the two middle numbers don't match, you can't multiply the matrices!

When the number of columns of  $A$  equals the number of rows of  $B$  the two matrices are said to be **conformable** and the product  $AB$  is obtained as follows.

### Definition D.16: Multiplication of Two Matrices

Let  $A$  be an  $m \times n$  matrix and let  $B$  be an  $n \times p$  matrix of the form

$$B = [B_1 \cdots B_p]$$

where  $B_1, \dots, B_p$  are the  $n \times 1$  columns of  $B$ . Then the  $m \times p$  matrix  $AB$  is defined as follows:

$$AB = A[B_1 \cdots B_p] = [(AB)_1 \cdots (AB)_p]$$

where  $(AB)_k$  is an  $m \times 1$  matrix or column vector which gives the  $k^{\text{th}}$  column of  $AB$ .

Consider the following example.

### Example D.17: Multiplying Two Matrices

Find  $AB$  if possible.

$$A = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 2 & 1 \end{bmatrix}, B = \begin{bmatrix} 1 & 2 & 0 \\ 0 & 3 & 1 \\ -2 & 1 & 1 \end{bmatrix}$$

**Solution.** The first thing you need to verify when calculating a product is whether the multiplication is possible. The first matrix has size  $2 \times 3$  and the second matrix has size  $3 \times 3$ . The inside numbers are equal, so  $A$  and  $B$  are conformable matrices. According to the above discussion  $AB$  will be a  $2 \times 3$  matrix. Definition D.16 gives us a way to calculate each column of  $AB$ , as follows.

$$\left[ \underbrace{\begin{bmatrix} 1 & 2 & 1 \\ 0 & 2 & 1 \end{bmatrix}}_{\text{First column}} \begin{bmatrix} 1 \\ 0 \\ -2 \end{bmatrix}, \underbrace{\begin{bmatrix} 1 & 2 & 1 \\ 0 & 2 & 1 \end{bmatrix}}_{\text{Second column}} \begin{bmatrix} 2 \\ 3 \\ 1 \end{bmatrix}, \underbrace{\begin{bmatrix} 1 & 2 & 1 \\ 0 & 2 & 1 \end{bmatrix}}_{\text{Third column}} \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} \right]$$

You know how to multiply a matrix times a vector, using Definition D.13 for each of the three columns. Thus

$$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 2 & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & 0 \\ 0 & 3 & 1 \\ -2 & 1 & 1 \end{bmatrix} = \begin{bmatrix} -1 & 9 & 3 \\ -2 & 7 & 3 \end{bmatrix}$$



Since vectors are simply  $n \times 1$  or  $1 \times m$  matrices, we can also multiply a vector by another vector.

### Example D.18: Vector Times Vector Multiplication

Multiply if possible  $\begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} [1 \ 2 \ 1 \ 0]$ .

**Solution.** In this case we are multiplying a matrix of size  $3 \times 1$  by a matrix of size  $1 \times 4$ . The inside numbers match so the product is defined. Note that the product will be a matrix of size  $3 \times 4$ . Using Definition D.16, we can compute this product as follows

$$\begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} [1 \ 2 \ 1 \ 0] = \left[ \underbrace{\begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}}_{\text{First column}} [1], \underbrace{\begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}}_{\text{Second column}} [2], \underbrace{\begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}}_{\text{Third column}} [1], \underbrace{\begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}}_{\text{Fourth column}} [0] \right]$$

You can use Definition D.13 to verify that this product is

$$\begin{bmatrix} 1 & 2 & 1 & 0 \\ 2 & 4 & 2 & 0 \\ 1 & 2 & 1 & 0 \end{bmatrix}$$



**Example D.19: A Multiplication Which is Not Defined**

*Find  $BA$  if possible.*

$$B = \begin{bmatrix} 1 & 2 & 0 \\ 0 & 3 & 1 \\ -2 & 1 & 1 \end{bmatrix}, A = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 2 & 1 \end{bmatrix}$$

**Solution.** First check if it is possible. This product is of the form  $(3 \times 3)(2 \times 3)$ . The inside numbers do not match and so you can't do this multiplication. ♠

In this case, we say that the multiplication is not defined. Notice that these are the same matrices which we used in Example D.17. In this example, we tried to calculate  $BA$  instead of  $AB$ . This demonstrates another property of matrix multiplication. While the product  $AB$  maybe be defined, we cannot assume that the product  $BA$  will be possible. Therefore, it is important to always check that the product is defined before carrying out any calculations.

Earlier, we defined the zero matrix  $0$  to be the matrix (of appropriate size) containing zeros in all entries. Consider the following example for multiplication by the zero matrix.

**Example D.20: Multiplication by the Zero Matrix**

*Compute the product  $A0$  for the matrix*

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

*and the  $2 \times 2$  zero matrix given by*

$$0 = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

**Solution.** In this product, we compute

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

Hence,  $A0 = 0$ . ♠

Notice that we could also multiply  $A$  by the  $2 \times 1$  zero vector given by  $\begin{bmatrix} 0 \\ 0 \end{bmatrix}$ . The result would be the  $2 \times 1$  zero vector. Therefore, it is always the case that  $A0 = 0$ , for an appropriately sized zero matrix or vector.

### D.1.4. The $i j^{th}$ Entry of a Product

---

In previous sections, we used the entries of a matrix to describe the action of matrix addition and scalar multiplication. We can also study matrix multiplication using the entries of matrices.

What is the  $i j^{th}$  entry of  $AB$ ? It is the entry in the  $i^{th}$  row and the  $j^{th}$  column of the product  $AB$ .

Now if  $A$  is  $m \times n$  and  $B$  is  $n \times p$ , then we know that the product  $AB$  has the form

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1j} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2j} & \cdots & b_{2p} \\ \vdots & \vdots & & \vdots & & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nj} & \cdots & b_{np} \end{bmatrix}$$

The  $j^{th}$  column of  $AB$  is of the form

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} b_{1j} \\ b_{2j} \\ \vdots \\ b_{nj} \end{bmatrix}$$

which is an  $m \times 1$  column vector. It is calculated by

$$b_{1j} \begin{bmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{m1} \end{bmatrix} + b_{2j} \begin{bmatrix} a_{12} \\ a_{22} \\ \vdots \\ a_{m2} \end{bmatrix} + \cdots + b_{nj} \begin{bmatrix} a_{1n} \\ a_{2n} \\ \vdots \\ a_{mn} \end{bmatrix}$$

Therefore, the  $i j^{th}$  entry is the entry in row  $i$  of this vector. This is computed by

$$a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^n a_{ik}b_{kj}$$

The following is the formal definition for the  $i j^{th}$  entry of a product of matrices.

**Definition D.21: The  $ij^{\text{th}}$  Entry of a Product**

Let  $A = [a_{ij}]$  be an  $m \times n$  matrix and let  $B = [b_{ij}]$  be an  $n \times p$  matrix. Then  $AB$  is an  $m \times p$  matrix and the  $(i, j)$ -entry of  $AB$  is defined as

$$(AB)_{ij} = \sum_{k=1}^n a_{ik}b_{kj}$$

Another way to write this is

$$(AB)_{ij} = [a_{i1} \ a_{i2} \ \cdots \ a_{in}] \begin{bmatrix} b_{1j} \\ b_{2j} \\ \vdots \\ b_{nj} \end{bmatrix} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj}$$

In other words, to find the  $(i, j)$ -entry of the product  $AB$ , or  $(AB)_{ij}$ , you multiply the  $i^{\text{th}}$  row of  $A$ , on the left by the  $j^{\text{th}}$  column of  $B$ . To express  $AB$  in terms of its entries, we write  $AB = [(AB)_{ij}]$ .

Consider the following example.

**Example D.22: The Entries of a Product**

Compute  $AB$  if possible. If it is, find the  $(3, 2)$ -entry of  $AB$  using Definition D.21.

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 1 \\ 2 & 6 \end{bmatrix}, B = \begin{bmatrix} 2 & 3 & 1 \\ 7 & 6 & 2 \end{bmatrix}$$

**Solution.** First check if the product is possible. It is of the form  $(3 \times 2)(2 \times 3)$  and since the inside numbers match, it is possible to do the multiplication. The result should be a  $3 \times 3$  matrix. We can first compute  $AB$ :

$$\left[ \begin{bmatrix} 1 & 2 \\ 3 & 1 \\ 2 & 6 \end{bmatrix} \begin{bmatrix} 2 \\ 7 \end{bmatrix}, \begin{bmatrix} 1 & 2 \\ 3 & 1 \\ 2 & 6 \end{bmatrix} \begin{bmatrix} 3 \\ 6 \end{bmatrix}, \begin{bmatrix} 1 & 2 \\ 3 & 1 \\ 2 & 6 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix} \right]$$

where the commas separate the columns in the resulting product. Thus the above product equals

$$\begin{bmatrix} 16 & 15 & 5 \\ 13 & 15 & 5 \\ 46 & 42 & 14 \end{bmatrix}$$

which is a  $3 \times 3$  matrix as desired. Thus, the  $(3, 2)$ -entry equals 42.

Now using Definition D.21, we can find that the  $(3,2)$ -entry equals

$$\begin{aligned}\sum_{k=1}^2 a_{3k}b_{k2} &= a_{31}b_{12} + a_{32}b_{22} \\ &= 2 \times 3 + 6 \times 6 = 42\end{aligned}$$

Consulting our result for  $AB$  above, this is correct!

You may wish to use this method to verify that the rest of the entries in  $AB$  are correct. ♠

Here is another example.

### Example D.23: Finding the Entries of a Product

Determine if the product  $AB$  is defined. If it is, find the  $(2,1)$ -entry of the product.

$$A = \begin{bmatrix} 2 & 3 & 1 \\ 7 & 6 & 2 \\ 0 & 0 & 0 \end{bmatrix}, B = \begin{bmatrix} 1 & 2 \\ 3 & 1 \\ 2 & 6 \end{bmatrix}$$

**Solution.** This product is of the form  $(3 \times 3)(3 \times 2)$ . The middle numbers match so the matrices are conformable and it is possible to compute the product.

We want to find the  $(2,1)$ -entry of  $AB$ , that is, the entry in the second row and first column of the product. We will use Definition D.21, which states

$$(AB)_{ij} = \sum_{k=1}^n a_{ik}b_{kj}$$

In this case,  $n = 3$ ,  $i = 2$  and  $j = 1$ . Hence the  $(2,1)$ -entry is found by computing

$$(AB)_{21} = \sum_{k=1}^3 a_{2k}b_{k1} = [ a_{21} \ a_{22} \ a_{23} ] \begin{bmatrix} b_{11} \\ b_{21} \\ b_{31} \end{bmatrix}$$

Substituting in the appropriate values, this product becomes

$$[ a_{21} \ a_{22} \ a_{23} ] \begin{bmatrix} b_{11} \\ b_{21} \\ b_{31} \end{bmatrix} = [ 7 \ 6 \ 2 ] \begin{bmatrix} 1 \\ 3 \\ 2 \end{bmatrix} = 1 \times 7 + 3 \times 6 + 2 \times 2 = 29$$

Hence,  $(AB)_{21} = 29$ .

You should take a moment to find a few other entries of  $AB$ . You can multiply the matrices to check that your answers are correct. The product  $AB$  is given by

$$AB = \begin{bmatrix} 13 & 13 \\ 29 & 32 \\ 0 & 0 \end{bmatrix}$$



### D.1.5. Properties of Matrix Multiplication

As pointed out above, it is sometimes possible to multiply matrices in one order but not in the other order. However, even if both  $AB$  and  $BA$  are defined, they may not be equal.

#### Example D.24: Matrix Multiplication is Not Commutative

Compare the products  $AB$  and  $BA$ , for matrices  $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ ,  $B = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$

**Solution.** First, notice that  $A$  and  $B$  are both of size  $2 \times 2$ . Therefore, both products  $AB$  and  $BA$  are defined. The first product,  $AB$  is

$$AB = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 4 & 3 \end{bmatrix}$$

The second product,  $BA$  is

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 3 & 4 \\ 1 & 2 \end{bmatrix}$$

Therefore,  $AB \neq BA$ .



This example illustrates that you cannot assume  $AB = BA$  even when multiplication is defined in both orders. If for some matrices  $A$  and  $B$  it is true that  $AB = BA$ , then we say that  $A$  and  $B$  **commute**. This is one important property of matrix multiplication.

The following are other important properties of matrix multiplication. Notice that these properties hold only when the size of matrices are such that the products are defined.

#### Proposition D.25: Properties of Matrix Multiplication

The following hold for matrices  $A$ ,  $B$ , and  $C$  and for scalars  $r$  and  $s$ ,

$$A(rB + sC) = r(AB) + s(AC) \quad (4.6)$$

$$(B + C)A = BA + CA \quad (4.7)$$

$$A(BC) = (AB)C \quad (4.8)$$

## D.1.6. The Transpose

Another important operation on matrices is that of taking the **transpose**. For a matrix  $A$ , we denote the transpose of  $A$  by  $A^T$ . Before formally defining the transpose, we explore this operation on the following matrix.

$$\begin{bmatrix} 1 & 4 \\ 3 & 1 \\ 2 & 6 \end{bmatrix}^T = \begin{bmatrix} 1 & 3 & 2 \\ 4 & 1 & 6 \end{bmatrix}$$

What happened? The first column became the first row and the second column became the second row. Thus the  $3 \times 2$  matrix became a  $2 \times 3$  matrix. The number 4 was in the first row and the second column and it ended up in the second row and first column.

The definition of the transpose is as follows.

### Definition D.26: The Transpose of a Matrix

Let  $A$  be an  $m \times n$  matrix. Then  $A^T$ , the **transpose** of  $A$ , denotes the  $n \times m$  matrix given by

$$A^T = [a_{ij}]^T = [a_{ji}]$$

The  $(i, j)$ -entry of  $A$  becomes the  $(j, i)$ -entry of  $A^T$ .

Consider the following example.

### Example D.27: The Transpose of a Matrix

Calculate  $A^T$  for the following matrix

$$A = \begin{bmatrix} 1 & 2 & -6 \\ 3 & 5 & 4 \end{bmatrix}$$

**Solution.** By Definition D.26, we know that for  $A = [a_{ij}]$ ,  $A^T = [a_{ji}]$ . In other words, we switch the row and column location of each entry. The  $(1, 2)$ -entry becomes the  $(2, 1)$ -entry.

Thus,

$$A^T = \begin{bmatrix} 1 & 3 \\ 2 & 5 \\ -6 & 4 \end{bmatrix}$$

Notice that  $A$  is a  $2 \times 3$  matrix, while  $A^T$  is a  $3 \times 2$  matrix.

The transpose of a matrix has the following important properties .



**Lemma D.28: Properties of the Transpose of a Matrix**

Let  $A$  be an  $m \times n$  matrix,  $B$  an  $n \times p$  matrix, and  $r$  and  $s$  scalars. Then

1.  $(A^T)^T = A$
2.  $(AB)^T = B^T A^T$
3.  $(rA + sB)^T = rA^T + sB^T$

The transpose of a matrix is related to other important topics. Consider the following definition.

**Definition D.29: Symmetric and Skew Symmetric Matrices**

An  $n \times n$  matrix  $A$  is said to be **symmetric** if  $A = A^T$ . It is said to be **skew symmetric** if  $A = -A^T$ .

We will explore these definitions in the following examples.

**Example D.30: Symmetric Matrices**

Let

$$A = \begin{bmatrix} 2 & 1 & 3 \\ 1 & 5 & -3 \\ 3 & -3 & 7 \end{bmatrix}$$

Use Definition D.29 to show that  $A$  is symmetric.

**Solution.** By Definition D.29, we need to show that  $A = A^T$ . Now, using Definition D.26,

$$A^T = \begin{bmatrix} 2 & 1 & 3 \\ 1 & 5 & -3 \\ 3 & -3 & 7 \end{bmatrix}$$

Hence,  $A = A^T$ , so  $A$  is symmetric. ♠

**Example D.31: A Skew Symmetric Matrix**

Let

$$A = \begin{bmatrix} 0 & 1 & 3 \\ -1 & 0 & 2 \\ -3 & -2 & 0 \end{bmatrix}$$

Show that  $A$  is skew symmetric.

**Solution.** By Definition D.29,

$$A^T = \begin{bmatrix} 0 & -1 & -3 \\ 1 & 0 & -2 \\ 3 & 2 & 0 \end{bmatrix}$$

You can see that each entry of  $A^T$  is equal to  $-1$  times the same entry of  $A$ . Hence,  $A^T = -A$  and so by Definition D.29,  $A$  is skew symmetric. ♠

### D.1.7. The Identity and Inverses

---

There is a special matrix, denoted  $I$ , which is called to as the **identity matrix**. The identity matrix is always a square matrix, and it has the property that there are ones down the main diagonal and zeroes elsewhere. Here are some identity matrices of various sizes.

$$[1], \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The first is the  $1 \times 1$  identity matrix, the second is the  $2 \times 2$  identity matrix, and so on. By extension, you can likely see what the  $n \times n$  identity matrix would be. When it is necessary to distinguish which size of identity matrix is being discussed, we will use the notation  $I_n$  for the  $n \times n$  identity matrix.

The identity matrix is so important that there is a special symbol to denote the  $ij^{th}$  entry of the identity matrix. This symbol is given by  $I_{ij} = \delta_{ij}$  where  $\delta_{ij}$  is the **Kronecker symbol** defined by

$$\delta_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases}$$

$I_n$  is called the **identity matrix** because it is a **multiplicative identity** in the following sense.

#### Lemma D.32: Multiplication by the Identity Matrix

Suppose  $A$  is an  $m \times n$  matrix and  $I_n$  is the  $n \times n$  identity matrix. Then  $AI_n = A$ . If  $I_m$  is the  $m \times m$  identity matrix, it also follows that  $I_mA = A$ .

We now define the matrix operation which in some ways plays the role of division.

#### Definition D.33: The Inverse of a Matrix

A square  $n \times n$  matrix  $A$  is said to have an **inverse**  $A^{-1}$  if and only if

$$AA^{-1} = A^{-1}A = I_n$$

In this case, the matrix  $A$  is called **invertible**.

Such a matrix  $A^{-1}$  will have the same size as the matrix  $A$ . It is very important to observe that the inverse of a matrix, if it exists, is unique. Another way to think of this is that if it acts like the inverse, then it **is** the inverse.

**Theorem D.34: Uniqueness of Inverse**

Suppose  $A$  is an  $n \times n$  matrix such that an inverse  $A^{-1}$  exists. Then there is only one such inverse matrix. That is, given any matrix  $B$  such that  $AB = BA = I$ ,  $B = A^{-1}$ .

The next example demonstrates how to check the inverse of a matrix.

**Example D.35: Verifying the Inverse of a Matrix**

Let  $A = \begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix}$ . Show  $\begin{bmatrix} 2 & -1 \\ -1 & 1 \end{bmatrix}$  is the inverse of  $A$ .

**Solution.** To check this, multiply

$$\begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} 2 & -1 \\ -1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = I$$

and

$$\begin{bmatrix} 2 & -1 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = I$$

showing that this matrix is indeed the inverse of  $A$ . ♠

Unlike ordinary multiplication of numbers, it can happen that  $A \neq 0$  but  $A$  may fail to have an inverse. This is illustrated in the following example.

**Example D.36: A Nonzero Matrix With No Inverse**

Let  $A = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$ . Show that  $A$  does not have an inverse.

**Solution.** One might think  $A$  would have an inverse because it does not equal zero. However, note that

$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} -1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

If  $A^{-1}$  existed, we would have the following

$$\begin{aligned} \begin{bmatrix} 0 \\ 0 \end{bmatrix} &= A^{-1} \left( \begin{bmatrix} 0 \\ 0 \end{bmatrix} \right) \\ &= A^{-1} \left( A \begin{bmatrix} -1 \\ 1 \end{bmatrix} \right) \\ &= (A^{-1}A) \begin{bmatrix} -1 \\ 1 \end{bmatrix} \\ &= I \begin{bmatrix} -1 \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} -1 \\ 1 \end{bmatrix} \end{aligned}$$

This says that

$$\begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} -1 \\ 1 \end{bmatrix}$$

which is impossible! Therefore,  $A$  does not have an inverse. ♠

In the next section, we will explore how to find the inverse of a matrix, if it exists.

### D.1.8. Finding the Inverse of a Matrix

---

In Example D.35, we were given  $A^{-1}$  and asked to verify that this matrix was in fact the inverse of  $A$ . In this section, we explore how to find  $A^{-1}$ .

Let

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

as in Example D.35. In order to find  $A^{-1}$ , we need to find a matrix  $\begin{bmatrix} x & z \\ y & w \end{bmatrix}$  such that

$$\begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} x & z \\ y & w \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

We can multiply these two matrices, and see that in order for this equation to be true, we must find the solution to the systems of equations,

$$\begin{aligned} x + y &= 1 \\ x + 2y &= 0 \end{aligned}$$

and

$$\begin{aligned} z + w &= 0 \\ z + 2w &= 1 \end{aligned}$$

Writing the augmented matrix for these two systems gives

$$\left[ \begin{array}{cc|c} 1 & 1 & 1 \\ 1 & 2 & 0 \end{array} \right]$$

for the first system and

$$\left[ \begin{array}{cc|c} 1 & 1 & 0 \\ 1 & 2 & 1 \end{array} \right] \tag{4.9}$$

for the second.

Let's solve the first system. Take  $-1$  times the first row and add to the second to get

$$\left[ \begin{array}{cc|c} 1 & 1 & 1 \\ 0 & 1 & -1 \end{array} \right]$$

Now take  $-1$  times the second row and add to the first to get

$$\left[ \begin{array}{cc|c} 1 & 0 & 2 \\ 0 & 1 & -1 \end{array} \right]$$

Writing in terms of variables, this says  $x = 2$  and  $y = -1$ .

Now solve the second system, 4.9 to find  $z$  and  $w$ . You will find that  $z = -1$  and  $w = 1$ .

If we take the values found for  $x, y, z$ , and  $w$  and put them into our inverse matrix, we see that the inverse is

$$A^{-1} = \begin{bmatrix} x & z \\ y & w \end{bmatrix} = \begin{bmatrix} 2 & -1 \\ -1 & 1 \end{bmatrix}$$

After taking the time to solve the second system, you may have noticed that exactly the same row operations were used to solve both systems. In each case, the end result was something of the form  $[I|X]$  where  $I$  is the identity and  $X$  gave a column of the inverse. In the above,

$$\begin{bmatrix} x \\ y \end{bmatrix}$$

the first column of the inverse was obtained by solving the first system and then the second column

$$\begin{bmatrix} z \\ w \end{bmatrix}$$

To simplify this procedure, we could have solved both systems at once! To do so, we could have written

$$\left[ \begin{array}{cc|cc} 1 & 1 & 1 & 0 \\ 1 & 2 & 0 & 1 \end{array} \right]$$

and row reduced until we obtained

$$\left[ \begin{array}{cc|cc} 1 & 0 & 2 & -1 \\ 0 & 1 & -1 & 1 \end{array} \right]$$

and read off the inverse as the  $2 \times 2$  matrix on the right side.

This exploration motivates the following important algorithm.

**Matrix Inverse Algorithm** Suppose  $A$  is an  $n \times n$  matrix. To find  $A^{-1}$  if it exists, form the augmented  $n \times 2n$  matrix

$$[A|I]$$

If possible do row operations until you obtain an  $n \times 2n$  matrix of the form

$$[I|B]$$

When this has been done,  $B = A^{-1}$ . In this case, we say that  $A$  is **invertible**. If it is impossible to row reduce to a matrix of the form  $[I|B]$ , then  $A$  has no inverse.

This algorithm shows how to find the inverse if it exists. It will also tell you if  $A$  does not have an inverse.

Consider the following example.

**Example D.37: Finding the Inverse**

Let  $A = \begin{bmatrix} 1 & 2 & 2 \\ 1 & 0 & 2 \\ 3 & 1 & -1 \end{bmatrix}$ . Find  $A^{-1}$  if it exists.

**Solution.** Set up the augmented matrix

$$[A|I] = \left[ \begin{array}{ccc|ccc} 1 & 2 & 2 & 1 & 0 & 0 \\ 1 & 0 & 2 & 0 & 1 & 0 \\ 3 & 1 & -1 & 0 & 0 & 1 \end{array} \right]$$

Now we row reduce, with the goal of obtaining the  $3 \times 3$  identity matrix on the left hand side. First, take  $-1$  times the first row and add to the second followed by  $-3$  times the first row added to the third row. This yields

$$\left[ \begin{array}{ccc|ccc} 1 & 2 & 2 & 1 & 0 & 0 \\ 0 & -2 & 0 & -1 & 1 & 0 \\ 0 & -5 & -7 & -3 & 0 & 1 \end{array} \right]$$

Then take 5 times the second row and add to  $-2$  times the third row.

$$\left[ \begin{array}{ccc|ccc} 1 & 2 & 2 & 1 & 0 & 0 \\ 0 & -10 & 0 & -5 & 5 & 0 \\ 0 & 0 & 14 & 1 & 5 & -2 \end{array} \right]$$

Next take the third row and add to  $-7$  times the first row. This yields

$$\left[ \begin{array}{ccc|ccc} -7 & -14 & 0 & -6 & 5 & -2 \\ 0 & -10 & 0 & -5 & 5 & 0 \\ 0 & 0 & 14 & 1 & 5 & -2 \end{array} \right]$$

Now take  $-\frac{1}{5}$  times the second row and add to the first row.

$$\left[ \begin{array}{ccc|ccc} -7 & 0 & 0 & 1 & -2 & -2 \\ 0 & -10 & 0 & -5 & 5 & 0 \\ 0 & 0 & 14 & 1 & 5 & -2 \end{array} \right]$$

Finally divide the first row by  $-7$ , the second row by  $-10$  and the third row by  $14$  which yields

$$\left[ \begin{array}{ccc|ccc} 1 & 0 & 0 & -\frac{1}{7} & \frac{2}{7} & \frac{2}{7} \\ 0 & 1 & 0 & \frac{1}{2} & -\frac{1}{2} & 0 \\ 0 & 0 & 1 & \frac{1}{14} & \frac{5}{14} & -\frac{1}{7} \end{array} \right]$$

Notice that the left hand side of this matrix is now the  $3 \times 3$  identity matrix  $I_3$ . Therefore, the inverse is the  $3 \times 3$  matrix on the right hand side, given by

$$\left[ \begin{array}{ccc} -\frac{1}{7} & \frac{2}{7} & \frac{2}{7} \\ \frac{1}{2} & -\frac{1}{2} & 0 \\ \frac{1}{14} & \frac{5}{14} & -\frac{1}{7} \end{array} \right]$$



It may happen that through this algorithm, you discover that the left hand side cannot be row reduced to the identity matrix. Consider the following example of this situation.

### Example D.38: A Matrix Which Has No Inverse

Let  $A = \begin{bmatrix} 1 & 2 & 2 \\ 1 & 0 & 2 \\ 2 & 2 & 4 \end{bmatrix}$ . Find  $A^{-1}$  if it exists.

**Solution.** Write the augmented matrix  $[A|I]$

$$\left[ \begin{array}{ccc|ccc} 1 & 2 & 2 & 1 & 0 & 0 \\ 1 & 0 & 2 & 0 & 1 & 0 \\ 2 & 2 & 4 & 0 & 0 & 1 \end{array} \right]$$

and proceed to do row operations attempting to obtain  $[I|A^{-1}]$ . Take  $-1$  times the first row and add to the second. Then take  $-2$  times the first row and add to the third row.

$$\left[ \begin{array}{ccc|ccc} 1 & 2 & 2 & 1 & 0 & 0 \\ 0 & -2 & 0 & -1 & 1 & 0 \\ 0 & -2 & 0 & -2 & 0 & 1 \end{array} \right]$$

Next add  $-1$  times the second row to the third row.

$$\left[ \begin{array}{ccc|ccc} 1 & 2 & 2 & 1 & 0 & 0 \\ 0 & -2 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & -1 & -1 & 1 \end{array} \right]$$

At this point, you can see there will be no way to obtain  $I$  on the left side of this augmented matrix. Hence, there is no way to complete this algorithm, and therefore the inverse of  $A$  does not exist. In this case, we say that  $A$  is not invertible.

If the algorithm provides an inverse for the original matrix, it is always possible to check your answer. To do so, use the method demonstrated in Example D.35. Check that the products  $AA^{-1}$  and  $A^{-1}A$  both equal the identity matrix. Through this method, you can always be sure that you have calculated  $A^{-1}$  properly!

One way in which the inverse of a matrix is useful is to find the solution of a system of linear equations. Recall from Definition D.15 that we can write a system of equations in matrix form, which is of the form  $AX = B$ . Suppose you find the inverse of the matrix  $A^{-1}$ . Then you could multiply both sides of this equation on the left by  $A^{-1}$  and simplify to obtain

$$\begin{aligned} (A^{-1})AX &= A^{-1}B \\ (A^{-1}A)X &= A^{-1}B \\ IX &= A^{-1}B \\ X &= A^{-1}B \end{aligned}$$

Therefore we can find  $X$ , the solution to the system, by computing  $X = A^{-1}B$ . Note that once you have found  $A^{-1}$ , you can easily get the solution for different right hand sides (different  $B$ ). It is always just  $A^{-1}B$ .

We will explore this method of finding the solution to a system in the following example.

### Example D.39: Using the Inverse to Solve a System of Equations

*Consider the following system of equations. Use the inverse of a suitable matrix to give the solutions to this system.*

$$\begin{aligned}x + z &= 1 \\x - y + z &= 3 \\x + y - z &= 2\end{aligned}$$

**Solution.** First, we can write the system of equations in matrix form

$$AX = \begin{bmatrix} 1 & 0 & 1 \\ 1 & -1 & 1 \\ 1 & 1 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \\ 2 \end{bmatrix} = B \quad (4.10)$$

The inverse of the matrix

$$A = \begin{bmatrix} 1 & 0 & 1 \\ 1 & -1 & 1 \\ 1 & 1 & -1 \end{bmatrix}$$

is

$$A^{-1} = \begin{bmatrix} 0 & \frac{1}{2} & \frac{1}{2} \\ 1 & -1 & 0 \\ 1 & -\frac{1}{2} & -\frac{1}{2} \end{bmatrix}$$

Verifying this inverse is left as an exercise.

From here, the solution to the given system 4.10 is found by

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = A^{-1}B = \begin{bmatrix} 0 & \frac{1}{2} & \frac{1}{2} \\ 1 & -1 & 0 \\ 1 & -\frac{1}{2} & -\frac{1}{2} \end{bmatrix} \begin{bmatrix} 1 \\ 3 \\ 2 \end{bmatrix} = \begin{bmatrix} \frac{5}{2} \\ -2 \\ -\frac{3}{2} \end{bmatrix}$$



What if the right side,  $B$ , of 4.10 had been  $\begin{bmatrix} 0 \\ 1 \\ 3 \end{bmatrix}$ ? In other words, what would be the solution to

$$\begin{bmatrix} 1 & 0 & 1 \\ 1 & -1 & 1 \\ 1 & 1 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 3 \end{bmatrix}?$$

By the above discussion, the solution is given by

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = A^{-1}B = \begin{bmatrix} 0 & \frac{1}{2} & \frac{1}{2} \\ 1 & -1 & 0 \\ 1 & -\frac{1}{2} & -\frac{1}{2} \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 3 \end{bmatrix} = \begin{bmatrix} 2 \\ -1 \\ -2 \end{bmatrix}$$

This illustrates that for a system  $AX = B$  where  $A^{-1}$  exists, it is easy to find the solution when the vector  $B$  is changed.

# E. Determinants

---

## E.1 Basic Techniques and Properties

---

### Outcomes

- A. Evaluate the determinant of a square matrix using either Laplace Expansion or row operations.
- B. Demonstrate the effects that row operations have on determinants.
- C. Verify the following:
  - (a) The determinant of a product of matrices is the product of the determinants.
  - (b) The determinant of a matrix is equal to the determinant of its transpose.

### E.1.1. Cofactors and $2 \times 2$ Determinants

---

Let  $A$  be an  $n \times n$  matrix. That is, let  $A$  be a square matrix. The **determinant** of  $A$ , denoted by  $\det(A)$  is a very important number which we will explore throughout this section.

If  $A$  is a  $2 \times 2$  matrix, the determinant is given by the following formula.

#### Definition E.1: Determinant of a Two By Two Matrix

Let  $A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$ . Then  
$$\det(A) = ad - cb$$

The determinant is also often denoted by enclosing the matrix with two vertical lines. Thus

$$\det \begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc$$

The following is an example of finding the determinant of a  $2 \times 2$  matrix.

**Example E.2: A Two by Two Determinant**

Find  $\det(A)$  for the matrix  $A = \begin{bmatrix} 2 & 4 \\ -1 & 6 \end{bmatrix}$ .

**Solution.** From Definition E.1,

$$\det(A) = (2)(6) - (-1)(4) = 12 + 4 = 16$$



The  $2 \times 2$  determinant can be used to find the determinant of larger matrices. We will now explore how to find the determinant of a  $3 \times 3$  matrix, using several tools including the  $2 \times 2$  determinant.

We begin with the following definition.

**Definition E.3: The  $ij^{\text{th}}$  Minor of a Matrix**

Let  $A$  be a  $3 \times 3$  matrix. The  $ij^{\text{th}}$  **minor** of  $A$ , denoted as  $\text{minor}(A)_{ij}$ , is the determinant of the  $2 \times 2$  matrix which results from deleting the  $i^{\text{th}}$  row and the  $j^{\text{th}}$  column of  $A$ .

In general, if  $A$  is an  $n \times n$  matrix, then the  $ij^{\text{th}}$  minor of  $A$  is the determinant of the  $(n-1) \times (n-1)$  matrix which results from deleting the  $i^{\text{th}}$  row and the  $j^{\text{th}}$  column of  $A$ .

Hence, there is a minor associated with each entry of  $A$ . Consider the following example which demonstrates this definition.

**Example E.4: Finding Minors of a Matrix**

Let

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

Find  $\text{minor}(A)_{12}$  and  $\text{minor}(A)_{23}$ .

**Solution.** First we will find  $\text{minor}(A)_{12}$ . By Definition E.3, this is the determinant of the  $2 \times 2$  matrix which results when you delete the first row and the second column. This minor is given by

$$\text{minor}(A)_{12} = \det \begin{bmatrix} 4 & 2 \\ 3 & 1 \end{bmatrix}$$

Using Definition E.1, we see that

$$\det \begin{bmatrix} 4 & 2 \\ 3 & 1 \end{bmatrix} = (4)(1) - (3)(2) = 4 - 6 = -2$$

Therefore  $\text{minor}(A)_{12} = -2$ .

Similarly,  $\text{minor}(A)_{23}$  is the determinant of the  $2 \times 2$  matrix which results when you delete the second row and the third column. This minor is therefore

$$\text{minor}(A)_{23} = \det \begin{bmatrix} 1 & 2 \\ 3 & 2 \end{bmatrix} = -4$$

Finding the other minors of  $A$  is left as an exercise.



The  $ij^{\text{th}}$  minor of a matrix  $A$  is used in another important definition, given next.

### Definition E.5: The $ij^{\text{th}}$ Cofactor of a Matrix

Suppose  $A$  is an  $n \times n$  matrix. The  $ij^{\text{th}}$  **cofactor**, denoted by  $\text{cof}(A)_{ij}$  is defined to be

$$\text{cof}(A)_{ij} = (-1)^{i+j} \text{minor}(A)_{ij}$$

It is also convenient to refer to the cofactor of an entry of a matrix as follows. If  $a_{ij}$  is the  $ij^{\text{th}}$  entry of the matrix, then its cofactor is just  $\text{cof}(A)_{ij}$ .

### Example E.6: Finding Cofactors of a Matrix

Consider the matrix

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

Find  $\text{cof}(A)_{12}$  and  $\text{cof}(A)_{23}$ .

**Solution.** We will use Definition E.5 to compute these cofactors.

First, we will compute  $\text{cof}(A)_{12}$ . Therefore, we need to find  $\text{minor}(A)_{12}$ . This is the determinant of the  $2 \times 2$  matrix which results when you delete the first row and the second column. Thus  $\text{minor}(A)_{12}$  is given by

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

Then,

$$\text{cof}(A)_{12} = (-1)^{1+2} \text{minor}(A)_{12} = (-1)^{1+2} (-2) = 2$$

Hence,  $\text{cof}(A)_{12} = 2$ .

Similarly, we can find  $\text{cof}(A)_{23}$ . First, find  $\text{minor}(A)_{23}$ , which is the determinant of the  $2 \times 2$  matrix which results when you delete the second row and the third column. This minor is therefore

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

Hence,

$$\text{cof}(A)_{23} = (-1)^{2+3} \text{minor}(A)_{23} = (-1)^{2+3} (-4) = 4$$



You may wish to find the remaining cofactors for the above matrix. Remember that there is a cofactor for every entry in the matrix.

We have now established the tools we need to find the determinant of a  $3 \times 3$  matrix.

### Definition E.7: The Determinant of a Three By Three Matrix

Let  $A$  be a  $3 \times 3$  matrix. Then,  $\det(A)$  is calculated by picking a row (or column) and taking the product of each entry in that row (column) with its cofactor and adding these products together. This process when applied to the  $i^{\text{th}}$  row (column) is known as **expanding along the  $i^{\text{th}}$  row (column)** as is given by

$$\det(A) = a_{i1}\text{cof}(A)_{i1} + a_{i2}\text{cof}(A)_{i2} + a_{i3}\text{cof}(A)_{i3}$$

When calculating the determinant, you can choose to expand any row or any column. Regardless of your choice, you will always get the same number which is the determinant of the matrix  $A$ . This method of evaluating a determinant by expanding along a row or a column is called **Laplace Expansion or Cofactor Expansion**.

Consider the following example.

### Example E.8: Finding the Determinant of a Three by Three Matrix

Let

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

Find  $\det(A)$  using the method of Laplace Expansion.

**Solution.** First, we will calculate  $\det(A)$  by expanding along the first column. Using Definition E.7, we take the 1 in the first column and multiply it by its cofactor,

$$1(-1)^{1+1} \left| \begin{array}{cc} 3 & 2 \\ 2 & 1 \end{array} \right| = (1)(1)(-1) = -1$$

Similarly, we take the 4 in the first column and multiply it by its cofactor, as well as with the 3 in the first column. Finally, we add these numbers together, as given in the following equation.

$$\det(A) = \underbrace{1(-1)^{1+1} \left| \begin{array}{cc} 3 & 2 \\ 2 & 1 \end{array} \right|}_{\text{cof}(A)_{11}} + \underbrace{4(-1)^{2+1} \left| \begin{array}{cc} 2 & 3 \\ 2 & 1 \end{array} \right|}_{\text{cof}(A)_{21}} + \underbrace{3(-1)^{3+1} \left| \begin{array}{cc} 2 & 3 \\ 3 & 2 \end{array} \right|}_{\text{cof}(A)_{31}}$$

Calculating each of these, we obtain

$$\det(A) = 1(1)(-1) + 4(-1)(-4) + 3(1)(-5) = -1 + 16 - 15 = 0$$

Hence,  $\det(A) = 0$ .

As mentioned in Definition E.7, we can choose to expand along any row or column. Let's try now by expanding along the second row. Here, we take the 4 in the second row and multiply it to its cofactor, then add this to the 3 in the second row multiplied by its cofactor, and the 2 in the second row multiplied by its cofactor. The calculation is as follows.

$$\det(A) = 4(-1)^{2+1} \overbrace{\begin{vmatrix} 2 & 3 \\ 2 & 1 \end{vmatrix}}^{\text{cof}(A)_{21}} + 3(-1)^{2+2} \overbrace{\begin{vmatrix} 1 & 3 \\ 3 & 1 \end{vmatrix}}^{\text{cof}(A)_{22}} + 2(-1)^{2+3} \overbrace{\begin{vmatrix} 1 & 2 \\ 3 & 2 \end{vmatrix}}^{\text{cof}(A)_{23}}$$

Calculating each of these products, we obtain

$$\det(A) = 4(-1)(-2) + 3(1)(-8) + 2(-1)(-4) = 0$$

You can see that for both methods, we obtained  $\det(A) = 0$ . ♠

As mentioned above, we will always come up with the same value for  $\det(A)$  regardless of the row or column we choose to expand along. You should try to compute the above determinant by expanding along other rows and columns. This is a good way to check your work, because you should come up with the same number each time!

We present this idea formally in the following theorem.

### Theorem E.9: The Determinant is Well Defined

*Expanding the  $n \times n$  matrix along any row or column always gives the same answer, which is the determinant.*

We have now looked at the determinant of  $2 \times 2$  and  $3 \times 3$  matrices. It turns out that the method used to calculate the determinant of a  $3 \times 3$  matrix can be used to calculate the determinant of any sized matrix. Notice that Definition E.3, Definition E.5 and Definition E.7 can all be applied to a matrix of any size.

For example, the  $i j^{th}$  minor of a  $4 \times 4$  matrix is the determinant of the  $3 \times 3$  matrix you obtain when you delete the  $i^{th}$  row and the  $j^{th}$  column. Just as with the  $3 \times 3$  determinant, we can compute the determinant of a  $4 \times 4$  matrix by Laplace Expansion, along any row or column.

Consider the following example.

### Example E.10: Determinant of a Four by Four Matrix

Find  $\det(A)$  where

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 4 & 2 & 3 \\ 1 & 3 & 4 & 5 \\ 3 & 4 & 3 & 2 \end{bmatrix}$$

**Solution.** As in the case of a  $3 \times 3$  matrix, you can expand this along any row or column. Lets pick the third column. Then, using Laplace Expansion,

$$\det(A) = 3(-1)^{1+3} \begin{vmatrix} 5 & 4 & 3 \\ 1 & 3 & 5 \\ 3 & 4 & 2 \end{vmatrix} + 2(-1)^{2+3} \begin{vmatrix} 1 & 2 & 4 \\ 1 & 3 & 5 \\ 3 & 4 & 2 \end{vmatrix} +$$

$$4(-1)^{3+3} \begin{vmatrix} 1 & 2 & 4 \\ 5 & 4 & 3 \\ 3 & 4 & 2 \end{vmatrix} + 3(-1)^{4+3} \begin{vmatrix} 1 & 2 & 4 \\ 5 & 4 & 3 \\ 1 & 3 & 5 \end{vmatrix}$$

Now, you can calculate each  $3 \times 3$  determinant using Laplace Expansion, as we did above. You should complete these as an exercise and verify that  $\det(A) = -12$ . ♠

The following provides a formal definition for the determinant of an  $n \times n$  matrix. You may wish to take a moment and consider the above definitions for  $2 \times 2$  and  $3 \times 3$  determinants in context of this definition.

### Definition E.11: The Determinant of an $n \times n$ Matrix

Let  $A$  be an  $n \times n$  matrix where  $n \geq 2$  and suppose the determinant of an  $(n-1) \times (n-1)$  has been defined. Then

$$\det(A) = \sum_{j=1}^n a_{ij} \text{cof}(A)_{ij} = \sum_{i=1}^n a_{ij} \text{cof}(A)_{ij}$$

The first formula consists of expanding the determinant along the  $i^{th}$  row and the second expands the determinant along the  $j^{th}$  column.

In the following sections, we will explore some important properties and characteristics of the determinant.

## E.1.2. The Determinant of a Triangular Matrix

There is a certain type of matrix for which finding the determinant is a very simple procedure. Consider the following definition.

### Definition E.12: Triangular Matrices

A matrix  $A$  is upper triangular if  $a_{ij} = 0$  whenever  $i > j$ . Thus the entries of such a matrix below the main diagonal equal 0, as shown. Here, \* refers to any nonzero number.

$$\begin{bmatrix} * & * & \cdots & * \\ 0 & * & \cdots & : \\ \vdots & \vdots & \ddots & * \\ 0 & \cdots & 0 & * \end{bmatrix}$$

A lower triangular matrix is defined similarly as a matrix for which all entries above the main diagonal are equal to zero.

The following theorem provides a useful way to calculate the determinant of a triangular matrix.

### Theorem E.13: Determinant of a Triangular Matrix

Let  $A$  be an upper or lower triangular matrix. Then  $\det(A)$  is obtained by taking the product of the entries on the main diagonal.

The verification of this Theorem can be done by computing the determinant using Laplace Expansion along the first row or column.

Consider the following example.

### Example E.14: Determinant of a Triangular Matrix

Let

$$A = \begin{bmatrix} 1 & 2 & 3 & 77 \\ 0 & 2 & 6 & 7 \\ 0 & 0 & 3 & 33.7 \\ 0 & 0 & 0 & -1 \end{bmatrix}$$

Find  $\det(A)$ .

**Solution.** From Theorem E.13, it suffices to take the product of the elements on the main diagonal. Thus  $\det(A) = 1 \times 2 \times 3 \times (-1) = -6$ .

Without using Theorem E.13, you could use Laplace Expansion. We will expand along the first col-

umn. This gives

$$\det(A) = 1 \begin{vmatrix} 2 & 6 & 7 \\ 0 & 3 & 33.7 \\ 0 & 0 & -1 \end{vmatrix} + 0(-1)^{2+1} \begin{vmatrix} 2 & 3 & 77 \\ 0 & 3 & 33.7 \\ 0 & 0 & -1 \end{vmatrix} + \\ 0(-1)^{3+1} \begin{vmatrix} 2 & 3 & 77 \\ 2 & 6 & 7 \\ 0 & 0 & -1 \end{vmatrix} + 0(-1)^{4+1} \begin{vmatrix} 2 & 3 & 77 \\ 2 & 6 & 7 \\ 0 & 3 & 33.7 \end{vmatrix}$$

and the only nonzero term in the expansion is

$$1 \begin{vmatrix} 2 & 6 & 7 \\ 0 & 3 & 33.7 \\ 0 & 0 & -1 \end{vmatrix}$$

Now find the determinant of this  $3 \times 3$  matrix, by expanding along the first column to obtain

$$\begin{aligned} \det(A) &= 1 \times \left( 2 \times \begin{vmatrix} 3 & 33.7 \\ 0 & -1 \end{vmatrix} + 0(-1)^{2+1} \begin{vmatrix} 6 & 7 \\ 0 & -1 \end{vmatrix} + 0(-1)^{3+1} \begin{vmatrix} 6 & 7 \\ 3 & 33.7 \end{vmatrix} \right) \\ &= 1 \times 2 \times \begin{vmatrix} 3 & 33.7 \\ 0 & -1 \end{vmatrix} \end{aligned}$$

Next use Definition E.1 to find the determinant of this  $2 \times 2$  matrix, which is just  $3 \times -1 - 0 \times 33.7 = -3$ . Putting all these steps together, we have

$$\det(A) = 1 \times 2 \times 3 \times (-1) = -6$$

which is just the product of the entries down the main diagonal of the original matrix! ♠

You can see that while both methods result in the same answer, Theorem E.13 provides a much quicker method.

In the next section, we explore some important properties of determinants.

## E.2 Applications of the Determinant

---

### Outcomes

- A. Apply Cramer's Rule to solve a  $2 \times 2$  or a  $3 \times 3$  linear system.

### E.2.1. Cramer's Rule

---

Recall that we can represent a system of linear equations in the form  $AX = B$ , where the solutions to this system are given by  $X$ . Cramer's Rule gives a formula for the solutions  $X$  in the special case that  $A$  is a square invertible matrix. Note this rule does not apply if you have a system of equations in which there is a different number of equations than variables (in other words, when  $A$  is not square), or when  $A$  is not invertible.

Suppose we have a system of equations given by  $AX = B$ , and we want to find solutions  $X$  which satisfy this system. Then recall that if  $A^{-1}$  exists,

$$\begin{aligned} AX &= B \\ A^{-1}(AX) &= A^{-1}B \\ (A^{-1}A)X &= A^{-1}B \\ IX &= A^{-1}B \\ X &= A^{-1}B \end{aligned}$$

Hence, the solutions  $X$  to the system are given by  $X = A^{-1}B$ . Since we assume that  $A^{-1}$  exists, we can use the formula for  $A^{-1}$  given above. Substituting this formula into the equation for  $X$ , we have

$$X = A^{-1}B = \frac{1}{\det(A)} \text{adj}(A)B$$

Let  $x_i$  be the  $i^{th}$  entry of  $X$  and  $b_j$  be the  $j^{th}$  entry of  $B$ . Then this equation becomes

$$x_i = \sum_{j=1}^n [a_{ij}]^{-1} b_j = \sum_{j=1}^n \frac{1}{\det(A)} \text{adj}(A)_{ij} b_j$$

where  $\text{adj}(A)_{ij}$  is the  $i,j^{th}$  entry of  $\text{adj}(A)$ .

By the formula for the expansion of a determinant along a column,

$$x_i = \frac{1}{\det(A)} \det \begin{bmatrix} * & \cdots & b_1 & \cdots & * \\ \vdots & & \vdots & & \vdots \\ * & \cdots & b_n & \cdots & * \end{bmatrix}$$

where here the  $i^{th}$  column of  $A$  is replaced with the column vector  $[b_1 \dots, b_n]^T$ . The determinant of this modified matrix is taken and divided by  $\det(A)$ . This formula is known as Cramer's rule.

We formally define this method now.

**Procedure E.15: Using Cramer's Rule**

Suppose  $A$  is an  $n \times n$  invertible matrix and we wish to solve the system  $AX = B$  for  $X = [x_1, \dots, x_n]^T$ . Then Cramer's rule says

$$x_i = \frac{\det(A_i)}{\det(A)}$$

where  $A_i$  is the matrix obtained by replacing the  $i^{\text{th}}$  column of  $A$  with the column matrix

$$B = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix}$$

We illustrate this procedure in the following example.

**Example E.16: Using Cramer's Rule**

Find  $x, y, z$  if

$$\begin{bmatrix} 1 & 2 & 1 \\ 3 & 2 & 1 \\ 2 & -3 & 2 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

**Solution.** We will use method outlined in Procedure E.15 to find the values for  $x, y, z$  which give the solution to this system. Let

$$B = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

In order to find  $x$ , we calculate

$$x = \frac{\det(A_1)}{\det(A)}$$

where  $A_1$  is the matrix obtained from replacing the first column of  $A$  with  $B$ .

Hence,  $A_1$  is given by

$$A_1 = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 2 & 1 \\ 3 & -3 & 2 \end{bmatrix}$$

Therefore,

$$x = \frac{\det(A_1)}{\det(A)} = \frac{\begin{vmatrix} 1 & 2 & 1 \\ 2 & 2 & 1 \\ 3 & -3 & 2 \end{vmatrix}}{\begin{vmatrix} 1 & 2 & 1 \\ 3 & 2 & 1 \\ 2 & -3 & 2 \end{vmatrix}} = \frac{1}{2}$$

Similarly, to find  $y$  we construct  $A_2$  by replacing the second column of  $A$  with  $B$ . Hence,  $A_2$  is given by

$$A_2 = \begin{bmatrix} 1 & 1 & 1 \\ 3 & 2 & 1 \\ 2 & 3 & 2 \end{bmatrix}$$

Therefore,

$$y = \frac{\det(A_2)}{\det(A)} = \frac{\begin{vmatrix} 1 & 1 & 1 \\ 3 & 2 & 1 \\ 2 & 3 & 2 \end{vmatrix}}{\begin{vmatrix} 1 & 2 & 1 \\ 3 & 2 & 1 \\ 2 & -3 & 2 \end{vmatrix}} = -\frac{1}{7}$$

Similarly,  $A_3$  is constructed by replacing the third column of  $A$  with  $B$ . Then,  $A_3$  is given by

$$A_3 = \begin{bmatrix} 1 & 2 & 1 \\ 3 & 2 & 2 \\ 2 & -3 & 3 \end{bmatrix}$$

Therefore,  $z$  is calculated as follows.

$$z = \frac{\det(A_3)}{\det(A)} = \frac{\begin{vmatrix} 1 & 2 & 1 \\ 3 & 2 & 2 \\ 2 & -3 & 3 \end{vmatrix}}{\begin{vmatrix} 1 & 2 & 1 \\ 3 & 2 & 1 \\ 2 & -3 & 2 \end{vmatrix}} = \frac{11}{14}$$



Cramer's Rule gives you another tool to consider when solving a system of linear equations.



## F. $\mathbb{R}^n$

---

### F.1 Vectors in $\mathbb{R}^n$

---

#### Outcomes

- A. Find the position vector of a point in  $\mathbb{R}^n$ .

The notation  $\mathbb{R}^n$  refers to the collection of ordered lists of  $n$  real numbers, that is

$$\mathbb{R}^n = \{(x_1 \dots x_n) : x_j \in \mathbb{R} \text{ for } j = 1, \dots, n\}$$

In this chapter, we take a closer look at vectors in  $\mathbb{R}^n$ . First, we will consider what  $\mathbb{R}^n$  looks like in more detail. Recall that the point given by  $0 = (0, \dots, 0)$  is called the **origin**.

Now, consider the case of  $\mathbb{R}^n$  for  $n = 1$ . Then from the definition we can identify  $\mathbb{R}$  with points in  $\mathbb{R}^1$  as follows:

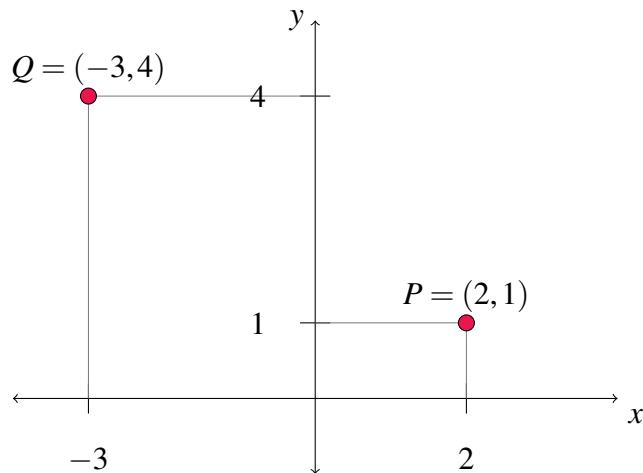
$$\mathbb{R} = \mathbb{R}^1 = \{(x_1) : x_1 \in \mathbb{R}\}$$

Hence,  $\mathbb{R}$  is defined as the set of all real numbers and geometrically, we can describe this as all the points on a line.

Now suppose  $n = 2$ . Then, from the definition,

$$\mathbb{R}^2 = \{(x_1, x_2) : x_j \in \mathbb{R} \text{ for } j = 1, 2\}$$

Consider the familiar coordinate plane, with an  $x$  axis and a  $y$  axis. Any point within this coordinate plane is identified by where it is located along the  $x$  axis, and also where it is located along the  $y$  axis. Consider as an example the following diagram.



Hence, every element in  $\mathbb{R}^2$  is identified by two components,  $x$  and  $y$ , in the usual manner. The coordinates  $x, y$  (or  $x_1, x_2$ ) uniquely determine a point in the plane. Note that while the definition uses  $x_1$  and  $x_2$  to label the coordinates and you may be used to  $x$  and  $y$ , these notations are equivalent.

Now suppose  $n = 3$ . You may have previously encountered the 3-dimensional coordinate system, given by

$$\mathbb{R}^3 = \{(x_1, x_2, x_3) : x_j \in \mathbb{R} \text{ for } j = 1, 2, 3\}$$

Points in  $\mathbb{R}^3$  will be determined by three coordinates, often written  $(x, y, z)$  which correspond to the  $x$ ,  $y$ , and  $z$  axes. We can think as above that the first two coordinates determine a point in a plane. The third component determines the height above or below the plane, depending on whether this number is positive or negative, and all together this determines a point in space. You see that the ordered triples correspond to points in space just as the ordered pairs correspond to points in a plane and single real numbers correspond to points on a line.

The idea behind the more general  $\mathbb{R}^n$  is that we can extend these ideas beyond  $n = 3$ . This discussion regarding points in  $\mathbb{R}^n$  leads into a study of vectors in  $\mathbb{R}^n$ . While we consider  $\mathbb{R}^n$  for all  $n$ , we will largely focus on  $n = 2, 3$  in this section.

Consider the following definition.

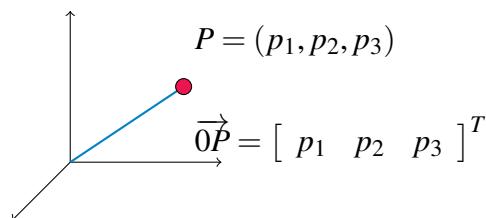
### Definition F.1: The Position Vector

Let  $P = (p_1, \dots, p_n)$  be the coordinates of a point in  $\mathbb{R}^n$ . Then the vector  $\overrightarrow{OP}$  with its tail at  $0 = (0, \dots, 0)$  and its tip at  $P$  is called the **position vector** of the point  $P$ . We write

$$\overrightarrow{OP} = \begin{bmatrix} p_1 \\ \vdots \\ p_n \end{bmatrix}$$

For this reason we may write both  $P = (p_1, \dots, p_n) \in \mathbb{R}^n$  and  $\overrightarrow{OP} = [p_1 \cdots p_n]^T \in \mathbb{R}^n$ .

This definition is illustrated in the following picture for the special case of  $\mathbb{R}^3$ .

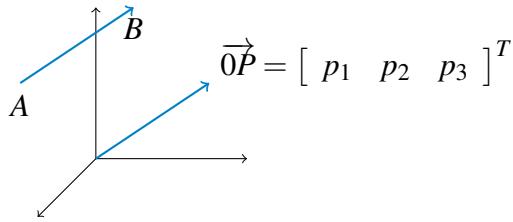


Thus every point  $P$  in  $\mathbb{R}^n$  determines its position vector  $\overrightarrow{OP}$ . Conversely, every such position vector  $\overrightarrow{OP}$  which has its tail at  $0$  and point at  $P$  determines the point  $P$  of  $\mathbb{R}^n$ .

Now suppose we are given two points,  $P, Q$  whose coordinates are  $(p_1, \dots, p_n)$  and  $(q_1, \dots, q_n)$  respectively. We can also determine the **position vector from  $P$  to  $Q$**  (also called the **vector from  $P$  to  $Q$** ) defined as follows.

$$\overrightarrow{PQ} = \begin{bmatrix} q_1 - p_1 \\ \vdots \\ q_n - p_n \end{bmatrix} = \overrightarrow{OQ} - \overrightarrow{OP}$$

Now, imagine taking a vector in  $\mathbb{R}^n$  and moving it around, always keeping it pointing in the same direction as shown in the following picture.



After moving it around, it is regarded as the same vector. Each vector,  $\vec{OP}$  and  $\vec{AB}$  has the same length (or magnitude) and direction. Therefore, they are equal.

Consider now the general definition for a vector in  $\mathbb{R}^n$ .

### Definition F.2: Vectors in $\mathbb{R}^n$

Let  $\mathbb{R}^n = \{(x_1, \dots, x_n) : x_j \in \mathbb{R} \text{ for } j = 1, \dots, n\}$ . Then,

$$\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$$

is called a **vector**. Vectors have both size (magnitude) and direction. The numbers  $x_j$  are called the **components** of  $\mathbf{x}$ .

Using this notation, we may use  $\mathbf{p}$  to denote the position vector of point  $P$ . Notice that in this context,  $\mathbf{p} = \vec{OP}$ . These notations may be used interchangeably.

You can think of the components of a vector as directions for obtaining the vector. Consider  $n = 3$ . Draw a vector with its tail at the point  $(0,0,0)$  and its tip at the point  $(a,b,c)$ . This vector is obtained by starting at  $(0,0,0)$ , moving parallel to the  $x$  axis to  $(a,0,0)$  and then from here, moving parallel to the  $y$  axis to  $(a,b,0)$  and finally parallel to the  $z$  axis to  $(a,b,c)$ . Observe that the same vector would result if you began at the point  $(d,e,f)$ , moved parallel to the  $x$  axis to  $(d+a,e,f)$ , then parallel to the  $y$  axis to  $(d+a,e+b,f)$ , and finally parallel to the  $z$  axis to  $(d+a,e+b,f+c)$ . Here, the vector would have its tail sitting at the point determined by  $A = (d,e,f)$  and its point at  $B = (d+a,e+b,f+c)$ . It is the **same vector** because it will point in the same direction and have the same length. It is like you took an actual arrow, and moved it from one location to another keeping it pointing the same direction.

We conclude this section with a brief discussion regarding notation. In previous sections, we have written vectors as columns, or  $n \times 1$  matrices. For convenience in this chapter we may write vectors as the transpose of row vectors, or  $1 \times n$  matrices. These are of course equivalent and we may move between both notations. Therefore, recognize that

$$\begin{bmatrix} 2 \\ 3 \end{bmatrix} = [2 \ 3]^T$$

Notice that two vectors  $\mathbf{u} = [u_1 \cdots u_n]^T$  and  $\mathbf{v} = [v_1 \cdots v_n]^T$  are equal if and only if all corresponding components are equal. Precisely,

$$\begin{aligned}\mathbf{u} &= \mathbf{v} \text{ if and only if} \\ u_j &= v_j \text{ for all } j = 1, \dots, n\end{aligned}$$

Thus  $[1 \ 2 \ 4]^T \in \mathbb{R}^3$  and  $[2 \ 1 \ 4]^T \in \mathbb{R}^3$  but  $[1 \ 2 \ 4]^T \neq [2 \ 1 \ 4]^T$  because, even though the same numbers are involved, the order of the numbers is different.

For the specific case of  $\mathbb{R}^3$ , there are three special vectors which we often use. They are given by

$$\mathbf{i} = [1 \ 0 \ 0]^T$$

$$\mathbf{j} = [0 \ 1 \ 0]^T$$

$$\mathbf{k} = [0 \ 0 \ 1]^T$$

We can write any vector  $\mathbf{u} = [u_1 \ u_2 \ u_3]^T$  as a linear combination of these vectors, written as  $\mathbf{u} = u_1\mathbf{i} + u_2\mathbf{j} + u_3\mathbf{k}$ . This notation will be used throughout this chapter.

## F.2 Algebra in $\mathbb{R}^n$

---

### Outcomes

- A. Understand vector addition and scalar multiplication, algebraically.
- B. Introduce the notion of linear combination of vectors.

Addition and scalar multiplication are two important algebraic operations done with vectors. Notice that these operations apply to vectors in  $\mathbb{R}^n$ , for any value of  $n$ . We will explore these operations in more detail in the following sections.

### F.2.1. Addition of Vectors in $\mathbb{R}^n$

---

Addition of vectors in  $\mathbb{R}^n$  is defined as follows.

**Definition F.3: Addition of Vectors in  $\mathbb{R}^n$** 

If  $\mathbf{u} = \begin{bmatrix} u_1 \\ \vdots \\ u_n \end{bmatrix}$ ,  $\mathbf{v} = \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix} \in \mathbb{R}^n$  then  $\mathbf{u} + \mathbf{v} \in \mathbb{R}^n$  and is defined by

$$\begin{aligned}\mathbf{u} + \mathbf{v} &= \begin{bmatrix} u_1 \\ \vdots \\ u_n \end{bmatrix} + \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix} \\ &= \begin{bmatrix} u_1 + v_1 \\ \vdots \\ u_n + v_n \end{bmatrix}\end{aligned}$$

To add vectors, we simply add corresponding components. Therefore, in order to add vectors, they must be the same size.

Addition of vectors satisfies some important properties which are outlined in the following theorem.

**Theorem F.4: Properties of Vector Addition**

The following properties hold for vectors  $\mathbf{u}, \mathbf{v}, \mathbf{w} \in \mathbb{R}^n$ .

- *The Commutative Law of Addition*

$$\mathbf{u} + \mathbf{v} = \mathbf{v} + \mathbf{u}$$

- *The Associative Law of Addition*

$$(\mathbf{u} + \mathbf{v}) + \mathbf{w} = \mathbf{u} + (\mathbf{v} + \mathbf{w})$$

- *The Existence of an Additive Identity*

$$\mathbf{u} + \mathbf{0} = \mathbf{u} \tag{6.1}$$

- *The Existence of an Additive Inverse*

$$\mathbf{u} + (-\mathbf{u}) = \mathbf{0}$$

The additive identity shown in equation 6.1 is also called the **zero vector**, the  $n \times 1$  vector in which all components are equal to 0. Further,  $-\mathbf{u}$  is simply the vector with all components having same value as those of  $\mathbf{u}$  but opposite sign; this is just  $(-1)\mathbf{u}$ . This will be made more explicit in the next section when we explore scalar multiplication of vectors. Note that subtraction is defined as  $\mathbf{u} - \mathbf{v} = \mathbf{u} + (-\mathbf{v})$ .

## F.2.2. Scalar Multiplication of Vectors in $\mathbb{R}^n$

Scalar multiplication of vectors in  $\mathbb{R}^n$  is defined as follows.

### Definition F.5: Scalar Multiplication of Vectors in $\mathbb{R}^n$

If  $\mathbf{u} \in \mathbb{R}^n$  and  $k \in \mathbb{R}$  is a scalar, then  $k\mathbf{u} \in \mathbb{R}^n$  is defined by

$$k\mathbf{u} = k \begin{bmatrix} u_1 \\ \vdots \\ u_n \end{bmatrix} = \begin{bmatrix} ku_1 \\ \vdots \\ ku_n \end{bmatrix}$$

Just as with addition, scalar multiplication of vectors satisfies several important properties. These are outlined in the following theorem.

### Theorem F.6: Properties of Scalar Multiplication

The following properties hold for vectors  $\mathbf{u}, \mathbf{v} \in \mathbb{R}^n$  and  $k, p$  scalars.

- The Distributive Law over Vector Addition

$$k(\mathbf{u} + \mathbf{v}) = k\mathbf{u} + k\mathbf{v}$$

- The Distributive Law over Scalar Addition

$$(k + p)\mathbf{u} = k\mathbf{u} + p\mathbf{u}$$

- The Associative Law for Scalar Multiplication

$$k(p\mathbf{u}) = (kp)\mathbf{u}$$

- Rule for Multiplication by 1

$$1\mathbf{u} = \mathbf{u}$$

We now present a useful notion you may have seen earlier combining vector addition and scalar multiplication

### Definition F.7: Linear Combination

A vector  $\mathbf{v}$  is said to be a **linear combination** of the vectors  $\mathbf{u}_1, \dots, \mathbf{u}_n$  if there exist scalars,  $a_1, \dots, a_n$  such that

$$\mathbf{v} = a_1\mathbf{u}_1 + \dots + a_n\mathbf{u}_n$$

For example,

$$3 \begin{bmatrix} -4 \\ 1 \\ 0 \end{bmatrix} + 2 \begin{bmatrix} -3 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} -18 \\ 3 \\ 2 \end{bmatrix}.$$

Thus we can say that

$$\mathbf{v} = \begin{bmatrix} -18 \\ 3 \\ 2 \end{bmatrix}$$

is a linear combination of the vectors

$$\mathbf{u}_1 = \begin{bmatrix} -4 \\ 1 \\ 0 \end{bmatrix} \text{ and } \mathbf{u}_2 = \begin{bmatrix} -3 \\ 0 \\ 1 \end{bmatrix}$$

## F.3 Geometric Meaning of Vector Addition

---

### Outcomes

- A. Understand vector addition, geometrically.

Recall that an element of  $\mathbb{R}^n$  is an ordered list of numbers. For the specific case of  $n = 2, 3$  this can be used to determine a point in two or three dimensional space. This point is specified relative to some coordinate axes.

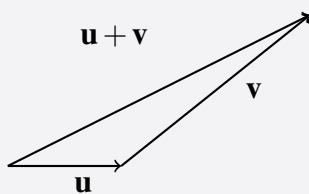
Consider the case  $n = 3$ . Recall that taking a vector and moving it around without changing its length or direction does not change the vector. This is important in the geometric representation of vector addition.

Suppose we have two vectors,  $\mathbf{u}$  and  $\mathbf{v}$  in  $\mathbb{R}^3$ . Each of these can be drawn geometrically by placing the tail of each vector at 0 and its point at  $(u_1, u_2, u_3)$  and  $(v_1, v_2, v_3)$  respectively. Suppose we slide the vector  $\mathbf{v}$  so that its tail sits at the point of  $\mathbf{u}$ . We know that this does not change the vector  $\mathbf{v}$ . Now, draw a new vector from the tail of  $\mathbf{u}$  to the point of  $\mathbf{v}$ . This vector is  $\mathbf{u} + \mathbf{v}$ .

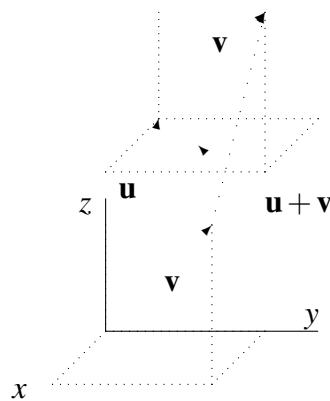
The geometric significance of vector addition in  $\mathbb{R}^n$  for any  $n$  is given in the following definition.

### Definition F.8: Geometry of Vector Addition

*Let  $\mathbf{u}$  and  $\mathbf{v}$  be two vectors. Slide  $\mathbf{v}$  so that the tail of  $\mathbf{v}$  is on the point of  $\mathbf{u}$ . Then draw the arrow which goes from the tail of  $\mathbf{u}$  to the point of  $\mathbf{v}$ . This arrow represents the vector  $\mathbf{u} + \mathbf{v}$ .*



This definition is illustrated in the following picture in which  $\mathbf{u} + \mathbf{v}$  is shown for the special case  $n = 3$ .



Notice the parallelogram created by  $\mathbf{u}$  and  $\mathbf{v}$  in the above diagram. Then  $\mathbf{u} + \mathbf{v}$  is the directed diagonal of the parallelogram determined by the two vectors  $\mathbf{u}$  and  $\mathbf{v}$ .

When you have a vector  $\mathbf{v}$ , its additive inverse  $-\mathbf{v}$  will be the vector which has the same magnitude as  $\mathbf{v}$  but the opposite direction. When one writes  $\mathbf{u} - \mathbf{v}$ , the meaning is  $\mathbf{u} + (-\mathbf{v})$  as with real numbers. The following example illustrates these definitions and conventions.

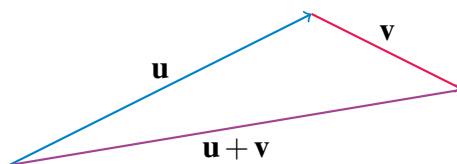
### Example F.9: Graphing Vector Addition

Consider the following picture of vectors  $\mathbf{u}$  and  $\mathbf{v}$ .

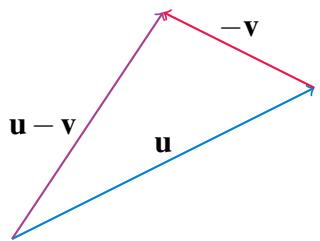


Sketch a picture of  $\mathbf{u} + \mathbf{v}$ ,  $\mathbf{u} - \mathbf{v}$ .

**Solution.** We will first sketch  $\mathbf{u} + \mathbf{v}$ . Begin by drawing  $\mathbf{u}$  and then at the point of  $\mathbf{u}$ , place the tail of  $\mathbf{v}$  as shown. Then  $\mathbf{u} + \mathbf{v}$  is the vector which results from drawing a vector from the tail of  $\mathbf{u}$  to the tip of  $\mathbf{v}$ .



Next consider  $\mathbf{u} - \mathbf{v}$ . This means  $\mathbf{u} + (-\mathbf{v})$ . From the above geometric description of vector addition,  $-\mathbf{v}$  is the vector which has the same length but which points in the opposite direction to  $\mathbf{v}$ . Here is a picture.



## F.4 Length of a Vector

---

### Outcomes

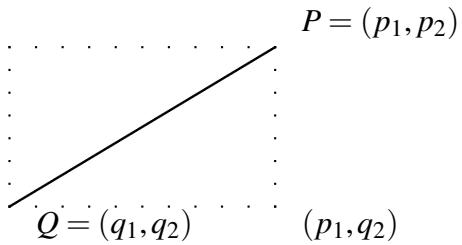
- A. Find the length of a vector and the distance between two points in  $\mathbb{R}^n$ .
- B. Find the corresponding unit vector to a vector in  $\mathbb{R}^n$ .

In this section, we explore what is meant by the length of a vector in  $\mathbb{R}^n$ . We develop this concept by first looking at the distance between two points in  $\mathbb{R}^n$ .

First, we will consider the concept of distance for  $\mathbb{R}$ , that is, for points in  $\mathbb{R}^1$ . Here, the distance between two points  $P$  and  $Q$  is given by the absolute value of their difference. We denote the distance between  $P$  and  $Q$  by  $d(P, Q)$  which is defined as

$$d(P, Q) = \sqrt{(P - Q)^2} \quad (6.1)$$

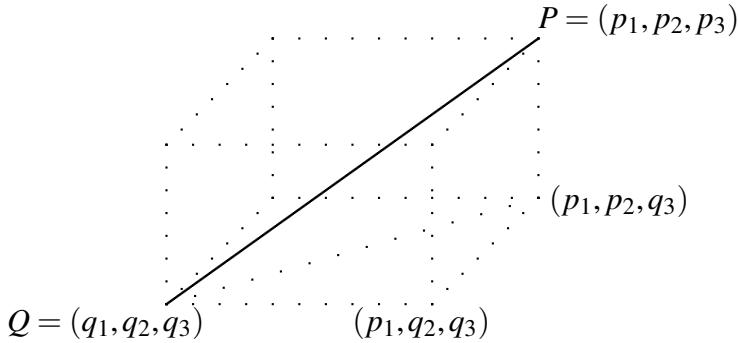
Consider now the case for  $n = 2$ , demonstrated by the following picture.



There are two points  $P = (p_1, p_2)$  and  $Q = (q_1, q_2)$  in the plane. The distance between these points is shown in the picture as a solid line. Notice that this line is the hypotenuse of a right triangle which is half of the rectangle shown in dotted lines. We want to find the length of this hypotenuse which will give the distance between the two points. Note the lengths of the sides of this triangle are  $|p_1 - q_1|$  and  $|p_2 - q_2|$ , the absolute value of the difference in these values. Therefore, the Pythagorean Theorem implies the length of the hypotenuse (and thus the distance between  $P$  and  $Q$ ) equals

$$\left( |p_1 - q_1|^2 + |p_2 - q_2|^2 \right)^{1/2} = \left( (p_1 - q_1)^2 + (p_2 - q_2)^2 \right)^{1/2} \quad (6.2)$$

Now suppose  $n = 3$  and let  $P = (p_1, p_2, p_3)$  and  $Q = (q_1, q_2, q_3)$  be two points in  $\mathbb{R}^3$ . Consider the following picture in which the solid line joins the two points and a dotted line joins the points  $(q_1, q_2, q_3)$  and  $(p_1, p_2, q_3)$ .



Here, we need to use Pythagorean Theorem twice in order to find the length of the solid line. First, by the Pythagorean Theorem, the length of the dotted line joining  $(q_1, q_2, q_3)$  and  $(p_1, p_2, q_3)$  equals

$$\left( (p_1 - q_1)^2 + (p_2 - q_2)^2 \right)^{1/2}$$

while the length of the line joining  $(p_1, p_2, q_3)$  to  $(p_1, p_2, p_3)$  is just  $|p_3 - q_3|$ . Therefore, by the Pythagorean Theorem again, the length of the line joining the points  $P = (p_1, p_2, p_3)$  and  $Q = (q_1, q_2, q_3)$  equals

$$\begin{aligned} & \left( \left( \left( (p_1 - q_1)^2 + (p_2 - q_2)^2 \right)^{1/2} \right)^2 + (p_3 - q_3)^2 \right)^{1/2} \\ &= \left( (p_1 - q_1)^2 + (p_2 - q_2)^2 + (p_3 - q_3)^2 \right)^{1/2} \end{aligned} \quad (6.3)$$

This discussion motivates the following definition for the distance between points in  $\mathbb{R}^n$ .

#### Definition F.10: Distance Between Points

Let  $P = (p_1, \dots, p_n)$  and  $Q = (q_1, \dots, q_n)$  be two points in  $\mathbb{R}^n$ . Then the distance between these points is defined as

$$\text{distance between } P \text{ and } Q = d(P, Q) = \left( \sum_{k=1}^n |p_k - q_k|^2 \right)^{1/2}$$

This is called the **distance formula**. We may also write  $|P - Q|$  as the distance between  $P$  and  $Q$ .

From the above discussion, you can see that Definition F.10 holds for the special cases  $n = 1, 2, 3$ , as in Equations 6.1, 6.2, 6.3. In the following example, we use Definition F.10 to find the distance between two points in  $\mathbb{R}^4$ .

#### Example F.11: Distance Between Points

Find the distance between the points  $P$  and  $Q$  in  $\mathbb{R}^4$ , where  $P$  and  $Q$  are given by

$$P = (1, 2, -4, 6)$$

and

$$Q = (2, 3, -1, 0)$$

**Solution.** We will use the formula given in Definition F.10 to find the distance between  $P$  and  $Q$ . Use the distance formula and write

$$d(P, Q) = \left( (1-2)^2 + (2-3)^2 + (-4 - (-1))^2 + (6-0)^2 \right)^{\frac{1}{2}} = 47$$

Therefore,  $d(P, Q) = \sqrt{47}$ .



There are certain properties of the distance between points which are important in our study. These are outlined in the following theorem.

### Theorem F.12: Properties of Distance

Let  $P$  and  $Q$  be points in  $\mathbb{R}^n$ , and let the distance between them,  $d(P, Q)$ , be given as in Definition F.10. Then, the following properties hold .

- $d(P, Q) = d(Q, P)$
- $d(P, Q) \geq 0$ , and equals 0 exactly when  $P = Q$ .

There are many applications of the concept of distance. For instance, given two points, we can ask what collection of points are all the same distance between the given points. This is explored in the following example.

### Example F.13: The Plane Between Two Points

Describe the points in  $\mathbb{R}^3$  which are at the same distance between  $(1, 2, 3)$  and  $(0, 1, 2)$ .

**Solution.** Let  $P = (p_1, p_2, p_3)$  be such a point. Therefore,  $P$  is the same distance from  $(1, 2, 3)$  and  $(0, 1, 2)$ . Then by Definition F.10,

$$\sqrt{(p_1 - 1)^2 + (p_2 - 2)^2 + (p_3 - 3)^2} = \sqrt{(p_1 - 0)^2 + (p_2 - 1)^2 + (p_3 - 2)^2}$$

Squaring both sides we obtain

$$(p_1 - 1)^2 + (p_2 - 2)^2 + (p_3 - 3)^2 = p_1^2 + (p_2 - 1)^2 + (p_3 - 2)^2$$

and so

$$p_1^2 - 2p_1 + 14 + p_2^2 - 4p_2 + p_3^2 - 6p_3 = p_1^2 + p_2^2 - 2p_2 + 5 + p_3^2 - 4p_3$$

Simplifying, this becomes

$$-2p_1 + 14 - 4p_2 - 6p_3 = -2p_2 + 5 - 4p_3$$

which can be written as

$$2p_1 + 2p_2 + 2p_3 = -9 \tag{6.4}$$

Therefore, the points  $P = (p_1, p_2, p_3)$  which are the same distance from each of the given points form a plane whose equation is given by 6.4. ♠

We can now use our understanding of the distance between two points to define what is meant by the length of a vector. Consider the following definition.

### Definition F.14: Length of a Vector

Let  $\mathbf{u} = [u_1 \cdots u_n]^T$  be a vector in  $\mathbb{R}^n$ . Then, the length of  $\mathbf{u}$ , written  $\|\mathbf{u}\|$  is given by

$$\|\mathbf{u}\| = \sqrt{u_1^2 + \cdots + u_n^2}$$

This definition corresponds to Definition F.10, if you consider the vector  $\mathbf{u}$  to have its tail at the point  $0 = (0, \dots, 0)$  and its tip at the point  $U = (u_1, \dots, u_n)$ . Then the length of  $\mathbf{u}$  is equal to the distance between  $0$  and  $U$ ,  $d(0, U)$ . In general,  $d(P, Q) = \|\overrightarrow{PQ}\|$ .

Consider Example F.11. By Definition F.14, we could also find the distance between  $P$  and  $Q$  as the length of the vector connecting them. Hence, if we were to draw a vector  $\overrightarrow{PQ}$  with its tail at  $P$  and its point at  $Q$ , this vector would have length equal to  $\sqrt{47}$ .

We conclude this section with a new definition for the special case of vectors of length 1.

### Definition F.15: Unit Vector

Let  $\mathbf{u}$  be a vector in  $\mathbb{R}^n$ . Then, we call  $\mathbf{u}$  a **unit vector** if it has length 1, that is if

$$\|\mathbf{u}\| = 1$$

Let  $\mathbf{v}$  be a vector in  $\mathbb{R}^n$ . Then, the vector  $\mathbf{u}$  which has the same direction as  $\mathbf{v}$  but length equal to 1 is the corresponding unit vector of  $\mathbf{v}$ . This vector is given by

$$\mathbf{u} = \frac{1}{\|\mathbf{v}\|} \mathbf{v}$$

We often use the term **normalize** to refer to this process. When we **normalize** a vector, we find the corresponding unit vector of length 1. Consider the following example.

### Example F.16: Finding a Unit Vector

Let  $\mathbf{v}$  be given by

$$\mathbf{v} = [ \begin{array}{ccc} 1 & -3 & 4 \end{array}]^T$$

Find the unit vector  $\mathbf{u}$  which has the same direction as  $\mathbf{v}$ .

**Solution.** We will use Definition F.15 to solve this. Therefore, we need to find the length of  $\mathbf{v}$  which, by Definition F.14 is given by

$$\|\mathbf{v}\| = \sqrt{v_1^2 + v_2^2 + v_3^2}$$

Using the corresponding values we find that

$$\begin{aligned}\|\mathbf{v}\| &= \sqrt{1^2 + (-3)^2 + 4^2} \\ &= \sqrt{1 + 9 + 16} \\ &= \sqrt{26}\end{aligned}$$

In order to find  $\mathbf{u}$ , we divide  $\mathbf{v}$  by  $\sqrt{26}$ . The result is

$$\begin{aligned}\mathbf{u} &= \frac{1}{\|\mathbf{v}\|} \mathbf{v} \\ &= \frac{1}{\sqrt{26}} [1 \ -3 \ 4]^T \\ &= \left[ \frac{1}{\sqrt{26}} \ -\frac{3}{\sqrt{26}} \ \frac{4}{\sqrt{26}} \right]^T\end{aligned}$$

You can verify using the Definition F.14 that  $\|\mathbf{u}\| = 1$ .



## F.5 Geometric Meaning of Scalar Multiplication

---

### Outcomes

A. Understand scalar multiplication, geometrically.

Recall that the point  $P = (p_1, p_2, p_3)$  determines a vector  $\mathbf{p}$  from 0 to  $P$ . The length of  $\mathbf{p}$ , denoted  $\|\mathbf{p}\|$ , is equal to  $\sqrt{p_1^2 + p_2^2 + p_3^2}$  by Definition F.10.

Now suppose we have a vector  $\mathbf{u} = [u_1 \ u_2 \ u_3]^T$  and we multiply  $\mathbf{u}$  by a scalar  $k$ . By Definition F.5,  $k\mathbf{u} = [ku_1 \ ku_2 \ ku_3]^T$ . Then, by using Definition F.10, the length of this vector is given by

$$\sqrt{(ku_1)^2 + (ku_2)^2 + (ku_3)^2} = |k| \sqrt{u_1^2 + u_2^2 + u_3^2}$$

Thus the following holds.

$$\|k\mathbf{u}\| = |k| \|\mathbf{u}\|$$

In other words, multiplication by a scalar magnifies or shrinks the length of the vector by a factor of  $|k|$ . If  $|k| > 1$ , the length of the resulting vector will be magnified. If  $|k| < 1$ , the length of the resulting vector will shrink. Remember that by the definition of the absolute value,  $|k| > 0$ .

What about the direction? Draw a picture of  $\mathbf{u}$  and  $k\mathbf{u}$  where  $k$  is negative. Notice that this causes the resulting vector to point in the opposite direction while if  $k > 0$  it preserves the direction the vector points. Therefore the direction can either reverse, if  $k < 0$ , or remain preserved, if  $k > 0$ .

Consider the following example.

**Example F.17: Graphing Scalar Multiplication**

Consider the vectors  $\mathbf{u}$  and  $\mathbf{v}$  drawn below.



Draw  $-\mathbf{u}$ ,  $2\mathbf{v}$ , and  $-\frac{1}{2}\mathbf{v}$ .

**Solution.**

In order to find  $-\mathbf{u}$ , we preserve the length of  $\mathbf{u}$  and simply reverse the direction. For  $2\mathbf{v}$ , we double the length of  $\mathbf{v}$ , while preserving the direction. Finally  $-\frac{1}{2}\mathbf{v}$  is found by taking half the length of  $\mathbf{v}$  and reversing the direction. These vectors are shown in the following diagram.



Now that we have studied both vector addition and scalar multiplication, we can combine the two actions. Recall Definition F.7 of linear combinations of column matrices. We can apply this definition to vectors in  $\mathbb{R}^n$ . A linear combination of vectors in  $\mathbb{R}^n$  is a sum of vectors multiplied by scalars.

In the following example, we examine the geometric meaning of this concept.

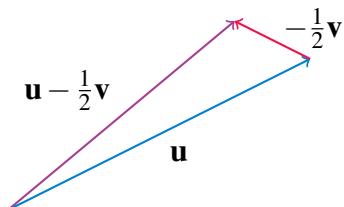
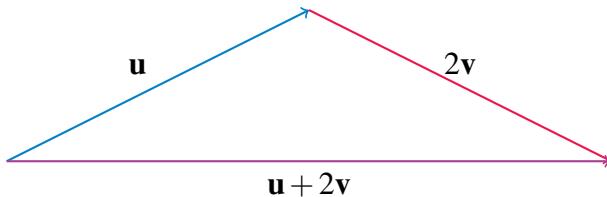
**Example F.18: Graphing a Linear Combination of Vectors**

Consider the following picture of the vectors  $\mathbf{u}$  and  $\mathbf{v}$



Sketch a picture of  $\mathbf{u} + 2\mathbf{v}$ ,  $\mathbf{u} - \frac{1}{2}\mathbf{v}$ .

**Solution.** The two vectors are shown below.



## F.6 The Dot Product

---

### Outcomes

A. Compute the dot product of vectors, and use this to compute vector projections.

### F.6.1. The Dot Product

---

There are two ways of multiplying vectors which are of great importance in applications. The first of these is called the **dot product**. When we take the dot product of vectors, the result is a scalar. For this reason, the dot product is also called the **scalar product** and sometimes the **inner product**. The definition is as follows.

#### Definition F.19: Dot Product

Let  $\mathbf{u}, \mathbf{v}$  be two vectors in  $\mathbb{R}^n$ . Then we define the **dot product**  $\mathbf{u} \bullet \mathbf{v}$  as

$$\mathbf{u} \bullet \mathbf{v} = \sum_{k=1}^n u_k v_k$$

The dot product  $\mathbf{u} \bullet \mathbf{v}$  is sometimes denoted as  $(\mathbf{u}, \mathbf{v})$  where a comma replaces  $\bullet$ . It can also be written as  $\langle \mathbf{u}, \mathbf{v} \rangle$ . If we write the vectors as column or row matrices, it is equal to the matrix product  $\mathbf{v}\mathbf{w}^T$ .

Consider the following example.

**Example F.20: Compute a Dot Product**

Find  $\mathbf{u} \bullet \mathbf{v}$  for

$$\mathbf{u} = \begin{bmatrix} 1 \\ 2 \\ 0 \\ -1 \end{bmatrix}, \mathbf{v} = \begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \end{bmatrix}$$

**Solution.** By Definition F.19, we must compute

$$\mathbf{u} \bullet \mathbf{v} = \sum_{k=1}^4 u_k v_k$$

This is given by

$$\begin{aligned} \mathbf{u} \bullet \mathbf{v} &= (1)(0) + (2)(1) + (0)(2) + (-1)(3) \\ &= 0 + 2 + 0 - 3 \\ &= -1 \end{aligned}$$



With this definition, there are several important properties satisfied by the dot product.

**Proposition F.21: Properties of the Dot Product**

Let  $k$  and  $p$  denote scalars and  $\mathbf{u}, \mathbf{v}, \mathbf{w}$  denote vectors. Then the dot product  $\mathbf{u} \bullet \mathbf{v}$  satisfies the following properties.

- $\mathbf{u} \bullet \mathbf{v} = \mathbf{v} \bullet \mathbf{u}$
- $\mathbf{u} \bullet \mathbf{u} \geq 0$  and equals zero if and only if  $\mathbf{u} = 0$
- $(k\mathbf{u} + p\mathbf{v}) \bullet \mathbf{w} = k(\mathbf{u} \bullet \mathbf{w}) + p(\mathbf{v} \bullet \mathbf{w})$
- $\mathbf{u} \bullet (k\mathbf{v} + p\mathbf{w}) = k(\mathbf{u} \bullet \mathbf{v}) + p(\mathbf{u} \bullet \mathbf{w})$
- $\|\mathbf{u}\|^2 = \mathbf{u} \bullet \mathbf{u}$

The proof is left as an exercise. This proposition tells us that we can also use the dot product to find the length of a vector.

**Example F.22: Length of a Vector**

*Find the length of*

$$\mathbf{u} = \begin{bmatrix} 2 \\ 1 \\ 4 \\ 2 \end{bmatrix}$$

*That is, find  $\|\mathbf{u}\|$ .*

**Solution.** By Proposition F.21,  $\|\mathbf{u}\|^2 = \mathbf{u} \bullet \mathbf{u}$ . Therefore,  $\|\mathbf{u}\| = \sqrt{\mathbf{u} \bullet \mathbf{u}}$ . First, compute  $\mathbf{u} \bullet \mathbf{u}$ .

This is given by

$$\begin{aligned} \mathbf{u} \bullet \mathbf{u} &= (2)(2) + (1)(1) + (4)(4) + (2)(2) \\ &= 4 + 1 + 16 + 4 \\ &= 25 \end{aligned}$$

Then,

$$\begin{aligned} \|\mathbf{u}\| &= \sqrt{\mathbf{u} \bullet \mathbf{u}} \\ &= \sqrt{25} \\ &= 5 \end{aligned}$$



You may wish to compare this to our previous definition of length, given in Definition F.14.

The **Cauchy Schwarz inequality** is a fundamental inequality satisfied by the dot product. It is given in the following theorem.

**Theorem F.23: Cauchy Schwarz Inequality**

*The dot product satisfies the inequality*

$$|\mathbf{u} \bullet \mathbf{v}| \leq \|\mathbf{u}\| \|\mathbf{v}\| \tag{6.1}$$

*Furthermore equality is obtained if and only if one of  $\mathbf{u}$  or  $\mathbf{v}$  is a scalar multiple of the other.*

Notice that this proof was based only on the properties of the dot product listed in Proposition F.21. This means that whenever an operation satisfies these properties, the Cauchy Schwarz inequality holds. There are many other instances of these properties besides vectors in  $\mathbb{R}^n$ .

The Cauchy Schwarz inequality provides another proof of the **triangle inequality** for distances in  $\mathbb{R}^n$ .

**Theorem F.24: Triangle Inequality**

For  $\mathbf{u}, \mathbf{v} \in \mathbb{R}^n$

$$\|\mathbf{u} + \mathbf{v}\| \leq \|\mathbf{u}\| + \|\mathbf{v}\| \quad (6.2)$$

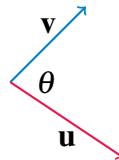
and equality holds if and only if one of the vectors is a non-negative scalar multiple of the other.

Also

$$\|\|\mathbf{u}\| - \|\mathbf{v}\|\| \leq \|\mathbf{u} - \mathbf{v}\| \quad (6.3)$$

**F.6.2. The Geometric Significance of the Dot Product**

Given two vectors,  $\mathbf{u}$  and  $\mathbf{v}$ , the **included angle** is the angle between these two vectors which is given by  $\theta$  such that  $0 \leq \theta \leq \pi$ . The dot product can be used to determine the included angle between two vectors. Consider the following picture where  $\theta$  gives the included angle.

**Proposition F.25: The Dot Product and the Included Angle**

Let  $\mathbf{u}$  and  $\mathbf{v}$  be two vectors in  $\mathbb{R}^n$ , and let  $\theta$  be the included angle. Then the following equation holds.

$$\mathbf{u} \cdot \mathbf{v} = \|\mathbf{u}\| \|\mathbf{v}\| \cos \theta$$

In words, the dot product of two vectors equals the product of the magnitude (or length) of the two vectors multiplied by the cosine of the included angle. Note this gives a geometric description of the dot product which does not depend explicitly on the coordinates of the vectors.

Consider the following example.

**Example F.26: Find the Angle Between Two Vectors**

Find the angle between the vectors given by

$$\mathbf{u} = \begin{bmatrix} 2 \\ 1 \\ -1 \end{bmatrix}, \mathbf{v} = \begin{bmatrix} 3 \\ 4 \\ 1 \end{bmatrix}$$

**Solution.** By Proposition F.25,

$$\mathbf{u} \cdot \mathbf{v} = \|\mathbf{u}\| \|\mathbf{v}\| \cos \theta$$

Hence,

$$\cos \theta = \frac{\mathbf{u} \bullet \mathbf{v}}{\|\mathbf{u}\| \|\mathbf{v}\|}$$

First, we can compute  $\mathbf{u} \bullet \mathbf{v}$ . By Definition F.19, this equals

$$\mathbf{u} \bullet \mathbf{v} = (2)(3) + (1)(4) + (-1)(1) = 9$$

Then,

$$\begin{aligned}\|\mathbf{u}\| &= \sqrt{(2)(2) + (1)(1) + (1)(1)} = \sqrt{6} \\ \|\mathbf{v}\| &= \sqrt{(3)(3) + (4)(4) + (1)(1)} = \sqrt{26}\end{aligned}$$

Therefore, the cosine of the included angle equals

$$\cos \theta = \frac{9}{\sqrt{26}\sqrt{6}} = 0.7205766\dots$$

With the cosine known, the angle can be determined by computing the inverse cosine of that angle, giving approximately  $\theta = 0.76616$  radians. ♠

Another application of the geometric description of the dot product is in finding the angle between two lines. Typically one would assume that the lines intersect. In some situations, however, it may make sense to ask this question when the lines do not intersect, such as the angle between two object trajectories. In any case we understand it to mean the smallest angle between (any of) their direction vectors. The only subtlety here is that if  $\mathbf{u}$  is a direction vector for a line, then so is any multiple  $k\mathbf{u}$ , and thus we will find complementary angles among all angles between direction vectors for two lines, and we simply take the smaller of the two.

### Example F.27: Find the Angle Between Two Lines

*Find the angle between the two lines*

$$L_1 : \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 0 \end{bmatrix} + t \begin{bmatrix} -1 \\ 1 \\ 2 \end{bmatrix}$$

and

$$L_2 : \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 0 \\ 4 \\ -3 \end{bmatrix} + s \begin{bmatrix} 2 \\ 1 \\ -1 \end{bmatrix}$$

**Solution.** You can verify that these lines do not intersect, but as discussed above this does not matter and we simply find the smallest angle between any directions vectors for these lines.

To do so we first find the angle between the direction vectors given above:

$$\mathbf{u} = \begin{bmatrix} -1 \\ 1 \\ 2 \end{bmatrix}, \mathbf{v} = \begin{bmatrix} 2 \\ 1 \\ -1 \end{bmatrix}$$

In order to find the angle, we solve the following equation for  $\theta$

$$\mathbf{u} \bullet \mathbf{v} = \|\mathbf{u}\| \|\mathbf{v}\| \cos \theta$$

to obtain  $\cos \theta = -\frac{1}{2}$  and since we choose included angles between 0 and  $\pi$  we obtain  $\theta = \frac{2\pi}{3}$ .

Now the angles between any two direction vectors for these lines will either be  $\frac{2\pi}{3}$  or its complement  $\phi = \pi - \frac{2\pi}{3} = \frac{\pi}{3}$ . We choose the smaller angle, and therefore conclude that the angle between the two lines is  $\frac{\pi}{3}$ . ♠

We can also use Proposition F.25 to compute the dot product of two vectors.

### Example F.28: Using Geometric Description to Find a Dot Product

Let  $\mathbf{u}, \mathbf{v}$  be vectors with  $\|\mathbf{u}\| = 3$  and  $\|\mathbf{v}\| = 4$ . Suppose the angle between  $\mathbf{u}$  and  $\mathbf{v}$  is  $\pi/3$ . Find  $\mathbf{u} \bullet \mathbf{v}$ .

**Solution.** From the geometric description of the dot product in Proposition F.25

$$\mathbf{u} \bullet \mathbf{v} = (3)(4) \cos(\pi/3) = 3 \times 4 \times 1/2 = 6$$



Two nonzero vectors are said to be **perpendicular**, sometimes also called **orthogonal**, if the included angle is  $\pi/2$  radians ( $90^\circ$ ).

Consider the following proposition.

### Proposition F.29: Perpendicular Vectors

Let  $\mathbf{u}$  and  $\mathbf{v}$  be nonzero vectors in  $\mathbb{R}^n$ . Then,  $\mathbf{u}$  and  $\mathbf{v}$  are said to be **perpendicular** exactly when

$$\mathbf{u} \bullet \mathbf{v} = 0$$

Consider the following example.

### Example F.30: Determine if Two Vectors are Perpendicular

Determine whether the two vectors,

$$\mathbf{u} = \begin{bmatrix} 2 \\ 1 \\ -1 \end{bmatrix}, \mathbf{v} = \begin{bmatrix} 1 \\ 3 \\ 5 \end{bmatrix}$$

are perpendicular.

**Solution.** In order to determine if these two vectors are perpendicular, we compute the dot product. This is given by

$$\mathbf{u} \bullet \mathbf{v} = (2)(1) + (1)(3) + (-1)(5) = 0$$

Therefore, by Proposition F.29 these two vectors are perpendicular. ♠



# G. Spectral Theory

## G.1 Eigenvalues and Eigenvectors of a Matrix

### Outcomes

- A. *Describe eigenvalues geometrically and algebraically.*
- B. *Find eigenvalues and eigenvectors for a square matrix.*

Spectral Theory refers to the study of eigenvalues and eigenvectors of a matrix. It is of fundamental importance in many areas and is the subject of our study for this chapter.

### G.1.1. Definition of Eigenvectors and Eigenvalues

In this section, we will work with the entire set of complex numbers, denoted by  $\mathbb{C}$ . Recall that the real numbers,  $\mathbb{R}$  are contained in the complex numbers, so the discussions in this section apply to both real and complex numbers.

To illustrate the idea behind what will be discussed, consider the following example.

#### Example G.1: Eigenvectors and Eigenvalues

Let

$$A = \begin{bmatrix} 0 & 5 & -10 \\ 0 & 22 & 16 \\ 0 & -9 & -2 \end{bmatrix}$$

Compute the product  $AX$  for

$$X = \begin{bmatrix} 5 \\ -4 \\ 3 \end{bmatrix}, X = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

What do you notice about  $AX$  in each of these products?

**Solution.** First, compute  $AX$  for

$$X = \begin{bmatrix} 5 \\ -4 \\ 3 \end{bmatrix}$$

This product is given by

$$AX = \begin{bmatrix} 0 & 5 & -10 \\ 0 & 22 & 16 \\ 0 & -9 & -2 \end{bmatrix} \begin{bmatrix} -5 \\ -4 \\ 3 \end{bmatrix} = \begin{bmatrix} -50 \\ -40 \\ 30 \end{bmatrix} = 10 \begin{bmatrix} -5 \\ -4 \\ 3 \end{bmatrix}$$

In this case, the product  $AX$  resulted in a vector which is equal to 10 times the vector  $X$ . In other words,  $AX = 10X$ .

Let's see what happens in the next product. Compute  $AX$  for the vector

$$X = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

This product is given by

$$AX = \begin{bmatrix} 0 & 5 & -10 \\ 0 & 22 & 16 \\ 0 & -9 & -2 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} = 0 \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

In this case, the product  $AX$  resulted in a vector equal to 0 times the vector  $X$ ,  $AX = 0X$ .

Perhaps this matrix is such that  $AX$  results in  $kX$ , for every vector  $X$ . However, consider

$$\begin{bmatrix} 0 & 5 & -10 \\ 0 & 22 & 16 \\ 0 & -9 & -2 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} -5 \\ 38 \\ -11 \end{bmatrix}$$

In this case,  $AX$  did not result in a vector of the form  $kX$  for some scalar  $k$ . ♠

There is something special about the first two products calculated in Example G.1. Notice that for each,  $AX = kX$  where  $k$  is some scalar. When this equation holds for some  $X$  and  $k$ , we call the scalar  $k$  an **eigenvalue** of  $A$ . We often use the special symbol  $\lambda$  instead of  $k$  when referring to eigenvalues. In Example G.1, the values 10 and 0 are eigenvalues for the matrix  $A$  and we can label these as  $\lambda_1 = 10$  and  $\lambda_2 = 0$ .

When  $AX = \lambda X$  for some  $X \neq 0$ , we call such an  $X$  an **eigenvector** of the matrix  $A$ . The eigenvectors of  $A$  are associated to an eigenvalue. Hence, if  $\lambda_1$  is an eigenvalue of  $A$  and  $AX = \lambda_1 X$ , we can label this eigenvector as  $X_1$ . Note again that in order to be an eigenvector,  $X$  must be nonzero.

There is also a geometric significance to eigenvectors. When you have a **nonzero** vector which, when multiplied by a matrix results in another vector which is parallel to the first or equal to **0**, this vector is called an eigenvector of the matrix. This is the meaning when the vectors are in  $\mathbb{R}^n$ .

The formal definition of eigenvalues and eigenvectors is as follows.

**Definition G.2: Eigenvalues and Eigenvectors**

Let  $A$  be an  $n \times n$  matrix and let  $X \in \mathbb{C}^n$  be a **nonzero vector** for which

$$AX = \lambda X \quad (7.1)$$

for some scalar  $\lambda$ . Then  $\lambda$  is called an **eigenvalue** of the matrix  $A$  and  $X$  is called an **eigenvector** of  $A$  associated with  $\lambda$ , or a  $\lambda$ -eigenvector of  $A$ .

The set of all eigenvalues of an  $n \times n$  matrix  $A$  is denoted by  $\sigma(A)$  and is referred to as the **spectrum** of  $A$ .

The eigenvectors of a matrix  $A$  are those vectors  $X$  for which multiplication by  $A$  results in a vector in the same direction or opposite direction to  $X$ . Since the zero vector  $0$  has no direction this would make no sense for the zero vector. As noted above,  $0$  is never allowed to be an eigenvector.

Let's look at eigenvectors in more detail. Suppose  $X$  satisfies 7.1. Then

$$\begin{aligned} AX - \lambda X &= 0 \\ \text{or} \\ (A - \lambda I)X &= 0 \end{aligned}$$

for some  $X \neq 0$ . Equivalently you could write  $(\lambda I - A)X = 0$ , which is more commonly used. Hence, when we are looking for eigenvectors, we are looking for nontrivial solutions to this homogeneous system of equations!

Recall that the solutions to a homogeneous system of equations consist of basic solutions, and the linear combinations of those basic solutions. In this context, we call the basic solutions of the equation  $(\lambda I - A)X = 0$  **basic eigenvectors**. It follows that any (nonzero) linear combination of basic eigenvectors is again an eigenvector.

Suppose the matrix  $(\lambda I - A)$  is invertible, so that  $(\lambda I - A)^{-1}$  exists. Then the following equation would be true.

$$\begin{aligned} X &= IX \\ &= ((\lambda I - A)^{-1}(\lambda I - A))X \\ &= (\lambda I - A)^{-1}((\lambda I - A)X) \\ &= (\lambda I - A)^{-1}0 \\ &= 0 \end{aligned}$$

This claims that  $X = 0$ . However, we have required that  $X \neq 0$ . Therefore  $(\lambda I - A)$  cannot have an inverse!

Recall that if a matrix is not invertible, then its determinant is equal to 0. Therefore we can conclude that

$$\det(\lambda I - A) = 0 \quad (7.2)$$

Note that this is equivalent to  $\det(A - \lambda I) = 0$ .

The expression  $\det(xI - A)$  is a polynomial (in the variable  $x$ ) called the **characteristic polynomial of  $A$** , and  $\det(xI - A) = 0$  is called the **characteristic equation**. For this reason we may also refer to the eigenvalues of  $A$  as **characteristic values**, but the former is often used for historical reasons.

The following theorem claims that the roots of the characteristic polynomial are the eigenvalues of  $A$ . Thus when 7.2 holds,  $A$  has a nonzero eigenvector.

### Theorem G.3: The Existence of an Eigenvector

*Let  $A$  be an  $n \times n$  matrix and suppose  $\det(\lambda I - A) = 0$  for some  $\lambda \in \mathbb{C}$ .*

*Then  $\lambda$  is an eigenvalue of  $A$  and thus there exists a nonzero vector  $X \in \mathbb{C}^n$  such that  $AX = \lambda X$ .*

## G.1.2. Finding Eigenvectors and Eigenvalues

Now that eigenvalues and eigenvectors have been defined, we will study how to find them for a matrix  $A$ .

First, consider the following definition.

### Definition G.4: Multiplicity of an Eigenvalue

*Let  $A$  be an  $n \times n$  matrix with characteristic polynomial given by  $\det(xI - A)$ . Then, the multiplicity of an eigenvalue  $\lambda$  of  $A$  is the number of times  $\lambda$  occurs as a root of that characteristic polynomial.*

For example, suppose the characteristic polynomial of  $A$  is given by  $(x - 2)^2$ . Solving for the roots of this polynomial, we set  $(x - 2)^2 = 0$  and solve for  $x$ . We find that  $\lambda = 2$  is a root that occurs twice. Hence, in this case,  $\lambda = 2$  is an eigenvalue of  $A$  of multiplicity equal to 2.

We will now look at how to find the eigenvalues and eigenvectors for a matrix  $A$  in detail. The steps used are summarized in the following procedure.

### Procedure G.5: Finding Eigenvalues and Eigenvectors

*Let  $A$  be an  $n \times n$  matrix.*

1. First, find the eigenvalues  $\lambda$  of  $A$  by solving the equation  $\det(xI - A) = 0$ .
2. For each  $\lambda$ , find the basic eigenvectors  $X \neq 0$  by finding the basic solutions to  $(\lambda I - A)X = 0$ .

*To verify your work, make sure that  $AX = \lambda X$  for each  $\lambda$  and associated eigenvector  $X$ .*

We will explore these steps further in the following example.

### Example G.6: Find the Eigenvalues and Eigenvectors

*Let  $A = \begin{bmatrix} -5 & 2 \\ -7 & 4 \end{bmatrix}$ . Find its eigenvalues and eigenvectors.*

**Solution.** We will use Procedure G.5. First we find the eigenvalues of  $A$  by solving the equation

$$\det(xI - A) = 0$$

This gives

$$\begin{aligned}\det\left(x\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} - \begin{bmatrix} -5 & 2 \\ -7 & 4 \end{bmatrix}\right) &= 0 \\ \det\begin{bmatrix} x+5 & -2 \\ 7 & x-4 \end{bmatrix} &= 0\end{aligned}$$

Computing the determinant as usual, the result is

$$x^2 + x - 6 = 0$$

Solving this equation, we find that  $\lambda_1 = 2$  and  $\lambda_2 = -3$ .

Now we need to find the basic eigenvectors for each  $\lambda$ . First we will find the eigenvectors for  $\lambda_1 = 2$ . We wish to find all vectors  $X \neq 0$  such that  $AX = 2X$ . These are the solutions to  $(2I - A)X = 0$ .

$$\begin{aligned}(2\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} - \begin{bmatrix} -5 & 2 \\ -7 & 4 \end{bmatrix}) \begin{bmatrix} x \\ y \end{bmatrix} &= \begin{bmatrix} 0 \\ 0 \end{bmatrix} \\ \begin{bmatrix} 7 & -2 \\ 7 & -2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} &= \begin{bmatrix} 0 \\ 0 \end{bmatrix}\end{aligned}$$

The augmented matrix for this system and corresponding reduced row-echelon form are given by

$$\left[ \begin{array}{cc|c} 7 & -2 & 0 \\ 7 & -2 & 0 \end{array} \right] \rightarrow \dots \rightarrow \left[ \begin{array}{cc|c} 1 & -\frac{2}{7} & 0 \\ 0 & 0 & 0 \end{array} \right]$$

The solution is any vector of the form

$$\begin{bmatrix} \frac{2}{7}s \\ s \end{bmatrix} = s \begin{bmatrix} \frac{2}{7} \\ 1 \end{bmatrix}$$

Multiplying this vector by 7 we obtain a simpler description for the solution to this system, given by

$$t \begin{bmatrix} 2 \\ 7 \end{bmatrix}$$

This gives the basic eigenvector for  $\lambda_1 = 2$  as

$$\begin{bmatrix} 2 \\ 7 \end{bmatrix}$$

To check, we verify that  $AX = 2X$  for this basic eigenvector.

$$\begin{bmatrix} -5 & 2 \\ -7 & 4 \end{bmatrix} \begin{bmatrix} 2 \\ 7 \end{bmatrix} = \begin{bmatrix} 4 \\ 14 \end{bmatrix} = 2 \begin{bmatrix} 2 \\ 7 \end{bmatrix}$$

This is what we wanted, so we know this basic eigenvector is correct.

Next we will repeat this process to find the basic eigenvector for  $\lambda_2 = -3$ . We wish to find all vectors  $X \neq 0$  such that  $AX = -3X$ . These are the solutions to  $((-3)I - A)X = 0$ .

$$\begin{aligned} \left( (-3) \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} - \begin{bmatrix} -5 & 2 \\ -7 & 4 \end{bmatrix} \right) \begin{bmatrix} x \\ y \end{bmatrix} &= \begin{bmatrix} 0 \\ 0 \end{bmatrix} \\ \begin{bmatrix} 2 & -2 \\ 7 & -7 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} &= \begin{bmatrix} 0 \\ 0 \end{bmatrix} \end{aligned}$$

The augmented matrix for this system and corresponding reduced row-echelon form are given by

$$\left[ \begin{array}{cc|c} 2 & -2 & 0 \\ 7 & -7 & 0 \end{array} \right] \rightarrow \cdots \rightarrow \left[ \begin{array}{cc|c} 1 & -1 & 0 \\ 0 & 0 & 0 \end{array} \right]$$

The solution is any vector of the form

$$\begin{bmatrix} s \\ s \end{bmatrix} = s \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

This gives the basic eigenvector for  $\lambda_2 = -3$  as

$$\begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

To check, we verify that  $AX = -3X$  for this basic eigenvector.

$$\begin{bmatrix} -5 & 2 \\ -7 & 4 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} -3 \\ -3 \end{bmatrix} = -3 \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

This is what we wanted, so we know this basic eigenvector is correct. ♠

The following is an example using Procedure G.5 for a  $3 \times 3$  matrix.

### Example G.7: Find the Eigenvalues and Eigenvectors

*Find the eigenvalues and eigenvectors for the matrix*

$$A = \begin{bmatrix} 5 & -10 & -5 \\ 2 & 14 & 2 \\ -4 & -8 & 6 \end{bmatrix}$$

**Solution.** We will use Procedure G.5. First we need to find the eigenvalues of  $A$ . Recall that they are the solutions of the equation

$$\det(xI - A) = 0$$

In this case the equation is

$$\det \left( x \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} - \begin{bmatrix} 5 & -10 & -5 \\ 2 & 14 & 2 \\ -4 & -8 & 6 \end{bmatrix} \right) = 0$$

which becomes

$$\det \begin{bmatrix} x-5 & 10 & 5 \\ -2 & x-14 & -2 \\ 4 & 8 & x-6 \end{bmatrix} = 0$$

Using Laplace Expansion, compute this determinant and simplify. The result is the following equation.

$$(x-5)(x^2 - 20x + 100) = 0$$

Solving this equation, we find that the eigenvalues are  $\lambda_1 = 5$ ,  $\lambda_2 = 10$  and  $\lambda_3 = 10$ . Notice that 10 is a root of multiplicity two due to

$$x^2 - 20x + 100 = (x-10)^2$$

Therefore,  $\lambda_2 = 10$  is an eigenvalue of multiplicity two.

Now that we have found the eigenvalues for  $A$ , we can compute the eigenvectors.

First we will find the basic eigenvectors for  $\lambda_1 = 5$ . In other words, we want to find all non-zero vectors  $X$  so that  $AX = 5X$ . This requires that we solve the equation  $(5I - A)X = 0$  for  $X$  as follows.

$$\left( 5 \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} - \begin{bmatrix} 5 & -10 & -5 \\ 2 & 14 & 2 \\ -4 & -8 & 6 \end{bmatrix} \right) \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

That is you need to find the solution to

$$\begin{bmatrix} 0 & 10 & 5 \\ -2 & -9 & -2 \\ 4 & 8 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

By now this is a familiar problem. You set up the augmented matrix and row reduce to get the solution. Thus the matrix you must row reduce is

$$\left[ \begin{array}{ccc|c} 0 & 10 & 5 & 0 \\ -2 & -9 & -2 & 0 \\ 4 & 8 & -1 & 0 \end{array} \right]$$

The reduced row-echelon form is

$$\left[ \begin{array}{ccc|c} 1 & 0 & -\frac{5}{4} & 0 \\ 0 & 1 & \frac{1}{2} & 0 \\ 0 & 0 & 0 & 0 \end{array} \right]$$

and so the solution is any vector of the form

$$\begin{bmatrix} \frac{5}{4}s \\ -\frac{1}{2}s \\ s \end{bmatrix} = s \begin{bmatrix} \frac{5}{4} \\ -\frac{1}{2} \\ 1 \end{bmatrix}$$

where  $s \in \mathbb{R}$ . If we multiply this vector by 4, we obtain a simpler description for the solution to this system, as given by

$$t \begin{bmatrix} 5 \\ -2 \\ 4 \end{bmatrix} \quad (7.3)$$

where  $t \in \mathbb{R}$ . Here, the basic eigenvector is given by

$$X_1 = \begin{bmatrix} 5 \\ -2 \\ 4 \end{bmatrix}$$

Notice that we cannot let  $t = 0$  here, because this would result in the zero vector and eigenvectors are never equal to 0! Other than this value, every other choice of  $t$  in 7.3 results in an eigenvector.

It is a good idea to check your work! To do so, we will take the original matrix and multiply by the basic eigenvector  $X_1$ . We check to see if we get  $5X_1$ .

$$\begin{bmatrix} 5 & -10 & -5 \\ 2 & 14 & 2 \\ -4 & -8 & 6 \end{bmatrix} \begin{bmatrix} 5 \\ -2 \\ 4 \end{bmatrix} = \begin{bmatrix} 25 \\ -10 \\ 20 \end{bmatrix} = 5 \begin{bmatrix} 5 \\ -2 \\ 4 \end{bmatrix}$$

This is what we wanted, so we know that our calculations were correct.

Next we will find the basic eigenvectors for  $\lambda_2, \lambda_3 = 10$ . These vectors are the basic solutions to the equation,

$$\left( 10 \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} - \begin{bmatrix} 5 & -10 & -5 \\ 2 & 14 & 2 \\ -4 & -8 & 6 \end{bmatrix} \right) \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

That is you must find the solutions to

$$\begin{bmatrix} 5 & 10 & 5 \\ -2 & -4 & -2 \\ 4 & 8 & 4 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

Consider the augmented matrix

$$\left[ \begin{array}{ccc|c} 5 & 10 & 5 & 0 \\ -2 & -4 & -2 & 0 \\ 4 & 8 & 4 & 0 \end{array} \right]$$

The reduced row-echelon form for this matrix is

$$\left[ \begin{array}{ccc|c} 1 & 2 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array} \right]$$

and so the eigenvectors are of the form

$$\begin{bmatrix} -2s-t \\ s \\ t \end{bmatrix} = s \begin{bmatrix} -2 \\ 1 \\ 0 \end{bmatrix} + t \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}$$

Note that you can't pick  $t$  and  $s$  both equal to zero because this would result in the zero vector and eigenvectors are never equal to zero.

Here, there are two basic eigenvectors, given by

$$X_2 = \begin{bmatrix} -2 \\ 1 \\ 0 \end{bmatrix}, X_3 = \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}$$

Taking any (nonzero) linear combination of  $X_2$  and  $X_3$  will also result in an eigenvector for the eigenvalue  $\lambda = 10$ . As in the case for  $\lambda = 5$ , always check your work! For the first basic eigenvector, we can check  $AX_2 = 10X_2$  as follows.

$$\begin{bmatrix} 5 & -10 & -5 \\ 2 & 14 & 2 \\ -4 & -8 & 6 \end{bmatrix} \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} -10 \\ 0 \\ 10 \end{bmatrix} = 10 \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}$$

This is what we wanted. Checking the second basic eigenvector,  $X_3$ , is left as an exercise. ♠

It is important to remember that for any eigenvector  $X$ ,  $X \neq 0$ . However, it is possible to have eigenvalues equal to zero. This is illustrated in the following example.

### Example G.8: A Zero Eigenvalue

Let

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

Find the eigenvalues and eigenvectors of  $A$ .

**Solution.** First we find the eigenvalues of  $A$ . We will do so using Definition G.2.

In order to find the eigenvalues of  $A$ , we solve the following equation.

$$\det(xI - A) = \det \begin{bmatrix} x-2 & -2 & 2 \\ -1 & x-3 & 1 \\ 1 & -1 & x-1 \end{bmatrix} = 0$$

This reduces to  $x^3 - 6x^2 + 8x = 0$ . You can verify that the solutions are  $\lambda_1 = 0, \lambda_2 = 2, \lambda_3 = 4$ . Notice that while eigenvectors can never equal 0, it is possible to have an eigenvalue equal to 0.

Now we will find the basic eigenvectors. For  $\lambda_1 = 0$ , we need to solve the equation  $(0I - A)X = 0$ . This equation becomes  $-AX = 0$ , and so the augmented matrix for finding the solutions is given by

$$\left[ \begin{array}{ccc|c} -2 & -2 & 2 & 0 \\ -1 & -3 & 1 & 0 \\ 1 & -1 & -1 & 0 \end{array} \right]$$

The reduced row-echelon form is

$$\left[ \begin{array}{ccc|c} 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array} \right]$$

Therefore, the eigenvectors are of the form  $t \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$  where  $t \neq 0$  and the basic eigenvector is given by

$$X_1 = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$$

We can verify that this eigenvector is correct by checking that the equation  $AX_1 = 0X_1$  holds. The product  $AX_1$  is given by

$$AX_1 = \begin{bmatrix} 2 & 2 & -2 \\ 1 & 3 & -1 \\ -1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

This clearly equals  $0X_1$ , so the equation holds. Hence,  $AX_1 = 0X_1$  and so 0 is an eigenvalue of  $A$ .

Computing the other basic eigenvectors is left as an exercise. ♠

In the following sections, we examine ways to simplify this process of finding eigenvalues and eigenvectors by using properties of special types of matrices.

### G.1.3. Eigenvalues and Eigenvectors for Special Types of Matrices

---

A special type of matrix we will consider in this section is the triangular matrix. Recall Definition E.12 which states that an upper (lower) triangular matrix contains all zeros below (above) the main diagonal. Remember that finding the determinant of a triangular matrix is a simple procedure of taking the product of the entries on the main diagonal.. It turns out that there is also a simple way to find the eigenvalues of a triangular matrix.

In the next example we will demonstrate that the eigenvalues of a triangular matrix are the entries on the main diagonal.

#### Example G.9: Eigenvalues for a Triangular Matrix

Let  $A = \begin{bmatrix} 1 & 2 & 4 \\ 0 & 4 & 7 \\ 0 & 0 & 6 \end{bmatrix}$ . Find the eigenvalues of  $A$ .

**Solution.** We need to solve the equation  $\det(xI - A) = 0$  as follows

$$\det(xI - A) = \det \begin{bmatrix} x-1 & -2 & -4 \\ 0 & x-4 & -7 \\ 0 & 0 & x-6 \end{bmatrix} = (x-1)(x-4)(x-6) = 0$$

Solving the equation  $(x-1)(x-4)(x-6) = 0$  for  $x$  results in the eigenvalues  $\lambda_1 = 1, \lambda_2 = 4$  and  $\lambda_3 = 6$ . Thus the eigenvalues are the entries on the main diagonal of the original matrix. ♠

The same result is true for lower triangular matrices. For any triangular matrix, the eigenvalues are equal to the entries on the main diagonal. To find the eigenvectors of a triangular matrix, we use the usual procedure.

## G.2 Positive Semi-Definite Matrices

---

Wikipedia - Definite Symmetric Matrix

In linear algebra, a symmetric  $n \times n$  real matrix  $M$  is said to be **positive-definite** if the scalar  $z^\top M z$  is strictly positive for every non-zero column vector  $z$  of  $n$  real numbers. Here  $z^\top$  denotes the transpose of  $z$ .<sup>1</sup> When interpreting  $Mz$  as the output of an operator,  $M$ , that is acting on an input,  $z$ , the property of positive definiteness implies that the output always has a positive inner product with the input, as often observed in physical processes.

More generally, a complex  $n \times n$  Hermitian matrix  $M$  is said to be **positive-definite** if the scalar  $z^* M z$  is strictly positive for every non-zero column vector  $z$  of  $n$  complex numbers. Here  $z^*$  denotes the conjugate transpose of  $z$ . Note that  $z^* M z$  is automatically real since  $M$  is Hermitian.

**Positive semi-definite** matrices are defined similarly, except that the above scalars  $z^\top M z$  or  $z^* M z$  must be positive *or zero* (i.e. non-negative). **Negative-definite** and **negative semi-definite** matrices are defined analogously. A matrix that is not positive semi-definite and not negative semi-definite is called **indefinite**.

The matrix  $M$  is positive-definite if and only if the bilinear form  $\langle z, w \rangle = z^\top M w$  is positive-definite (and similarly for a positive-definite sesquilinear form in the complex case). This is a coordinate realization of an inner product on a vector space.<sup>2</sup> Some authors use more general definitions of definiteness, including some non-symmetric real matrices, or non-Hermitian complex ones.

---

<sup>1</sup>W  
2

### G.2.1. Definitions

In the following definitions,  $\mathbf{x}^\top$  is the transpose of  $\mathbf{x}$ ,  $\mathbf{x}^*$  is the conjugate transpose of  $\mathbf{x}$  and  $\mathbf{0}$  denotes the  $n$ -dimensional zero-vector.

#### Definition G.10: Definiteness for Real Matrices

An  $n \times n$  symmetric real matrix  $M$  is said to be

- **positive-definite** if  $\mathbf{x}^\top M\mathbf{x} > 0$  for all non-zero  $\mathbf{x}$  in  $\mathbb{R}^n$ ,
- **positive semidefinite** or **non-negative-definite** if  $\mathbf{x}^\top M\mathbf{x} \geq 0$  for all  $\mathbf{x}$  in  $\mathbb{R}^n$ ,
- **negative-definite** if  $\mathbf{x}^\top M\mathbf{x} < 0$  for all non-zero  $\mathbf{x}$  in  $\mathbb{R}^n$ ,
- **negative-semidefinite** or **non-positive-definite** if  $\mathbf{x}^\top M\mathbf{x} \leq 0$  for all  $\mathbf{x}$  in  $\mathbb{R}^n$ ,
- An  $n \times n$  symmetric real matrix which is neither positive semidefinite nor negative semidefinite is called **indefinite**.

#### Example G.11: Identity Matrix

The identity matrix  $I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$  is positive-definite (and as such also positive semi-definite). It is a real symmetric matrix, and, for any non-zero column vector  $\mathbf{z}$  with real entries  $a$  and  $b$ , one has

$$\mathbf{z}^\top \mathbf{z} = [a \ b] \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = a^2 + b^2.$$

Seen as a complex matrix, for any non-zero column vector  $\mathbf{z}$  with complex entries  $a$  and  $b$  one has

$$\mathbf{z}^* \mathbf{z} = [\bar{a} \ \bar{b}] \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \bar{a}a + \bar{b}b = |a|^2 + |b|^2$$

Either way, the result is positive since  $\mathbf{z}$  is not the zero vector (that is, at least one of  $a$  and  $b$  is not zero).

**Example G.12: Positive definite**

The real symmetric matrix

$$M = \begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{bmatrix}$$

is positive-definite since for any non-zero column vector  $\mathbf{z}$  with entries  $a, b$  and  $c$ , we have

$$\mathbf{z}^\top M \mathbf{z} = (\mathbf{z}^\top M) \mathbf{z} \quad (7.1)$$

$$= [(2a-b) \quad (-a+2b-c) \quad (-b+2c)] \quad (7.2)$$

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} \quad (7.3)$$

$$= (2a-b)a + (-a+2b-c)b + (-b+2c)c \quad (7.4)$$

$$= 2a^2 - ba - ab + 2b^2 - cb - bc + 2c^2 \quad (7.5)$$

$$= 2a^2 - 2ab + 2b^2 - 2bc + 2c^2 \quad (7.6)$$

$$= a^2 + a^2 - 2ab + b^2 + b^2 - 2bc + c^2 + c^2 \quad (7.7)$$

$$= a^2 + (a-b)^2 + (b-c)^2 + c^2 \quad (7.8)$$

This result is a sum of squares, and therefore non-negative; and is zero only if  $a = b = c = 0$ , that is, when  $\mathbf{z}$  is the zero vector.

**Example G.13:  $A^\top A$** 

For any real invertible matrix  $A$ , the product  $A^\top A$  is a positive definite matrix. A simple proof is that for any non-zero vector  $\mathbf{z}$ , the condition  $\mathbf{z}^\top A^\top A \mathbf{z} = (\mathbf{A}\mathbf{z})^\top (\mathbf{A}\mathbf{z}) = \|\mathbf{A}\mathbf{z}\|^2 > 0$ , since the invertibility of matrix  $A$  means that  $\mathbf{A}\mathbf{z} \neq 0$ .

The example  $M$  above shows that a matrix in which some elements are negative may still be positive definite. Conversely, a matrix whose entries are all positive is not necessarily positive definite, as for example

$$N = \begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix},$$

for which  $[-1 \ 1] N [-1 \ 1]^\top = -2 < 0$ .

## G.2.2. Eigenvalues

---

### Theorem G.14: Eigenvalue Characterizations

Let  $M$  be an  $n \times n$  Hermitian matrix.

- $M$  is positive definite if and only if all of its eigenvalues are positive.
- $M$  is positive semi-definite if and only if all of its eigenvalues are non-negative.
- $M$  is negative definite if and only if all of its eigenvalues are negative
- $M$  is negative semi-definite if and only if all of its eigenvalues are non-positive.
- $M$  is indefinite if and only if it has both positive and negative eigenvalues.

Let  $P^{-1}DP$  be an eigendecomposition of  $M$ , where  $P$  is a unitary complex matrix whose rows comprise an orthonormal basis of eigenvectors of  $M$ , and  $D$  is a *real* diagonal matrix whose main diagonal contains the corresponding eigenvalues. The matrix  $M$  may be regarded as a diagonal matrix  $D$  that has been re-expressed in coordinates of the basis  $P$ . In particular, the one-to-one change of variable  $y = Pz$  shows that  $z^*Mz$  is real and positive for any complex vector  $z$  if and only if  $y^*Dy$  is real and positive for any  $y$ ; in other words, if  $D$  is positive definite. For a diagonal matrix, this is true only if each element of the main diagonal—that is, every eigenvalue of  $M$ —is positive. Since the spectral theorem guarantees all eigenvalues of a Hermitian matrix to be real, the positivity of eigenvalues can be checked using Descartes' rule of alternating signs when the characteristic polynomial of a real, symmetric matrix  $M$  is available.

## G.2.3. Quadratic forms

---

The (purely) quadratic form associated with a real  $n \times n$  matrix  $M$  is the function  $Q : \mathbb{R}^n \rightarrow \mathbb{R}$  such that  $Q(x) = x^\top Mx$  for all  $x$ .  $M$  can be assumed symmetric by replacing it with  $\frac{1}{2}(M + M^\top)$ .

A symmetric matrix  $M$  is positive definite if and only if its quadratic form is a strictly convex function.

More generally, any quadratic function from  $\mathbb{R}^n$  to  $\mathbb{R}$  can be written as  $x^\top Mx + x^\top b + c$  where  $M$  is a symmetric  $n \times n$  matrix,  $b$  is a real  $n$ -vector, and  $c$  a real constant. This quadratic function is strictly convex, and hence has a unique finite global minimum, if and only if  $M$  is positive definite. For this reason, positive definite matrices play an important role in optimization problems.

## G.2.4. Properties

---

### G.2.4.1. Inverse of positive definite matrix

---

Every positive definite matrix is invertible and its inverse is also positive definite.<sup>3</sup> If  $M \geq N > 0$  then  $N^{-1} \geq M^{-1} > 0$ .<sup>4</sup> Moreover, by the min-max theorem, the  $k$ th largest eigenvalue of  $M$  is greater than the  $k$ th largest eigenvalue of  $N$ .

### G.2.4.2. Scaling

---

If  $M$  is positive definite and  $r > 0$  is a real number, then  $rM$  is positive definite.<sup>5</sup>

### G.2.4.3. Addition

---

If  $M$  and  $N$  are positive definite, then the sum  $M + N$  is also positive definite.<sup>6</sup>

### G.2.4.4. Multiplication

---

- If  $M$  and  $N$  are positive definite, then the products  $MNM$  and  $NMN$  are also positive definite. If  $MN = NM$ , then  $MN$  is also positive definite.
- If  $M$  is positive semidefinite, then  $Q^\top MQ$  is positive semidefinite. If  $M$  is positive definite and  $Q$  has full column rank, then  $Q^\top MQ$  is positive definite.

### G.2.4.5. Cholesky decomposition

---

For any matrix  $A$ , the matrix  $A^*A$  is positive semidefinite, and  $\text{rank}(A) = \text{rank}(A^*A)$ . Conversely, any Hermitian positive semi-definite matrix  $M$  can be written as  $M = LL^*$ , where  $L$  is lower triangular; this is the Cholesky decomposition. If  $M$  is not positive definite, then some of the diagonal elements of  $L$  may be zero.

A hermitian matrix  $M$  is positive definite if and only if it has a unique Cholesky decomposition, i.e. the matrix  $M$  is positive definite if and only if there exists a unique lower triangular matrix  $L$ , with real and strictly positive diagonal elements, such that  $M = LL^*$ .

<sup>3</sup>, p. 397

<sup>4</sup>, Corollary 7.7.4(a)

<sup>5</sup>, Observation 7.1.3

<sup>6</sup>

### G.2.4.6. Square root

---

A matrix  $M$  is positive semi-definite if and only if there is a positive semi-definite matrix  $B$  with  $B^2 = M$ . This matrix  $B$  is unique,<sup>7</sup> is called the square root of  $M$ , and is denoted with  $B = M^{\frac{1}{2}}$  (the square root  $B$  is not to be confused with the matrix  $L$  in the Cholesky factorization  $M = LL^*$ , which is also sometimes called the square root of  $M$ ).

If  $M > N > 0$  then  $M^{\frac{1}{2}} > N^{\frac{1}{2}} > 0$ .

### G.2.4.7. Submatrices

---

Every principal submatrix of a positive definite matrix is positive definite.

## G.2.5. Convexity

---

The set of positive semidefinite symmetric matrices is convex. That is, if  $M$  and  $N$  are positive semidefinite, then for any  $\alpha$  between 0 and 1,  $\alpha M + (1 - \alpha)N$  is also positive semidefinite. For any vector  $x$ :

$$x^\top (\alpha M + (1 - \alpha)N)x = \alpha x^\top Mx + (1 - \alpha)x^\top Nx \geq 0.$$

This property guarantees that semidefinite programming problems converge to a globally optimal solution.

### G.2.5.1. Further properties

---

A Hermitian matrix is positive semidefinite if and only if all of its principal minors are nonnegative. It is however not enough to consider the leading principal minors only, as is checked on the diagonal matrix with entries 0 and 1.

### G.2.5.2. Block matrices

---

A positive  $2n \times 2n$  matrix may also be defined by blocks:

$$M = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$$

where each block is  $n \times n$ . By applying the positivity condition, it immediately follows that  $A$  and  $D$  are hermitian, and  $C = B^*$ .

We have that  $z^* M z \geq 0$  for all complex  $z$ , and in particular for  $z = [v, 0]^\top$ . Then

---

<sup>7</sup>, Theorem 7.2.6 with  $k = 2$

$$\begin{bmatrix} v^* & 0 \\ B^* & D \end{bmatrix} \begin{bmatrix} A & B \\ 0 & D \end{bmatrix} \begin{bmatrix} v \\ 0 \end{bmatrix} = v^* Av \geq 0.$$

A similar argument can be applied to  $D$ , and thus we conclude that both  $A$  and  $D$  must be positive definite matrices, as well.

Converse results can be proved with stronger conditions on the blocks, for instance using the Schur complement.

### G.2.5.3. Local extrema

---

A general quadratic form  $f(\mathbf{x})$  on  $n$  real variables  $x_1, \dots, x_n$  can always be written as  $\mathbf{x}^\top M \mathbf{x}$  where  $\mathbf{x}$  is the column vector with those variables, and  $M$  is a symmetric real matrix. Therefore, the matrix being positive definite means that  $f$  has a unique minimum (zero) when  $\mathbf{x}$  is zero, and is strictly positive for any other  $\mathbf{x}$ .

More generally, a twice-differentiable real function  $f$  on  $n$  real variables has local minimum at arguments  $x_1, \dots, x_n$  if its gradient is zero and its Hessian (the matrix of all second derivatives) is positive semi-definite at that point. Similar statements can be made for negative definite and semi-definite matrices.

### G.2.5.4. Covariance

---

In statistics, the covariance matrix of a multivariate probability distribution is always positive semi-definite; and it is positive definite unless one variable is an exact linear function of the others. Conversely, every positive semi-definite matrix is the covariance matrix of some multivariate distribution.

## G.2.6. External links

---

- Wolfram MathWorld: Positive Definite Matrix



## **Part VI**

# **Other Appendices**



# A. Some Prerequisite Topics

---

The topics presented in this section are important concepts in mathematics and therefore should be examined.

## A.1 Sets and Set Notation

---

A set is a collection of things called elements. For example  $\{1, 2, 3, 8\}$  would be a set consisting of the elements 1, 2, 3, and 8. To indicate that 3 is an element of  $\{1, 2, 3, 8\}$ , it is customary to write  $3 \in \{1, 2, 3, 8\}$ . We can also indicate when an element is not in a set, by writing  $9 \notin \{1, 2, 3, 8\}$  which says that 9 is not an element of  $\{1, 2, 3, 8\}$ . Sometimes a rule specifies a set. For example you could specify a set as all integers larger than 2. This would be written as  $S = \{x \in \mathbb{Z} : x > 2\}$ . This notation says:  $S$  is the set of all integers,  $x$ , such that  $x > 2$ .

Suppose  $A$  and  $B$  are sets with the property that every element of  $A$  is an element of  $B$ . Then we say that  $A$  is a subset of  $B$ . For example,  $\{1, 2, 3, 8\}$  is a subset of  $\{1, 2, 3, 4, 5, 8\}$ . In symbols, we write  $\{1, 2, 3, 8\} \subseteq \{1, 2, 3, 4, 5, 8\}$ . It is sometimes said that “ $A$  is contained in  $B$ ” or even “ $B$  contains  $A$ ”. The same statement about the two sets may also be written as  $\{1, 2, 3, 4, 5, 8\} \supseteq \{1, 2, 3, 8\}$ .

We can also talk about the *union* of two sets, which we write as  $A \cup B$ . This is the set consisting of everything which is an element of at least one of the sets,  $A$  or  $B$ . As an example of the union of two sets, consider  $\{1, 2, 3, 8\} \cup \{3, 4, 7, 8\} = \{1, 2, 3, 4, 7, 8\}$ . This set is made up of the numbers which are in at least one of the two sets.

In general

$$A \cup B = \{x : x \in A \text{ or } x \in B\}$$

Notice that an element which is in *both*  $A$  and  $B$  is also in the union, as well as elements which are in only one of  $A$  or  $B$ .

Another important set is the intersection of two sets  $A$  and  $B$ , written  $A \cap B$ . This set consists of everything which is in *both* of the sets. Thus  $\{1, 2, 3, 8\} \cap \{3, 4, 7, 8\} = \{3, 8\}$  because 3 and 8 are those elements the two sets have in common. In general,

$$A \cap B = \{x : x \in A \text{ and } x \in B\}$$

If  $A$  and  $B$  are two sets,  $A \setminus B$  denotes the set of things which are in  $A$  but not in  $B$ . Thus

$$A \setminus B = \{x \in A : x \notin B\}$$

For example, if  $A = \{1, 2, 3, 8\}$  and  $B = \{3, 4, 7, 8\}$ , then  $A \setminus B = \{1, 2, 3, 8\} \setminus \{3, 4, 7, 8\} = \{1, 2\}$ .

A special set which is very important in mathematics is the empty set denoted by  $\emptyset$ . The empty set,  $\emptyset$ , is defined as the set which has no elements in it. It follows that the empty set is a subset of every set. This is true because if it were not so, there would have to exist a set  $A$ , such that  $\emptyset$  has something in it which is not in  $A$ . However,  $\emptyset$  has nothing in it and so it must be that  $\emptyset \subseteq A$ .

We can also use brackets to denote sets which are intervals of numbers. Let  $a$  and  $b$  be real numbers. Then

- $[a, b] = \{x \in \mathbb{R} : a \leq x \leq b\}$
- $[a, b) = \{x \in \mathbb{R} : a \leq x < b\}$
- $(a, b) = \{x \in \mathbb{R} : a < x < b\}$
- $(a, b] = \{x \in \mathbb{R} : a < x \leq b\}$
- $[a, \infty) = \{x \in \mathbb{R} : x \geq a\}$
- $(-\infty, a] = \{x \in \mathbb{R} : x \leq a\}$

These sorts of sets of real numbers are called intervals. The two points  $a$  and  $b$  are called endpoints, or bounds, of the interval. In particular,  $a$  is the *lower bound* while  $b$  is the *upper bound* of the above intervals, where applicable. Other intervals such as  $(-\infty, b)$  are defined by analogy to what was just explained. In general, the curved parenthesis,  $($ , indicates the end point is not included in the interval, while the square parenthesis,  $[$ , indicates this end point is included. The reason that there will always be a curved parenthesis next to  $\infty$  or  $-\infty$  is that these are not real numbers and cannot be included in the interval in the way a real number can.

To illustrate the use of this notation relative to intervals consider three examples of inequalities. Their solutions will be written in the interval notation just described.

### Example A.1: Solving an Inequality

Solve the inequality  $2x + 4 \leq x - 8$ .

**Solution.** We need to find  $x$  such that  $2x + 4 \leq x - 8$ . Solving for  $x$ , we see that  $x \leq -12$  is the answer. This is written in terms of an interval as  $(-\infty, -12]$ . ♠

Consider the following example.

### Example A.2: Solving an Inequality

Solve the inequality  $(x + 1)(2x - 3) \geq 0$ .

**Solution.** We need to find  $x$  such that  $(x + 1)(2x - 3) \geq 0$ . The solution is given by  $x \leq -1$  or  $x \geq \frac{3}{2}$ . Therefore,  $x$  which fit into either of these intervals gives a solution. In terms of set notation this is denoted by  $(-\infty, -1] \cup [\frac{3}{2}, \infty)$ . ♠

Consider one last example.

### Example A.3: Solving an Inequality

Solve the inequality  $x(x+2) \geq -4$ .

**Solution.** This inequality is true for any value of  $x$  where  $x$  is a real number. We can write the solution as  $\mathbb{R}$  or  $(-\infty, \infty)$ . ♠

In the next section, we examine another important mathematical concept.

## A.2 Well Ordering and Induction

---

We begin this section with some important notation. Summation notation, written  $\sum_{i=1}^j i$ , represents a sum. Here,  $i$  is called the index of the sum, and we add iterations until  $i = j$ . For example,

$$\sum_{i=1}^j i = 1 + 2 + \cdots + j$$

Another example:

$$a_{11} + a_{12} + a_{13} = \sum_{i=1}^3 a_{1i}$$

The following notation is a specific use of summation notation.

### Notation A.4: Summation Notation

Let  $a_{ij}$  be real numbers, and suppose  $1 \leq i \leq r$  while  $1 \leq j \leq s$ . These numbers can be listed in a rectangular array as given by

$$\begin{array}{cccc} a_{11} & a_{12} & \cdots & a_{1s} \\ a_{21} & a_{22} & \cdots & a_{2s} \\ \vdots & \vdots & & \vdots \\ a_{r1} & a_{r2} & \cdots & a_{rs} \end{array}$$

Then  $\sum_{j=1}^s \sum_{i=1}^r a_{ij}$  means to first sum the numbers in each column (using  $i$  as the index) and then to add the sums which result (using  $j$  as the index). Similarly,  $\sum_{i=1}^r \sum_{j=1}^s a_{ij}$  means to sum the vectors in each row (using  $j$  as the index) and then to add the sums which result (using  $i$  as the index).

Notice that since addition is commutative,  $\sum_{j=1}^s \sum_{i=1}^r a_{ij} = \sum_{i=1}^r \sum_{j=1}^s a_{ij}$ .

We now consider the main concept of this section. Mathematical induction and well ordering are two extremely important principles in math. They are often used to prove significant things which would be hard to prove otherwise.

**Definition A.5: Well Ordered**

A set is well ordered if every nonempty subset  $S$ , contains a smallest element  $z$  having the property that  $z \leq x$  for all  $x \in S$ .

In particular, the set of natural numbers defined as

$$\mathbb{N} = \{1, 2, \dots\}$$

is well ordered.

Consider the following proposition.

**Proposition A.6: Well Ordered Sets**

*Any set of integers larger than a given number is well ordered.*

This proposition claims that if a set has a lower bound which is a real number, then this set is well ordered.

Further, this proposition implies the principle of mathematical induction. The symbol  $\mathbb{Z}$  denotes the set of all integers. Note that if  $a$  is an integer, then there are no integers between  $a$  and  $a + 1$ .

**Theorem A.7: Mathematical Induction**

*A set  $S \subseteq \mathbb{Z}$ , having the property that  $a \in S$  and  $n + 1 \in S$  whenever  $n \in S$ , contains all integers  $x \in \mathbb{Z}$  such that  $x \geq a$ .*

**Proof.** Let  $T$  consist of all integers larger than or equal to  $a$  which are not in  $S$ . The theorem will be proved if  $T = \emptyset$ . If  $T \neq \emptyset$  then by the well ordering principle, there would have to exist a smallest element of  $T$ , denoted as  $b$ . It must be the case that  $b > a$  since by definition,  $a \notin T$ . Thus  $b \geq a + 1$ , and so  $b - 1 \geq a$  and  $b - 1 \notin S$  because if  $b - 1 \in S$ , then  $b - 1 + 1 = b \in S$  by the assumed property of  $S$ . Therefore,  $b - 1 \in T$  which contradicts the choice of  $b$  as the smallest element of  $T$ . ( $b - 1$  is smaller.) Since a contradiction is obtained by assuming  $T \neq \emptyset$ , it must be the case that  $T = \emptyset$  and this says that every integer at least as large as  $a$  is also in  $S$ . ♠

Mathematical induction is a very useful device for proving theorems about the integers. The procedure is as follows.

**Procedure A.8: Proof by Mathematical Induction**

*Suppose  $S_n$  is a statement which is a function of the number  $n$ , for  $n = 1, 2, \dots$ , and we wish to show that  $S_n$  is true for all  $n \geq 1$ . To do so using mathematical induction, use the following steps.*

1. **Base Case:** Show  $S_1$  is true.
2. Assume  $S_n$  is true for some  $n$ , which is the **induction hypothesis**. Then, using this assumption, show that  $S_{n+1}$  is true.

*Proving these two steps shows that  $S_n$  is true for all  $n = 1, 2, \dots$*

We can use this procedure to solve the following examples.

### Example A.9: Proving by Induction

*Prove by induction that  $\sum_{k=1}^n k^2 = \frac{n(n+1)(2n+1)}{6}$ .*

**Solution.** By Procedure A.8, we first need to show that this statement is true for  $n = 1$ . When  $n = 1$ , the statement says that

$$\begin{aligned}\sum_{k=1}^1 k^2 &= \frac{1(1+1)(2(1)+1)}{6} \\ &= \frac{6}{6} \\ &= 1\end{aligned}$$

The sum on the left hand side also equals 1, so this equation is true for  $n = 1$ .

Now suppose this formula is valid for some  $n \geq 1$  where  $n$  is an integer. Hence, the following equation is true.

$$\sum_{k=1}^n k^2 = \frac{n(n+1)(2n+1)}{6} \quad (1.1)$$

We want to show that this is true for  $n + 1$ .

Suppose we add  $(n+1)^2$  to both sides of equation 1.1.

$$\begin{aligned}\sum_{k=1}^{n+1} k^2 &= \sum_{k=1}^n k^2 + (n+1)^2 \\ &= \frac{n(n+1)(2n+1)}{6} + (n+1)^2\end{aligned}$$

The step going from the first to the second line is based on the assumption that the formula is true for  $n$ . Now simplify the expression in the second line,

$$\frac{n(n+1)(2n+1)}{6} + (n+1)^2$$

This equals

$$(n+1) \left( \frac{n(2n+1)}{6} + (n+1) \right)$$

and

$$\frac{n(2n+1)}{6} + (n+1) = \frac{6(n+1) + 2n^2 + n}{6} = \frac{(n+2)(2n+3)}{6}$$

Therefore,

$$\sum_{k=1}^{n+1} k^2 = \frac{(n+1)(n+2)(2n+3)}{6} = \frac{(n+1)((n+1)+1)(2(n+1)+1)}{6}$$

showing the formula holds for  $n + 1$  whenever it holds for  $n$ . This proves the formula by mathematical induction. In other words, this formula is true for all  $n = 1, 2, \dots$ .

Consider another example.

### Example A.10: Proving an Inequality by Induction

Show that for all  $n \in \mathbb{N}$ ,  $\frac{1}{2} \cdot \frac{3}{4} \cdots \frac{2n-1}{2n} < \frac{1}{\sqrt{2n+1}}$ .

**Solution.** Again we will use the procedure given in Procedure A.8 to prove that this statement is true for all  $n$ . Suppose  $n = 1$ . Then the statement says

$$\frac{1}{2} < \frac{1}{\sqrt{3}}$$

which is true.

Suppose then that the inequality holds for  $n$ . In other words,

$$\frac{1}{2} \cdot \frac{3}{4} \cdots \frac{2n-1}{2n} < \frac{1}{\sqrt{2n+1}}$$

is true.

Now multiply both sides of this inequality by  $\frac{2n+1}{2n+2}$ . This yields

$$\frac{1}{2} \cdot \frac{3}{4} \cdots \frac{2n-1}{2n} \cdot \frac{2n+1}{2n+2} < \frac{1}{\sqrt{2n+1}} \frac{2n+1}{2n+2} = \frac{\sqrt{2n+1}}{2n+2}$$

The theorem will be proved if this last expression is less than  $\frac{1}{\sqrt{2n+3}}$ . This happens if and only if

$$\left( \frac{1}{\sqrt{2n+3}} \right)^2 = \frac{1}{2n+3} > \frac{2n+1}{(2n+2)^2}$$

which occurs if and only if  $(2n+2)^2 > (2n+3)(2n+1)$  and this is clearly true which may be seen from expanding both sides. This proves the inequality.

Let's review the process just used. If  $S$  is the set of integers at least as large as 1 for which the formula holds, the first step was to show  $1 \in S$  and then that whenever  $n \in S$ , it follows  $n + 1 \in S$ . Therefore, by the principle of mathematical induction,  $S$  contains  $[1, \infty) \cap \mathbb{Z}$ , all positive integers. In doing an inductive proof of this sort, the set  $S$  is normally not mentioned. One just verifies the steps above.

## B. 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 <b>all</b> 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
<code>pip install package-name</code>	Install the specified package.
<code>pip install --upgrade package-name</code>	Update the specified package.
<code>pip freeze</code>	Display the version number on all installed packages.
<code>pip --help</code>	Display the documentation for pip.

See [https://pip.pypa.io/en/stable/user\\_guide/](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/](https://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, Jupyter 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  $\text{\LaTeX}$ , and can embed 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.



# C. 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.

$$A[0] = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix} \quad A[2,1] = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \text{} & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix}$$

## 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.”

$$A[1] = A[1,:] = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix} \quad A[:,2] = \begin{bmatrix} \times & \times & \text{} & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix}$$

$$A[1:,:2] = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix} \quad A[1:-1,1:-1] = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix}$$

## Stacking

---

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

$$A = \begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{bmatrix}$$

$$B = \begin{bmatrix} * & * & * \\ * & * & * \\ * & * & * \end{bmatrix}$$

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

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

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()`.

$$x = [ \times \quad \times \quad \times \quad \times ]$$

$$y = [ * \quad * \quad * \quad * ]$$

$$\text{np.hstack}((x, y, x)) = [ \times \quad \times \quad \times \quad \times \quad * \quad * \quad * \quad * \quad \times \quad \times \quad \times \quad \times ]$$

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

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

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.

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} \quad x = [10 \ 20 \ 30]$$

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

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

## 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`.

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

$$\text{A.sum(axis=0)} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix} = [ 4 \ 8 \ 12 \ 16 ]$$

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

# D. Plot Customization and Matplotlib Syntax Guide

---

**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.

- 

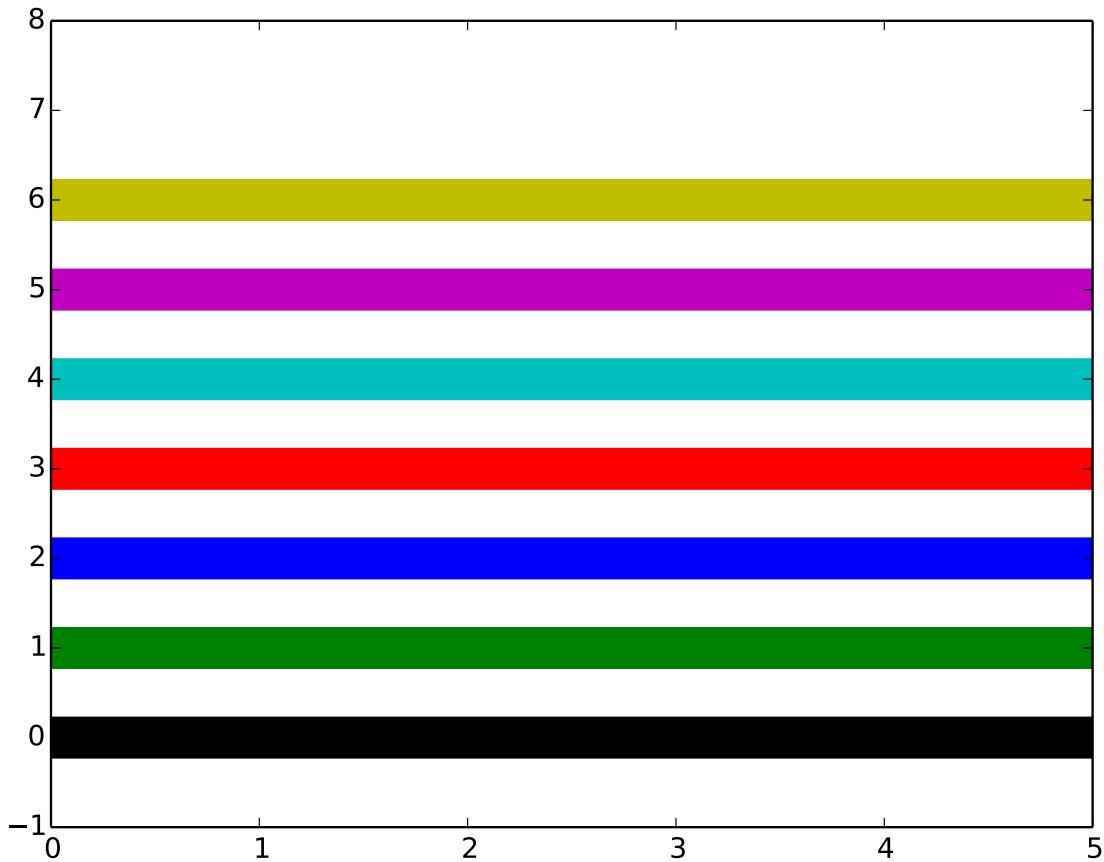
Matplotlib recognizes some basic built-in colors.

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 D.1.

```
1 import numpy as np
2 from matplotlib import pyplot as plt
3
4 colors = np.array(["k", "g", "b", "r", "c", "m", "y", "w"])
5 x = np.linspace(0, 5, 1000)
6 y = np.ones(1000)
7
8 for i in xrange(8):
9     plt.plot(x, i*y, colors[i], linewidth=18)
```

10



**Figure D.1: A display of all the built-in colors.**

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

### colors.py

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. Similarly, `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 D.2.

```

1 lw = np.linspace(.5, 15, 8)
2
3 for i in xrange(8):
4     plt.plot(x, i*y, colors[i], linewidth=lw[i])
5
6 plt.ylim([-1, 8])
7 plt.show()

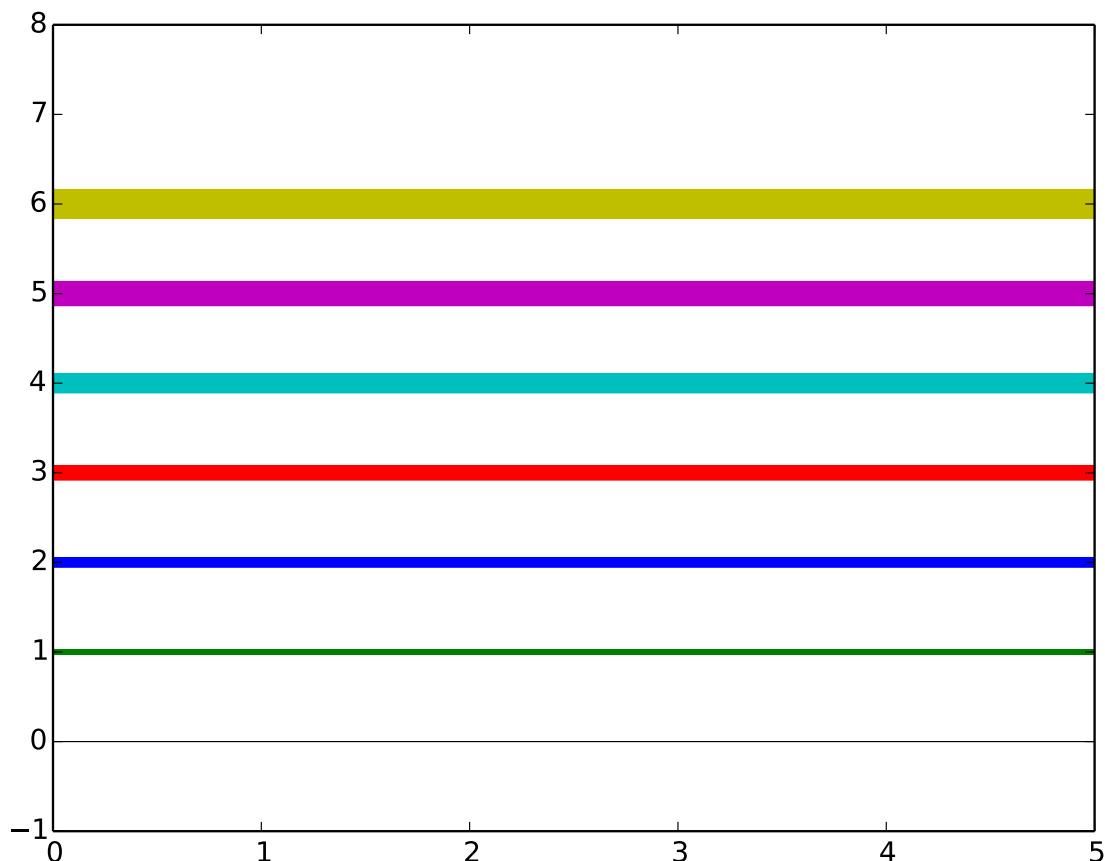
```

**linewidth.py**

## Style

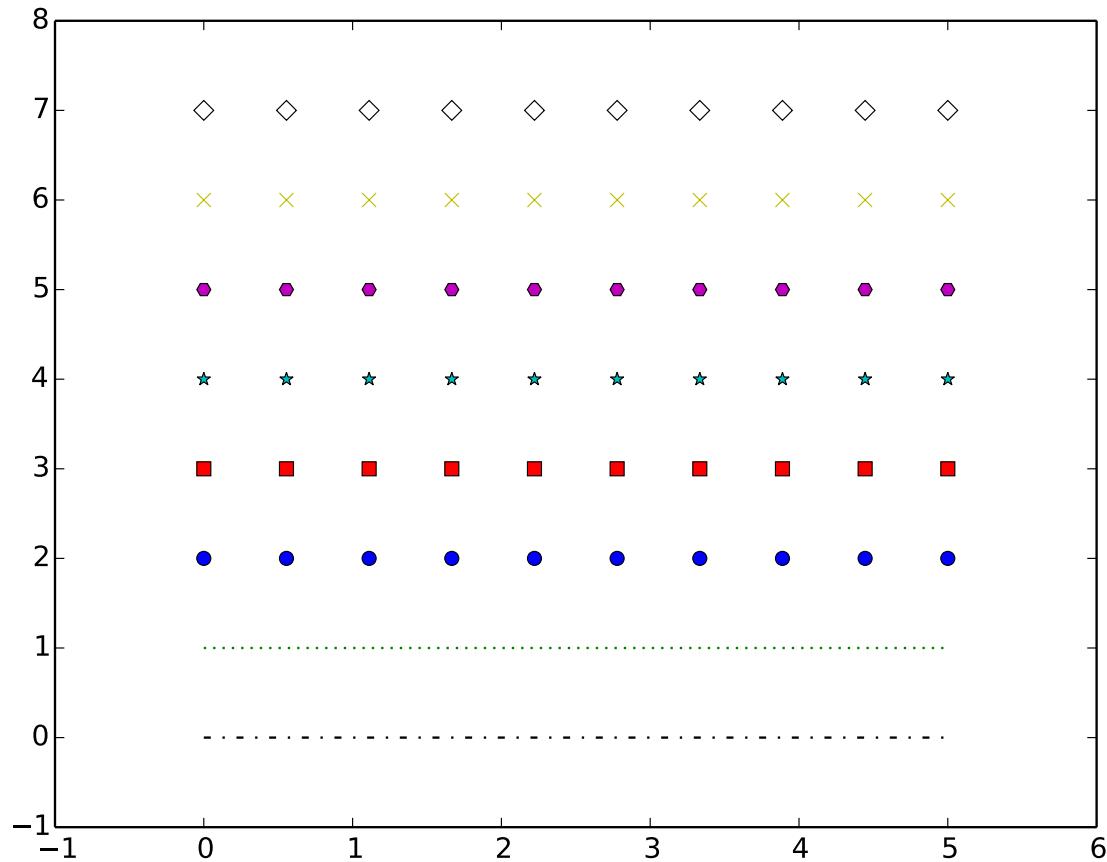
---

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



**Figure D.2:** plot of varying linewidths.

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



**Figure D.3: plot of varying linestyles.**

The following displays how `linestyle` can be implemented. It is displayed in Figure D.3.

```

1 x = np.linspace(0, 5, 10)
2 y = np.ones(10)
3 ls = np.array(['-.', ':', 'o', 's', '*', 'H', 'x', 'D'])
4
5 for i in xrange(8):
6     plt.plot(x, i*y, colors[i]+ls[i])
7
8 plt.axis([-1, 6, -1, 8])
plt.show()

```

**linestyle.py**

## 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 D.4.

```

1 for i in xrange(8):
2     plt.plot(x, i*y, colors[i]+ls[i])
3
4 plt.title("My Plot of Varying Linestyles", fontsize = 20, color = "gold")
5 plt.xlabel("x-axis", fontsize = 10, color = "darkcyan")
6 plt.ylabel("y-axis", fontsize = 10, color = "darkcyan")
7
8 plt.axis([-1, 6, -1, 8])
9 plt.show()

```

**text.py**

See <http://matplotlib.org> for Matplotlib documentation.

## D.1 Jupyter Notebooks

---

<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>

## D.2 Reading and Writing

---

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

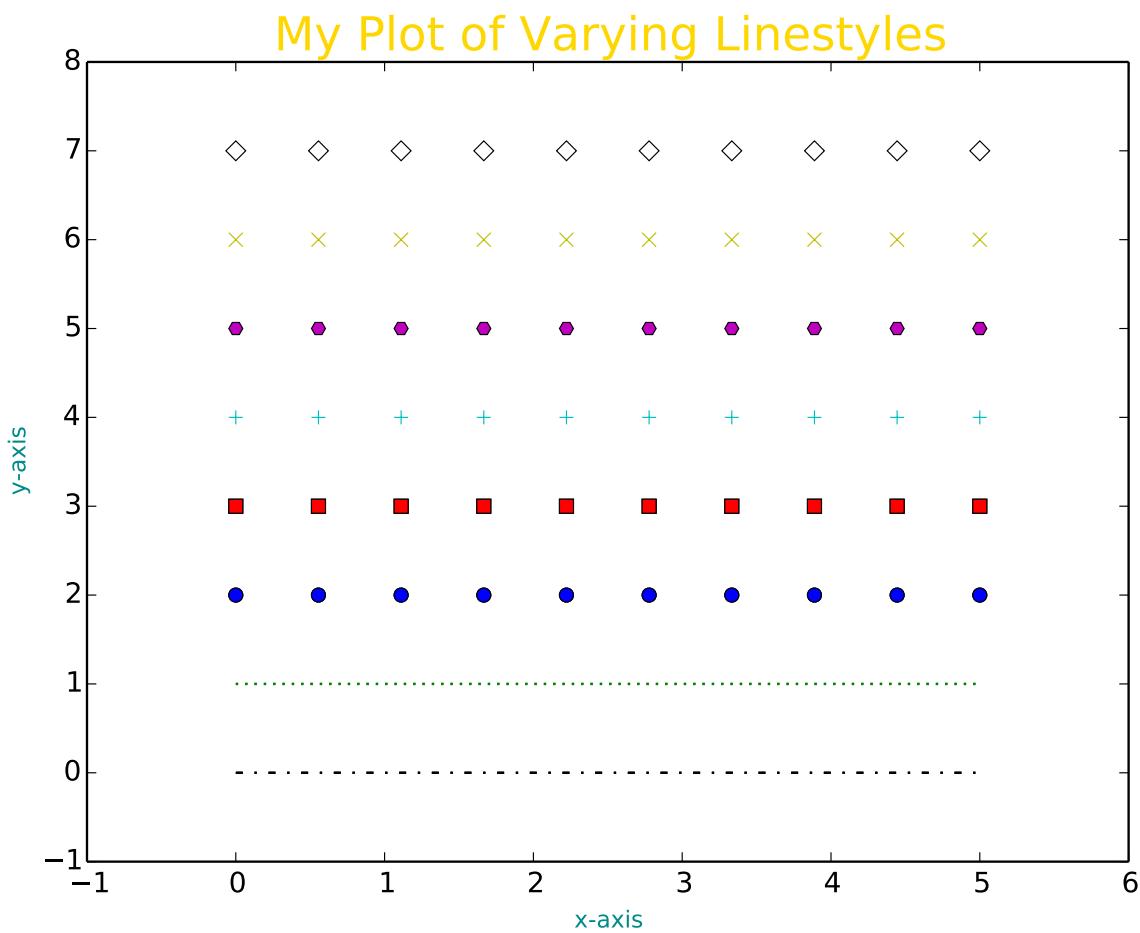


Figure D.4: plot of varying linestyles using text labels.

## D.3 Python Crash Course

---

<https://github.com/rpmuller/PythonCrashCourse>

## D.4 Excel Solver

---

### D.4.1. 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

## D.4.2. Links

---

Loan Example

Several Examples including TSP

## D.5 GECODE and MiniZinc

---

Open constraint programming toolkit. <https://www.gecode.org/> See also MiniZinc, which is a modeler that uses GECODE. <https://www.minizinc.org/>.

Also, they have two coursera courses on using their code: <https://www.coursera.org/learn/basic-modeling?action=enroll&authMode=signup>  
<https://www.coursera.org/learn/advanced-modeling?action=enroll&authMode=signup>

## D.6 Optaplanner

---

Open source software to solve a variety of problems with local heuristics. All code is in Java.

<https://www.optaplanner.org/>

## D.7 Python Modeling/Optimization

---

### D.7.1. SCIP

---

SCIP youtube channel

Youtube! SCIP solving MINLP Circle Packing Problem Model and Code in SCIP SCIP - Python Interface Demonstration

## D.7.2. Pyomo

---

Excellent modeling language. Open source. Many features. <http://www.pyomo.org/>

## D.7.3. Python-MIP

---

Awesome new modeling language for python that is very efficient at setting up optimization problems.  
Loads CBC binaries. Can be installed with pip. <https://python-mip.com/>

## D.7.4. Local Solver

---

<https://www.localsolver.com/> <https://www.youtube.com/watch?v=4aw9PM09U5Q>

## D.7.5. GUROBI

---

Solver takes too long to find Incumbent solution:

It might be better to focus on trying to find heuristic solutions faster. You can do this with

## D.7.6. CPLEX

---

ILOG CPLEX optimization Studio

<https://www.youtube.com/watch?v=IwYt5bzrhxA>

## D.7.7. Scipy

---

[http://scipy-lectures.org/advanced/mathematical\\_optimization/index.html](http://scipy-lectures.org/advanced/mathematical_optimization/index.html)

# D.8 Julia

---

## D.8.1. JuMP

---

<https://www.juliaopt.org/> <https://jump.dev/JuMP.jl/dev/>

## D.9 LINDO/LINGO

---

<https://www.lindo.com/>

## D.10 Foundations of Machine Learning

---

Free course with excellent videos on foundations of machine learning

## D.11 Convex Optimization

---

<https://www.youtube.com/watch?v=thuYiebq1cE&t=925s> <https://www.youtube.com/watch?v=40ifjG2kJQ>