BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF MECHANICAL ENGINEERING
FAKULTA STROJNÍHO INŽENÝRSTVÍ

INSTITUTE OF MATHEMATICS
ÚSTAV MATEMATIKY

EFFICIENT IMPLEMENTATION OF ADVANCED OPTIMIZATION ALGORITHMS
POKROČILÉ OPTIMALIZAČNÍ ALGORITMY A JEJICH EFEKTIVNÍ IMPLEMENTACE

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR
AUTOR PRÁCE
Bc. Jaroslav Talpa

SUPERVISOR
VEDOUCÍ PRÁCE
RNDr. Pavel Popela, Ph.D.

BRNO 2020

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF MECHANICAL ENGINEERING

Specification Master's Thesis

Department:	Institute of Mathematics
Student:	**Bc. Jaroslav Talpa**
Study programme:	Applied Sciences in Engineering
Study branch:	Mathematical Engineering
Supervisor:	**RNDr. Pavel Popela, Ph.D.**
Academic year:	2019/20

Pursuant to Act no. 111/1998 concerning universities and the BUT study and examination rules, you have been assigned the following topic by the institute director Master's Thesis:

Efficient Implementation of Advanced Optimization Algorithms

Concise characteristic of the task:

Student will study research area of composed optimization problems. He will focus on modern decomposition approaches and utilization of recent optimization solvers as building stones of advanced composed optimization algorithms. The emphasis will be put on the development of advanced algorithms and their efficient software implementation. He will also study theoretical properties and transformations of models from the viewpoint of solvability. The student's participation on research projects (e.g., SPIL, TIRSM etc.] will be welcome with involvement of supervisors specialists .Dr. Ing. Somplak and Ing. V. Nevrly.

Goals Master's Thesis:

1. The overview written on composed optimization programs.
2. Utilization of modelling tools and solvers for mathematical programming.
3. Research on properties and transformations of selected mathematical programs for the chosen application area.
4. Design and development of advanced algorithms.
5. Efficient algorithms' implementation.
6. Test computations for real–world engineering application.

Recommended bibliography:

BAZARAA, M. S., SHERALI, H. D. a SHETTY, C. M. Nonlinear programming: theory and algorithms. 2nd ed. New York: John Wiley & Sons. ISBN 0471599735. 1993.

BIRGE, J. R. a LOUVEAUX, F. Introduction to Stochastic Programming. Springer Verlag, 1997. ISBN: 978-1-4614-0236-7.

BOYD, S. a VANDENBERGHE, L. Convex Optimization. Cambridge: Cambridge University Press, 2004. ISBN 978-0-521-83378-3.

KALL, P. a WALLACE, S. W. Stochastic Programming. New York: John Wiley & Sons, 1993. ISBN 978-0471951582.

PARDALOS, P. M. a RESENDE, M. G. C. (eds.). Handbook of applied optimization. Oxford: Oxford University Press, 2002. ISBN 0195125940.

RUSZCZYNSKI, A. et al. Handbooks in Operations Research and Management Science, vol. 10: Stochastic Programming. Amsterdam: Elsevier, 2003. ISBN 978-0-444-50854-6.

Deadline for submission Master's Thesis is given by the Schedule of the Academic year 2019/20

In Brno,

L. S.

prof. RNDr. Josef Šlapal, CSc. Director of the Institute	doc. Ing. Jaroslav Katolický, Ph.D. FME dean

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA STROJNÍHO INŽENÝRSTVÍ

Zadání diplomové práce

Ústav:	Ústav matematiky
Student:	**Bc. Jaroslav Talpa**
Studijní program:	Aplikované vědy v inženýrství
Studijní obor:	Matematické inženýrství
Vedoucí práce:	**RNDr. Pavel Popela, Ph.D.**
Akademický rok:	2019/20

Ředitel ústavu Vám v souladu se zákonem č.111/1998 o vysokých školách a se Studijním a zkušebním řádem VUT v Brně určuje následující téma diplomové práce:

Pokročilé optimalizační algoritmy a jejich efektivní implementace

Stručná charakteristika problematiky úkolu:

Student se seznámí s problematikou řešení složitých optimalizačních úloh. Zaměří se na moderní dekompoziční přístupy a využití existujících optimalizačních řešičů, jako stavebních kamenů pokročilých optimalizačních algoritmů. Důraz bude kladen na vývoj pokročilých algoritmů a jejich efektivní softwarové implementace. Student se z pohledu řešitelnosti uvažovaných úloh bude také zabývat studiem vlastností optimalizačních modelů a jejich vhodnými transformacemi. Předpokládá se zapojení studenta do řešení optimalizačních úloh vybraných výzkumných projektů (SPIL, TIRSM aj.) za účasti konzultantů specialistů: Dr. Ing. R. Šompláka a Ing. V. Nevrlého.

Cíle diplomové práce:

1. Studium problematiky složitých optimalizačních úloh.
2. Studium a osvojení využívání implementací řešičů matematického programování.
3. Studium vlastností a transformací vybraných modelů matematického programování pro zvolenou třídu aplikačních úloh.
4. Návrh a vývoj pokročilých algoritmů pro studované modely.
5. Efektivní implementace algoritmů kombinující řešiče matematického programování včetně testování možností paralelizace.
6. Testovací výpočty, případná aplikace na vybraný inženýrský problém.

Seznam doporučené literatury:

BAZARAA, M. S., SHERALI, H. D. a SHETTY, C. M. Nonlinear programming: theory and algorithms. 2nd ed. New York: John Wiley & Sons. ISBN 0471599735. 1993.

BIRGE, J. R. a LOUVEAUX, F. Introduction to Stochastic Programming. Springer Verlag, 1997. ISBN: 978-1-4614-0236-7.

BOYD, S. a VANDENBERGHE, L. Convex Optimization. Cambridge: Cambridge University Press, 2004. ISBN 978-0-521-83378-3.

KALL, P. a WALLACE, S. W. Stochastic Programming. New York: John Wiley & Sons, 1993. ISBN 978-0471951582.

PARDALOS, P. M. a RESENDE, M. G. C. (eds.). Handbook of applied optimization. Oxford: Oxford University Press, 2002. ISBN 0195125940.

RUSZCZYNSKI, A. et al. Handbooks in Operations Research and Management Science, vol. 10: Stochastic Programming. Amsterdam: Elsevier, 2003. ISBN 978-0-444-50854-6.

Termín odevzdání diplomové práce je stanoven časovým plánem akademického roku 2019/20

V Brně, dne

L. S.

prof. RNDr. Josef Šlapal, CSc. ředitel ústavu	doc. Ing. Jaroslav Katolický, Ph.D. děkan fakulty

Abstrakt
Tato diplomová práce se zabývá tématikou konvexní optimalizace a to konkrétně modifikacemi algoritmu ADMM, společně s problematikou proximálních operátorů. Jedna z verzí ADMM je pak implementována v programovacím jazyce Julia s důrazem na obecnost a efektivnost této implementace, a dále aplikována na rozsáhlou úlohu z oblasti odpadového hospodářství.

Summary
This master's thesis concerns itself with the topic of convex optimization, specifically formulations of the ADMM algorithm, together with the area of proximal operators. One of these versions of ADMM is then implemented in the Julia programming languge with an emphasis on the reusability and efficiency of this implementation, and is further applied to a large model from the field of waste management.

Klíčová slova
konvexní optimalizace, proximální operátory, ADMM, jazyk Julia, odpadové hospodářství

Keywords
convex optimization, proximal operators, ADMM, the Julia language, waste management

TALPA, J. *Pokročilé optimalizační algoritmy a jejich efektivní implementace.* Brno: Vysoké učení technické v Brně, Fakulta strojního inženýrství, 2020. 63 s. Vedoucí RNDr. Pavel Popela, Ph.D.

Rozšířený abstrakt

Teoretická část této práce se zabývá oblastí konvexní optimalizace. První kapitola zavádí základní pojmy, jako je konvexní množina a funkce a její optimalita. Společně s nimi jsou také představeny proximální operátory zavedené vzorcem

$$\mathbf{prox}_{\lambda,f}(\mathbf{x}_0) := \underset{\mathbf{x}}{\operatorname{argmin}} \Big(f(\mathbf{x}) + \frac{1}{2}\lambda\|\mathbf{x} - \mathbf{x}_0\|_2^2 \Big),$$

(více v Definici 1.15), na které lze nahlížet jako na minimalizaci f v okolí nějakého bodu $\mathbf{x}_0$. Jejich chování je dále ilustrováno na příkladu proximálního minimalizačního algoritmu.

Druhá kapitola pak odvozuje konvexní optimalizační metodu, ADMM (*Alternating Direction Method of Multipliers*) (2.4) na základě jejích předchůdců, duálního stoupání a metody rozšířeného Lagrangianu. Pro obecný optimalizační problém

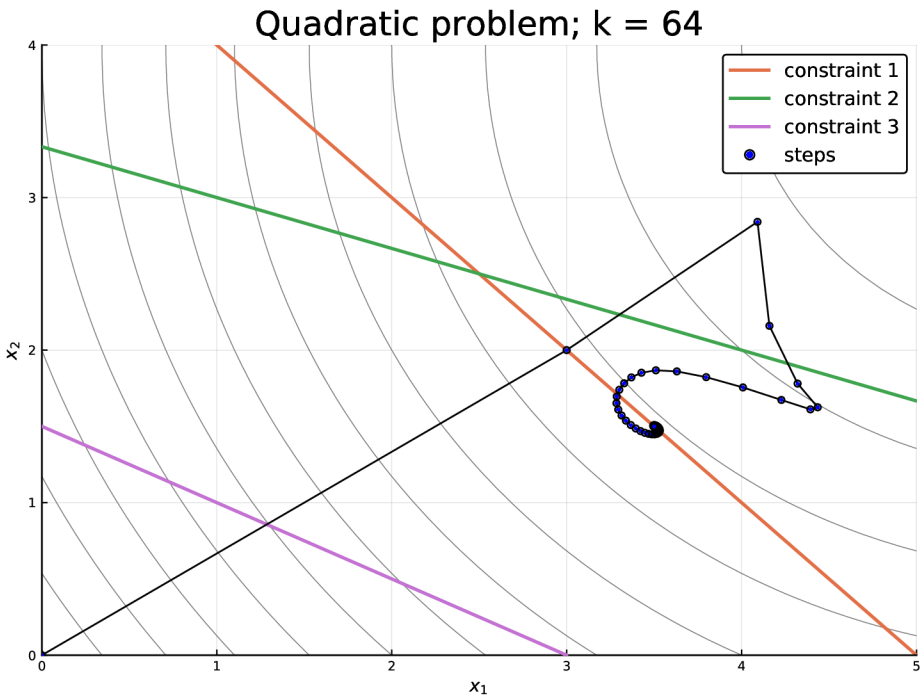
$$\text{minimize } f(\mathbf{x}) + g(\mathbf{x}),$$

je metoda zavedena následovně

$$\begin{aligned}
\mathbf{x}^{k+1} &:= \mathbf{prox}_{\lambda,f}(\mathbf{z}^k - \mathbf{u}^k) \\
\mathbf{z}^{k+1} &:= \mathbf{prox}_{\lambda,g}(\mathbf{x}^{k+1} + \mathbf{u}^k) \\
\mathbf{u}^{k+1} &:= \mathbf{u}^k + \mathbf{x}^{k+1} - \mathbf{z}^{k+1}.
\end{aligned}$$

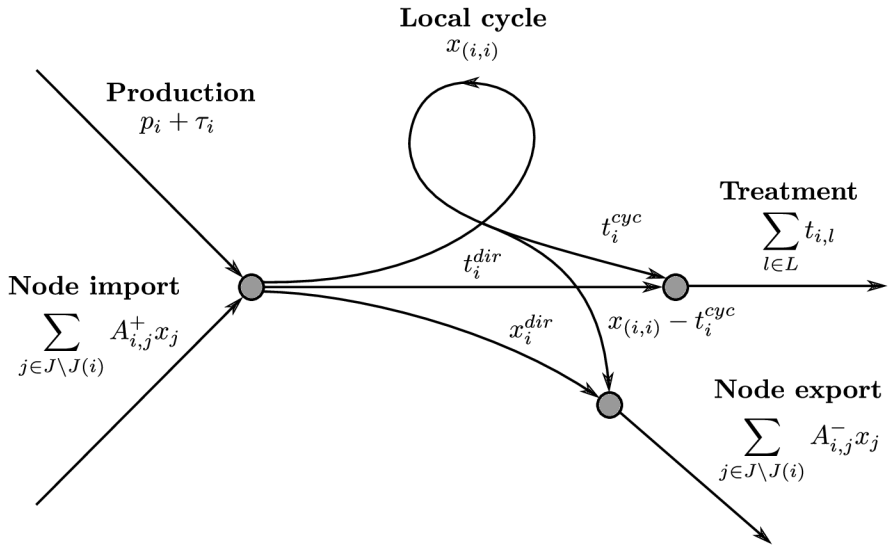
(Pro podmínky a podrobnější vysvětlení viz (2.4)) Tato kapitola dále obsahuje formulaci ADMM pro lineárně omezené problémy (2.8), která je později implementována a aplikována. Také je zmíněna možnost distribuované nebo paralelní modifikace pro pokročilé složené problémy, nazývaná *block splitting method* (2.14).

Výpočtová část se týká implemetace výše uvedené metody ADMM (2.8) a její aplikací. Metoda je ve třetí kapitole implementována v jazyku Julia, zaměřeného na numerické výpočty. Julia obsahuje nástoroje pro snadnou implemetaci dané metody s cílem široké použitelnosti a efektivnosti kódu. Metoda je také ilustrována na dvou malých příkladech, z nichž jeden je zobrazen níže (Obrázek 1). Jedná se o aplikaci na úlohu, kdy účelová funkce je paraboloid (znázorněn na obrázku šedě), avšak optimum hledáme pouze ve vyznačené oblasti (viz sekce 3.5).



Obrázek 1: Trajektorie ADMM pro kvadratický problém (viz. sekce 3.5)

Závěrečná (čtvrtá) kapitola je zaměřena na hlavní optimalizační problém této práce. Jedná se o velký multikriteriální model z oblasti odpadového hospodářství, který je vyvíjen ve spolupráci s výzkumným týmem na Ústavu procesního inženýrství, FSI, VUT v Brně a vychází z jejich předchozí publikace (viz [15]). Problém se týká analýzy nakládání s kaly z čističek odpadních vod, kdy dostupná data představují síť na úrovni obcí s rozšířenou působností (ORP), avšak obsahují nesrovnalosti. Model se pak snaží tyto nesrovnalosti pomocí bilancí sítě a zavedení chybových proměnných srovnat. Protože původní definice modelu obsahovala víceznačné proměnné, vzniklé vlivem agregace dat pro jednotlivé uzly, musel být každý uzel nahrazen zjednodušenou "místní sítí" (Obrázek 2). Pro více informací viz sekce 4.2.1. Analýza běhu implementace modelu odhalila možná místa pro jeho zefektivnění, především lepší práce s "krokovým" parametrem ρ (viz Tabulka 4.1). Také závislost řešení jednotlivých kritérií na jejich relativní váze β (viz Obrázek 4.2) poslouží jako základ pro další výzkum multikriteriální optimalizace, jelikož se často vyskytuje v reálných aplikacích. Výsledky samotného výpočtu pak budou použity v rámci dalších výzkumných projektů.



Obrázek 2: Diagram místní sítě pro agregovaný uzel (viz sekce 4.2.1).

I declare that I have written this master's thesis on the topic *Efficient Implementation of Advanced Optimization Algorithms* independently under the supervision of RNDr. Pavel Popela, Ph.D., using the literature and sources listed in the bibliography.

Bc. Jaroslav Talpa

I would like to thank my supervisor, RNDr. Pavel Popela, Ph.D., for valuable comments and advice on the content of this thesis, as well as my colleagues, Ing. Vlastimil Nevrlý and Ing. Radovan Šomplák, Ph.D. from the Institute of Process Engineering, FME, BUT, for providing me with resources and for cooperation on the case study that is the final result of the computational part of this thesis.

Bc. Jaroslav Talpa

Contents

Introduction

The theoretical part of this thesis concerns itself with the area of convex optimization. First chapter introduces basic concepts, such as convex set and function and its optimality. Also the concept of proximal operators gets covered, with a simple demonstration of its behavior on the proximal minimization algorithm.

Second chapter then builds up a convex optimization method, the alternating direction method of multipliers (ADMM) on its predecessors, the dual ascent and augmented Lagrangian methods. This chapter then features the formulation for linearly constrained problems, which is later being implemented and applied, and also mentions the possibility of distributed or parallelized modification for advanced composed problems, called the block splitting method.

The computational part covers the aforementioned implementation and application of the linearly constrained ADMM. Third chapter focuses on the implementation itself, which aims for general reusability and efficiency of the computation. The method is also illustrated on two example problems.

The final (fourth) chapter focuses on the main optimization problem of this thesis, a large multi-criteria waste management model that is being developed with cooperation from a research team situated at The Institute of Process Engineering, FME, BUT, and is based on their previous work (see [15]). Together with the model definition are also being described the steps needed for the application of the ADMM implementation on it. The behavior of the finished computation is then discussed, together with options for further improvement.

Part I

Theoretical part

Chapter 1

Basic concepts

This chapter covers some basic concepts and definitions connected with the area of convex optimization, which are commonly used throughout the text. It also introduces the notion of proximal operators, which are later used as a major part of the main subject of study for this thesis, the ADMM algorithm.

1.1 Convex sets and functions

Convex sets and functions are, as the name suggests, a cornerstone of the area of convex optimization. One of the reasons, why this area is so significant, is the fact, that convex functions hold a useful property in connection to their extremal points (as discussed in the next section). This then allows to simplify some processes aimed at obtaining these.

Definition 1.1. [1] A set C is *convex* if the line segment between any two points from C also belongs to C, i.e., if $\forall x_1, x_2 \in C$ and $\forall \lambda \in [0; 1]$ holds true

$$\lambda x_1 + (1 - \lambda) x_2 \in C.$$

The definition of convex set then can be in turn used to define a convex function by requiring its epigraph, geometrically the set of points "above" the graph of the function, to be also convex.

Definition 1.2. [2] Let $f : \mathbb{R}^n \longrightarrow \mathbb{R} \cup \{+\infty; -\infty\}$, then the set

$$\textbf{epi } f := \{(\mathbf{x}, t) | \mathbf{x} \in \mathbb{R}^n, t \in \mathbb{R}, t \geq f(\mathbf{x})\}$$

is called the *epigraph* of f.

Definition 1.3. [2] A function $f : \mathbb{R}^n \longrightarrow \mathbb{R} \cup \{+\infty; -\infty\}$ is said to be *convex* if **epi** f is convex.

Note that the function f is defined for the whole of $\mathbb{R}^n$ and can take on extended values of $\pm\infty$[1]. We can *extend* any real-valued convex function φ defined only for a subset $C \subset \mathbb{R}^n$ into this form with the use of a concept of indicator functions.

Definition 1.4. [2, 3] Let $C \subseteq \mathbb{R}^n$ be a set, then its *indicator function* $I_C : \mathbb{R}^n \longrightarrow \{0; +\infty\}$ is defined as

$$I_C(\mathbf{x}) = \begin{cases} 0, & \text{for } \mathbf{x} \in C \\ +\infty, & \text{for } \mathbf{x} \notin C. \end{cases} \tag{1.1}$$

The extension then can be done addition, i.e. $f(\mathbf{x}) = \varphi(\mathbf{x}) + I_C(\mathbf{x})$, where $\varphi : C \subset \mathbb{R}^n \longrightarrow \mathbb{R}$. This process of extension is also tied to the notion of an effective domain of a convex function. [2]

Definition 1.5. [2] Let $f : \mathbb{R}^n \longrightarrow \mathbb{R} \cup \{+\infty; -\infty\}$ be convex, then its *effective domain*, denoted by $\mathbf{dom} f$ is a set on which f is real-valued, i.e.

$$\mathbf{dom} f = \{\mathbf{x} \in \mathbb{R}^n | f(x) < +\infty\}.$$

The original domain C of φ then can be viewed as an effective domain of the newly formed f.

For purposes of further discussion about convex functions, as the extended values $\pm\infty$ allow for some "unwanted" corner cases, the following restrictions on the concept of a convex function will be introduced.

Definition 1.6. [2] A convex function f is said to be *proper* if **epi** f is non-empty and contains no vertical lines, i.e if

$$\exists \mathbf{x} \in \mathbb{R}^n : f(\mathbf{x}) < +\infty$$

and

$$\forall \mathbf{x} \in \mathbb{R}^n : f(\mathbf{x}) > -\infty.$$

Definition 1.7. [3] A convex function f is said to be *closed proper* if f is proper and **epi** f is a closed set.

[1] The introduction of extended values creates some edge cases such as terms $\infty + \infty$. As these are more or less technicalities they will be omitted for brevity. For more information about these see [2].

1.2 Optimality

Finding the optimal point of a function is the main goal of optimization. This section will introduce basic definitions and theorems covering this process.

Definition 1.8. [4] Let $f : C \subseteq \mathbb{R}^n \longrightarrow \mathbb{R}$, then we call $\mathbf{x}^* \in C$ a *global minimum* of f if

$$f(\mathbf{x}) \geq f(x^*);\ \forall \mathbf{x} \in C.$$

Definition 1.9. [4] Let $f : C \subseteq \mathbb{R}^n \longrightarrow \mathbb{R}$, then we call $\mathbf{x}^* \in C$ a *local minimum* of f if

$$\exists N_\epsilon(\mathbf{x}^*) : f(\mathbf{x}) \geq f(x^*);\ \forall \mathbf{x} \in N_\epsilon \cap C.$$

Following theorem highlights the importance of convex functions, as finding a local minimum of a function is usually easier, than finding the global one.

Theorem 1.10. [4] Let $f : C \longrightarrow \mathbb{R}$ be convex and C a convex set. If $\mathbf{x}^* \in \mathbb{R}^n$ is a local minimum of f, then it is also its global minimum.

It should be noted that minimizing a real-valued f over $C \subset \mathbb{R}^n$ is the same as minimizing its proper convex extension over $\mathbb{R}^n$. [2]

Theorem 1.10 guarantees, that for finding a global minimum of a convex function, we just need to search for a spot that does not improve in its neighborhood. Such a condition for optimality of a point can be expressed by the use of a concept of subdifferentials.

Definition 1.11. [2, 4] A vector $\xi \in \mathbb{R}^n$ is called a *subgradient* of convex function f at $\mathbf{x}^* \in \mathbb{R}^n$ if

$$f(\mathbf{x}) \geq f(\mathbf{x}^*) + \xi^\top(\mathbf{x} - \mathbf{x}^*);\ \forall \mathbf{x}.$$

Definition 1.12. [2, 4] The set of all subgradients of f at $\mathbf{x}$ is called the *subdifferential* of f at $\mathbf{x}$, i.e.

$$\partial f(\mathbf{x}) = \{\xi \in \mathbb{R}^n | \xi \text{ is a subgradient of } f \text{ at } \mathbf{x}\}.$$

Naturally, if the function is differentiable, then the subgradients simply reduce to a gradient.

Theorem 1.13. [2, 4] If f is convex and differentiable at $\mathbf{x}$, then $\partial f(\mathbf{x}) = \{\nabla f(\mathbf{x})\}$.

These terms then allow us to formulate a general optimality condition for a convex function.

Theorem 1.14. [2, 4] Let $f : C \longrightarrow \mathbb{R}$ be convex, then $\mathbf{x}^*$ belongs to its global minimum if and only if there exists a subgradient of f at $\mathbf{x}^*$ equal to zero, i.e. $\mathbf{0} \in \partial f(\mathbf{x})$.

1.3 Proximal operators

Definition 1.15. [3] Let $f : \mathbb{R}^n \longrightarrow \mathbb{R} \cup \{+\infty\}$ be a closed proper convex function, then the *proximal operator* $\mathbf{prox}_{\lambda,f} : \mathbb{R}^n \longrightarrow \mathbb{R}^n$ of f with argument $\lambda > 0; \lambda \in \mathbb{R}$ is defined as

$$\mathbf{prox}_{\lambda,f}(\mathbf{x}_0) := \underset{\mathbf{x}}{\operatorname{argmin}} \left(f(\mathbf{x}) + \frac{1}{2}\lambda \|\mathbf{x} - \mathbf{x}_0\|_2^2 \right) \tag{1.2}$$

The proximal operator can be viewed as a minimization step of f in the vicinity of $\mathbf{x_0}$. The ratio between minimization of f and "closeness" to $\mathbf{x_0}$ is then controlled by the argument λ.

Following is a list of some properties of the proximal operator, which are later useful for deriving analytical formulas or closed-forms of concrete proximal operators. [3]

- **Separable sum:** If f is separable across two variables, meaning that $f(\mathbf{x}, \mathbf{y}) = \varphi(\mathbf{x}) + \psi(\mathbf{y})$, then

$$\mathbf{prox}_{\lambda,f}(\mathbf{x}_0, \mathbf{y}_0) = (\mathbf{prox}_{\lambda,\varphi}(\mathbf{x}_0), \mathbf{prox}_{\lambda,\psi}(\mathbf{y}_0)). \tag{1.3}$$

- **Postcomposition:** If $f(\mathbf{x}) = \alpha\varphi(\mathbf{x}) + b; \alpha > 0$, then

$$\mathbf{prox}_{\lambda,f}(\mathbf{x}_0) = \mathbf{prox}_{\alpha\lambda,\varphi}(\mathbf{x}_0). \tag{1.4}$$

- **Affine addition:** If $f(\mathbf{x}) = \varphi(\mathbf{x}) + \mathbf{a}^\top\mathbf{x} + \mathbf{b}$, then

$$\mathbf{prox}_{\lambda,f}(\mathbf{x}_0) = \mathbf{prox}_{\lambda,\varphi}(\mathbf{x}_0 - \lambda\mathbf{a}). \tag{1.5}$$

- **Indicator function:** If $C \in \mathbb{R}^n$ is closed nonempty convex set, then the proximal operator of indicator function I_C reduces to Euclidian projection onto C:

$$\mathbf{prox}_{\lambda,I_C}(\mathbf{x}_0) = \Pi_C(\mathbf{x}_0) = \underset{\mathbf{x}\in C}{\operatorname{argmin}} \|\mathbf{x} - \mathbf{x}_0\|_2. \tag{1.6}$$

1.3.1 Proximal minimization

The fixed point of the proximal operator is the minimizer of f, i.e.

$$\mathbf{prox}_{\lambda,f}(\mathbf{x}^*) = \mathbf{x}^*$$

if and only if $\mathbf{x}^*$ minimizes f. This leads to a simple *proximal minimization algorithm*, which will be used to illustrate the behavior of proximal operators:

$$\mathbf{x}^{k+1} := \mathbf{prox}_{\lambda f}(\mathbf{x}^k),$$

where k is the iteration counter. If f has a minimum, then $\mathbf{x}^k$ converges to one of the minimizers and $f(\mathbf{x}^k)$ to the optimal value (see [3]). When applied to the following explanatory problem

$$\text{minimize } (x_1 - 6)^2 + (x_2 - 4)^2 + I_{\{x_1 \geq 1\}} + I_{\{x_2 \geq 2\}}$$

with starting point $[0, 0]$, the algorithm follows paths for different values of λ as displayed in Figure 1.1.

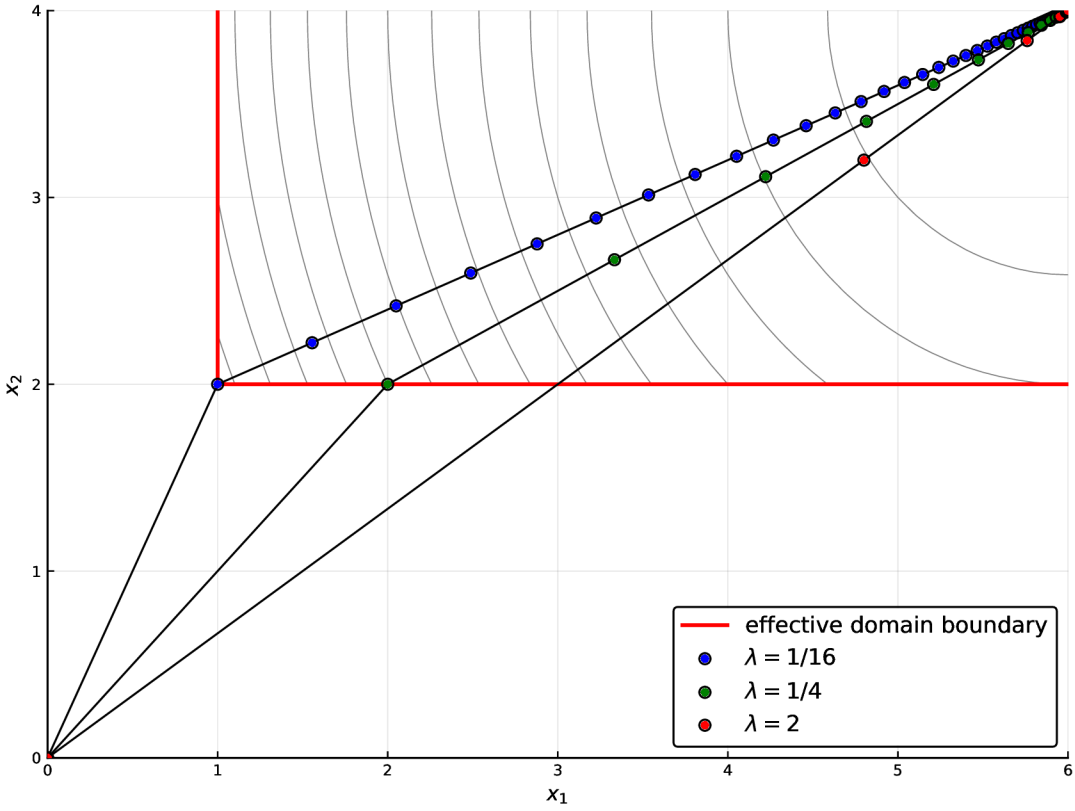


Figure 1.1: Proximal algorithm trajectories for different values of λ. (objective contours are in gray)

As it is clear from the figure, the proximal operator projects steps ending outside of the effective domain of f onto its boundary. For small step size $\lambda = 1/16$ the first step ends up being projected completely. For larger $\lambda = 1/4$ is the step big enough to be projected only in the direction of x_2 and for $\lambda = 2$ happens no projection at all. The next steps the algorithm takes then proceeds to the minimal point $[6, 4]$, however at different pace as dictated by λ.

Chapter 2

Alternating Direction Method of Multipliers

The ADMM algorithm will be introduced in this chapter. Before that, two optimization methods will be briefly mentioned to serve as its precursors, namely the dual ascent method and the augmented Lagrangian method. After that, the modification of the algorithm for linearly constrained problems will be derived, to serve as a basis for the implementation in the following chapter. The block splitting version of ADMM is mentioned at the end of this chapter as an solution for models that needs to be split across multiple processes.

2.1 Precursors

2.1.1 Dual ascent and decomposition

First, lets consider a general convex optimization problem with equality constraints

$$\begin{aligned} &\text{minimize } f(\mathbf{x}) \\ &\quad \text{s.t.: } c_i(\mathbf{x}) = 0; \text{ for } i \in I, \end{aligned} \tag{2.1}$$

where $f : \mathbb{R}^n \longrightarrow \mathbb{R}$ is convex and $c_i : \mathbb{R}^n \longrightarrow \mathbb{R}$; for $i \in I$, are the equality constraints. The *Lagrangian* for (2.1) is then

$$L(\mathbf{x}, \mathbf{y}) = f(\mathbf{x}) + \mathbf{y}^\top \mathbf{c}(\mathbf{x})$$

and the associated *dual problem* is

$$\max_{\mathbf{y}} \inf_{\mathbf{x}} L(\mathbf{x}, \mathbf{y})$$

with *dual variable* $\mathbf{y} \in \mathbb{R}^m$. [5, 6]

The *dual ascent method* steps for solving the dual problem can be described as

$$\begin{aligned}\mathbf{x}^{k+1} &:= \underset{\mathbf{x}}{\operatorname{argmin}}\, L(\mathbf{x}, \mathbf{y}^k)\\ \mathbf{y}^{k+1} &:= \mathbf{y}^k + \alpha^k \mathbf{c}(\mathbf{x}^k),\end{aligned}$$

with $\alpha^k > 0$ as the step size and k as the iteration counter. The importance of this algorithm is in that it allows for a separability of problems, where otherwise separable objective is coupled by constrained variables, e.g. $A\mathbf{x} = \mathbf{b}$. In that case the primal update step can be carried out separately for each partition of f, i.e. the dual ascent method modifies to *dual decomposition* [6]:

$$\begin{aligned}\mathbf{x}_i^{k+1} &:= \underset{\mathbf{x}_i}{\operatorname{argmin}}\, L_i(\mathbf{x}_i, \mathbf{y}^k)\\ \mathbf{y}^{k+1} &:= \mathbf{y}^k + \alpha^k \mathbf{c}(\mathbf{x}^k).\end{aligned}$$

For more information about decomposition see [7] and [8] and for its use in stochastic programming see [9].

2.1.2 Augmented Lagrangian

Definition 2.1. Consider a modification of (2.1):

$$\begin{aligned}\text{minimize } & f(\mathbf{x}) + \frac{\rho}{2}\sum_{i\in\varepsilon} c_i^2(\mathbf{x})\\ \text{s.t.: } & c_i(\mathbf{x}) = 0;\ \text{for } i \in I,\end{aligned} \tag{2.2}$$

then the Lagrangian for this problem is called the *augmented Lagrangian* [4, 6] and is defined as

$$L_\rho(\mathbf{x}, \mathbf{y}) = f(\mathbf{x}) + \sum_{i\in\varepsilon} \mathbf{y}_i c_i(\mathbf{x}) + \frac{\rho}{2}\sum_{i\in\varepsilon} c_i^2(\mathbf{x}). \tag{2.3}$$

The problem then can be solved by the dual ascent method [4]:

$$\begin{aligned}\mathbf{x}^{k+1} &:= \underset{\mathbf{x}}{\operatorname{argmin}}\, L_\rho(\mathbf{x}, \mathbf{y}^k)\\ \mathbf{y}^{k+1} &:= \mathbf{y}^k + \rho^k \mathbf{c}(\mathbf{x}^k).\end{aligned}$$

Note, that the use of the quadratic penalty term in the objective should in many cases lead to better convergence properties (for more information see [4]).

2.2 General algorithm

Definition 2.2. Let $f, g : \mathbb{R}^n \longrightarrow \mathbb{R} \cup \{+\infty\}$ be closed proper convex functions. Then, considering an optimization problem of the form

$$\text{minimize } f(\mathbf{x}) + g(\mathbf{x}),$$

we can formulate the *alternating direction method of multipliers* (ADMM) as follows:

$$\begin{aligned} \mathbf{x}^{k+1} &:= \mathbf{prox}_{\lambda,f}(\mathbf{z}^k - \mathbf{u}^k) \\ \mathbf{z}^{k+1} &:= \mathbf{prox}_{\lambda,g}(\mathbf{x}^{k+1} + \mathbf{u}^k) \\ \mathbf{u}^{k+1} &:= \mathbf{u}^k + \mathbf{x}^{k+1} - \mathbf{z}^{k+1} \end{aligned} \tag{2.4}$$

where k is the iteration counter, $\lambda > 0$, $\mathbf{u}, \mathbf{z} \in \mathbb{R}^n$ are newly introduced variables (see (2.5) and the following steps) and **prox** is the proximal operator (see Definition 1.15). [3]

The ADMM algorithm, sometimes called the *Douglas-Rachford operator splitting* is a convex optimization method with very general convergence conditions (see [6] §3.2). It aims to combine the decomposability of dual ascent and the convergence properties of the augmented Lagrangian. The objective terms, which both may encode constraints, are handled separately and only by the use of their proximal operators. This means that ADMM is best suited for problems where proximal operators of f and g are evaluated easily (as is the case for many functions found in practical applications), however proximal of $f + g$ is not. [3, 6, 10]

The algorithm (2.4) can be derived in these following steps. Rewriting the original problem $\min_{\mathbf{x}}\{f(\mathbf{x}) + g(\mathbf{x})\}$, with the introduction of new optimization variable $\mathbf{z} \in \mathbb{R}^n$, as

$$\begin{aligned} &\text{minimize } f(\mathbf{x}) + g(\mathbf{z}) \\ &\text{s.t.: } \mathbf{x} - \mathbf{z} = \mathbf{0}, \end{aligned}$$

sometimes referred to as a *consensus form* because of the *consensus constraint* $\mathbf{x} - \mathbf{z} = 0$, then yields an augmented Lagrangian of a form

$$L_\rho(\mathbf{x}, \mathbf{z}, \mathbf{y}) = f(\mathbf{x}) + g(\mathbf{z}) + \mathbf{y}^\top(\mathbf{x} - \mathbf{z}) + \frac{\rho}{2}\|\mathbf{x} - \mathbf{z}\|_2^2. \tag{2.5}$$

By applying the dual ascent method, we obtain

$$\begin{aligned} (\mathbf{x}^{k+1}, \mathbf{z}^{k+1}) &:= \operatorname*{argmin}_{\mathbf{x},\mathbf{z}} L_\rho(\mathbf{x}, \mathbf{z}, \mathbf{y}^k) \\ \mathbf{y}^{k+1} &:= \mathbf{y}^k + \rho^k(\mathbf{x}^{k+1} - \mathbf{z}^{k+1}). \end{aligned}$$

The optimization in the first step can be done separately (hence the term *Alternating* in the name of the method), similarly to a single pass of *Gauss-Seidel* method:

$$\begin{aligned}\mathbf{x}^{k+1} &:= \operatorname*{argmin}_{\mathbf{x}} L_\rho(\mathbf{x}, \mathbf{z}^k, \mathbf{y}^k)\\ \mathbf{z}^{k+1} &:= \operatorname*{argmin}_{\mathbf{z}} L_\rho(\mathbf{x}^{k+1}, \mathbf{z}, \mathbf{y}^k)\\ \mathbf{y}^{k+1} &:= \mathbf{y}^k + \rho^k(\mathbf{x}^{k+1} - \mathbf{z}^{k+1}).\end{aligned}$$

By substituting (2.5) and omitting constant terms, as these do not affect the minimization results, we obtain

$$\begin{aligned}\mathbf{x}^{k+1} &:= \operatorname*{argmin}_{\mathbf{x}} \Big(f(\mathbf{x}) + \mathbf{y}^{k\top}\mathbf{x} + \frac{\rho}{2}\|\mathbf{x} - \mathbf{z}^k\|_2^2\Big)\\ \mathbf{z}^{k+1} &:= \operatorname*{argmin}_{\mathbf{z}} \Big(g(\mathbf{z}) - \mathbf{y}^{k\top}\mathbf{z} + \frac{\rho}{2}\|\mathbf{x}^{k+1} - \mathbf{z}\|_2^2\Big)\\ \mathbf{y}^{k+1} &:= \mathbf{y}^k + \rho^k(\mathbf{x}^{k+1} - \mathbf{z}^{k+1}),\end{aligned}$$

and by including the linear terms into the norm we arrive at

$$\begin{aligned}\mathbf{x}^{k+1} &:= \operatorname*{argmin}_{\mathbf{x}} \Big(f(\mathbf{x}) + \frac{\rho}{2}\|\mathbf{x} - \mathbf{z}^k + \frac{1}{\rho}\mathbf{y}^k\|_2^2\Big)\\ \mathbf{z}^{k+1} &:= \operatorname*{argmin}_{\mathbf{z}} \Big(g(\mathbf{z}) + \frac{\rho}{2}\|\mathbf{x}^{k+1} - \mathbf{z} - \frac{1}{\rho}\mathbf{y}^k\|_2^2\Big)\\ \mathbf{y}^{k+1} &:= \mathbf{y}^k + \rho^k(\mathbf{x}^{k+1} - \mathbf{z}^{k+1}).\end{aligned}$$

Finally, by substituting $\mathbf{u}^k = \frac{1}{\rho}\mathbf{y}^k$ and $\lambda = \frac{1}{\rho}$ we obtain the general form of ADMM, the same as in (2.4) [3, 6], because

$$\mathbf{prox}_{\lambda,\varphi(\mathbf{x})}(\mathbf{x}_0) = \operatorname*{argmin}_{\mathbf{x}} \Big(\varphi(\mathbf{x}) + \frac{1}{2}\lambda\|\mathbf{x} - \mathbf{x}_0\|_2^2\Big).$$

2.3 Formulation for problems with linear constraints

The general ADMM algorithm (2.4) will be now transformed for general linearly constrained problem

$$\begin{aligned}&\text{minimize } \varphi(\mathbf{x})\\ &\text{s.t.: } A\mathbf{x} = \mathbf{b}\end{aligned} \tag{2.6}$$

where φ is closed proper convex. The reason for doing this is that the model described in Chapter 4 can be transformed into this form, thus allowing the use of ADMM. First, the problem (2.6) needs to be modified into

$$\text{minimize } f(\mathbf{z}) + I_{\{A\mathbf{x}=\mathbf{y}\}}(\mathbf{z}), \tag{2.7}$$

where $\mathbf{z} = (\mathbf{x}, \mathbf{y})$ is a collective vector for the original variables $\mathbf{x}$ and newly introduced right-hand-side (RHS) variables $\mathbf{y}$, $f(\mathbf{z}) = \varphi(\mathbf{x}) + I_{\{\mathbf{y}=\mathbf{b}\}}(\mathbf{y})$ is a new objective function with RHS variables $\mathbf{y}$ being constrained to the RHS vector b and $I_{\{A\mathbf{x}=\mathbf{y}\}}$ is the indicator function of the linear constraints, i.e. of the set $\{(\mathbf{x}, \mathbf{y});\ A\mathbf{x} = \mathbf{y}\}$.

Applying the general form algorithm (2.4) on the modified linearly constrained problem (2.7) then results in ADMM formulation for problems with linear constraints (as $I_{\{A\mathbf{x}=\mathbf{y}\}}(\mathbf{z})$ represents $g(\mathbf{x})$ from (2.4)):

$$\begin{aligned}
\mathbf{x}^{k+1/2} &:= \mathbf{prox}_{\lambda,\varphi}(\mathbf{x}^k - \tilde{\mathbf{x}}^k) \\
\mathbf{y}^{k+1/2} &:= \Pi_{\{\mathbf{b}\}}(\mathbf{y}^k - \tilde{\mathbf{y}}^k) \\
(\mathbf{x}^{k+1}, \mathbf{y}^{k+1}) &:= \Pi_A(\mathbf{x}^{k+1/2} + \tilde{\mathbf{x}}^k, \mathbf{y}^{k+1/2} + \tilde{\mathbf{y}}^k) \\
\tilde{\mathbf{x}}^{k+1} &:= \tilde{\mathbf{x}}^k + \mathbf{x}^{k+1/2} - \mathbf{x}^{k+1} \\
\tilde{\mathbf{y}}^{k+1} &:= \tilde{\mathbf{y}}^k + \mathbf{y}^{k+1/2} - \mathbf{y}^{k+1}
\end{aligned} \tag{2.8}$$

As the function f can be separated into $\varphi(\mathbf{x})$ and $I_{\{\mathbf{y}=\mathbf{b}\}}(\mathbf{y})$, using (1.3), its proximal operator is handled also separately. Because of (1.6), the proximal of indicator function is a simple projection onto vector $\mathbf{b}$. Similarly, proximal of the second function is also a projection, in this case onto the graph of the linear operator A. As the variable vectors x and y do not overlap, the last (dual) update step can also be handled separately. [11]

For example problems illustrating this formulation (2.8) of ADMM see section 3.3.

2.3.1 Efficient graph projection

The graph projection $\Pi_A(\mathbf{c}, \mathbf{d})$ (with general arguments) from the third step of (2.8) is equivalent to solving a minimization problem with variables $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{y} \in \mathbb{R}^m$,

$$\begin{aligned}
\text{minimize } & \frac{1}{2}\|\mathbf{x} - \mathbf{c}\|_2^2 + \frac{1}{2}\|\mathbf{y} - \mathbf{d}\|_2^2 \\
\text{s.t.: } & A\mathbf{x} = \mathbf{y}.
\end{aligned} \tag{2.9}$$

By applying the KKT optimality conditions [4] with dual variable λ,

$$\begin{aligned}\mathbf{0} &= \frac{1}{2}\nabla(\|\mathbf{x}-\mathbf{c}\|_2^2 + \|\mathbf{y}-\mathbf{d}\|_2^2) - \lambda\nabla(\mathbf{y}-A\mathbf{x})\\ \mathbf{0} &= (\mathbf{x}-\mathbf{c}, \mathbf{y}-\mathbf{d}) + \lambda(A, -\mathbf{1}),\end{aligned}$$

we obtain the following system

$$\begin{pmatrix} I & 0 & A^\top \\ 0 & I & -I \\ A & -I & 0 \end{pmatrix} \begin{pmatrix} \mathbf{x} \\ \mathbf{y} \\ \lambda \end{pmatrix} = \begin{pmatrix} \mathbf{c} \\ \mathbf{d} \\ \mathbf{0} \end{pmatrix}. \tag{2.10}$$

By expressing $\lambda = \mathbf{y} - \mathbf{d}$ from the second equation and substituting it into the first one we obtain

$$(I\ A^\top\ 0) \begin{pmatrix} \mathbf{x} \\ \mathbf{y} \\ \lambda \end{pmatrix} = \mathbf{c} + A^\top\mathbf{d}.$$

This allows to simplify (2.10) into

$$\begin{pmatrix} I & A^\top \\ A & -I \end{pmatrix} \begin{pmatrix} \mathbf{x} \\ \mathbf{y} \end{pmatrix} = \begin{pmatrix} \mathbf{c} + A^\top\mathbf{d} \\ \mathbf{0} \end{pmatrix}. \tag{2.11}$$

This system can be then solved using block elimination. By expressing $\mathbf{x} = A^{-1}\mathbf{y}$ from the second equation and then substituting into the first one,

$$\begin{aligned} A^{-1}\mathbf{y} + A^\top\mathbf{y} &= \mathbf{c} + A^\top\mathbf{d} \\ \mathbf{y} + AA^\top\mathbf{y} &= A\mathbf{c} + AA^\top\mathbf{d} \\ \mathbf{y} &= (I + AA^\top)^{-1}(A\mathbf{c} + AA^\top\mathbf{d}), \end{aligned}$$

we obtain the following two-step solution (with $\mathbf{x}$ from the first equation for already computed $\mathbf{y}$)

$$\begin{aligned} \mathbf{y} &:= (I + AA^\top)^{-1}(A\mathbf{c} + AA^\top\mathbf{d}) \\ \mathbf{x} &:= \mathbf{c} + A^\top\mathbf{d} - \mathbf{y}. \end{aligned} \tag{2.12}$$

Evaluation of the first step can be done via *Cholesky factorization* $I + AA^\top = LL^\top$. Because this factorization depends only on the constraint matrix A, it can be created only once and then reused in subsequent iterations, or even in other models with the same matrix A but with different objective functions[1]. [11]

[1] As it is done in Chapter 4, when dealing with a multi-criteria model.

2.4 ADMM Block splitting

When facing problems with large datasets, the resulting matrix may be so big, that handling the model as a whole may be impractical or outright impossible. This can be solved by a problem partitioning, where each partition is handled by a separate process and each resulting block of the constraint matrix is required only locally.

If the objective function φ from (2.6) is *block separable*, meaning

$$\varphi(\mathbf{x}) = \sum_{j=1}^{N} \varphi_j(\mathbf{x}_j),$$

we can divide the original problem (2.7) across M block rows and N block columns, with the introduction of new "local" variables $\mathbf{x}_{i,j}$ and $\mathbf{y}_{i,j}$ for each block (i, j), formulating an equivalent form

$$\begin{aligned} \text{minimize } & \sum_{j=1}^{N} \varphi_j(\mathbf{x}_j) + \sum_{i=1}^{M} I_{\{\mathbf{y}_i = \mathbf{b}_i\}}(\mathbf{y}_i) + \sum_{i=1}^{M}\sum_{j=1}^{N} I_{\{A_{i,j}\mathbf{x}_{i,j} = \mathbf{y}_{i,j}\}} \\ \text{s.t.: } & \mathbf{x}_j = \mathbf{x}_{i,j} \\ & \mathbf{y}_i = \sum_{j=1}^{N} \mathbf{y}_{i,j}. \end{aligned} \tag{2.13}$$

This form then leads to a reformulation of (2.8) into the *block splitting algorithm* [11],

$$\begin{aligned} \mathbf{x}_j^{k+1/2} &:= \mathbf{prox}_{\lambda,\varphi_j}(\mathbf{x}_j^k - \tilde{\mathbf{x}}_j^k) \\ \mathbf{y}_i^{k+1/2} &:= \Pi_{\{\mathbf{b}_i\}}(\mathbf{y}_i^k - \tilde{\mathbf{y}}_i^k) \\ (\mathbf{x}_{i,j}^{k+1}, \mathbf{y}_{i,j}^{k+1}) &:= \Pi_{A_{i,j}}(\mathbf{x}_{i,j}^{k+1/2} - \tilde{\mathbf{x}}_{i,j}^k, \mathbf{y}_{i,j}^{k+1/2} - \tilde{\mathbf{y}}_{i,j}^k) \\ (\mathbf{x}_j^{k+1}, \{\mathbf{x}_{i,j}^{k+1}\}_{i=1}^M) &:= \mathbf{avg}(\mathbf{x}_j^{k+1} + \tilde{\mathbf{x}}_j^k, \{\mathbf{x}_{i,j}^{k+1} + \tilde{\mathbf{x}}_{i,j}^k\}_{i=1}^M) \\ (\mathbf{y}_i^{k+1}, \{\mathbf{y}_{i,j}^{k+1}\}_{j=1}^N) &:= \mathbf{exch}(\mathbf{y}_i^{k+1} + \tilde{\mathbf{y}}_i^k, \{\mathbf{y}_{i,j}^{k+1} + \tilde{\mathbf{y}}_{i,j}^k\}_{j=1}^N) \\ \tilde{\mathbf{z}}^{k+1} &:= \tilde{\mathbf{z}}^k + \mathbf{z}^{k+1/2} - \mathbf{z}^{k+1}, \end{aligned} \tag{2.14}$$

where the dual variables $\tilde{\mathbf{x}}$ and $\tilde{\mathbf{y}}$ from (2.8) are also partitioned and have their "block-local" counterparts and $\mathbf{z}$ is a vector of all $\mathbf{x}_j$, $\mathbf{y}_i$, $\mathbf{x}_{i,j}$, $\mathbf{y}_{i,j}$ (similarly for $\tilde{\mathbf{z}}$); where **avg** is an elementwise average (done on multiple vectors but denoted collectively in one line), setting the values of the output vector to the elementwise average of the input vector; **exch** is an *exchange*

operator[11], defined for $\mathbf{exch}(\mathbf{c}, \{\mathbf{c}_j\}_{j=1}^N)$ as

$$\mathbf{y}_{i,j} \coloneqq \mathbf{c}_j + \frac{\mathbf{c} - \sum_{j=1}^N \mathbf{c}_j}{N+1}$$

$$\mathbf{y}_i \coloneqq \mathbf{c} - \frac{\mathbf{c} - \sum_{j=1}^N \mathbf{c}_j}{N+1}.$$

The **avg** (consensus) and **exch** (exchange) operators represent the projections onto the constraints of (2.13). It should be noted that the graph projection onto $A_{i,j}$ is done only with the "local" variables and thus can be parallelized. [11]

As the model in Chapter 4 does not require the partitioning of A, the block splitting algorithm is mentioned only in the theoretical part of this thesis. It can, however, be employed on it in the future if the need arises.

Part II

Computational part

Chapter 3

Implementing the ADMM algorithm

3.1 The Julia programming language

As the implementation of the ADMM algorithm described in this chapter is written in the Julia programing language, this section will briefly describe the language itself and a few of its features, that are used in the implementation.

Julia [13] is a high-level, high-performance, dynamic language supporting object-oriented and functional paradigms. While allowing for wide variety of applications, it is best-suited for numerical and scientific computations. Function calling and overloading adheres to the *multiple dispatch* scheme [12], meaning that method overloads are chosen based on the runtime (ie. dynamic) type of the call's arguments. The type system is also dynamic, meaning that types of variables and function arguments need not to be explicitly declared in the code, however defining them may lead to shorter execution times as the compiler may produce a more lean bytecode. [13]

The Julia language uses a just-in-time compiler (built around LLVM compiler), that, roughly speaking, converts the source code into its abstract syntax tree (AST) representation [13], which is then compiled into the platform's native code, while optimizing the result at each step. This means that Julia is not an interpreted language, but rather compiles the source code as needed. [12]

The intermediate AST representation of the code also allows for LISP-like macros, in Julia marked with the symbol "`@`". These are functions, executed by the compiler itself, that take in an AST of the input and return another AST, usually but not exclusively, a modification of the original. This result is then passed along the compiler's pipeline in place of the original macro

call. [12, 13]

3.2 Main algorithm

The following implementation of the ADMM algorithm was designed with generality and re-usability in mind. Proximal operator computations are delegated to the user, thus allowing for more general problems. This also allows the user, in many cases, to simplify and speed-up the proximal computation greatly, which cannot be done (or would be very difficult) by a general algorithm. This approach is later used to reduce the solving of underlying proximal operator minimization to a few lines of direct computation, e.g for the example problems (3.2) and (3.5) .

3.2.1 Input arguments

The function `ADMM_LCP!` [1] contains the implementation of the algorithm, specifically of its formulation for linearly constrained problems (2.8). It has multiple input arguments describing the model to be solved and precision parameters for the stopping criterion.

```
@inbounds function ADMM_LCP!(
  x´::AbstractArray{<:Real,1},  # x´: vector of variables
  A::AbstractArray{<:Real,2},   #  A: constraint coefficients
  F,                            #  F: cholesky factorization of (I+AA')
  b::AbstractArray{<:Real,1},   #  b: RHS vector
  max_iter::Integer,            # max_iter: maximum iteration count
  ρ::Real,                      #  ρ: stopping criterion weight
  εabs::Real,                   # εabs: absolute error
  εrel::Real,                   # εrel: relative error
  prox!;                        # prox: proximal operator
  )
```

The parameter `x´` serves as so-called ”input-output” argument for the model variables. The user is expected to allocate this vector beforehand and then retrieve the solution from it after the function returns. Matrix `A` represents the constraints matrix. Because only the `::AbstractArray` type is being required here, the sparse matrix implementation is also allowed, thus saving memory in the case of large models. `F` then contains the *Cholesky factorization* of matrix $I + A \cdot A^\top$, which is later used for more efficient computation of

[1] To adhere to the Julia language naming convention, the name of the function ends with an exclamation mark ”!”, as it modifies contents of one or more of its input arguments, namely the variables vector `x´`. See: https://docs.julialang.org/en/latest/manual/style-guide/#Append-!-to-names-of-functions-that-modify-their-arguments-1

projection onto the constraint matrix (see Section 2.3.1). Because this factorization depends only on constraints and can be reused with different objective functions, it is being requested here from the user to allow for caching, ie. to let the user compute it only once and save it, as this can be a time consuming operation. Because the sparse and non-sparse implementations do not have common supertype, the type of this parameter is left unspecified. Vector `b` is simply the right-hand-side counterpart to constraints matrix `A`.

Following parameters are used to control the precision and running time of the algorithm. As a hard stop, the `max_iter` value specifies maximum allowed number of iterations the program will perform. Parameters `εabs`, `εrel` and `ρ` are used for the precision based stopping criterion. The value `ρ` is also used as the argument for the proximal operators.

Lastly the `prox!` argument is a user-defined function responsible for evaluation of the proximal operator of the objective function. The function `prox!` is expected to be in this general form:

```
function prox!(ρ,x,x0)
  # Evaluation of the proximal operator, with x0 being the input
  # and with x being overwritten with the result of the computation.
  return
end
```

The parameter `ρ` is being passed for convenience and has the same value as the one from the main function's argument list.

3.2.2 Initialization and main loop

In order to reduce the number of memory allocations and thus the garbage collector running time, all of the variables are created only once and then changed in place. This also holds true for some intermediate results, so the amount of newly allocated arrays is as small as possible. Most of the variables are initialized with a value of zero, with the exception of vector `y´`, which is created as a copy of the RHS vector `b`, and matrix `AAt`, which is a cached result of $A \cdot A^\top$. This initialization block is omitted in the source code being presented.

```
41    k = 1
42    while k <= max_iter
43      # primal update
44      prox!(ρ, x´, x - x̃ )
45
46      # projection onto constraints
47      @. c = x´ + x̃
48      @. d = y´ + ỹ
```

```
49      y .= F \ (A*c .+ AAt*d)
50      x .= c .+ A'*(d.-y)
51
52      # dual update
53      @. x̃ =  c - x
54      @. ỹ =  d - y
55
56      # stopping criterion
57      z′[xr] .= x′; z′[yr] .= y′
58      z[xr] .= x ; z[yr] .= y
59      z̃[xr] .= x̃ ; z̃[yr] .= ỹ
60      εpri  = √n * εabs + εrel * max(norm(z′), norm(z))
61      εdual = √n * εabs + εrel * norm(ρ*z̃)
62      res = norm(z′ - z)
63      res_dual = norm(-ρ*(z-z_last))
64
65      if (res <= εpri) && (res_dual <= εdual)
66        break
67      end
68
69      z_last .= z
70      k += 1
71    end
72
73    return k, x̃, ỹ
74  end
```

The first part of the loop updates the primal variable `x′` (equiv. to $\mathbf{x}^{k+1/2}$), employing the provided proximal operator `prox!`. Because the other primal variable `y′` (equiv. to $\mathbf{y}^{k+1/2}$) would be just updated with the values of `b` and it is not being changed anywhere else, this step is skipped completely. The second part performs the projection onto the constraint matrix, using the backsolve "\" operator with the provided Cholesky factorization of $(I + A \cdot A^\top)$. Lastly the dual variables are updated, using the intermediate variables `c` and `d` from the previous step. The stopping criterion (as described in [11] and [6]) is then checked and if both residuals are small enough, the loop terminates. The function then returns the number of iterations `k` and both dual vectors[2].

3.2.3 Code optimization techniques

A big problem when dealing with computations on vectors is a memory allocation and subsequent garbage collection. For example, when two vectors

[2] the first primal vector `x′` is already available to the user and `y′` is always equal to `b`, so they need not to be returned

are being added, the result needs to be stored in a third, newly allocated array. It is clear, how this can quickly grow out of hands when there are many vector operations being performed, such as in the implementation in this chapter. In such cases a good practice in Julia is to use the vectorized operations denoted by ".", e.g. `x .+ y` or `abs.(x)`. These operations are then applied on each member of the vector separately, and moreover, if there are more of these in one expression, they all get fused together and evaluated in one single loop. This means that there are no intermediate array allocations needed. Vectorized assignment, i.e. `x .= y`, writes the right hand side `y` into the target vector `x` member by member, rather then reassigning the variable x to reference y instead. This allows to preallocate the space needed for the results. Because attaching a dot to every operation in an expression is cumbersome and results in a less readable code, the macro `@.` exists, converting each operation found in its input into their vectorized version.

The other thing, that results in unnecessary array allocation, is accessing only a portion of the array, e.q. `x[1:5]`. This results in a new array being created, that is a copy of the requested part. For this occasion, Julia offers a macro called `@views`, which converts these "slice" expressions into "views", that simply just hold a reference to the original array, resulting in no data copied. This macro is used e.g. in implementations of the proximal operator formulas in section 3.3, as within these, each block of variables from the common variable vector needs to be handled separately.

Lastly, when working with arrays with known dimensions, a macro called `@inbounds` can be applied to an expression or a whole function. This then means, that when accessing arrays by an index, the value of the index is not checked to be within the array's bounds. This of course speeds up array operations, but needs to be used carefully, as out of bounds indexes may lead to crashes or data corruption. [14]

3.3 Example problems

In this section, the aforementioned ADMM implementation will be applied to two simple optimization problems. Both problems feature the same linear constraints, but differ in objective functions. The constraints, defined by inequalities

$$\begin{aligned} x_1 + \quad x_2 &\leq 5 \\ x_1 + \; 3x_2 &\leq 10 \\ x_1 + \; 2x_2 &\geq 3 \\ x_1; x_2 &\geq 0 \end{aligned} \tag{3.1}$$

create a convex polygon as the feasible set of the problem. They are set up in such a way that the polygon does not include the starting point $(0, 0)$

In order to use the ADMM algorithm, the constraints (3.1) are transformed into equalities by introducing slack variables x_3, x_4 and x_5. The non-negativity constraints are encoded into the objective in the form of indicator function $I_{\{\mathbf{x}\geq 0\}}$, resulting in the following model

$$\begin{aligned} \text{minimize} \quad & z(\mathbf{x}) + I_{\{\mathbf{x}\geq 0\}}(\mathbf{x}) \\ \text{s.t.:} \quad & x_1 + x_2 + x_3 = 5 \\ & x_1 + 3x_2 + x_4 = 10 \\ & x_1 + 2x_2 - x_5 = 3 \end{aligned}$$

3.3.1 Linear objective function

In the first example, a linear function with the added indicator for the non-negativity constraints serves as the objective:

$$z_1(\mathbf{x}) = -x_1 - 2x_2 + I_{\{\mathbf{x}\geq 0\}}(\mathbf{x}). \tag{3.2}$$

The proximal operator for the function as a whole need not to be derived. Splitting the function across individual variables and using (1.3) allows us to create and evaluate proximal operators only for the respective scalar terms. For the slack variables, the only term in z_1 is the indicator $I_{\{x\geq 0\}}$. The proximal operator then reduces to simple projection (see (1.6)) onto positive reals:

$$\mathbf{prox}_{\rho, I_{\{x\geq 0\}}}(x_0) = \max\{0; x_0\}. \tag{3.3}$$

For the original variables, included in terms with form of $\varphi_1(x) = cx + I_{\{x\geq 0\}}$, the affine addition property (1.5) can be used:

$$\mathbf{prox}_{\rho, \varphi_1}(x_0) = \mathbf{prox}_{\rho, I_{\{x\geq 0\}}}(x_0 - \rho c).$$

Then, by substituting into (3.3), we obtain

$$\mathbf{prox}_{\rho, \varphi_1}(x_0) = \max\{0; x_0 - \rho c\}. \tag{3.4}$$

The operators (3.3) and (3.4) can then be encoded for the algorithm like in this code snippet:

```julia
function prox1!(ρ,x,x0)
  x[1] = max(0, x0[1] +  ρ)
  x[2] = max(0, x0[2] + 2ρ)
  @. x[3:5] = max(0, @view x0[3:5])
  return
end
```

Running the algorithm, with parameters `ρ=1/2`, `εabs=1e-6` and `εrel=1e-4`, converges after 80 iterations, arriving at the inexact solution $(x_1, x_2) = (2.49741, 2.50103)$ (exact solution being $(2.5, 2.5)$). The algorithm trajectory is shown in Figure 3.1 on the next page. It can be observed that the algorithm first takes big steps approximately along the gradient of the objective and then in decreasing manner oscillates along the optimum point, almost parallel to the gradient. It should be noted, that ADMM is being applied to the linear problem only to serve as an example and that it is not the ideal method for this class of problems.

3.3.2 Quadratic objective function

In the second example, quadratic terms are used instead of the linear ones

$$z_2(\mathbf{x}) = (x_1 - 6)^2 + (x_2 - 4)^2 + I_{\{\mathbf{x}\geq 0\}}(\mathbf{x}), \tag{3.5}$$

representing a paraboloid centered at the point $(4, 6)$. Following the same strategy as in the previous example, the proximal operator needs to be derived only for scalar terms with individual variables. Slack variables are again only present in the indicator function, allowing for the usage of (3.3).

The original variables are included in terms of the form

$$\varphi_2(x) = (x + b)^2 + I_{\{x\geq 0\}}.$$

Following Definition 1.15 for proximal operators, we obtain

$$\mathbf{prox}_{\rho,\varphi_2}(x_0) = \underset{x}{\operatorname{argmin}} \left((x + b)^2 + I_{\{x\geq 0\}}(x) + \frac{1}{2\rho}\|x - x_0\|_2^2\right).$$

The result of this operator can never be negative, because then the indicator term and the whole expression would attain the value of $+\infty$. Because otherwise for non-negative x the indicator term returns 0, the expression can be simplified into

$$\mathbf{prox}_{\rho,\varphi_2}(x_0) = \max\{0; \underset{x}{\operatorname{argmin}} \left((x + b)^2 + \frac{1}{2\rho}\|x - x_0\|_2^2\right)\}.$$

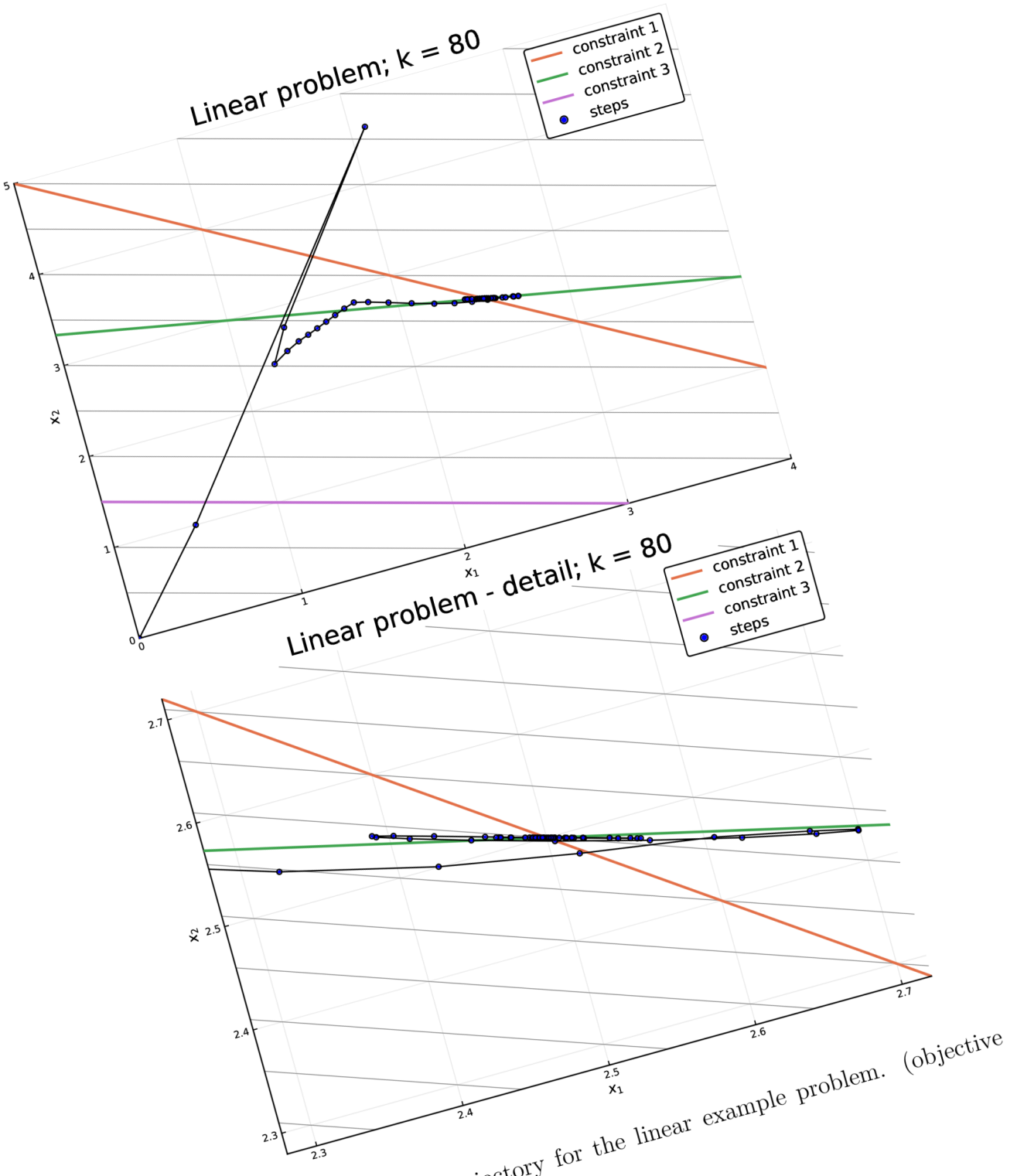


Figure 3.1: ADMM trajectory for the linear example problem. (objective contours are in gray)

By simplifying the optimality condition for the inner expression

$$\begin{aligned} 0 &= \nabla\Big((x+b)^2 + \frac{1}{2\rho}\|x - x_0\|_2^2\Big) \\ 0 &= 2(x+b) + \frac{x - x_0}{\rho} \\ x^* &= \frac{x_0 - 2\rho b}{2\rho + 1}, \end{aligned}$$

we obtain the formula for the proximal operator

$$\mathbf{prox}_{\rho,\varphi_2}(x_0) = \max\left\{0; \frac{x_0 - 2\rho b}{2\rho + 1}\right\}. \tag{3.6}$$

Similarly to the previous example, the operators (3.3) and (3.6) are encoded as follows:

```
function prox1!(ρ,x,x0)
  x[1] = max(0, (x0[1] + 12ρ)/(2ρ + 1))
  x[2] = max(0, (x0[2] +  8ρ)/(2ρ + 1))
  @. x[3:5] = max(0, @view x0[3:5])
  return
end
```

Running the algorithm, with the same parameters `ρ=1/2`, `εabs=1e-6` and `εrel=1e-4`, converges after 64 iterations, arriving at the inexact solution $(x_1, x_2) = (3.50068, 1.49961)$ (exact solution being $(3.5, 1.5)$). The algorithm trajectory is shown below in Figure 3.2. As in the previous example, the algorithm takes a few big steps along the objective's gradient, but then converges in a spiral pattern to the optimal point, rather then oscillating along one line.

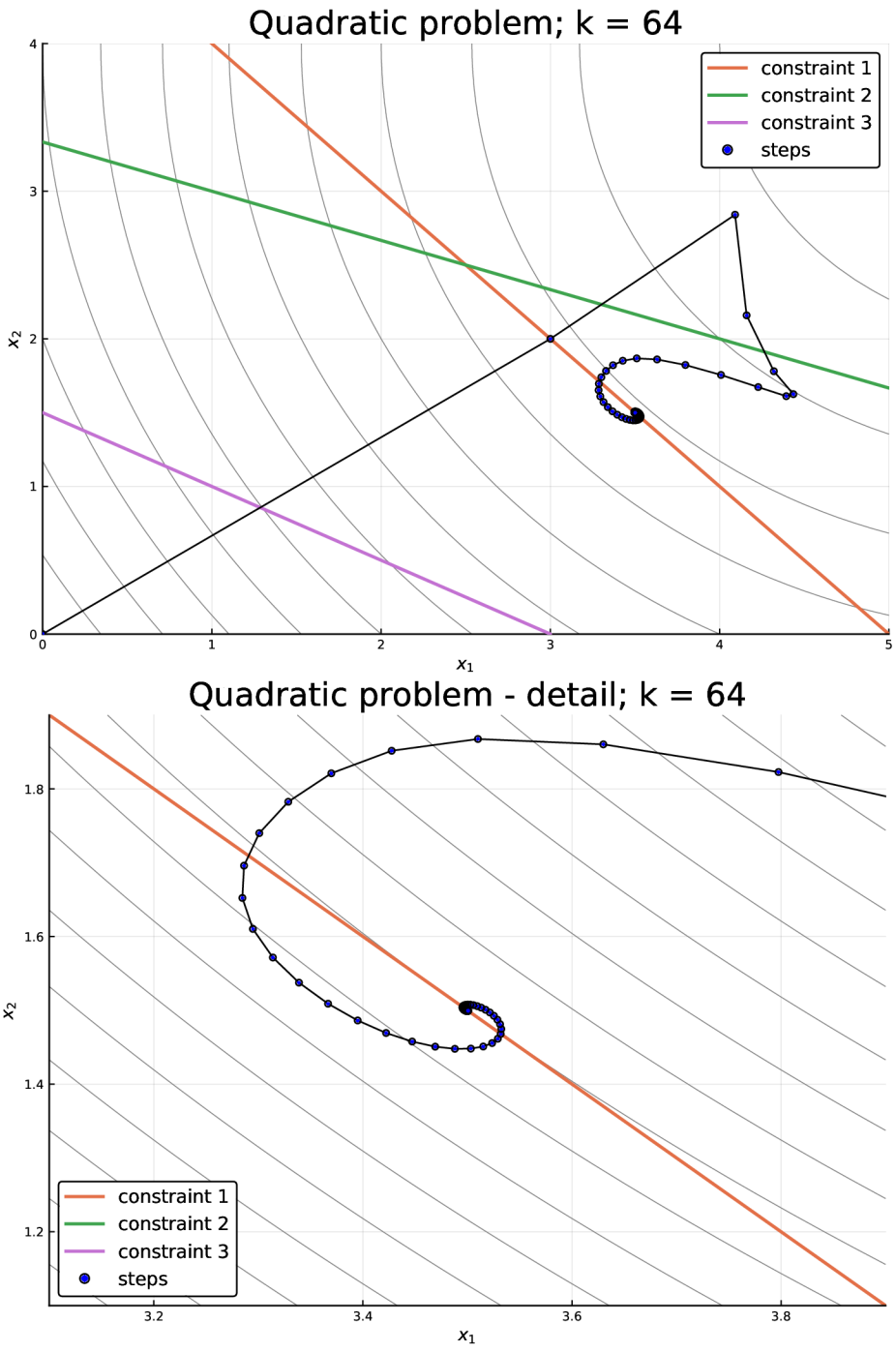


Figure 3.2: ADMM trajectory for the quadratic example problem. (objective contours are in gray)

Chapter 4

Case study

4.1 Introduction

The case study in this thesis has been done in cooperation with the research team based at Institute of Process Engineering, Faculty of Mechanical Engineering, BUT. It is an analysis of the production and treatment of waste sludge from waste water treatment plants in The Czech Republic. The available dataset represents a transportation network at the municipal level of ORPs[1], that pose as individual nodes with recorded amounts of produced and treated sludge, with multiple options for the treatment. The flow along edges of this graph is then reported two times for each edge, from the side of each respective sender and receiver. Because this is a real dataset, it is not safe from errors. The flow amounts are not always equal to each other, as they may seem to be at first glance. Also the total sum of produced sludge is not the same as the sum over all of the treatment options.

For further analysis of this dataset, a mathematical programming model has been devised, aimed at providing a best estimate of the real flow and treatment values. Furthermore, the resulting estimate contains an information about a distribution of the sludge from each entry point in the network, an information that is not present in the original dataset. This proposed model is then solved for the dataset by the ADMM algorithm implementation described in Chapter 3.

Similar model has been already used by the aforementioned research team for the analysis of a dataset covering the energy recovery of a bulky waste in The Czech Republic (for more info about the paper see [15]). The objective

[1] ORP is an abbreviation of a term for a municipal unit in The Czech Republic, one level above individual towns. It usually represents a group of villages, individual middle-sized cities or, for large cities, their districts. Because this term does not have an equivalent in English, it will be referenced in the text by its original abbreviation.

function was linearized by the introduction of auxiliary variables and then solved using the GAMS solver. However, the model was solved only on the level of larger regions due to memory limitations. Applying ADMM allows the objective to stay non-linear, thus reducing the number of variables needed and also decreases the number of constraints, as the model inequalities can be encoded via the use of indicator functions in the objective. Also if the model needs to be applied to a large dataset, that requires the memory to be split across multiple devices, the block splitting form (2.14) of ADMM can be employed. The addition of the local network (see section 4.2.1) is also an improvement on the previous model.

4.2 Model description

First lets introduce a few sets needed for description of the model. The 277 ORPs, i.e the nodes of the network, form the set I. Variables connected to these are usually denoted with an index i, but when it is necessary, an index o is used. This facilitates the distinction between a waste origin node o and a local node i. The 1070 reported connections, i.e. the edges, create set J and, for brevity, the symbol $J(i)$ represents a set containing only the cycle (i, i) for node i. Set L then represents all of the available treatment options, i.e.: material recovery, energy recovery, export, alteration and option called "others". As the recorded flows contain two values for each edge, these are divided into two scenarios, marked with a $-$ index for the outflow values and a $+$ index for the inflow.

The following is a list of symbols and equations forming the model definition. All of the variables are assumed to be from the set of real numbers $\mathbb{R}$.

Input data

$x_j^{\pm}$ recorded flow on the edge j for given scenario (+ or -)

$A_{i,j}^{\pm}$ incidence matrix for given scenario (+ or -)

p_i recorded production in the node i

$t_{i,l}$ recorded amount of sludge treated in the node i by the method l

d_j length of the edge j

Parameters

α_o weight of each producer (used for stability analysis)

β weight between the two objective functions

W weight of penalization

$\delta_{i,o}$ index equality indicator (1 for $i = o$, 0 otherwise)

w_j weight of the edge j (in interval from 0 to 1)

a threshold for the zero penalization

z_1^*, z_2^* optima of the two objective functions

Variables

τ_i error in the recorded production

$\epsilon_j^{\pm}$ error in the recorded flow for given scenario (+ or -)

$x_{j,o}$ amount of flow originally from the node o going through the edge j

$t_{i,o}^{dir}$ amount of sludge treated directly in the node i

$t_{i,o}^{cyc}$ amount of sludge treated after local transportation at the node i

$x_{i,o}^{dir}$ amount of flow going directly through the node i

$t_{i,o,l}^{O}$ amount of sludge treated in the node i by the method l, originating from the node o

Objective functions

$$z_1 : \sum_{j \in J} w_j |\epsilon_j^+ + \epsilon_j^-| + W \sum_{i \in I} |\tau_i - a p_i| \tag{4.1}$$

$$z_2 : \sum_{j \in J} \sum_{o \in I} d_j \alpha_o x_{j,o} + W \sum_{i \in I} |\tau_i - a p_i| \tag{4.2}$$

$$z_3 : \frac{\beta z_1}{z_1^*} + \frac{(1-\beta) z_2}{z_2^*} \tag{4.3}$$

Constraints

$$\delta_{i,o}(p_i + \tau_i) + \sum_{j \in J} A^+_{i,j} x_{j,o} = \sum_{j \in J} A^-_{i,j} x_{j,o} + t^{cyc}_{i,o} + t^{dir}_{i,o} \qquad \forall i, o \in I \quad (4.4)$$

$$p_o + \tau_o = \sum_{i \in I} (t^{cyc}_{i,o} + t^{dir}_{i,o}) \qquad \forall o \in I \quad (4.5)$$

$$x^-_j + \epsilon^-_j = \sum_{i \in I} \sum_{o \in I} A^-_{i,j} x_{j,o} \qquad \forall j \in J \quad (4.6)$$

$$x^+_j + \epsilon^+_j = \sum_{i \in I} \sum_{o \in I} A^+_{i,j} x_{j,o} \qquad \forall j \in J \quad (4.7)$$

$$\delta_{i,o}(p_i + \tau_i) + \sum_{j \in J \setminus J(i)} A^+_{i,j} x_{j,o} = x^{dir}_{i,o} + t^{dir}_{i,o} + \sum_{j \in J(i)} x_{j,o} \qquad \forall i, o \in I \quad (4.8)$$

$$x^{dir}_{i,o} + \sum_{j \in J(i)} x_{j,o} - t^{cyc}_{i,o} = \sum_{j \in J \setminus J(i)} A^-_{i,j} x_{j,o} \qquad \forall i, o \in I \quad (4.9)$$

$$\sum_{l \in L} t_{i,l} = \sum_{o \in I} (t^{cyc}_{i,o} + t^{dir}_{i,o}) \qquad \forall i \in I \quad (4.10)$$

$$\sum_{o \in I} t^O_{i,o,l} = t_{i,l} \qquad \forall i \in I; \forall l \in L \quad (4.11)$$

$$\sum_{l \in L} t^O_{i,o,l} = t^{cyc}_{i,o} + t^{dir}_{i,o} \qquad \forall i, o \in I \quad (4.12)$$

$$x^-_j + \epsilon^-_j \geq 0 \qquad \forall j \in J \quad (4.13)$$

$$x^+_j + \epsilon^+_j \geq 0 \qquad \forall j \in J \quad (4.14)$$

$$p_i + \tau_i \geq 0 \qquad \forall i \in I \quad (4.15)$$

$$x_{j,o}, t^{dir}_{i,o}, t^{cyc}_{i,o}, t^O_{i,o,l}, x^{dir}_{i,o} \geq 0 \qquad \forall i, o \in I; \forall j \in J; \forall l \in L \quad (4.16)$$

The model has, as mentioned above, two optimum criteria that are described by the equations (4.1) and (4.2). The first criterion z_1 (4.1) tries to minimize the weighted sum of absolute errors from the estimate. The weight parameter is defined $\forall j \in J$ as follows:

$$w_j = \begin{cases} M, & \text{if } x^-_j - x^+_j = 0 \\ \dfrac{x^-_j + x^+_j}{2|x^-_j - x^+_j|}, & \text{otherwise.} \end{cases} \quad (4.17)$$

The second criterion z_2 (4.2) penalizes long-distance transportation by which it tries to simulate economic considerations of actors in the network. These two criteria are then combined together, with weight β, in the objective function z_3 (4.3) of the whole model. Because the values of the criteria are not comparable, these need to be in a normalized form. This is done by solving the model with only one of the criteria as the objective function and then dividing the criteria terms by their respective separate objective values, denoted by z_1^* and z_2^*. Finally, the model can be then solved for the third time, now with the whole objective function.

The first group of constraints (4.4) is a balance equation for each individual node. The total amount of produced and imported sludge must be the same as export and treatment. Constraints (4.5) then represent a balance for each producer, meaning that the sludge being produced in a node gets fully processed. The possible errors in reported production data is represented by the variable τ_i. The treatment data amounts are considered as trustful, meaning that there is no error variable being associated with them. Relationships between the estimated flow and its differences from the known values are described by equalities (4.6) and (4.7), for scenarios $-$ and $+$ respectively.

Equations (4.8), (4.9) and (4.10) describe the local network of each node that arises from the data being aggregated (see 4.2.1 below). The distribution of produced amounts between the known levels of different means of treatment is represented by the variable $t^O_{i,o,l}$ and equalities (4.11) and (4.12). Finally the inequalities (4.13), (4.14), (4.15) and (4.16) are simply enforcing non-negativity for all of the estimated production, treatment and flow amounts.

4.2.1 Local networks and aggregation

It is highly impractical, when working with networks on such a large scale, to model every single node and edge in the system. This means, such as is in this case, that the data is available only in a more coarse aggregated version, where nodes represent whole local networks of individual producers, transshipment facilities and treatment plants. When the flow in this more detailed network passes through multiple of these nodes, that are being aggregated into one singular node, it gets then represented by a cycle, because the origin and target nodes become identical. These arising cycles are, however, ambiguous on their own. In order to make sense of them, the flow passing through each respective aggregated node and a potential cycle needs to be divided among multiple variables. It needs to be differentiated between a scenario where the flow passes through several internal points of the node (flow $x_{(i,i)}$ on the cycle (i, i)) or when it only "bounces" in one place, introducing

new variable x_i^{dir}. Moreover the treatment plants can accept the sludge after a chain of local transportations (t_i^{cyc}) or be serviced directly from outside (t_i^{dir}). These relations then create simplified versions of the local networks in each node of the aggregated version (see Figure 4.1) and are represented in the model by constraints (4.8), (4.9) and (4.10).

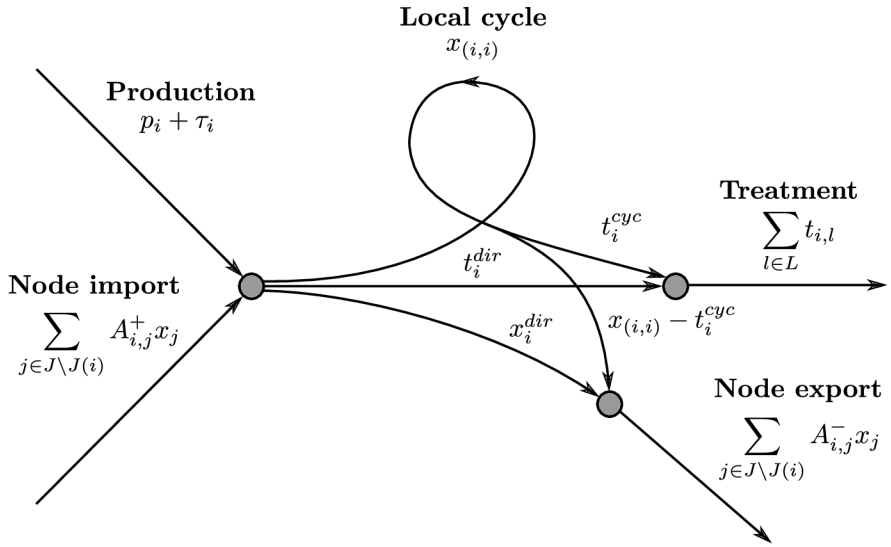


Figure 4.1: Local network diagram for aggregated node i.

4.3 Model transformation for the ADMM algorithm

In order to apply the ADMM algorithm, the model needs to be in the general form

$$\begin{aligned} \text{minimize} \quad & f(\mathbf{x}) \\ \text{s.t.:} \quad & A\mathbf{x} = \mathbf{b} \\ & \mathbf{x} \in \mathbb{R}^n, \end{aligned}$$

where $f : \mathbb{R}^n \longrightarrow \mathbb{R} \cup \{+\infty\}$ is closed proper convex. The model that is being described in 4.2 almost conforms to this form, with the exception of several non-negativity inequalities. However, because the objective function f can take on extended value of $+\infty$, these inequalities can be encoded by the use

of indicator functions. A model of form

$$\begin{aligned} \text{minimize} \quad & f_0(\mathbf{x}) \\ \text{s.t.:} \quad & A\mathbf{x} = \mathbf{b} \\ & \mathbf{x} \geq \mathbf{c} \\ & \mathbf{x} \in \mathbb{R}^n \end{aligned}$$

can be then transformed into

$$\begin{aligned} \text{minimize} \quad & f_0(\mathbf{x}) + I_{\{\mathbf{x} \geq \mathbf{c}\}}(\mathbf{x}) \\ \text{s.t.:} \quad & A\mathbf{x} = \mathbf{b} \\ & \mathbf{x} \in \mathbb{R}^n, \end{aligned}$$

where $I_{\{\mathbf{x} \geq \mathbf{c}\}}(\mathbf{x})$ represents a piecewise sum of indicator functions for each individual inequality

$$I_{\{\mathbf{x} \geq \mathbf{c}\}}(\mathbf{x}) = \sum_{i=1}^{n} I_{\{x_i \geq c_i\}}(x_i).$$

In this form, the transformed model is then ready for the ADMM algorithm:

$$\begin{aligned} \text{minimize} \quad & z + I_{\mathcal{K}} \\ \text{s.t.:} \quad & A\mathbf{x} = \mathbf{b} \\ & \mathbf{x} \in \mathbb{R}^n, \end{aligned}$$

where $I_{\mathcal{K}}$ is the indicator function for all of the inequalities present in the original model

$$\begin{aligned} I_{\mathcal{K}} = I_{\{\epsilon^- \geq -x^-\}} + I_{\{\epsilon^+ \geq -x^+\}} + I_{\{\tau \geq -p\}} + I_{\{x \geq 0\}} \\ + I_{\{t^{dir} \geq 0\}} + I_{\{t^{cyc} \geq 0\}} + I_{\{x^{dir} \geq 0\}}, \end{aligned} \tag{4.18}$$

$\mathbf{x}$ is a vector of all of the model variables, A is the coefficient matrix for all of the equality constraints and $\mathbf{b}$ is a vector of their right-hand-side.

It should be noted, that because the solution for variable t^O is ambiguous in the original model, the variable, together with its constraints (4.11) and (4.12), is being omitted from the transformed model and is later solved with a different heuristic approach in Section 4.5.

4.3.1 Simplifying the proximal operators

The usage of the ADMM algorithm requires to evaluate the proximal operator of the objective function in each iteration. This can be in general a

time consuming operation, however in our case the proximal operator can be greatly simplified. Because all of the three objective functions z_1, z_2, and z_3 can be split under addition into terms containing only a single variable (even after adding the indicator function $I_{\mathcal{K}}$ from previous section), the proximal operator can be evaluated variable-wise (see (1.3)), thus allowing for transformation into simpler scalar formulas.

The split terms from all of the objective functions then appear in these three scalar forms:

$$\varphi_{ind}(x) = I_{\{x \geq a\}} \tag{4.19}$$

$$\varphi_{lin}(x) = cx + I_{\{x \geq a\}} \tag{4.20}$$

$$\varphi_{abs}(x) = c|x - b| + I_{\{x \geq a\}}. \tag{4.21}$$

Deriving a formula for the proximal of (4.19) is straightforward, as the proximal operator of an indicator function is equivalent to the (Euclidian) projection onto the set being indicated, in this case a ray (see (1.6)). Values below the point a get projected onto it, the rest stays the same:

$$\mathbf{prox}_{\rho,\varphi_{ind}}(x_0) = \max\{a; x_0\}. \tag{4.22}$$

Proximal operator for the second function (4.20) can be, applying (1.5), transformed into

$$\mathbf{prox}_{\rho,\varphi_{lin}}(x_0) = \mathbf{prox}_{\rho,\varphi_{ind}}(x_0 - \rho c)$$

and after the substitution of 4.22, we obtain

$$\mathbf{prox}_{\rho,\varphi_{lin}}(x_0) = \max\{a; x_0 - \rho c\}. \tag{4.23}$$

The formula for the third proximal operator for function (4.21) will be derived from the definition:

$$\mathbf{prox}_{\rho,\varphi_{abs}}(x_0) = \underset{x}{\operatorname{argmin}} \left(c|x - b| + I_{\{x \geq a\}}(x) + \frac{1}{2\rho}\|x - x_0\|_2^2\right).$$

First, if x would be $< a$, then the value of the indicator function would be $+\infty$, overpowering the other terms and reducing the operator into a simple projection, similar to (4.22):

$$\mathbf{prox}_{\rho,\varphi_{abs}}(x_0) = \max\{a; \underset{x}{\operatorname{argmin}} \left(c|x - b| + \frac{1}{2\rho}\|x - x_0\|_2^2\right)\}.$$

The general optimality condition of the inner expression is

$$0 \in \nabla(\frac{1}{2\rho}\|x - x_0\|_2^2) + \partial(c|x - b|)$$
$$0 \in x - x_0 + c\rho\, \partial|x - b|.$$

If $x \neq b$, then the subgradient $\partial|x - b| = \operatorname{sgn}(x - b)$ and by substituting $\bar{x} = x - b$ and $\bar{x}_0 = x_0 - b$ we obtain

$$0 = \bar{x} - \bar{x}_0 + c\rho \operatorname{sgn}(\bar{x})$$
$$\bar{x} = \bar{x}_0 - c\rho \operatorname{sgn}(\bar{x}). \tag{4.24}$$

Now

$$\bar{x} < 0 \implies \bar{x}_0 + c\rho < 0$$
$$\bar{x}_0 < -c\rho$$

and

$$\bar{x} > 0 \implies \bar{x}_0 - c\rho < 0$$
$$\bar{x}_0 > c\rho,$$

meaning that $|\bar{x}_0| > c\rho$ and $\operatorname{sgn}(\bar{x}) = \operatorname{sgn}(\bar{x}_0)$. By substituting the latter into (4.24) we obtain

$$\bar{x} = \bar{x}_0 - c\rho \operatorname{sgn}(\bar{x}_0);\ \text{for } |\bar{x}_0| > c\rho. \tag{4.25}$$

In the case of $x = b$ (ie. $\bar{x} = 0$), the subgradient $\partial|x - b|$ is equal to the interval $[-1; 1]$, resulting in

$$0 \in c\rho[-1; 1] + b - x_0$$
$$\bar{x}_0 \in [-c\rho; c\rho] \implies |\bar{x}_0| \leq c\rho,$$

meaning that

$$\bar{x} = 0;\ \text{for } |\bar{x}_0| < c\rho. \tag{4.26}$$

The combination of (4.25) and (4.26)

$$\bar{x} = \begin{cases} 0, & \text{for } |\bar{x}_0| \leq c\rho \\ \bar{x}_0 - c\rho \operatorname{sgn}(\bar{x}_0), & \text{for } |\bar{x}_0| > c\rho \end{cases}$$

can be merged, by using $\operatorname{sgn}(\bar{x}_0) \cdot |\bar{x}_0| = \bar{x}_0$, into

$$\bar{x} = \operatorname{sgn}(\bar{x}_0) \cdot \max\{0; |\bar{x}_0| - c\rho\}$$

and by substituting back for $\bar{x}$ and $\bar{x}_0$ we get the closed form of the optimum

$$x = b + \operatorname{sgn}(x_0 - b) \cdot \max\{0; |x_0 - b| - c\rho\}.$$

The closed form of the whole proximal operator is then

$$\mathbf{prox}_{\rho,\varphi_{abs}}(x_0) = \max\{a; b + \operatorname{sgn}(x_0 - b) \cdot \max\{0; |x_0 - b| - c\rho\}\}. \tag{4.27}$$

4.4 Applying ADMM

4.4.1 Preparation

The implementation of the ADMM algorithm that is being used (described in Chapter 3) requires mainly three things. Linear constraints of the whole model described by a *single* matrix A, accompanied by its RHS vector $\mathbf{b}$, Cholesky factorization of matrix $(I + A \cdot A^\top)$ and proximal operator of the objective function. The following is a description of the process leading to obtaining these.

The dataset for the model contains all the input values ($x_j^{\pm}$, $A_{i,j}^{\pm}$, p_i, $t_{i,l}$, d_j) and the weights w_j are also already known and pre-calculated. These values together are enough for creation of the constraint matrix A. Because the variables of the model are represented by a multi-indexed vectors, they need to be first "unwrapped" and joined together in a single vector. The library `CatViews` allowed to streamline this process greatly. When provided with a list of lenghts of the individual vectors, it allocates and returns a single vector with a combined length of these, together with views referencing individual parts in it and also with their index ranges. The matrix A, together with the vector b is then populated in blocks for each group of constraints and for each variable vector (referenced by the generated index ranges). Because most of the variables appear inside a sum that does not sum all of the indices, the resulting pattern of coefficients in these blocks is usually very sparse. Also some of the blocks are left empty all-together, as not all of the variables appear in all of the constraints. This means that the sparse storage implementation for the matrix is ideal. The resulting matrix has 157181 rows (constraints) by 399844 columns (variables) and a sparsity ratio of about 0.003%. Creating the factorization object then simply requires the use of function `cholesky`, which works on both sparse and dense matrices.

Because of the derived formulas (4.22), (4.23) and (4.27), implementing the proximal operators is quite straightforward. The only problem stems from the "control" parameters α and β, as they appear in the objective functions and also in-turn in their proximal operators. Values of these are not known beforehand as we may want to tweak them in the following computations. Moreover the multi-criterial objective function z_3 needs the optimum values z_1^* and z_2^* of the respective criteria. To overcome this obstacle a *partial application* approach is used. A sort of "creation" function gets defined, which when provided with values of these parameters, in-turn returns another function implementing the proximal operator itself, with the parameters already set. For example, the "creator" function for z_3 is implemented as follows

```
function make_prox_z3(k1,k2,αd)
  k12W = (k1+k2)*W
  k1w = k1 .* w
  k2αd = k2.*αd
  return (ρ,x,x0) -> @views begin
    proxϕabs!(x[vrng[1]], ρ,   -p, ap, k12W, x0[vrng[1]])
    proxϕabs!(x[vrng[2]], ρ, -x_a,  0,  k1w, x0[vrng[2]])
    proxϕabs!(x[vrng[3]], ρ, -x_b,  0,  k1w, x0[vrng[3]])
    proxϕlin!(x[vrng[4]], ρ, 0, k2αd, x0[vrng[4]])
    proxϕind!(x[vrng[5]], 0, x0[vrng[5]])
    proxϕind!(x[vrng[6]], 0, x0[vrng[6]])
    proxϕind!(x[vrng[7]], 0, x0[vrng[7]])
  end
end
```

The array `vrng` used in the snippet simply stores ranges of the individual variable vectors in the concatenated vector `x`.

The returned anonymous inner function is a *closure*, meaning that it captures all of the variables defined in the outer function and also all of the surrounding data variables, which are omitted in the snippet, meaning that they are still available even if they fall out of scope. The individual proximal operators in it are implemented accordingly to their derived formulas.

```
function proxϕabs!(x,ρ,a,b,c,x0)
  @. x = max(a, b + sign(x0 - b) * max(0, abs(x0 - b) - ρ*c))
end

function proxϕlin!(x,ρ,a,c,x0)
  @. x = max(a, x0 - ρ*c)
end

function proxϕind!(x,a,x0)
  @. x = max(a, x0)
end
```

The input parameters `k1` and `k2` respectively are later being defined as

$$k_1 = \frac{\beta}{z_1^*} \cdot 10^8$$
$$k_2 = \frac{1-\beta}{z_2^*} \cdot 10^8,$$

serving as the weights for both criteria. These were originally left unscaled, but because z_1^* and z_2^* reach high values and $\beta \in [0;1]$, the resulting multipliers were too small. This led to numerical floating-point errors, thus requiring a scaling by an appropriate factor. As this just represents multiplying the objective function by a constant, the optimal solution stays unchanged.

4.4.2 Execution

With all of the model data prepared, the model was ready to be solved. First for the two criteria objectives, yielding the values z_1^* and z_2^*. These were then in turn used for the final computation, optimizing the whole objective z_3 with the weight parameter β set at 0.5, representing both criteria equally. The precision parameters were set as $\epsilon_{rel} = 10^-4$ and $\epsilon_{abs} = 10^-8$ and upper bound on iterations was set to 650000. The parameter ρ was found for each objective function by the golden ratio algorithm on model with highly aggregated data (six node graph) in order to minimize the running time. As the relation between ρ and the running time is in general not convex, the local optimization found only a *good estimate*, rather then the optimal value[2]. Table 4.1 then displays execution details for these computations.

	iteration count	exec. time [h]	parameter ρ
1. criterion z_1	650000	75.24	718.79
2. criterion z_2	156486	18.30	97.94
whole model z_3	27431	3.07	1.75

Table 4.1: Execution details for each objective function of the model

There seems to be a big disproportion in the execution time for each objective function. Objective z_3 finishes after the smallest time out of the three, even though it encompasses both of the criteria. Their execution times also differ by a lot, also considering that z_1 reaches the iteration limit and is forced to stop. These discrepancies may be linked to the choice of the parameter ρ and its selection should be a basis for further study.

To inspect the convergence rate of the algorithm, the values of both primal and dual residues for z_3 were recorded as shown in Table 4.2. It is quite clear that most of the improvement was achieved in the beginning of the computation, however the change for higher iterations was still deemed large enough to warrant the use of the chosen precision values. The expressions for calculating the primal and dual residues can be found in [6] and [11].

[2]It should be also noted that there is no guarantee for these ρ values to be performing well on the main model.

iteration	primal residuum	dual residuum
1	12534.3	123353.47
500	299.52	289.32
1000	167.26	126.42
2000	85.77	76.43
4000	50.42	39.01
8000	24.83	12.44
16000	16.53	3.31
27432	9.13	2.58

Table 4.2: ADMM convergence for z_3

4.4.3 Weight of the criteria

The weight β between the two criteria was set ambivalently as 0.5. However, observing its impact on the results may be a good starting point for a further research. In order to measure this impact, the objective z_3 was repeatedly solved with different values of β ranging from 0 to 1 with 0.05 step increments. Values of the individual criteria were calculated from the result for each of these steps and are, together with the value of z_3, displayed in Figure 4.2

For values of β equal to zero, or one, the objective z_3 is equal to the normalized z_2, or z_1 respectively. Because of the normalization, the result should always be equal to one in these points, as we are dividing the same values. As it is clear from the picture, the other objective then becomes grossly unsatisfied, reaching multiples of its optimal value[3].

Because the objective z_3 is always a trade-off between the two criteria, its value is also in the interval between them, as seen in the figure. Now because both normalized z_1 and z_2 are, as mentioned before, attaining on one end large values and are equal to one on the other, they must be equal to each other somewhere in the middle. Now, because z_3 is always between those two values, it must be in this cross equal to them too, as it has nowhere else to go (as is also evident from the figure). This crossing point, at least in this case, does not occur at the value $\beta = 0.5$, hinting at the possibility, that the first objective z_1 is somehow inherently harder to satisfy than the other one.

It should be noted, that as multi-criteria optimization is not in the scope of this thesis, these assumptions presented above are not based on proper mathematical foundations and are presented as-is. They will, however, serve as a base for further research, as real-life optimization applications often incorporate multiple criteria to be satisfied.

[3]The value for z_1 that is being cut off in the figure is almost 5200

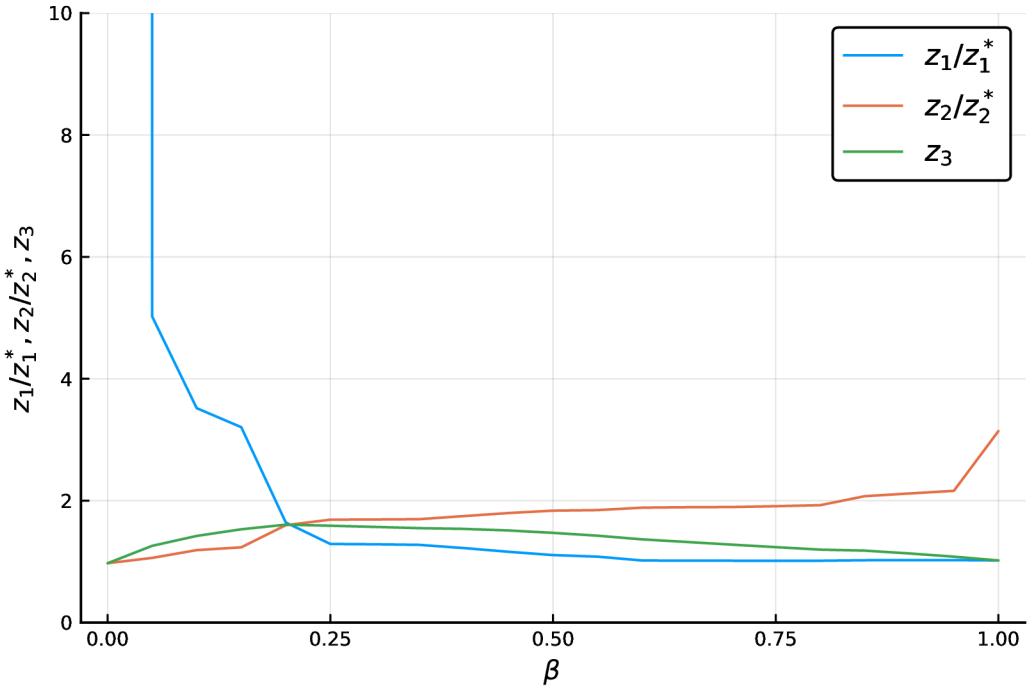


Figure 4.2: Relation between z_3 and normalized z_1 and z_2 for different values of β

4.5 Heuristic for variable t^O

In the previous computations, as mentioned before, the variable $t^O_{i,o,l}$ was not included and omitted. The reason for this is that $t^O_{i,o,l}$ is without an additional information ambiguous in the context of the proposed model. On one hand, the sum $\sum_{o \in I} t^O_{i,o,l}$ is known as it is equal to the dataset value $t_{i,l}$ representing the treatment method l in node i together for all origin nodes o, providing the distinction between different treatment methods. On the other hand, the distinction between different origins is handled by $t^{cyc}_{i,o} + t^{dir}_{i,o}$, which is again equal to $\sum_{l \in L} t^O_{i,o,l}$, however now the information about the different methods is lost. As it is quite clear, connecting these two sides by the variable $t^O_{i,o,l}$ can be done in infinitely many ways, as there is no inherent reason present in the model, why an origin node should be connected to a particular treatment option.

In order to assign some meaningful values to $t^O_{i,o,l}$ a following reasoning was decided upon. The treatment options were ranked by their perceived cost and rareness, as transporting the sludge further is usually connected

with more rare and costly operations, whereas the most common option is usually applied in the vicinity of the waste water treatment plant. This means, that flow originating from the farthest node should be assigned to the highest rated treatment options and only after it is "depleted" then the next farthest node gets assigned to the best options left and so on. This heuristic approach can then be viewed as a continuous knapsack problem (see [16]) with multiple knapsacks being filled in a predefined sequence. As with the original problem, a greedy approach is sufficient to reach the optimal solution. Following is the implementation of this approach in the Julia language.

```
# tL: type-sorted treatment #
L_order = [2, 3, 6, 1, 5, 4]
L_perm = sortperm(L_order)
tL = t[:,L_perm]

# tO: distance-sorted treatment #
tO = tC .+ tD
O_perm = [sortperm(dist[i,:]; rev = true) for i = 1:nI]
O_order = [sortperm(O_perm[i]; rev = true) for i = 1:nI]
for i = 1:nI
        tO[i,:] .= tO[i,O_perm[i]]
end
```

The data parameter $t_{i,l}$ is represented by `tL` and similarly `tO` represents the origin side $t^{cyc}_{i,o}+t^{dir}_{i,o}$. As these variables are not indexed in the "preferred" order, they need to be sorted first. However, because these orderings need to be reversed in the end, a permutation for each of these is created by the use of `sortperm` holding a mapping from the ordered indices to the unordered original. Ordering for treatment options `L_order` is of course the same across the network, whereas the distance-to-origin ordering `O_order` is different for each node i.

```
tOL = zeros(nI, nO, nL); tOr = copy(tO); tLr = copy(tL)
for i = 1:nI, o = 1:nO, l = 1:nL
  o_res = tOr[i,o]
  l_res = tLr[i,l]
  if l_res >= o_res
    tLr[i,l] -= o_res
    tOr[i,o] = 0
    tOL[i,O_perm[i][o],L_perm[l]] = o_res
  else
    tOr[i,o] -= l_res
    tLr[i,l] = 0
    tOL[i,O_perm[i][o],L_perm[l]] = l_res
  end
end
```

The information about the currently "unassigned" amounts is kept in variables `tOr` and `tLr`, which are initiated with the full values of `tO` and `tL` and then subtracted from as needed. The heuristic is run sequentially across all of the nodes, iterating over the origin nodes and treatment options, which are now sorted in the preferred way. The unassigned amounts are compared for each origin-treatment pair (o, l), with the smaller one being set to 0 and fully assigned to the bigger, which is then adjusted by this same amount. This value is also recorded in the variable `tOL` (representing $t^O_{i,o,l}$). As it is being accessed by the use of the previously created permutation vectors `L_perm` and `O_perm` it is already in the original indexing. After the loop returns, all of the amounts had been assigned and the heuristic is done.

As this heuristic is rather simplistic, a more sophisticated approach may be in order. However that would probably require additional information about the system which may serve as a basis for further research of this topic.

Conclusions

At first, we may conclude that all of the goals of this thesis were met. The theoretical part overviews the area of convex optimization together with the more advanced topic of proximal operators, which are gaining in popularity lately. These are then used in order to introduce alternating direction method of multipliers a modern convex optimization method, together with its modification suited for the real-world problem that is also being covered. The possibility of this method to be used in a distributed fashion is also mentioned.

The concrete ADMM formulation is successfully implemented in the Julia programing language, as it is designed for technical and numerical computation and provides suitable tools for their efficient implementation, which are being taken advantage of. This implementation is then successfully tested and used on the large multi-criterial waste management model that is developed together with a research team based at The Institute of Process Engineering, FME, BUT, and based on their previous work (see [15]). The end summary of the computation process reveals potential for further improvement of the implementation, as the largely different execution times of individual criteria may be shortened by more precise adjustments of the step parameter ρ. A quick analysis of the criteria weight β also uncovers interesting results and will lead to a further research, as multi-criterial models are common in applied optimization.

The results will be used by the project Computer Simulations for Effective Low-Emission Energy funded as project No. CZ.02.1.01/0.0/0.0/16026/0008392 and by the project 470 Sustainable Process Integration Laboratory SPIL, funded as project No. CZ.02.1.01/0.0/0.0/15 003/0000456, both by the Czech Republic Operational Programme Research and Development, Education, Priority 1: Strengthening capacity for quality research.

Bibliography

[1] Boyd, Stephen P., and Lieven Vandenberghe. *Convex optimization*. Cambridge: Cambridge University Press, 2004.

[2] Rockafellar, R. Tyrrell. *Convex Analysis*. Princeton, NJ: Princeton University Press. 1970.

[3] N. Parikh and S. Boyd, "Proximal Algorithms", *Foundations and Trends in Optimization*, vol. 1, no. 3, pp. 123-231, 2014.

[4] M. S. Bazaraa, H. D. Sherali, and C. M. Shetty, *Nonlinear programming: theory and algorithms*, 3rd ed. Hoboken: John Wiley, 2006.

[5] J. Nocedal and S. J. Wright, *Numerical optimization*, 2nd ed. New York: Springer, 2006.

[6] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein, "Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers", *Foundations and Trends in Machine Learning*, vol. 3, no. 1, pp. 1-122.

[7] A. P. Ruszczyński and A. Shapiro, *Stochastic programming*. Boston: Elsevier, 2003.

[8] J. R. Birge and F. Louveaux, *Introduction to stochastic programming*, 2nd ed. New York: Springer, 2011.

[9] P. Kall and S. W. Wallace, *Stochastic programming*. New York: Wiley, 1994.

[10] P. M. Pardalos and M. G. C. Resende, *Handbook of applied optimization*. New York, N.Y.: Oxford University Press, 2001.

[11] N. Parikh and S. Boyd, "Block Splitting for Distributed Optimization", *Mathematical Programming Computation*, vol. 6, no. 1, pp. 77-102, 2014.

[12] The Julia Project, 2020. *The Julia Language Documentation*. Available at: https://docs.julialang.org/en/v1/ [Accessed June 25, 2020].

[13] J. Bezanson, A. Edelman, S. Karpinsky, and V. B. Shah, "Julia: A fresh approach to numerical computing", *SIAM review*, vol. 59, no. 1, pp. 65-98, 2017.

[14] The Julia Project, 2020. Performance Tips. *The Julia Language Documentation*. Available at: https://docs.julialang.org/en/v1/manual/performance-tips/index.html [Accessed June 25, 2020].

[15] R. Šomplák, V. Nevrlý, V. Smejkalová, Z. Šmídová, and M. Pavlas, "Bulky waste for energy recovery: Analysis of spatial distribution", *Energy*, vol. 181, pp. 827-839, 2019.

[16] M. T. Goodrich and R. Tamassia, "The Fractional Knapsack Problem", in *Algorithm Design: Foundations, Analysis, and Internet Examples*, John Wiley & Sons, 2002, pp. 259-260.

Appendix A

Source code and resources

This appendix displays some selected source files from the computational part of this thesis. The whole project is also included in the attached archive file. The folder ‘`data`‘ contains the input data for the model as well as julia binary data files (`*.jld`) for cached intermediate steps or raw results. Folder ‘`results`‘ then contains the results in a tabular `csv` form.

ADMM_LCP_simple.jl

Contains the main ADMM algorithm as described in Chapter 3. The files `ADMM_LCP.jl` and `ADMM_LCP_X.jl` contain a slightly modified version with code for benchmarking and additional information. Lastly, `ADMM_examples.jl` contains code responsible for the two examples in Chapter 3 and `Prox_alg.jl` creates Figure 1.1.

```
@inbounds function ADMM_LCP!(
    x´::AbstractArray{<:Real,1},    # x´: vector of variables
    A::AbstractArray{<:Real,2},     #  A: constraint coefficients
    F,                              #  F: cholesky factorization of (I+AA')
    b::AbstractArray{<:Real,1},     #  b: RHS vector
    max_iter::Integer,              # max_iter: maximum iteration count
    ρ::Real,                        #  ρ: stopping criterion weight
    εabs::Real,                     # εabs: absolute error
    εrel::Real,                     # εrel: relative error
    prox!;                          # prox: proximal operator
    )

    # sizes
    M = length(b)
    N = length(x´)
    @assert (M,N) == size(A)

```

```
    # variables
    x = zeros(N)
    x̃  = zeros(N)

    y  = zeros(M)
    y′ = copy(b)
    ỹ  = zeros(M)

    # projection
    AAt = A*A'
    c = zeros(N)
    d = zeros(M)

    # stopping criterion
    n = M+N
    xr = 1:N
    yr = N+1:n

    z = zeros(M+N)
    z′ = zeros(M+N)
    z̃  = zeros(M+N)
    z_last = zeros(M+N)

    k = 1
    while k <= max_iter
       # primal update
       prox!(ρ, x′, x - x̃ )
       @. y′ = b

       # projection onto constraints
       @. c = x′ + x̃
       @. d = y′ + ỹ
       y .= F\(A*c .+ AAt*d)
       x .= c .+ A'*(d.-y)

       # dual update
       @. x̃ =  c - x
       @. ỹ =  d - y

       # stopping criterion
       z′[xr] .= x′; z′[yr] .= y′
        z[xr] .= x ;  z[yr] .= y
        z̃[xr] .= x̃ ;  z̃[yr] .= ỹ
       εpri  = √n * εabs + εrel * max(norm(z′), norm(z))
       εdual = √n * εabs + εrel * norm(ρ*z̃ )
       res = norm(z′ - z)
       res_dual = norm(-ρ*(z-z_last))

```

```
        if (res <= εpri) && (res_dual <= εdual)
            break
        end

        z_last .= z
        k += 1
      end

      return k, x̃ , ỹ
end

function ADMM_factorization(A::AbstractArray{<:Real,2})
    return cholesky(I + A*A')
end
```

MFlow_ADMM.jl

The main file of the computation. The input data file can be created from the excel sheets using `MFlow_data.jl` and the model itself is handled by `MFlow_model.jl`. The results, together with the results from `MFlow_tOL.jl` are converted to the tabular files by `Kaly_excel.jl`.

```
using LinearAlgebra, SparseArrays
using JLD
include("MFlow_model.jl")
include("ADMM_LCP.jl")

@load "data/kaly.jld"
const data = (setI, setJ, setL, d, p, w, x_a, x_b, t, A_a, A_b, a, W)
const v, (τ, ϵ_a, ϵ_b, x, xD, tC, tD), var_beg, var_end = MFlow_variables(nI,
 ↪  nJ, nL, n0)

@load "data/opt_rho.jld"
const e_abs = 1e-8
const e_rel = 1e-4
const max_iter = 650_000

(make_prox_z1, make_prox_z2, make_prox_z3) =
MFlow_make_prox(data...)

(z1, z2, z3) =
MFlow_objective(data...)

const M, N = size(A)
const F = ADMM_factorization(A)
```

```
23
24  println("=== Z1 ===")
25  prox_z1! = make_prox_z1()
26  fill!(v, 0)
27  @time z1_iter, = ADMM_LCP!(v, A, F, b, max_iter, ρ1, e_abs, e_rel, prox_z1!;
    ↪  echo = true)
28  z1_res = z1(τ, ϵ_a, ϵ_b)
29  @show z1_iter
30  @show z1_res
31
32  const α = Vector{Float64}(undef, n0)
33  fill!(α, 1)
34  αd = vec([α[o]*d[j] for j=1:nJ,  o=1:n0])
35
36  println("=== Z2 ===")
37  fill!(v, 0)
38  prox_z2! = make_prox_z2(αd)
39  @time z2_iter, = ADMM_LCP!(v, A, F, b, max_iter, ρ2, e_abs, e_rel, prox_z2!;
    ↪  echo = true)
40  z2_res = z2(τ, x, α)
41  @show z2_iter
42  @show z2_res
43
44  println("=== Z3 ===")
45  k1 =     β/z1_res * 1e8
46  k2 = (1-β)/z2_res * 1e8
47  prox_z3! = make_prox_z3(k1,k2,αd)
48  fill!(v, 0)
49  @time z3_iter, vd, yd = ADMM_LCP!(v, A, F, b, max_iter, ρ3, e_abs, e_rel,
    ↪  prox_z3!; echo = true)
50  z3_res = z3(τ, ϵ_a, ϵ_b, x, α, β, z1_res, z2_res)
51  @show z3_iter
52  @show z3_res
53
54  @save "data/kaly_res.jld" v τ ϵ_a ϵ_b x xD tC tD
```

MFlow_tOL.jl

Contains the heuristic algorithm from Section 4.5.

```
1  using LinearAlgebra
2  using JLD
3
4  include("MFlow_model.jl")
5  @load "data/kaly.jld"
```

```
@load "data/kaly_res.jld"
@load "data/kaly_dist.jld"

v′, (τ, ϵ_a, ϵ_b, x, xD, tC, tD), var_beg, var_end = MFlow_variables(nI, nJ,
 ↪  nL, nO)
v′ .= v

# tL: type-sorted treatment #
L_order = [2, 3, 6, 1, 5, 4]
L_perm = sortperm(L_order)
tL = t[:,L_perm]

# tO: distance-sorted treatment #
tO = tC .+ tD
O_perm = [sortperm(dist[i,:]; rev = true) for i = 1:nI]
O_order = [sortperm(O_perm[i]; rev = true) for i = 1:nI]
for i = 1:nI
    tO[i,:] .= tO[i,O_perm[i]]
end

# "product" knapsack #
tOL = zeros(nI, nO, nL)
tOr = copy(tO)
tLr = copy(tL)
for i = 1:nI, o = 1:nO, l = 1:nL
    o_res = tOr[i,o]
    l_res = tLr[i,l]

    if l_res >= o_res
        tLr[i,l] -= o_res
        tOr[i,o] = 0
        tOL[i,O_perm[i][o],L_perm[l]] = o_res
    else
        tOr[i,o] -= l_res
        tLr[i,l] = 0
        tOL[i,O_perm[i][o],L_perm[l]] = l_res
    end
end

@save "data/kaly_tOL.jld" tOL
```