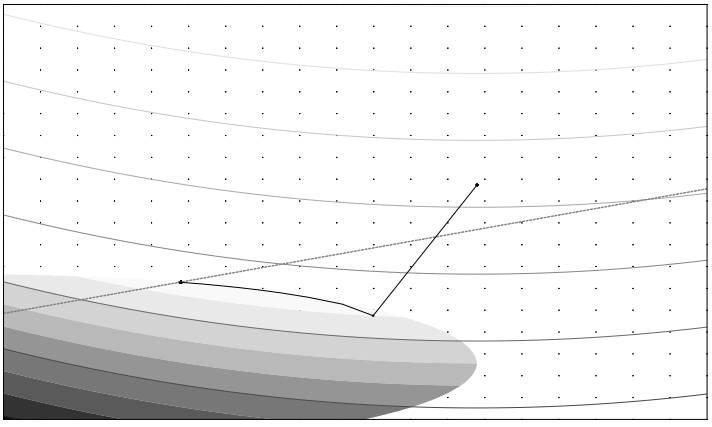
Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Jonas Hjulstad

An Eigen-based C++ Linesearch Filter Barrier Interior-Point Algorithm

Master's thesis in Cybernetics and Robotics
Supervisor: Sebastien Gros
Co-supervisor: Morten Hovd
January 2022



NTNU
Norwegian University of
Science and Technology

Jonas Hjulstad

An Eigen-based C++ Linesearch Filter Barrier Interior-Point Algorithm

Master's thesis in Cybernetics and Robotics
Supervisor: Sebastien Gros
Co-supervisor: Morten Hovd
January 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

NTNU
Norwegian University of
Science and Technology

Summary

This thesis will present an interior-point optimization library based on the original implementation of the Interior Point OPTimizer[1]. The intention with the implementation is to utilize resources, abstractions and libraries found under the modern C++17 standard, and features from the functional programming paradigm to improve both performance, flexibility and readability.

A linesearch filter-barrier method is implemented, with linear algebraic operations computed using the templated C++ library Eigen[2]. The implementation is interfaced with the Constrained and Unconstrained Testing Environment (CUTEst[3]) to access sets of nonlinear optimization problems. Using dense operations, the implementation is able to outperform IPOPT (with BLAS and MUMPS[4]) in computation time, but requires more iterations than IPOPT in order to converge to a solution.

For a given set of optimization problems the dense solver is able to converge for a majority of problems with linear constraints, regardless of nonlinearity in the objective. The solver still fails to converge for nonlinear constraints.

The functional design in the implementation is constrained by the compiled language, but is successfully able to replace many of IPOPT's object oriented design patterns without impairing performance. Some improvements to the functional design will require runtime compilation, which can be pursued in future designs.

Overall, the implemented solver is not robust enough to qualify for general use, and has the potential for improvements on many of the topics presented in the thesis. Still, performance, convergence and design is considered satisfactory with respect to the scope of this thesis.

Sammendrag

Denne masteroppgaven handler om implementasjon av en interior-point optimaliseringsalgoritme basert på den originale implementasjonen av Interior Point OPTimizer[1]. Hensikten med implementasjone er å untnytte ressurser, abstraksjoner og bibliotek utviklet under C++17 standarden, og å trekke inn egenskaper fra det funksjonelle programmeringsparadigme for å forbedre både ytelse, fleksibilitet og lesbarhet.

En linesearch filter-barrier metode er implementert, hvor lineær-algebraiske beregninger er utført med template C++ bibliotek Eigen[2]. Implementasjonen er configurert opp mot Constrained and Unconstrained Testing Environment (CUTEst[3]), som gir tilgang til sett med ulineære problemer. Med beregningsoperasjoner på vanlige matriser klarer implementasjonen å løse problemer med lavere beregnings tid enn IPOPT (med BLAS og MUMPS[4]), men krever flere iterasjoner for å konvergere til en optimal løsning.

For et subset av problemene i CUTEst klarer implementasjonen å konvergere for flertallet av problemene som er begrenset med lineære likhetsligninger, uavhengig av ulineariteten i objektivfunksjonen. Algoritmen har fortsatt problemer med ulineære begrensninger.

Det funksjonelle designet i implementasjonen er begrenset av det kompilerte programmeringsspråket som er brukt, men klarer å erstatte mange av IPOPT's objektorienterte komponenter uten å gjøre algoritmen ineffektiv. Noen forbedringer på det funksjonelle designet vil kreve rekompilering under kjøretid, som kan bli implementert i fremtidige design.

Totalt sett er implementasjonen ikke robst nok til å kvalifisere for generell bruk på ulineære optimaliseringsproblem, og den har potensiale for forbedring på mange av de ulike feltene som har blitt presentert i oppgaven. Fortsatt så kan ytelsen, konvergeringsresultatene og designet anses som tilfredstillende med hensyn til omfanget av oppgaven.

Contents

Figures

Tables

Code Listings

Glossary

ℓ_p-metric A continuum of metrics ranging from the sum of absolute element difference ℓ_1 and the euclidean metric ℓ_2, to the largest element metric ℓ_∞..

ad hoc When necessary or when needed.

atomic operation Indivisible operation that cannot be interfered by concurrent operations.

concurrency Multiple computations occuring at the same time.

condition number Metric for the sensitivity of a linear system.

declarative programming Paradigm that describes a programs logic without describing its control flow.

destructor Method invoked at the end of an objects lifetime..

dual method Method applied to the dual formulation (according to the duality principle) of the optimization problem.

early binding Static linkage of an implementation/resource at compile time.

function overloading Provide implementation for a method that is invoked via *early binding*.

function overriding Provide an implementation for a method that is invoked via *late binding*.

function signature Traits for a function to be distinguishable from others. In C++: The function name, input argument types and ordering of input argument types.

functor A generic type that allows a function to be applied inside it without having any modifications occuring to that type. Immutable C++ objects with callable methods are functors..

header-only library library where an implementation is provided with every definition in the header.

heap Non-contigous, dynamically allocated memory.

hermitian matrix A complex square matrix equal to its own conjugate transpose.

imperative programming Paradigm that modifies a programs state in order to determine its control flow.

Just-In-Time compilation Compilation of code right before execution, used to incorporate runtime information into the compilation process.

lambda capture In C++: Input list for capturing outside data or references to be used in a lambda expression.

lambda closure In C++: Instantiation of a lambda expression, equivalent to an object instance of a class.

late binding Dynamic linkage of an implementation/resource at runtime.

lazy evaluation Avoidance of expression evaluations until they are strictly necessary.

Makefiles Build automation scripts used to compile and link code.

memoization Storage of function evaluation results for repeated usage, in order to avoid repetitious computations.

merit function A scalar-valued function used to relate value to input arguments.

move semantics Programming concepts used to move allocated resources instead of copying them.

name alias In C++: Synonyms for typenames, used to make undetermined/-complicated types readable.

nullspace Linear subspace of a matrix which is mapped to the zero-vector..

Object Composition The combination of simpler objects to form complex ones. An object may *have* another with out *being* one.

orthogonalization Transformation of a set of elements to a set of orthogonal elements that span the same subspace.

overload ambiguity Undistinguishable *function signatures* between multiple functions/methods.

pass-by-reference Data passed to a function without copying. (Passed as an 'implicitly dereferenced' pointer).

pass-by-value A copy of data passed to a function.

pivoting strategies Rearrangemenent of matrix rows/columns to improve the numerical stability of a linear solve.

Polymorphism The trait of having different forms, provision of a single interface to entities of different types.

primal method Method applied to the primal formulation (according to the duality principle) of the optimization problem.

protected data members Data members accessible only to base class and subject derived class..

stack Fast, contigous-allocated memory with implicit allocation/deallocation.

superlinear convergence Convergence that is faster than linear convergence but slower than quadratic..

trivial destructor Destructor that performs no action.

virtual interface Class declaring methods that needs to be implemented by derived classes.

virtual table A table with pointers which determines where interface methods are pointed. Used to dynamically dispatch methods at runtime.

Acronyms

ADT Abstract Data Type.

BEAM Björn's Erlang Abstract Machine.

BLAS Basic Linear Algebra Subprograms.

CCS Compressed Column Storage.

CRTP Curiously Recurring Template Pattern.

CUTE/CUTEr/CUTEst Constrained and Unconstrained Test Environment (with safe threads).

DAE Differential Algebraic Equation.

DFS Depth First Search.

GCC GNU Compiler Collection.

GHC Glaskow Haskell Compiler.

HS Hock-Schittkowski.

IPOPT Interior Point OPTimizer.

JIT Just In Time.

KKT Karush-Kuhn-Tucker.

LAPACK Linear Algebra PACKage.

LICQ Linear Independent Constraint Qualification.

LLVM Low Level Virtual Machine.

LP Linear Program.

MKL Math Kernel Library.

MPS Mathematical Programming System.

MUMPS MUltifrontal Massively Parallel Solver.

NLP Nonlinear Program.

OCP Optimal Control Problem.

ODE Ordinary Differential Equation.

OOD Object-Oriented Design.

PaaS Platform as a Service.

QP Quadratic Program.

SIF Standard Input Format.

SIMD Single Instruction Multiple Data.

SQP Sequential Quadratic Program.

SSE Streaming SIMD Extensions.

Chapter 1

Introduction

1.1 Background

Interior point methods are well-established methods that were initially discovered and used for Linear Programs (LPs) and later convex Quadratic Programs(QPs), and have been used for more than 35 years at this point. These methods have attractive convergence properties within large-scale optimization, due to its consistent, gradual progress towards inequality-constrained problem solutions. Interior point methods have remained competitive against the older simplex methods, and provide a stronger, theoretical worst-case complexity. The open-source COIN-OR Interior Point Optimizer (IPOPT [1]) provide an interface to a wide range of algorithms within the field of interior point methods, a library that is well-established and frequently maintained by developers. The IPOPT-library excels on solving large-scale nonlinear programs (NLPs) with the support of flexible algorithms and a wide range of linear solvers.

1.2 Motivation

It is now over 15 years since the release of the C++ implemented IPOPT 3.0, a time period where many new features and developments of linear algebra libraries have been provided to the language. These additions provide a great oppurtunity to explore and learn the inner workings of the IPOPT solver, and to reimplement some of its components with the support of efficient, modern libraries.

The implementation of IPOPT is based on an *object-oriented* design (OOD), where its dynamic behaviour is constrained to come from internal states of objects that interact with each other. This enables IPOPT to be very flexible within the framework created by its objects, but it also constrains additions to cooperate with the already existing framework.

NLPs span a large set of problems with many different types of dynamics,

and the optimal method to converge to a solution of a NLP can be very different for each one. This encourages adaptive choice of solver methods that are better implemented in a *declarative*, functional design, where the solver is able to branch freely without concern for internal states and dependencies on class interfaces. This thesis aims to develop a framework for interior-point methods which is functionally dynamic, but also computationally efficient.

Furthermore, understanding how the IPOPT-solver works is useful in order properly tune the solver, and get full advantage of the algorithms it provides. It is also a solid first-step in algorithmic design for numerical optimization, which has a much larger field of applications than OCPs.

1.3 Relation to Cybernetics

In the field of cybernetics, numerical optimization is important to find solutions to optimal control problems (OCPs). The dimension size of OCPs increase when the control input resolution to a problem increases, or the time horizon for the control problem is extended. This generally leads to large, sparse NLPs constrained by ordinary differential equations (ODEs) or differential algebraic equations (DAEs), where both control inputs states potentially are inequality-constrained. Determining the activeness of these constraints becomes combinatorially difficult with its dimension, which makes interior-point methods favorable for such problems once they grow large.

1.4 Problem Statement

The scope of this thesis aims to explore the initial implementation of IPOPT, to implement a similar solver using libraries and features provided in modern C++. This can be concretisized into multiple objectives:

- Implement a robust solver that is able to converge on a wide range of differentiable NLPs, regardless of nonlinearity, constraints and dimensions.
- Implement the solver with the foundation of an efficient, linear algebra library with comparable performance to IPOPT.
- Apply functional programming to avoid hidden dependencies, improve readability, enable additional optimization strategies and give the solver better capabilities to dynamically change behaviour.

1.5 Preliminaries

A larger part of the theory presented in this thesis is related to the actual implementation of a solver and the dependencies involved. Prior knowledge with object-oriented C++ programming and templates is recommended, even though many of the related concepts are explained throughout the thesis.

Prior knowledge in numerical optimization is strongly recommended. A lot of the theory in the first sections refer and relates to other optimization theory that is not explained in this thesis. NTNU-course TTK4135 Optimization and Control covers most of these recommendations.

In order to configure and use the code project some knowledge about compilers, path configurations and linking is recommended. Some Fortran subroutines are used to interface the project to a problem testing library.

1.6 Contributions

The original implementation paper for IPOPT [1] and preliminary knowledge from NTNU-courses TTK4135 Optimization and Control and TK8115 Numerical Optimal Control are the main contributing theoretical resources in this thesis. Besides these preliminaries (which are literature-referenced in the thesis) and supervisor guidance, other contributions mainly comes from online resources.

1.7 Software

Selecting appropriate software is very important when designing numerical optimization algorithms. This ranges from the abstractions and efficiency provided by libraries inside a programming language, up to the tools required to organize, efficiently compile and even visualize the code.

1.7.1 Programming Language

The original IPOPT-implementation[1] was written in Fortran but later reimplemented in C++. Both languages are compiled languages which perform all optimization prior to execution. In contrast, interpreted languages (like Python and Ruby) performs code execution without compiling the whole program in one instance. This makes interpreted languages attractive in terms of flexibility, an interpreted language implementation would be able to exploit runtime information to optimize the future decisions of the program. The interior-point algorithm implemented in this thesis depends on linear algebra and operations where a lot of optimization can be resolved before runtime, which is why C++ became the favored language.

1.7.2 Compiler

Clang and GCC has been used as compilers in this project. GCC and Clang offer different optimization features, since Clang intermediately compiles to Low-Level Virtual Machine (LLVM[5]) bytecode.

Name	Version
Clang	10.0.0-4
GCC	9.3.0

1.7.3 CMake

CMake is a build automation tool that enable projects to be built with different compilers system-independently. CMake use configuration files to generate *Makefiles* in a projects subfolders. Makefiles are used by another build automation tool (Make), and consists of rules describing how and in which sequence the project files should be compiled and linked. CMake simplifies the process of building and linking libraries, avoids redundant re-compilation, and configures correct paths for header inclusion and library linking in the project subfolders. Selecting Ninja as CMake generator enables by default multithreaded building, which was found to reduce the overall build time of the project.

Name	Description	Version
CMake	Build tool	3.16.3
GNU Make	Build tool	4.2.1
Ninja	Generator	3.15

Table 1.1: Build automation software used

1.7.4 Eigen

The efficiency of the interior-point algorithm presented in this thesis heavily depends on linear algebra, which makes it important to use subroutines that exploits features of the computer architecture. Basic Linear Algebra Subprograms (BLAS [6]) address this by implementing architecture-specific subroutines. Abstractions on top of BLAS like LAPACK and Matlab provide better application interfaces that remain computationally efficient, but are not able to optimize expressions before evaluating them.

Optimization of expressions can in C++ be implemented by using *templates*, which is used by linear algebra libraries like Eigen[2] and Armadillo [7]. These libraries provide performance benchmark comparisons that depends on different optimization flags and architectures, which makes it difficult to find the overall most suitable library for this application. A fairly recent benchmarking study [8] indicates that Eigen gives the overall better performance in terms of dot products, norms, element access and compilation time, compared to other actively developed libraries. Because of this (and other features discussed in the thesis) Eigen is considered to be the best availiable alternative for this application.

1.7.5 Visualization Tools

Doxygen[9] is the standard documentation tool to use for C++ projects, which provides graph-visualization tools to produce inheritance and call graphs for classes and functions. Doxygen generate `.dot`-format files that is used to generate graph images using Graphviz[10]. Additionally, an online diagram visualization tool called Diagrams.net[11] for cases where it was not possible to produce call and inheritance graphs.

Name	Description	Version
Python	Interpreter	3.9.5
Matplotlib	Plotting library	3.5.1
pybind11	C++ Binders library	2.9.0
Doxygen	Documentation tool	1.8.17

Table 1.2: Visualizaion software

1.7.6 Docker

Docker[12] is a Platform-as-a-service (PaaS) used to run software in containers that are virtualized and made independent of operating system specific dependencies. Docker is used to simplify the installation process of the thesis project by providing a docker Ubuntu image, which contains the software required to compile and execute the code.

1.8 Operating System

The project is configured on Linux operating system Ubuntu 20.04.3 LTS, but can also be compiled on non-Unix systems.

1.9 System Specification

All numerical results have been computed on a Microsoft Surface Pro 7 with an Intel(R) Core(TM) i7-1065G7 CPU 1.30GHz processor, using synchronous 3733 MHz LPDDR4-memory.

1.10 Structure of the Thesis

The theory introduced in this thesis ranges over many different topics.

Most of the theory presented in IPOPTs implementation paper[1] is presented in section 2 and 3. Section 2 presents theory and heuristics for interior-point method and its subproblems, while section 3 presents the filter responsible for rejecting/accepting trial steps.

Section 4 presents pseudocodes for the algorithms in the thesis which are based on the introduced theory.

Section 5 and 6 present some of the different design features used in IPOPT, and the alternatives used in the thesis implementation. These sections present object-oriented and functional design theory in context with C++ and IPOPT, and provides some examples to demonstrate why certain design decisions were made.

Section 8 presents the code implementation. The overall structure of the project is presented, followed by the different components that make up the solver.

Section 9 explains some optimization features related to the linear algebra library and C++ compilers. Section 10 presents dense decomposition-based linear solvers, followed by a brief view of the equivalent sparse solvers.

Section 11 presents the results obtained from running the solver on a test set of NLPs, where illustrations are provided both for the convergence of states, multipliers and objective values. Performance and convergence of the solver is compared to IPOPT.

Section 12 discuss design decisions made based on sections 5 and 6 and compare them to IPOPTs design, along with a review of the results in section 11. The overall state of the solver is discussed, along with potential causes for the occations where the solver failed to converge.

The work in this thesis has revealed many improvements that could be made, and other interesting methods that could be implemented in the same framework. Theory and potential applications that was studied but not used is summarized in section 14.

Chapter 2

Interior Point Optimization

The interior point optimization of nonlinear problems will in this thesis concern problems on the following form:

$$\begin{aligned} \min_{x} \quad & f(x) \\ \text{s.t.} \quad & g(x) \leq 0 \\ & h(x) = 0 \\ & x_{lb} \leq x \leq x_{ub} \\ x \in \mathbb{R}^{N_x}, g(x) \in \mathbb{R}^{N_g}, & h(x) \in \mathbb{R}^{N_h} \end{aligned} \tag{2.1}$$

$f(x)$ is the objective function to be minimized, subject to inequality constraints $g(x)$ and equality constraints $h(x)$. $g(x)$ and $h(x)$ are useful notations for formal definitions, but in the implementation they will be merged in a different formulation:

$$\begin{aligned} \min_{w} \quad & f(x) \\ \text{s.t.} \quad & c_E(x) = h(x) = 0 \\ & c_I(x) - s = \begin{bmatrix} g(x) - s_g \\ x - s_{ub} \\ s_{lb} - x \end{bmatrix} = 0 \\ & s = \begin{bmatrix} s_g \\ s_{ub} \\ s_{lb} \end{bmatrix} \in \mathbb{R}^{N_s} \end{aligned} \tag{2.2}$$

This formulation is not the same as the in the original paper since it allows x to be negative.

2.1 Conditions for Optimality

Some necessary conditions must be fulfilled in order for a point to qualify as a local optimum x^*. These conditions are the necessary Karush-Kuhn-Tucker (KKT) optimality conditions.

2.1.1 Stationarity

$$\nabla f(x^*)+\nabla c_E(x^*)^T\lambda + \nabla(c_I(x^*)+s^*)^T z^* = 0 \tag{2.3}$$

$$z = \begin{bmatrix} z_I \\ z_{lb} \\ z_{ub} \end{bmatrix} \in \mathbb{R}^{N_z}, \quad \lambda \in \mathbb{R}^{N_h} \tag{2.4}$$

KKT-stationarity holds for local extreme values in the lagrangian function. This does not guarantee a local minimum, and may hold for infeasible constraints/multipliers, but it is necessary in order to qualify for it.

2.1.2 Primal Feasibility

$$c_I(x^*) - s^* = 0 \tag{2.5}$$

$$c_E(x^*) = 0 \tag{2.6}$$

$$s^* \geq 0 \tag{2.7}$$

$$\tag{2.8}$$

2.1.3 Dual Feasibility

$$z_i^* \geq 0, \quad i \in [1, \dots N_z] \tag{2.9}$$

The inequality multipliers are strictly positive for active constraints and 0 for inactive. Negative multipliers imply inequality constraint violations.

2.1.4 Complementary Slackness

$$\sum_{i=1}^{N_s} z_i s_i = 0 \tag{2.10}$$

$$\tag{2.11}$$

Complementary slackness ensures that inequality constraint multipliers does not contribute to satisfying the stationary condition unless the corresponding constraint is active (which occurs when $s_{(.)} = 0$).

2.2 Barrier Problem

Distinguishing between active and inactive inequality constraints become increasingly difficult with the number of constraints. For small problems this can be resolved by iteratively checking combinations of constraints (Active-set methods), but this becomes combinatorially difficult with the size of the problem. Larger problems are more efficiently solved using barrier approximations. Parameter μ ensures that the logarithmic barriers (which can be interpreted as equality constraints with μ as a multiplier) are satisfied on the interior feasible side of the inequality constraints.

$$\begin{aligned} \min_{x} \quad & \varphi(x,s) = f(x) - \mu \sum_{i=1}^{N_s} \ln(s_i) \\ \text{s.t.} \quad & c_E(x) = 0 \\ & c_I(x) + s = 0 \end{aligned} \tag{2.12}$$

Introducing the barrier function can be interpreted as a peturbation to the complementary slackness condition:

$$\sum_{i=1}^{N_s} z_i s_i = \mu \tag{2.13}$$

$$\tag{2.14}$$

This peturbation removes the need to 'guess' on wether the inequality constraint is active ($s_i = 0, z_i \geq 0$) or inactive ($s_i > 0, z_i = 0$), but instead reveals the solution gradually by iteratively solving barrier subproblems and reducing μ.

The lagrange equation for the barrier problem is attainable by introducing equality constraint multipliers.

$$\mathcal{L}(x,\lambda,z) = f(x) + c_E(x)\lambda + (c_I(x)+s)z - \mu \sum_{i=1}^{N_s} \ln(s_i) \tag{2.15}$$

The lagrangian can be used to derive Karush-Kuhn-Tucker (KKT) conditions for the nonlinear problem. The stationary condition is satisfied at an extremum of the lagrangian with respect to all variables x, s_g, s_{ub} and s_{lb}.

$$\nabla_x f(x) + \nabla_x c_E(x)^T \lambda + \nabla_x c_E(x)^T z = 0 \tag{2.16}$$

$$c_E(x) = 0 \tag{2.17}$$

$$c_I(x) - s = 0 \tag{2.18}$$

$$z - \mu S^{-1} = 0 \tag{2.19}$$

Where S is a main diagonal matrix with elements of s (This notation generally holds for upper-case letters). By multiplying equation 2.19 with its diagonal

matrix of slack variables, KKT-conditions for a dual-peturbed KKT-system is obtained:

$$zS - \mu = 0 \tag{2.20}$$

The dual conditions often avoid numerical issues by only having to invert the slack-variables once when computing Newton steps. Gradient methods take advantage from avoiding *ill-conditioned* (section 10) KKT-systems with this approach, but it also increases the performance of Newton steps by reducing non-linearity near the local optimum [13, p.586].

2.3 Formulation of the Sequential Quadratic Program

A local approximation of the objective at x is given by the second-order Taylor expansion of $f(x)$.

$$f(x) + \nabla f(x)^T d_x + \frac{1}{2} d_x^T \nabla^2_{xx} f(x) d_x \approx f(x + d_x)$$

By applying linearization to the constraints the resulting subproblem becomes a linearly constrained quadratic program (QP).

$$\begin{aligned} \min_{d_x, d_s} \quad & \nabla_x f(x)^T d_x + \frac{1}{2} d_x^T \nabla^2_{xx} f(x) d_x \\ \text{s.t.} \quad & \\ & \nabla_x c_E(x)^T d_x + c_E(x) = 0 \\ & \nabla_x c_I(x)^T d_x + c_I(x) - d_s - s = 0 \end{aligned} \tag{2.21}$$

Which yields the following lagrangian:

$$\begin{aligned} \mathcal{L}_{SQP}(x, \lambda, z) = d_x^T \nabla_x f(x) + \frac{1}{2} d_x^T \nabla_{xx} f(x) d_x \\ - (d_\lambda + \lambda)^T (\nabla_x c_E(x)^T d_x + c_E(x)) \\ - (d_z + z)^T (\nabla_x c_I(x)^T d_x + c_I(x) - d_s - s) \end{aligned} \tag{2.22}$$

Finding the extremum of $\mathcal{L}_{SQP}$ will yield a local approximate solution to the barrier problem (2.12) when is convex and $c_E(x)$ concave with feasible multipliers. Slack-variable s can be substituted with a chosen definition for multiplier z, which determines if the interior-point method is *primal* or *dual*. Using the dual definition $Z = \mu S^{-1}$ the lagrangian slack-variables d_s, s are reduced to a constant.

$$-(d_z + z)^T (d_s - s) = \mu \tag{2.23}$$

2.4 Newton's Method for KKT-Conditions

Applying Newton's method on the Sequential Quadratic Program (SQP) in equation 2.21 yields the same equation as when applied directly on the primal-dual KKT-conditions (equations 2.19, 2.5 and 2.6). This is consistent with the fact that a full Newton-Rhapson step minimizes quadratic models. Minimizing the SQP-objective leads to the following Newton step (With dual multiplier $Z = \mu S^{-1}$):

$$\begin{bmatrix} \nabla_{xx}\mathcal{L}(x) & \nabla_x c_E(x) & \nabla_x c_I(x) \\ \nabla_x c_E(x)^T & 0 & 0 \\ \nabla_x c_I(x)^T & 0 & -\Sigma^{-1} \end{bmatrix} \begin{bmatrix} d_x \\ -d_\lambda \\ -d_z \end{bmatrix} = - \begin{bmatrix} \nabla_x f(x) - \nabla_x c_E(x)^T \lambda - \nabla_x c_I(x)^T z \\ c_E(x) \\ c_I(x) - \mu Z^{-1} e \end{bmatrix} \tag{2.24}$$

By eliminating the row for $-d_z$ we obtain the following system:

$$\begin{bmatrix} \nabla^2_{xx}\mathcal{L}(x,\lambda,z) + \nabla_x c_I(x)^T \Sigma \nabla_x c_I(x) & \nabla_x c_E(x) \\ \nabla_x c_E(x)^T & 0 \end{bmatrix} \begin{bmatrix} d_x \\ -d_\lambda \end{bmatrix} = - \begin{bmatrix} \nabla_x \varphi(x) - \nabla_x c_E(x)^T \lambda \\ c_E(x) \end{bmatrix} \tag{2.25}$$

Steps in the inequality multipliers can be solved with separate equations, with $\Sigma = S^{-1}z, z = \mu S^{-1}$ this reduces to :

$$d_z = \mu c_I(x)^{-1} - \Sigma \nabla_x c_I(x) d_x - z \tag{2.26}$$

This step is very similar to the one presented in the implementation paper (equation (12)[1]), and only differs in the generalization of constraints.

2.5 Backtracking Linesearch

With d_x, d_λ determined it is possible to evaluate trial steps in this direction. These trial steps will be evaluated by the filter (described in section 3) in order to determine if the step is to be rejected. The series of trial steps used in the original implementation [1, p.30] update the trial step as $\alpha_l = 2^{-l}\alpha^{max}$, where l is the iterate of the trial steps. Backtracking linesearch is well suited for Newton methods since it quickly backtracks to a value where the second-order approximation is good, without making the step size too small.

2.6 Fraction-to-the-boundary Rule

Dual feasibility states that inequality multipliers have to be positive in order to be feasible. One approach to ensure this is to maintain positivity by constraining the step length:

$$\alpha(p_0, p) = \max\left\{\alpha \in (0,1] : p_0 + \alpha p \geq (1-\tau_j)p_0\right\} \tag{2.27}$$

Where τ_j is evaluated for every barrier subproblem:

$$\tau_j = \max(\tau_{min}, 1-\mu_j) \tag{2.28}$$

In contrast to the main implementation paper[1, p.30] the rule is only applied to the inequality multipliers (with separate steps for z_g, z_{ub}, z_{lb}), since x in this case is allowed to be negative. This

2.7 Measurement of Optimality Error

A common way to terminate the search for applications of Newton's method is to stop once some norm $||r(x)||$ is reduced below a tolerance level for target function $r(x)$. The same applies in the search for optimal solutions of barrier problems, but IPOPT additionally use factor scalings to obtain a decent balance of priorities between the KKT-conditions:

$$E_\mu(x,\lambda,z) = \max\left\{\frac{||\nabla_x f(x) + \nabla_x c_E(x)^T\lambda - c_I(x)^T z||_\infty}{s_d}, ||c(x)||_\infty, \frac{||c_E(x)^T Z - \mu e||_\infty}{s_c}\right\} \tag{2.29}$$

Scaling factors s_d and s_c are used to counteract numerical difficulties that may arise in the multiplier values. Close to linearly dependent constraints may result in very large multiplier values, which makes it reasonable to rescale the KKT-conditions with respect to the largest element of the multipliers:

$$s_d = \max\left\{1, \frac{||\lambda||_1 + ||z||_1}{(N_h+N_g)s_{max}}\right\} \qquad s_c = \max\left\{1, \frac{||z||_1}{N_g s_{max}}\right\} \tag{2.30}$$

The search for a local optimal solution will be terminated for the barrier subproblem when the stopping condition is met:

$$E_{\mu_j}(x,\lambda,z) \leq \kappa_\epsilon \mu_j \tag{2.31}$$

For some constant $\kappa_\epsilon > 0$ and barrier subproblem-index $j = [0,\ldots,j_{max}]$.

2.8 Barrier Parameter Update

It is desirable for the sequence of barrier parameters $(\mu_0, \mu_1, \ldots)$ to converge towards 0 in order to statisfy the global stopping criterion $E_0(x,\lambda,z)$. By choosing an appropriate barrier parameter update rule, it becomes possible to prove convergence rates for the resulting trajectory of barrier subproblems. The update rule used in the implementation paper is proven to rise to *superlinear* convergence [14, p.15].

$$\mu_{j+1} = \max\left\{\frac{\epsilon_{tol}}{10}, \min\left\{\kappa_\mu \mu_j, \mu_j^{\theta_\mu}\right\}\right\} \tag{2.32}$$

Where $\kappa_\mu \in (0,1)$ contributes to a more aggressive linear decrease when μ_j is large, and $\theta_\mu \in (1,2)$ decreases superlinearly once sufficiently close to the solution. The update rule is capped with ϵ_{tol} to not reduce μ_j to unreasonably, numerically difficult small values.

2.9 Inertia Correction

In order to ensure that the linear system for the Newton step (equation 2.25) has a unique solution it is in circumstances necessary to modify the Newton step jacobian. Full rank of the hessian and the equality constraint gradient can be enforced by adding diagonal matrices $\delta_w, \delta_c \geq 0$:

$$\begin{bmatrix} \nabla^2_{xx}\mathcal{L}(x,\lambda,z) + \nabla_x c_I(x)^T \Sigma \nabla_x c_I(x) + \delta_w & \nabla_x c_E(x) \\ \nabla_x c_E(x)^T & -\delta_c \end{bmatrix} \tag{2.33}$$

This matrix is invertible if the top-left block of the matrix (referred to as the "Primal-Dual Barrier-term hessian") projected onto the null-space of $\nabla_x c_E(x)$ is positive definite. This is satisfied when the matrix has exactly N_x positive, N_h negative and 0 eigenvalues equal to 0. It is desirable to preserve the original matrix as much as possible, which is done by iteratively increasing and decreasing δ_w, δ_c until the eigenvalue-conditions are satisfied. Inertia correction then gets terminated when the solution to the system is finite.

2.10 Hessian Safeguards

The primal KKT system has important properties to ensure global convergence, while the dual have better local convergence properties. It is possible to maintain parts of both properties by ensuring that the primal-dual hessian ($\Sigma_{PD} = S^{-1}Z$) does not deviate too much from the primal hessian ($\Sigma_P = \mu S^{-2}$). This is done by ensuring that diagonal elements $\sigma^{(i)}_{PD}$ stays within some interval around $\sigma^{(i)}_P$:

$$\sigma^{(i)}_{PD,k+1} \in \left[\frac{\mu_j}{\kappa_\Sigma} s_k^{(i)-2}, \kappa_\Sigma \mu_j s_k^{(i)-2}\right] \tag{2.34}$$

Where $\kappa_{Sigma} \geq 1$ is used to scale the interval. Multiplying by $s^{(i)}_k$ gives the interval for $z^{(i)}_{k+1}$ and consequently the following reset rule:

$$z^{(i)}_{k+1} = \max\left\{\min\left\{z^{(i)}_{k+1}, \frac{\kappa_\Sigma \mu_j}{s^{(i)}_k}\right\}, \frac{\mu_j}{\kappa_\Sigma s^{(i)}_k}\right\} \tag{2.35}$$

2.11 Second-Order Correction

The linearization of $c_E(x)$ may lead to the filter rejecting steps that result in good progress towards a solution. This is called the *maratos effect*[13, p.440],

which can be counteracted by solving the KKT system (equation 2.25) with equality constraints evaluated at the trial step. Second-order corrections use the same gradient evaluation $\nabla_x c_E(x)^T$ as in the original step.

$$\nabla_x c_E(x)^T d_x^{cor} + c(x + \alpha d_x) = 0 \tag{2.36}$$

The corrected direction is obtained by adding together the computed steps:

$$d_x^{cor} = \alpha d_x + d_x^{SOC} \tag{2.37}$$

The same KKT jacobian as in equation 2.25 is used to solve this KKT-system.

$$\begin{bmatrix} \nabla_{xx}^2 \mathcal{L}(x,\lambda,z) + \nabla_x c_I(x)^T \Sigma \nabla_x c_I(x) & \nabla_x c_E(x) \\ \nabla_x c_E(x)^T & 0 \end{bmatrix} \begin{bmatrix} d_x^{cor} \\ -d_\lambda \end{bmatrix} = -\begin{bmatrix} \nabla_x \varphi(x) - \nabla_x c_E(x)^T \lambda \\ \alpha c_E(x) + c_E(x + \alpha d_x) \end{bmatrix} \tag{2.38}$$

In contrast to the paper implementation, the step size in the corrected direction is always a full step in this case, since x itself is not (directly) subject to the fraction-to-the-boundary rule, and there is no reason to restrict the first step for normal Newton-Rhapson steps[13, p.59].

2.12 Initial Multipliers

Initial multipliers for equality constraints are obtained by minimizing the dual infeasibility (equation 2.16), yielding the following linear system:

$$\begin{bmatrix} I & \nabla_x c_E(x) \\ \nabla_x c_E(x)^T & 0 \end{bmatrix} \begin{bmatrix} w \\ \lambda_0 \end{bmatrix} = \begin{bmatrix} \nabla_x f(x_0) - z_{g,0} - z_{ub,0} - z_{lb,0} \\ 0 \end{bmatrix} \tag{2.39}$$

Where w is discarded after the computation.

2.13 Initialization

Primal dual interior point methods puts no restriction on the initial point x_0, λ_0. However, when a solver is initialized on the infeasible side of inequality constraints, the value of $\varphi(x)$ becomes undefined. While it is fully possible to compute the newton direction without $\varphi(x)$, the Fletcher-Leyffer filter[15] presented in the next section lose its ability to accept steps that reduce the objective value $f(x)$. The implementation paper solves this by correcting the initial guess x_0 to the interior, feasible side. While this correction is trivial for bounds x_{lb}, x_{ub}, it is harder to determine for nonlinear inequality constraints. The restoration phases (section 3.3) will address this issue.

Chapter 3

Fletcher-Leyffer Filter

Solving the interior-point barrier subproblem using Newton-Rhapson may result in numerical instability when the initial guess x is far from all local optimal solutions x^*. This results in an unsystematic convergence, where no consistent decrease in the objective value or constraint violation is imposed. One way to overcome this is to introduce merit functions [13, p.435] to penalize the constraint violation. This introduces the challenge of determining the merit multiplier, whose solution is intertwined with the optimal solution x^* itself. A bad choice of merit multipliers will either limit the progress of the Newton-steps (too large), or have a negligible effect (too small).

It is desirable to use full Newton-Rhapson steps as much as possible, as this yields quadratic convergencewhen close to x^*. In order to enable larger steps while maintaining objective reduction the Fletcher and Leyffer-filter [15] instead interprets constrained objectives as a bi-objective.

$$\min_x \quad \varphi_{\mu_j}(x) \tag{3.1}$$

$$\min_x \quad \theta(x) \triangleq ||c_E(x)|| \tag{3.2}$$

This perspective is more general than using a merit function. Sufficient reduction criteria on merit functions are a subset of filter methods where constraints are incorporated into a single objective:

$$\min_x \varphi_\mu(x) - \lambda_M \sum_{i=1}^{m} ||c_{E,i}(x)|| \tag{3.3}$$

3.1 Step Acceptance Conditions

Sufficient reduction in $\varphi_{\mu_j}(w)$ or $\theta(w)$ are set as criteria for accepting a trial point in the implementation of IPOPT.

$$\theta(x_k + \alpha_k d_{x,k}) \leq (1-\gamma_\theta)\theta(x_k) \tag{3.4}$$
$$\varphi_{\mu_j}(x_k + \alpha_k d_{x,k}) \leq \varphi_{\mu_j}(x_k) - \gamma_\varphi \theta(x_k) \tag{3.5}$$
$$\gamma_\theta, \gamma_\varphi \in (0,1)$$

These criteria force the constraint infeasibility to strictly decrease. φ_{μ_j} is allowed to increase in order to satisfy the sufficient reduction in θ. When the value of θ reach a threshold ($\theta \leq \theta_{min}$), no further reduction in θ is prioritized. Instead, a "Switching condition" is introduced, which compares the gradient estimate reduction in φ_{μ_j} to the current constraint infeasibility.

$$\nabla\varphi_{\mu_j}(x_k)^T d_{x,k} < 0, \qquad \alpha_{k,l}[-\nabla\varphi_{\mu_j}(x_k)^T d_{x,k}]^{s_\varphi} > \delta[\theta(x_k)]^{s_\theta} \tag{3.6}$$

With $s_\varphi \geq 1, s_\theta > 1, \delta > 0$. These two conditions ensure that the descent direction is negative, and that the decrease in φ_{μ_j} is significant compared to the current constraint violation. If equation 3.6 holds the trial step has to satisfy the Armijo condition[13, p.33], which is necessary to avoid convergence to non-optimal points.

$$\varphi_{\mu_j}(x_k + \alpha_{k,l} d_{x,k}) \leq \varphi_{\mu_j}(x_k) + \eta_\varphi \alpha_{k,l} \nabla\varphi_{\mu_j}(x_k)^T d_{x,k} \tag{3.7}$$

If either (3.4) or (3.5) is satisfied the trial point will be accepted by the filter, assuming that $\theta \geq \theta_{min}$.

3.2 Filter Update

The filter itself keeps track of upper bounds for accepting (θ, φ), and is initialized with a maximum tolerance for constraints:

$$\mathcal{F}_0 \triangleq \{(\theta,\varphi) \in \mathbb{R}^2 : \theta \geq \theta_{max}\} \tag{3.8}$$

The filter stores value-pairs for the accepted trial points satisfying the bi-objective reduction condition.

$$\bar{\theta}_k \triangleq (1-\gamma_\theta)\theta(x_k) \tag{3.9}$$
$$\bar{\varphi}_k \triangleq \varphi_{\mu_j}(x_k) - \gamma_\theta \theta(x_k) \tag{3.10}$$

Value-pairs updates the filter with a new condition each iteration k when the bi-objective reduction is satisfied.

$$\mathcal{F}_{k+1} = \mathcal{F}_k \cup \left\{(\theta,\varphi) \in \mathbb{R}^2 : \theta \geq \bar{\theta}_k \cap \varphi \geq \bar{\varphi}_k\right\} \tag{3.11}$$

An old value-pair is *dominated* by a new one if both values in the new pair are smaller than in the old pair. At this point the old pair is redundant, and can be removed from the filter. Updating the filter with such pairs ensures that cycles does not occur, newer iterates cannot return to the neighborhood of x_k.

3.3 Feasibility Restoration Phase

When the step size resulting from the filter linesearch becomes too small or the KKT-system becomes very ill-conditioned, other strategies should be considered in order to reach a new solution acceptable to the filter. The implementation papers solution is to intermediately minimize the constraint infeasibility:

$$\begin{aligned} \min_{\bar{x}} \quad & ||c_E(x)||_1 + \frac{\zeta}{2}||D_R(\bar{x} - \bar{x}_R)||_2^2 \\ s.t \quad & c_I(x) \geq 0 \end{aligned} \tag{3.12}$$

Where $D_R(.)$ is used to scale the deviation from the initial reference point $\bar{x}_R$, with diagonal elements $d_R^{(i)}$:

$$d_R^{(i)} = \min\Big(1, \frac{1}{\bar{x}_R^{(i)}}\Big), \quad i = [0, \ldots, N_x - 1] \tag{3.13}$$

In words this optimization problem tries to minimize the distance to equality constraints without leaving the old iterate, while staying on the feasible side of $c_I(x)$.

3.4 Equality Restoration Phase

Coefficients of $c_E(x)$ in equation 3.12can be separated for the positive elements and negative elements ($\bar{p}, \bar{n} \in N_h$)), resulting in a smooth reformulation:

$$\min_{\bar{x}, \bar{p}_E, \bar{n}_E} \quad \rho \sum_{i=0}^{N_h - 1} (\bar{p}_E^{(i)} + \bar{n}_E^{(i)}) + \frac{\zeta}{2}||D_R(\bar{x} - \bar{x}_R)||_2^2 \tag{3.14}$$

$$s.t \quad c_E(x) + \bar{p}_E - \bar{n}_E = 0 \tag{3.15}$$

$$\bar{p}_E, \bar{n}_E \geq 0$$

$$c_I(x) \geq 0 \tag{3.16}$$

Where $\rho > 0$ is used to scale the objective with respect to the constraints, and $\zeta > 0$ is the ℓ_2-regularization parameter. This problem formulation can be solved with the same algorithm as the original barrier problem (equation 2.12).

The feasibility restoration phase can also encounter infeasible iterates. Instead of recursively calling new restoration phases, the infeasible iterate $\bar{x}_t$ is instead fixed in equation 3.14, allowing $\bar{p}$ and $\bar{n}$ to be solved with least squares:

$$\bar{\eta}^{(i)} = \frac{\bar{\mu} - \rho c_E^{(i)}(\bar{x})}{2\rho} + \sqrt{\left(\frac{\bar{\mu} - \rho c_E^{(i)}(\bar{x})}{2\rho}\right)^2 + \frac{\bar{\mu} c_E^{(i)}(\bar{x})}{2\rho}} \tag{3.17}$$

$$\bar{p}^{(i)} = c_E^{(i)}(\bar{x}) + \eta^{(i)} \tag{3.18}$$

3.5 Inequality Restoration Phase

Equations 3.12 to 3.18 summarize how the restoration phase works in the implementation paper. However, the separation of inequality constraints $c_I(x)$ from $c_E(x)$ introduce the possibility of two different kinds of restoration phases. The first restoration phase occurs under conditions where only $||c_E(x)||$ is violated and the barrier subproblem fails. However, if any of the inequality constraints also are infeasible ($c_I(x)^{(i)} < 0$ for one $i \in N_g$), a restoration phase where c_E is replaced by c_I is initialized:

$$\min_{\bar{x}, \bar{p}_I, \bar{n}_I} \quad \rho \sum_{i=0}^{N_h - 1} (\bar{p}_I^{(i)} + \bar{n}_I^{(i)}) + \frac{\zeta}{2} ||D_R(\bar{x} - \bar{x}_R)||_2^2 \tag{3.19}$$

$$s.t \quad g(\bar{x}) + \bar{p}_I - \bar{n}_I - \Delta_{c_I} = 0 \tag{3.20}$$

$$\bar{p}_I, \bar{n}_I \geq 0$$

The aim for this restoration phase is to find a point on the feasible side of the inequality constraints. In order to find a point acceptable to the filter it is useful to add a negative offset to push the next iterate over to a point where φ_{μ_j} is feasible, which is enforced with Δ_{c_I} in equation 3.20.

$$\Delta_{c_I} = \min\left\{\kappa_{c_I} ||g(\bar{x}_0)||_2, \Delta_{c_I,max}\right\} \tag{3.21}$$

Convergence failure invokes a least squares problem equivalent to equations 3.17, 3.18.

Inequality 3.16 is ignored in the thesis implementation, in the hope that sequences of alternating restoration phases are more robust. For example, an equality restoration phase may pull the next iterate to a point that is infeasible with respect to $c_I(x)$, which further activates an inequality restoration phase.

Chapter 4

Algorithms

There are several steps of the IPOPT-algorithm that require iterative procedures to find steps and values that satisfy the given optimality conditions. These algorithms are presented as pseudocodes intended to summmarize their steps, and to be more relatable to the thesis code implementation. These algorithms are not equivalent to the exact steps in the implementation paper [1], but generally use the same parameters. The algorithms are presented bottom-up according to their execution hierarchy in the code. 'Algorithms' will in context with the implementation refer to these pseudocodes and their corresponding subroutines.

4.1 Inertia Correction

Algorithm 1 Inertia Correction Algorithm

$\delta_c \leftarrow 0$
if acceptable eigenvalues(KKT_matrix) **then**
  Obtain d_x, d_λ by solving (2.25)
  $\delta_{w,old} \leftarrow \delta_w$
  Return if d_x, d_λ is feasible
end if
$\delta_w \leftarrow \kappa_w^- \delta_{w,old}$
while $\delta_w < \delta_{w,max}$ **do**
  if any zero_eigenvalues(corrected_KKT_matrix(δ_w, δ_c)) **then** increase δ_c
  end if
  if acceptable eigenvalues(corrected_KKT_matrix(δ_w, δ_c)) **then**
    Obtain d_x, d_λ by solving (2.25) with corrected_KKT_matrix(δ_w, δ_c)
    $\delta_{w,old} \leftarrow \delta_w$
    Return if d_x, d_λ is feasible
  end if
  Increase δ_w
end while
Return correction failure

Storing the previous value of δ_w prevents unneccessary computations for steps where the correction required is consistent. A fraction of $\delta_{w,old}$ is tested at the first iteration in an attempt to reduce the inertia. The following iterations increase δ_w exponentially.

4.2 Second-Order Correction

Algorithm 2 Second-order correction algorithm pseudocode

if $\theta(x+d_x) < \theta(x)$ and (3.6), (3.7) holds for (x, d_x) **then**
  $d_x^{cor} \leftarrow d_x$
  for $i \leftarrow 1$ to p_{max} **do**
    $c_k^{soc} \leftarrow c(x) + c(x + d_x^{cor})$
    Obtain $d_x^{cor}, d_\lambda^{cor}$ by solving (2.38) with c^{soc}
    if $\varphi_{\mu_j}(x + d_x^{cor}) \leq \varphi_{\mu_j}(x) + \eta_\varphi \nabla\varphi_{\mu_j}(x) d_x$ **then**
      $d_x \leftarrow d_x^{cor}$
      $d_\lambda \leftarrow d_\lambda^{cor}$
      Return with accepted step size
    end if
    if $\theta(x) - \theta(x + d_x^{cor}) < \kappa_{soc}$ **then**
      Return to normal linesearch
    end if
  end for
  Return to normal linesearch
end if

This implementation is simplified to only work with full steps $\alpha = 1$. The Armijo condition check differ from the one used in the filter by using the barrier gradient ($\nabla\varphi_{\mu_j}(x)d_x$), while evaluating a the barrier function in a different direction ($\varphi_{\mu_j}(x + d_x^{cor})$).

4.3 Linesearch

Algorithm 3 Linesearch algorithm pseudocode

$\alpha \leftarrow 1$
if $\theta(x_k + \alpha d_x) > \theta(x_k)$ **then**
    if Second-order correction (A.2) succeeds **then**
        Return with accepted step α
    end if
end if
for $i \leftarrow 1$ to l_{max} **do**
    if α satisfies section 3's conditions **then**
        Return with accepted step α
    end if
    if $\alpha < \alpha_{min}$ **then**
        Return failure
    end if
    $\alpha /= 2$
end for
Return failure

4.4 Barrier Subproblem

Algorithm 4 Barrier subproblem algorithm pseudocode

Initialize barrier functor φ_{μ_j} with μ_j
Initialize filter $\mathcal{F}_k$ with provided filter_set
for $k \leftarrow 0$ to max_iter **do**
  if $(E_{\mu_j}(x_k, \lambda_k, z_k) \leq \kappa_\epsilon \mu_j)$ **then**
    Return with x_k, λ_k, z_k
  end if
  Evaluate KKT_matrix and KKT_vector in (2.25)
  if Solving (2.25) for d_x, d_λ with algorithm 1 fails **then**
    Return failure
  end if
  Update position and direction of $\mathcal{F}_k$ with x_k, d_x
  if Linesearch with algorithm 3 fails **then**
    Return Failure
  end if
  Solve d_z with (2.26)
  Limit $\alpha, \alpha_{z_g} \alpha_{z_{ub}}, \alpha_{z_{lb}}$ with Fraction-to-the-boundary rule (2.27)
  $x_{k+1}, \lambda_{k+1}, z_{k+1} \mathrel{+}= \alpha d_x, \alpha d_\lambda, [\alpha_{z_g}, \alpha_{z_{ub}}, \alpha_{z_{lb}}] \cdot d_z$
  Correct z_{k+1} with hessian safeguards (2.34)
end for
Return failure

4.5 Restoration Phase

Algorithm 5 Restoration phase pseudocode

Construct objective $f_R(w,z)$ with x_j (3.14)
Compute $\bar{\eta}_0, \bar{p}_0$ with (3.17, 3.18)
$w_0 \leftarrow [x_0, \bar{\eta}_0, \bar{p}_0]^T$
$\bar{z}_0 \leftarrow [\min(\rho, z_x), \bar{\mu}\bar{\eta}_0^{-1}, \bar{\mu}\bar{p}_0^{-1}]$
$\bar{\lambda}_0 \leftarrow 0$
for $t \leftarrow 0$ to t_{max} **do**
  if Barrier subproblem algorithm 4 fails for f_R **then**
    break
  end if
  if $(E_{\bar{\mu}_t}(w_t, \bar{\lambda}_t, \bar{z}_t) \leq \kappa_\epsilon \bar{\mu}_t)$ **then**
    if $t = 0$ **then**
      break
    end if
    Return with $x_{j+1} = w_t[:N_x]$
  end if
  Update fraction-to-the-boundary parameter $\bar{\tau}_t$ with (2.28)
  Assign $\bar{\mu}_{t+1}$ with (2.32)
end for
Compute $\bar{\eta}_t, \bar{p}_t$ in w_t with (3.17, 3.18)
Return with $x_{j+1} = w_t[:N_x]$

Algorithm 5 is used both for the equality-constrained restoration phase and the inequality-constrained, only differing in the objective function f_R and variables $w, \bar{\lambda}, \bar{z}$.

4.6 Linesearch Filter-Barrier

Algorithm 6 Linesearch Filter-Barrier algorithm pseudocode

Compute λ_0 with (2.39)
$z_0 \leftarrow \mu_0 c_I(x_0)^{-1}$
for $j \leftarrow 0$ to j_{max} **do**
  if $(E_{\mu_j}(x_k, \lambda_k, z_k) \leq \epsilon_{tol})$ **then**
    $x^*, \lambda^*, z^* \leftarrow x_k, \lambda_k, z_k$
    Return success
  end if
  if Barrier subproblem algorithm 6 fails **then**
    if Any inequality constraints are violated **then**
      Get x_{j+1} from inequality restoration phase with x_j, z_j
    else
      Get x_{j+1} from equality restoration phase with x_j, z_j
    end if
    Compute λ_{j+1} with (2.39)
    $z_{j+1} \leftarrow \mu_j c_I(x_{j+1})^{-1}$
  end if
  Update fraction-to-the-boundary parameter τ_j with (2.28)
  Assign μ_{j+1} with (2.32)
end for
Return failure

Inequality restoration phases are prioritized over equality restoration phases, since filter-feasible iterates are prioritized over equality constraint violation.

Chapter 5

Object-Oriented Design

The C++ programming language is well established for object-oriented design, and its advantages are visible in the implementation of IPOPT. Here, objects are used to hide the implementation of data structures as Abstract Data Types (ADTs), which form higher abstraction hierarchies that holds the computed quantities that are required to run IPOPT algorithms. OOD lets objects have mutable internal data, which enables the control flow of the program to be described *imperatively* by the change of these internal states. The topics discussed in the following sections cover broadly used design patterns (which are relevant for IPOPT's implementation) and different types of polymorphism used in OOD.

5.1 Design Patterns

The IPOPT-algorithms are composed of different types of objects that interact with each other. These objects follow different well-known OOD *design patterns* which Larman and Gamma [16] defines as:

'...descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context' [16, p.3]

Larman and Gamma divides design patterns in three categories: Creational, structural and behavioral patterns. The characteristics of the patterns used by IPOPT will be briefly summarized.

5.1.1 Behavioral Patterns

Behavioral patterns are intended provide clear distinctions between the different behaviors of an algorithm by separating them using classes/objects.

The *strategy* pattern[16, p. 315] is useful for implementing algorithms that only differ in behavior, but share other traits (like interface-access and data).

This is widely used through the base class AlgorithmStrategy in IPOPT, which provides common access to the computed data and logging methods it provides.

The *observer* pattern[16, p.293] is useful for introducing dependencies where one object needs to be notified of a change of state in the other. This is useful for caching iteration values, and IPOPT use this pattern to cache the performed linear algebra operations by making all vectors and matrices inherit from an `IpObserver` base class.

5.1.2 Creational Patterns

Creational patterns abstract the instantiaion of objects, which can simplify and make the instantiating code invariant to the details encapsulated in that object. The creational patterns give a lot of flexibility in how, when and which type that is created.

In order to construct its algorithm IPOPT makes use of the *builder* pattern[16, p.97]. This pattern separates the construction of its algorithm from the usage, and makes it possible to configure different strategies for the steps of the algorithm. Table 5.1 summarize the components used by the builder, all of which are classes inheriting from a same base class called `AlgorithmStrategy`.

`SearchDirectionCalculator`	Computes the search direction.
`LineSearch`	Implements computation of step length α and initialization of fallback-mechanism.
`MuUpdate`	Updates barrier parameter μ.
`ConvergenceChecker`	Implements conditions for optimality, responsible for termination of the algorithm.
`IterateInitializer`	Allocates, computes and initializes necessary data and option-configuration structures. Computes least-square primal and dual multiplier initial values.
`IterationOutput`	Implements output summary for each iteration.
`HessianUpdater`	Implements method for computing the hessian.
`EqMultiplierCalculator`	Implements the computation of initial equality multiplier estimates.

Table 5.1: Relevant AlgorithmStrategy-Interfaces used by the IPOPT-builder

This pattern clearly separates the different stages of the IPOPT-algorithm and provides the flexibility to change each step without affecting the others. It allows each component to operate as a black box, provided that the methods specified in its interface has been implemented correctly.

Another pattern used is *factory methods*[16, p.107], which use convenient methods for constructing objects of an unknown type. Three factory methods are used by the IPOPT-builder, which all are used to configure a solver for the primal-dual system. The pattern lets the subclasses be responsible for their own

instantiation. Considering that IPOPT supports a wide range of linear solvers (Harwell Subroutine Library [17] and SPRAL[18], PARADISO[19–21] and more) with different interfaces, it is useful to provide abstractions for this at a low level in the hierarchy.

5.1.3 Structural Patterns

Structural patterns are concerned with how structures are formed from components, they are used to introduce abstracting and adapting interfaces that clarifies and enables communication between objects.

The *adapter* pattern[16, p.139] is used to modify an object to fit the type criterion of others. IPOPT use a TNLP adapter (Triplet NonLinear Program) to allow `TNLP` objects to inherit from the `IpNLP`-class.

5.2 Polymorphism

Polymorphism is the trait of having different forms, which is a characteristic that applies to many areas in object-oriented design. With a set of data members and methods given in a *base class*, the base class can be considered polymorphic with respect to the *derived classes* that inherits from it. This type of polymorphism is very useful for exposing the relevant properties of an object when they are needed. Another approach to achieve polymorphic behavior is *object composition*, where complex behavior is built by adding simpler objects as data members inside a class. Object composition results in better encapsulation than inheritance, and is used when an object *has* sub-objects, in contrast to inheritance where a class *is* a base class.

5.2.1 Dynamic Polymorphism

Dynamic polymorphism resolves method and data types at *runtime*. One example is virtual interfaces in C++, which are base classes that declares methods without implementing them. The inheriting classes need to provide implementations for these methods through method *overriding*.

Code listing 5.1: Inheritance from a virtual interface in C++

```
class A
{
public:
    virtual void method() = 0;
};

class B : public A
{
public:
    void method()
    {
        cout << "Implementation_B" << endl;
```

```
    }
};

class C : public A
{
public:
    void method()
    {
        cout << "Implementation_C" << endl;
    }
};
```

B and C in listing 5.1 override `::method()` in A. The override is caused by *late binding*, which is enforced by the `virtual`-keyword. This implies that a *virtual table* is created in base class A, where pointers to all method overrides are stored at runtime. These pointers provide additional flexibility at runtime, but the same time restricts optimization capabilities. IPOPT makes heavy use of virtual interfaces to present the methods required in its objects.

5.2.2 Static Polymorphism

Static polymorphism resolves method and data types at *compile time*. By replacing the virtual method in listing 5.1 with an actual implementation, *function overloading* (section 6.5.1) will instead be performed with B and C's methods. No virtual table is generated, the pointers to the derived methods are fixed at compile time. These methods cannot be replaced at runtime, which allows additional optimization to be performed by the compiler.

5.2.3 Parametric Polymorphism

Parametric polymorphism is a useful trait for defining objects that only differ in the the types used. True parametric polymorphism allows objects with the same behavior to be recognised as the same type, regardless of the data types contained within it. C++ does not exhibit true parametric polymorphism, but class templates achieve a very similar behavior.

Code listing 5.2: Class template example using sparse and dense Eigen-vectors

```
template <typename T>
class A
{
private:
T data_;
public:

    A(const T& data): data_(data){}

    void print()
    {
        cout << data_ << endl;
    }
```

```
};


int main()
{
    Eigen::VectorXd dense_x(2);
    dense_x << 1.0, 0.0;
    A Dense_Obj(dense_x);
    Dense_Obj.print();

    Eigen::SparseVector<double> sparse_x(2);
    A Sparse_Obj(sparse_x);
    Sparse_Obj.print();
}
```

```
$./Parametric_example
1
0
(4,0)
```

Listing 5.2 presents a class template `A<T>` which is used to generate two distinct classes `A<Eigen::VectorXd>, A<Eigen::SparseVector<double>>`. Class templates allow multiple types to be implemented and manipulated using the same code, effectively avoiding repitious code duplication. The compiler will deduce the types used and instantiate the necessary classes.

The usage of templates leads to new challenges for inheritance. A templated base class `A<T>` will encounter issues when trying to deduce pointers for methods to templated derived classes. This issue is solved by passing the type of the derived class to the base class, which is a design pattern known as the Curiously Recurring Template Pattern (CRTP).

```
template <typename Derived, typename T>
class A
{
public:
T data_;

    A(const T& data): data_(data){}

    void print()
    {
        static_cast<Derived*>(this)->print();
    }
};

template <typename T>
class B: public A<B<T>, T>
{
    using Base = A<B<T>, T>;
    using Base::data_;

    public:
    B(const T& data): Base(data){}
    void print()
    {
        cout << data_ << endl;
```

```
    }

};

template <typename T>
class C: public A<C<T>, T>
{
    using Base = A<C<T>, T>;
    using Base::data_;

    public:
    C(const T& data): Base(data){}
    void print()
    {
        cout << data_ << endl;
    }

};
```

In this case the compiler is unable to override or overload functions, so the base class has to be explicitly guided towards calling the corresponding method in the derived class using `static_cast`. CRTP exhitbits static polymorphism, and enables interface-classes with the same optimization capabilities found in function overloading.

Figure 5.1: Inheritance diagrams for CRTP-example

Figure 5.1 shows that B and C does not inherit from the same base class, and therefore does not exhibit true parametric polymorphism.

5.3 Overview of IPOPTs Implementation

Graph visualizations of IPOPT isolates the complexity of its components, which makes it easier to see how the algorithm works in total. The following visualizations are generated from IPOPT's provided C++-example, which default-instantiates an `IpoptApplication`-object. Its constructor further configures a `Journalist` , which is responsible for messaging, printing and logging of results. Default algorithm-parameters are instantiated in `RegisteredOptions`, followed by construction of the core algorithm by invoking the `AlgorithmBuilder`.

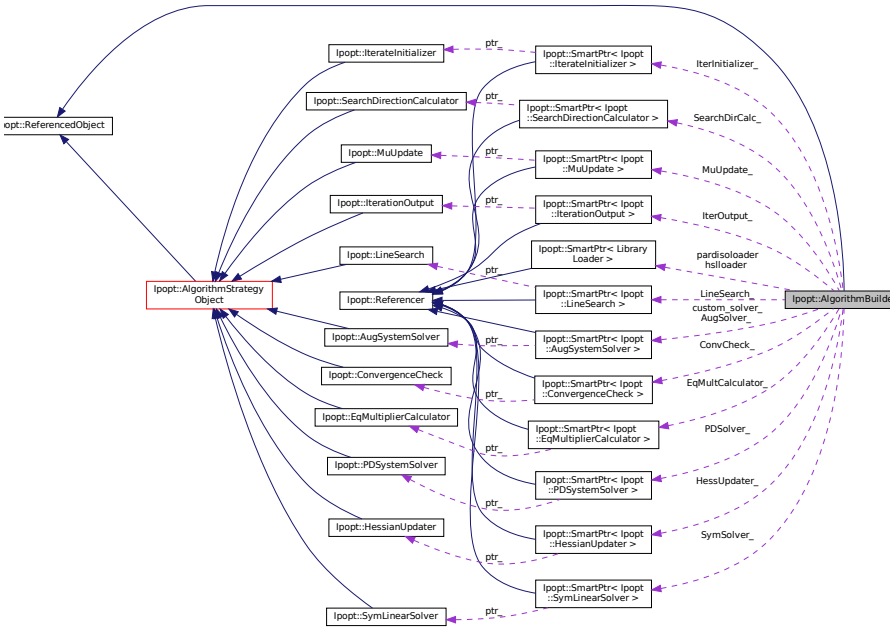


Figure 5.2: `AlgorithmBuilder` collaboration graph

This instantiates a wide range of `AlgorithmStrategy`-objects, which illustrates the flexibility that the library has to offer. Each component may be replaced by a custom implementation with the use of inheritance, which makes simple to swap out any component on demand.

One important consequence of the AlgorithmBuilder's design is how data is exposed to the objects.

The collaboration graph of `AlgorithmStrategy` in Figure 5.3 shows that the class has direct access to the data contained in `IpoptCalculatedQuantities`, which is where all the computed data is cached. `IpoptCalculatedQuantitites` enables read-only access to the data with **`const`** pointers which protects cache from being

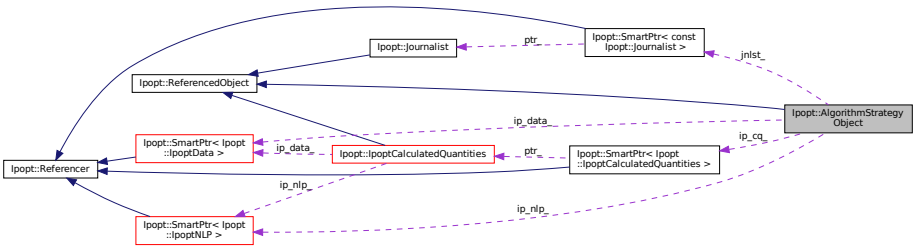


Figure 5.3: `AlgorithmStrategy` collaboration graph

modified. `IpoptCalculatedQuantities` uses a form of *memoization* (section 6.6.1) to compute and retrieve information from the same functions. While this design simplifies the process of accessing data for the objects, it can introduce hidden dependencies between the different stages of the algorithm. Moreover, different types of cached data may be completely independent and irrelevant with respect to eachother, making it unneccessary to The functional design in the next chapter will introduce alternatives which will be used to replace these patterns.

Chapter 6

Functional Design

In contrast to *imperative* programming that use statements to change a programs state, *declarative* programming abstract away the control flow of the application and instead focuses on formulating what the desired outcome is. Functional programming is a declarative paradigm that treats functions as first-class citizens, meaning that general operations used by other entities in the language also is supported for functions (operators, argument passing, modifications, etc...). Instead of using composition of states to create a desired behavior, functional programming compose behavior by building, combining and modifying functions.

6.1 Pure Functions

Functional programming introduce natural encapsulation by using pure functions. Its properties are:

- Function return values are identical for identical input arguments.
- The function application has no side effects, no mutation of data/instructions occurs outside its local environment.

Functional programming with pure functions is a very restrictive design, but it has many attractive properties.

- Error detection - Encountered errors must have been caused by the local environment of the function, or by its input arguments.
- Parallel execution - Any data-independent pure function may be executed in parallel, since there are no side effects.
- Memoization - Identical return values for identical arguments makes it possible to store and reuse previously computed data.

6.2 Lambda functions

Highly idealized functional programming languages (like Scheme[22] and Haskell[23]) solve expressions in coherence with *lambda calculus*[24]. These languages performs computations in terms of variable bindings and substitutions of expressions, in contrast to the imperative approach of modifying internal states. When computations are performed with functions treated as data it is convenient to allow functions to be declared *anonymously*, without a name identifier.

6.2.1 Lambda Expressions and Functors

There are many situations in the IPOPT algorithm design that can greatly benefit from using anonymous functions. Considering that the majority of evaluations performed in the algorithm is built on combinations of the user-provided NLP, dynamically breaking and building abstractions would improve the flexibility and readability of the algorithm. Listing 6.1 show examples where barrier and linesearch evaluation functions are created with lambda expressions.

Code listing 6.1: Example applications for lambda expressions in IPOPT-algorithm design

```
using Vec_s = Eigen::Matrix<double, Ns, 1>;
using Vec_x = Eigen::Matrix<double, Nx, 1>;
Vec_s s;
objective f;

auto phi = [&f, &S](auto& x, const double& mu)
{return f(x) - mu*s.cwiseLog().cwiseInverse();};

const Vec_x x0;
const Vec_x d_x;
auto f_LineSearch = [&f, &x0, &d_x](const double& alpha){return f(x0 + alpha*d_x)
    ;};
```

Data used to construct the lambda expressions can be passed as *captures* ([]), allowing the expression to be dependent on other variables outside its own scope. Besides this, the input argument types and function body instructions are resolved at compile-time. These expressions allow to some extent functions to be treated as data, but they do not create new instructions at runtime. This makes lambda expressions comparable to functor classes. It can be shown that the disassembly output of a *lambda closure* (lambda-expression instance) is equivalent to a functor defined with the same instructions[25]. This also holds for captures, which makes the following functor equivalent to the first example of listing 6.1.

Code listing 6.2: Functor-equivalent to listing 6.1's phi-expression

```
struct phi
{
    phi(objective &f, Vec_s &s) : f_(f), s_(s) {}
    auto operator()(auto &x, const double &mu)
    {
```

```
        return f_(x) - mu * s_.cwiseLog().cwiseInverse();
    }

private:
    objective &f_;
    Vec_s &s_;
};
```

The caveat with lambda-expressions is that C++ does not allow the use of pointers to closures that capture data, and capturing data is necessary to be able to use user-defined NLP-methods. This means that new lambda expressions would have to be defined in every function scope, or that it would have its lifetime maintained in a custom allocated scope, which is the purpose of the function-wrapper `std::function` from the C++ standard library. However, `std::function` makes use of *heap allocation* in order to store capture data once its size surpasses $\sim$ 16 bytes. Heap-allocated function-wrappers are slower than stack-allocated functors, thus functors became the design choice for the implementation.

6.2.2 Currying

A function for adding an input to a constant r may be expressed as follows:

$$(\lambda y.\lambda x.x + y)r \xrightarrow{\beta} \lambda x.x + r \tag{6.1}$$

Here $\lambda x, \lambda y$ denotes function input arguments for two anonymous functions, which is used to represent a sequence of functions (ie. curried representation) on the left hand side. An anonymous version of a function $f(x) = x + r$ is β-*substituted* from a more general sum-function with r as the binding variable. This is how computations are performed in lambda calculus (along with α-renaming and η-extensionality). Currying and input transformations are useful for abstracting away input arguments that are to be used repeatedly.

Currying results in a new function, which in C++ is only achieveable during compile time. Lambda expressions can be used to present an illusion of currying, but the same number of input arguments are evaluated for the underlying declared functions.

6.2.3 Higher-Order Functions

Higher-order functions is a built-in advantage in lambda calculus which makes it possible to create complex behavior by passing simpler functions as arguments to others.

$$(\lambda y.\lambda x.x + y)\lambda z.z^2 \xrightarrow{\beta} \lambda z.\lambda x.x + z^2 \tag{6.2}$$

Passing functions as input arguments to other functions is a way of building complex behaviour with simpler components.

In C++ it is possible to pass function pointers as input arguments, but it is only possible to compose new functions at compile time.

6.3 Recursion

For and while-loops are not achievable in a pure functional design. Loops are allowed to mutate data outside of its iteration scope, which can lead as a side effects for the upcoming iterations. By making a pure function call itself with different input arguments it is still possible to perform iterative procedures. Compared to for and while-loops, recursion introduce a property that both can be considered both more restrictive and enabling. A recursive iteration only need to account for its local variables, which could be a better basis for dynamically branching algorithms.

Local optimization solvers tend to use information obtained around the current iterate, which makes access to previous iteration data redundant from the perspective of the algorithm. This also applies to the implementation in this thesis, which makes it worthwhile to consider the advantages and disadvantages recursion creates.

6.3.1 C++ Function Call Overhead

C++ allocates local variables by pushing them onto the *stack*, where new copies are added (pass-by-value) every time a new function is called. This makes it possible to perform operations that do not concern the local variables, and later return and use them in other operations. In recursive calls this can lead to high stack memory consumption, especially if input arguments are copied for each recursive call.

Code listing 6.3: Stack overflow with pass-by-value recursive function

```
double recursive(std::array<double, 10000> arg, int& i)
{
    std::cout << i << std::endl;
    i++;
    return recursive(arg, i);
}


int main()
{
    std::array<double, 10000> arg;
    int i = 0;
    recursive(arg, i);
}
```

```
...
99
100
101
103
```

```
Segmentation fault (core dumped)
```

Listing 6.3 demonstrates the limitations of the stack, where an array of 80000 bytes are passed by value. With a default stack size of 8192MB a stack overflow occurs after an infeasibly small number of iterations. The number of iterations drastically increase (259226) when the array instead is passed by reference. In this case the stack size is incremented with the pointer size (8 bytes) of arrays. Even in this case, the maximum number of function calls will not be sufficient to call all subroutines in a linesearch filter barrier-algorithm without using return values.

6.3.2 Y-Combinator

In lambda calculus recursion is achieved with Y-combinators:

$$(\lambda x.xx)(\lambda x.xx) \xrightarrow{\beta} (\lambda x.xx)(\lambda x.xx) \tag{6.3}$$

Performing a β-reduction on a Y-combinator results in a new Y-combinator. This is a practically infeasible problem to reduce, since no bounded number of β-reductions is able to bind all variables of the expression. The Y-combinator can be modified with conditional termination conditions, which would make recursive sequence of functions finite. Reductions performed on a finite recursive expression is comparable to the expression-reductions that Eigen would perform on recursive calls in C++.

6.3.3 Tail Recursion

Tail recursive functions are functions where a recursive call is the last thing executed in the function body. This type of recursion enables tail-call elimination, which effectively removes the recursive function call overhead.

Code listing 6.4: Tail recursive function C++ example

```
int recursive(int N, int total)
{
    if (N == 0)
    {
        return total;
    }
    else
    {
        return recursive(N - 1, total + 1);
    }
}
```

```
recursive(N, 0)
recursive(N - 1, 1)
recursive(N - 2, 2)
...
```

The tail-call eliminated representation allows the function calls to be computed sequentially without the need for additional stack-allocation of new variables.

Tail-call optimization can be enforced on the GCC-compiler in the case where objects used are destructible within the scope of the recursive function, and a recursive function call is the last operation performed wihin its scope. Listing 6.5 demonstrates an example application for tail-call optimization.

Code listing 6.5: Tail-call optimization with default destructors

```
struct A
{
    double data;

};
inline A recursive(A a1,
                   A a2,
                   const int N)
{
    a1.data /= a2.data;
    if (N <= 0)
        return a2;
    return recursive(a1, a2, N - 1);
}

int main()
{
    A a1{10.};
    A a2{2.};
    recursive(a1, a2, 10);
}
```

Code listing 6.6: Tail-call optimization assembly output (g++ -O2)

```
        .type   _Z9recursiveR1AS0_RKi.isra.0, @function
_Z9recursiveR1AS0_RKi.isra.0:
.LFB2:
        .cfi_startproc
        movsd   (%rdi), %xmm0
        divsd   (%rsi), %xmm0
        movsd   %xmm0, (%rdi)
        testl   %edx, %edx
        jg      .L2
        movsd   (%rsi), %xmm0
        ret
        .p2align 4,,10
        .p2align 3
.L2:
        divsd   (%rsi), %xmm0
        movsd   %xmm0, (%rdi)
        subl    $1, %edx
        jne     .L2
        movsd   (%rsi), %xmm0
        ret
        .cfi_endproc
```

The assembly output for `recursive` contains no calls to itself, which shows that GCC has transformed the recursive calls to a loop. This example could

even be extended to perform alternating recursive calls (e.g. `recursive_0` calls `recursive_1`, `recursive_1` calls `recursive_0`), Unfortunately this only holds for *trivial* destructors, which is necessary to guarantee that the behavior of the optimized code remains the same. Once a non-trivial destructor is defined for A the compiler is unable to tail-call optimize. This makes this type of optimization incompatible with Eigen, which defines its classes with *protected* destructors. Moreover, the C++ standard gives no guarantee of tail-call optimization, which could make the performance of the code highly dependent on the compiler used.

6.3.4 Encapsulating Loops

Since tail-call optimization is incompatible with Eigen, it is desirable to find other iterative methods that remain efficient, but which also separates *atomic operations*, indivisible operations that cannot be interfered by any concurrent processes.

Many of the steps in the interior-point algorithm is conditionally dependent on values. Some examples are tolerance-dependent termination criteria, the filter conditions and the eigenvalue condition in inertia correction. Replacing these conditions with function branches for all cases would be combinatorially infeasible. A good compromise is to instead separate atmoic operations in smaller functions that are called inside the loop.

This ensures that potential errors can be traced back through the previous sequence of function calls. All of the iterative algorithms presented in section 4 are implemented according to this guideline.

6.4 Design Patterns

Functional programming replace all OOD-patterns with higher-order functions. This can be demonstrated for the strategy-pattern:

Code listing 6.7: Functional equivalent to the strategy design pattern

```
void strategy_0()
{
    std::cout << "Strategy_0" << std::endl;
}

void strategy_1()
{
    std::cout << "Strategy_1" << std::endl;
}

void application(void (*strategy)(void))
{
    strategy();
}
```

Instead of relying on inheritance polymorphism to get different behaviour, the functional approach solves this by passing the behaviour to the function. Corresponding equivalents can be found for other OOD-patterns[26] using function composition.

6.5 Polymorphism

In constrast to the object-oriented approach where objects with different internal types can be treated equally, functional polymorphism allow functions to behave differently depending on the provided types of input arguments.

6.5.1 Function Overloading

Function overloading, also known as *Ad Hoc* polymorphism refers to functions that are able to take arguments of many different types, but exhibit different behavior for those types. One example is a function `add` that return the sum inputs of type `integer`, but returns a concatenated `string` inputs of type `string`. The Eigen-library use function overloading to implement its linear algebra operations, which also applies to operations in-between dense and sparse matrices.

6.5.2 Parametric Polymorphism

Parameter-polymorphic functions are in constrast to overloaded functions invariantly to input types. Templated functions in C++ have a similar behaviour, but generate function overloads for the used parameter types.

Code listing 6.8: Example of parameter polymorphism in functions

```
template <typename T>
void function(T arg)
{
    cout << arg << endl;
}



int main()
{
    function(0);
    function("string");
}
```

```
(gdb)
info functions
...
0x0000000000001254  void function<int>(int)
0x000000000000128c  void function<char const*>(char const*)
```

Listing 6.8 shows that one function is generated for each type passed to the template function.

6.6 Optimization

6.6.1 Memoization

When entities in a program are restricted to have no internal state it becomes harder to share results of expensive computations. However, since a pure function always returns identical values for identical input arguments, it is possible to cache the input-output value pairs for reuse, and return the output value whenever the subject function is called with the corresponding input. Memoization trade off memory for computational speed, which can be a very attractive property for the nonlinear solver.

6.6.2 Lazy Evaluation

Lazy evaluation postpones the evaluation of expressions until a value is needed. This makes it possible to avoid computation of uneccessary, intermediate values, which can be more efficient both in terms of computations and memory usage. The unevaluated expressions are instead simplified by applying β-reductions.

6.7 A Note on Self-Modifying Code

It should be emphasized that β substitutions imply that instructions themselves are modified as data, since no internal state is held by the lambda functions. This is fundamentally different from the way most high-level languages translate code to binaries, the majority (C++ included) do not implement support to mutate instructions at runtime. This also holds for functional programming languages like Scheme (interpreted, compiled or JIT), Erlang/Elixir (BEAM Virtual Machine) and Haskell (GHC Runtime System), which instead use abstractions in form of compilers or interpreters which at some level compiles C or Assembly code. True in-memory modification of instructions during runtime is achievable in Assembly language, but instruction memory modification can behave very differently from one CPU architecture to another (and be very cache-inefficient), making it very difficult to construct higher-level compilers to work this way. Furthermore, considering the difficulties with determining the intention of self-modification, most operating systems enforce "write xor execute" to protect computers from malicious code.

The templated Eigen-expressions are promising in this context, since they can be reduced before runtime, but these reductions are not able to work with information obtained at runtime. A good compromise in this case would be to introduce a *Just-In-Time*-compiler, which allows runtime data to be used to compile new binaries.

6.8 Deduction and Specification of Template Parameters

There are many different ways to pass compile-time configurable information to the different components of a C++ application. The most verbose approach is to explicitly specify template parameters whenever a templated component is used. In this case the compiler can sequentially resolve the required template instantiations, and the application will compile. However when large templated components build on top of each other, the list of template parameters grow larger, and it becomes increasingly difficult to interpret the composition and relations of types in the code. This is counteracted by introducing *name aliases* for the impractically long typenames. Types and name aliases during compile-time are analogous to values and variables at runtime.

Explicitly specifying a template parameter is useful when the parameter defines some behavior in a subject function/class that is not deducible from its input arguments. One relevant example is the choice of a linear solver, which in Eigen can be specified with a template parameter.

Code listing 6.9: Explicit specification of template parameters for a wrapper to an Eigen linear solver

```
template <typename LinSolver, int Nx>
Eigen::Matrix<double, Nx, 1>
Solve(Eigen::Matrix<double, Nx, Nx>& A, Eigen::Matrix<double, Nx, 1>& b)
{
    LinSolver S(A);
    return S.solve(b);
}

constexpr int Nx = 2;

int main()
{
    using Mat = Eigen::Matrix<double, Nx, Nx>;
    using Vec = Eigen::Matrix<double, Nx, 1>;
    Mat A;
    A << 1 , 0 , 0, 1;
    Vec b;
    b << 1, 2;

    Vec result = Solve<Eigen::LDLT<Mat>, Nx>(A, b);

}
```

In the case of listing 6.9 both `Nx` and `LinSolver` have to be specified, even though `Nx` is present in the dimensions of the input arguments. It is not possible (as of C++ 17) to partially deduce function template parameters.

The less verbose, opposite approach is to let the compiler deduce parameters through *implicit type deduction*. Implicit type deduction avoids the need of specifying template parameters by using the input arguments to functions and constructors (C++17) to deduce types. This approach is useful in order to avoid having to repeatedly specify types bottom-up.

Code listing 6.10: Implicit deduction of Eigen input arguments

```
template <int Nx, int Nc>
void fun(Eigen::Matrix<double, Nx, Nx>& Mat_x,
Eigen::Matrix<double, Nc, 1>& Vec_g)
{
    using Mat = Eigen::Matrix<double, Nx, Nx>;
    using Vec = Eigen::Matrix<double, Nc, 1>;
    // ...
}

template <typename Mat, typename Vec>
void fun(Mat& Mat_x,
Vec& Vec_c)
{
    constexpr int Nx = Mat::RowsAtCompileTime;
    constexpr int Nc = Vec::RowsAtCompileTime;
    // ...
}
```

Listing 6.10 presents two useful deduction methods used for retrieving the types and dimensions of Eigen-matrices. Problem dimensions are generally deduced at lower levels in the implementation, while the matrix types themselves are sometimes deduced in higher levels where dimensions are irrelevant.

While functions are constrained to only allow full parameter specification/deduction, classes can be partially specialized and also constructor-deduced (C++17). Templated methods inside class templates are able to combine the advantages of the two approaches, leading to a reformulation of the linear solver function in listing 6.9.

Code listing 6.11: Partial implicit class template deduction

```
template <typename LinSolver>
struct CustomLinSolver
{
    template <int Nx>
    static void Solve(Eigen::Matrix<double, Nx, Nx>& A,
                      Eigen::Matrix<double, Nx, 1>& b)
    {
        LinSolver S(A);
        S.solve(b);
    }
};

constexpr int Nx = 2;
int main()
{
    using Mat = Eigen::Matrix<double, Nx, Nx>;
    using Vec = Eigen::Matrix<double, Nx, 1>;
    Mat A;
    A << 1, 0, 0, 1;
    Vec b;
    b << 1, 2;

    CustomLinSolver<Eigen::LDLT<Mat>>::Solve(A, b);
}
```

Chapter 7

Constrained and Unconstrained Testing Environment

The Constrained and Unconstrained Testing Environment (CUTE [27]/CUTEst [3]) has been widely used to interface numerical solvers to constrained/unconstrained optimization problems. The library is an adaptation of an older library written Fortran, which is fully compatible with C++ code. Solvers that have been configured for CUTEst can be applied to a wide range of problem-files written in the Standard Input Format (SIF). When the files have been decoded (using the sifdecode-library provided with CUTEst) it is possible evaluate objective, constraints, jacobians and the hessian by calling functions from a provided header file (cutest.h).

7.1 Standard Input Format

The Standard Input Format (SIF) is a general input format for specifying nonlinear optimization problems. The input format is based on the other well-established Mathematical Programming System (MPS) which features an input format for linear and linear mixed-integer problems. The SIF-format considers the problem formulation used in the large-scale nonlinear optimization solver LANCELOT [28]. The SIF reference report [29] describes how the components of a nonlinear problem can be formulated using indicator cards. One example of a SIF-file will be included to demonstrate how problems are specified.

7.1.1 2-Dimensional Rosenbrock NLP in SIF-Format

The Rosenbrock function is a commonly used, non-convex optimization problem for testing optimization algorithms. The problem problem is known to converge slowly for gradient-based algorithms. The following formulation is

provided in a SIF-file with CUTE/CUTEst.

$$\min_x f(x) \quad = \quad 100(x_2 - x_1^2)^2 + (1 - x_1)^2 \tag{7.1}$$

The objective contains two groups $(x_2 - x_1^2)$ and $(1 - x_1)$, which are named G1 and G2 in the SIF-file. G2 contains a constant in addition to variable x_1, which is specified under CONSTANT. G1 and G2 are both quadratic groups, but G1 additionally contains a quadratic element. Groups and elements are first declared under GROUP/ELEMENT TYPE, and later specified under the ELEMENTS-section. Element function SQ(V1) and group function L2(GVAR) is declared to account for x_1^2 and the quadratic groups respectively. Function value, gradient and hessian are defined for these functions under ELEMENTS.

The constant element 1.0 is added to G2 under CONSTANTS. Quadratic element x_1^2 is associated with its element type under ELEMENT USES, yielding E1. The objective function is assembled under GROUP USES, where L2 is applied to all groups by setting it as DEFAULT, and $-x_1^2$ is added to G1. Bounds and initial values are trivially specified, resulting in the full SIF-file:

Code listing 7.1: ROSENBR.SIF

```
***************************
* SET UP THE INITIAL DATA *
***************************

NAME          ROSENBR

*   Problem :
*   *********

*   The ever famous 2 variables Rosenbrock "banana valley" problem

*   Source:  problem 1 in
*   J.J. More', B.S. Garbow and K.E. Hillstrom,
*   "Testing Unconstrained Optimization Software",
*   ACM Transactions on Mathematical Software, vol. 7(1), pp. 17-41, 1981.

*   SIF input: Ph. Toint, Dec 1989.

*   classification SUR2-AN-2-0

VARIABLES

    X1
    X2

GROUPS

 N  G1        X2        1.0
 N  G1        'SCALE'   0.01
 N  G2        X1        1.0

CONSTANTS

    ROSENBR   G2        1.0
```

```
BOUNDS

 FR ROSENBR   'DEFAULT'

START POINT

    ROSENBR   X1        -1.2
    ROSENBR   X2         1.0

ELEMENT TYPE

 EV SQ        V1

ELEMENT USES

 T  E1        SQ
 V  E1        V1                    X1

GROUP TYPE

 GV L2        GVAR

GROUP USES

 T  'DEFAULT' L2
 XE G1        E1         -1.0

OBJECT BOUND

 LO ROSENBR              0.0

*   Solution

*LO SOLTN                0.0

ENDATA

***********************
* SET UP THE FUNCTION *
* AND RANGE ROUTINES  *
***********************

ELEMENTS      ROSENBR

INDIVIDUALS

 T  SQ
 F                      V1 * V1
 G  V1                  V1 + V1
 H  V1        V1        2.0

ENDATA

**********************
* SET UP THE GROUPS *
* ROUTINE           *
**********************

GROUPS        ROSENBR
```

```
INDIVIDUALS

 T  L2
 F                      GVAR * GVAR
 G                      GVAR + GVAR
 H                      2.0

ENDATA
```

A simpler introduction is provided in LANCELOT[28, p.15].

7.2 Sifdecoder

The sifdecoder[30] is responsible for translating components specified in the SIF-files into usable fortran-subroutines that evaluates objective function, constraints, their gradients and the objective function hessian.

Code listing 7.2: Decoding and compiling ROSENBR.SIF

```
$sifdecoder ROSENBR.SIF
 Problem name: ROSENBR

 Double precision version will be formed

 The objective function uses 1 linear group
 The objective function uses 1 nonlinear group

 There are 2 free variables


 File successfully decoded
 $ls
 AUTOMAT.d  ELFUN.f  EXTER.f  GROUP.f  OUTSDIF.d  RANGE.f
 $gfortran -c *.f
 $ls
 AUTOMAT.d  ELFUN.f  ELFUN.o  EXTER.f  EXTER.o  GROUP.f  GROUP.o
 OUTSDIF.d  RANGE.f  RANGE.o
```

The sifdecoder produce four fortran-files in Listing 7.2, which is compiled to object-files using the gfortran-compiler. These object files can be compiled together with C++ source files. Function declarations for the subroutines are found a C-header file provided with CUTEst (Cutest.h), which can be included directly by specifying its path to the compiler. The files have distinct purposes:

- ELFUN.f - Evaluates numerical functions for nonlinear element types and its derivatives.
- GROUP.f - Evaluate numerical functions for group types and its derivatives.
- RANGE.f - Transforms elemental to internal variables.
- EXTERN.f - Contains user-provided fortran subroutines.
- OUTSDIF.d - Contains problem structure data, used to initialize the problem.

7.3 Hock-Schittkowski Collection

The Hock-Schittkowski (HS) collection[31] contains a wide range of NLPs which tests how robst a solver is to numerical difficulties. Some of the traits of these problems are the following:

- Badly scaled objectives, variables and constraints.
- Non-smooth model functions.
- Ill-conditioned optimization problems.
- Infinitely/several different optimal local solutions.
- Solutions where constraint qualification is not satisfied.

The HS collection is an ideal application for the dense solver because of the small problem dimensions.

7.3.1 Problem Classification

The CUTEst problem set classify its problems with strigs in the following format:

```
XXXr-XX-n-m
```

The first letter encodes the type of objective function. The HS problem set uses objectives quadratic (Q), nonlinear (O), sum of squares (S) and constant (C). The second letter encodes the type of constraints. The HS problem set uses linear (L), quadratic (Q), nonlinear (O) and bounds only (B). The third letter and 'r' gives the smoothnes and highest derivative order of the problem. The following 'XX' gives miscellaneous info, while 'n' and 'm' gives the problem dimensions.

Chapter 8

Implementation

8.1 Overview

The project excessively use directories to separate the its different components. This structure also works well with CMake, which adds `CMakeLists.txt`-files in each directory that enables executables and libraries with relative paths.

8.1.1 Top-Level Structure

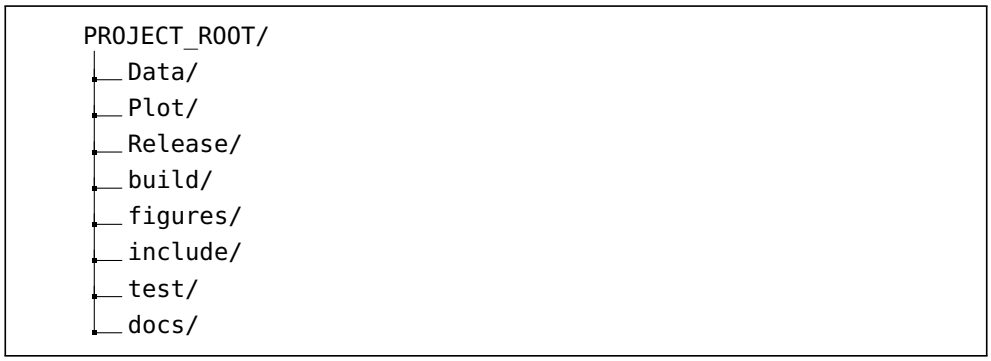


Figure 8.1: Top-level folder project structure

The top-level is used to separate the library from the data, parameters, executables and plotting tools. The core directories can are summarized as follows:

- include - Contains all header-files used by the project. (Excluding cutest.h)
- Data - Contains data produced by executables and parameters. SIF-problems are decoded in its SIF-subdirectory, and matrices/vectors are provided to QP-objectives using.csv-files in its QP-subdirectory.
- Plot - Contains source files for Python-binders and Python-files used to plot results, along with build build/plot automation scripts
- test - Contains source code for all executable examples

- build - Contains the resulting binary outputs from building the project in debug mode (generated by CMake)
- Release - Contains the resulting binary outputs from building the project with optimization flags (generated by CMake)
- docs - Directory for doxygen graphs and latex convergence plot generation.

8.1.2 Header-Library Structure

The include-directory is structured to clearly separate the different components of the library. Almost all header files are given individual subdirectories.

```
include/
├── Common/
│   ├── Barrier_Parameter_Update/
│   ├── Circular_Buffer/
│   ├── Memoizer/
│   ├── Utils/
│   ├── Cutest.hpp
│   └── EigenDataTypes.hpp
├── Dense/
│   ├── Algorithms/
│   │   ├── Barrier_Subproblem/
│   │   ├── Inertia_Correction/
│   │   ├── LSFB/
│   │   ├── LineSearch/
│   │   ├── Restoration_Phase/
│   │   └── SOC/
│   ├── Functors/
│   │   ├── Barrier/
│   │   ├── Filter/
│   │   └── Objective/
│   ├── Hessian_Safeguards/
│   ├── Initial_Multipliers/
│   ├── KKT_System/
│   │   ├── Inequality_Multiplier_Solver.hpp
│   │   └── KKT_System_Dense.hpp
│   ├── Optimality/
│   └── Restoration_Least_Squares/
├── SIF_Dimensions/
│   ├── Dimensions.csv
│   └── Dimensions.hpp
└── Sparse (Deprecated)/
```

Figure 8.2: Include-directory structure

Datatype-invariant functions (Memoizer, Utilities and Common) are implemented in headers near the top-level, along with compile-time parameters for SIF-files (SIF_Dimensions). The core components of the solver are implemented in the Dense and Sparse modules. The different components of the solver follow the same separation as presented in this thesis, with smaller routines isolated into separate headers (Optimality, Initial_Multipliers, KKT_System,...).

8.2 Memoizers

Memoizers are implemented as simple storage classes where templated pairs of data are stored. Memoizers contain an internal apache-licenced circular buf-

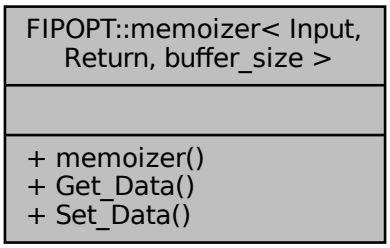


Figure 8.3: The memoizer-class

fer[32], which holds a fixed capacity of element pairs `Input, Return`. New input-output pairs will overwrite the oldest entry in the buffer.

8.3 The NLP Functors

8.3.1 Motivation

In common with object-oriented design it is desirable to abstract away elements that are irrelevant in different parts of an application. Functions can be grouped inside objects similarly to other data (Either by storing pointers inside them, or by specifying them as methods inside its class). This is the purpose of functors, which is very useful for abstracting away user-defined NLPs from the interface of the algorithm.

8.3.2 Objective

A user-defined NLP needs to provide evaluations for the objective and constraints, in addition to their first derivatives and hessians. These functions along with the constraint and state dimensions (N_x, N_g, N_h) are the fundamental properties all objectives should provide.

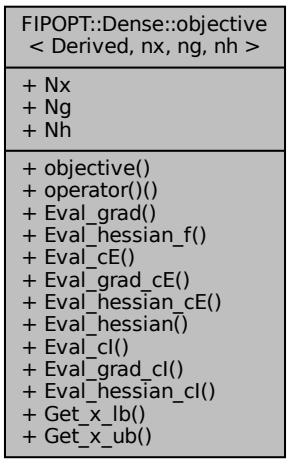


Figure 8.4: The dense base objective class

`operator()`	$f(x)$
`Eval_grad`	$\nabla_x f(x)$
`Eval_hessian_f`	$\nabla^2_{xx} f(x)$

Solver side:		**User side:**	
`Eval_cE`	$c_E(x)$	`Eval_h`	$h(x)$
`Eval_grad_cE`	$\nabla_x c_E(x)$	`Eval_grad_h`	$\nabla_x h(x)$
`Eval_hessian_cE`	`Eval_hessian_h`	`Eval_hessian_h`	$\sum_{i=1}^{N_h} \lambda_i \nabla^2_{xx} h^{(i)}(x)$
`Eval_cI`	$c_I(x)$	`Eval_g`	$g(x)$
`Eval_grad_cI`	$\nabla_x c_I(x)$	`Eval_grad_g`	$\nabla_x g(x)$
`Eval_hessian_cI`	`Eval_hessian_g`	`Eval_hessian_g`	$\sum_{i=1}^{N_g} \lambda_{g,i} \nabla^2_{xx} g^{(i)}(x)$

Table 8.1: Description of `objective`'s methods

Note: λ_g in table 8.1 are equality multipliers unique to the inequality restoration phase, since this is the only algorithm utilizing `Eval_hessian_g`.

The implemented methods pass input arguments as `MatrixBase<T>`, which enables the methods to form Eigen-expressions without performing any evaluation into temporary allocations [33]. The base class accounts for bounds and use the corresponding notation. For example, the base class uses `::Eval_cI(.)` and `::Eval_cE(.)`, while the derived implementation needs to provide the corresponding `::Eval_g(.)` and `::Eval_h(.)`.

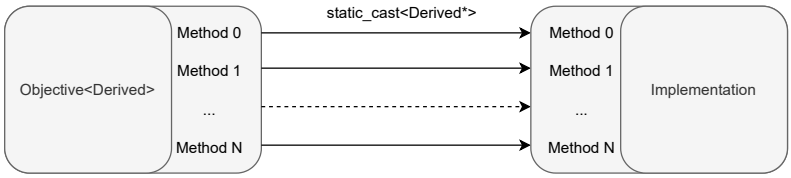


Figure 8.5: Method dispatching for base objectives

8.3.3 Memoized Objective

The inputs and return values of methods can be memoized by inheriting from a memoized class. This is achieved by redirecting the function calls from base objective classes to memoization-wrappers.

8.3.4 Messenger and Observer

A messenger-layer can be used to obtain computed values without interfering with the design of the algorithm. The messenger pass the input/return value

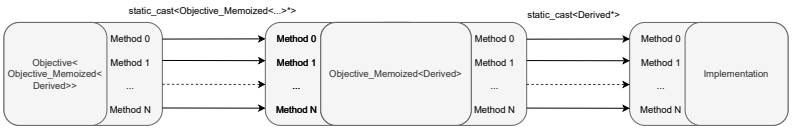


Figure 8.6: Method dispatching for memoized objectives

pairs for each method to a corresponding one in an observer-object.

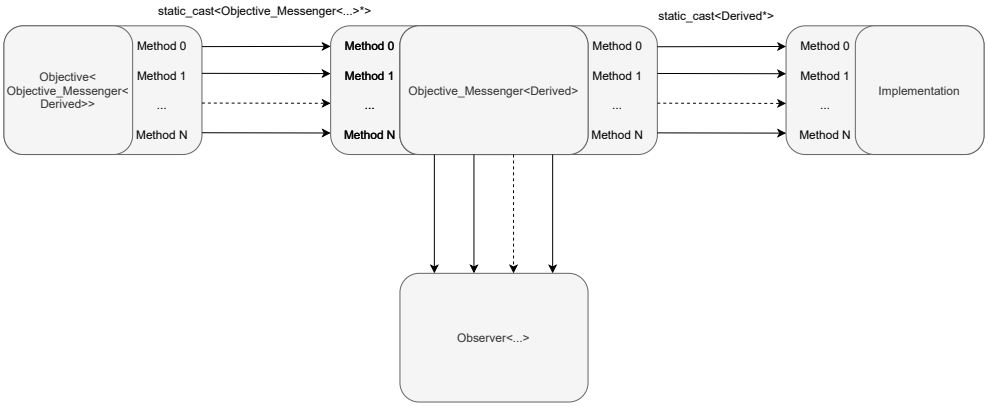


Figure 8.7: Method dispatching for messenger objectives

Figure 8.7 shows how a messenger pass the input/output data passed through it to an attached observer, which can be used to process the input/output data without interfering with the algorithmic design.

8.3.5 Journalist

The journalist-classes are derived from observers and used to write the intermediate input/return value pairs to separate csv-files. The journalist is intended

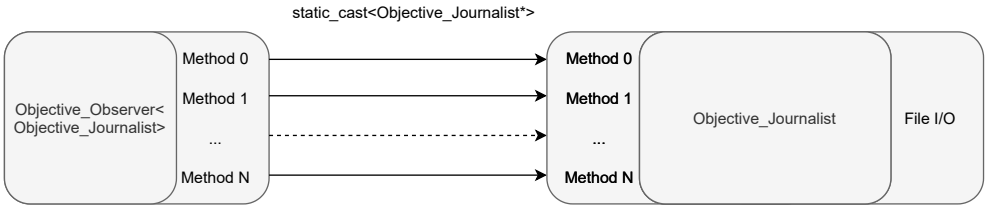


Figure 8.8: Method dispatching for journalist objectives

to demonstrate how data can be extracted and used during runtime.

8.3.6 Objectives for Quadratic Programs

Generic classes for Quadratic Programs (QPs) with linear constraints can be created by providing implementations to the base classes.

$$\begin{aligned} \min_{x} \quad & \frac{1}{2} x^T Q x + c^T x \\ \text{s.t.} \quad & \\ & Ax - b = 0 \\ & Dx - e \geq 0 \\ & x_{lb} \leq x \leq x_{ub} \end{aligned} \tag{8.1}$$

Parameters $Q, c, A, b, D, e, x_{lb}, x_{ub}$ are passed into the constructor of the class, which allows dimensions N_x, N_g, N_h to be implicitly deduced. A templated base class (`objective_QP_dense_base<>`) enables the same implementation to be used for normal, memoized and messenger objectives.

8.3.7 Decoded SIF Objectives

By calling the appropriate functions from `cutest.h` it is possible to load parameters for decoded SIF-problems, and to call dense/sparse fortran-subroutines to evaluate values. SIF-NLP classes use the same inheritance design as QPs to enable normal, memoized and messenger objectives. Figure 8.9 illustrates the similarities for the basic QP and SIF objectives.

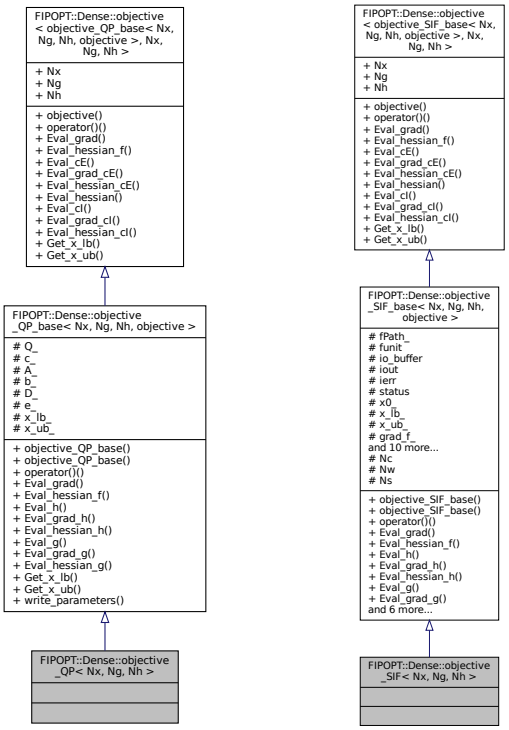


Figure 8.9: Inheritance diagrams for normal dense QP and SIF-objectives

8.3.8 Restoration Phase Objectives

Restoration phase objectives depend on the methods from the original NLP objectives, which are included with object composition. The restoration phase objectives implement equation 3.14 for inequality and equality constraints, yielding `equality_restoration_dense` and `inequality_restoration_dense` in figure 8.10.

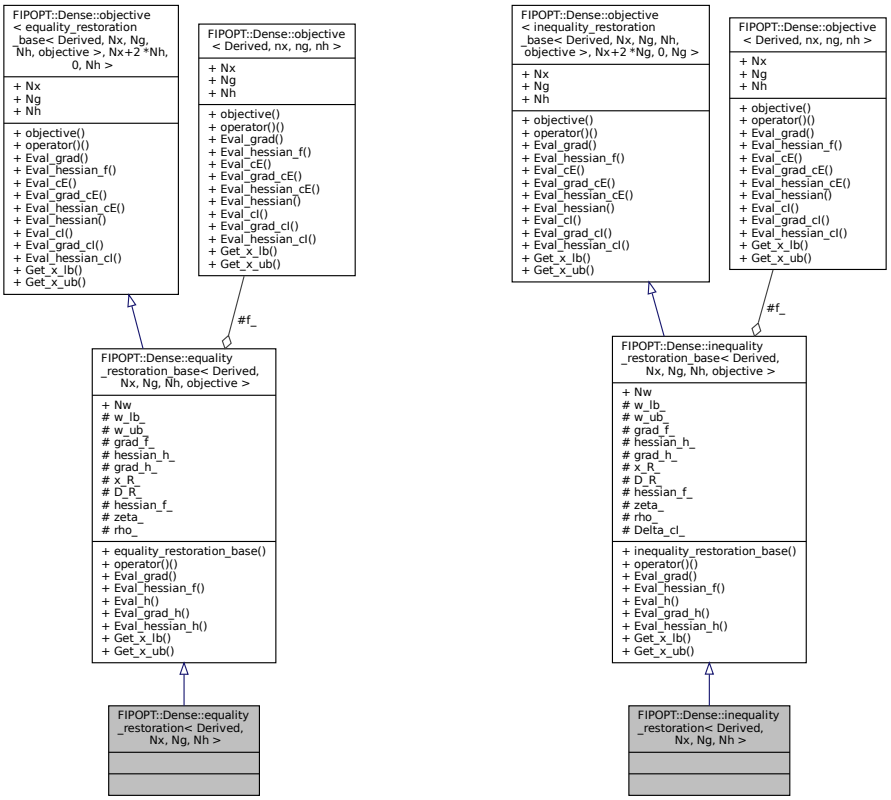


Figure 8.10: Inheritance diagrams for normal restoration phase objectives

8.3.9 Unifying Sparse and Dense Functors

If an algorithm requires evaluations using both dense and sparse methods it is desirable to derive a functor provides both.

Unfortunately this also introduces *overload ambiguity*. When calling a method from `objective<>` the *function signatures* of the dense and sparse NLP methods are exactly the same, despite having different return values. This means that the compiler is unable to determine which method to use, effectively making the unified functor a redundant abstraction. One way to resolve this is to use a pass-by-reference input argument in order to incorporate the return value into the function signature, but this would break the return-value flow used throughout the rest of the implementation.

8.3.10 Barrier Functor

The barrier functor implements methods for computing the barrier function and its gradient.

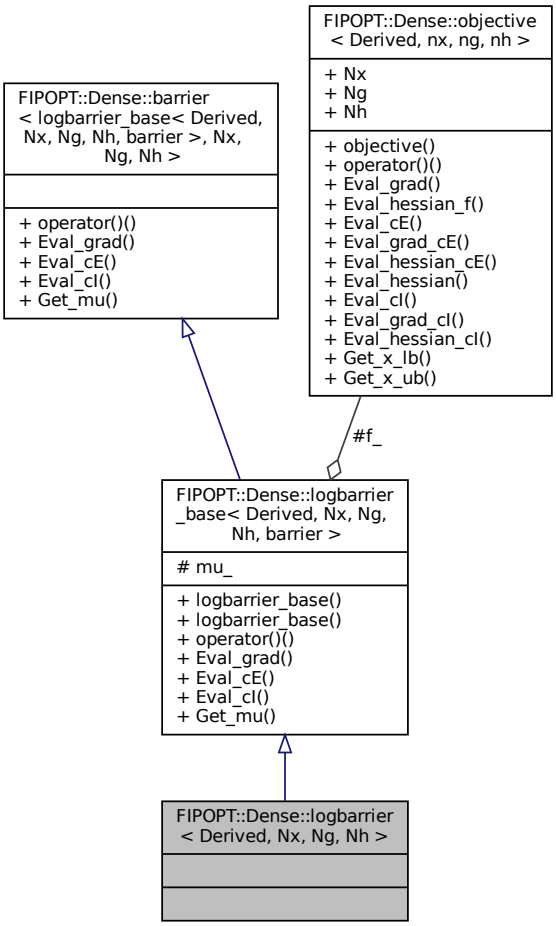


Figure 8.11: Logarithmic barrier functor collaboration graph

The base class use object composition to access methods from the NLP-objective, which makes it possible to use all memoized and base variants of the base NLP objective classes.

8.4 Fletcher-Leyffer Filter

The `FL_filter` class implements methods for all filter conditions specified in subsection 3. The filter assigns x_k, d_x for the current iteration, and use object com-

position to access methods for $\varphi_\mu(x)$ and $\theta(x)$. Due to the high amount of state comparisons performed inside the filter it is impractical to use pure functions to perform the conditional checks. The full evaluation and update of the filter is performed in one function call (`::Eval_Update_Filter(const double& alpha)`), which will update the filter by adding a `condition_pair` to the filters internal state.

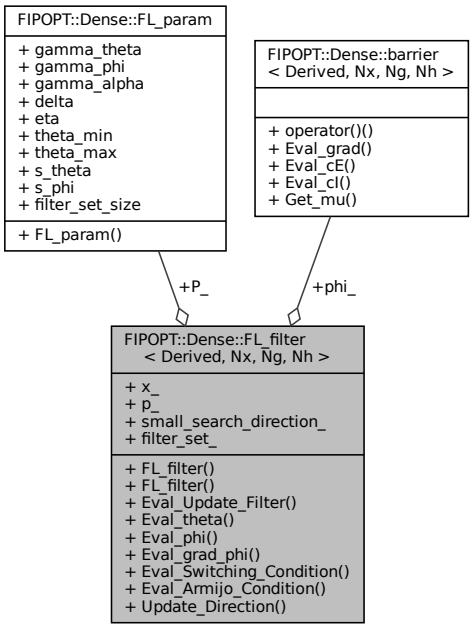


Figure 8.12: FL_filter collaboration graph

The filter currently contains internal state that alters how its methods operate. This is not good with respect to the functional design, but was intended to give better abstractions for the current application. The `FL_filter` class handles most of the acceptance and rejection decisions made by the algorithm.

Lambda-expressions were considered to implement $\theta(\alpha)$ and $\varphi(\alpha)$ as objectives fixed in a linesearch at (x_k, d_{x_k}). However, due to the arguments in section 6.2.1 and the overhead produced when the expression is replaced, it was simpler and more readable to introduce methods for linesearch directly inside the filter.

8.5 Algorithms

Algorithms follow some of the same design principles as the functors, but only implements one behavior each through a single function call. Parameters used in the algorithms are packaged in data structures (`_Param.hpp`) and status re-

turn values are provided to interpret their outcomes (`_Status.hpp`). Instead of presenting the detailed function signatures for the algorithms, the calls that are used inside the code is instead presented to provide explanations in context of its usage. Many of the algorithms use `static` member functions to explicitly specify template parameters.

8.5.1 Inertia Correction

Code listing 8.1: Inertia correction function call

```
IC_solver<LinSolver>::
Solve(KKT_mat, KKT_vec, d_x, d_lbd, delta_w_last, mu, fPath, IC_param)
```

The function computes and eigenvalues using the Eigen's general eigenvalue-solver, and corrects inertia according to algorithm 1. The result of a successful inertia correction is stored directly into `d_x, d_lbd`, and the previous correction value `delta_w_last` is updated. `SOC_param` contains all inertia-correction parameters. Correction scalars δ_w, δ_c are written to `fPath/delta.csv` after each successful correction.

8.5.2 Second-Order Correction

Code listing 8.2: Second-order correction function call

```
SOC_system<LinSolver>::
Solve(f, phi, F, KKT_mat, x, lbd, z, d_x, d_lbd, alpha, SOC_param)
```

Second-order correction requires objective `f`, barrier `phi` and filter `F`, and solves Newton system (2.38) with a `KKT_mat` fixed. `alpha = 1` by default, and accepted second-order corrections are assigned back to `d_x, d_lbd`.

8.5.3 Linesearch

Code listing 8.3: Second-order correction linesearch function call

```
SOC_Filter_LineSearch<LinSolver>
::Solve(f, phi, F, KKT_mat,  x, lbd, z, d_x, d_lbd, alpha)
```

This function implements algorithm 3 and additionally forwards input arguments to call second-order correction. Trial step sizes are assigned directly to `alpha`.

8.5.4 Barrier Subproblem

Code listing 8.4: Barrier subproblem function call

```
barrier_subproblem<LinSolver>::Solve(f, x, lbd, z, mu, tau, fPath)
```

This function implements algorithm 4 which involves allocation of the KKT-system (2.25), search directions d_x, d_λ, d_z and step sizes $\alpha, \alpha_{z_g}, \alpha_{z_{ub}}, \alpha_{z_{lb}}$. Barrier `phi` and filter `F` are constructed in its scope, before listings 8.1 and 8.3 are iteratively called to solve the subproblem. A `BS_journalist` is constructed to output iteration data to files `x.csv, lbd.csv, z.csv, obj.csv, theta.csv` and `alpha.csv` in directory `fPath`.

8.5.5 Linesearch Filter-Barrier

Code listing 8.5: Linesearch Filter Barrier function call

```
LSFB<LinSolver>::Solve(f, x0, fPath);
```

The top-level call for the algorithm only require the objective function and an initial state. Multipliers are allocated and initialized in its scope, and barrier parameter μ_j and fraction-to-the-boundary parameter τ_j are updated as each subproblem is solved. A `LSFB_journalist` is constructed to output iteration data to `x.csv, lbd.csv, z.csv, mu.csv, obj.csv, theta.csv` in directory `fPath`.

8.5.6 Restoration Phase

Code listing 8.6: Restoration phase fuction call

```
Restoration_Solver<LinSolver, Restoration_Objective>::
Solve_LSFB(f, x, z, mu, fPath, P)
```

With the addition of template parameter `Restoration_Objective` it becomes possible to use the same function for both restoration objectives `inequality_restoration<>`, `equality_restoration<>`. The restoration phase differs from the LSFB algorithm in its initialization of multipliers, and the least-square estimate (equations 3.17, 3.18) that it evaluates upon failure. The restoration phase utilize the `LSFB_journalist` to output iteration data similarly to the original linesearch filter-barrier method.

8.6 Python Binders

Pybind11 [34] provides simple integration between C++ and Python for both function and classes. Enabling components from C++ to be called from Python allows for prototyping that avoids recompilation. Binders can also be created for templated classes and methods as demonstrated in listing 8.7.

Code listing 8.7: Python-binder for the objective_QP_dense-class

```
using Mat = Eigen::MatrixXd;
template <int Nx, int Ng, int Nh>
const objective_QP_dense<Nx, Ng, Nh> load_QP(const Mat &Q,
                                             const Mat &c,
                                             const Mat &A,
                                             const Mat &b,
```

```
                                            const Mat &D,
                                            const Mat &e,
                                            const Mat &x_lb,
                                            const Mat &x_ub)
{
    objective_QP_dense<Nx, Ng, Nh> f(Q, c, A, b, D, e, x_lb, x_ub);
    return f;
}

PYBIND11_MODULE(Binder_QP, m)
{

    using objective_QP = objective_QP_dense<Nx, Ng, Nh>;
    py::class_<objective_QP>(m, "objective_QP")
        .def("__call__", &objective_QP::template operator()<Vec_x>,
        py::return_value_policy::reference_internal);
    m.def(("load_QP" + name).c_str(), &load_QP<Nx, Ng, Nh>);
}
```

This creates a class called `objective_QP` and a function `load_QP` for loading the class in Python, which can be imported under library name Binder_QP. Pybind11 has built-in support for the Eigen-library. Python-binders are used to implement plotting-tools under the Plot-subdirectory in the project root folder.

8.7 The Sparse and Dense Modules

The solver have initially been implemented using fixed-size dense matrices. Fixed-size matrices provide most optimization through vectorization and operation features, and avoid dynamic allocation. Since the dense module is designed to use fixed-size matrices, it requires problem dimensions to be known at compile-time.

Eigens sparse matrices are on the other hand dynamically allocated, and does not provide the same matrix operations that is used in fixed-size matrices. Slicing, indexing, assignments and block operations are not supported in the same way for sparse matrices, which had some impact on the design of the sparse module, which deviated from the dense module design. The module was intended to provide results on the large-scale SIF-problems provided with CUTEst, but the sparse solver is in its current state not able to solve the problems.

While there is a deviation in the dense and sparse module designs, the majority of the sparse module's components operate correctly, including the `objective_SIF` functor that provide a sparse interface to CUTEst's subroutines.

8.8 Overview of the Implementation

Figure 8.13 shows a simplified overview that summarize the core components of the linesearch filter-barrier algorithm. Different `filter_set` are used for each barrier subproblem.

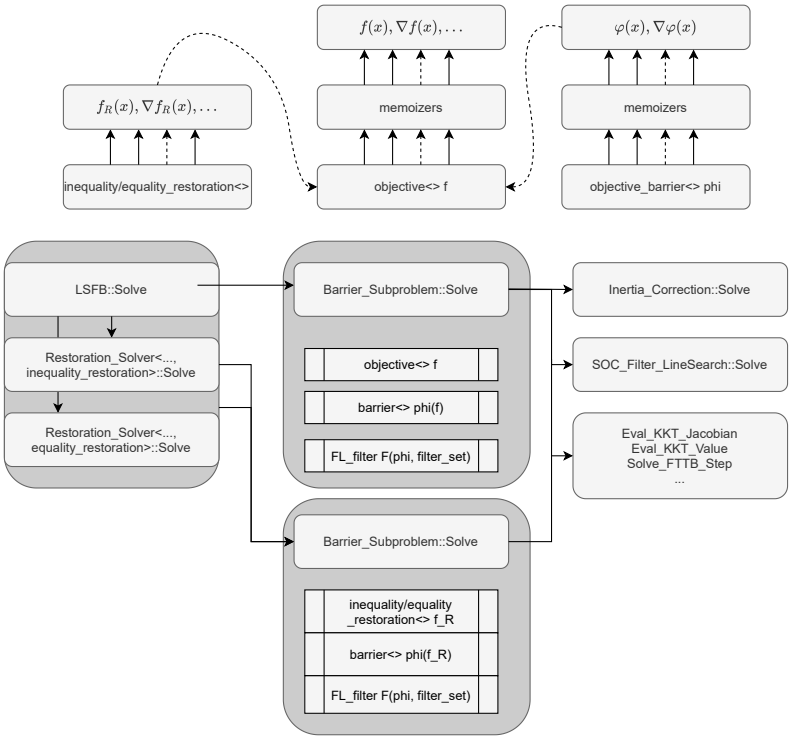


Figure 8.13: Collaboration overview of the thesis implementation

Chapter 9

Optimization Features

Compiled languages come with the advantage of knowing all instructions prior to execution. This enables a wide range of optimization capabilities, that can impact both the speed and the memory consumption of the resulting executable code.

9.1 Eigen

Eigen is a C header-only template library for linear algebra, and is used to both store and perform computations with vectors and matrices in the implementation. All of the classes provided by Eigen builds on top of the C++ standard library, it is actively developed and compatible with most modern C++ compilers. By being a templated header-only library it is in contrast to static and dynamic libraries not compiled as separate units from the main application. This is because the functions and classes produced by the library depends on the implementation itself.

By letting the library know which types and dimensions that are going to be used prior to its compilation, it is able to adapt better to the subject architecture and perform optimization on other levels that dynamic/static libraries cannot.

9.1.1 MatrixBase

Datatypes and expressions in Eigen are implemented as derived classes of some common base. Dense matrices and operations derive from the common `MatrixBase` that is implemented in the library's Dense module. Likewise sparse matrices derived from the common `SparseMatrixBase` implemented in the corresponding Sparse module. The bases can be combined in expressions, which forms a dense `MatrixBase`-expression as a result. By using bases instead of vectors/matrices it is possible to pass unevaluated expressions between functions, which enable Eigen to form expression trees that can be reduced at compile time. This makes the expressions lazily evaluated, which removes the need for temporary storage allocations of intermediate results.

9.1.2 SIMD

Single Instruction Multiple Data (SIMD) enables the same instruction to be performed on multiple data elements in parallel, which can be used to speed up linear algebra operations on vectors and matrices of small sizes. CPUs provide access to SIMD execution models through Streaming SIMD Extensions (SSE), which are Assembly instruction sets that can be used when specified to the C++ compiler. Eigen utilize SSE-instructions to vectorize both dynamic and fixed-size matrices and vectors.

9.1.3 Loop Unrolling

Loop unrolling improves runtime performance by reducing the number of iterations of a loop at the expense of adding additional instructions, effectively trading off binary size for runtime speed. Eigen greatly benefits from loop unrolling in small fixed-size matrices.

9.1.4 Aliasing

When a matrix/vector appears on both the left and right hand side of an expression a temporary allocation is required in order to perform the computation. Eigen provides `.noalias()` to disable temporary evaluations in cases where linear operations are safe to perform in place.

9.2 Compiler Optimization

9.2.1 Inlining

The `inline`-keyword is used to specify to the compiler that the body of the subject function can be replaced with its function calls. This removes the stack overhead caused by the function call, and also enables the compiler to optimize the function body code in context with the scope its function was called from. Inlining is very useful in combination with the CRTP-classes (section 5.2.3), since it removes the function calls required to dispatch to the derived implementation, and further enables optimization of the implementation in context of its caller. Inlining is useful for small function bodies. The compiler may ignore inlining in cases where it is inefficient.

9.2.2 Copy Elision

All values passed to and returned from functions in C++ are copies, which can potentially lead to an unnecessary amount of data copying and allocations. C++ compilers prevent this by eliminating the redundant temporaries using return value optimization. Additionally, modern compilers also move the construction of objects directly into the destination address, instead of copying

the data after construction. This generally makes it possible to avoid custom move semantics in the code.

9.2.3 Optimization Flags

Compilers provide a wide range of flags to enable optimization features. There are different flags for each compiler, whose effect can be very different depending on the implementation. Overall, C++ compilers group their flags in different optimization levels which increasingly improve runtime speed, binary size and memory usage from levels -O0 to -O3. Higher levels introduce higher levels of code transformation that is more expensive to compute, leading to a longer compile time. Additionally two flags are passed to enable instruction sets:

`-mfpmath=SSE` - Enables scalar floating point instructions from the SSE instruction set

`-march=native` - Enables GCC to utilize intruction sets specific to the local machine

Chapter 10

Linear Solvers

There is a wide range of different algorithms used to solve linear systems. In all algorithms there is a trade-off between accuracy in the result and the computational expense. The accuracy of a solution to a linear system is dependent on the *condition number* of its matrix, which quantifies how sensitive the system will be to errors in the input. The condition number for a *normal* matrix in an ℓ^2 metric space can be determined by its eigenvalues:

$$\kappa_{cond}(A) = \frac{\lambda_{max}(A)}{\lambda_{min}(A)} \tag{10.1}$$

This condition number impacts all linear solvers, but some types suffer more than others. Direct methods find the 'exact' (down to a κ_{cond}-dependent accuracy) solution to the system in a finite number of steps. Direct methods use row/column eliminiations and pivoting strategies to transform the system to another form, where front and back-substitution methods are used to obtain the solution. Matrix decompositions are used to reach these forms, and each type of decomposition have different properties for numerical stability, accuracy and computational efficiency. Eigen use direct decomposition methods for its dense linear solvers and provides performance benchmarks[35] and requirements[36] for the different types decompositions.

10.1 Dense Decompositions

10.1.1 Cholesky Decomposition

The Cholesky decomposition decompose a real, *hermitian* matrix into a matrix product consisting of triangular matrix L and its transpose:

$$LL^T x = b \tag{10.2}$$

$$Ly = b, \quad y = L^T x \tag{10.3}$$

The resulting equation is solved for x with two forward/back substitutions. This is the fastest decomposition available, but comes with strict requirements as a cost. The Cholesky algorithm require roots of the matrix elements, but this can be avoided by leaving unfactorized root-terms in a diagonal matrix D. This is the robust Cholesky decomposition (LDLT). The presented equality-constrained KKT-system (equation 2.25) always have negative eigenvalues, which makes LLT infeasible for solving subproblems, but LDLT is preferable.

10.1.2 QR-Decomposition

QR-decompositions transform real, square matrices into an orthogonal (Q) and upper-triangular (R) matrix-product:

$$QR = A \tag{10.4}$$

The QR-decomposition is useful for computing least-squares initial multipliers λ_0 since it avoids computing the covariance matrix A^TA by first *orthogonalizing* the system. Eigen provide QR-decompositions with algorithms using *householder-transformations*, which are numerically stable and provide better accuracy than the Cholesky decompositions.

10.1.3 LU-Decomposition

The LU-decomposition transform real, square matrices into an lower (L) and upper (U) triangular matrix-product:

$$LU = A \tag{10.5}$$

LU-decompositions has no symmetry requirement, which makes them roughly twice as slow as the LLT-decomposition. Eigen has a *partial pivoting* LU-decomposition algorithm implemented which requires A to be invertible.

10.2 Sparse Decompositions

There is a wide range of available sparse linear solvers which exploit the sparsity of the linear system in different ways. Equivalently to the dense decompositions there are also direct sparse solvers that aim to factorize and obtain a triangular matrix. Sparse decompositions have not been used to obtain results for this thesis, but they are important in order to be able to efficiently solve large-scale problems in the future.

10.2.1 Sparsity Pattern

Sparse linear solvers utilize the sparsity pattern of the systems matrix A, which is used to avoid performing computations with zero-elements. The sparse matrices

in Eigen by default stores its sparsity pattern in a *Compressed Column Storage* format (CCS), which allows a sparse matrix to be represented with an array of its nonzero values, and the row indices for these elements. This effectively reduces the worst-case time for element access and insertions to the total number of nonzero elements in the systems matrix.

10.2.2 Triangular Solve

Once a sparse matrix has been factorized, the system can be solved by traversing the sparsity pattern as a graph and solving the system for the elements encountered through that graph. Performing a *Depth First Search* while solving for the encountered elements will solve the triangular system (Figure 10.1).

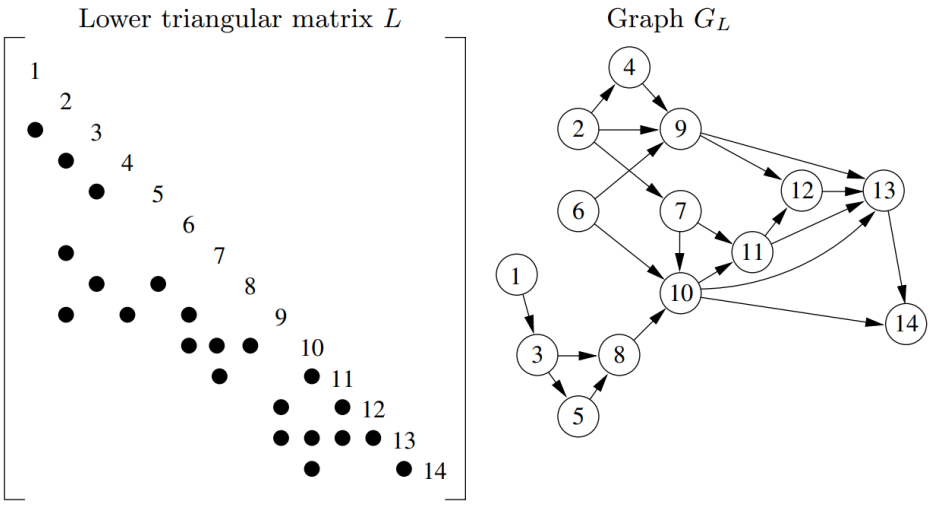


Figure 10.1: Graph view of a sparse triangular matrix[37]

MatLabs backslash operator (`mldivide`) utilize this to achieve a $O(n + nnzh)$ [37, p.35], where n is the square matrix dimension of L and $nnzh$ the nonzero elements.

10.2.3 Direct Decompositions

Eigen provides sparse LLT, LDLT, LU and QR decompositions with similar matrix requirements and properties to the dense decompositions. Similarly to the triangular solve the decomposisions only have to traverse through elements related to the sparsity pattern, which will make the worst-case solve time dependent on $nnzh$ rather than only the full square matrix dimension.

Chapter 11

Results

11.1 Convergence Results

A selection of problems have been selected to demonstrate the convergence of the algorithm. Some quadratic programs have been specified directly, while nonlinear programs are obtained from the Hock-Schittkowski collection. Problems are solved to optimality error tolerance $\epsilon_{tol} = 1e^{-8}$ overall, and subproblem tolerance given by the stopping condition (2.31). The problem is set to terminate if repeated inequality or equality restoration phases are called without any progress. Parameters for the algorithm are summarized in appendix A.

The barrier function values are plotted as a filled contour while objective function values are plotted with contour lines. The infeasible region is marked with small x-markers. Equality constraints are plotted with contour levels to some tolerance ($c_E(x) < 0.1$), which means that the feasible curve will be somewhere in an interval between two contour levels.

Iteration plots show all barrier subproblem iterations, where the transition to a new barrier parameter is marked with dots. The plots are shown without subproblem iterations for restoration phases. Sudden spikes in iteration values (present in the end of HS13, HS14 and start of HS14) are caused by the return from restoration phases.

Note: All results are obtained from the fixed-size dense module.

11.1.1 Linear Inequality-Constrained Quadratic Program

$$\begin{aligned} \min_{w} \quad & \frac{1}{2} x^T \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} x \\ \text{s.t.} \quad & \begin{bmatrix} 1 & 1 \end{bmatrix} x \geq 20 \\ & x_0 = \begin{bmatrix} 40 \\ 25 \end{bmatrix} \end{aligned} \tag{11.1}$$

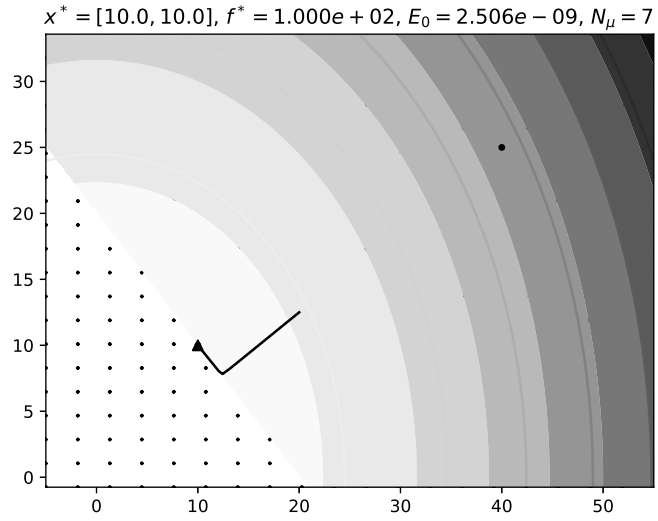


Figure 11.1: Full trajectory for linear inequality-constrained QP

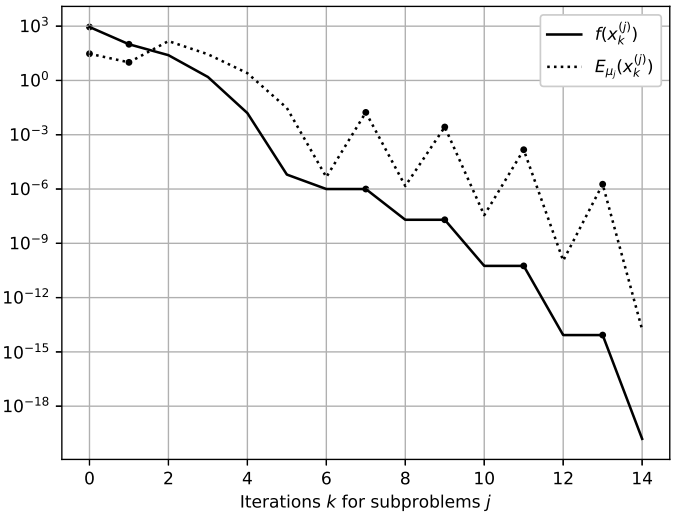


Figure 11.2: Iteration optimality for linear inequality-constrained QP

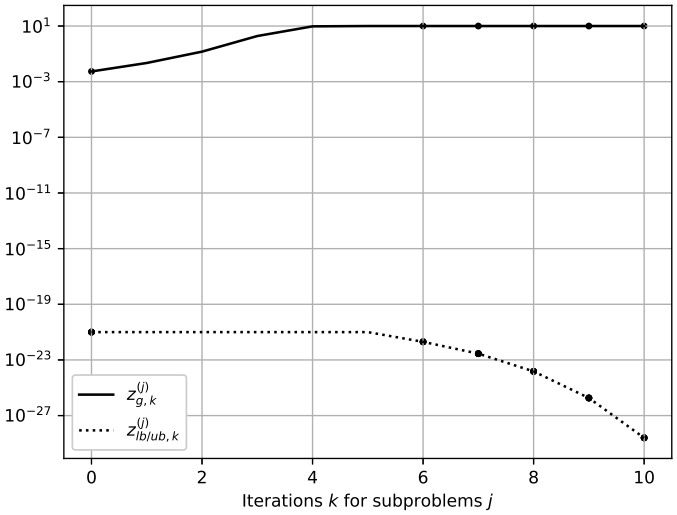


Figure 11.3: Inequality multipliers for inequality-constrained QP

11.1.2 Inequality-Constrained Quadratic Program (Infeasible)

$$\begin{aligned} \min_{w} \quad & \frac{1}{2} x^T \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} x \\ \text{s.t.} \quad & \begin{bmatrix} 1 & 1 \end{bmatrix} x \leq 20 \\ & x_0 = \begin{bmatrix} 40 \\ 25 \end{bmatrix} \end{aligned} \tag{11.2}$$

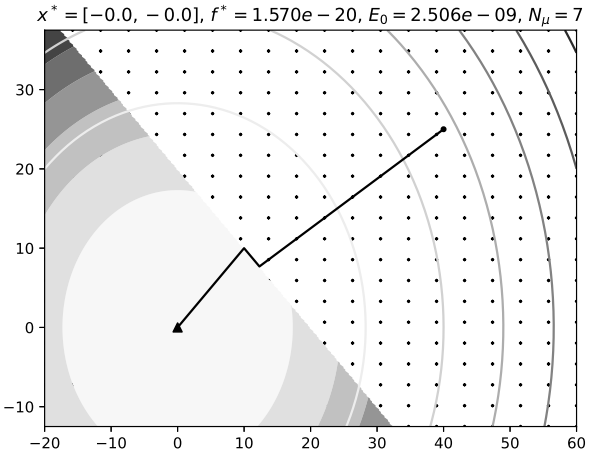


Figure 11.4: Full trajectory for linear inequality-constrained QP (infeasibile)

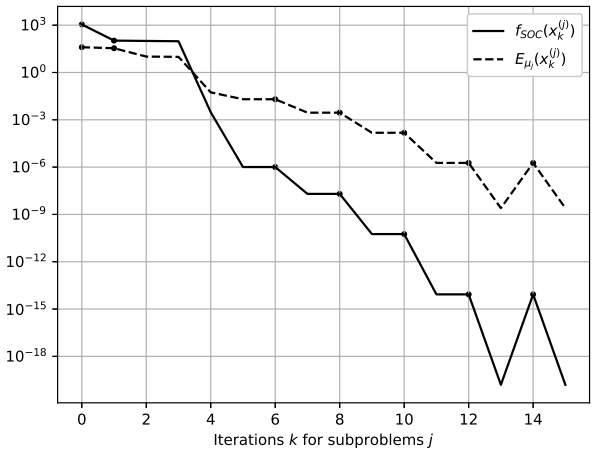


Figure 11.5: Iteration optimality for linear inequality-constrained QP (Infeasible)

11.1.3 Nonlinear Inequality-Constrained Quadratic Program

$$\begin{aligned} \min_{w} \quad & \frac{1}{2} x^T \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} x \\ \text{s.t.} \quad & x_0^2 + x_1^2 \geq 1 \\ & x_0 = \begin{bmatrix} 2 \\ 1 \end{bmatrix} \end{aligned} \tag{11.3}$$

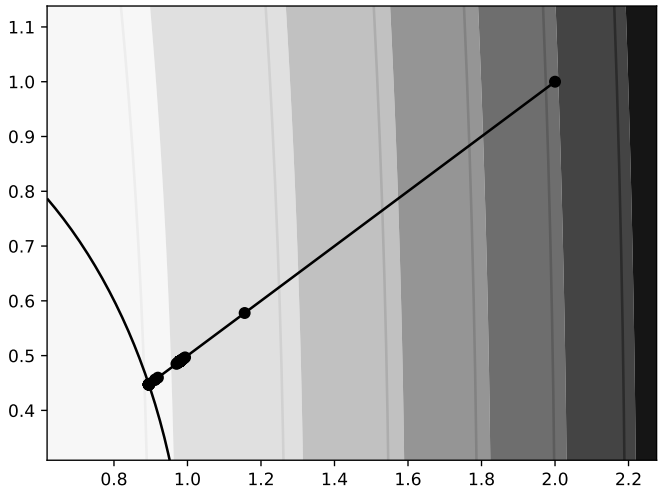


Figure 11.6: Full trajectory for nonlinear inequality-constrained QP

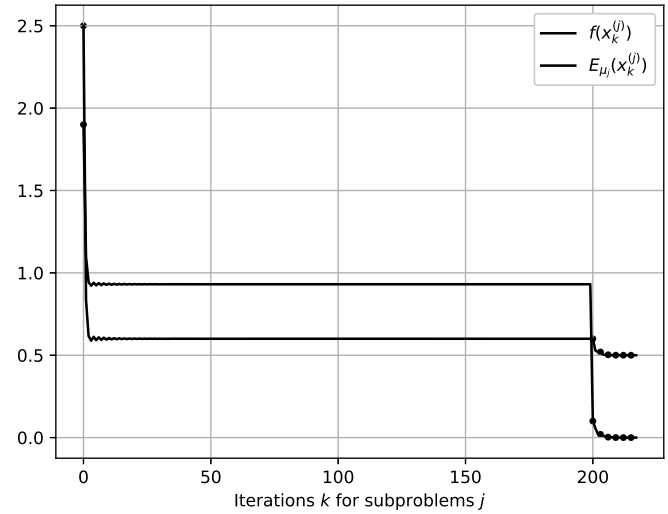


Figure 11.7: Iteration optimality for nonlinear inequality-constrained QP

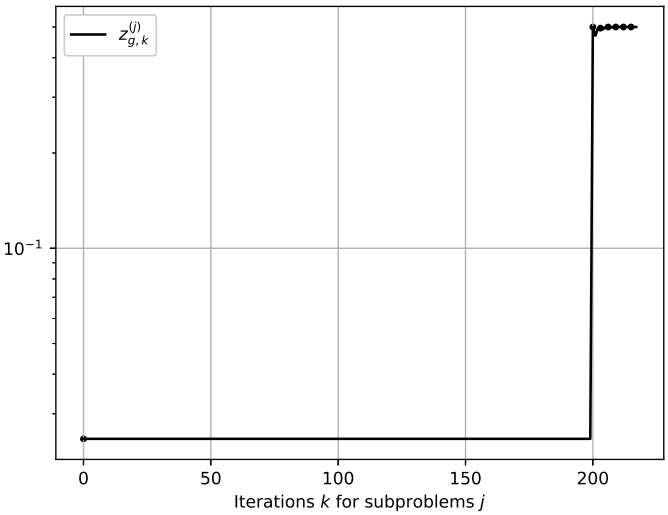


Figure 11.8: Inequality multipliers for inequality-constrained QP

11.1.4 Nonlinear Equality-Constrained Quadratic Program

$$\begin{aligned} \min_{w} \quad & \frac{1}{2} x^T \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix} x \\ \text{s.t.} \quad & x_0^2 + x_1^2 = 1 \\ & x_0 = \begin{bmatrix} 1 \\ 1e^{-3} \end{bmatrix} \end{aligned} \tag{11.4}$$

Solution was obtained with modified switching-condition filter-parameters $s_\varphi = 2.3, s_\theta = 1.0$.

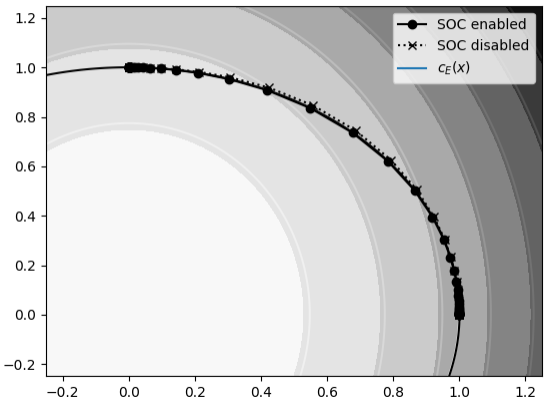


Figure 11.9: Trajectories for nonlinear equality-constrained QP

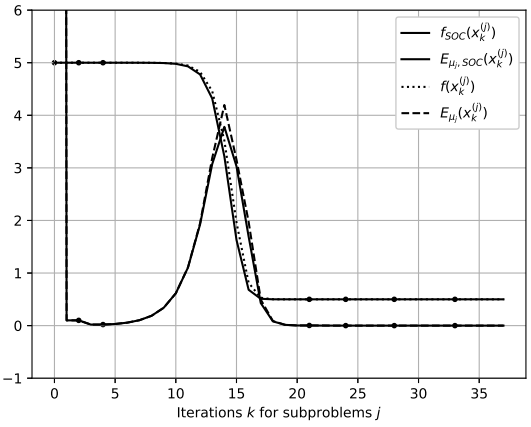


Figure 11.10: Iteration optimality for nonlinear equality-constrained QP

11.1.5 Unconstrained 2-Dimensional Rosenbrock Function

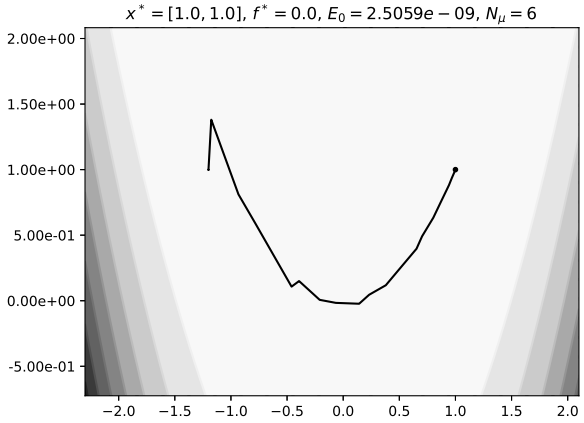


Figure 11.11: Full trajectory for 2-Dimensional Rosenbrock NLP

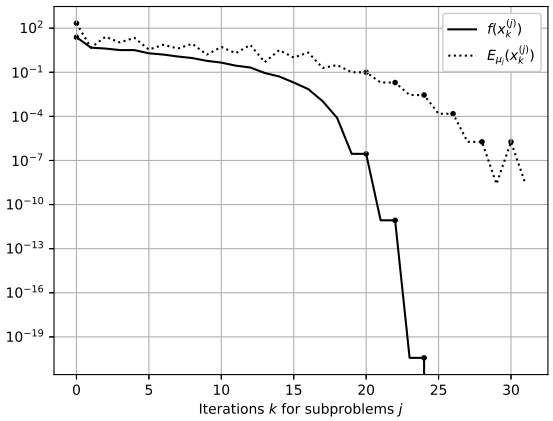


Figure 11.12: Iteration optimality for 2-Dimensional Rosenbrock NLP

11.1.6 HS2 (Converged)

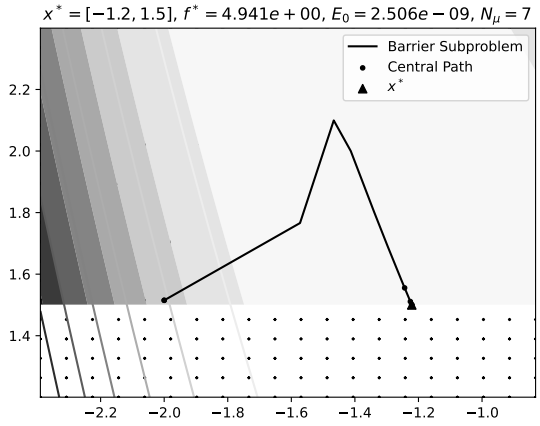


Figure 11.13: Full trajectory for HS2

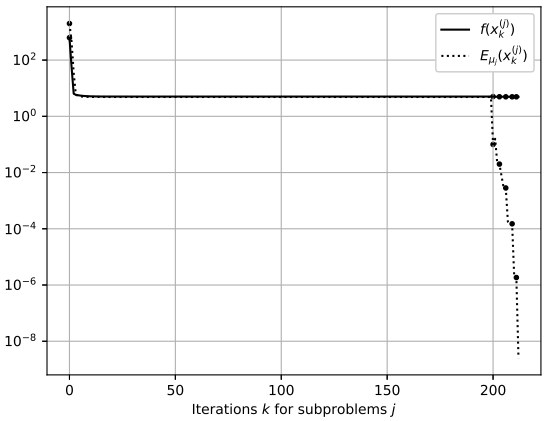


Figure 11.14: Iteration optimality for HS2

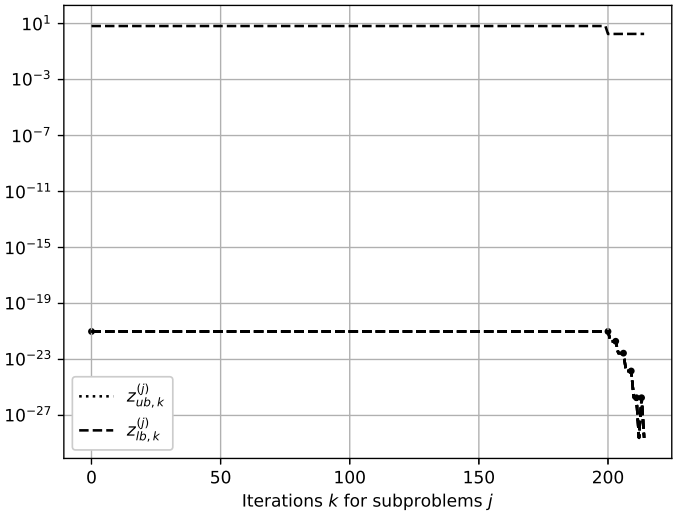


Figure 11.15: Inequality multipliers for HS2

11.1.7 HS12 (Not Converged)

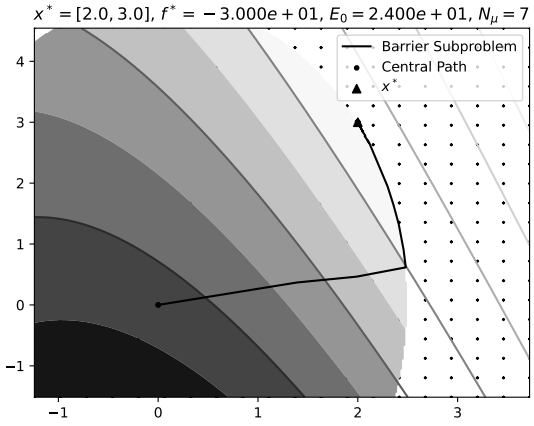


Figure 11.16: Full trajectory for HS12

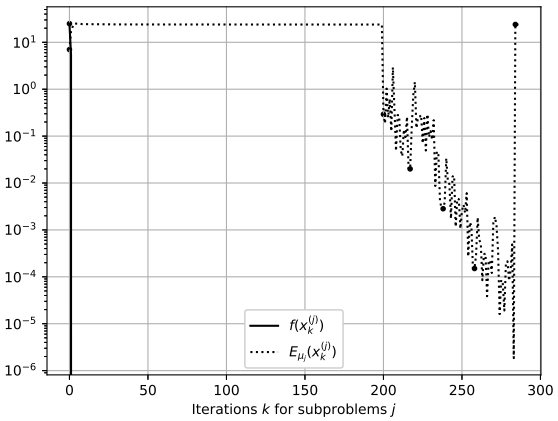


Figure 11.17: Iteration optimality for HS12

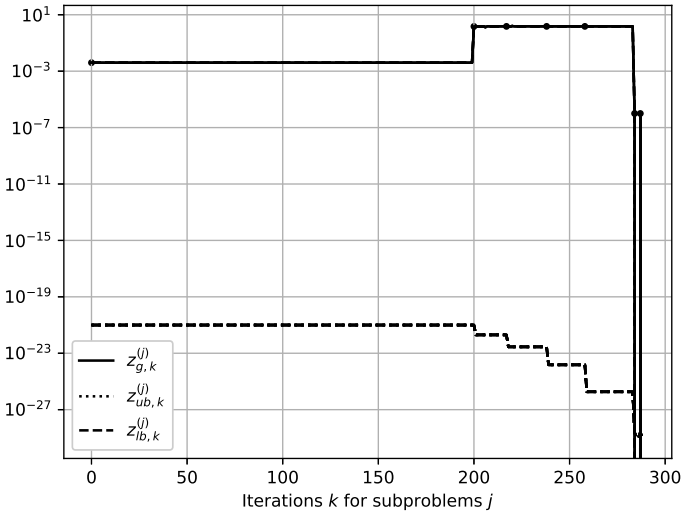


Figure 11.18: Inequality multipliers for HS12

11.1.8 HS13 (Not Converged)

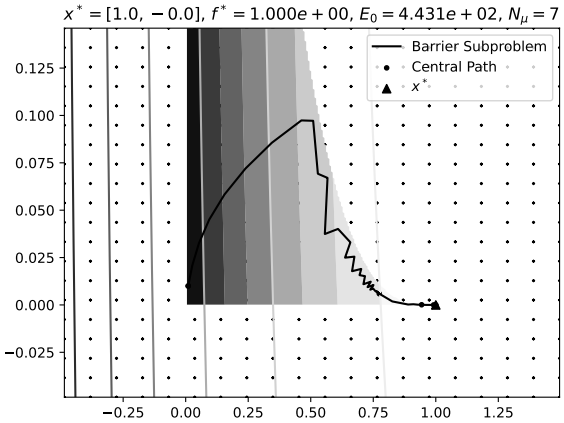


Figure 11.19: Full trajectory for HS13

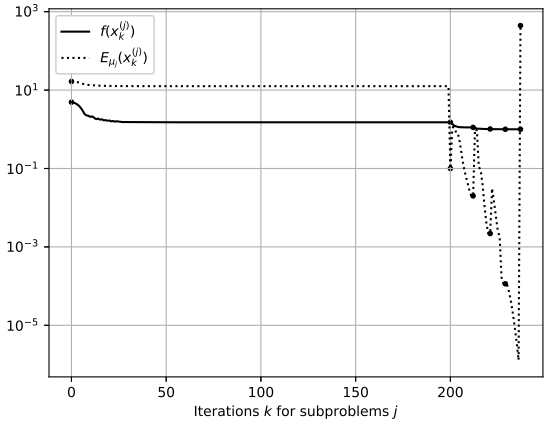


Figure 11.20: Iteration optimality for HS13

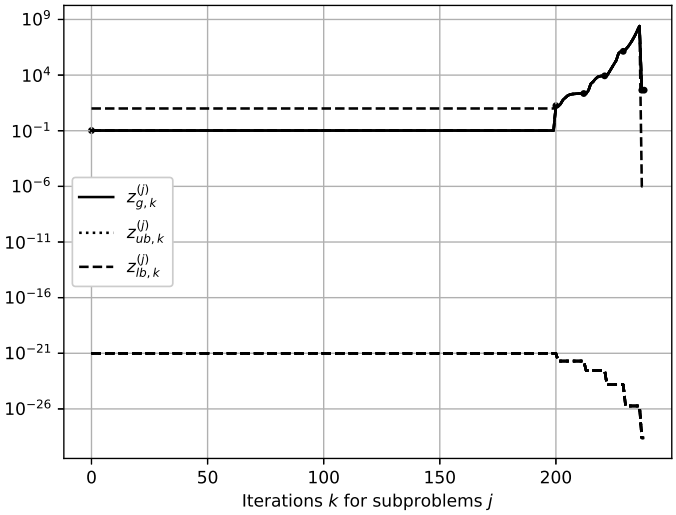


Figure 11.21: Inequality multipliers for HS13

11.1.9 HS14 (Converged)

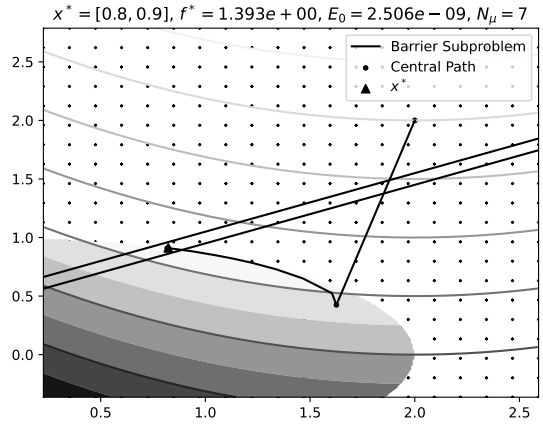


Figure 11.22: Full trajectory for HS14

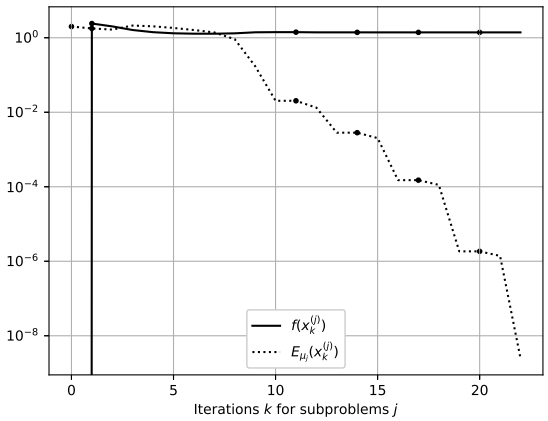


Figure 11.23: Iteration optimality for HS14

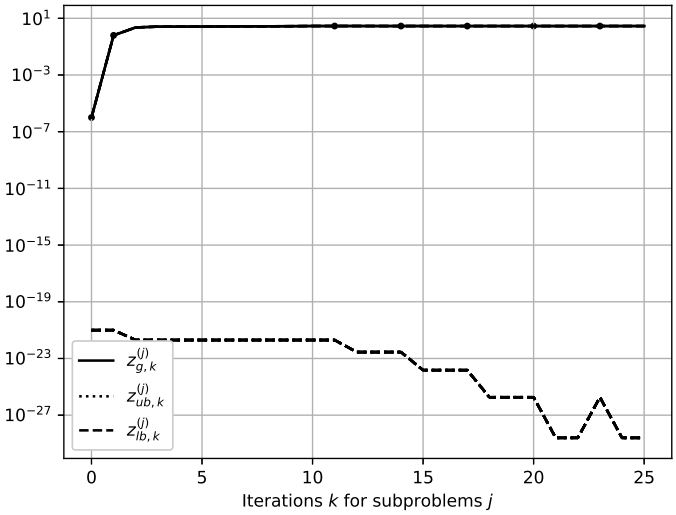


Figure 11.24: Inequality multipliers for HS14

11.1.10 HS17 (Converged)

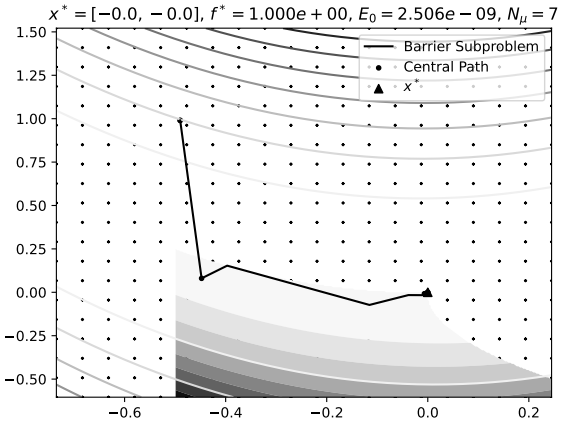


Figure 11.25: Full trajectory for HS17

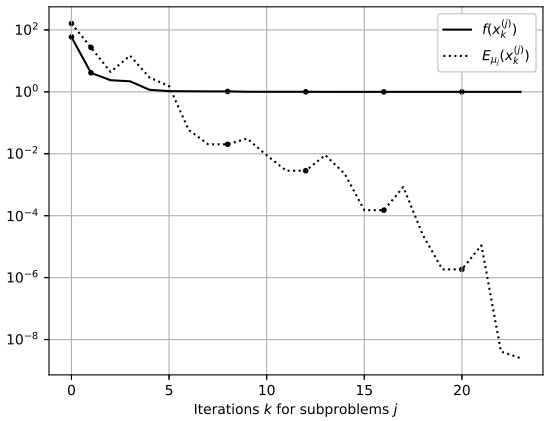


Figure 11.26: Iteration optimality for HS17

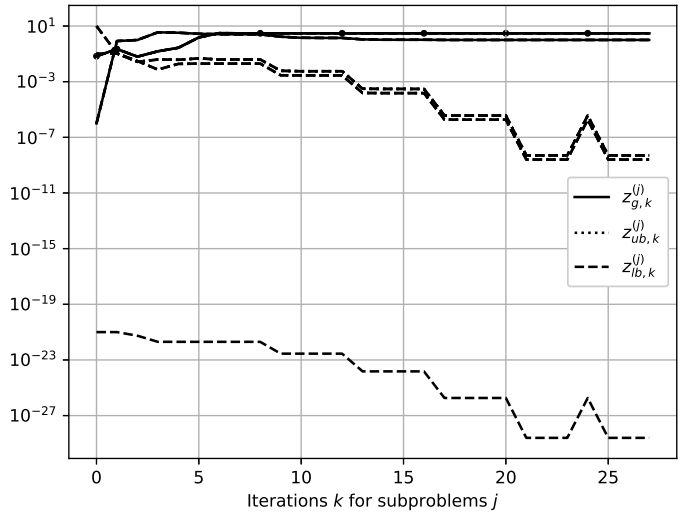


Figure 11.27: Inequality multipliers for HS17

11.1.11 Additional Hock-Schittkowski Results

Plots for remaining problems from the Hock-Schittkowski test set are provided in a separate document[38].

11.2 Performance Results

Performance results are obtained by running both IPOPT and the thesis implementation on the Hock-Schittkowski collection. The solve time is obtained using Python's `timeit()`-function, which executes the binary multiple times and averages the execution time. PartialPivLu has been used as the linear solver for these results.

The binaries for solving the SIF-problems have been compiled with the following optimization flags:

```
-msse2 -mfpmath=sse -march=native -O3
```

IPOPT has been configured and installed only with Intel Math Kernel Library (MKL) as BLAS-provider[39] according to its installation instructions[40]. IPOPT is configured with MUltifrontal Massively Parallel Solver (MUMPS[4]) as linear solver. Memoizers were configured with a ringbuffer size of 4 input-output data pairs for the memoized solve.

The thesis implementation successfully converged and terminated for 55 of the total 118 HS-problems, which are the problems illustrated in 11.28 and 11.29, and came sufficiently close to convergence (without termination) for 5

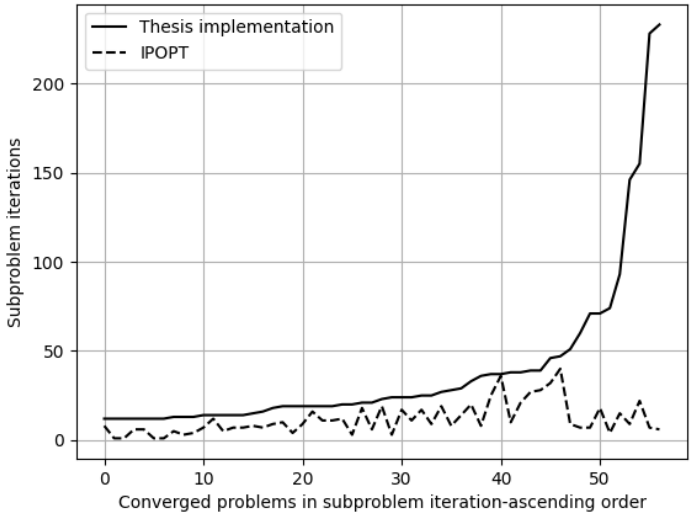


Figure 11.28: Subproblem iterations for converged HS-problems

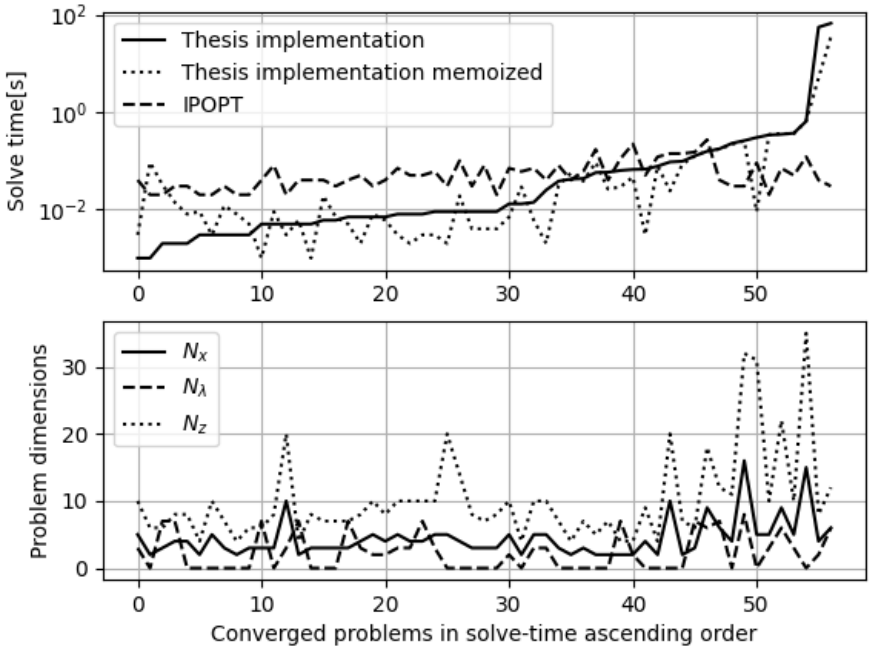


Figure 11.29: Problem dimensions and time consumption for converged HS-problems

problems, resulting in 58 problems that did not converge. 'Sufficiently close' is considered to be the distance to IPOPT's optimal value:

$$||x^*_{thesis} - x^*_{IPOPT}||_\infty \leq 1e^{-3} \tag{11.5}$$

	Converged	Sufficiently Close	Not Converged
Inequality Restoration	3	4	37
Equality Restoration	13	1	22

Table 11.1: Number of problems that used restoration phases in converged/unsolved problems

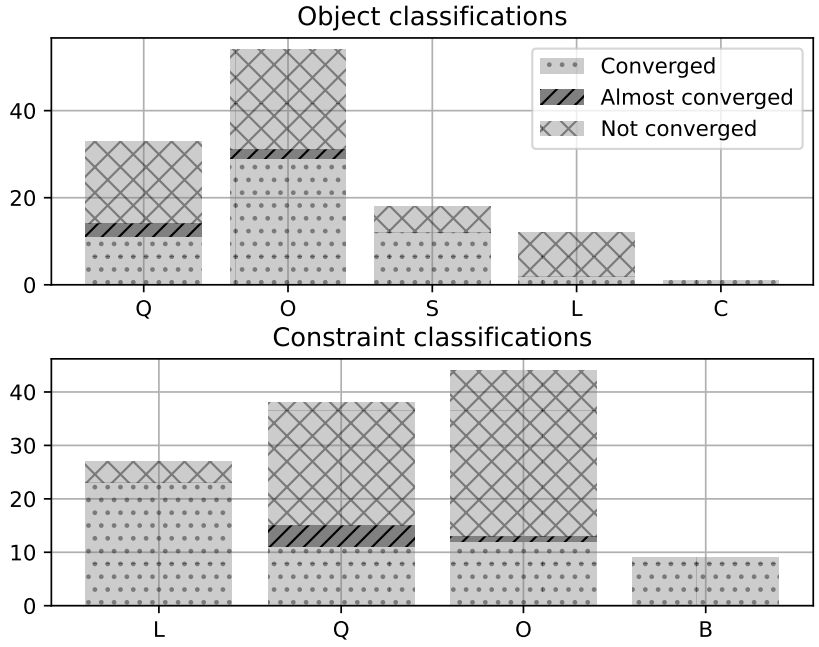


Figure 11.30: Classification counters for HS-problems (separate)

All problems are classified as regular in smoothness with continous second order derivatives everywhere.

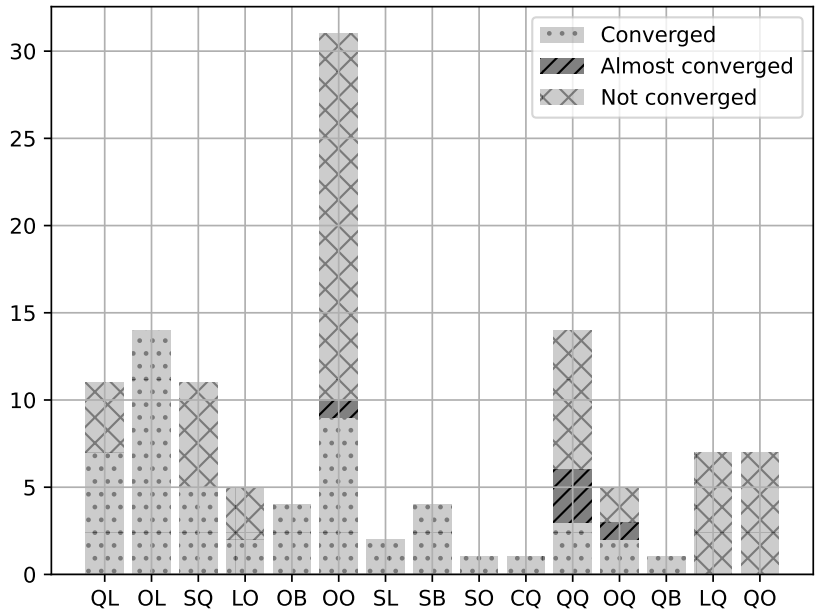


Figure 11.31: Classification counters for HS-problems (combined)

Chapter 12

Discussion

12.1 Design

Different design choices had to be made in order to weigh readability and flexibility against efficiency in the implementation. Some components of the solver have been made in a declarative style, while others exploit the efficiency of imperative programming. Some aspects of this will be discussed by reviewing these different components.

12.1.1 The NLP and Barrier Functors

The NLP functors can be considered as one of the most successful components of the implementation. The objective-classes (along with the derived memoized and messenger-objectives) pass Code Complete's checklist for class quality [41, p.158].

- The objective and barrier-classes (and its derivatives) have one central purpose and is named appropriately to emphasize that purpose.
- These classes properly encapsulate data, and avoid redundant internal states. The only data members that can alter behaviour in these classes are kept in the memoizers which are protected with `::get_data, ::set_data` -methods. In other cases the data is initialized as constants or marked private.
- The classes contain a limited number of methods that is appropriate to cover their purpose. This makes it easier to interpret the purpose of a class and makes the class more readable.

The design of the restoration phase algorithm illustrates how seamlessly this abstraction can be used to introduce new NLPs. The class templates provide instantiations for problems of any size, and enable any templated function in which it is used to adapt to that size, all evaluated at compile-time. From the user-perspective a new objective can be constructed in a few lines of code, without any side-effects. This was a significant advantage in the implement-

ation of the restoration phase, which is able to re-use almost all subroutines besides the main LSFB algorithm.

12.1.2 Fletcher-Leyffer Filter

The Fletcher-Leyffer filter turned out to become one of the most inconsistent design choices in the implementation. The filter tries to encapsulate bi-objective pairs $(\theta(x), \varphi(x))$ and to make all decisions for the algorithm dependent on these states. $(\theta(x), \varphi(x))$-dependent decisions have to be made in all algorithms contained within a barrier subproblem, which consequently makes the `FL_filter` class essential everywhere at this level. The filter violates many of the traits in the aforementioned class quality checklist.

- The filter does not have a central purpose. The filter decides the acceptability of a step while it simoultaneously modifies its internal state for future decisions.
- The name of the class does not describe its purpose(s). This makes the class act as a common namespace for methods that are correlated with different purposes.

In order to address these issues the pairs of $(\theta(x), \varphi(x))$ were removed from the filter class and treated as an external abstract data type. This could (in retrospect) be a better choice, and should be pursued in a redesign. The concept of a filter could itself be a counter-intuitive design choice which allows several different components to be recognised under the same name. For example, the Armijo descent condition (3.7) and the switching condition (3.6) are not only relevant to the acceptance of a step, but also used to determine if second-order corrections should be performed.

12.1.3 The Algorithms

The algorithms generally follow the same implementation structure:

Algorithm 7 General algorithm structure pseudocode

Require: Input arguments
  Allocate and Initialize data
  for $i \leftarrow 0$ to i_{max} **do**
    if Feasible condition **then**
      Return success
    end if
    Atomic action
    Atomic action
    ...
  end for
  Return failure

By isolating allocations to the initial stage of each algorithm, and separating computations into atomic actions in the iterative loop, it becomes simpler to backtrace to exactly where and why the solvers fails to converge to a feasible point. This structure is also very useful to resolve compile-time/runtime allocation errors in the design process, since they often are caused by the same dimension parameters.

The algorithms follow guidelines from Code Complete's checklist for quality routines[41, p.185] with (for efficiency reasons) some relaxations.

- Each algorithm is an essential component of the solver that is functionally cohesive, it is centered around fulfilling one purpose.
- The algorithms are loosely coupled to its subroutines, allowing subroutines to be replaced without side-effects. One exception to this property is the Fletcher-Leyffer filter, which violates this with its internal state.
- The length of each algorithm are appropriate to its purpose, ranging from 30-50 lines of code from the lower level (Inertia/second-order correction) to higher level routines (LSFB/Barrier subproblems).
- Input arguments are constrained to a readable size (≤ 7) by grouping algorithm-specific parameters into data structures.
- The return value for each algorithm clearly describes its outcome with status-encoded C++ enumerations, and all outcomes are properly handled by the parent routine.

Input arguments are also used to return values, which makes it harder to determine each arguments purpose. Some distinction is made by enforcing immutability on the arguments with the `const`-keyword.

12.1.4 Journalists and Memoization

The functor-journalists combined with the journalists for the algorithms manage to retrieve almost all values that are computed in the solver, and they are able to do so in a way that minimally interfere with the design of the algorithms. The algorithm journalists were found to be the most useful for the application in this thesis, but the functor journalists could become useful in cases where one wants to obtain data on a lower level. The functor layer design used in the memoization and messager functors depend on repetitious code, but it can be argued that the purpose of that code outweighs its size.

12.1.5 Typenames and Template Parameters

One persisting large overhead is the specification of types for each function, which (as mentioned in section 6.8) can either be deduced or explicitly specified. The behavior of types should be constrained, which is why types like `objective<Derived, Nx, Ng, Nh>` is used instead of a single template parameter `Objective`. Vector and matrix dimensions add additional overhead, but this was found to be more concise and easier to read than the implicit alternative.

Template parameters have generally been avoided, besides for problem dimensions, CRTP-classes and configuration of the linear solver. Another exception is the nested `template <class, int, int, int> typename Base` parameter for objectives, which enables memoization, messengers and journalists to be configured under the same implementation.

Code listing 12.1: Partial parameter specification of SIF-objectives

```
template <int Nx, int Ng, int Nh>
struct objective_SIF :
public objective_SIF_base<Nx, Ng, Nh, objective>
{
    using objective_SIF_base<Nx, Ng, Nh, objective>::objective_SIF_base;
};

template <int Nx, int Ng, int Nh>
struct objective_SIF_memoized :
public objective_SIF_base<Nx, Ng, Nh, objective_memoized>
{
    using objective_SIF_base<Nx, Ng, Nh, objective_memoized>::objective_SIF_base;
};

template <int Nx, int Ng, int Nh>
struct objective_SIF_journalist :
public objective_SIF_base<Nx, Ng, Nh, objective_journalist>
{
    using objective_SIF_base<Nx, Ng, Nh, objective_journalist>::objective_SIF_base;
};
```

Configuring typenames this way leads to extra typename configuration, which is less readable than direct implementations. Templated object composition could be a viable alternative to this approach.

12.1.6 Replacement of Design Patterns

The polymorphism from `AlgorithmStrategy`-classes from table 5.1 have been removed and instead directly implemented into the algorithms. This is less flexible, but can easily be adapted by adding a higher-order function as input argument or input template parameter. The `AlgorithmBuilder` naturally disappears with this choice, but would otherwise be replaced with function composition.

The `messenger` and `observer` bear greater resemblance to namespaces for functions than to actual notifiers and observers found in the OOD observer pattern, but they provide the opportunity to use OOD-patterns outside of the algorithms. Factory methods are redundant because of Eigen's abstractions, and structural patterns can be resolved with currying and transformations of input arguments.

12.1.7 Comparison to IPOPT

The thesis implmenetation provides a working, decentralized alternative to IPOPT's shared `IpIpoptCalculatedQuantities`, which results in less verbose, readable

code that has better resemblance to the theoretical equations.

Functional design was intended to simplify dynamically changing behaviour, but this property has been restricted by the loops in the implementation. For example, in order to switch to a restoration phase the barrier subproblem needs to obtain a state that determines if the loop in `Barrier_Subproblem::Solve` should be terminated, only to initialize a new loop in the restoration phase afterwards. Replacing this behaviour requires `Barrier_Subproblem::Solve` to be replaced, instead of modifying the transition step between two smaller functions.

Modifying/extending the behaviour of IPOPT would require the addition of `AlgorithmStrategy`-classes, which further needs to be accounted for in the the `AlgorithmBuilder` and `IpoptAlgorithm`-classes, before finally extending the cache in `IpoptCalculatedQuantities` which potentially none of the other `AlgorithmStragegy`-classes have any relation to.

In total it is possible to argue that the thesis implementation provides better encapsulation, but there are still improvements to be made with respect to the design. Fortunately, most of the components of the solver can be replaced independently, which will simplify this process.

The current setup is configured with binders to Python, automated scripts for execution and interface to CUTEst makes it easy to configure problems, obtain the convergence data and visualize the convergence. This framework in itself is a very useful outcome of the thesis that can also be used to prototype numerical algorithms in the future.

12.2 Performance

12.2.1 Inertia Correction

The inertia correction correctly increases δ_w when the primal-dual hessian starts becoming nonconvex. The exponential increase in the trial δ_w's have in some cases been correlated with convergence failure on some of the SIF-problems, which could indicate that additional parameter tuning is needed, but this has to be reviewed together with the other components of the solver. The constraint independence correction δ_c is added when eigenvalues approach 0. Overall inertia correction has improved the robustness of the algorithm, and makes it possible to find descent directions in nonconvex cases where the default KKT-system would yield ascending directions.

HS13 (Section 11.1.8) was intended to demonstrate inertia correction with the loss of the Linear Independent Constraint Qualification (LICQ) which occurs at x^*. However, the superlinear update rule results in a too rough transition when μ becomes small, which leads to an inaccurate update of inequality multipliers z_g, z_{lb}, which throws the linesearch direction of the next iterate off on the wrong course. The solver converged successfully once the superlinear update ($\mu_{j+1} = \mu_j^{\theta_\mu}$) was disabled, indicating that robustness could be improved by modifying this rule.

Inertia correction was found to be a very costly procedure when performed on large, dense matrices, which indicates that the eigenvalue-solver module in Eigen becomes a bottleneck in this case. This performance issue could be solved by resorting to a large-scale eigenvalue solver library called Spectra[42] which builds on top of Eigen, and instead compute a subset of the eigenvalues in order to estimate the inertia.

12.2.2 Second-Order Correction

In the nonlinear equality-constrained QP (section 11.1.4) the solver achieves full-step convergence both with and without second-order corrections. This is not caused by the correction itself, but by the modified switching conditions which enables the filter to accept steps that increase the contraint violation. The solver was not able to converge with the original switching parameters ($s_\theta = 1.10, s_\varphi = 2.30$) which prevented full-step acceptance.

As a consequence of the milder switching condition, full steps are accepted both with and without second-order corrections. This does not impact the convergence in this specific problem, but the non-corrected trajectory converge with slightly higher optimality values ($E_{\mu_j}(x, \lambda, z), f(x), \theta(x)$). Still, the example is able to demonstrate that second-order correction works as intended.

12.2.3 Linesearch

The linesearch is kept rather simple in this project, which makes it easy to verify that it operates correctly. Each trial step in a linesearch provide information that be used to improve the search for new trial steps. Reducing the total number of trial steps is a good approach to improving the efficiency of the solver without any other dependencies, and should be pursued in future designs.

12.2.4 Barrier Subproblem, Linesearch Filter-Barrier and the Restoration Phases

The general algorithm structure in algorithm 7 ensures that the computationally expensive operations performed in each algorithm occurs within some atomic actions inside the algorithms loop. The few expensive function calls performed in the barrier subproblem, linesearch filter-barrier and restoration phase algorithms are easy to profile. Performance profiling these higher-level algorithms show that the linesearch and inertia correction accounts for the most computationally expensive components of the solver, while the smaller pass-by-reference operations performed in these higher level algorithms are inexpensive.

The restoration phases solve KKT-systems using exactly the same routines as the linesearch filter barrier-algorithm, which leads to solving an unnecessary large linear system that could be reduced with substitution. The number

of states in the restoration objectives can grow large with the number of constraints ($N_w = N_x + 2N_g$ for inequality restoration, $N_w = N_x + 2N_h$ for equality restoration). The performance would scale better with an equivalent sparse implementation.

12.2.5 Memoization

Figure 11.29 show that the memoizers are able to reduce the solver time. This is reassuring for the functional design, and shows that decentralized memoization is able to perform well. Increasing the buffer size did not further reduce the computation time. Some improvements could be made to the buffer, which is forced to recompute values at x_k when the circular buffer gets filled with evaluations at $x_k + \alpha d_x$ during the linesearch.

12.2.6 Compile-Time

The compile-time of fixed-size matrix executables is too long for both GCC and CLANG, measured up to 20 seconds with an executable size of 98Mb for SIF-executables. Reducing the size of the produced object files could make it possible to compile a static libary for the desired problem dimensions.

The number of instantiations of functions and classes is reasonable with respect to their usage. For example, running `objdump` on the cmake-cached object file `SIF_LSFB.cpp.o` can be used to count the number of symbols for a function:

```
$ objdump -t SIF_LSFB.cpp.o | grep Eval_hessian_f | wc -l
$ 14
```

Another approach to reduce the compile-time is to limit the scope for the eigen-expressions, possibly by performing recompilations after a fixed number of loop iterations. Sequentially compiling and executing code this way could be performed with a Just-In-Time compiler (JIT), which prioritize compile-time speed over optimization.

12.3 Convergence

12.3.1 Subproblem Iterations

The subproblem iteration results for the HS-problems (figure 11.28) shows that the thesis implementation consistently requires more evaluations than IPOPT. This convergence issue is likely to either be a parameter adjustment issue related to the acceptance of steps in the filter, or it could be caused by convergence issues in either the primal states or the multipliers. Convergence towards the optimality conditions have different dependencies than in IPOPT's implementation which makes it difficult to sequentially compare the trajectories of the solvers.

The number of subproblem iterations for the majority of the converged problems are promising, and most of the problems have a significantly shorter solve time than IPOPT. This confirms that the dense solvers are useful for small-scale problems. Comparing a dense solver for small-scale problems with IPOPT can be misleading however, since IPOPT is a large-scale problem solver.

12.3.2 Restoration Phases

The inequality restoration phase show a promising convergence in the inequality-constrained quadratic program with infeasible initial point (section 11.1.2), where it is able to bring the iterates to a feasible, optimal point, terminate and return to the normal LSFB-algorithm.

Table 11.1 shows that inequality restoration phases struggle to achieve successful termination when inequality restorations are invoked. Only 4 inequality restoration phases were sufficiently close to a solution without termination, which indicates that there are issues with the transition back to the normal LSFB algorithm. Convergence failure for problems with infeasible initial points is a recurring problem in the case of the HS-dataset, which raises the question on the robustness of the inequality restoration phase. However, the convergence of HS14 (section 11.1.9) and HS17 (section 11.1.10) show that the concept of an inequality restoration phase can be useful under the right conditions.

The equality restoration phases have been invoked for some HS-problems with mixed results. Equality restoration phases were used in 13 of problems that successfully converged, and only 1 that was close. Multiplier re-initialization after restoration phases seem to operate correctly, but the converged restoration point $\bar{x}^*$ could itself be the issue. Some inspected restoration phases were observed to converge to points with high infeasibility in E_{μ_j}, indicating that there are issues related to the computation of directions $d_{\bar{x}}, d_{\bar{\lambda}}$.

Both of the restoration phases can encounter situations where there are multiple solutions to the optimal distance metric $||D_R(\bar{x}-\bar{x}_R)||_2^2$. Modifying this penalty to a different ℓ_p norm when the system becomes ill-conditioned could provide unique solutions to situations that are infeasible for ℓ_2 penalty.

The search for one unique minimizer of the restoration phase is a very strict requirement, which could be relaxed by increasing the optimality tolerance ε_R. This would throw the next iterate into some neighborhood of the actual solution $\bar{x}^*, \bar{\lambda}^*, \bar{z}^*$ which needs to be carefully scaled by ϵ_R.

12.3.3 Problem Classifications

Figure 11.30 shows that the solver is able to coverge for all problems with bounds only, and for the majority of problems with linear constraints. Quadratic and nonlinear constraints have a very large rate of failure, which could be an indication of malfunction in the constraint hessian evaluations (`::Eval_hessian_h`, `::Eval_hessian_g`) or the context in which they are used. In figure 11.31 this

rate is visible in all categories with last letter Q and O.

All problems involving bounds (XB) and linear constraints (XL) converge regardless of objective type. This confirms that both inequality and equality multipliers contribute to the search direction in a meaningful way, but also that the filter is able to reduce step lengths correctly both for nonlinear objectives, and for the nonlinear cost from the barrier function.

The usage of classification on a larger problem set has proven to be a useful method for discovering convergence issues. Specific testing on tailored problems was initially expected to be both simpler and more efficient, but problem classification are able to show trends that reveal problems from a higher level perspective.

12.3.4 Overall Convergence

In total the convergence results are not good enough to consider this to be a robust solver, and it consistenly requires more iterations than IPOPT in order to converge. Considering that the Hock-Schittkowski collection is designed to test the solver on many different sources of numerical issues, it is reassuring to see that the solver is able to converge on a large portion of them.

Chapter 13

Conclusion

The work on the solver throughout this thesis has provided insight into the challenges of implementing algorithms for numerical optimization. Despite these challenges the results brings some closure to the initial problem statement, which will be briefly discussed.

- Implement a robust solver that is able to converge on a wide range of differentiable NLPs, regardless of nonlinearity, constraints and dimensions.

In its current state the thesis implementation can not be considered robust, and it does not converge towards arbitrary differentiable NLPs. The robustness is significantly better for all linearly constrained NLPs. Additionally, this does not scale well with the dimension of the problem, which leads to the conclusion that there are many improvements to be made in order to achieve an overall robust, dimension-invariant solver.

- Implement the solver with the foundation of an efficient, linear algebra library with comparable performance to IPOPT.

Eigen has provided linear algebra and a solver that beat IPOPT in its default configuration for small problems. Considering that IPOPT is designed for large-scale problems it is still undetermined if the Eigen-based implementation will outperform IPOPT.

- Apply functional programming to avoid hidden dependencies, improve readability, enable additional optimization strategies and give the solver better capabilities to dynamically change behaviour.

Functional design combined with restrictive programming of loops have ensured that the solver does not suffer from hidden dependencies. Functional design, memoization and Eigen-abstractions improve readability both on the lower and higher levels.

Dynamic behaviour and many of the implemented optimization features are constrained to occur at compile-time using templates. Incorporating runtime information into this is an improvement to be made in the future.

The educational benefits of designing the solver has been significant, all the

way from optimization theory, language paradigms and linear algebra to C++ standards and code debugging. Considering that the objectives were ambitious, the resulting library can be considered a success with respect to the scope of this thesis, which is likely to be re-used for future projects in numerical optimization.

Chapter 14

Future Work

14.1 Resolve Convergence for Nonlinear Constrained Problems

Classification results show that the majority of unconverged problems are nonlinearly constrained. The cause of these problems is likely to be related to a minor implementation detail, but no solution was found within the scope of this thesis. These convergence issues should be fixed before refactoring the sparse module.

14.2 Improve Robustness with Additional Heuristics

There are still many heuristics presented in the IPOPT implementation paper[1] that could improve performance, but these heuristics need to be considered carefully with respect to the differences in the thesis implementation. Testing the heuristics with tailored problems and obtaining results on classified problem sets was found to be very useful.

14.3 Resolve Sparse Runtime Errors

Results from a working, sparse solver will be more comparable to the results from IPOPT, and could make it possible solve large-scale problems in an acceptable amount of time. Once the robustness of the dense module has been improved, the sparse module can be adapted to follow its design structure.

14.4 Static Libraries with Explicit Instantiation

The compile time for the classes in the fixed-size matrix dense module becomes large when modifications that affect a longer chain of header dependencies is performed. By explicitly instantiating template parameters for template classes

and functions it is possible to isolate and compile smaller components of the library as a static library, which can be afterwards linked after compiling the main source file. This is not a problem for dynamically allocated dense/sparse matrices, since they do not require problem dimensions at compile time. However, for fixed-size matrices there is currently a lot of duplicate code compiled, which should be preventable.

Code listing 14.1: Instantiation prevention and explicit instantiation

```
//Instantiation header:
template class A<int>;
template class A<double>;
template void fun<int>(int& arg);
template void fun<double>(double& arg);

//User headers:
extern template class A<int>;
extern ...
```

Listing 14.1 shows how classes and functions can be explicitly instantiated in one file, and declared as `extern` in other headers that wants to use the same binary code.

14.5 CasADI Interface

CasADI[43] is an open-source tool for optimization and algorithmmic differentiation, which serves as a great tool for optimization of OCPs. CasADI provides a symbolic framework which simplifies the process of converting system dynamics to constraints, which is very useful when the number of constraints grow large. Systems with ODE-dynamics are often solved with shooting methods, where both the order of the integrator (e.g. Runge-Kutta or collocation polynomial orders) and the number of 'trajectory segments' in multiple shooting add additional constraints to the problem. Additionally, explicit instantiation can be used to reduce the size of the compiled object files.

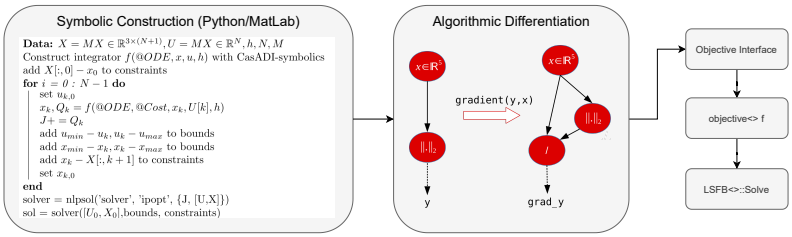


Figure 14.1: CasADI Interface[44]

CasADI simplifies this process by passing symbolic variables into integrators, resulting in sparse expression trees which can be algorithmically differentiated to obtain jacobian evaluation functions and sparsity patterns. CasADI was

initially designed for algorithmic differentiation, and its state-of-the-art implementation should compare well against the HSL AD02 [17] subroutines used in CUTEst.

14.6 JIT-compilation

The current compile-time for fixed-size matrices is long and results in large-sized binaries, which makes the compilation process unacceptable to interface with the runtime of other applications (like CasADI). Additionally, compiling the full scope of the solver before runtime prevents optimization based on runtime information.

This is also a problem in templated recursive functions, since a new function is potentially instantiated every recursive call. By setting a smaller maxium recursion depth it becomes possible to prevent stack overflow and optimize Eigen-expressions with runtime information.

Alternatively, the JIT-compiler also presents the oppurtunity to outsource the algorithms to a different language. Higher-level decisions be performed in a functional programming language could enable recursion, lazy evaluation and expression reductions to be performed during runtime.

14.7 Trust-Region Methods

Trust-region methods were studied as a potential candidate to the implemented linesearch. Trust-region methods is the dual approach to linesearch, which determines the distance for a step before the direction. A trust-region based interior-point algorithm using dogleg steps[45] is provided by the authors of the original implementation paper. Determining the search direction can often yield long steps when the local approximation is accurate, but gives no guaranteed convergence rate. Trust-region methods can (depending on the subproblem step methods) guarantee convergence under the assumption that the primal hessian Σ_P is used[13, p.590], while linesearch may fail for interior-point when subject to bounds/inequalities. [13, p.587].

14.8 Linesearch with Interpolation

The gradients and barrier objective values obtained in linesearch can be used to build a local approximation around the evaluated trial steps, using interpolation. The interpolated model becomes a minimization problem that potentially yields faster convergence to an acceptable trial step without making the step too short. An algorithm called zoom[13, p.61] use values at previous trial steps to form new step bounds and interpolations, gradually converging towards a local minimum. Quadratic/qubic interpolation models can be interchangably

used depending on the curvature, where qubic models usually yields quadratic convergence[13, p.59].

14.9 Barrier Parameter Update Rule

Exploring different barrier parameter rules should be done in order to avoid some of the numerical difficulties that were encountered in some of the HS-problems. The update rules researched for this thesis[14] are autoregressive, which results in a fixed sequence of barrier parameters. Introducing states, multipliers and optimality values would enable the barrier parameter to update according to the iterates optimality, states and multipliers[46].

Chapter 15

Installation and Build Instructions

15.1 Installation with Docker

The docker image for this project can be downloaded from the provided link[47]. A README with installation instructions is provided on the page. The docker installation is recommended over the manual installation in order to avoid path configurations and linking.

15.2 CUTEst

The SIF-executables in the project requires access to decoded SIF-files, which is attained from the CUTEst-library. The library can be installed by following the instructions provided in the librarys GitHub-repository[48]. Path configurations in CMake use environmental variables `MASTSIF` and `CUTEst` which is configured during this installation.

15.3 Pybind11

The top-level `CMakeLists.txt`-file is configured to locate an installation of the Pybind11 header-only library, which can be installed following the instructions on its documentation website [49]. Pybind11 requires access to Python header files (`python-dev/python3-dev`). The Pybind11-requirement can be removed by removing the CMakeLists-requirements, and by excluding the Plot-directory from the CMake directory hierarchy.

15.4 The Project

The project can be cloned from its GitHub-repository[50].With CMake, GCC/Clang and dependencies configured the project can be built by running `cmake ..` from the build directory. CMake will attempt to automatically locate Pybind11 and the Python header files installed.

15.5 Usage

15.5.1 Decode SIF-Objective

Custom CMake-targes have been added in `PROJECT_ROOT/Data/SIF/Problem/CMakeLists.txt` which decodes and compiles a target SIF problem when `cmake ..` is executed. The SIF problem name can be specified in `PROJECT_ROOT/CMakeLists.txt`, or during configuration:

```
cmake .. -DSIF_PROBLEM=PROBNAME
```

Manual compilation of SIF problems is described in section 7.2.

A dimension preloader-script is used to produce a header file for dimensions N_x, N_g, N_h. One of the CMake custom targets compiles and executes `Dimension_Preloader.cpp`, resulting in `PROJECT_ROOT/include/SIF_Dimensions/Dimensions.hpp`. This header is included by SIF executables, which enables them to pass N_x, N_g, N_h as compile-time parameters.

15.5.2 Configure Paths

All source files for C++ executables and Python scripts contain absolute paths which needs to be configured. Paths used in CMake are made relative to `CMAKE_PROJECT_SOURCE_DIR`.

15.5.3 Make

Running `Make` in any of the CMake-directories will build targets for it and its subdirectories.

15.5.4 Execute

Executables are available in `PROJECT_ROOT/build/test/` and plotting scripts in `PROJECT_ROOT/Plot/`.

15.5.5 Configure Layers

`PROJECT_ROOT/test/SIF/Run_SIF.cpp` demonstrates how layers can be configured for the objectives. Listing 15.1 shows the object instantiations.

Code listing 15.1: Layer configurations for SIF objectives

```
objective_SIF<Nx, Ng, Nh> f(SIF_path + "OUTSDIF.d");

objective_SIF_memoized<Nx, Ng, Nh> f(SIF_path + "OUTSDIF.d");

observer_journalist<Nx, Ng, Nh> journalist_f(journalist_ID, output_path);
objective_SIF_journalist<Nx, Ng, Nh> f(SIF_path + "OUTSDIF.d", journalist_f);
```

15.5.6 Configure Parameters

Parameters currently do not have an input-argument based configuration, but are instead configured by editing default constructors in each `_Param.hpp`-file.

15.5.7 Read Data

Data is outputted to paths configured in the executables. Default output folders are located under `PROJECT_ROOT/Data/`, which will output directory hierarchies with data from the algorithm and objective journalists.

The journalist objective in listing 15.1 will produce a folder with the journalists' ID, located under `output_path`.

```
journalist_f/
├── Eval_f_data.csv
├── Eval_f_input.csv
├── Eval_g_data.csv
├── Eval_g_input.csv
├── Eval_grad_data.csv
├── Eval_grad_g_data.csv
├── Eval_grad_g_input.csv
├── Eval_grad_h_data.csv
├── Eval_grad_h_input.csv
├── Eval_grad_input.csv
├── Eval_h_data.csv
├── Eval_h_input.csv
├── Eval_hessian_f_data.csv
├── Eval_hessian_f_input.csv
├── Eval_hessian_g_data.csv
├── Eval_hessian_g_input.csv
├── Eval_hessian_h_data.csv
└── Eval_hessian_h_input.csv
```

Figure 15.1: `objective_journalist` output directory

15.5.8 Plot Data

`Trajectory_Plot.py`, `Objective_Plot.py` and `Multiplier_Plot.py` are available under subfolders `QP/`, `SIF/` in `PROJECT_ROOT/Plot/`. In the case of SIF problems the binders needs to be recompiled in order to produce correct contour plots (The SIF-binder has the decoded SIF-problem as a dependency). Folder and system path configurations apply here aswell.

15.5.9 Automated SIF Problem Set Optimization

A set of bash-scripts are used to run lists of SIF-problems. These scripts are available under `PROJECT_ROOT/Plot/SIF/HS/`, and automate both the building process for optimization and the building of binders for plotting.

15.5.10 Doxygen Graph Generation

Doxygen is configured to generate graphs in the `PROJECT_ROOT/include/dense/` directory. The graphs are generated by running `doxygen Doxyfile` in the `PROJECT_ROOT/docs/` directory, which can be navigated by opening `PROJECT_ROOT/docs/html/index.html`.

[1] A. Wächter and L. T. Biegler, 'On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming,' *Mathematical Programming*, vol. 106, no. 1, pp. 25–57, 2006, ISSN: 1436-4646. DOI: 10.1007/s10107-004-0559-y. [Online]. Available: https://doi.org/10.1007/s10107-004-0559-y.

[2] G. Guennebaud, B. Jacob *et al.*, *Eigen v3*, http://eigen.tuxfamily.org, 2010.

[3] N. I. M. Gould, D. Orban and P. L. Toint, 'Cutest: A constrained and unconstrained testing environment with safe threads for mathematical optimization,' *Computational Optimization and Applications*, vol. 60, no. 3, pp. 545–557, Apr. 2015, ISSN: 1573-2894. DOI: 10.1007/s10589-014-9687-3. [Online]. Available: https://doi.org/10.1007/s10589-014-9687-3.

[4] P. Amestoy, A. Buttari, I. Duff, A. Guermouche, J.-Y. L'Excellent and B. Uçar, 'Mumps,' in *Encyclopedia of Parallel Computing*, D. Padua, Ed. Boston, MA: Springer US, 2011, pp. 1232–1238, ISBN: 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4_204. [Online]. Available: https://doi.org/10.1007/978-0-387-09766-4_204.

[5] C. Lattner and V. Adve, 'LLVM: A compilation framework for lifelong program analysis and transformation,' pp. 75–88, Mar. 2004.

[6] L. S. Blackford, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry *et al.*, 'An updated set of basic linear algebra subprograms (blas),' *ACM Transactions on Mathematical Software*, vol. 28, no. 2, pp. 135–151, 2002.

[7] C. Sanderson and R. Curtin, 'Armadillo: A template-based c++ library for linear algebra.,' *Journal of Open Source Software, Vol. 1, pp. 26, 2016.*,

[8] R. Poya. (2020). 'A look at the performance of expression templates in c++: Eigen vs blaze vs fastor vs armadillo vs xtensor,' [Online]. Available: https://romanpoya.medium.com/a-look-at-the-performance-of-expression-templates-in-c-eigen-vs-blaze-vs-fastor-vs-armadillo-vs-2474ed38d982.

[9] (). 'Doxygen,' [Online]. Available: https://github.com/doxygen/doxygen.

[10] J. Ellson, E. R. Gansner, E. Koutsofios, S. C. North and G. Woodhull, 'Graphviz and dynagraph – static and dynamic graph drawing tools,' pp. 127–148, 2003.

[11] (2022). 'About diagrams.net,' [Online]. Available: https://www.diagrams.net/about.

[12] D. Merkel, 'Docker: Lightweight linux containers for consistent development and deployment,' *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

[13] S. J. W. Jorge Nocedal, *Numerical Optimization*. Springer New York, 2006. DOI: 10.1007/978-0-387-40065-5. [Online]. Available: https://doi.org/10.1007%2F978-0-387-40065-5.

[14] R. H. Byrd, G. Liu and J. Nocedal, *On the local behavior of an interior point method for nonlinear programming*, 1998.

[15] R. Fletcher and S. Leyffer, 'Nonlinear programming without a penalty function,' *Mathematical Programming*, vol. 91, no. 2, pp. 239–269, 2002, ISSN: 1436-4646. DOI: 10.1007/s101070100244. [Online]. Available: https://doi.org/10.1007/s101070100244.

[16] C. Larman and E. Gamma, *Applying UML and Patterns: An Introduction to Object-oriented Analysis and Design and Iterative Development*, ser. Addison-Wesley professional computing series. Addison-Wesley, 2005, ISBN: 9783827328243. [Online]. Available: https://books.google.no/books?id=jUvf7wMUGcUC.

[17] (). 'The hsl mathematical software library,' [Online]. Available: https://www.hsl.rl.ac.uk/.

[18] (2014). 'Spral: An open-source library for sparse linear algebra.' version 2014-03-20, [Online]. Available: http://www.numerical.rl.ac.uk/spral.

[19] C. Alappat, A. Basermann, A. R. Bishop, H. Fehske, G. Hager, O. Schenk, J. Thies and G. Wellein, 'A recursive algebraic coloring technique for hardware-efficient symmetric sparse matrix-vector multiplication,' *ACM Trans. Parallel Comput.*, vol. 7, no. 3, Jun. 2020, ISSN: 2329-4949. DOI: 10.1145/3399732. [Online]. Available: https://doi.org/10.1145/3399732.

[20] M. Bollhöfer, O. Schenk, R. Janalik, S. Hamm and K. Gullapalli, 'State-of-the-art sparse direct solvers,' A. Grama and A. H. Sameh, Eds., pp. 3–33, 2020. DOI: 10.1007/978-3-030-43736-7_1. [Online]. Available: https://doi.org/10.1007/978-3-030-43736-7_1.

[21] M. "Bollhöfer, A. Eftekhari, S. Scheidegger and O. Schenk, 'Large-scale sparse inverse covariance matrix estimation,' *SIAM Journal on Scientific Computing*, vol. 41, no. 1, A380–A401, 2019. DOI: 10.1137/17M1147615. eprint: https://doi.org/10.1137/17M1147615. [Online]. Available: https://doi.org/10.1137/17M1147615.

[22] R. K. Dybvig, *The SCHEME programming language*. Mit Press, 2009.

[23] S. Marlow *et al.*, 'Haskell 2010 language report,' *Available online http://www. haskell. org/(May 2011)*, 2010.

[24] A. Church, 'A set of postulates for the foundation of logic,' *Annals of Mathematics*, vol. 33, no. 2, pp. 346–366, 1932, ISSN: 0003486X. [Online]. Available: http://www.jstor.org/stable/1968337.

[25] N. Jarus. (2016). 'C++ lambdas under the hood,' [Online]. Available: https://web.mst.edu/~nmjxv3/articles/lambdas.html.

[26] P. Ferraggi. (2020). 'Do you need design patterns in functional programming?' [Online]. Available: https://dev.to/patferraggi/do-you-need-design-patterns-in-functional-programming-370c.

[27] I. Bongartz, A. R. Conn, N. Gould and P. L. Toint, 'Cute: Constrained and unconstrained testing environment,' *ACM Trans. Math. Softw.*, vol. 21, no. 1, pp. 123–160, Mar. 1995, ISSN: 0098-3500. DOI: 10.1145/200979.201043. [Online]. Available: https://doi.org/10.1145/200979.201043.

[28] A. R. Conn, N. I. M. Gould and P. L. Toint, *Lancelot: A FORTRAN Package for Large-Scale Nonlinear Optimization (Release A)*. Berlin, Heidelberg: Springer-Verlag, 1992, ISBN: 038755470X.

[29] D. O. Andrew R. Conn Nicholas I. M. Gould and P. L. Toint, 'The sif reference report (revised version),' 2018.

[30] N. I. M. Gould, D. Orban and P. L. Toint, 'Cuter and sifdec: A constrained and unconstrained testing environment, revisited,' *ACM Trans. Math. Softw.*, vol. 29, pp. 373–394, 2003.

[31] W. Hock and K. Schittkowski, 'Test examples for nonlinear programming codes,' *Journal of Optimization Theory and Applications*, vol. 30, pp. 127–129, Jan. 1980. DOI: 10.1007/BF00934594.

[32] J. Masiulis. (2018). 'A constexpr stl style circular buffer implementation,' [Online]. Available: https://github.com/JustasMasiulis/circular_buffer.

[33] (). 'Writing functions taking eigen types as parameters,' [Online]. Available: https://eigen.tuxfamily.org/dox/TopicFunctionTakingEigenTypes.html.

[34] W. Jakob, J. Rhinelander and D. Moldovan, *Pybind11 — seamless operability between c++11 and python*, https://github.com/pybind/pybind11, 2016.

[35] (). 'Benchmark of dense decompositions,' [Online]. Available: https://eigen.tuxfamily.org/dox/group__DenseDecompositionBenchmark.html.

[36] (). 'Catalogue of dense decompositions,' [Online]. Available: https://eigen.tuxfamily.org/dox/group__TopicLinearAlgebraDecompositions.html.

[37] '3. solving triangular systems,' in *Direct Methods for Sparse Linear Systems*, pp. 27–36. DOI: 10.1137/1.9780898718881.ch3. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9780898718881.ch3. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9780898718881.ch3.

[38] J. Hjulstad, 'Fipopt convergence results for the hock-schittkowski test set,' 2022.

[39] (2007). 'Intel® math kernel library,' [Online]. Available: http://www.ehu.eus/sgi/ARCHIVOS/mklman.pdf.

[40] (2022). 'Installing ipopt,' [Online]. Available: https://coin-or.github.io/Ipopt/INSTALL.html.

[41] S. McConnell, *Code Complete, Second Edition*. USA: Microsoft Press, 2004, ISBN: 0735619670.

[42] Y. Qiu. (2021). 'Spectra c++ library for large scale eigenvalue problems,' [Online]. Available: https://spectralib.org/.

[43] J. A. E. Andersson, J. Gillis, G. Horn, J. B. Rawlings and M. Diehl, 'CasADi – A software framework for nonlinear optimization and optimal control,' *Mathematical Programming Computation*, vol. 11, no. 1, pp. 1–36, 2019. DOI: 10.1007/s12532-018-0139-4.

[44] (2018). 'Casadi,' [Online]. Available: https://web.casadi.org/.

[45] R. H. Byrd, M. E. Hribar and J. Nocedal, 'An interior point algorithm for large-scale nonlinear programming,' *SIAM Journal on Optimization*, vol. 9, no. 4, pp. 877–900, 1999. DOI: 10.1137/S1052623497325107. eprint: https://doi.org/10.1137/S1052623497325107. [Online]. Available: https://doi.org/10.1137/S1052623497325107.

[46] J. Nocedal, A. Wächter and R. Waltz, 'Adaptive barrier update strategies for nonlinear interior methods,' *SIAM Journal on Optimization*, vol. 19, pp. 1674–1693, Jan. 2009. DOI: 10.1137/060649513.

[47] (2022). 'Ttk4900-master-thesis docker image,' [Online]. Available: https://hub.docker.com/repository/docker/jonashj/ttk4900-master-thesis-fipopt.

[48] (2021). 'The constrained and unconstrained testing environment with safe threads (cutest) for optimization software,' [Online]. Available: https://github.com/ralna/CUTEst.

[49] W. Jakob. (2017). 'Installing the library,' [Online]. Available: https://pybind11.readthedocs.io/en/stable/installing.html.

[50] (2022). 'Ttk4900-master-thesis github repository,' [Online]. Available: https://github.com/jonasbhjulstad/Master_Thesis.

Appendix A

Parameters

The reader is referred to the original implementation [1] for additional details. Parameters unique to the thesis implementation are asterisk-marked ($*$).

$\delta_{w,min}$	Initial inertia lower bound	$1e^{-20}$
$\delta_{w,0}$	Initial inertia	$1e^{-4}$
$\delta_{w,max}$	Maximum inertia	$1e^{40}$
$\bar{\delta}_c$	Constraint inertia coefficient	$1e^{-8}$
κ_w^-	Inertia decrease coefficient	0.33
κ_w^+	Inertia increase coefficient	8.00
$\bar{\kappa}_w^+$	Initial inertia increase coefficient	100
κ_c	Constraint inertia increase coefficient	0.25
iter_max*	Maximum loop iterations	500
zero_tol*	Zero-eigenvalue tolerance	$1e^{-6}$

Table A.1: Inertia correction algorithm parameters

max_iter	Maximum loop iterations	4
κ_{SOC}	Reduction criterion coefficient	0.99

Table A.2: Second-order correction algorithm parameters

iter_max*	Maximum loop iterations	100
κ_Σ	Hessian safeguard coefficient	$1e^{10}$
κ_ϵ	Barrier optimality coefficient	1.00
τ_{min}	Minimum FTTB-parameter	0.99
s_{max}	Maximum optimality scaling	100
λ_{max}	Maximum λ initial value	$1e^3$

Table A.3: Barrier subproblem algorithm parameters

iter_max*	Maximum loop iterations	100
λ_{max}	Maximum λ initial value	$1e^{3}$
s_{max}	Maximum optimality scaling	100
ϵ_{tol}	Optimality error tolerance	$1e^{-8}$
ρ	Restoration phase slack-variable scaling	$1e^{3}$
κ_1, κ_2	Bounds-correction parameters	$1e^{-2}$

Table A.4: Linesearch Filter-Barrier algorithm parameters

$\bar{\Delta}_{c_I}$*	Maximum constraint optimality offset	$1e^{3}$
κ_{c_I}*	Constraint optimality offset coefficient	$1e^{-4}$

Table A.5: Inequality restoration phase algorithm parameters

Bounds for constraint violations are evaluated at initialization:

$$\theta_{min} = 1e^{-4} \max\left\{1, \theta(0)\right\} \tag{A.1}$$

$$\theta_{max} = 1e^{4} \max\left\{1, \theta(0)\right\} \tag{A.2}$$

γ_θ	Constraint violation reduction coefficient	$1e^{-5}$
γ_φ	barrier objective reduction coefficient	$1e^{-5}$
δ	Switching feasibility coefficient	1.00
γ_α	Step infeasibility coefficient	0.05
s_θ, s_φ	Switching feasibility power coefficients	$1.10, 2.30$
η	Armijo reduction coefficient	$1e^{-4}$

Table A.6: Fletcher-Leyffer filter parameters

Jonas Hjulstad

Interior-Point Methods in Numerical Optimization

NTNU
Norwegian University of
Science and Technology