

Exploring novel designs of NLP solvers

Architecture and Implementation of WORHP

by

Dennis Wassel

Thesis submitted to University of Bremen
for the degree of Dr.-Ing.

March 2013.

Date of Defense: 25.04.2013

1st referee: Prof. Dr. Christof Büskens

2nd referee: Prof. Dr. Matthias Gerdts

3rd examiner: Prof. Dr. Angelika Bunse-Gerstner

4th examiner: Dr. Matthias Knauer

Contents

List of Listings	v
List of Figures	vii
Foreword	ix
1. Nonlinear Programming	1
1.1. Mathematical Foundations	2
1.2. Penalty and barrier methods	7
1.3. Sequential Quadratic Programming	9
1.3.1. Derivative Approximations	11
1.3.2. Optimality and Termination Criteria	18
1.3.3. Hessian Regularization	22
1.3.4. Prepare the Quadratic Problem	23
1.3.5. Determine Step Size	26
1.3.6. Recovery Strategies	30
2. Architecture of WORHP	31
2.1. Practical Problem Formulation	33
2.2. Sparse Matrices	35
2.2.1. Coordinate Storage format	35
2.2.2. Compressed Column format	36
2.3. Data Housekeeping	38
2.3.1. The traditional <i>many arguments</i> convention	39
2.3.2. The USI approach in WORHP	40
2.4. Reverse Communication	43
2.4.1. Division into <i>stages</i>	44
2.4.2. Implementation considerations	46
2.4.3. Applications	48
2.5. Serialization	49
2.5.1. Hotstart functionality	49

2.5.2.	Reading parameters	49
2.5.3.	Serialization format	50
3.	Technical Implementation	53
3.1.	Hybrid implementation	55
3.1.1.	Interoperability issues	55
3.1.2.	Interoperability improvements	59
3.1.3.	Data structure handling in WORHP	63
3.2.	Automatic code generation	68
3.2.1.	Data structure definition	69
3.2.2.	Definition file	70
3.2.3.	Template files	72
3.2.4.	Applications of automatic code generation	73
3.3.	Interactive mode	76
3.4.	Workspace Management	79
3.4.1.	Automatic workspace management	79
3.4.2.	Dedicated dynamic memory	81
3.5.	Internal XML parser	85
3.6.	Established shortcomings, bugs and workarounds	87
3.6.1.	Intel Visual Fortran 10.1	87
3.6.2.	Windows	87
3.7.	Maintaining compatibility across versions	89
4.	Solver Infrastructure	91
4.1.	Configuration and Build system	92
4.1.1.	Available build tools	92
4.1.2.	Basics of <code>make</code>	94
4.1.3.	Directory structure	95
4.1.4.	Pitfalls of recursive <code>make</code>	96
4.1.5.	Non-recursive <code>make</code>	99
4.2.	Version and configuration info	104
4.3.	Testing Approach	107
4.4.	Testing Infrastructure	110
4.4.1.	Parallel testing script	110
4.4.2.	Solver output processing	113
4.5.	Parameter Tuning	116
4.5.1.	Why tune solver parameters?	116
4.5.2.	Sweeping the Parameter Space	117
4.5.3.	Examples	120
4.5.4.	Conclusions	127
4.6.	Future directions for testing	128
A.	Examples	131
A.1.	Bypassing const-ness	131

List of terms	133
List of acronyms	141
Bibliography	143

List of Listings

2.1. Interface of <code>sn0ptA</code> (the advanced SNOPT interface)	38
2.2. User action query and reset	48
3.1. Example of a non-standard Fortran data structure	55
3.2. Example of a C-interoperable Fortran type	60
3.3. C <code>struct</code> corresponding to the Fortran type in listing 3.2.	61
3.4. Snippet of the data structure definition file	70
3.5. Help output of interactive mode	77
3.6. Detail help output of interactive mode	78
3.7. Data hiding in plain C through incomplete types.	90
4.1. Recursive build sequence (clean rebuild)	97
4.2. Recursive build cascade due to newer file system timestamps	98
4.3. Strongly abridged version of <code>build.mk</code> of the core subdirectory	100
4.4. Slightly abridged version of <code>modules.mk</code> of the core subdirectory	100
4.5. Abridged machine configuration file	102
4.6. Abridged build configuration file	103
4.7. Output of <code>eglibc</code> on Ubuntu 12.04	105
4.8. Sample output of shared <code>libworhp</code> on Linux platforms	106
4.9. Summary for a run of the Hock/Schittkowski test set	110
4.10. Complete output of the run script for a 10-problem list	115
4.11. Example parameter combination input file for the sweep script.	118
4.12. Example with four heuristic sweep merit values	120
A.1. Example for bypassing const-ness and call-by-value.	131
A.2. Example for bypassing const-ness and call-by-reference.	132

List of Figures

1.1.	General principle of derivative-based minimization algorithms	9
1.2.	Schematic view of WORHP's implementation of the SQP algorithm	11
1.3.	Three examples for variable grouping	14
1.4.	Induced graphs for the examples in figure 1.3	15
1.5.	Concept of SBFGS	18
1.6.	Schematic view of the Armijo rule	27
1.7.	Schematic view of the Wolfe-Powell rule	28
1.8.	Schematic view of the filter method	29
2.1.	Schematic view of Direct vs. Reverse Communication.	43
2.2.	Graphic representation of WORHP's major stages	45
3.1.	Schematic of data structure memory layout	58
3.2.	Hierarchy of the serialization module	75
3.3.	Chunk-wise operation of the XML parser	86
4.1.	Fortran module hierarchy in WORHP.	101
4.2.	Size distributions of the (AMPL-)CUTer and COPS 3.0 test sets.	108
4.3.	Categories of the (AMPL-)CUTer test set.	109
4.4.	Master-slave communication pattern of the testing script	111
4.5.	Graphs generated by gnuplot for a single-parameter sweep	119
4.6.	Parameter sweep results for <code>RelaxMaxPen</code>	121
4.7.	Dispersion of utime results for <code>RelaxMaxPen</code> = 10^7 sweep	122
4.8.	Parameter sweep results for <code>ScaleFacObj</code>	126
A.1.	The world's first computer bug	135

Foreword

This doctoral thesis and the process of learning, research and exploration leading to it constitute a substantial achievement, which is only made possible by the support of a number of wonderful people:

First and foremost my parents, whose unconditional support for my wish to learn enabled me to follow the path that eventually lead me here.

Said path was opened to me, and on occasion creatively obstructed, diverted, and extended by my supervisor Prof. Dr. Christof Büskens. By challenging my design decisions, he forced me to produce sound justification, and by putting seemingly outlandish demands on WORHP’s capabilities kept my colleagues and me on our toes to continue pushing the mathematical and technical boundaries. He has my gratitude for being a highly approachable and caring tutor far beyond the minimum required of a supervisor.

As unofficial co-supervisor and official second referee, Prof. Dr. Matthias Gerdt has my gratitude for his continued, unwavering and amicable efforts to teach me the theoretical foundations of our craft and for helping me iron out a number of mathematical inaccuracies in chapter 1.

Furthermore, I want to thank my wife Mehlike for trying to have me focus on writing up, instead of pursuing more interesting side-projects, and for introducing Minnoş into our house—having a purring cat lying in one’s lap is surprisingly relaxing while writing.

My present or former colleagues Bodo Blume, Patrik Kalmbach, Matthias Knauer, Martin Kunkel, Tim Nikolayzik, Hanne Tiesler, Jan Tietjen, Jan Vogelsang, and Florian Wolff deserve further credit for giving feedback on WORHP, often coming up with challenging requirements and suggestions on its functionality, and for uncomplainingly suffering the occasional stumbling block along its development path.

Finally, I am indebted to Astrid Buschmann-Göbels for her professional and competent editing to iron out weak formulations and grammatical errors in the manuscript; the proper use of (American) English grammar—especially punctuation—and “strong” adjectives are her credit, whereas any remaining errors are mine alone.

Besides the contributions of these individuals, the development of WORHP was only possible through generous funding from the German Federal Ministry of Economics and Technology (grants 50 JR 0688 and 50 RL 0722), the European Aerospace Agency’s TEC-EC division (GSTP-4 G603-45EC and GSTP-5 G517-045EC), and the University of Bremen.

Nonlinear Programming

Nothing at all takes place in the universe in which some rule of maximum or minimum does not appear.

(Leonhard Euler)

1.1. Mathematical Foundations	2
1.2. Penalty and barrier methods	7
1.3. Sequential Quadratic Programming	9
1.3.1. Derivative Approximations	11
1.3.2. Optimality and Termination Criteria	18
1.3.3. Hessian Regularization	22
1.3.4. Prepare the Quadratic Problem	23
1.3.5. Determine Step Size	26
1.3.6. Recovery Strategies	30

This chapter is intended to give a concise introduction into the key mathematical and methodological concepts of nonlinear optimization, and an overview of Sequential Quadratic Programming (SQP) by reference to its implementation in WORHP. It is neither meant to be complete nor exhaustive, but instead to provide a broad overview of important concepts, to establish conventions, to provide points of reference, and to motivate the considerations laid out in the following chapters.

The mathematical problem formulation loosely follows the conventions of Geiger and Kanzow in [27], while the “spirit” of approaching numerical optimization from a pragmatic, problem-driven perspective is more apparent in Gill, Murray and Wright [29], although they use different (but equivalent) notational and mathematical conventions.

1.1. Mathematical Foundations

One possible standard formulation for constrained nonlinear optimization problems is

$$\begin{array}{ll}
 \min_{x \in \mathbb{R}^n} & f(x) \\
 \text{subject to} & g(x) \leq 0 \\
 & h(x) = 0
 \end{array}
 \quad (\text{NLP})$$

with problem-specific functions

$$\begin{aligned}
 f &: \mathbb{R}^n \rightarrow \mathbb{R}, \\
 g &: \mathbb{R}^n \rightarrow \mathbb{R}^{m_1}, \\
 h &: \mathbb{R}^n \rightarrow \mathbb{R}^{m_2},
 \end{aligned}$$

all of which should be twice continuously differentiable. If neither g nor h are present, the problem is called *unconstrained* and usually solved numerically using Newton's method or the Nelder-Mead simplex algorithm (cf. [26]), depending on the problem size, the smoothness of f and the availability of ∇f and $\nabla^2 f$.

The **objective function** f , the **inequality constraints** g and the **equality constraints** h may be linear, quadratic or generally nonlinear. No convexity is required by the general problem formulation, although convex problems have the attractive general property that a local minimum is also a global one. In the strictly convex case, this minimum is unique, whereas general non-convex problems may have any number of local minima; $f(x) = \sin(x)$ illustrates this tangibly.

The problem formulation (NLP) may be (ab)used to *maximize* an objective function by minimizing the objective $f^* := -f$.

The following concepts and classifications with respect to the constraints will accompany us through all considerations on NLP methods:

Box constraints: Constraints on the optimization variables x , i.e. constraints of the form $g(x) = \pm x_j - c$ or $h(x) = x_j - c$ with a constant $c \in \mathbb{R}$. These are sometimes also referred to as **Simple Bounds**.

Theoretically speaking, equality box constraints are an indication of flawed modeling, since the respective variable is a constant; not so in practice, where fixing certain variables is sometimes very convenient to simplify the implementation of the model, for instance to accommodate fixed initial or final states in discretized optimal control problems.

Active constraints: An inequality constraint $g_i(x) \leq 0$ is called *active* (at the point x), if $g_i(x) = 0$. Equality constraints are always active, so this distinction is meaningful for inequality constraints only. The concept of being *active* is applied

to box constraints in complete analogy; to differentiate, one may refer to them as *active box constraints*.

It is sometimes convenient to define the **Active Set** I as the index set of all active constraints, like $I(x) := \{i \in \{1, \dots, m_1\} \mid g_i(x) \text{ is active}\}$.

Inactive constraints: An inequality constraint $g_i(x) \leq 0$ is called *inactive* (at the point x), if $g_i(x) < 0$, or equivalently, if it is not active.

Feasible set: The *feasible set* is defined as the set of all points x , where both inequality and equality constraints are satisfied, i.e. the set

$$\{x \in \mathbb{R}^n \mid g(x) \leq 0 \text{ and } h(x) = 0\}.$$

The feasible set may also be empty, in which case problem (NLP) is, for all intents and purposes, ill-posed.

The NLP problem class encompasses various other classes of optimization problems, for instance *quadratic problems* of the form

$$\begin{aligned} \min_{x \in \mathbb{R}^n} \quad & \frac{1}{2}x^\top Qx + r^\top x \\ \text{subject to} \quad & Ax \leq b \\ & Cx = d \end{aligned}$$

with matrices Q, A and C and vectors b, c, d and r of appropriate dimensions. The quadratic problem is convex if $Q \geq 0$; we will see later that this case is of great relevance to SQP methods, because their concept is to locally approximate (NLP) by quadratic problems with (essentially) $Q > 0$, since solutions to this problem are unique (in the strictly convex case) and efficient numerical methods to find them are known.

Lagrange Function & Multipliers

Constrained optimization relies heavily on the *Lagrange function*, which is defined in terms of (NLP) as

$$L(x, \lambda, \mu) := f(x) + \lambda^\top g(x) + \mu^\top h(x),$$

where $\lambda \in \mathbb{R}^{m_1}$ and $\mu \in \mathbb{R}^{m_2}$, are called the *Lagrange multipliers* (or *dual variables* – the optimization variables x are referred to as *primal variables* in this context). Other sources define L with different signs, or impose inequality constraints as $g(x) \geq 0$; these conventions result in various subsequent sign changes, but luckily all conceivable permutations of signs yield equivalent formulations of the same problem—with the exception of f , where a sign change turns minimization into maximization problems and vice versa.

The dual variables are often underappreciated, since the primal variables are at the center of interest to the underlying optimization problem, and instances where the multipliers hold a tangible meaning are uncommon; economic models are a notable exception to

1. Nonlinear Programming

this rule: Here the multipliers are referred to as *accounting prices* or *opportunity costs*, since they represent lost benefits due to constraints. Results from parametric sensitivity analysis[10, 11] generalize this widely known fact, asserting that the Lagrange multipliers in a local optimum describe the sensitivity of the optimal objective function value to perturbations of the corresponding constraint.

More precisely: Suppose we replace $g_i(x) \leq 0$ in (NLP) by $g_i(x) \leq \varepsilon$ and solve the perturbed problem to obtain the optimal solution x_ε^* . If g_i is active at x_ε^* and additional smoothness and regularity assumptions hold, then

$$f(x_\varepsilon^*) = f(x_0^*) - \lambda_i \cdot \varepsilon + o(\varepsilon^2) \quad \text{for “sufficiently small” } \varepsilon,$$

i.e. in terms of sensitivity differentials $\lambda_i = -\frac{\partial f}{\partial \varepsilon}(x_0^*)$.

Conditions for optimal solutions

Definition 1 (KKT conditions). A point $(x^*, \lambda^*, \mu^*) \in \mathbb{R}^n \times \mathbb{R}^{m_1} \times \mathbb{R}^{m_2}$ satisfies the Karush-Kuhn-Tucker (KKT) conditions, iff

- | | | |
|-----|--|---|
| (a) | $\nabla_x L(x^*, \lambda^*, \mu^*) = 0,$ | Optimality |
| (b) | $g_i(x^*) \leq 0,$
$h_j(x^*) = 0,$ | $i = 1, \dots, m_1,$
$j = 1, \dots, m_2,$
Feasibility |
| (c) | $\lambda_i^* g_i(x^*) = 0,$ | $i = 1, \dots, m_1,$
Complementarity |
| (d) | $\lambda_i^* \geq 0,$ | $i = 1, \dots, m_1,$ |

and is called *KKT point* in this case.

If the problem is unconstrained, the KKT conditions degenerate to the universally known necessary condition for local minima $\nabla f(x^*) = 0$. Likewise, the Karush-Kuhn-Tucker conditions constitute the first-order *necessary* optimality conditions for (NLP), if the constraints at x^* satisfy certain regularity assumptions (so-called *constraint qualifications*). One of various formulations is the following:

Theorem 2 (First-order necessary conditions under LICQ).

If x^* is a local minimum of (NLP) such that the gradients of all active constraints

$$\nabla g_i(x^*), \quad i \in I(x^*) \quad \text{and} \quad \nabla h_j(x^*), \quad j = 1, \dots, m_2$$

are pair-wise linearly independent (i.e. they satisfy the *linear independence constraint qualification* LICQ), then there exist uniquely determined multipliers λ^* and μ^* such that (x^*, λ^*, μ^*) satisfies the KKT conditions.

Proof. Proofs of this fundamental theorem, or variations thereof, can be found in any textbook on constrained optimization, for instance [27, Satz 2.41]. \square

Similar assertions can be made under weaker assumptions, involving bundles of directions called *tangential cones*, but the LICQ are among the most concise and intuitive ones. It is the central statement of Theorem 2 which justifies that NLP methods are designed to find KKT points.

For *sufficient* optimality conditions, we can again find inspiration in the unconstrained case, where the Hessian matrix $\nabla^2 f(x^*)$ has to be positive definite, i.e. $d^\top \nabla^2 f(x^*) d > 0$ for all $d \neq 0$; this essentially translates to the constrained case, where the subspace of “directions” on which the Hessian matrix $\nabla_{xx}^2 L$ has to be positive definite are further qualified:

Theorem 3 (Sufficient conditions).

Given the primal variables x^* of a KKT point of (NLP), we define

$$A(x^*) := \{d \in \mathbb{R}^n \mid \nabla g_i(x^*)^\top d \leq 0, \ i \in I(x^*)\} \cap \ker \nabla h(x^*).$$

If the Hessian has positive curvature, i.e. $d^\top \nabla_{xx}^2 L(x^*, \lambda^*, \mu^*) d > 0$ for all non-zero “directions” $d \in A$, then x^* is a strict local minimum of (NLP).

Proof. See [27, Satz 2.55], noting that in general $A(x^*) \supsetneq \mathcal{T}_2(x^*)$, hence Theorem 3 is slightly more restrictive, but, by dispensing with tangential cones, is simpler to formulate. \square

The crux of Theorem 3 is the fact that it is expensive to validate numerically, if the problem is large. For this reason, it is uncommon for large-scale NLP methods to check second-order conditions (necessary or sufficient) at all, silently accepting that KKT points may also be saddle points or even local maxima.

Alternative optimality conditions

A generalization of KKT points, **Fritz-John** points enable us to cover additional problems, where LICQ is violated. We can reformulate criterion (a) from definition 1 as

$$\nabla f(x) = -\lambda^\top \nabla g(x) - \mu^\top \nabla h(x).$$

If g and h satisfy LICQ, this ensures existence and uniqueness of the multipliers. For a Fritz-John point, this requirement is relaxed, instead considering

$$(FJ) \quad \lambda_0 \nabla f(x) = -\lambda^\top \nabla g(x) - \mu^\top \nabla h(x),$$

with an additional multiplier $\lambda_0 \geq 0$.

The case $\lambda_0 > 0$ is equivalent to the KKT conditions, with multipliers $\tilde{\lambda} = \frac{\lambda}{\lambda_0}$ and $\tilde{\mu} = \frac{\mu}{\lambda_0}$, which explains why λ_0 is normalized to 1 and therefore omitted.

The interesting case is $\lambda_0 = 0$ and $(\lambda, \mu) \neq 0$, where both conditions are *not* equivalent.

Example 4 (Problem 13 from [32]). The optimum of

$$\begin{aligned}
 \min_{x,y} \quad & (x-2)^2 + y^2 \\
 \text{(HS13)} \quad & \text{subject to} \quad x, y \geq 0 \\
 & (1-x)^3 \geq y
 \end{aligned}$$

is $(x, y) = (1, 0)$. Writing out (FJ) for $(x, y) = (1, 0)$ yields

$$\lambda_0 \begin{pmatrix} -2 & 0 \end{pmatrix} = - \begin{pmatrix} \lambda_1 & \lambda_2 & \lambda_3 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & -1 \end{pmatrix},$$

which has non-trivial solutions $(\lambda_0, \dots, \lambda_3) = (0, 0, c, c)$ with arbitrary $c \in \mathbb{R}$, hence the optimum is not a KKT, but a Fritz-John point.

Mainstream NLP solvers are designed to find KKT points, i.e. points where $\lambda_0 \neq 0$, and therefore struggle with Fritz-John points, where the implicit normalization using $\tilde{\lambda} = \frac{\lambda}{\lambda_0}$ leads to “exploding” multipliers. WORHP is designed to find KKT points, but employs a heuristic to detect Fritz-John points, whose main criterion is to watch for overly large multipliers.

1.2. Penalty and barrier methods

Penalty methods are historically the oldest approach to constrained nonlinear optimization, and nowadays their use is limited to being employed as *merit functions* to determine a suitable step size (see section 1.3.5). The concept of penalty methods is to transform a constrained minimization problem into an unconstrained one, for which classical approaches like Newton's method can be used.

Penalty functions are extensions of the objective f to the constrained case, and as such have the property that smaller values are better. In general, they take the form

$$(1.1) \quad \Phi(x; \eta) = f(x) + \Psi(g(x), h(x); \eta)$$

with a *penalty term* $\Psi: \mathbb{R}^{m_1+m_2} \times \mathbb{R}^p \rightarrow \mathbb{R}$ and *penalty parameters* $\eta \in \mathbb{R}^p$. More general forms exist, but all established penalty functions fall into this category.

The penalty term Ψ is a scalar measure of feasibility, attaining smaller values the “closer” x is to a feasible point. Penalty terms are often constructed to be differentiable, at least almost everywhere, since this makes them accessible for derivative-based minimization methods. The penalty parameters η are updated iteratively during minimization: Increasing them forces the unconstrained minimization method towards a feasible point, while decreasing allows for f to attain lower, i.e. better, values.

A *penalty method* is a combination of a penalty term Ψ and an update strategy for the penalty parameters η . Both are chosen to strike a balance between the objective and feasibility, which is the key (and difficult) concept of penalty methods.

Penalty methods are classified, depending on their relationship with the original constrained problem: If local minima of (1.1) coincide with the local minima of (NLP), the penalty method is called *exact*, otherwise it is *inexact*. Both classes have handicaps:

- Exact penalty functions cannot be differentiable in the optimum, if they are of the form of the form $\Phi = f + \Psi$ (other forms exist). This requires modifications to the unconstrained minimization algorithm, as performed, for instance, in the nonsmooth Newton method (which, despite its name, is not applicable to penalty functions).
- Inexact penalty methods only return approximate solutions to rather low precisions. To achieve higher precisions, great penalty parameters are needed, which substantially affect the condition number of the unconstrained minimization problem.

Furthermore, penalty functions may be unbounded below, i.e. even a global minimum of the constrained problem may be transformed into a truly local minimum of the penalty; this can cause significant problems, if the minimization method has no safeguards against unbounded problems.

By design, penalty methods produce infeasible iterates. For this reason, their penalty functions are sometimes referred to as *exterior* penalty functions. This property can be problematic, if the underlying problem is only defined on feasible points, in which case *barrier methods* provide a viable alternative. These were popular in the 1960s

1. Nonlinear Programming

(cf. for instance [23] from 1955) but lost interest with the advent of newer methods like SQP, which were regarded as more efficient. Shortly after Karmarkar published his polynomial-complexity algorithm [42] for linear programming in 1984, its similarity to the seemingly obsolete barrier methods was discovered, and they became a key element of the interior-point methods (cf. the overview article [21] for an account).

In contrast to the (exterior) penalty methods, whose iterates converge to points in the feasible set from the outside, barrier methods approach the boundary of the feasible set from the inside. Their name is derived from the “barrier” that is erected on the boundary of the feasible set to keep the iterates inside; for this reason, barrier methods are also referred to as *interior* penalty methods. Applied to problem (NLP) with $m_2 = 0$, the (logarithmic) barrier function is

$$B(x; \tau) = f(x) - \tau \sum_{i=1}^{m_1} \log(-g_i(x)),$$

where $\tau > 0$ is called *barrier parameter*. Other barrier terms, like $\sum_i (-g_i(x))^{-1}$ have also been considered; their common property is to penalize points close to the boundary of the feasible set, i.e. where $g_i(x) \rightarrow 0^-$. By letting $\tau \rightarrow 0^+$ during iterative minimization of B , we allow x to approach this boundary. It can in fact be shown, under mild assumptions, that $\lim_{\tau \rightarrow 0^+} x^*(\tau) = x^*$.

Quite similar to the exterior penalty methods, barrier methods suffer from inherent ill-conditioning as $\tau \rightarrow 0^+$. Furthermore, the line search of the superordinate unconstrained minimization algorithm must be aware of the problem structure, to restrict evaluations of B to feasible trial points x^+ such that $g_i(x^+) < 0$, or otherwise modify B to cope with infeasible trial points.

1.3. Sequential Quadratic Programming

Building on the concise theoretical foundations in section 1.1, this part introduces Sequential Quadratic Programming as a prominent and successful method for the numerical solution of (NLP).

SQP methods for general nonlinear problems were first considered as early as 1976 by Han [31], by generalizing results from the PhD thesis [76] of Wilson in 1963, who had considered a special case. Despite their advanced age of 50 years, SQP methods are still used widely, for instance in KNITRO (to solve barrier problems), NPSOL, SNOPT, NLPQLP, or the MATLAB optimization toolbox—and, of course, in WORHP, being the youngest solver of the above enumeration. SQP uses derivatives to iteratively minimize local approximations of the nonlinear functions, a general principle shared by most derivative-based descent methods for nonlinear problems, sketched in figure 1.1.

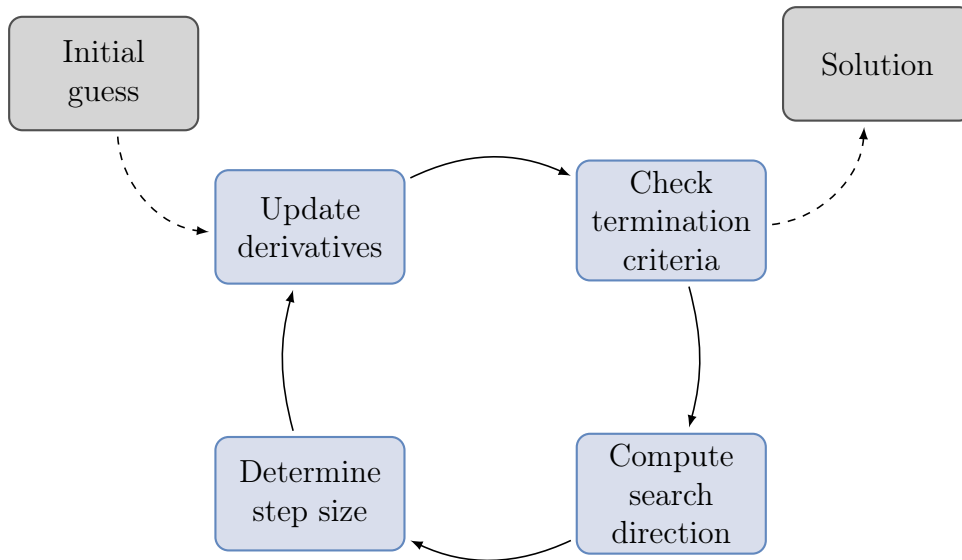


Figure 1.1.: General principle of derivative-based minimization algorithms: The choice and computation of the search direction is the linchpin of every method.

Computing the Search Direction

The fundamental idea of SQP is to find KKT points by applying Newton’s method, giving it good (local) convergence properties, but entail the use of second derivatives: To motivate the approach, first consider (NLP) without inequality constraints g , leaving us with the equality-constrained, nonlinear problem

$$\begin{aligned}
 (\text{NLP}_{\text{eq}}) \quad & \min_{x \in \mathbb{R}^n} f(x) \\
 & \text{subject to } h(x) = 0.
 \end{aligned}$$

1. Nonlinear Programming

Its KKT points can very concisely be characterized as points (x, μ) that satisfy the nonlinear equation

$$\Phi(x, \mu) = \begin{pmatrix} \nabla_x L(x, \mu) \\ h(x) \end{pmatrix} = 0$$

Assuming that both f and h are twice continuously differentiable, we can apply Newton's method to derive the iteration scheme

$$(x^{[k+1]}, \mu^{[k+1]}) = (x^{[k]}, \mu^{[k]}) - \Phi'(x^{[k]}, \mu^{[k]})^{-1} \Phi(x^{[k]}, \mu^{[k]}).$$

By setting $(\Delta x, \Delta \mu) := (x^{[k+1]} - x^{[k]}, \mu^{[k+1]} - \mu^{[k]})$, we can rearrange the Newton iteration above to an iterative solution of the linear equation system

$$\Phi'(x, \mu) \begin{pmatrix} \Delta x \\ \Delta \mu \end{pmatrix} = -\Phi(x, \mu).$$

We can spell this out, using $\Phi' = \begin{pmatrix} \nabla_{xx}^2 L & \nabla h^\top \\ \nabla h & 0 \end{pmatrix}$, to result in

$$\begin{aligned} \nabla_{xx}^2 L(x, \mu) \Delta x + \nabla h(x)^\top \Delta \mu &= -\nabla_x L(x, \mu) \\ \nabla h(x) \Delta x &= -h(x), \end{aligned}$$

which we can simplify by defining new multipliers $\mu^+ := \mu + \Delta \mu$ to

$$\begin{aligned} \nabla_{xx}^2 L(x, \mu) \Delta x + \nabla h(x)^\top \mu^+ &= -\nabla f(x) \\ \nabla h(x) \Delta x &= -h(x). \end{aligned} \quad (\text{KKT}_{\text{eq}})$$

Applying some “reverse engineering”, we notice that (KKT_{eq}) are the KKT conditions of the equality-constrained quadratic problem

$$\begin{aligned} \min_{\Delta x \in \mathbb{R}^n} \quad & \frac{1}{2} \Delta x^\top \nabla_{xx}^2 L(x, \mu) \Delta x + \nabla f(x)^\top \Delta x \\ \text{subject to} \quad & h(x) + \nabla h(x) \Delta x = 0, \end{aligned} \quad (\text{QP}_{\text{eq}})$$

noting that the optimization variables are Δx , with x and μ being constant for (QP_{eq}) . This derivation for *equality-constrained* problems motivates the addition of inequality constraints in the same manner, which brings us to the quadratic subproblem of the SQP approach

$$\begin{aligned} \min_{d \in \mathbb{R}^n} \quad & \frac{1}{2} d^\top \nabla_{xx}^2 L(x, \lambda, \mu) d + \nabla f(x)^\top d \\ \text{subject to} \quad & g(x) + \nabla g(x) d \leq 0, \\ & h(x) + \nabla h(x) d = 0. \end{aligned} \quad (\text{QP})$$

Since it is based on Newton's method, SQP inherits its major properties: Local quadratic convergence, noticeable dependence on the initial guess, and the need to perform line search to foster global convergence.

The SQP method revolves around the quadratic problem and provides various extensions over the generic descent algorithm in figure 1.1. These are sketched in figure 1.2 and laid out in more detail in the following paragraphs, which blend aspects of generic SQP methods on the one hand, and distinctive features of WORHP on the other.

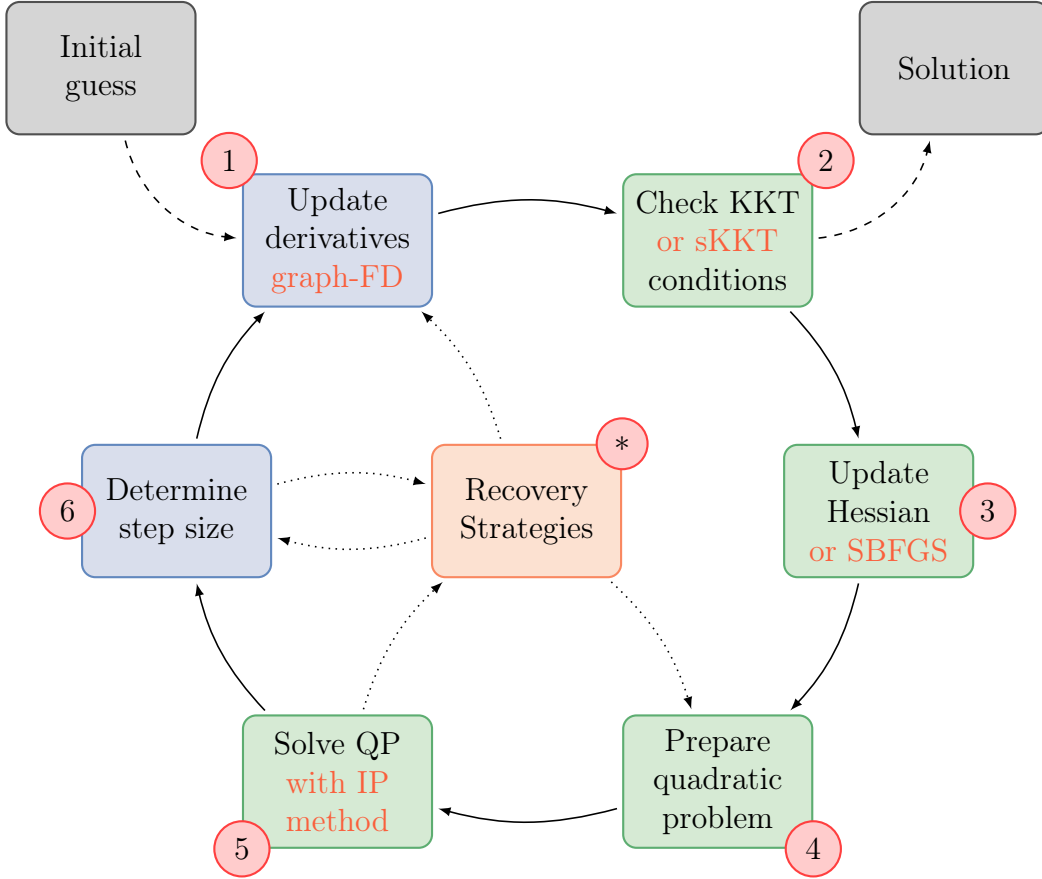


Figure 1.2.: Schematic view of WORHP's implementation of the SQP algorithm. SQP extensions over figure 1.1 are highlighted in green, whereas WORHP-specific elements are shown in orange. The recovery strategies are only activated if necessary.

1.3.1. Derivative Approximations 1

In addition to the above assertion, that the search direction is the linchpin of minimization algorithms, efficient and precise evaluation of the derivatives is the second cornerstone. This is step 1 of figure 1.1.

In virtually all cases, the most efficient and precise approach is to calculate the first and second derivatives of f, g and h using pen and paper or computer-algebra systems, and to write or generate code that evaluates them for given x . Unfortunately, this approach is generally confined to academic or textbook problems and not applicable to most real-world optimization problems; derivative approximations are the remedy, but

1. Nonlinear Programming

need further refinement before they become usable, since WORHP is designed as a *sparse* NLP solver, suitable for sparse high-dimensional optimization problems with thousands or millions of variables and constraints.

Definition 5 (Sparsity). A matrix $A = (a_{ij}) \in \mathbb{R}^{m \times n}$ is called **sparse**, if $a_{ij} = 0$ for “many” pairs (i, j) .

Applied to a differentiable function $f = (f_1, \dots, f_m): \mathbb{R}^n \rightarrow \mathbb{R}^m$, **sparsity** is the structural property of its Jacobian matrix ∇f that

$$(\nabla f)_{ij} = \frac{\partial f_i}{\partial x_j} \equiv 0$$

for “many” pairs (i, j) . These entries are called **structural zeros**, whereas those entries, where the derivative does *not* vanish everywhere, are called **nonzero entries**. Note that both properties only depend on the structure of f , not on the point, where ∇f is evaluated; as such, nonzero entries may well attain 0 as numerical value. The number of nonzero entries is commonly denoted as n_{nz} .

The term sparsity is also used to describe the ratio $\frac{n_{nz}}{n \cdot m} \in (0, 1]$. Matrices or derivatives with sparsity 1 are called **dense**, whereas ratios close to 0 are called (very) **sparse**.

Example 6. Consider the functions and first derivatives

$$\begin{aligned} f(x) &= 1 + x_2 + \frac{1}{3}x_3^3, & g(x) &= \begin{pmatrix} 4x_1x_2 + 5 \\ 3x_1^2 + 8x_3 \\ 10x_3 + 12 \end{pmatrix} \\ \nabla f(x) &= (0, 1, x_3^2), & \nabla g(x) &= \begin{pmatrix} 4x_2 & 4x_1 & 0 \\ 6x_1 & 0 & 8 \\ 0 & 0 & 10 \end{pmatrix}. \end{aligned}$$

Their derivatives have the inherent sparsity structures

$$\nabla f(x) = \begin{pmatrix} \times & \times & \times \end{pmatrix}, \quad \nabla g(x) = \begin{pmatrix} \times & \times & \\ \times & & \times \\ & & \times \end{pmatrix}.$$

where ‘ \times ’ is used to denote a nonzero entry, independent of x or the actual numeric values.

The Lagrange function for this problem is

$$L(x, \lambda) = 1 + x^2 + \frac{1}{3}x_3^3 + \lambda_1(4x_1x_2 + 5) + \lambda_2(3x_1^2 + 8x_3) + \lambda_3(10x_3 + 12)$$

with Hessian matrix and its sparsity structure

$$\nabla_{xx}^2 L(x, \lambda) = \begin{pmatrix} 6\lambda_2 & 4\lambda_1 & 0 \\ 4\lambda_1 & 0 & 0 \\ 0 & 0 & 2x_3 \end{pmatrix} \quad \nabla_{xx}^2 L(x, \lambda) = \begin{pmatrix} \times & \times & \\ \times & & \\ & & \times \end{pmatrix}$$

□

Finite differences

Derivatives of any order can be approximated by the universally known finite difference (FD) approach, although its inherent ill-conditioning, together with finite precision arithmetic imposes tight limits on achievable derivative order and precision.

To compute derivatives, WORHP has a finite difference module that employs the standard methods

(first-order) forward / backward differences

$$\frac{\partial f}{\partial x_i}(x) = \frac{f(x \pm h \cdot e_i) - f(x)}{h} + o(|h|),$$

and (first-order) central differences

$$\frac{\partial f}{\partial x_i}(x) = \frac{f(x + h \cdot e_i) - f(x - h \cdot e_i)}{2h} + o(h^2)$$

to compute first (partial) derivatives, and also second (partial) derivatives using, for instance, **second-order forward differences**

$$\frac{\partial^2 f}{\partial x_i \partial x_j}(x) = \frac{f(x + h \cdot (e_i + e_j)) - f(x + h \cdot e_i) - f(x + h \cdot e_j) + f(x)}{h^2} + o(h)$$

Gill, Murray and Wright [29] give a more extensive account on FD approximations, along with considerations on the optimal choice of the perturbation h as compromise between cut-off and cancellation error.

Coming back to WORHP's design for solving large-scale sparse problems, techniques for computing FD approximations with tolerable effort are crucial. The technique for ∇f is obvious: Only calculate FD approximations for its nonzero entries, cutting the required evaluations of f from n to $n_{nz} \leq n$, assuming that the unperturbed value $f(x)$ is cached.

Computing FD approximations for ∇g , however (and in complete analogy ∇h , which we will omit here), is not as simple, since the general case is that g depends on most, if not all variables x_j . If we can evaluate g component-wise, i.e. $g_i(x)$, we are in the same case as with ∇f . Unfortunately, this case is the exception, and g is usually evaluated as "black-box", vector-valued function.

The solution is to *group* the variables x_j according to the dependency of the components g_i on them: Two variables x_j and x_k can be *paired*, if each component g_i depends on at most one of them. Applying this approach to all variables, we can construct $1 \leq n_g \leq n$ groups, each of which contains one or more variables. The case $n_g = n$ may indicate that one g_i

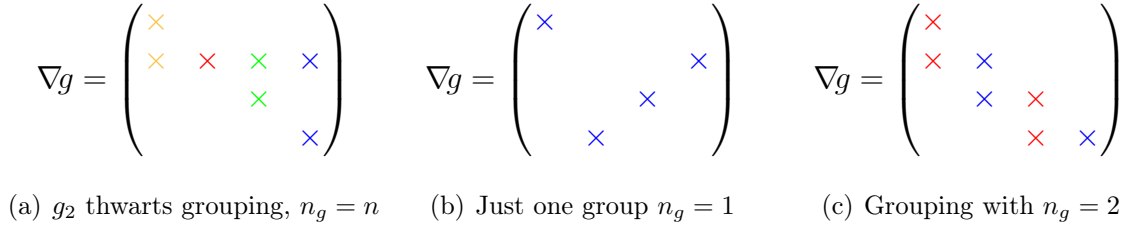


Figure 1.3.: Three examples for variable grouping; per-column coloring indicates the variable groups.

depends on all x_j , whereas the (uncommon) case $n_g = 1$ implies, that the components g_i depend on pair-wise different variables x_j , for instance if ∇g has a (permuted) diagonal sparsity structure. Three graphical examples in figure 1.3 illustrate the concept.

This variable grouping is of practical relevance for our purposes, since it allows to reduce the number of evaluations of g for computing a forward (or backward) difference approximation of ∇g from n to n_g :

Consider example 1.3 (c), where perturbing x_1 by evaluating $g(x + h \cdot e_1)$ only affects g_1 and g_2 , whereas $g_i(x + h \cdot e_1) = g_i(x)$ for $i = 3, 4$; vice versa, if we perturb x_3 . Thus we can aggregate (x_1, x_3) and (x_2, x_4) , evaluate

$$g(x + h(e_1 + e_3)) \quad \text{and} \quad g(x + h(e_2 + e_4)),$$

and compute the appropriate differences to calculate forward differences. This approach only requires 2 evaluations, where the naive one needs 4; as alluded above, it is no coincidence that $2 = n_g$; the construction principle of the groups allows to aggregate the perturbations for all variables in the group, thus reducing the number of evaluations from n to n_g .

Besides proper bookkeeping, the pivotal task in group-FD approximations is thus to determine groups such that n_g is minimal. Considering graph theory, it can be shown that this task is equivalent to solving the *graph-coloring* problem, whose best-known instance is the coloring of maps, such that no two countries with a common border have the same color. Put in graph-theoretical terms, if two vertices are connected by an edge, they must have different colors. Figure 1.4 shows these so-called *induced* graphs for the examples in figure 1.3.

The graph-coloring problem has the unfortunate property of being *NP-hard*, i.e. there exists no (known) algorithm to solve it in polynomial time, and it is conjectured and generally believed that no such algorithm exists; the traveling salesman problem is of the same complexity class. The FD module thus uses heuristics with polynomial runtime complexity that usually yield good results. A detailed account of the technique, the various heuristics, numerical results, and an extension to second-order finite differences is given in [41].

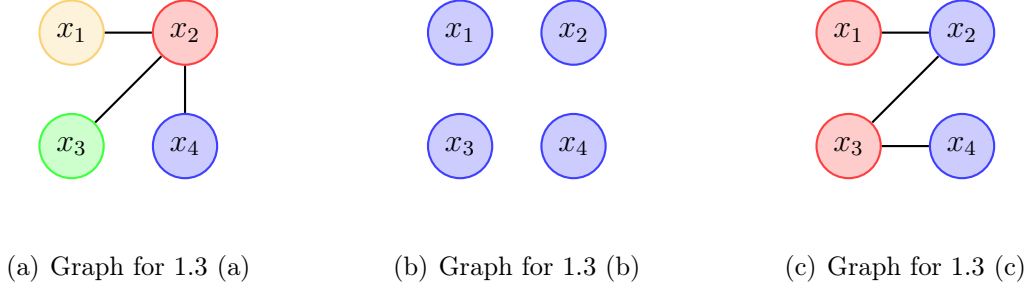


Figure 1.4.: Induced graphs for the examples in figure 1.3. The graphs are constructed by connecting two variables x_k and x_j , if there exists a g_i that depends on both.

The group-FD technique is of particular value to WORHP, because the full discretization or collocation approaches for discretizing optimal control problems typically result in high-dimensional, but very sparse and regularly structured matrices. The grouping approach can be highly efficient for this class of problems, and the (optimal) number of required groups is independent of the grid, a property we call *discretization-invariance*.

Dense BFGS update

To motivate the BFGS update, a small detour to the one-dimensional case is in order. Newton's method for a given differentiable function $f: \mathbb{R} \rightarrow \mathbb{R}$ is

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)},$$

and has local convergence order 2, if f' is Lipschitz continuous. Its drawback, however, is that it requires knowledge of the derivative and *two* function evaluations per iteration. The secant method

$$x_{k+1} = x_k - \frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})} \cdot f(x_k)$$

on the other hand has a lower convergence order of $\frac{1}{2}(1 + \sqrt{5}) \approx 1.618$, but only requires *one* function evaluation per iteration, if $f(x_{k-1})$ is cached. This leads to the secant method often being faster, despite its lower convergence order, and even if f and f' are still moderately cheap to evaluate; the speed advantage increases, as evaluations of f and f' get more expensive.

Back to SQP methods: The derivation with Newton's method leads to (QP), which requires second derivatives $\nabla_{xx}^2 L(x, \lambda, \mu)$. With the same motivation as in the one-dimensional case, we could try and replace this derivative by an extension of the secant method to the multi-dimensional case. Independently of each other, Broyden[9], Fletcher[20], Goldfarb[30], and Shanno[64] followed this path and devised the **BFGS** update in 1970. The alternative name *quasi-Newton* method(s) seems to have been coined by Shanno.

1. Nonlinear Programming

In its modern form, the BFGS method is an iterative rank-2 update of an initial matrix $H^{[0]}$ that maintains positive definiteness, if $H^{[0]}$ is positive definite, and the so-called curvature condition is met. $H^{[0]}$ is often chosen as identity matrix. The iterative update can be thought of as accumulating curvature information in the BFGS matrix, although it does *not* approximate the Hessian in a matrix-norm sense. In analogy to the one-dimensional secant method, it can be shown that the BFGS update maintains superlinear convergence properties of the SQP method.

To compute the rank-2 update, only first-order derivative information is required, namely

$$y^{[k]} := \nabla L(x^{[k+1]}, \lambda^{[k+1]}, \mu^{[k+1]}) - \nabla L(x^{[k]}, \lambda^{[k+1]}, \mu^{[k+1]}).$$

Note that the multipliers are indeed constant in both terms, the difference is computed with respect to x only. Abbreviating $H^+ = H^{[k+1]}$, $H = H^{[k]}$, $y = y^{[k]}$, and $d = d^{[k]}$, the update is then computed by

$$H^+ := H + \frac{yy^\top}{y^\top d} - \frac{Hdd^\top H^\top}{d^\top H d}.$$

If H is positive definite, then H^+ is as well iff $y^\top d > 0$; controlling this curvature condition is necessary to ensure that (QP) can be solved.

In contrast to evaluating $\nabla_{xx}^2 L(x, \lambda, \mu)$, this rank-2 update is cheap to compute from already known quantities. The lower convergence rate of the BFGS method is compensated by its cheaper computation and the fact no second derivatives are required.

It is obvious from the vector-vector products that both terms on the right hand side in general result in dense matrices. Therefore, we have to assume that H^+ is dense, even if H was sparse. This limits its usefulness to small- or medium-scale¹ problems, but motivates to contrive variations of the quasi-Newton methods to extend them into the sparse domain.

Sparse BFGS update techniques

Three considerations steer our search for sparse variants:

- The update maintains positive definiteness,
- the matrices enable superlinear convergence of the SQP method,
- the sparsity structure of $\nabla_{xx}^2 L(x, \lambda, \mu)$ is reproduced or at least covered by the update to adequately approximate its curvature.

We will loosely follow the extensive account in [41].

¹Neither of these terms is rigorously defined; 1,000 and 5,000 variables/constraints are reasonable, “fuzzy” borders between the small-, medium-, and large-scale domains.

Assume we have the following sparsity structure:

$$\nabla_{xx}^2 L = \begin{pmatrix} \times & \times & \times & & & \\ \times & \times & & & & \\ \times & & \times & & & \\ & & & \times & \times & \\ & & & \times & \times & \end{pmatrix}.$$

Observing that

$$\begin{pmatrix} B_1 & 0 \\ 0 & B_2 \end{pmatrix} > 0 \iff B_1 > 0 \wedge B_2 > 0,$$

an obvious approach is to cover the matrix above with a 3×3 and a 2×2 matrix, and to perform separate BFGS updates on these sub-matrices. It is obvious from the previous observation that the resulting update is again positive definite. The block approach is easily extended to arbitrary number and sizes of the individual blocks, but can only cover matrices whose sparsity structure is separable, which is far from general. The more realistic tridiagonal structure

$$\nabla_{xx}^2 L = \begin{pmatrix} \times & \times & & & & \\ \times & * & + & & & \\ & + & * & \times & & \\ & & \times & * & + & \\ & & & + & + & \end{pmatrix}$$

can be covered, if we allow *single intersections* of the blocks, in this case of four 2×2 blocks; the intersecting elements are marked by $*$. It can be shown that positive definiteness of the update can be maintained by a (potentially iterative) update of the input y , partitioned according to the block it belongs to. To also cover matrices of the form

$$\nabla_{xx}^2 L = \begin{pmatrix} \times & \times & \times & & & \\ \times & * & * & + & & \\ \times & * & * & * & \times & \\ & + & * & * & \times & \\ & & \times & \times & \times & \end{pmatrix}$$

we need to allow *multiple intersections* of the blocks, here three 3×3 blocks; elements subject to single intersections are $*$, and the central element present in all three blocks is marked as $*$. An extension of the single-intersection processing of y is able to guarantee positive definiteness for the multiply-intersecting block update as well. Furthermore, it can be proven that the superlinear convergence properties of the dense BFGS method carry over to these block-BFGS methods, even in the multiply-intersecting case (the others being special cases of this general one).

Two issues remain: Firstly, the block-BFGS matrices are generally denser than the original sparsity pattern, due to our approach of covering it by blocks; unless the cover happens

1. Nonlinear Programming

to exactly match the original pattern, structural zeros are inevitably lost. Secondly, a sparsity structure like

$$\nabla_{xx}^2 L = \begin{pmatrix} \times & \times & & \times \\ \times & \times & \times & \\ & \times & \times & \\ \times & & & \times \end{pmatrix}$$

warrants a different approach, since the only block update that covers this structure consists of a single, dense block. Permutations are a possible means of transforming structures like the above into more tractable structures. For our block cover, the matrix bandwidth is an indication of the required block size, thus bandwidth-reducing reordering algorithms such as (reverse) Cuthill-McKee [14, 28] are an option.

The sparse BFGS (SBFGS) method of WORHP uses a different approach, which could be described as block-wise implicit permutation and fusing. Figure 1.5 illustrates the idea on the last example sparsity pattern above.

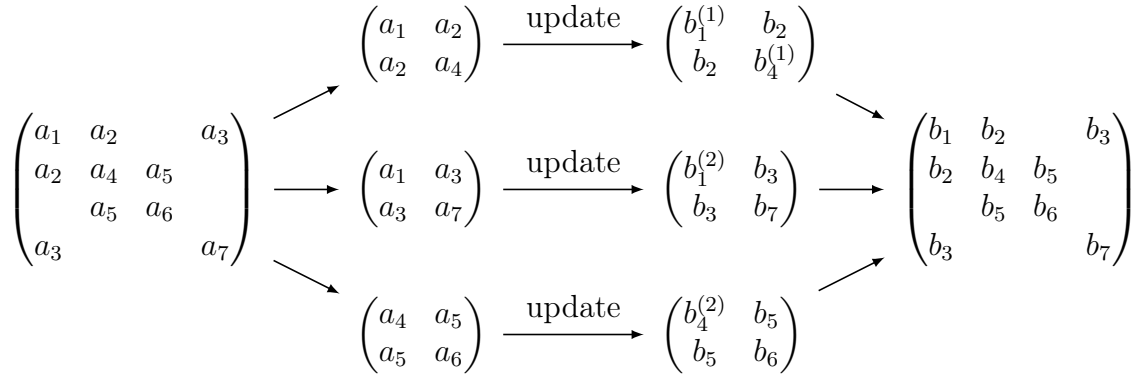


Figure 1.5.: Concept of SBFGS: Entries are collected across the matrix by implicit permutations to create dense blocks, on which the regular BFGS update is performed. Since some elements may be present in multiple blocks (here $b_1^{(i)}, b_4^{(i)}$), fusing them (to b_1 and b_4) requires special techniques to maintain positive definiteness.

The key idea of SBFGS is to collect entries from the whole matrix to aggregate them to dense block matrices; entries are rejected, if aggregating them would introduce a structural zero in the block matrix. For this reason, a_3 and a_7 are not aggregated into the first block, because this would require a nonzero right of/below a_5 . A parameter controls, whether these blocks can grow to arbitrary size, or have a maximum size. A proof of the superlinear convergence properties of SBFGS is given in [41].

1.3.2. Optimality and Termination Criteria 2

This section is concerned with the various termination criteria of WORHP, i.e. step 2 in figure 1.1. To keep our notation concise, we introduce the *numerical constraint violation*:

Definition 7. The numerical constraint violation c is a scalar quantity that allows to gauge quantitatively, how well a point x satisfies the constraints of the standard optimization problem (NLP):

$$(CV) \quad c(x) := \max_{(i,j)} \left(\max(0, g_i(x)), |h_j(x)| \right) \\ \text{for all } (i, j) \in \{1, \dots, m_1\} \times \{1, \dots, m_2\}$$

In the strict sense, x is feasible iff $c(x) = 0$.

Scaled KKT conditions

Due to their design, the KKT conditions are the canonical termination criterion for SQP methods; despite their different design, this is also true for the Interior-Point methods. Adapted to the realities of finite precision arithmetic, and given a triplet of small, positive tolerances $\varepsilon_{\text{opti}}$, $\varepsilon_{\text{feas}}$, and $\varepsilon_{\text{comp}}$, the standard KKT conditions impose

- | | | |
|-----|---|--------------------------------------|
| (a) | $\ \nabla_x L(x, \lambda, \mu)\ \leq \varepsilon_{\text{opti}},$ | Optimality |
| (b) | $c(x) \leq \varepsilon_{\text{feas}}$ | Feasibility |
| (c) | $\lambda_i g_i(x) \leq \varepsilon_{\text{comp}},$ | $i = 1, \dots, m_1,$ Complementarity |
| (d) | $\lambda_i \geq 0,$ | $i = 1, \dots, m_1,$ |

where $\|\cdot\|$ can be any norm, but usually $\|\cdot\|_p$ with $p \in \{1, 2, \infty\}$.

Unfortunately, the standard KKT conditions are not robust against numerical error; criterion (a) is particularly sensitive against small perturbations of the derivatives, such as the error incurred from finite-difference approximations. Due to WORHP's approach for solving the quadratic subproblems, the multipliers introduce further rounding errors: The QP-solver never returns multipliers that are exactly zero, even if the corresponding constraint is inactive.

Criterion (b) causes the solver to struggle if the constraints are badly scaled, but it is a design decision of WORHP to *not* scale the constraints. This is not at least because there is no canonical way of scaling them, and because we do not want to “soften” the notion of feasibility, as some other NLP solvers do²; a point that is feasible in WORHP's sense is *guaranteed* to satisfy all constraints to the prescribed precision.

An inspiration for improving on the standard conditions flows from the users' wish to determine values of x that satisfy

$$|f(x^*) - f(x)| \leq \varepsilon_{\text{opti}}.$$

²The *elastic mode* of SNOPT is a prime example: Troublesome constraints are removed from the problem and instead added to the objective, using a penalty approach to (try and) satisfy them.

1. Nonlinear Programming

Applying various heuristics and estimates to the above estimate, and performing an amount of numerical experimentation finally leads to the *scaled* KKT conditions, to replace the scaling-agnostic criterion (a) by

$$(a_s) \quad \|\nabla_x L(x, \lambda, \mu)\|_\infty \leq \frac{\varepsilon_{\text{opti}} \cdot \max(1, |f(x)|) + \max(\|\lambda^\top g(x)\|_\infty, \|\mu^\top h(x)\|_\infty)}{\|d\|_\infty},$$

dubbed *scaled* KKT conditions, abbreviated *sKKT*. Variants (a_s) and (a) differ only slightly with well-scaled problems, but the scaled one is more robust against numerical errors and badly scaled problems. The scaled KKT conditions are also chosen to terminate the solver, if cancellation errors start to blanket the numerical quantities, thus obstructing further iteration progress.

Low-pass filters

With a similar rationale, the *low-pass* filters try and detect the “trend” of the iteration process—a concept that is intuitively clear to every human user who observes the running solver, but not usually considered by NLP solvers. This termination criterion is supposed to become active when a human user would terminate the solver, because the progress indicators do not improve noticeably anymore. Its name is derived from viewing the progress indicators as time-dependent signal, where changes between consecutive iterations have high frequency and are filtered, while the low-frequency long-term trend is to be observed.

Given an iterate $x^{[k]}$, the low-pass filter computes the two quantities

$$\begin{aligned} r_f^+ &:= \alpha_f \cdot f(x^{[k]}) + (1 - \alpha_f)r_f, \quad \text{and} \\ r_g^+ &:= \alpha_g \cdot c(x^{[k]}) + (1 - \alpha_g)r_g, \end{aligned}$$

and terminates whenever

$$\frac{|r_f^+ - r_f|}{\max(1, |r_f^+|)} < 10^{-3}\varepsilon_{\text{opti}}, \quad \text{or} \quad \frac{|r_g^+ - r_g|}{\max(1, |r_g^+|)} < 10^{-3}\varepsilon_{\text{feas}}.$$

A third low-pass filter is interwoven with the *merit function*, detailed in section 1.3.5.

Acceptable solutions

To appreciate the notion of an “acceptable” solution, we have to adopt an engineer’s point of view. Considered with mathematical rigor, an iterate $(x^{[k]}, \lambda^{[k]}, \mu^{[k]})$ either satisfies one of the termination criteria, in which case the solver terminates, or it does not, in which case the solver carries on. If, however, the problem is “stubborn” and resists our attempts

to find optimal points to high precision, or if solutions *close* to an optimal solution are helpful for the user, returning *acceptable* solutions makes sense.

An iterate $(x^{[k]}, \lambda^{[k]}, \mu^{[k]})$ is considered *acceptable*, if it satisfies the (scaled) KKT conditions to a lower precision; by default, WORHP uses $\varepsilon_{\circ}^{\text{acc}} := \sqrt{\varepsilon_{\circ}}$ instead of the strict tolerance ε_{\circ} , for $\circ \in \{\text{opti}, \text{feas}\}$, although this can be chosen by the user. Obviously, setting $\varepsilon_{\circ}^{\text{acc}} = \varepsilon_{\circ}$ disables this behavior.

The notion is useful only, if the solver is forced to terminate *before* finding a strictly optimal solution. This can be caused by

- reaching a user-defined limit (maxiter, timeout),
- slow progress,
- an error condition in the model that cannot be alleviated, or
- user intervention.

The notion of acceptable solutions is complemented by saving the best acceptable iterate found so far. This, again, is justified from a practical point of view: While the theory of optimization holds that under certain conditions (including properties of the problem and choice of the step size) successive iterates are *better* in a well-defined way, practical problems have an unfortunate tendency to violate theoretical assertions. Given a sufficiently “uncooperative” problem, we can sometimes observe that *both* feasibility and optimality measure actually decline from one iterate to the next. WORHP therefore *stores* an iterate $(x^{[k]}, \lambda^{[k]}, \mu^{[k]})$, if it satisfies the acceptable tolerances, and updates it, if a *better* iterate is found. Since we do not want to depend on a merit function or the filter (cf. section 1.3.5 for details on both) to judge this, a new iterate $(x^{[k]}, \lambda^{[k]}, \mu^{[k]})$ is considered as *better* than an already stored iterate (x_s, λ_s, μ_s) according to the simple rules of algorithm 1:

Algorithm 1 for updating stored acceptable solutions

```

if  $(x_s, \lambda_s, \mu_s)$  is feasible to strict tolerance  $\varepsilon_{\text{feas}}$  then
  if  $(x^{[k]}, \lambda^{[k]}, \mu^{[k]})$  is feasible to strict tolerance  $\varepsilon_{\text{feas}}$  then
    if  $f(x^{[k]}) < f(x_s)$  then
      Update  $(x_s, \lambda_s, \mu_s) \leftarrow (x^{[k]}, \lambda^{[k]}, \mu^{[k]})$  ▷ feasible and better objective
  else
    if  $(x^{[k]}, \lambda^{[k]}, \mu^{[k]})$  is less infeasible than  $(x_s, \lambda_s, \mu_s)$  then
      Update  $(x_s, \lambda_s, \mu_s) \leftarrow (x^{[k]}, \lambda^{[k]}, \mu^{[k]})$  ▷ improves feasibility

```

The focus of algorithm 1 on feasibility betrays WORHP’s deeper purpose as optimization component of optimal control problem solvers, but also caters for many practical engineering problems, where infeasible solutions are unphysical and therefore unusable.

Storing acceptable solutions can be useful when dealing with a model that is particularly difficult to optimize and any solution that is better than the initial guess is useful; this situation frequently arises in the early development phase of a new model, or when applying mathematical optimization to a *simulation* model for the first time. Setting the

1. Nonlinear Programming

acceptable tolerances to high absolute values instructs WORHP to start storing iterates early.

Additional heuristics

As mentioned earlier, many practical optimization problems have the unfortunate tendency to disregard mathematical prerequisites, such as f, g and h being sufficiently smooth, satisfying a constraint qualification in the optimum, or for the whole problem to be bounded from below. WORHP's heuristics can be considered as last-ditch effort to find “good” solutions even in cases, where our theory and the numerical methods fail.

Most simple, the *unbounded problem* detection heuristic triggers, if the current iterate is feasible, if the last search direction $d^{[k]}$ is a descent direction (i.e. $\nabla f(x^{[k]})^\top d^{[k]} < 0$) and the objective falls below a fixed limit, -10^{20} by default.

The *Fritz-John point* and *nondifferentiable point* detection heuristics are laid out in algorithm 2:

Algorithm 2 for detecting Fritz-John or nondifferentiable points

```

if  $\|d^{[k]}\|_2 \leq \varepsilon_{\text{opti}}$  then                                ▷ small search direction
  if  $c(x^{[k]}) \leq \varepsilon_{\text{feas}}$  then                                ▷ feasible point
    if  $|f(x^{[k]}) - f(x^{[k-1]})| \leq \varepsilon_{\text{opti}}$  then            ▷ slow progress
      if  $\|\nabla_x L(x^{[k]}, \lambda^{[k]}, \mu^{[k]})\|_\infty \geq \varepsilon_{\text{opti}}^{-1}$  then
        Terminate: Optimum may be nondifferentiable.
      if  $\|\lambda^{[k]}\|_\infty > c_{\text{FJ}}$  or  $\|\mu^{[k]}\|_\infty > c_{\text{FJ}}$  then
        Terminate: Optimum may be Fritz-John point.

```

Algorithm 2 deserves a short explanation: The first heuristic triggers for points where the values of a derivative “explode” close to the nondifferentiable point—this is not always true, with $x \mapsto |x|$ being a very simple counterexample, hence the heuristic would probably have been more aptly named as *singularity* heuristic.

The detection heuristic for Fritz-John points assumes that $\lambda^{[k]}, |\mu^{[k]}| \rightarrow \infty$ if converging towards a Fritz-John, since $\lambda_0 \rightarrow 0$ in this case. Since the multipliers may also show this behavior in a KKT point, the approach is only a heuristic one and may produce false alarms.

1.3.3. Hessian Regularization 3

At the core if every SQP method, quadratic problems of the form

$$\begin{aligned}
 & \min_{x \in \mathbb{R}^n} && \frac{1}{2}x^\top Qx + r^\top x \\
 & \text{subject to} && Ax \leq b \\
 & && Cx = d,
 \end{aligned}$$

have to be solved. For the solution to be well-defined and unique, Q has to be positive definite on the subspace spanned by the kernels of A and C , i.e.

$$x^T Q x > 0 \text{ for all } x \in \text{span}\{\ker A, \ker C\},$$

otherwise we lose uniqueness, or the problem may be unbounded. One can guess that ensuring this numerically would require QR, singular value or Schur decompositions, all of which are too expensive in the large-scale case. A common approach is therefore to ensure $Q > 0$ on the whole space, instead of the subspace dependent on A and C . If Q is a BFGS matrix, we remember that positive definiteness is one of its most important properties. If, however, $Q = \nabla_{xx}^2 L$, some work is in order:

To ensure positive definiteness of Q without prohibitive computational costs, Levenberg-Marquadt regularization may be used. This is done by determining a lower bound on the smallest eigenvalue of Q , the so-called *Gerschgorin bound*

$$\sigma = \min_{1 \leq i \leq n} \left(q_{ii} - \sum_{j \neq i}^n |q_{ij}| \right)$$

and using it to update Q by

$$(1.2) \quad Q^+ = Q + \tau \cdot \max\{-\sigma, 1\} I.$$

Here I denotes the identity matrix and τ is a factor which is adapted iteratively: choosing $\tau = 1$ guarantees positive definiteness, since it shifts the spectrum such that it is bounded from below by 1, but the Hessian matrix may be perturbed drastically if the Gerschgorin bound produces a negative σ with great modulus, either because the smallest eigenvalue is negative with great modulus, or because the Gerschgorin estimate is bad. Due to the inexact nature of the Gerschgorin estimate, $\tau = 1$ is sufficient but not necessary for establishing positive definiteness. The iterative update of WORHP tries to let $\tau^{[k]} \searrow 0$ as $x^{[k]} \rightarrow x^*$, since $\nabla_{xx}^2 L$ can be shown to have the right spectral properties in a minimum (confer also theorem 3), hence no regularization is required in a neighborhood of x^* .

1.3.4. Prepare the Quadratic Problem 4

Besides the regularization of the Hessian matrix, further modifications of the linear-quadratic problem approximation are either necessary or beneficial. Constraint relaxation ensures that the constraints can be satisfied, although they are only approximated to first order, while the weak-Active Set method was devised to use less memory and computational time by problem size reduction.

Constraint Relaxation

Problem (QP) may not always have a solution, which can be demonstrated by a very simple example. This also serves to demonstrate one of the dangers of starting from a

1. Nonlinear Programming

trivial initial guess, such as $x^{[0]} = 0$; the exact definition of what constitutes a trivial initial guess depends, of course, on the optimization problem.

Example. Consider the nonlinear inequality constraint

$$g(x) = 1 - x^2 \leq 0.$$

Linearizing this constraint at $x^{[0]} = 0$, we arrive at the inequality

$$g(x^{[0]}) + g'(x^{[0]}) \cdot d = 1 + 0 \cdot d \leq 0,$$

which obviously cannot be satisfied, hence (QP) cannot be solved.

To work around the problem of unsolvable QPs, Powell introduced the idea to perform a **constraint relaxation** (also referred to as **elastic constraints**) by introducing a *constraint relaxation variable* $\delta \in [0, 1]$ and an associated *constraint relaxation penalty* parameter $\eta_r > 0$, and using them to reformulate (QP) as

$$\begin{aligned} \min_{d \in \mathbb{R}^n, \delta \in [0, 1]} \quad & \frac{1}{2} d^T \nabla_{xx}^2 L(x, \lambda, \mu) d + \nabla f(x)^T d + \frac{\eta_r}{2} \delta^2 \\ \text{(rQP)} \quad & \text{subject to} \quad (1 - \sigma_i \delta) g_i(x) + \nabla g_i(x)^T d \leq 0, \quad i = 1, \dots, m_1 \\ & (1 - \delta) h(x) + \nabla h(x)^T d = 0. \end{aligned}$$

with

$$\sigma_i = \begin{cases} 0, & \text{if } g_i(x) < 0, \text{ i.e. } g_i \text{ is inactive} \\ 1, & \text{otherwise} \end{cases}$$

In practice, (rQP) is transformed into the standard QP formulation through

$$Q_r = \begin{pmatrix} \nabla_{xx}^2 L & 0 \\ 0 & \eta_r \end{pmatrix} \quad \text{and minimizing over} \quad d_r = \begin{pmatrix} d \\ \delta \end{pmatrix}.$$

The key advantage of constraint relaxation is that $(d, \delta) = (0, 1)$ is always feasible for (rQP). If, however, the solution (d, δ) satisfies $\delta = 0$, it is optimal for (QP). The relaxation variable can be controlled through the penalty term η_r : If it is too small, the resulting search direction d will not sufficiently decrease the constraint violation, hence the SQP method cannot converge to a feasible point. If η_r is chosen too large, the condition number of the extended matrix Q_r is large and the QP-solver may fail, or yield an imprecise solution. To find a compromise between both extremes, η_r is initially small and iteratively increased if δ exceeds a given upper bound.

By default, WORHP uses a single relaxation variable for all constraints. One can alternatively choose to use $m_1 + m_2$ relaxation variables, i.e. one for each constraint, which increases the QP dimensions and computational time, but may increase stability and overall performance for problems with “difficult” constraints.

Weak-Active Set

The weak-Active Set method is inspired by the established Active Set method for solving inequality-constrained quadratic problems. The concept of the (strict) Active Set method is to identify the set I of active indices. The quadratic problem (QP) is then modified to

$$\begin{aligned}
 (\text{QP}_{\text{AS}}) \quad & \min_{d \in \mathbb{R}^n} \quad \frac{1}{2} d^T \nabla_{xx}^2 L(x, \lambda, \mu) d + \nabla f(x)^T d \\
 & \text{subject to} \quad g_i(x) + \nabla g_i(x) d = 0, \quad i \in I \\
 & \quad \quad \quad h(x) + \nabla h(x) d = 0.
 \end{aligned}$$

i.e. the inactive inequality constraints are identified and dropped from the problem. The manual pen-and-paper solution of optimization problems is very similar to this approach.

The modified problem is equality-constrained and can be solved without further ado by solving a system of linear equations. The task of identifying the Active Set is nontrivial, however, because it may change between iterations (although it stabilizes when the SQP method converges to a solution), and because the solver essentially has to make a prediction for the active set at the *next* iterate $x^{[k+1]}$ that is computed using $I(x^{[k+1]})$. Since g is nonlinear, the prediction can be inaccurate, hence an iterative update procedure is needed, with an exponential worst-case performance of $O(2^{m_1})$.

While Interior-Point methods do not suffer from this complication, introducing the slack variables increases the problem size; this effect is strongest if the problem has many inequalities, which is very pronounced in discretized optimal control problems with path constraints. To take advantage of the benefits of both methods, the *weak-Active Set* method uses a conservative estimate to remove inequality constraints, and thus reduce the size of the quadratic subproblems (QP). This method does not share the difficulties of its strict precursor, since the estimate of the active set may be fuzzy, and any “suspicious” inequality constraints can be left untouched, since there is no need to transform them into equality constraints.

Definition (weak-Active Set). Let $x \in \mathbb{R}^n$ and $\delta_w \in \mathbb{R}$ with $\delta_w > 0$. A constraint $g_i(x)$ is called weak-active, when $g_i(x) \leq \delta_w$. Equality constraints are always weak-active. The *weak-Active Set* I_w is given by $I_w(x) = \{i \in \{1, \dots, m_1\} \mid g_i(x) \text{ is weak-active}\}$.

Applying this definition to (QP) gives us

$$\begin{aligned}
 (\text{QP}_{\text{wAS}}) \quad & \min_{d \in \mathbb{R}^n} \quad \frac{1}{2} d^T \nabla_{xx}^2 L(x, \lambda, \mu) d + \nabla f(x)^T d \\
 & \text{subject to} \quad g_i(x) + \nabla g_i(x) d \leq 0, \quad i \in I_w(x) \\
 & \quad \quad \quad h(x) + \nabla h(x) d = 0.
 \end{aligned}$$

The key difference to (QP_{AS}) is that the inequality constraints do not need to be “forced” to equalities, since the QP-solver is able to handle them by using an Interior-Point method, whereas older QP-solvers were only able to solve equality-constrained problems.

In contrast to the original method, the weak-Active Set technique is used to lower the computational effort, and not as a problem transformation required by the available methods.

1.3.5. Determine Step Size 6

Owing to its “heritage” as an application of Newton’s method, SQP methods have to perform *line search* to achieve global convergence as well. The task is to determine a step size α , usually $\alpha \in (0, 1]$, such that

$$x^{[k+1]} := x^{[k]} + \alpha d^{[k]}$$

satisfies some performance criterion. If the problem is unconstrained, as in Newton’s method, this criterion is simply the objective f to be minimized. In constrained optimization, however, feasibility is an additional, usually conflicting criterion to consider. Complementarity can be construed as yet another criterion, but is only considered in the termination criteria of WORHP.

Assuming we have a scalar performance criterion $\Phi: \mathbb{R}^n \rightarrow \mathbb{R}$ that behaves similar to f , in that smaller values mean better performance, we can define the scalar function

$$(1.3) \quad \phi: \mathbb{R} \rightarrow \mathbb{R}, \quad \phi(\alpha) := \Phi(x^{[k]} + \alpha d^{[k]}),$$

Further assuming that Φ is smooth, so is ϕ , and this invites us to apply methods from unconstrained, one-dimensional minimization, such as the secant or Newton’s method, to determine

$$\alpha_{\min} = \arg \min_{\alpha \in (0,1]} \phi(\alpha).$$

However, considering that this line search is but a single step embedded in the iterative scheme of the whole minimization algorithm, and that Φ is potentially expensive to evaluate—since it will have to involve f , g , and h —, determining the exact minimum α_{\min} is unnecessary effort.

Instead of performing (exact) line search, NLP methods in general perform variants of *inexact line search* which test a (small) number of trial step sizes α_i . Since the theory holds that $\alpha = 1$ is the optimal step size in a neighborhood of x^* , $\alpha_0 = 1$ is generally used as first trial step, with further steps defined through

$$\alpha_i = \beta^i, \quad i = 1, \dots, i_{\max}$$

with a decrease factor $\beta \in (0, 1)$ and an upper limit to the number of trial steps $i_{\max} \in \mathbb{N}$. WORHP modifies the choice of trial step sizes slightly, since for certain problem classes, $\alpha_0 = 1 - \varepsilon$ seems to be beneficial. Therefore, WORHP uses trial step sizes $\alpha_i = \alpha_0 \beta^i$, where α_0 can be influenced through a solver parameter.

Acceptance or rejection of a trial step size depends on the employed globalization method, where WORHP uses *merit functions* and a (2D-) *filter* as two major alternatives.

Merit functions

The function Φ in equation (1.3) is called *merit function*, which is an equivalent concept to the penalty functions shown in section 1.2. WORHP implements two merit functions: The *Augmented Lagrangian*

$$(AL) \quad L_a(x, \lambda, \mu; \gamma, \eta) = f(x) + \frac{1}{2} \sum_{i=1}^{m_1} \frac{1}{\gamma_i} \left((\max\{0, \lambda_i + \gamma_i g_i(x)\})^2 - \lambda_i^2 \right) + \sum_{i=1}^{m_2} \mu_i h_i(x) + \frac{1}{2} \sum_{i=1}^{m_2} \eta_i h_i^2(x)$$

and the L_1 merit function (also known as *exact L_1 penalty function*)

$$(L_1) \quad L_1(x; \gamma, \eta) = f(x) + \left(\sum_{i=1}^{m_1} \gamma_i \max(0, g_i(x)) + \sum_{i=1}^{m_2} \eta_i |h_i(x)| \right).$$

The SQP algorithm is highly sensitive to the choice of the penalty parameters (γ, η) , and their update is non-trivial. While the Augmented Lagrangian is differentiable, the L_1 merit function has nondifferentiable points. However, the L_1 merit function does possess a directional derivative along the search direction, and thus $\phi(\alpha) := \Phi(x^{[k]} + \alpha d^{[k]})$ is differentiable for $\Phi = L_1$.

Given a merit function Φ and the function ϕ constructed from it, various strategies exist to decide whether to accept or reject a trial step size α_i , established ones being the *Armijo(-Goldstein) rule* and the *Wolfe-Powell rule*:

The Armijo rule accepts points α that satisfy the Armijo-Goldstein condition

$$\phi(\alpha) \leq \phi(0) + \sigma \alpha \phi'(0).$$

The “strictness” parameter $\sigma \in (0, 1)$ can be used to control the required amount of improvement; higher values cause the Armijo rule to accept fewer points. Figure 1.6 illustrates the concept of the Armijo rule graphically.

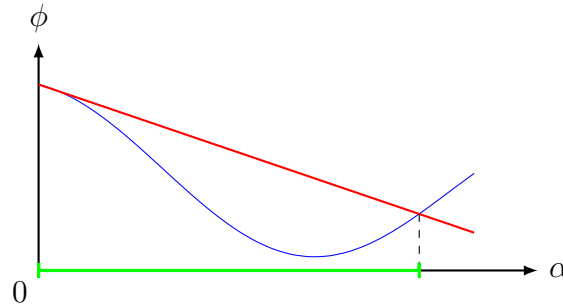


Figure 1.6.: Schematic view of the Armijo rule: The acceptance region, i.e. the set of points for which ϕ lies below the Armijo-Goldstein line is highlighted in green.

1. Nonlinear Programming

Bounding the iterates from above guarantees a minimum amount of improvement, although not enough to make the Armijo rule *efficient*, which is a technical criterion used to proof the global convergence properties of SQP methods; it can be shown, however, that the *scaled* Armijo rule is efficient, if the trial step sizes are chosen as $\alpha_i = s\beta^i$, $i = 1, \dots, i_{\max}$ and $s > 0$ is chosen appropriately. However, the scaled variant precludes the trial step size $\alpha = 1$, which is required to “inherit” the quadratic convergence order from Newton’s method. Alternatively, choosing $\alpha_i = \beta^{\pm i}$, i.e. interleaved *increasing* and decreasing trial step sizes also makes the Armijo rule efficient. In practice, WORHP employs the standard Armijo-Goldstein condition to enable locally *quadratic* convergence, even though *global* convergence cannot be proven for this approach.

The Wolfe-Powell rule is stricter than the Armijo rule, by adding a second condition. For given parameters $\sigma \in (0, \frac{1}{2})$ and $\rho \in [\sigma, 1)$, it accepts points α that satisfy

$$\begin{aligned}\phi(\alpha) &\leq \phi(0) + \sigma\alpha\phi'(0) & \text{and} \\ \phi'(\alpha) &\geq \rho\phi'(0)\end{aligned}$$

or alternatively, for the *strict* Wolfe-Powell rule

$$|\phi'(\alpha)| \leq -\rho\phi'(0).$$

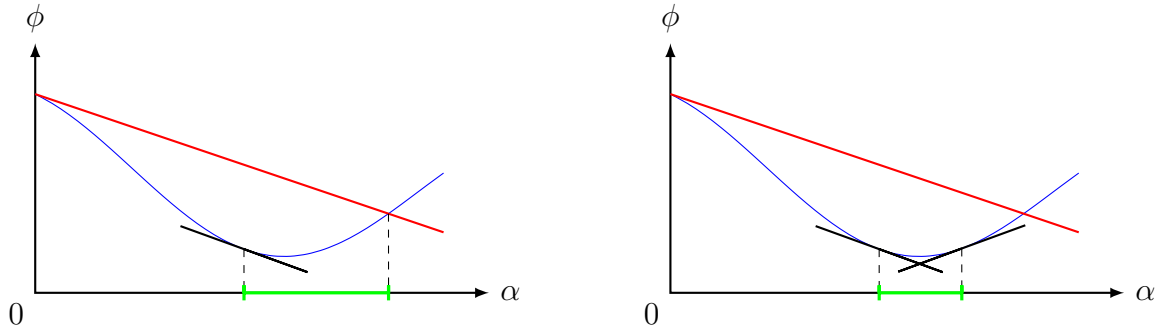


Figure 1.7.: Schematic view of the Wolfe-Powell rule (left), and its strict counterpart (right). The acceptance region is highlighted in green, smaller than for the Armijo rule, and yet smaller in the strict case, where only the derivative condition is active.

The advantage of the Wolfe-Powell rule is that it is *efficient*, i.e. it can be proven (under certain assumptions on the problem) that an SQP method converges globally, if it uses one of the Wolfe-Powell strategies to choose the step size. Its great disadvantage is that it requires additional derivative evaluations for every trial step size, where the Armijo rule only requires function evaluations (which may already be expensive). For this reason, WORHP exclusively employs the Armijo rule with one of the merit functions.

The filter method

The suitable choice of the penalty term and penalty parameters is one of the difficulties of using merit functions, owing to the approach of combining two (conflicting) criteria

into a single, scalar one. The filter method is a 2-dimensional criterion that considers objective and feasibility separately, thus circumventing this difficulty. A trial iterate $x^{[k+1]} = x^{[k]} + \alpha d^{[k]}$ is accepted by the filter if it improves either the objective function f or the constraint violation c , as defined in (CV).

Definition (dominance). The point $(f(x^{[k]}), c(x^{[k]}))$ is said to be *dominated* by the point $(f(x^{[j]}), c(x^{[j]}))$ if both

$$f(x^{[j]}) \leq f(x^{[k]}) \quad \text{and} \quad c(x^{[j]}) \leq c(x^{[k]}).$$

A filter \mathcal{F} is a list of pairs $(f(x), c(x))$ such that no entry dominates any other. A trial iterate $x^{[k+1]}$ is accepted by the filter if the corresponding pair $(f(x^{[k+1]}), c(x^{[k+1]}))$ is not dominated by any filter entry. The pair $(f(x^{[k+1]}), c(x^{[k+1]}))$ can then be added to the filter and all entries that are dominated by this new pair have to be deleted. Otherwise a new trial iterate with a smaller step size is tested.

The filter can be initialized as $\mathcal{F} = \{\infty, \infty\}$ or, with an upper bound to the constraint violation, as $\mathcal{F} = \{-\infty, c_{\max}\}$.

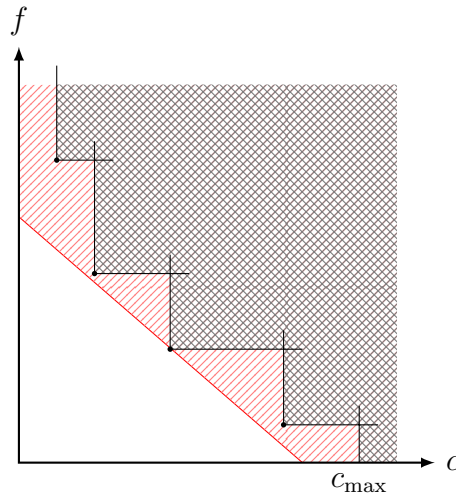


Figure 1.8.: Schematic view of the filter method and comparison with a fictional merit function. The rejection region of the filter (checkered) is much smaller than that of the merit function (striped).

The simple principle of the filter method has to be extended by various additional conditions and heuristics to prevent circling behavior, convergence to infeasible points or slow overall progress. Furthermore, the update of the τ parameter in the Hessian regularization (1.2) relies on the merit function, by comparing predicted and actual reduction. No scalar criterion like this is available in the filter method, hence the regularization operates on the inertia of the Hessian. For this reason, the filter method in WORHP depends on linear algebra solvers that compute the inertia.

An extensive account of the theory, implementation in WORHP, and numerical results is given in [45].

1.3.6. Recovery Strategies *

The line search is an obvious Achilles heel of the algorithm, since all previous steps influence its outcome; a bad initial guess, faulty derivatives, or an imprecise solution of the QP. All of these influences can cause a line search breakdown; unfortunately, backtracing the breakdown to its cause next to impossible, unless prohibitively expensive analysis tools constantly supervise the solver. The recovery strategies were devised as cheap(er) heuristics to be activated for a single iterations after a line search breakdown or an error condition in the QP solver.

WORHP has the following recovery strategies:

- The SLP (Successive *Linear* Programming) strategy replaces the Hessian or BFGS matrix in (QP) by the identity, which essentially reduces the SQP method to a first-order steepest descent method. The rationale is that Q may have a high condition number, which either causes a breakdown of the QP or linear solver, or later in the line search, due to an inexact search direction. Replacing Q by the well-conditioned identity eliminates this source of error.
- The non-monotone strategy allows local increase of the merit function, by replacing the current value of the merit function $\Phi(x^{[k]})$ by its lowpass-filtered value, which is always greater.
- The dual feasibility strategy activates the so-called *dual feasibility mode* which tries to find a feasible point by solving the least-squares problem $\min \|c(x)\|_2^2$. In filter and Interior-Point methods, this step is called *feasibility restoration*. The term *dual* stems from the solution method: The QP solver internally solves linear systems of the form

$$\begin{pmatrix} Q & A^T \\ A & 0 \end{pmatrix} \begin{pmatrix} x \\ \lambda \end{pmatrix} = \begin{pmatrix} f \\ c \end{pmatrix},$$

which can be used to solve the normal equation $A^T A d = -A^T c$ by rewriting it as

$$\begin{pmatrix} I & A \\ A^T & 0 \end{pmatrix} \begin{pmatrix} l \\ d \end{pmatrix} = \begin{pmatrix} 0 \\ -A^T c \end{pmatrix}.$$

This is akin to the dual problem, hence the designation as *dual* feasibility mode.

- As a last resort, the force strategy can be interpreted as a restart of the optimization run. It forces the line search to accept the step size α_f determined such that $\|\alpha_f d^{[k]}\| = 1$ to make a sufficiently great step to escape the current problematic region.

Two further strategies, the *ascent* and the *cut* strategy, are heuristics that can charitably be described as experimental and are better left undocumented.

Architecture of WORHP

A spider conducts operations that resemble those of a weaver, and a bee puts to shame many an architect in the construction of her cells. But what distinguishes the worst architect from the best of bees is this, that the architect raises his structure in imagination before he erects it in reality.

(Karl Marx, *Das Kapital*)

2.1. Practical Problem Formulation	33
2.2. Sparse Matrices	35
2.2.1. Coordinate Storage format	35
2.2.2. Compressed Column format	36
2.3. Data Housekeeping	38
2.3.1. The traditional <i>many arguments</i> convention	39
2.3.2. The USI approach in WORHP	40
2.4. Reverse Communication	43
2.4.1. Division into <i>stages</i>	44
2.4.2. Implementation considerations	46
2.4.3. Applications	48
2.5. Serialization	49
2.5.1. Hotstart functionality	49
2.5.2. Reading parameters	49
2.5.3. Serialization format	50

2. Architecture of WORHP

The architecture of WORHP was designed to overcome technical shortcomings of traditional mathematical software architectures, to permit a high degree of user interaction with the solver, to allow a higher degree of modularity than traditional mathematical software does and to enable *both* industrial-grade usability and its use as an academic experimental platform.

The term “traditional mathematical software” very loosely refers to many established and mature software packages written in some FORTRAN (i.e. pre-90) dialect, which often excel in terms of performance or precision, but face difficulties regarding usability or maintainability due to the remarkable restrictions of the language; from an architectural point of view, WORHP is an attempt to right some of the wrongs imposed on programmers during the long reign of the old FORTRAN standards and dialects.

The tools developed for and used in WORHP to this end are the Unified Solver Interface (USI) that defines a simple, yet powerful interfacing convention both externally and internally, and together with the Reverse Communication (RC) paradigm allows an unprecedented degree of interaction with a piece of mathematical software, and in combination with the XML module offers interesting possibilities for inspection, analysis and result processing. The final major tool of the bunch is WORHP’s workspace and memory management, the weak spot of many FORTRAN software packages’ usability.

2.1. Practical Problem Formulation

The standard formulation for constrained nonlinear optimization problems considered in chapter 1 is

$$\begin{aligned}
 \text{(NLP)} \quad & \min_{x \in \mathbb{R}^n} f(x) \\
 & \text{subject to } g(x) \leq 0 \\
 & h(x) = 0,
 \end{aligned}$$

which is very suitable and convenient for the theoretical considerations. However, it can be cumbersome to formulate practical optimization problems, for instance a box constraint $1 \leq x_i \leq 2$. This is unfortunate for various reasons:

- It places an additional burden on the users, who have to reformulate their optimization problems to match the abstract mathematical notation. This can be particularly irksome, if the solver is integrated into existing models.
- In the case of box constraints, information is lost by transforming them into the standard inequality formulation; box constraints receive special treatment in various places, which improves both performance and stability.
- During development and integration, it may be useful to “switch” individual constraints on and off, or soften equality constraints to (tight) inequalities.

Therefore, WORHP operates on problems in the more flexible formulation

$$\begin{aligned}
 \text{(OP)} \quad & \min_{x \in \mathbb{R}^n} F(x) \\
 & \text{subject to } \begin{pmatrix} l \\ L \end{pmatrix} \leq \begin{pmatrix} x \\ G(x) \end{pmatrix} \leq \begin{pmatrix} u \\ U \end{pmatrix}
 \end{aligned}$$

with bounds

$$l, u \in \mathbb{R}^n \quad \text{and} \quad L, U \in \mathbb{R}^m$$

and functions

$$F: \mathbb{R}^n \rightarrow \mathbb{R} \quad \text{and} \quad G: \mathbb{R}^n \rightarrow \mathbb{R}^m.$$

Formulations (NLP) and (OP) are equivalent through transformations of the form

$$\begin{aligned}
 g(x) = \pm x_i + c \leq 0 & \iff c = l_i \leq x_i \quad \text{or} \quad x_i \leq u_i = -c, \\
 \left. \begin{aligned} g_1(x) &= -c(x) + c_1 \leq 0 \\ g_2(x) &= c(x) + c_2 \leq 0 \end{aligned} \right\} & \iff G(x) = c(x) \text{ and } L = c_1, U = -c_2, \\
 h(x) = c(x) - c = 0 & \iff G(x) = c(x) \text{ and } L = U = c.
 \end{aligned}$$

Since f is not affected by these transformations, we can always choose $F \equiv f$ and need not differentiate further between both. The constraints G , on the other hand, and their dimension m depend on the functions g and h , and on their dimensions m_1 and m_2 in a non-trivial way; only $m \geq m_2$ holds in general, irrespective of the problem structure.

Note on multipliers

Unlike the convention for the mathematical formulation (NLP), WORHP uses multipliers λ for box constraints and μ for the general (G) constraints. In case of two-sided constraints, two (NLP)-multipliers are associated with a single constraint, but it only has a single (OP)-multiplier. This is only seemingly problematic: For equality constraints, both (NLP)-multipliers coincide, thus the (OP)-multiplier takes their common value; for inequality constraints, at most one of them can be active at any point, and the (OP)-multiplier attains the value of the active (NLP)-multiplier, the inactive one being silently discarded.

2.2. Sparse Matrices

Due to its fundamental design as a solver for large-scale problems, WORHP operates exclusively on sparse matrices (but can be “tricked” into operating on dense matrices). As laid out in definition 5 and illustrated by example 6, the central idea behind sparse matrix representations is the omission of zero entries, which saves memory and computational time. Nonzero entries define the sparse matrix structure, which is inherent to every instance of (NLP) and constant, while the actual values of the nonzero entries depend on the current point (note that nonzero entries may of course attain the numeric value 0).

WORHP accepts two different sparse matrix representations: the Coordinate Storage (CS) format, or the Compressed Column (CC) format. They differ in their storage requirements and their suitability for certain matrix operations. Both formats in general, and WORHP’s specializations of them, are described in the following paragraphs. WORHP’s conventions for sparse matrix representation allow it to naturally operate on dense matrices given in Fortran’s storage sequence for 2D arrays, without conversions or special indexing; only the Hessian matrix with its “diagonal-last” ordering is an exception to that rule.

2.2.1. Coordinate Storage format

The *coordinate storage* or *triplet* format is the “intuitive” sparse matrix format, where a nonzero matrix entry $a = (A)_{ij}$ corresponds to an (a, i, j) triplet. These triplets are split up into the `val`, `row` and `col` arrays such that $(\text{val}_k, \text{row}_k, \text{col}_k) = (a, i, j)_k$ for some enumeration $k = 1, \dots, n_{nz}$ of the nonzero entries; a priori, no kind of ordering has to be assumed for this construction. Thus the storage requirement for the CS format is n_{nz} double + $2n_{nz}$ integer.

Since WORHP uses the CS format internally, it is the preferred sparse matrix format.

While the general format makes no assumptions on the sorting or uniqueness of the entries, WORHP imposes the following restrictions:

- CS-1. Entries are sorted ascending in column-major order.
- CS-2. Each matrix entry may be present at most once.
- CS-3. **Vectors only:** The column index is omitted for sparse vectors.
- CS-4. **Hessian only:** Only the lower triangular part is saved; first the strictly lower triangular (i.e. subdiagonal) entries h_{ij} with $i > j$ in column-major order, last *all* diagonal entries h_{ii} in ascending order, regardless of whether the problem has a nonzero entry at h_{ii} .

Restriction CS-1 is of special importance, because it allows the use of more efficient algorithms, simplifies conversion between CS and CC, and since its memory layout coincides with that of dense matrices in Fortran, WORHP can easily be integrated into dense problem formulations. Restriction CS-4 is necessary to enable the Hessian regularization by diagonal update, described in section 1.3.3.

2. Architecture of WORHP

A number of matrix operations can be carried out in a simple and efficient way with the CS format, among them

- Matrix transpose (by swapping row and col, but changing the ordering),
- Adding or removing entries,
- Mx and $y^T M$ multiplications.

Example 8. We consider again the functions in example 6, and their derivatives

$$F(x) = 1 + x_2 + \frac{1}{3}x_3^3, \quad G(x) = \begin{pmatrix} 4x_1x_2 + 5 \\ 3x_1^2 + 8x_3 \\ 10x_3 + 12 \end{pmatrix},$$

$$\nabla F(x) = (0, 1, x_3^2), \quad \nabla G(x) = \begin{pmatrix} 4x_2 & 4x_1 & 0 \\ 6x_1 & 0 & 8 \\ 0 & 0 & 10 \end{pmatrix},$$

$$\text{and } \nabla_{xx}^2 L(x, \mu) = \begin{pmatrix} 6\mu_2 & 4\mu_1 & 0 \\ 4\mu_1 & 0 & 0 \\ 0 & 0 & 2x_3 \end{pmatrix}$$

The gradient DF, Jacobian DG and Hessian HM encoded in WORHP's CS format are then given by

- DF: `val` = $(1, x_3^2)$, `row` = $(2, 3)$, `col` = \emptyset , because of CS-3.
- DG: `val` = $(4x_2, 6x_1, 4x_1, 8, 10)$, `row` = $(1, 2, 1, 2, 3)$, `col` = $(1, 1, 2, 3, 3)$.
- HM: `val` = $(4\mu_1, 6\mu_2, 0, 2x_3)$, `row` = $(2, 1, 2, 3)$, `col` = $(1, 1, 2, 3)$. Note that, because of CS-4, the elements are ordered as $(h_{21}, h_{11}, h_{22}, h_{33})$ and $h_{22} \equiv 0$ despite being a structural zero is included, because it is a diagonal element.

2.2.2. Compressed Column format

The *compressed column* format, closely related to the Harwell-Boeing format for saving sparse matrices to files, is a more technical sparse matrix format, where a nonzero matrix entry $a = (A)_{ij}$ corresponds to an (a, i) pair (plus additional indices per column). These pairs are split up into the `val` and `row` arrays as $(\text{val}_k, \text{row}_k) = (a, i)_k$ for an enumeration $k = 1, \dots, n_{nz}$ which orders the nonzero entries such that $j_{k+1} \geq j_k$. In other words, the nonzero entries are sorted ascending by their column index, so that every column corresponds to a contiguous block of elements:

$$\text{val and row} = \left(\boxed{\text{column}_1}, \boxed{\text{column}_2}, \dots, \boxed{\text{column}_n} \right)$$

The column indices `colp` are the start indices into these blocks such that all entries $(\text{val}_k, \text{row}_k)$ for $k = \text{col}_p, \dots, \text{col}_{p+1} - 1$ belong to column p (hence, `col` has dimension

$n+1$, its values are non-decreasing, with mandatory values $\text{col}_1 = 1$ and $\text{col}_{n+1} = n_{nz} + 1$. If two consecutive values $\text{col}_p = \text{col}_{p+1}$, then column p is empty.

WORHP imposes some additional restrictions on the CC format:

CC-1. Entries are sorted by row in each column, i.e. for all $(\text{val}_k, \text{row}_k)$ we have $\text{row}_{k+1} > \text{row}_k$, for all $k = \text{col}_p, \dots, \text{col}_{p+1} - 2$, for all columns p .

CC-2. **Vectors only:** The column index is omitted for sparse vectors.

CC-3. **Hessian only:** Only the lower triangular part is saved, and *all* diagonal entries h_{ii} , regardless of whether the problem has a nonzero entry at h_{ii} .

The restriction CC-1 is of special importance, because it ensures that $(\text{val}_k, \text{row}_k)$ where $k = \text{col}_i$ corresponds to the diagonal entry h_{ii} in the Hessian, and it arranges the memory layout to coincide with that of dense matrices in Fortran, so WORHP can easily be integrated into dense problem formulations.

Note that restriction CC-1, together with CS-1 and CS-2 ensures that both formats only differ by their col indices – the representation of vectors is thus identical.

For Hessian matrices, CC-3 implies that each column has at least one entry, hence col is strictly increasing, i.e. $\text{col}_{p+1} \geq \text{col}_p + 1$ for all p .

The CC format is most efficient for vector-matrix multiplications, and for quick access to columns (and Hessian diagonal entries if CC-1 is in effect).

Example 9. Consider the functions in Example 6. The gradient DF, Jacobian DG and Hessian HM encoded in WORHP's CC format are then given by

DF: identical to the CS case.

DG: $\text{val} = (4x_2, 6x_1, 4x_1, 8, 10)$, $\text{row} = (1, 2, 1, 2, 3)$, $\text{col} = (1, 3, 4, 6)$.

HM: $\text{val} = (6\mu_2, 4\mu_1, 0, 2x_3)$, $\text{row} = (1, 2, 2, 3)$, $\text{col} = (1, 3, 4, 5)$. Note that, because of CC-3, the “non-nonzero” entry $h_{22} \equiv 0$ is included, because it is a diagonal element.

2.3. Data Housekeeping

NLP solvers move substantial amounts of data between caller and their internals. The technical limitation of legacy numerical software to scalars and arrays as routine arguments results in heavy, burdensome exterior interfaces and often “infects” the interior interfaces as well. This can be a burden on both the user and the maintainer, especially so when making modifications that add further required arguments.

The Unified Solver Interface (USI) is an attempt to break the time-honored tradition of established numerical software to use routine interfaces with more than a handful of arguments, by replacing it with clean, concise interfaces with low maintenance. With no intention to denigrate, the `snOptA` interface of SNOPT in listing 2.1 provides a very demonstrative example of the clutter found in legacy interfaces, particularly of FORTRAN software.

```

1  subroutine snOptA
2  & ( Start, nF, n, nxname, nFname,
3  &   ObjAdd, ObjRow, Prob, usrfun,
4  &   iAfun, jAvar, lenA, neA, A,
5  &   iGfun, jGvar, lenG, neG,
6  &   xlow, xupp, xnames, Flow, Fupp, Fnames,
7  &   x, xstate, xmul, F, Fstate, Fmul,
8  &   INFO, mincw, miniw, minrw,
9  &   nS, nInf, sInf,
10 &   cu, lencu, iu, leniu, ru, lenru,
11 &   cw, lencw, iw, leniw, rw, lenrw )
12 external
13 &   usrfun
14 integer
15 &   INFO, lenA, lencu, lencw, lenG, leniu, leniw, lenru, lenrw,
16 &   mincw, miniw, minrw, n, neA, neG, nF, nFname, nInf, nS,
17 &   nxname, ObjRow, Start, iAfun(lenA), iGfun(lenG), iu(leniu),
18 &   iw(leniw), jAvar(lenA), jGvar(lenG), xstate(n), Fstate(nF)
19 double precision
20 &   ObjAdd, sInf, A(lenA), F(nF), Fmul(nF), Flow(nF), Fupp(nF),
21 &   ru(lenru), rw(lenrw), x(n), xlow(n), xmul(n), xupp(n)
22 character
23 &   Prob*8, cu(lencu)*8, cw(lencw)*8,
24 &   Fnames(nFname)*8, xnames(nxname)*8

```

Listing 2.1: Interface of `snOptA` (the advanced SNOPT interface), not including the interface definitions of the user-defined functions.

While Object-oriented Programming (OOP) provides a feasible alternative—as seen in `Ipopt`—WORHP’s design constraints precluded an OOP approach. The central concept behind the Unified Solver Interface is as simple as powerful: All solver-relevant data is kept in four data structures. The USI describes the calling interface of any routine that expects instances of these four data structures and the fact that almost all high-level solver

routines share this interface. We will shed some light on the rationale and consequences of this design in this section.

2.3.1. The traditional *many arguments* convention

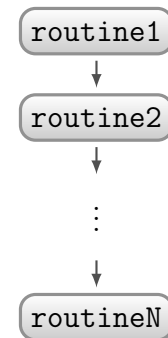
Let us consider listing 2.1, considering it as an arbitrary representative of complex numerical software: It shows (part of) the classical FORTRAN-style interface of SNOPT's advanced interface `snOptA` and expects an impressive total of 49(!) arguments to be supplied by its caller. We will dub this approach the *many arguments* interface convention.

One advantage of the *many arguments* convention lies in its technical simplicity: All programming languages of practical relevance to numerical computing support arrays and function pointers, and are thus able to call this routine.

A major disadvantage of this convention lies in the pure number of arguments, each of a certain data type, and most with an individual length, and their positions in the calling sequence; since FORTRAN does not yet support keyword arguments, order matters. Every caller thus has to duplicate a sizable block of argument declarations in the calling function, followed by a similarly sizable call.

Let us further consider the *many arguments* convention from a technical and maintenance point of view: Most software packages with a minimum amount of technical or algorithmic complexity will be structured into various routines¹. This implies a calling hierarchy.

Let us assume for the sake of the argument that `routine1` calls `routine2` which then calls `routine3` etc., until we reach `routineN`. In general, each of these routines will have its own specific interface `routine k (arg $_1^k$, arg $_2^k$, ..., arg $_{n_k}^k$)`, and quite often $\{\arg_i^j \mid i = 1, \dots, n_j\} \subset \{\arg_i^k \mid i = 1, \dots, n_k\}$ for $j \geq k$. holds, except for low-level functions like the highly abstract and specialized BLAS routines, or plain `printf`.



If we expand or otherwise modify `routineN` –for instance by adding a parameter setting that the user can modify– we add `arg $_{n_N+1}^N$` to `routineN`, and in turn need to modify all superordinate $N-1$ interfaces by adding this new argument to all calling routines. Depending on the depth N , the structure and size of the overall code and the usage of `routine k` in other places, the refactoring effort can be considerable; and to add insult to injury, since legacy FORTRAN compilers were unable to perform complete argument checking, it was exceptionally easy to forget refactoring a call or an interface somewhere, which could cause anything from no notable change to completely erratic program behavior afterwards.

¹ As a matter of fact, it is considered bad software design *not* to structure software into as many routines as sensible and appropriate (using descriptive names), to minimize code duplication, enhance maintainability and make the code more readable.

2. Architecture of WORHP

Luckily, modern compilers will mostly be able to spot human mistakes like this one, but it seems that tools for automatic refactoring of this kind only exist in commercial IDEs for C++ code; the refactoring effort is still there, and we only considered a linear calling hierarchy; if `routineN` were a low-level function called by *many* superordinate functions, the refactoring effort quickly reaches prohibitive dimensions.

For the sake of completeness, this scenario could be covered by defining $\text{arg}_{n_N+1}^N$ as global, thus bypassing the calling functions. Although purists will consider this as gross heresy against modern programming conventions, this practice might be acceptable for static parameters (say, a print-level); extending this to actual data arguments, however, leads us back to the bad old FORTRAN practices, where data was passed around in mixtures of routine arguments and `COMMON` blocks. It also undermines any attempts to keep the solver mostly state-less, which we will come to appreciate later in this section.

Another complication comes about, if type qualifiers/attributes are added to the arguments. In C, relevant type qualifiers for function arguments are `const` and `restrict`; the use of `volatile` is for the most part restricted to kernel and driver programming, and usually of no interest to programmers of sequential algorithms. The Fortran analog of `const` is the `INTENT_F90` attribute; the effect of C's optional `restrict` is the default in Fortran, where we have `ALIAS_F90` as complementary keyword for the default case in C. If any routine argument has a qualifier, the subordinate arguments have to inherit it. If, for instance, `routine1` qualifies an argument as read-only (`const` in C, or `intent(in)` in Fortran), and passes it on to `routine2`, it is a semantic error if the latter does *not* qualify it likewise. Going back to our `routineN` example, we can create a similar refactoring cascade, if we need to remove a read-only qualifier from one of its arguments, since all superordinate routines have to have the qualifier removed as well. This time, circumventing this refactoring cascade by using global access to the argument in question may even lead to surprising behavior (see A.1 for a worked-out example). Adding `restrict` to an argument of `routine1` leads to a similar cascade in reverse order for as long as the argument is passed on to the lower-level routine.

Conclusion

The *many arguments* convention leads to bloat and can easily turn into a maintenance burden, if the code under consideration is not static, but subject to modifications and extensions. This precludes its use in WORHP for the high-level routines.

2.3.2. The USI approach in WORHP

The central idea behind the Unified Solver Interface (USI) is to bundle all solver data in four data structures transparent to both user and solver, and create a single routine interface shared by all high-level routines of the solver. These four data structures are

OptVar: Named after the **o**ptimization **v**ariables (primal and dual), encapsulated in it, and additionally holds the essential problem dimensions and current values of the

objective and the constraints, and the constraint bounds. In simple cases, users need only interact with this data structure.

Workspace: The solver workspace, holding the various internal (often vector-valued) quantities used by NLP solvers, like the current penalty parameter for the merit function, or the objective scaling factor. For “historic” reasons, the workspace also holds the derivative matrices, although this is earmarked for refactoring during the next major API-breaking change. Section 3.4 gives an account of the workspace memory management.

Params: As its name suggests, this structure holds the solver parameters, meaning scalars or vectors that somehow influence the way the solver works, like the feasibility tolerance, or an upper limit to major iterations (“maxiter”). The mapping created by the serialization module (see section 2.5) between this data structure and a parameter file is used by default to set solver parameters at startup.

Control: Since WORHP’s architecture is based on Reverse Communication, it needs additional variables for flow control. Where other packages use a single integer flag, WORHP uses a more sophisticated and robust approach, encapsulated in this opaque structure, with dedicated querying routines in place for interaction with the user.

The namesake Unified Solver Interface is of the form `<RoutineName>(opt, wsp, par, cnt)` and shared by most high-level routines of WORHP.

This approach of encapsulating everything into data structures has few drawbacks:

- Since access qualifiers (`const` or `intent_F90`) only apply to all of the structure, and not individual components, this method of compiler-enforced access control is very limited. One can argue, though, that this lack of access control (which means prohibiting access to certain pieces of data) is in keeping with WORHP’s intention to deliberately allow some level of “tinkering” with its functionality. Experience also shows that almost all users leave the components well alone, lest they “break” something, which is actually preferable over a purely technical approach.
- Setting up the data structures, specifying dimensions, parameters and matrix structures, and doing so in the right order turns out to be a surprisingly complex process that is usually performed *behind* the scenes by many-argument solvers, or not at all, because these solvers do not have complex data structures to be initialized in the first place. WORHP users nevertheless seem quite capable to handle this complexity admirably well, since they are provided with commented usage examples that they can mimic.

Despite the abovementioned shortcomings concerning access control and data structure initialization, the advantages of the USI approach seem to outweigh its disadvantages:

- The use of a common interface does away with the need to remember or look up the precise signature of routine interfaces, i.e. the number, order, type and meaning of arguments². This is less of an issue with C/C++, since current IDEs (such as Visual

²The reader is hereby challenged to try and memorize the `snOptA` interface.

2. Architecture of WORHP

Studio, Eclipse, KDevelop, Code::Blocks etc.) have comfortable auto-completion features, but there is no known editor that offers the same for Fortran.

- Whenever the set of inputs or outputs of some routine change, only the central data structure definition needs to be modified (if anything at all), but the USI interface will remain constant for the foreseeable future. In particular, the refactoring cascade explained above will not be triggered.
- Experimental changes that require access to more data are not discouraged by the need for potentially extensive refactoring.
- The complete solver data is accessible everywhere. It is therefore possible to place code segments based solely on logical considerations, instead of being constrained by data locality – this means that code that belongs together can stay together, which is beneficial for maintainability.

Furthermore, the USI approach enabled the Hotstart functionality of the serialization module (see section 2.5). When the concept of keeping *all* solver data in these data structures is rigorously adhered to, it will leave the solver *stateless*, which is a major prerequisite for reentrancy, i.e. the ability to run several instances of the solver in parallel.

Unfortunately, the QP-solver component of WORHP currently uses private variables, leaving the whole solver *stateful* again, which poses a significant obstacle to all parallelization attempts.

Conclusion

The Unified Solver Interface is a simple, yet elegant approach to overcome the drawbacks of the *many arguments* convention when an OOP approach is not feasible.

It offers benefits to both the users and developers of the solver, which is in keeping with the design goal of WORHP to provide industrial-grade usability in a way that does not preclude its continued extension, maintenance and use as experimental platform.

2.4. Reverse Communication

Basing WORHP on the *Reverse Communication* paradigm is the second major building block of its novel architecture³. Calling this a paradigm is not as pretentious as it may seem on the first glance, because it fundamentally changes the interaction between the caller and the solver: Where the traditional approach (dubbed *Direct Communication*) hands over control to the solver for better or worse, Reverse Communication puts the caller back into the driver’s seat; figure 2.1 summarizes this difference in graphical form.

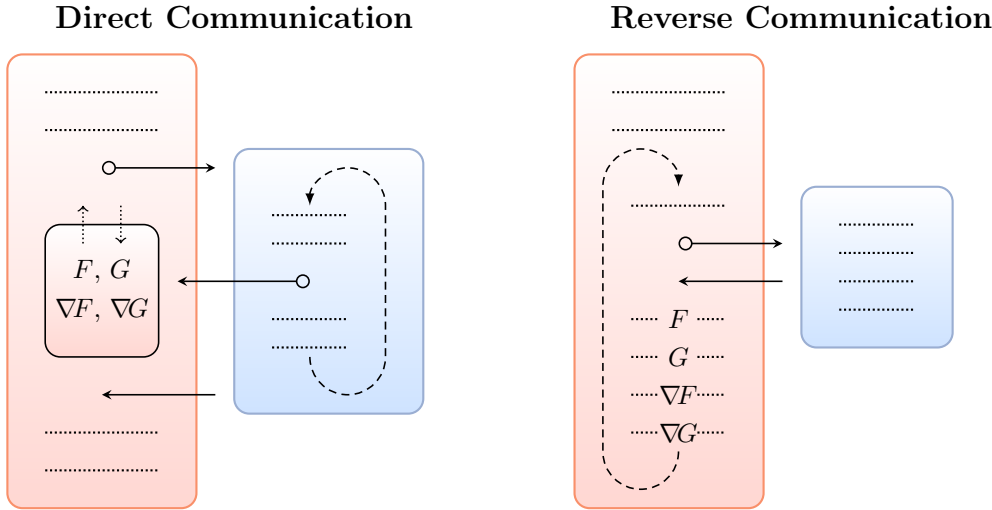


Figure 2.1.: Schematic view of Direct vs. Reverse Communication in an NLP solver: Here represents code, $\circ \rightarrow$ marks a function call and $---\rightarrow$ shows the location of the major iteration loop. We can clearly observe that the RC approach has to define fewer interfaces, and that no artificial partition between the calling program and the user functions has to be introduced by wrapping them into a subroutine that is callable by the external NLP solver.

Where Direct Communication receives control from the caller and only relents it whenever it seems fit (hopefully when the solver converged towards an optimal solution), Reverse Communication leaves control in the callers hand: The “Fire-and-Forget” is replaced by repeated calls to the solver, which performs some limited task only to return to the caller with instructions on what to do next. In the context of an NLP solver, the caller will repeatedly be asked to evaluate the objective and constraints and to make the updated values available to the solver, that will then carry on with the current iteration and return to the caller as soon as another user action is necessary, or some quantity needs to be updated.

Figure 2.1 showcases the main technical difference: The major iteration loop is “outsourced” to the caller, who can therefore exert some level of control over the iterative process. When

³It seems fair to note that WORHP is neither the only nor the first NLP solver using Reverse Communication – NLPQLP[62] probably is. However, WORHP appears to be the first NLP solver that fully utilizes the potential of RC by coupling it with the USI approach in section 2.3.

2. Architecture of WORHP

combined with the USI approach, that lays bare all solver data, Reverse Communication vastly increases the analysis and intervention options open to the caller. Given sufficient know-how and effort, the caller can bypass or even replace whole sections of the default iteration process, or just content himself with in-depth inspections of even the most obscure solver internals.

The Reverse Communication paradigm applied to WORHP results in the following changes when compared to traditional NLP solvers:

- The major iteration loop is built and controlled by the caller, not by WORHP; the solver *suggests* to continue the iteration or terminate it, but the caller controls it.
- WORHP communicates with the caller through so-called *user actions* that request him to evaluate a function, produce some kind of iteration output or perform no task at all and just call the WORHP NLP routine again.
- No solver-callable functions need to be provided by the user—the caller just needs to ensure that specific quantities are computed (in whichever way) whenever WORHP requests them. WORHP never calls any user-provided function at all.
- The caller can closely monitor the iteration process, override the default output and intervene as necessary.
- The caller is able to interfere with the NLP solver on a low-level algorithmic level, instead of being limited to controlling a handful of solver parameters (such as `MaxIter`). It is technically possible for sufficiently knowledgeable users to replace the QP solver, the line search or any other component of WORHP by their own code, without so much as touching its source code.

We should note that, even though these changes are substantial, users are *free* to, but not *forced* to actively use the additional freedom granted by Reverse Communication. The complexity of using WORHP like a standard Direct Communication solver is not significantly higher, and can be completely negated by hiding the RC interface behind a simpler DC interface. Currently, WORHP implements two such interfaces, one of them a DC+USI one (i.e. hiding the Reverse Communication loop behind the interface and accepting function pointers, but still using the high-level data structures), and the other one resembling a legacy NLP solver interface with the *many arguments* (see section 2.3) convention.

2.4.1. Division into *stages*

WORHP uses the terminology *stages* to describe the single steps between two Reverse Communication solver calls. Each major iteration passes through at least 10 Stages; possibly more if a recovery strategy is activated, the constraint relaxation penalty is increased, or if the trial step size $\alpha_0 = 1$ is rejected and actual line search begins.

A striking feature of WORHP's division of a major iteration into stages is the fine granularity, i.e. the fact that many intermediate stages do not require any user action, making them redundant during normal operation. This granularity is a deliberate design decision that is guided by the canonical steps of the NLP algorithm and aims at

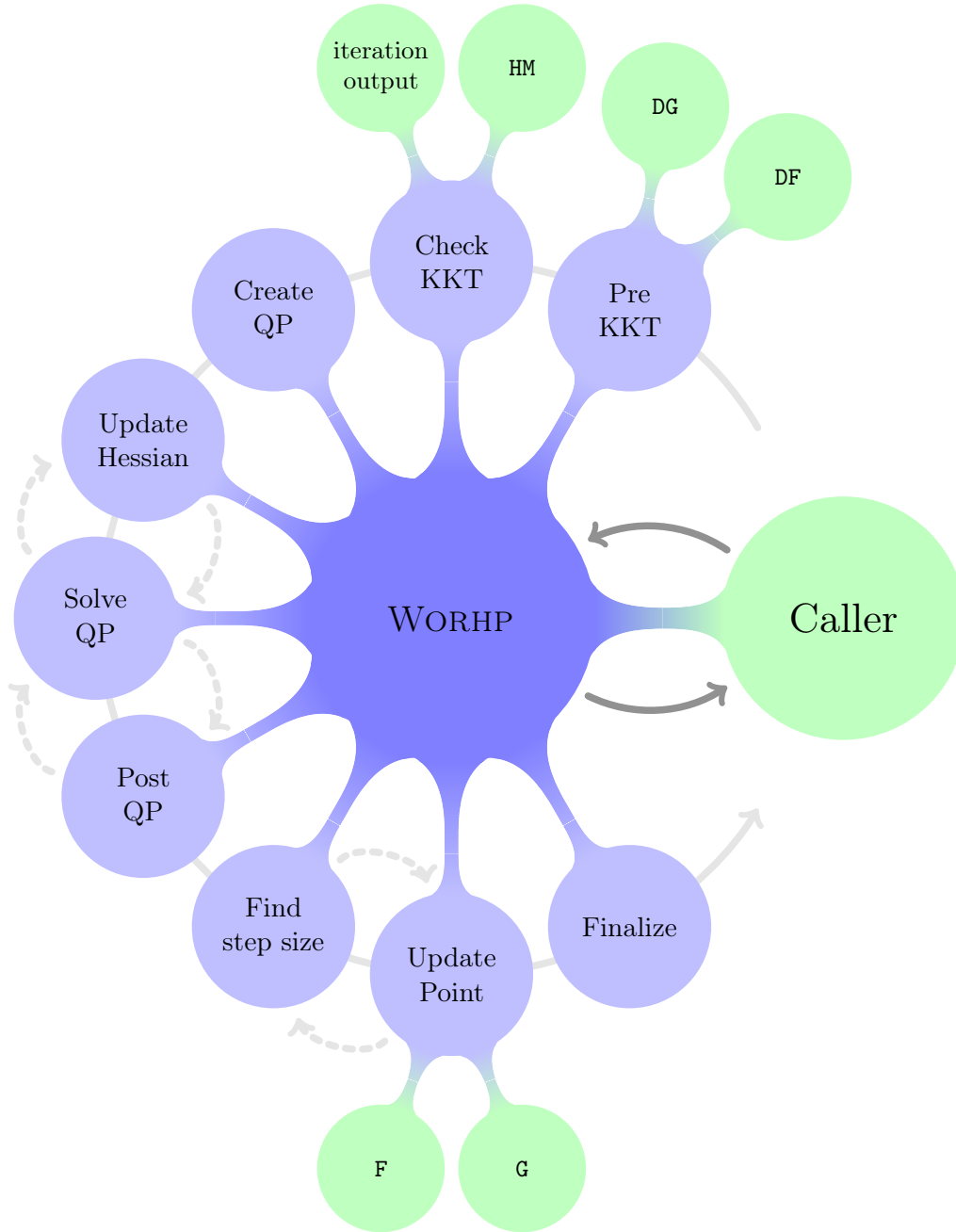


Figure 2.2.: Graphic representation of WORHP’s major stages. The stages are traversed in counter-clockwise order, with optional loops or jumps between them. Every stage returns back to the caller, who then calls WORHP again to activate the next stage. The green 3rd-level nodes denote actions (mostly evaluations) that are required of the caller before transitioning to the next stage.

exposing these steps to potential caller intervention. This design decision also accounts for simplifying the solver-internal control flow, since many stages may jump to other (hitherto unmentioned) non-default stages to recover from errors or switch the solver into

different operational modes. The current design with frequent entry and return allows for a clean implementation of this complex control flow without having to resort to mutually entwined sequences of `goto` and jump marks at arbitrary code points.

2.4.2. Implementation considerations

Since most users will be content with the default behavior of the solver, the more extreme scenarios of interventions are limited to few occasions, and the overhead of Reverse over Direct Communication is of interest to a majority of the users. The implementation thus has to strive to keep this overhead low.

Since the caller repeatedly polls for user actions and solver status, the implementation needs to provide simple and efficient means to this end. There exist software packages communicating through RC that approach the flow control and user action issue through a single status integer whose value encodes the solver status, user actions and the current or next stage of the solver. While this approach is indeed simple, its Achilles heel is the coupling of unrelated entities. WORHP can currently request the user actions

1. **Call `worhp()`**

This user action is needed because the RC finite difference routine temporarily “switches off” WORHP and takes control of the RC loop itself to request function evaluations.

2. **Produce iteration output**

The combination of USI and RC enables users to provide their own output routines, for instance because they prefer the clunky output of SNOPT, or because the output is further processed by text parsers that expect a specific format. Since this action is flagged exactly once per major iteration, it can also serve for user-specific logging, to flag a repaint of a graphical representation of the progress, or any other action that is performed once per major iteration.

3. **Evaluate F**

4. **Evaluate G**

5. **Evaluate DF**

6. **Evaluate DG**

7. **Evaluate HM**

The above flags prompt the caller to evaluate one of the user-defined functions, or to update a derivative of them. It is important that the derivative flags are not raised, if the user chooses to have them approximated by finite differences or one of the BFGS methods.

8. **call finite difference routine** to compute DF

9. **call finite difference routine** to compute DG

10. **call finite difference routine** to compute HM
11. **call finite difference routine** (*catch-all* of the above)

The finite difference routine must also communicate through RC, and thus has a set of user actions. The first three actually confound solver status and user action, but the *catch-all* is used to hide this from the user.

In addition to these 11 user actions, WORHP has 49 distinct statuses [sic] and 16 stages which allows a total of 2^{76} combinations (although most of no practical relevance), which overflows the range of common integer types. Even if the number of combinations were much lower, it is a maintenance, book-keeping and documentation burden to keep track of an extensive list of integers and their respective meanings. The obvious solution is to split status, stage and user action, and to introduce means to set, poll and reset them.

An NLP solver status (“iterating”, “error”, “optimal solution found” etc.) can aptly be represented by a simple integer number, since it cannot attain two distinct statuses at any time. This remains true for WORHP, but it can attain a *superposition* of statuses, if it terminates inside “acceptable” tolerances due to some error or limit condition—the acceptable status is then superposed with the original stopping condition. Named constants are defined by WORHP to avoid the use of magic numbers in user and library code to set or check for the status of the solver, for instance for the stopping criterion of the RC loop (Fortran version):

```
DO WHILE (cnt%status < terminateSuccess .AND. cnt%status > terminateError)
```

which is –even without any comment or external documentation– significantly more descriptive through the use of `terminateSuccess` and `terminateError` than the equivalent condition

```
DO WHILE (cnt%status < 1000 .AND. cnt%status > -1000)
```

by inserting the numeric values of the constants. Furthermore, the version using named constants continues to function (after recompiling), even if the actual numeric values change, while the magic number version will break.

WORHP stages can be represented by an integer, as well, but it may be beneficial for the solver to be able to look *back* at previous stages, if the current behavior depends on which stages were traversed before, and also to plan *ahead* for more than one single stage. The appropriate structure to provide this functionality is a ring buffer, whose maximum length is determined by the desired planning and retrospection horizons. Since a ring buffer is (slightly) more complex to handle than integer flags, getter and setter routines are needed; WORHP provides `GetStage` and `SetStage` routines that query or schedule stages and abstract from the ring buffer implementation. For the actual representation of the stages, WORHP defines named constants of the form `Init_SQP`, `Check_KKT` etc. By convention, stage constants are formed with underscores, while status constants such as `OptimalSolution` or `TerminatedByUser` are composed according to the CamelCase convention.

2. Architecture of WORHP

Finally, the user actions require a representation that allows them to be set and cleared independently of each other. As for the stages, WORHP provides getter and setter routines `[Add|Set|Get|Done]UserAction` that enable all necessary operations on user actions and abstract them from the implementation; the caller will only use `GetUserAction` to query a user action flag, and `DoneUserAction` to reset it as indicated in listing 2.2.

```
if (GetUserAction(cnt, evalF)) {  
    /* Update opt->F here */  
    DoneUserAction(cnt, evalF);  
}
```

Listing 2.2: User action query and reset, shown for the “evaluate F” case. The token `evalF` is another example of a named constant defined by WORHP.

One important benefit of working with these flags is the fact that WORHP is able to check whether all user action flags are reset, and is thus able to raise a warning, if one or more flags have *not* been reset, informing the user about a potential mistake. The simple approach with tests of the form

```
if (iFlag == 123) {  
    /* evaluate F */  
}
```

lacks this ability and will leave this type of user error unnoticed.

2.4.3. Applications

The interactive mode (see section 3.3) is a solver-internal application of the Reverse Communication architecture.

The *Sentinel* project (running from 2007 to 2010) was another application that aimed at providing a WORHP-specific external analysis tool that used the transparency provided by RC and USI to perform on-demand analysis of the solver data, in particular matrix conditions, scaling and other numerical issues and provide recommendations or perform real-time interventions, while the solver is running. Since a number of analyses and interventions could actually be performed in real-time, great portions of the project were merged into WORHP, for instance the recovery strategies (see section 1.3.6) that are partially responsible for WORHP’s above-average robustness in many test scenarios.

2.5. Serialization

The serialization module is a natural extension of WORHP's data handling: It enables serialization of the data structures to a file, and vice versa, deserialization from a file back to the data structures, i.e. it defines a one-to-one mapping between the in-memory representation of the C `structs` used by WORHP and a file on disk.

Since WORHP is stateless by design, this allows to *completely* capture the solver state.

2.5.1. Hotstart functionality

The most prominent application of the serialization module is the *Hotstart* functionality of WORHP, which allows to save and restore an optimization process. To this end, the solver is interrupted between any two stages (see section 2.4.1) and terminated after serializing its state. The state file can later be loaded by WORHP, producing bit-precise restorations of its state, which enables *seamless continuation* of the previously interrupted iteration. This seamless continuation is the defining capability of the Hotstart functionality and sets it apart from the warm start feature of other NLP solvers, which mostly includes restoring the BFGS matrix and possibly the multipliers, but disregards the significant internal state, such as penalty parameters (when using a merit function), the filter entries (when instead using a filter), best-so-far solution etc. If the Hotstart file is platform-independent (which it is, as outlined in section 2.5.3), users can migrate an optimization run between different computers and architectures, for instance from Linux to Windows, from 32-bit systems to 64-bit ones, from PowerPC to Itanium. Hotstart files are written by `WorhpToXML` and `HotstartSave` (Fortran/C), and read by `WorhpFromXML` and `HotstartLoad`. The routine names differ, because the C functions are wrappers around the Fortran routines that make heavy use of optional arguments, which is not readily portable to C using the established Fortran standard.

2.5.2. Reading parameters

In addition to the complete Hotstart, it is possible to perform an *incomplete* Hotstart by using only partial information; this may consist of a single data structure only or of a genuine subset of components, leaving the missing ones at default values. The obvious (and historically the first) application of this option is to outfit WORHP with a parameter file from which to read its parameters at runtime. This parameter file is technically an incomplete Hotstart file, which contains an image of the `Params` data structure only. The file may even be incomplete, in the sense of omitting some components, if one wishes to set only few, instead of all parameters, to specific values. This is enabled by first using the default initializer, followed by checking the presence of every component in the parameter file, overriding the initialized default value. Fortran callers can use the `WorhpFromXML` routine for reading incomplete Hotstart files (using optional arguments, as laid out earlier), while C callers are provided with the tailored `ReadXML` function.

2. Architecture of WORHP

One user-driven extension to the parameter reading is the ability to provide default parameters that differ from WORHP's defaults, while still allowing a parameter file to override them. Since the usual initialize-override approach is atomic from the caller's perspective, it cannot provide this functionality, because the initialization step overrides any caller-specific defaults. The solution is to separate both steps into two operations. WORHP provides a new `InitParams` routine to perform the initialization step, and non-initializing parameter reading routine for the second; the Fortran `WorhpFromXML` is extended by an optional `noInit` flag, while C callers use the new `ReadParamsNoInit` function.

2.5.3. Serialization format

The choice of a serialization format was guided by the following constraints:

- Readability:** Hotstart and, more importantly, parameter files should be easily human-readable, and editing them should be possible without imposing rigid formatting rules on users. This precludes the use of binary formats, even though they are more efficient.
- Portability:** Both Hotstart and parameter files must not be platform-dependent; portability of the Hotstart file between platforms is an explicit feature. The same is valid for parameter files, since the solver is supposed to show consistent results⁴ when running the same problem with the same parameter on different platforms. This, again, discourages the use of binary formats.
- Standardization:** If possible, the serialization format should conform to an established standard. This way, external tools will be available to edit or perform more sophisticated operations on the serialization files. Availability of syntax highlighting in an editor is tangible example for this demand.
- Efficiency:** With WORHP being designed for large-scale optimization, Hotstart files potentially need to hold a considerable amount of data. The serialization format should cause as little overhead as possible, although this goal contradicts readability and portability, since it essentially calls for a binary format.

Possible candidates are JSON, YAML and XML. Despite their differences, for instance in supporting types other than strings, none is significantly superior to the others for our purposes. Two arguments initially supported the choice of XML as WORHP's serialization format:

- XML is rather robust, whereas whitespace changes can significantly change the interpretation of (read: break) YAML data.

⁴Demanding *equal* behavior between platforms is unrealistic, since different processors, compilers and math libraries have subtle influences on the results of floating point arithmetic; instead we can realistically only demand consistent results and differences in the order of 1 ulp, unless such errors accumulate.

- Most, if not all, unixoid operating systems come equipped with *libxml2*, which seemed to facilitate portability in the initial development.

While the robustness argument is still valid, subtle differences between Linux distributions (most noticeable symbol versioning), make short work of portability. The internal XML parser presented in section 3.5 has therefore been added in late 2012 to cut dependencies on external tools. This in turn invalidates the second argument and opens the road to considering an alternative serialization format, since the XML grammar (and the resulting parser) is quite intricate and complex.

Technical Implementation

The compiler does not change its operation to conform to the meanings implied by your variable names.

(Quip on GCC Bugzilla)

3.1. Hybrid implementation	55
3.1.1. Interoperability issues	55
3.1.2. Interoperability improvements	59
3.1.3. Data structure handling in WORHP	63
3.2. Automatic code generation	68
3.2.1. Data structure definition	69
3.2.2. Definition file	70
3.2.3. Template files	72
3.2.4. Applications of automatic code generation	73
3.3. Interactive mode	76
3.4. Workspace Management	79
3.4.1. Automatic workspace management	79
3.4.2. Dedicated dynamic memory	81
3.5. Internal XML parser	85
3.6. Established shortcomings, bugs and workarounds	87
3.6.1. Intel Visual Fortran 10.1	87
3.6.2. Windows	87
3.7. Maintaining compatibility across versions	89

3. Technical Implementation

This chapter is concerned with the (software-)engineering details needed to implement the architecture and functionality outlined in the previous chapters.

The main aspects under consideration are:

- the mixed-language (*hybrid*) implementation of WORHP, which aims at harvesting the benefits of both C and Fortran and facilitates interfacing with both languages in a “natural” way;
- the automatic code generation, which became a technical necessity as the size and technical complexity of WORHP increased and has since then been established and extended to a convenient tool for relieving developers of many onerous tasks;
- the workspace management used by WORHP to handle dynamic memory requirements;
- the technical infrastructure to build the various solver interfaces for a wide range of configurations and target platforms, in a mostly automatic way, and to perform other maintenance or technical tasks.

3.1. Hybrid implementation

WORHP’s technical design revolves around the central decision to base its data management on a small set of rich `struct`-like data structures, instead of the conventional approach of keeping and communicating data in a potentially large collection of arrays and (array, size) pairs or, worse even, partitioned integer and real workspace arrays. It is further stipulated that interoperability with C/C++ should be possible and simple, without incurring any unjustified overhead –neither for the C/C++ nor the Fortran components– and that an implementation should be deployable on all major platforms.

3.1.1. Interoperability issues with legacy FORTRAN

Consistently separating words by spaces became a general custom about the tenth century A.D., and lasted until about 1957, when FORTRAN abandoned the practice.

(Sun FORTRAN Reference Manual)

The canonic choice for the data structure design is to base it on C `struct`, and reproduce its layout with Fortran constructs. If restricted to pre-2003 Fortran constructs, this approach is severely hampered by software portability and memory layout issues, but enjoys standardized language support starting with the 2003 standard and further extensions in the 2008 standard.

We will first consider legacy, i.e. pre-90 FORTRAN dialects to illustrate the technical issues involved, and then move on to the 90/95 standard and further to the substantial interoperability improvements of the most recent (2003 and 2008) Fortran standards.

Available Fortran constructs

Some FORTRAN dialects include the `RECORD` or `STRUCTURE` keywords, which are supported by some compilers, but being non-standard, unsupported by others. `COMMON` blocks

```
STRUCTURE /parameters/
  REAL*8    tol
  INTEGER*4  maxiter
  LOGICAL*1  switch
END STRUCTURE
```

Listing 3.1: Example of a non-standard Fortran data structure

offer a standard pre-90 alternative, but do not offer the conciseness of the `struct`-like component access.

3. Technical Implementation

Reliable and maintainable use of `COMMON` blocks typically requires crutches such as storing their definitions in a central definition file included by all routines that access the block.

Example `vector.inc`:

```
INTEGER      N, NMAX
PARAMETER   (NMAX=100)
REAL        X(NMAX)
COMMON /VECTOR/ V, N, NMAX
```

Example use:

```
SUBROUTINE A(I, X)
  INTEGER I
  REAL X
  INCLUDE vector.inc
  IF (I .GT. 0 .AND. I .LE. NMAX) V(i) = V(i) + X
END SUBROUTINE
```

With the advent of Fortran 90, *user-defined types* using the `TYPE` keyword were introduced as a standard-conforming replacement of the legacy `RECORD` and `STRUCTURE` constructs. This allows writing standard-conforming Fortran code using an analogue of the C `struct` type.

To achieve interoperability between a C `struct` and its companion Fortran `TYPE`, one has to restrict the component types to *interoperable* ones and precisely harmonize their *memory layouts*.

Type interoperability

Type interoperability is usually unproblematic if the components are restricted to primitive scalar types and arrays of them; types like C `int` will virtually always correspond to Fortran default `INTEGER`, which is a 4-byte signed integer type on almost all current, non-exotic platforms¹, although the C standard only requires it to have at least 2 bytes[38, §5.2.4.2.1].

Other C types like unions and bit-fields are not interoperable as such, since Fortran has no equivalent types for them, although a specific “instance” of a union type may in principle interoperate with Fortran², and bit-fields may be mapped to integers, which can either be manipulated and inspected by corresponding integer operations, or more expressively by the `IBITSF95`, `IBSETF95`, `IBCLRF95`, `IANDF95`, `IORF95` and `IEORF95` intrinsics.

¹Many assumptions like these hold almost universally nowadays, because current non-embedded computers are –with few exceptions– based on the x86 architecture. Special case handling in legacy code is an indication that things were much more diverse and complicated when platforms like VAX, PDP-11, System/360 or Crays were in common use for scientific computing.

²Consider the union `{int i; double d;}`; on most platforms, this union will be accessible by a single `REAL(8)` or a 2-element array of default `INTEGER`; which of these will hold `i` is implementation-dependent – it is safe to assume that every sane compiler will align `i` to 4 bytes, so it *will* either the upper or lower word.

The underlying size of the bit-field is a potential issue, but may be controlled by using dummy bits to force the compiler to pick a certain (minimum) size.

Problems may surface on uncommon platforms, though, since both the C and the Fortran standard impose few mandatory requirements on data types, such as minimum sizes or numeric ranges, and leave many details to the implementation. Therefore, consistency between many platforms exists by *convention*, but not by *standard*. This is rather unsatisfactory for portable software, which should be firmly based on mandatory aspects of the standards, instead of common, but not ubiquitous conventions.

Even common types like C `long` lack a portable mapping to Fortran, because its size depends on platform and bitness: `long` is a 4-byte integer on 32-bit Linux systems and all Windows systems, but an 8-byte integer on 64-bit Linux systems; any mapping to a Fortran type is thus platform dependent.

Fortran integers always being signed[40, §4.4.1], there is, strictly speaking, no interoperability with C's `unsigned` integer types. In practice, however, a Fortran integer *can* interoperate with its companion C `unsigned` integer type on a subset of its values, namely those, where the MSB is 0, i.e. on the “lower half” of its unsigned value range. This works because the *two's-complement* representation of signed integers for non-negative values is identical with that of unsigned integers, as shown by the following two remarks.

Definition 10 (Two's-complement). In **two's-complement** N bits $a_1 \dots a_N$, with $a_i \in \{0, 1\}$, $i = 1, \dots, N$ represent the natural number

$$\sum_{i=1}^{N-1} a_i 2^{i-1} - a_N 2^N.$$

The sequence $a_1 \dots a_N$ is called N -bit signed integer (in two's-complement representation). a_1 is called the least significant bit (LSB) and a_N is called most significant bit (MSB), irrespective of the physical storage order, which depends on the endianness.

Corollary 11. An N -bit signed integer in two's-complement representation has a numeric range of $[-2^N, \dots, 2^N - 1]$. The range is skewed towards negative numbers because it has one and only one representation of zero (namely $0 \dots 0$).

Its representation coincides with that of an N -bit unsigned integer for values in $[0, \dots, 2^N - 1]$. \square

Corollary 11 thus guarantees interoperability of Fortran `INTEGER` types with their companion C `unsigned` types for “sufficiently small” values. Should the `unsigned` value on the C side exceed the safe threshold, the Fortran side can easily detect this, since its value will be interpreted as a negative number.

It is worth mentioning that the C standard [38, §6.2.6.2(2)] actually leaves three options for the binary representations of signed integers to the implementation:

3. Technical Implementation

- a) *sign-and-magnitude*: bits $a_1 \dots a_{N-1}$ are value bits and the *sign bit* a_N is a genuine sign bit, analogous to the IEEE floating point data types.
- b) *one's-complement*: the sign bit has value $-(2^N - 1)$.
- c) *two's-complement*: the sign bit has value -2^N as described in Definition 10.

Since options a) and b) have computationally relevant issues (two representations of zero and non-trivial addition), modern platforms exclusively use c) for integer representation.

Memory layout

Matching the structures' memory layout is, by far, the more problematic issue with interoperability. A structure maps its components into memory by defining for each component an offset from the beginning of a structure instance, as illustrated by figure 3.1. Therefore, the memory layout between a C `struct` and its companion Fortran `TYPE` *matches*, if both structures have components of pairwise equal storage sizes and share the same offset mapping.

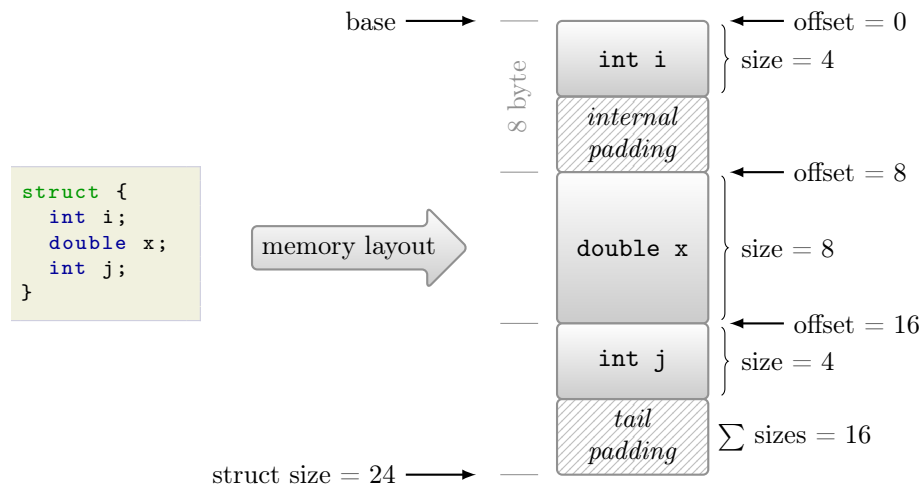


Figure 3.1.: Schematic of data structure memory layout, assuming *mandatory* 8 byte alignment. Members are accessed in memory by $\langle \text{base address} \rangle + \langle \text{offset} \rangle$. The alignment requirement (or rather the blunder in the ordering of the components) causes 25% overhead for padding due to the strict specifications of the C standard.

The C standard requires that these offsets are increasing in the order of the components, but the Fortran standard does not prohibit reordering of the structure components; this can be inhibited by using the `SEQUENCE` attribute to the `TYPE` definition[40, §4.5.1.2]. `COMMON` blocks may not be reordered[40, §5.5.2.1(1+2)] by the Fortran compiler³, which is why it is customary among experienced Fortran developers to sort its components by size to minimize the amount of padding added by the compiler: Both C[38, §6.7.2.1(12-13)] and Fortran[40, Note 4.20] compilers may add unnamed padding between adjacent

³This “allows” tricks such as accessing `IA(1)` in `COMMON/IA(2),IB(2)/` through `IB(-1)`.

components (“internal padding”) and padding behind the last element (“tail padding”) to maintain any implementation-dependent alignment requirements or suggestions (for performance reasons); figure 3.1 demonstrates both types of padding. Double precision numbers, for instance, should be aligned along 4-byte boundaries according the 386 ABI, while the AMD64 ABI suggests 8 byte alignment[37, Figure 3-1].

Determining the padding requires some degree of reverse engineering, because compilers do not provide information on the chosen memory layout. If the padding can be determined, modern C compilers offer mechanisms to enforce alignment (and thus implicitly the padding) down to **struct** component level, but these mechanisms are not covered by the standard and are thus compiler-dependent: the GNU compiler understands the `__attribute__((aligned(N)))` syntax or `-malign` switches, while Windows compilers use `#pragma pack` or `__declspec(align(N))` and variations on the `/ALIGN` switch. Other compilers may or may not offer interventions at this level, and probably add further syntax options to the above collection.

We can conclude that it probably *is* technically feasible to create one-to-one mappings between C **struct** instances and Fortran constructs with two common compiler families, but the memory layout issues are very difficult to tackle and not very portable. Basing a central aspect of the solver design on such fragile constructs is emphatically ill-advised.

The *Chasm Language Interoperability Tools*[59] were created to bridge this “chasm” between C and Fortran, which is useful for legacy compilers such as `g77`, `ifort` prior to version 10 and `gfortran` prior to version 4.3. Chasm is essentially left obsolete, however, by the interoperability features of Fortran 2003 (cf. [40, Chapter 15]) and later, which are (in essence) supported by `ifort` ≥ 10 or `GCC` ≥ 4.3 .

3.1.2. Interoperability improvements in newer standards

With the advent of Fortran 2003, the standardizing bodies acknowledged the importance of interoperability with C. Beyond providing standardized means for operating with C programs, the tools provided by the standard can also serve as a least common denominator for interfacing with other languages that have C interoperability features.

Fortran 2003 addresses both type interoperability and memory layout issues:

The standard-conforming way to select the Fortran type corresponding to a given interoperable C type is to use the `KIND` constants defined by the intrinsic `ISO_C_BINDINGF03` module.

Note that most compilers adhere to the “`KIND = number of bytes`” convention where `C_INTF03` is 4 and `C_DOUBLEF03` is 8, but neither does the standard require or advocate this convention, nor is it safe to rely on it: the NAG compiler, for instance, uses `KIND` constants 1, 2 and 3 for `REAL` types, instead of the conventional 4, 8 and 10/16⁴.

⁴Although the standard *does* allow this divergence from the widely established convention, many practitioners consider NAG’s choice as “silly”.

3. Technical Implementation

C type	Fortran 2003 type	Note
int	INTEGER(KIND=C_INT)	4 with most compilers
long	INTEGER(KIND=C_LONG)	4 or 8 with most compilers
size_t	INTEGER(KIND=C_SIZE_T)	size_t is unsigned!
float	REAL(KIND=C_FLOAT)	4 with most compilers
double	REAL(KIND=C_DOUBLE)	8 with most compilers
char	CHARACTER(KIND=C_CHAR)	see note on string interoperability
_Bool _{C99}	LOGICAL(KIND=C_BOOL)	

Table 3.1.: Interoperable scalar types defined by ISO_C_BINDING_{F03} (incomplete list).

The standard-conforming way to match the memory layout of a user-defined TYPE is to add the BIND(C) modifier to its definition, as shown in listing 3.2. In words of the standard:

A Fortran derived type is interoperable with a C struct type if the derived-type definition of the Fortran type specifies BIND(C), the Fortran derived type and the C struct type have the same number of components, and the components of the Fortran derived type have types and type parameters that are interoperable with the types of the corresponding components of the struct type. A component of a Fortran derived type and a component of a C struct type correspond if they are declared in the same relative position in their respective type definitions. [40, §15.2.3]

By adding the BIND(C) modifier we instruct the Fortran compiler to

- check all structure components for their interoperability,
- not reorder the structure components, as if SEQUENCE were in effect,
- use the same padding as the companion C compiler for the equivalent C struct.

Using the ISO_C_BINDING_{F03} features, we can thus create Fortran user-defined TYPEs that have a matching C struct, although both languages have non-interoperable constructs for which there is no equivalent construct in the respective other language. Examples include union and bit-fields in C, and ALLOCATABLE or POINTER types in Fortran.

```
TYPE, BIND(C) :: MyInteroperableType
  REAL(KIND=C_DOUBLE) :: X(3)
  INTEGER(KIND=C_INT) :: N
END TYPE MyInteroperableType
```

Listing 3.2: Example of a C-interoperable Fortran type

Interoperability of strings is not as straightforward as the one-to-one mapping between the numeric types, since they are “first-class citizens” in Fortran. The LEN attribute gives them a fixed length, such that no 0-termination is necessary, whereas in C, strings are (pointers to) character arrays, and their (string-)length is defined by the position of


```

struct MyInteroperableStruct {
    double X[3];
    int n;
}

```

Listing 3.3: C struct corresponding to the Fortran type in listing 3.2.

the 0-character. As component of a user-defined type, a `CHAR(KIND=C_CHAR, LEN=N) :: string` may be mapped to a `char string[N+1]` when care is taken of 0-termination and the undefined characters between the logical and the physical string length are initialized to blanks:

- If 0-termination is not maintained by the Fortran side, C routines accessing the string may read past its end, resulting in trailing garbage characters in the output and possible program termination by the OS due to illegal memory access.
- If the undefined intermediate characters are not set to blanks by the C side, Fortran routines accessing the string will use them until all `LEN=N` characters have been processed, resulting in garbage characters in the output.

Fortran can add 0-termination by setting `string(L+1:L+1) = C_NULL_CHARF03` (where $L \leq N$ is the number of characters; notice the slice-like syntax needed for accessing a single character, since Fortran strings are syntactically *different* from arrays of characters, although their in-memory representation is identical).

Since Fortran types with `ALLOCATABLE` or `POINTER` attribute are not (yet⁵) interoperable, dynamic arrays have to be implemented on the C side. Fortran can interoperate with C pointers through the `TYPE(C_PTR)F03` type, defined by the `ISO_C_BINDINGF03` module. Allocated C pointers cannot be directly accessed by Fortran, since Fortran arrays and pointers in addition to their type also have extent information, which C pointers are lacking. To create a Fortran pointer from an allocated C pointer, the `C_F_POINTERF03` routine takes a `TYPE(C_PTR)F03`, extent information and initializes a Fortran pointer (note that `TYPE(C_PTR)F03` is type-agnostic, hence the Fortran pointer type has to match the C pointer type, or accessing it may result in undefined behavior).

Interoperable functions

With *types* being interoperable, the next step is to consider interoperability of *functions*. While Fortran differentiates between functions and subroutines, from a C point of view the latter is the same as the first with no return value, i.e. `void`, hence we only need to consider functions (in C sense) without worrying about special treatment for (Fortran) subroutines.

⁵TR 29113 of JTC1/SC22/WG5 introduced `ISO_Fortran_binding.h` as header file provided by the companion C compiler to define an opaque Fortran array-descriptor type and macros to access the Fortran array and dimensions. Unfortunately, this part of the technical report was *not* adopted by the standardizing body and thus remains in limbo.

3. Technical Implementation

For a function to be interoperable, all of its arguments have to be interoperable and the types and order of function arguments have to match. Again, the `BIND(C)` modifier instructs the compiler to check for interoperability and behave cooperatively towards its companion C compiler. An important additional feature for functions is the ability to specify their assembly name – this is important, because Fortran compilers generally perform name mangling, which not only differs between compiler vendors, but is also influenced by the use (and naming) of encompassing modules; table 3.2 shows the somewhat diverse name mangling conventions used by a selection of different compilers. Using the `BIND(C, name="assembly_name")` modifier instructs the compiler to use the specified `name` as assembly name, instead of the result of the compiler-specific name mangling scheme. This greatly simplifies linking to C components in a portable way, since the assembly name can be chosen freely and works independently of the current compiler; it can even be used to make existing C functions (such as `stdlib` functions) available to Fortran – a technique employed by WORHP’s workspace management and described in detail in section 3.4.

SUBROUTINE `foo_bar()` becomes ...

compiler	freestanding	inside module <code>fred</code>
<code>g77/f2c</code>	<code>foo_bar__</code>	—
<code>gfortran</code> (Linux)	<code>foo_bar_</code>	<code>__fred_MOD_foo_bar</code>
<code>gfortran</code> (MinGW old winapi)	<code>_foo_bar_</code>	<code>___fred_MOD_foo_bar</code>
<code>gfortran</code> (MinGW new winapi)	<code>foo_bar_</code>	<code>__fred_MOD_foo_bar</code>
<code>ifort</code> (Linux)	<code>foo_bar_</code>	<code>fred_mp_foo_bar_</code>
<code>ifort</code> (Windows)	<code>_FOO_BAR</code>	<code>_FRED_mp_FOO_BAR</code>

Table 3.2.: Examples of Fortran name mangling with different compilers.

A minor complication is the fact that the so-called “argument association” of Fortran is best described as *call-by-reference* [40, §12.4.1], whereas C operates on local copies of all arguments, i.e. *call-by-value* [38, §6.5.2.2(4)]—this explains why the following (invalid) FORTRAN program prints 2 when compiled with some legacy compilers:

```
PROGRAM INVALID
  CALL PLUS(1)
  PRINT *, 1
END PROGRAM

SUBROUTINE PLUS(I)
  INTEGER I
  I = I+1
END SUBROUTINE
```

(this will not work with modern compilers, since they perform more complete checks on arguments, and furthermore put the integer constant 1 into a read-only data section [ELF: `.rodata`], hence the program is killed by the OS with a memory access violation when `PLUS` tries to increment the value at this address.)

The standard way of interoperating with Fortran is to have C pass all arguments by reference, i.e. by their address, even scalar arguments like integers or floating point numbers. To pass literals or the results of a function call or an arithmetic operation to Fortran routines, one has to explicitly introduce dummy variables, since Fortran accesses arguments by their addresses, which are undefined in the above cases.

The Fortran 2003 standard offers a convenient alternative to the usual “everything-pointer” approach by introducing the `VALUEF03` attribute that basically instructs the Fortran compiler to mimic the *call-by-value* behavior of C. The following code snippet demonstrates its use by defining a Fortran subroutine and a compatible C declaration to interface with it:

```
SUBROUTINE interoperable(X, N) BIND(C, name="interop")
  USE, INTRINSIC :: iso_c_binding
  INTEGER (C_INT), VALUE :: N
  DOUBLE (C_DOUBLE), INTENT(in) :: X(N)
  ...
END SUBROUTINE interoperable

/* C interface for the subroutine above */
void interop(const double *x, int n);
```

To interoperate with existing C functions, an `INTERFACEF90` needs to be defined, specifying the arguments and the assembly name of the C routine. The following example shows an interface for the `realloc` function of the C standard library:

```
INTERFACE
  TYPE (C_PTR) FUNCTION c_realloc(p, n) BIND(C, name = "realloc")
    USE, INTRINSIC :: iso_c_binding
    TYPE (C_PTR), VALUE :: p
    INTEGER (C_SIZE_T), VALUE :: n
  END FUNCTION c_realloc
END INTERFACE
```

3.1.3. Data structure handling in WORHP

With the technical aspects covered by the previous sections, we will now address the details of WORHP’s data structure handling. The aim of this “handling” by the solver back-end is to provide properly initialized data structures, workspace and manipulation functions to both the caller and the (numeric) main components of the solver.

The basic steps in a normal solver run are as follows:

1. The caller creates instances of the data structures.
2. *optional*: The solver parameters are initialized to default values, or to custom values by reading an XML parameter file.
3. The caller specifies the problem dimensions.

3. Technical Implementation

4. The back-end initializes all data structures, allocating storage for variables, matrix structures and workspace, and setting all counters, flags etc. to default values.
5. The caller specifies initial values, bounds and the sparse matrix structures (if available), using the previously allocated storage.
6. The caller starts the solver.
7. The solver, on its first call, uses the workspace handling routines to request a number of workspace slices, depending on the dimensions and other input of the caller, and continues its normal operation afterwards.
8. The solver requests additional workspace slices during its operation, either permanently (*on-demand*) or for temporary storage.
9. Upon reaching a termination criterion, the solver finally returns to the caller, who will usually save or otherwise process the results.
10. The back-end frees all allocated storage. The data structures can now be discarded or reused for another run, starting at 2.

1: Create instances

This will usually just consist of statically creating instances through

```
TYPE (OptVar)      :: opt
TYPE (Workspace)   :: wsp
TYPE (Params)      :: par
TYPE (Control)     :: cnt
```

```
OptVar      opt;
Workspace   wsp;
Params      par;
Control     cnt;
```

It is also possible in both languages to create pointers and allocate them at runtime. A technical complication of both approaches is the default initialization of the `initialized` flag in every data structure that serves as a safeguard against overwriting of user-defined values by defaults and memory leaks through double allocation of pointers. Currently the caller has to ensure that the flag is `false`, or the back-end will *skip* all initialization. A considerable extension of the approach to allocate pointers to the structures at run-time is to employ the *pimpl idiom* discussed in section 3.7

2: Initialize parameters

The caller can optionally initialize the parameter structure, or leave it to the back-end to initialize it to default values. The usual approach is to use the XML module (cf. section 2.5) to read a complete or partial parameter file to set all parameters to specified values; parameters not present in the file are set to their hard-coded defaults.

When WORHP is integrated into another piece of software, this behavior may be undesirable, since the program may need to provide a different (sub)set of default parameters, while still allowing the parameter file to override them. Section 2.5.2 describes how WORHP has decoupled initialization and parameter reading to accommodate these cases.

3: Specify problem dimensions

This step informs WORHP of the essential problem dimensions, which are required to provide storage for initial guesses, bounds and matrix structures. The caller will at least specify n and m , and the number of nonzeros of the derivatives, or the choice to treat them as dense matrices. The example below chooses dense DF with structure initialization by WORHP, dense DG *without* structure initialization by WORHP, and a Hessian HM whose nonzeros admit a diagonal structure with 10 sub-diagonal entries (since the diagonal is always dense, as laid out in section 2.2).

```
opt.n = 11;
opt.m = 42;
wsp.DF.nnz = WorhpMatrix_Init_Dense;
wsp.DG.nnz = opt.n * opt.m;
wsp.HM.nnz = opt.n;
```

The `WorhpMatrix_Init_Dense` flag is actually equivalent with setting `nnz` to its maximum possible value and therefore tautological; WORHP now handle both cases identically.

4: Call the initialization routine

With given problem dimensions, calling

```
CALL WorhpInit(opt, wsp, par, cnt)
IF (cnt%status /= FirstCall) THEN
  PRINT *, "example: Initialisation failed."
  STOP
END IF
```

will provide initialized data structures with allocated storage for initial values, bounds and matrix structures (if not already specified for flagged dense matrices). In addition, all internal

5: Specify initial values and structures

In this step, the caller provides initial values to primal and dual variables, and defines the bounds and (sparse) matrix structures, unless the latter are defined by WORHP in the previous step. The caller can actually “poll” a matrix, whether its structure needs to be defined through the `NeedStructure` flag of every matrix. This is useful for problems with dynamic behavior with respect to the matrix dimensions, and especially for the Hessian matrix HM, which can by the solver parameters be replaced by a blocked BFGS whose structure is determined and defined by WORHP; the matrix arrays are then left unallocated, in which case defining its structure without qualifying the respective code leads to memory access violations.

3. Technical Implementation

```
for(i = 0; i < opt.n; ++i) {
    opt.X[i]      = 1.0;
    opt.Lambda[i] = 0.0;
    opt.XL[i]     = -5.0;
    opt.XU[i]     = 5.0;
}
for(i = 0; i < opt.m; ++i) {
    opt.Mu[i] = 0.0;
    opt.GL[i] = -par.Infty;
    opt.GU[i] = 1.0;
}
if (wsp.HM.NeedStructure) {
    for(i = 0; i < wsp.HM.nnz; ++i) {
        /* Fortran indexing! */
        wsp.HM.row[i] = wsp.HM.col[i] = i + 1;
    }
}
```

Note that this code snippet—if continuing the above snippets—fails to define the sparse structure of DG, even though its dimension clearly indicates that DG must be dense.

6: Start the solver

This is a simple call of the form

```
IF (GetUserAction(cnt, callWorhp)) THEN
    CALL Worhp(opt, wsp, par, cnt)
    ! Do not reset callWorhp
END IF
```

The call needs to be qualified by the `GetUserAction`, since the Reverse Communication (RC) finite difference module “switches off” the NLP solver during its own operation.

7 & 8: Internal operations

These are operations internal to the solver, which involve the workspace handling routines described in section 3.4.

9: Terminate the optimization run

As laid out in section 2.4, the main iteration loop is built by the caller, and can in principle take any form. In all, but exceptional cases, however, the default form

```
while(cnt.status < TerminateSuccess && cnt.status > TerminateError) {  
    /* function evaluations, iteration output, etc. */  
}
```

will do just fine. WORHP sets `cnt.status` to values above `TerminateSuccess`, if successful, and below `TerminateError` otherwise. The `StatusMsg` routine then serves to translate the value of `status` into a human-readable message.

10: Free allocated storage

With WORHP making increasing use of dynamic memory allocation, this last step is mandatory to release all allocated memory, to either post-process the optimal solution, or to begin another solver run. The clean-up routine is very simply called as

```
CALL WorhpFree(opt, wsp, par, cnt)
```

Besides releasing all allocated memory, `WorhpFree` also zeros pointer members and resets workspace slice indices to ensure that no access to stale data is possible. The data structures can now be discarded or reused for another optimization run, starting at the parameter initialization.

3.2. Automatic code generation

Now that section 3.1 has established the technical means to create one-to-one mappings between a C `struct` and a Fortran `TYPE`, the next hurdle is to *maintain* the mapping as the solver evolves: As of mid-2012, the `trunk` development line of WORHP has about 400 data structure components (not counting nested ones) shared between four structures. Any modification to these structures, such as adding a new component, has to be performed identically on both the C and Fortran side, or otherwise “misunderstandings” will take place between both.

As a minimalistic example, consider

```
TYPE, BIND(C) :: ExampleType
  INTEGER(C_INT) :: I
  INTEGER(C_INT) :: J
END TYPE ExampleType
```

```
struct {
  int j;
  int i;
} ExampleStruct
```

which are interoperable in the technical sense, but the mapping between `I` and `J` on the Fortran side and `i` and `j` on the C side is different from what we would expect.

While the above mix-up is simple to spot and repair, keeping hundreds of structure components in sync is a drab, onerous and error-prone task for humans – given that computers were designed to relieve humans from work of this kind, automatic generation of the data structure definition code is the obvious solution to this problem.

The software community offers a rather large number of non-proprietary code generation tools⁶. The following constraints on the code generation tool of choice help in drastically narrowing down the options (in decreasing order of relevance):

License: The tool should be available under liberal license conditions such as Apache, BSD, MIT or (L)GPL and open-source. This ensures simple availability for all developers and long-term availability, even if the tool is discontinued.

Platform: With Linux being the primary development platform, the code generation tool should at least run natively on different Linux platforms. Availability for all target platforms is *not* necessary, since the generated source-code is committed to the source-code repository and can therefore be compiled everywhere; only *changes* to the data structures require running the code generator, but can also be emulated manually, if the change-set is simple.

Target languages: The tool should not be tailored to any specific language – many are designed to produce back-end code for web-applications, such as XML, HTML, ASP.NET, Java, SQL and others. No tool that targets Fortran could be identified, so it should be language-agnostic, or at least simple to use for generating a language that it was not designed for.

⁶http://en.wikipedia.org/wiki/Comparison_of_code_generation_tools lists more than 30 tools, about half of which are openly licensed (retrieved 2013-03-08)

Data representation: The code generator should use a single data file using a simple syntax for describing the data structure components and templates for generating the individual C and Fortran source files.

Extensibility: The generator tool should be easily extensible in addition to its own functionality, which involves providing code snippets in the language the tool is written in, if it is based on an interpreted or JIT language like Python or C#.

Tool language: The tool itself should be written in a common language that is readily available for Linux as primary development platform, such as C, C++, Perl, Python, ..., or with some restrictions Java, C#, Lisp/Scheme, since these depend on additional runtime environments.

The code generation tools *GNU AutoGen*[47] and *cog*[3] both satisfy the above requirements.

AutoGen is written in C and uses *guile*[5] as Scheme interpreter, which allows seamless extension of the inbuilt functionality through user-defined Scheme functions. It reads a single definition file containing arbitrary key-value pairs, similar to C `struct` definitions, and template files with AutoGen-specific syntax (which actually consists mostly of Scheme functions embedded in AutoGen tags) to generate output files. AutoGen is able to generate any textual output format, depending only on the template file, since it is completely language-agnostic; it does offer some auxiliary functions that facilitate generating C files, though.

cog is based on Python and its design is more general than AutoGen: *cog* searches for special tags in a file (which can have any textual format, provided it has block or line comments) and executes the enclosed code with the Python interpreter. This makes it possible to use *cog* for small, inline code generation task, but also for mimicking the behavior of AutoGen by including a central definition file in a Python-readable format.

3.2.1. Data structure definition

The current code generation for WORHP is based on AutoGen, since the concept and syntax of the definition file is closely related to the (initial) task of the code generation; in fact, the initial definition file was semi-automatically generated from the existing data structure code using *sed* and *Perl*, a task that was simplified by the close correlation between the AutoGen definition file syntax and the data structure C headers.

The most important distinction made by the code generation is between *basic* and *struct* i.e. derived types, and between *scalars* and *arrays*, since the initialization mechanisms differ greatly. Basic types are `double`, `float`, `int`, `size_t`, `counter`, `rwmt_index`, `iwmt_index` and `bool`, which are translated to the corresponding types in C and Fortran through simple translation tables. If a component fits none of the types above, it is assumed to be a struct type; struct types are derived types, such as the `WorhpMatrix` type, which have dedicated initialization and clean-up routines. Arrays may be static or

3. Technical Implementation

dynamic; both have sizes, and may have initial values. To accommodate dynamic arrays, the code for allocation and deallocation is automatically generated.

Since the solver parameters in **Params** (usually) have tightly prescribed valid values, it makes perfect sense to add them to their definition and generate code that performs value/range checking and, if necessary, issues a meaningful warning and sets them to sensible defaults.

For all types without given initial values, code is generated that sets them to defined values (usually 0, NULL or equivalent, -1 for workspace slice pointers—see section 3.4) to ensure deterministic behavior. The only drawback to this approach is the fact that it precludes valgrind from tracking accidental use of uninitialized values. On the other hand, since all components *are* initialized, no such use actually exists.

3.2.2. Definition file

The definition file is one of two main components of the automatic code generation framework. Listing 3.4 shows a snippet of WORHP’s data structure definition file for (part of) the **OptVar** structure. The whole file is about 3000 lines and 70 KB in size.

```
WorhpData = {
    struct = OptVar;
    inst   = opt;

    member = {
        name = n;
        type  = int;
    };
    member = {
        name = F;
        type  = double;
        value = "0.0";
    };
    member = {
        name = X;
        type  = double;
        size  = n;
    };
};
```

Listing 3.4: Snippet of the data structure definition file

As mentioned above, AutoGen allows free choice of the keys used. Here, each instance of **WorhpData** starts one of the four data structures. **struct** is the name of the structure to be defined, and **inst** is used for instances of this structure, most importantly in the generated function interfaces. The structure components are enumerated as **member** (since this is shorter), which have (quasi-)mandatory fields **name** and **type**. Table 3.3 lists additional field names recognized by the code generation.

Adding a new data structure component is achieved by adding it to the central definition file and running the appropriate command to auto-generate the source code. This is automatically performed by `make`, which monitors the time stamps of definition file and the generated files and is thus able to trigger the code generation if the definitions were changed – see section 4.1.5. No check or intervention is needed to ensure consistency between Fortran and C, since AutoGen iterates through all **members** deterministically and emits them in the order of the definition file.

Field	Meaning
name	Component name
type	Component data type
default	Default initial value (only valid for basic types)
value	Initial value (only valid for basic types). In <code>value[i]</code> form, overrides default for i -th array element (C indexing).
pointer	Pointer flag; no allocation or deallocation code is generated.
init	Initialization function (used for struct types)
free	Cleanup function (used for struct types)
size	Dynamic array size; must be an integer component of this structure.
initif	Condition for performing the automatic allocation, if size is present.
dim	Static array size; <code>dim[1] ... dim[k]</code> for k -dimensional array
bound	Bounds on scalar values; used to generate range checking code.
valid	Valid value; usually issued multiple times to enumerate valid values.
cmnt	Add comment; emit stand-alone block comment, if name is absent.
cmntC	Add a comment that is only emitted in the C files.

Table 3.3.: Recognized fields of the **member** key in the the AutoGen definition file, and their function

To define named constants or fixed array dimensions, a second set of top-level keys exists, named **Constant**, with fields **name**, **value**, and **cmnt**; the flag **dimension** is attached to constants that are used as array dimensions, although this (currently) does not require any special handling, hence it is ignored by the templates.

With over 400 components contained in the definition file, it is sensible to adhere to some kind of ordering, which is also reflected in the generated code. The current convention is to (roughly) sort components by size and type, and inside each type alphabetically by name. Exceptions exist, some for “historic” reasons and others for increasing locality, i.e. keeping dynamic array sizes like `dim_SBFGS_blockval` close to the respective array. Since re-sorting breaks ABI compatibility, it should be performed sparingly.

3.2.3. Template files

The AutoGen template files use a rather peculiar syntax, embedding the AutoGen expressions in user-selectable (within reason) two-character opening and closing tags, such as [= and =] or <\$ and \$>. The AutoGen expressions consist of macros defined by AutoGen, such as FOR .. ENDFOR, IF .. ELIF .. ELSE .. ENDIF etc., or evaluations of Scheme functions, triggered by the (. (<function> <args>)) syntax⁷. The basic template stub

```
[= FOR WorhpData =] [=
    FOR member =] [= (get "type") =] [= (get "name") =];
[= ENDFOR =] [=
    ENDFOR =]
```

emits semicolon-separated lines of **type-name** pairs, as needed for generating the C header file; for each instance of **WorhpData** in the definition file, it iterates through all **member** instances and emits the values of their **type** and **name** keys (without checking for presence or empty values etc.). The output generated from listing 3.4 by this template would be

```
int n;
double F;
double X;
```

We observe that whitespace *inside* AutoGen tags is insignificant, while everything *outside* AutoGen tags is emitted in the output file. This, together with the two-character start/end tags and the copious (over)use of parentheses in Scheme, leaves AutoGen template files looking rather cluttered. Proper indentation of nested macros together with generous use of comments (started by the # sign) is used in WORHP's template files in an attempt to expose the underlying structure and logic. It is probably safe to assume that a *cog*-based solution in Python would look cleaner.

The simple template above is clearly incomplete, since **X** is supposed to be a pointer (for dynamic allocation) instead of a scalar; furthermore, we also need to accommodate one- or multi-dimensional arrays of constant sizes or nested **structs** as data structure components; for the former, we have to take into account the reverse ordering of dimensions of C in comparison to Fortran; the definition file defines static dimensions in *Fortran* order, hence we have to consume them in reverse order. The complete template algorithm for generating the C header definition is described by algorithm 3.

The algorithm for generating the corresponding Fortran definitions is essentially equivalent to algorithm 3, but differs in its details: static array dimensions are given in natural order, only string members have to be handled differently, since the static array **char string[N]** corresponds to **CHARACTER(LEN=N) :: string** in Fortran.

⁷notice the copious use of parentheses for which Scheme/Lisp is (in)famous.

Algorithm 3 C struct definition generator

```

for <member> in <WorhpData> do
  if <name> is not empty then
    if <cmnt> is not empty then
      emit /* <cmnt> */                                ▷ used to comment single components
      start new line
      emit <type>                                         ▷ translation to C type by table look-up
      if <size> or <pointer> is present then
        append "*"
      emit <name>
      for  $k \leftarrow \langle \text{dim}[i] \rangle$  in reverse order do      ▷ empty loop if dim is not present
        append [ $k$ ]
      terminate line with ";"
    else
      if <cmnt> is not empty then
        emit <cmnt> as block comment                    ▷ used to structure the generated code

```

3.2.4. Applications of automatic code generation

The initial motivation for generating code was to keep the data structure definitions between C and Fortran in sync; this task is clearly accomplished. Since the inception and implementation, however, additional applications have become apparent, turning the code generation framework into a formidable workhorse. Its present incarnation, in addition to its initial task, generates code that

- performs a sanity check on the solver parameters and, where necessary, resets them to sensible defaults;
- defines serialization to and deserialization from XML files (see section 2.5.1) for each and every data structure component;
- in interactive mode (see section 3.3), displays lists of all accessible components as part of the `help` command, and allows the user to select most of the components for inspection or modification.

Code generation also simplifies the introduction of dedicated dynamic arrays to succeed the legacy notion of a shared, large workspaces; even though WORHP implements an automated form of workspace management (see section 3.4), this should be considered as a workaround for an obsolete software design pattern rather than a contemporary, well-designed solution: The conciseness and maintainability of (de)serialization code for only two workspaces provided the single most important argument for holding on to the workspace design pattern. However, with the advent of effortless automatic generation of this code for an arbitrary number of data structure components, the workspace design pattern has lost its justification with respect to (de)serialization, and will probably be phased out in the near future.

Initialization and clean-up code

The first obvious extension to generating the data structure definitions is the generation of code to initialize and free them. For initialization, WORHP receives the essential problem dimensions from the caller and allocates arrays and matrices according to these, and sets all scalar values to default values (0 for counters, -1 for workspace slice indices, NULL for pointers, and individual values for the parameters). The clean-up is responsible for freeing all allocated memory and for putting all data structures into a defined, deterministic state again; this is important for extensive programs that perform memory-intensive post-processing tasks after solving an optimization problem, and for running multiple optimization problems sequentially by re-using the same data structures.

WORHP provides routines `WorhpInit` and `WorhpFree` for the caller to perform these tasks. Besides some auxiliary tasks, their most important purpose is to call the solver-internal routines `Internal_Init<struct>` and `Internal_Free<struct>` for `<struct> ∈ {OptVar, Workspace, Params, Control}`, all eight of which are completely auto-generated for all four data structures.

Parameter sanity check

One of WORHP's features is the sanity check on the parameters available to the user. Although the feature is almost trivial to implement, an estimated 95% of it consisting of simple range or valid value checks, it is shunned by some established pieces of numerical software, possibly *because* of the repetitive and mechanic nature of the code that performs the sanity check; WORHP has about 150 runtime parameters—a few more, if program-dependent parameters, such as `MatrixCC`, are included in the count. Again, automatic code generation is perfectly suited to generate code for performing mechanic numeric range checks, which are formulated in the definition file through the `valid` field to enumerate valid values, and the `bound` field to define upper or lower (numeric) bounds. The current definition file contains about 130 instances of these fields. The remaining parameters cannot be mechanically checked for correctness, requiring manually created specialized code.

Generation of (de)serialization code

The most extensive application of AutoGen to generate highly repetitive code in WORHP targets the serialization and deserialization code that translates between the data structures and XML files in a platform-independent way. Figure 3.2 shows the parts of the serialization module; although only a single part of it is auto-generated, it accounts for more than 80% of the code in all mentioned files.

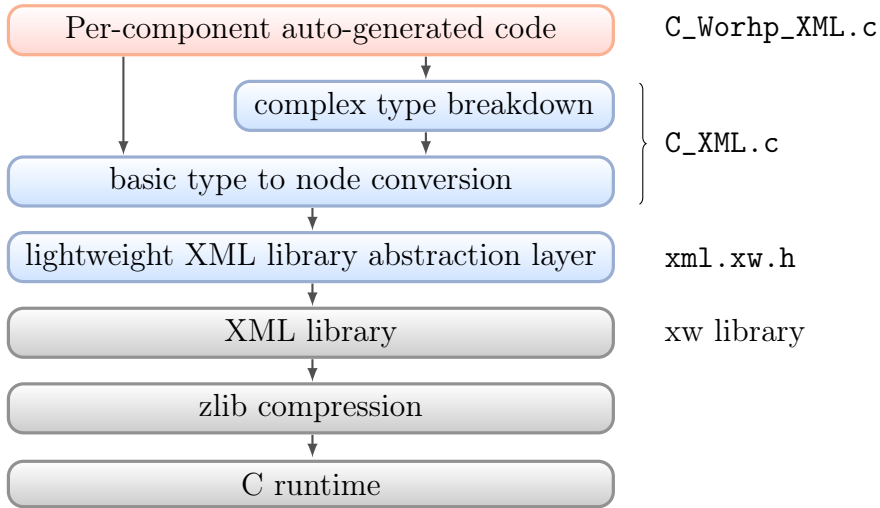


Figure 3.2.: Hierarchy of the serialization module, with the most specific parts at the top and the most generic ones at the bottom. The per-component part is the most substantial part, accounting for more than 80% of the code.

Figures and conclusion

All in all, AutoGen is used to generate 11 files with a total of almost 600 KB and over 16,500 lines, although 29% of both figures are accounted for by manually created include files, which contain code that is too specialized to be auto-generated. All templates and the definition file together make up only 100 KB and contain less than 4500 lines (3000 of which account for the definition file).

Given that the initial setup needed considerable effort, while the continued maintenance and extension is simple, and assuming that the distribution of effort is inverted in the manual approach—simple to set up, but increasingly harder to maintain with growing size—, automatic code generation is a maintenance boon for WORHP. Even if, for the sake of the argument, we assume that code which generates code is twice as hard to write and maintain as manually generated code (together with keeping all files in sync), the figures above suggest that automatic code generation still has a considerable advantage over writing all files by hand.

3.3. Interactive mode

The interactive mode of WORHP is motivated by the desire to be able to influence (long-running) optimization runs in real time. Examples for possible runtime interventions include increasing the amount of output to inspect conspicuous iteration progress, fine-tuning parameters such as `MaxIter` (to prevent unsuccessful termination), activating a feasibility mode, or saving the current iterate to a logfile. With traditional NLP solvers, this is either impossible to achieve, or at least requires a modification of the calling program and a restart of the optimization run.

While Reverse Communication does in principle offer very fine-grained control over the optimization process, its use is limited to the calling program. Equipping it with the necessary degree of intelligence and autonomy to react to any conceivable condition that would cause a human user to intervene is an unrealistic goal — a much more feasible approach is to actually enable runtime interventions: The interactive mode listens to keyboard commands `Ctrl+L` and `Ctrl+I`.

	ITER	OBJ	CON	sKKT	FLAGS	TIME
	[0 18]	-1.32300000E+01	9.87E-01	2.24E-02	Uin	4.00E-02
<code>Ctrl+L</code>	Worhp : Writing logfile * Logfile written					
	[1 119]	-2.24623190E+02	1.78E+00	4.14E+02	Uin	3.60E-01
	[2 75]	-2.23659601E+02	1.60E+00	3.22E+02	Uin	5.40E-01
<code>Ctrl+I</code>	(Worhp) norm opt.x 9.5461941162E+03					
	(Worhp) c					
	[3 36]	-2.23191095E+02	1.58E+00	8.76E+02	Uin	6.60E-01

The `Ctrl+L` keyboard command instructs WORHP to write the current iterate to a logfile (a partial or complete Hotstart file, see section 2.5.1) but otherwise continue the iteration process, while `Ctrl+I` pauses the optimization to present a rudimentary prompt to the user, offering various ways of data inspection and interaction through a number of simple commands. Listing 3.5 shows the help-screen output of the interactive mode, enumerating the commands understood by it, and listing 3.6 shows an exemplary listing of the components of the `OptVar` structure using the 1-argument form of the `help` command that is intended for use as reference.

Implementation

The interactive mode is implemented through a Fortran-based parser: Each user input is assumed to consist of at most 3 tokens, separated by whitespace. The first (and possibly only) token is always a command, possibly followed by an argument (two arguments in the `set` case). If the second argument is given, it is assumed to be the name of a data structure component. The code auto-generation is used to create and maintain a list


```

(Worhp) help
Usage: CMD [ARGS]

Control and help commands
  continue | c   Continue the optimisation.
  help      | h   Show this help.
  help <STR>      List members of data structure <STR>.
  status                Show status.
  step              | s   Step over next call.
  terminate | t   Gracefully terminate the solver.

Commands for inspecting WORHP data (CMD ITEM)
Specify ITEM in C syntax, e.g. "opt.N" or "wsp.nrws"
  absmax          Print max absolute value.
  absmin          Print min absolute value.
  max             Print max value.
  min            Print min value.
  norm | norm2    Print 2-norm.
  norm1          Print 1-norm.
  normmax        Print max-norm.
  print | p      Print a data item.

Commands for changing WORHP data (CMD ITEM VALUE)
  set             Set scalar or vector ITEM to VALUE.

```

Listing 3.5: Help output of interactive mode (highlighting added subsequently, not [yet] implemented)

of selectable components, to display them in a help screen, as shown in listing 3.6, and organized in a long `SELECT CASE` statement of the form

```

CASE("opt.n")
  P%iPtr => opt%n
CASE("opt.m")
  P%iPtr => opt%m
CASE("opt.f")
  P%dPtr => opt%F
CASE("opt.x")
  CALL C_F_POINTER(opt%X, X, [opt%n])
  P%Vec => X

```

Upon selection of a component, an appropriate Fortran pointer is pointed to the selected component. The pointer structure `P` is then passed to the handling routines, which perform the action selected by the first argument. This simple structure is only made possible by the introduction of automatic code generation (see section 3.2), since the code for the selection process is made up of roughly 800 exceedingly repetitive lines of Fortran code, which is unsuitable for manual maintenance.

3. Technical Implementation

```
(Worhp) help opt
Selectable members of "OptVar":
  int          opt.n
  int          opt.m
  int          opt.nGPart
  int          opt.iGPart
  double       opt.F
  double[]     opt.X
  double[]     opt.XL
  double[]     opt.XU
  double[]     opt.Lambda
  double[]     opt.G
  double[]     opt.GL
  double[]     opt.GU
  double[]     opt.Mu
  int[]        opt.GPart
```

Listing 3.6: Detail help output of interactive mode (highlighting added subsequently, not [yet] implemented)

Limitations

The current implementation state of the interactive mode is best described as proof-of-concept: Due to the technical nature of the keyboard-event-handler, it is only available on Linux platforms (and there limited to the AMPL and the library interface, since MATLAB heavily interferes with I/O). Its coverage is limited to primitive components, excluding the various **structs** used (such as the derivative matrices), element-wise inspection and manipulation of vectors is not possible and no failsafes are implemented to prevent “destructive” manipulations that will crash the solver.

3.4. Workspace Management

The use of two large workspaces (integer and real/double) to provide temporary workspace or to hold persistent vector-valued quantities is the only (mis)feature of legacy Fortran software that is still present in WORHP. Its introduction dates back to the inception of WORHP, where it enabled serialization and interoperability with C. However, the introduction of automatic code generation and the sparse matrix storage redesign raise the question, whether its use is still appropriate.

3.4.1. Automatic workspace management

Old FORTRAN codes require workspace, since the language did not allow dynamic memory allocation—at least not in a standard-conforming way that was compiler-independent. Therefore, the calling program had to provide large enough workspace by allocating sufficiently large, static arrays in the main program. This approach was initially chosen by the FORTRAN fathers to allocate all memory on the stack, which has a performance advantage over heap storage. Nowadays however, modern compilers may allocate even static arrays of a certain size on the heap, and dynamic memory allocation is ubiquitous, leaving the legacy approach obsolete.

WORHP uses a hybrid approach to workspaces: First, it estimates the required workspace size during initialization, using known dimensions, estimates and some heuristics. The integer and real workspaces are then allocated using the `malloc` family of functions from the C standard library. This workspace is then partitioned into so-called *slices* (derived from the term *array slice* for reasons that will become obvious) by the workspace management routines. The automatic management of the workspace is performed through a table of allocated slices, as shown in table 3.4.

1-indexing		0-indexing		size	ID	name
Start	End	Start	End			
0	9	1	10	10	1	vec1
10	14	11	15	5	2	vec2
15	34	16	35	20	0	
35	44	36	45	10	4	vec4

Table 3.4.: Example workspace management table. Note the third entry with ID 0, meaning that it has been freed and can be reused for the next allocation.

The Workspace Management Table (WMT) contains start and end indices for each workspace slice, as well as its size, a numeric ID and an optional name (for monitoring and debugging purposes). Each workspace has its dedicated WMT, thus referred to as IWMT (for integers) and RWMT (for reals). The “pointer” to an allocated slice is either its row index in the WMT, the ID, or the name. IDs are identical to the row index, but

3. Technical Implementation

were introduced to enable defragmenting the workspace, should the need arise, in which case the row indices would change, while the IDs remained constant. Workspace slices are supposed to be accessed through Fortran’s *array slice* syntax `v(n1:n2)`, although sub-slice or element-wise access is possible. Accessing a workspace slice requires to first set up a Fortran pointer to the respective workspace, since these are C pointers, or in Fortran syntax `TYPE(C_PTR)F03`, which cannot be accessed directly, but only through a Fortran pointer type initialized by the intrinsic `C_F_POINTERF03` subroutine:

```
DOUBLE, POINTER :: rws(:)
CALL C_F_POINTER(wsp%rws, rws, [wsp%nrws])
```

Given the (Fortran) workspace pointer, the slice with index `k` can then be accessed through

```
rws(wsp%RWMT(k,3):wsp%RWMT(k,4))
```

Since this is rather unwieldy, slice access is provided through preprocessor macros

`X_S(wsp,idx,iws)` for a whole slice,

`X_N(wsp,idx)` to return the size of the slice,

`X_E1(wsp,idx,iws,i)` to access a single element with 1-indexing (i.e. Fortran-style indexing), and

`X_R1(wsp,idx,iws,i,j)` to access a sub-slice (“range”) with 1-indexing,

where $X \in \{I, R\}$ for real and integer versions. Especially the range macro explains, why the 1- and 0-indexing start and end indices in table 3.4 seem to be swapped: To access, for instance, the first 2 elements of `vec2` in 0-indexing, the range access macro must return `rws(11+0:11+1)`, whereas with 1-indexing, it must instead return `rws(10+1:10+2)`. The choice to save indices for both indexing ways was made to save integer additions on access, trading (a small amount of) memory for fewer arithmetic operations, although it remains unclear, whether the alternative would be measurably slower (or faster).

Advantages

The main advantage of automatic workspace management is that it reduces sources of (human) error, when partitioning the workspace manually; the predecessor of WORHP suffered severely from this during some stages of its development. Slices are allocated by the `InitRWSlice` and `InitIWSlice` subroutines, which search for free space in the workspace and allocate it, or provide meaningful error messages upon failure to allocate; likewise, dedicated routines to free allocated slices exist. This enables the use of the workspace management for both providing temporary storage in solver subroutines and persistent storage for vectors.

The `Workspace` data structure holds various integers that serve as slice indices; unfortunately, no obvious way exists that enables the compiler to prevent confounding integer and real workspace slice indices. An initial version of WORHP also used the workspaces

to hold the sparse matrices; the sparse matrix data structure was essentially a wrapper around workspace indices and associated dimensions.

Besides eliminating sources of error, the workspace approach enabled the serialization of the data structures (before automatic code generation), since only two huge workspaces and a number of slice indices had to be (de)serialized, which can be managed through manually written code.

Disadvantages

Since the matrices were held there, both workspaces needed to be of significant size, which is especially critical given WORHP's design goal to solve large-scale problems. This is further aggravated by two factors: On 32-bit systems, the workspace sizes could overflow the integer range, and thereby the maximum size allocatable by the OS, even if all problem dimensions were well below critical dimensions. Furthermore, especially on systems already under load by a complex model, allocation of single, huge chunks of memory may easily fail, even if the sum of available memory is greater, due to memory fragmentation. The SBFGS module, with its very dynamic memory requirements put a further strain on the workspace management; in particular, it requires temporary storage, but cannot provide (reasonable) ex-ante estimates for the required sizes.

The redesign of the `WorhpMatrix` data structure had each matrix manage its own storage requirements through the use of C pointers, permitting dynamic allocation and resizing, and uses the Fortran2003 standard methods to access them from Fortran. This remedied all of the abovementioned shortcomings, since the workspace sizes shrunk, did not depend on the matrix dimensions, and the overall memory demand was distributed over various smaller chunks.

Older versions of WORHP did not estimate the required workspace, but left the choice to the user—although it was already allocated dynamically; the user's (forced) choice was limited to the sizes. This would lead to unsuccessful terminations of the solver due to lack of workspace. In particularly devious cases, the termination would happen shortly before reaching the optimum, when extra workspace is allocated upon finding (and storing) the first acceptable solution. This shortcoming was fixed through the initial workspace estimate used by the current version of WORHP, depending on known dimensions, estimates and heuristics.

Finally, the workspaces are a debugging burden, since the interpretation of their content depends on the workspace management tables, which greatly complicates the use of debuggers to inspect the solver data.

3.4.2. Dedicated dynamic memory

The rationale for the workspace approach was two-fold:

- Storage of the sparse matrices: Now obsolete through the `WorhpMatrix` type with separate memory management.

3. Technical Implementation

- Keeping the serialization code sufficiently concise for efficient manual maintenance: Left obsolete by automatic generation of the (de)serialization code.

WORHP has therefore been transitioning to dedicated dynamic memory in the form of C pointers to preserve interoperability with C. This mostly applies to persistent storage for vectors in the `Workspace` structure, while routine-local temporary storage is provided by `ALLOCATEF90`, taking care to prevent allocations inside loops or heavily used code sections to strictly limit the overhead from dynamic memory management. As soon as existing workspace slices are transitioned to dedicated storage, the workspaces can continue to be shrunk, until they can finally be completely removed.

The first application to use dedicated dynamic was the SBFGS module, which requires significant amounts of storage, but cannot provide reasonable ex-ante estimates. This is solved through a dedicated C pointer, and combined with reallocation accessible from Fortran by defining a Fortran-C-binding interface to the C standard library routines. The memory requirement of the working SBGFS method is monitored, and if additional memory is required, the amount of allocated storage is doubled. This approach is conservative enough to limit unused (= wasted) memory, and the number of required reallocation steps grows logarithmically; this is a standard approach to handle dynamically growing memory requirements, and is also used, for instance, by implementations of the STL `vector` class in C++.

Dedicated dynamic memory requires a C pointer and its dimension. WORHP uses conventions of the form

```
size_t dim_SBFGS_blocksize;  
int *SBFGS_blocksize;
```

to associate vectors with dimensions. Dimensions always precede the vectors, to ensure that the dimension is known before reading the vector from a Hotstart file.

Performance considerations

Dynamic memory management causes overhead, which can be drastic if allocate or free operations are performed inside a heavily rolling loop. If care is taken to prevent this scenario, no measurable adverse affects are to be expected.

To make a more reliable statement, Fortran's `ALLOCATE` mechanism is compared with WORHP's memory management through the two code snippets

```
DO j=1,N  
  DO i = 1, SIZE(idx,1)  
    ALLOCATE(idx(i)%v(i**2))  
  END DO  
  DO i = 1, SIZE(idx,1)  
    DEALLOCATE(idx(i)%v)  
  END DO  
END DO
```

for `ALLOCATE`, using the Fortran “trick” of encapsulating the allocatable component `v` inside a user-defined type to enable arrays of it, and

```
DO j=1,N
  DO i = 1, SIZE(idx,1)
    CALL InitIWSlice(status, wsp, idx(i), i**2, 'test')
  END DO
  DO i = 1, SIZE(idx,1)
    CALL FreeIWSlice(status, wsp, idx(i))
  END DO
END DO
```

External timing with the intrinsic `time` command of `bash` to measure the wall time for $N \in \{10^3, 10^4, 10^5, 10^6\}$ (`idx` has size 50) is used to determine a simple linear regression with `R`, which gives results

	Estimate	Std. Error	t value	Pr(> t)
(Intercept)	1.348e+00	2.474e-02	54.47	0.000337 ***
N	6.849e-06	4.924e-08	139.10	5.17e-05 ***

for `ALLOCATE` and

	Estimate	Std. Error	t value	Pr(> t)
(Intercept)	1.366e+00	2.086e-02	65.46	0.000233 ***
N	4.594e-05	4.152e-08	1106.49	8.17e-07 ***

for `WORHP` workspace. Both results show an extremely good fit of the linear models, which are

$$t_{\text{ALLOCATE}}(N) \approx 1.35 + 6.8 \cdot 10^{-6} \cdot N \quad \text{and} \\ t_{\text{IWSlice}}(N) \approx 1.37 + 4.6 \cdot 10^{-5} \cdot N.$$

Both methods for dynamic memory management have almost identical offsets (overhead of the surrounding program, which is equal for both), but Fortran `ALLOCATE` is about $\frac{4.6 \cdot 10^{-5}}{6.8 \cdot 10^{-6}} \approx 6.7$ times as fast as `WORHP`’s workspace management. Putting it in other terms, a combined call of `ALLOCATE`/`DEALLOCATE` requires roughly 270 ns⁸, whereas the `InitIWSlice`/`FreeIWSlice` pair adds up to about 1800 ns. Regarding the complexity of today’s super-scalar CPUs with various cache levels, these *absolute* measurements can at best provide rough estimates for actual numeric software under load, but the timing *ratios* provide more reliable guidance, since the test conditions are equal for both.

Considering that Fortran pointers are rather heavy-weight compared to the rudimentary C pointers, carrying stride and bounds information, and that Fortran `ALLOCATE` internally uses `malloc`, it is clear that the C standard library functions have even lower overheads than their Fortran siblings. In Fortran, an additional call to `C_F_POINTERF03` is required

⁸Core i7-2600, Ubuntu 12.04 x84_64, gfortran 4.6 with `-g -O3` flags

3. *Technical Implementation*

to access both the WORHP workspace and the dedicated dynamic memory, which causes some additional overhead. Measurements using the `callgrind` tool of `valgrind` indicate, however, that `C_F_POINTER` calls are rather light-weight, requiring 63 instructions and 8.7 ns per call to dynamic `libgfortran`, and 55 instruction and 6.8 ns per call to the static version.

Conclusion

The use of workspaces, along with the access macros to keep the code concise, and automatic workspace management through tables to eliminate sources of (human) error was an appropriate solution to enable serialization and reasonably dynamic memory management. The workspace approach carries with it various disadvantages, some of which affect WORHP's performance as solver for large-scale optimization problems. With the redesign to workspace-independent sparse matrix structures and the introduction of automatic code generation for serialization, the workspace approach is left without obvious advantages, and a technically feasible alternative exists in the form of dedicated dynamic memory. Since the latter also has a performance edge over WORHP's memory management through WMTs, the only reasonable conclusion is to phase out the legacy workspace approach, and replace it by dedicated storage. The question whether to perform this as sudden, disruptive replacement, or a slow, incremental one can only be answered in consideration of the currently active developments.

3.5. Internal XML parser

Section 2.5 describes the rationale and applications of the serialization module. While it was backed by the Gnome XML library *libxml2*[73] since its inception, the dependency on an external library hampers portability. In particular, *libxml2* (together with *its* dependencies) is a maintenance burden for non-unixoid platforms. The various XML standards (Namespaces, XPath, Relax NG, XInclude, XSLT and others) it implements are irrelevant for our purposes and needlessly increase complexity and code size; while the configuration system of *libxml2* allows to disable many of these modules, internal dependencies seem to forbid building a truly minimal XML library tailored to our needs.

The obvious solution is a dedicated, tailored, minimalistic XML parser that focuses on speed, low resource use, and portability. The only required functionality are serialization and deserialization between a DOM tree and an XML file; features like DTD checking or high tolerance against malformed documents are of little relevance, because the parser will encounter two types of files only: Parameter files, which are derived from the valid default parameter files, and Hotstart files, which are generated by WORHP and thus safe to assume as being valid.

Inspired by [77], the parser is based on Ragel[69, 70] to create a plain C finite-state machine (FSM) to parse the XML subset explained above, using the very terse grammar

```

1 attribute = ^(space | [/>=])> >buffer %attributeName space* '=' space*
2   (('\'\' ~\'\'\'* >buffer %attribute \'\'') | ('\"\' ~\"\'\'* >buffer
3   %attribute '\"'));
4 element = '<' space* >buffer ^(space | [/>])> %elementStart
5   (space+ attribute)* :>> (space* ('/' %elementEndSingle)? space* '>'
6   @element);
7 elementBody := space* <: ((~'<'> >buffer %text) <: space*)? element?
8   :>> ('<' space* '/' ~'>'> '>' @elementEnd);
9 main := space* element space*;

```

which has many similarities to regular expressions, but is interspersed with FSM instructions for extra unreadability. The Ragel code is mixed with the target code—C code in our case. Ragel then reads the file and reproduces the original file, with the Ragel instructions replaced by FSM code. The generated FSM code implements either a table-driven or goto-driven FSM (the latter being faster, but larger in code size—roughly 800 lines of C code in case of WORHP’s XML parser), whose code is completely unmaintainable, but also not usually in need of any maintenance. Modifications are performed in the mixed C-Ragel source file, which is then again transformed by Ragel into a plain-C file. This also implies that Ragel is only required, if changes to the FSM are introduced, while in all other cases the generated C file can be used; this is important to limit the number of dependencies on a build platform.

The generated FSM steps byte-wise through the XML file, matching elements, attributes and such according to its grammar. The *buffer* term is one of several embedded actions that calls an internal snippet of code to store the current FSM pointer for access to found elements, attributes, and content.

3. Technical Implementation

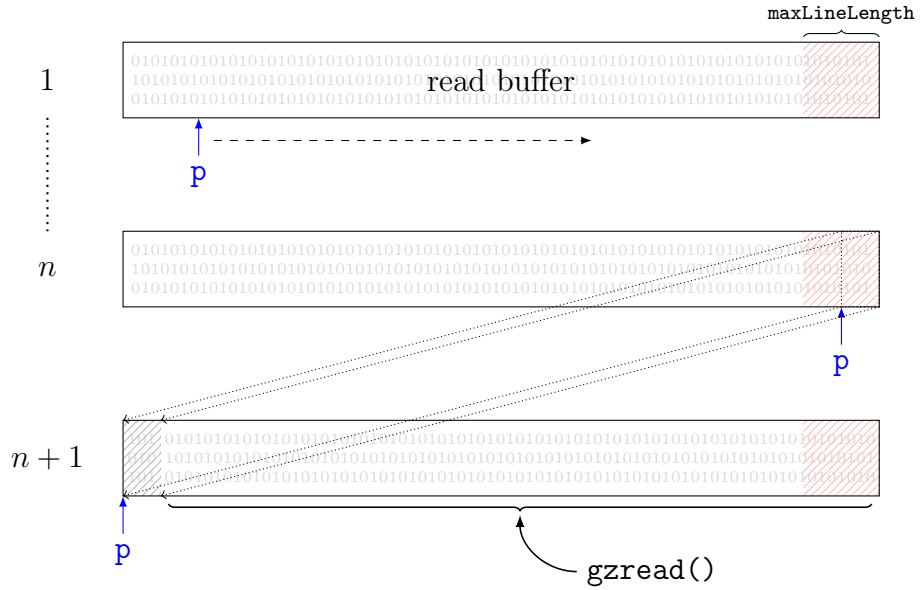


Figure 3.3.: Chunk-wise operation of the XML parser. This method assumes that the largest element found is bounded in size by `maxLineLength`; monitoring the FSM pointer `p` at the start and end of elements is then sufficient to ensure that the read buffer never runs empty, before the end of the XML file is reached.

The parser is combined with `zlib`[24] to directly read compressed files; while compressing files is certainly contra-productive for parameter files, since compressed files are not human-readable, it is useful to reduce the size of Hotstart files—typically by a factor of almost 10. Fortunately, `zlib`’s `gzread` reads uncompressed files as well, without requiring special flags. To reduce the memory footprint in case of large (Hotstart) files, XML files are parsed in chunks, instead of uncompressing the whole file in a single pass; while this increases the algorithmic and runtime complexity over the one-pass approach, it enables parsing greater XML files. The principle is sketched in figure 3.3. Since the parser constructs a DOM, the XML size is still limited by available memory, and requires SAX-like approaches to reduce its memory footprint, i.e. an event-driven XML parser that directly maps encountered elements to the respective data structure components during a single pass through the file. Since this would entail a complete redesign, a SAX parser will only be considered, if the XML file size becomes a limiting factor. Supposedly, the automatic code generation would greatly facilitate the mechanic component-matching between XML and data structures.

3.6. Established shortcomings, bugs and workarounds

Development and testing of WORHP has exposed a number of issues with compilers, runtimes and other development tools, especially in the area of C-Fortran-interoperability. The usual reaction of finding and implementing workarounds often accounts for quirky or outright strange code constructs, whose rationale is worth documenting.

3.6.1. Intel Visual Fortran 10.1

Intrinsic `C_NULL_PTRFortran` constant has non-zero value⁹

The QPSOL component uses C-pointers collected in the `QPWorkspace` struct to accept arguments and return results. Before calling QPSOL, these pointers are pointed to the respective matrices and vectors by a Fortran caller using the `C_LOCFortran` intrinsic. After the call, cleanup is performed by setting all pointers to `C_NULL_PTRFortran`, to prevent the final `free()` from trying to free memory that has been allocated by Fortran, which would result in a runtime error; note that `free(NULL)` is safe to call, since it performs no action[38, §7.20.3.2(2)].

The Intel Visual Fortran compiler has a defect, where the intrinsic `C_NULL_PTRFortran` constant has a non-zero(!) value, when using the shared runtime libraries¹⁰. This unconditionally leads to a runtime error, since WORHP will try to `free()` memory belonging to the Windows kernel (i.e. memory reserved by the runtime-linker for the Fortran runtime DLL).

3.6.2. Windows

Behavior of `atof` depends on locale

The whole family of IO routines of the Windows runtime depends on the currently active locale. While, in all fairness, this is not a bug, it can nevertheless lead to very surprising results when parsing parameter or Hotstart files (described in section 2.5.1) subject to different locale settings. The C locale defines `.` (point) as decimal point according to the ANSI standard, with the result that the textual representation of floating point numbers is of the form `9.87e+012`. If, however, the locale defines `,` (comma) as decimal point, ANSI-conforming representations of floating point numbers are parsed incorrectly. Consider the line

```
<DOUBLE name="eps">2.2204460492503131e-16</DOUBLE>
```

⁹Intel article DPD200049162 on the issue – retrieved 2011-09-27

¹⁰According to Intel developer Steve Lionel, this is due to a missing `dllexport` annotation, which results in the constant to attain its *location in memory* (where the shared runtime library data section resides), instead of the *value at this location* (being `0x0` of appropriate length), i.e. essentially one missing level of pointer dereferencing is causing the defect.

3. Technical Implementation

from WORHP's parameter file. This defines the quantity to be used as machine epsilon, which is mainly needed for comparisons. On a Windows system with active German (or similar) locale, , (comma) is used as decimal point, hence `atof` parses the 2 and discards the rest, in particular the exponent, resulting in `eps = 2` and rather odd behavior of WORHP.

Workarounds exist:

- Retrieve the currently active decimal point by using `localeconf`, and if unequal to . (point), convert all number representations in ANSI form to the current locale by replacing . by the localized decimal point before handing them over to `atof`.
- Change the locale settings of the operating system. This is problematic, since only few users will be able to, be it through lack of administrative rights or lack of knowledge. This can also be quite disruptive to the workflow and is therefore not recommended.
- Use `setlocale(LC_NUMERIC, "")` to reset the locale settings for number representations (back to ANSI). This may have an impact on the calling program, though, depending on the operating system.

3.7. Maintaining compatibility across versions

This section is concerned with future directions of the WORHP implementation. Although this is irrelevant for an academic NLP solver based on an experimental new architecture, it *is* relevant for WORHP, since it is, and will continue to be, marketed as a software product with a longer life cycle than just a few years.

A major issue of the current setup with non-opaque data structures is the fact that any change to the structures that add, remove or modify the type of a member break binary compatibility of the solver with existing code. Binary compatibility, also referred to as ABI compatibility, is a desirable goal for WORHP, as it allows for simple upgrade paths: where WORHP is included as shared library, an upgrade (for improved performance, introduction of a new feature or fixing a bug) consists solely of replacing said shared library by the new version.

ABI compatibility will usually require API compatibility, although from a purely technical point of view, neither implies the other – the ABI is concerned with the assembler level, while the API is a concept on the source code level; changing the semantics of a function while keeping its name and arguments does not change the ABI but *does* break the API, while adding a member to a data structure is not an issue for API compatibility but changes the ABI. In general, both should go hand in hand.

During the development of WORHP, changes (additions, deletions or renamings) to the data structures have by far been the most common source of API and ABI breakages.

A first attempt at alleviating the problem is to have WORHP *detect* that incompatible versions are mixed; this happens, for instance, if the shared library is replaced, while the C headers or Fortran module files (which define the data structures) are not updated, often resulting in subtle “one-off”-errors, i.e. observable offsets between subsequent components. Let us consider the example

```
struct {
    int m;
    int n;
} OldAPIStruct
```

```
struct {
    int k; /* new! */
    int m;
    int n;
} NewAPIStruct
```

and assume that an updated version of the shared library expects **NewAPIStruct**, but the caller passes **OldAPIStruct**; this happens, if only the shared library is replaced but the headers are left unchanged, although the API has changed. The user will notice that something is wrong, since any changes to **OldAPIStruct.m** are ignored (since **k** corresponds to some new functionality that the user is unaware of), and changes of the user to **OldAPIStruct.n** show up as changes to **NewAPIStruct.m** to the library.

In most cases, such offsets cause mayhem to the solver operation, since adjacent values may have very different values/ranges and confounding them will produce nonsensical settings.

3. Technical Implementation

One possible approach is the inclusion of a data structure revision that is incremented at every ABI-breaking change into both the library and the header/module files, such that a simple comparison at startup will immediately spot incompatible versions and issue an error. While this is an improvement over the “undetected mayhem” approach, it does not improve compatibility, but only detects incompatibility.

The *pimpl idiom* (= “pointer-to-implementation”, cf. [60, Ch. 3]), which is also called “Cheshire Cat” or “d-pointer” idiom, is a considerably advanced approach that splits a class¹¹ into a public component that remains stable over a long time horizon and a private component that may change. Since this implies hiding major portions of the data structures, appropriate access functions have to be provided. The implementation of the data hiding is solved in C by defining incomplete types (see listing 3.7), while Fortran seems to allow components with the `PRIVATEF90` attribute in public types.

```
/* pimpl.h */
struct hidden;
struct public {
    /* must not change between versions */
    double *X;
    struct hidden *opaque;
}
struct hidden *new_opaque();
void do_something(struct hidden *);

/* pimpl.c */
struct hidden {
    /* hidden components - may change between versions */
}
struct hidden *new_opaque() { ... }
void do_something(struct hidden *opaque) { ... }
```

Listing 3.7: Data hiding in plain C through incomplete types.

The beauty of this approach for WORHP is that the basic concept is rather simple to implement and sufficient for long-term ABI compatibility, since many central aspects of an NLP solver will not change. It is also considerably simpler than in the C++ case, where additional considerations, such as overloading etc. influence the actual binary compatibility between different versions of a pimpl class.

The downside of applying this approach to WORHP is the initial break of API and ABI compatibility, and the considerable amount of refactoring of the solver code it will require; the latter is somewhat mitigated by the fact that this refactoring will be highly mechanic and thus relatively simple to perform (semi-)automatically.

¹¹the pimpl idiom has been popularized by large C++ projects.

Solver Infrastructure

Wenn der Reiter nichts taugt, ist das Pferd schuld.
A bad workman blames his tools.

(Proverb)

4.1. Configuration and Build system	92
4.1.1. Available build tools	92
4.1.2. Basics of make	94
4.1.3. Directory structure	95
4.1.4. Pitfalls of recursive make	96
4.1.5. Non-recursive make	99
4.2. Version and configuration info	104
4.3. Testing Approach	107
4.4. Testing Infrastructure	110
4.4.1. Parallel testing script	110
4.4.2. Solver output processing	113
4.5. Parameter Tuning	116
4.5.1. Why tune solver parameters?	116
4.5.2. Sweeping the Parameter Space	117
4.5.3. Examples	120
4.5.4. Conclusions	127
4.6. Future directions for testing	128

4.1. Configuration and Build system

The configuration and build system of WORHP is used to build the available interfaces for its supported platforms with the chosen solver configuration. One of the most important tasks of a build system is to track *all* inter-dependencies of source and configuration files, on the one hand to prevent unnecessary recompilation or regeneration of objects, whose dependencies have not changed, and on the other hand, to reliably trigger the necessary recompilation or regeneration of objects whose dependencies *have* changed. Failure to achieve the first task typically leads to time-consuming recompilation cascades, while failure to achieve the latter will result in incomplete or faulty builds.

The solver configuration options to build WORHP include

- the choice of a target platform/compiler pair and system paths,
- debug versus release builds (compiler optimization options),
- enabling additional debug output for some solver components,
- static versus shared libraries,
- linear algebra solver support,
- enabling optional modules (XML, licensing).

As of late 2010, WORHP is supported on these platforms

Linux: *Debian*, *Redhat* and their derivatives,

Unix: *Solaris* and *OpenSolaris*,

Mac: using MacPorts[4],

Windows: *Server 2000*, *XP* and later,

using the GNU, Intel and Microsoft compilers. Combined with the fact that WORHP is being developed and tested on a number of systems with different compiler/bitness constellations, this leads to a large number of different possible configurations to build the solver. Together with the need to direct the automatic code generation and handle the mixed-language approach, creating and maintaining the build system for WORHP is a complex task.

4.1.1. Available build tools

The development of WORHP being Linux-centric, GNU `make`[68] is used as the main build infrastructure, since it is ubiquitous on all unixoid platforms and also available for the other supported platforms.

Possible alternatives to GNU `make` are

Boost.Build: Boost.Build[1] is a Jam-based build system to configure and build C++ libraries or applications for various platforms. It reads concise *Jamfiles* that are written in “Jam”, which has a close resemblance to `make`’s syntax.

Since Boost.Build is C++ centric, building libraries of non-C++ sources is only possible by some degree of hacking, for instance by writing explicit rules for all non-C++ source files; this, however, defeats the purpose of conciseness by automation, and introduces unwanted complexity. Also, in contrast to SCons and CMake, Boost.Build cannot resolve Fortran module dependencies.

The GNU Build System: Also being referred to as “Autotools”, the GNU Build System is a standard infrastructure component used to build most unixoid open-source software. While the complete build system actually consists of a number of interlocking pieces of software, namely GNU Autoconf[18], GNU Automake[2] and optionally GNU Autoheader (part of Autoconf) and GNU libtool[72], Autoconf is sometimes used synonymously for the complete build system.

The GNU Build System constructs a portable `configure` script that in turn instantiates a (mostly recursive) makefile hierarchy to build a software package on very different systems. The capabilities of the `configure` script include cross-compiling (although this is involved), checking compiler capabilities, and detecting and working around quirks, known bugs or missing features. Projects that use the GNU Build System are configured, built, and installed using the three famous commands

```
./configure
make
make install
```

The strength of the generated `configure` script lies in the fact that it performs very fine-grained checks for actual capabilities of the build environment, instead of relying on version look-ups or databases. A capability is tested by spawning a sub-process, which runs a shell-script, compiles a small piece of test code, or runs a system tool to inspect its output. This greatly eases porting an application, once the initial effort of setting of Autoconf has been suffered.

However, Autoconf has a number of serious issues:

- Autoconf is complex and has a very steep learning curve, partly due to the fact that it is built on M4, the GNU macro processor.
- The Autotools are notorious for subtle incompatibilities between their many different versions, which is a burden when manipulating the build system of more than one project, since this may require parallel installations of different Autotool versions.
- Many software packages that depend on Autoconf use recursive invocations of `make`. This is harmful for dependency tracking between different source directories, as described in 4.1.4.
- Autoconf is slow¹. Part of this performance issue is caused by the repeated spawning of sub-processes, which is a cheap operation on unixoid systems

¹See e.g. section 2.7.2 (test with many dependencies) at <http://retropaganda.info/~bohan/work/psycle/branches/bohan/wonderbuild/benchmarks/time.xml#2.7.2> – retrieved 2013-01-30.

4. Solver Infrastructure

(internally using `fork`), but a very costly operation e.g. on Windows platforms. Although the `configure` script does cache some of its results, in practice many of the testing operations are performed repeatedly by sub-`makes`, and not cached for other `configure` scripts by default, unless instructed so by the user.

CMake: CMake[33], the “Cross Platform Make” is a mature and powerful tool to generate *build files* to be used by the respective native build system, i.e. makefiles for unixoid system and Visual Studio project files for Windows. CMake has first-class support for Fortran 90² and claims to have facilities for tracking Fortran module dependencies.

SCons: SCons[46] is a software construction tool written in Python for C/C++, Fortran and other languages. Its power lies in the fact that its configuration files are also written in Python, thus enabling the developer to use a genuine full-featured programming language to tackle the build process. SCons provides a scanner for Fortran 90 (and later) files that parses USE statements and uses string transformations to track module dependencies.

Many less well-known build tools exist, such as *waf* (used by Cisco, and to build Samba or Node.js), *wonderbuild* (obscure) or *ninja* (used to build Google Chrome), but seeing that WORHP already requires various more or less exotic dependencies, choosing from established build tools is probably wise. Despite the Fortran module dependency tracking facilities of SCons and CMake, WORHP has thus been relying on `make` with manually created and maintained makefiles to prevent burdening developers with further software and language dependencies. While this approach arguably offers a lesser degree of automation, it allows much finer control of the details of the build process, which is especially useful given the degree of complexity caused by automatic code generation and the mixed-language approach.

4.1.2. Basics of make

GNU `make` and other versions are controlled by writing *makefiles* that specify rules consisting of *targets* and their *prerequisite*, and recipes to specify the commands to execute. `make` is completely string-based (so `dir` and `dir/` are regarded as two different entities) and has uncommon semantics for variable expansion. Dependencies in `make` are formulated as `targets : prerequisites` pairs, for instance `foo.o : foo.c foo.h`. All dependencies are condensed into a single directed acyclic graph (DAG), whose nodes correspond to targets and prerequisites and whose edges are the dependencies imposed by rules. It is useful to note that not every DAG is a tree, because multiple targets may share a prerequisite (expressed in tree terminology, one child may have more than one parent), but every tree is a DAG.

It is obviously problematic if the dependency graph is incomplete, because this will result in incomplete dependency tracking, and thus in incomplete or faulty builds, or even in

²See http://www.cmake.org/Wiki/CMake_Fortran_Issues – retrieved 2013-01-30.

failure to build. It is therefore imperative to create complete dependency graphs, lest the (potentially subtle) errors introduced by incomplete dependency tracking thwart all efficiency and reliability gains acquired by using a build system.

Common workarounds for the incomplete dependency graph problem are repeated build passes, to ensure everything has been build, a tendency to rebuild more than actually needed, or in extreme cases, frequent complete rebuilds (`make clean && make all`), to ensure that all changes in the source files are picked up. Neither of these workarounds is acceptable, and the necessity to do so is an indication for serious deficiencies.

4.1.3. Directory structure

The source code and auxiliary files for building and maintaining WORHP are distributed over a number of directories, according to their function.

Config Configuration files for different platforms, loosely following the naming convention `<user/host>-<vendor><bitness>.mk`, e.g. `ztm-gnu32.mk` is a configuration file for the `ztm` hosts at ZeTeM using the GNU Compiler Collection (GCC)[25] to build 32-bit objects.

auto AutoGen definition file to generate the WORHP data structures and AutoGen include file with custom Scheme functions.

bin Target directory for executables and Dynamic Link Libraries (DLLs). Auxiliary scripts, CUTer problem lists and parameter files also reside in this directory.

blas Source code of a BLAS-subset and some LAPACK routines with dependencies.

core Core components that provide fundamental modules and definitions, data structures, workspace handling, preprocessor macros, initialization routines and auxiliary routines to all higher-level components.

doc Documentation directory, including a Doxygen configuration file that is capable of producing ridiculous amounts of HTML or \LaTeX documentation, as well as the WORHP user manual and tutorial.

examples Example codes for the different supported interfaces and languages, including tests of the Hotstart functionality and a variable-size Fortran version of the CUTer `mccormck` problem, which can be used to study the asymptotic behavior of computation times and memory consumption of WORHP for very large dimensions.

hsl When WORHP is built with support for one or more of the HSL linear solvers, their source code with dependencies needs to be placed in this directory. Some auxiliary files are used to build shared libraries or provide a dummy METIS routine.

include Usually empty. If SuperLU or the AMPL library are built by WORHP because they are not available on the current system, their C header files are moved here.

4. Solver Infrastructure

- interfaces** Contains code and auxiliary files to build the MATLAB and AMPL interfaces of WORHP.
- lib** Target directory for libraries. Subdirectories named after the target platform, e.g. `win32` or `linux64` help to differentiate between different builds. If SuperLU or the AMPL library are built by WORHP, because they are not available on the current system, they are moved here as well.
- mod** Target directory for Fortran module files.
- obj** Target directory for object code.
- qpsol** Source directory for the QPSOL component, together with the `leqsol` component that provides an abstraction layer between different linear algebra solvers and QPSOL or other WORHP components.
- res** Directory for global non-source files, like a release-Makefile template, the Visual Studio solution or a script for tracking DLL dependencies.
- testing** Source directory for test routines other than the AMPL interface.
- worhp** Source directory for the NLP-specific components; among these are the user interfaces and the NLP algorithm, but also the finite difference and BFGS modules.
- xml** Source directory for the XML module, the XML library abstraction layer, and the `xw` dedicated WORHP XML parser.

4.1.4. Pitfalls of recursive make

`make` is frequently used recursively to build individual components of the final build target. The most common form of this approach is encountered where a project has multiple (sub-)directories with source code; `make` will recursively spawn sub-makes corresponding to the directory structure. The advantage of using a recursive `make` hierarchy lies in the (relative) simplicity and independence of each sub-`make`, as long as there are no or only few cross-directory dependencies.

Using `make` recursively has important drawbacks, however, as described in detail in [51] and some of these also affected the initial build infrastructure of WORHP.

This initial build infrastructure made heavy use of recursive `make` to build the sub-libraries `libcore`, `libqpsol`, `libwxml`, `libworhp` and `libblas` according to the directory structure in section 4.1.3. The most significant obstacle to a simple, clean, and reliable recursive build infrastructure turns out to be existence of multiple cross-directory dependencies (mostly on files in `core`, but not exclusively), resulting in the dependency information to be incomplete and fragmented over multiple sub-makes. Listings 4.1 and 4.2 show the condensed annotated build output to illustrate some consequences of this condition.

Running `make example` would start the following recursive process (with the current recursion level indicated by `make`-like labels):

- [0] The top-level `make` is only informed about the dependency of the executables on `libworhp` and `libblas` and spawns two sub-makes to create these.
- [1] the `libworhp` target is informed about its dependency on `libcore`, `libqpsol` and `libwxml` and spawns three sub-makes to create them.
- [2] `libcore` is created.
- [2] `libqpsol` is created, its dependency on some objects in `libcore`³ is satisfied, so no sub-make is spawned.
- [2] `libwxml` only knows about its dependency on `libcore` and spawns a sub-make for it.
- [3] `libcore` has already been created, the sub-make has nothing to do.
- [2] `libwxml` is created.
- [1] `libworhp` is created.
- [1] `libblas` is created.

```

make -C trunk/worhp -I .. -k libworhp.a
make[1]: Entering directory 'trunk/worhp'
make -C trunk/core -I .. -k libcore.a
make[2]: Entering directory 'trunk/core'
[ . . . libcore compiled . . . ]
make[2]: Leaving directory 'trunk/core'
make -C trunk/qpsol -I .. -k libqpsol.a
make[2]: Entering directory 'trunk/qpsol'
[ . . . libqpsol compiled . . . ]
make[2]: Leaving directory 'trunk/qpsol'
make -C trunk/xml -I .. -k libwxml.a
make[2]: Entering directory 'trunk/xml'
make -C trunk/core -I .. -k libcore.a^
make[3]: Entering directory 'trunk/core'
make[3]: "libcore.a" has already been updated.
make[3]: Leaving directory 'trunk/core'
[ . . . libwxml compiled . . . ]
make[2]: Leaving directory 'trunk/xml'
make[1]: Leaving directory 'trunk/worhp'
[ . . . libworhp compiled . . . ]
make -C trunk/blas -I .. -k libblas.a
make[1]: Entering directory 'trunk/blas'
[ . . . libblas compiled . . . ]
make[1]: Leaving directory 'trunk/blas'

```

Listing 4.1: Recursive build sequence (clean rebuild)

When repeating the same `make` command, an even worse recursive `make` cascade as in listing 4.2 is caused on some systems, because some background process or `make` itself has altered the content of `/tmp`, `/etc` or `/home` in the first run. Since directory timestamps

³GNU `make` can express dependencies on members of static libraries by `foo.o : libbar.a(frob.o)`

4. Solver Infrastructure

on Linux change when the directory content is altered, these now have newer timestamps than the WORHP sub-libraries; presumably due to some of the many implicit rules of GNU `make`, the sub-libraries are considered as outdated and `make` spawns sub-makes to update them. The sub-makes then correctly deduce that nothing has to be done, since no source or object file is actually newer than the respective static library, and thus do not touch them, keeping their timestamp unmodified. This results in repeated spawning of redundant sub-makes, e.g. 3 times for `libcore` (as dependency of `libworhp`, of `libqpsol` and `libwxml`).

```
make -C trunk/worhp -I .. -k libworhp.a
make[1]: Entering directory 'trunk/worhp'
make -C trunk/core -I .. -k libcore.a
make[2]: Entering directory 'trunk/core'
make[2]: "libcore.a" has already been updated.
make[2]: Leaving directory 'trunk/core'
make -C trunk/qpsol -I .. -k libqpsol.a
make[2]: Entering directory 'trunk/qpsol'
make[2]: "libqpsol.a" has already been updated.
make[2]: Leaving directory 'trunk/qpsol'
make -C trunk/xml -I .. -k libwxml.a
make[2]: Entering directory 'trunk/xml'
make -C trunk/core -I .. -k libcore.a
make[3]: Entering directory 'trunk/core'
make[3]: "libcore.a" has already been updated.
make[3]: Leaving directory 'trunk/core'
make[2]: Leaving directory 'trunk/xml'
make[1]: Leaving directory 'trunk/worhp'
make -C trunk/xml -I .. -k libwxml.a
make[1]: Entering directory 'trunk/xml'
make -C trunk/core -I .. -k libcore.a
make[2]: Entering directory 'trunk/core'
make[2]: "libcore.a" has already been updated.
make[2]: Leaving directory 'trunk/core'
make[1]: Leaving directory 'trunk/xml'
make -C trunk/qpsol -I .. -k libqpsol.a
make[1]: Entering directory 'trunk/qpsol'
make[1]: "libqpsol.a" has already been updated.
make[1]: Leaving directory 'trunk/qpsol'
make -C trunk/core -I .. -k libcore.a
make[1]: Entering directory 'trunk/core'
make[1]: "libcore.a" has already been updated.
make[1]: Leaving directory 'trunk/core'
make -C trunk/blas -I .. -k libblas.a
make[1]: Entering directory 'trunk/blas'
make[1]: Nothing to be done for "libblas.a".
make[1]: Leaving directory 'trunk/blas'
```

Listing 4.2: Recursive build cascade due to newer file system timestamps

The initial recursive build system thus causes unnecessary overhead due to suboptimal use of the dependency tracking features of `make`.

WORHP has several cross-directory dependencies, most of which were only known to the respective sub-`make`, and in most cases were formulated unspecifically as a dependency on another sub-library. This explains the undesirable sub-`make` overhead observed in listing 4.2, and it also caused subtle problems resulting in wrong code in some cases where the data structure layout (defined in the `core` subdirectory) had been changed; to add insult to injury, running parallel `make` jobs to speed up build times showed nondeterministic behavior, and failed most of the time, either at link-time because of incomplete builds, or at compile-time because of dependencies unknown to `make` (most often Fortran module dependencies).

4.1.5. Non-recursive make

The pitfalls of a recursive `make` hierarchy can be circumvented by running only a single instance of `make` with *unfragmented* and *complete* dependency information. This ensures that builds are complete, that all changes of the sources are reflected in the build output, that no unnecessary recompilation or regeneration is triggered, and that `make` is able to run parallel jobs to speed up compilation, without causing errors; altogether, a build system constructed in this manner is reliable and performant.

The construction of a single-instance build system requires the precise specification of *all* dependencies for *all* (intermediate or final) targets. Fortunately, GNU `make` has facilities

- to include makefiles from a list of directories,
- to formulate multiple patterns with *static pattern rules*,
- for various operations on strings, filenames and paths,
- and to create *templates* which can be used to instantiate whole recipes with templated variable and target names.

WORHP uses a modular makefile system consisting of a main (central) makefile that provides global definitions and recipes and local makefile stubs that are included by the main makefile adding different portions of the solver, according to the target that is supposed to be built. The main difference to the recursive approach is the fact that only a single dependency graph is created from the main makefile and added to by the included makefile stubs. Therefore (assuming correctness), dependencies are complete instead of fragmented over various, individually incomplete graphs.

Technically, the key concept consists of lists of source files (according to language and some other distinctions) that need to be compiled and included in the library or executable to be built; for C files that do not depend on special includes or compiler flags, this is `SRC_C`. The variable `SRC_DIRS` lists the subdirectories to include and is modified according to the current configuration. When triggering a build, `make` iterates through `SRC_DIRS` and for every subdirectory `dir` tries to include `dir/build.mk` and `dir/modules.mk`. The former is mandatory, since it defines the source files in `dir` and their complete dependencies, while the latter is optional and used to track Fortran module dependencies.

The `build.mk` makefile stubs add the source files in `dir` to `SRC_C` (and the other lists of source files), if necessary using conditionals to adapt to different configurations, and

4. Solver Infrastructure

```
SRC_C += \  
    core/C_std.c \  
    core/C_Worhp_Data.c  
  
SRC_F90 += \  
    core/std.F90 \  
    core/Worhp_Data.F90  
  
AUTOGEN_FILES += \  
    core/C_Worhp_Data.h \  
    core/C_Worhp_Data.c  
  
# Autogen dependencies  
core/C_Worhp_Data.h: core/include-Worhp_Data.h  
core/C_Worhp_Data.c: core/include-Worhp_Data.c  
  
# C source dependencies  
obj/C_std.o: core/C_std.c core/C_std.h  
obj/C_Worhp_Data.o: core/C_Worhp_Data.c core/C_Worhp_Data.h \  
    core/C_std.h
```

Listing 4.3: Strongly abridged version of `build.mk` of the core subdirectory. Since each makefile stub is included by the main makefile, all rules need to be formulated in terms of the root directory, instead of the local subdirectory, since this is where the main makefile is located.

define dependencies of the object files generated from them; the latter can be generated by using `gcc -MM` with some manual or automatic post-processing. Listing 4.3 shows an exemplary makefile stub for the core subdirectory.

```
# Manually generated Fortran module dependency file  
obj/cs.o : obj/std.o  
obj/ccm.o : obj/std.o obj/cs.o  
obj/std.o :  
obj/timer.o :  
obj/Worhp_Data.o : obj/std.o obj/cs.o obj/ccm.o obj/timer.o \  
obj/Worhp_Members.o : obj/std.o
```

Listing 4.4: Slightly abridged version of `modules.mk` of the core subdirectory. The module dependencies are formulated in terms of object files instead of module files, since this approach is portable across different compilers.

The `modules.mk` Fortran dependency files are only present in subdirectories with Fortran files that depend on or define modules; figure 4.1 illustrates the inter-module dependencies of WORHP. The Fortran dependency files track module dependencies by formulating them in terms of object files. While all common Fortran compilers generate module files, which would theoretically allow `make` to keep track of changes through them, the mapping

$module \mapsto module\ file\ name$ differs between compilers. To avoid configuration-dependent creation of vendor-specific module dependency rules, they are expressed in terms of object files, whose naming *is* independent of the compiler in use. Using module files instead would only be superior to the current approach, if a compiler offered an option to only produce a module file, and if this mode of operation were noticeably *cheaper* than proper compilation. By using sophisticated comparisons between new and old module files, recompilation of dependent files could be avoided, if a source file is recompiled, but the module remains unchanged. Unfortunately, module files created by `gfortran` contain a timestamp, hence the comparison has to be performed on the content only.

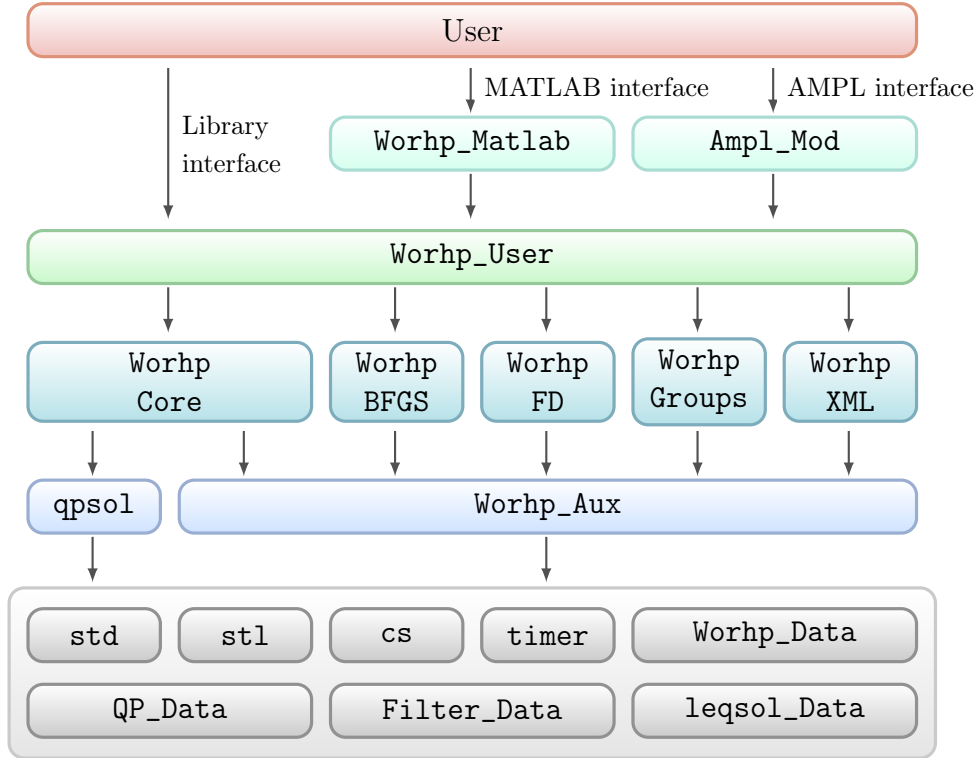


Figure 4.1.: Fortran module hierarchy in WORHP.

With the makefile stubs defining the source files and dependencies, the main makefile creates a list of object files (**OBJ_C** and others) to compile by string transformations on the source file lists (**SRC_C** and others). The main makefile further defines libraries (static or shared, depending on the configuration), and the final target mostly depends on the library. Building a target then triggers a chain of transitive dependencies $target \rightarrow library \rightarrow object\ files \rightarrow source\ files$. Given rules to compile source to object files, the main makefile has all necessary information to build its known targets.

The approach of splitting the makefile into one central and various included stub makefiles keeps changes localized, i.e. changes in one subdirectory only concern the local stub makefiles, while still enabling `make` to build an unfragmented dependency graph. Most importantly, this allows to reliably perform parallel builds, which is an inbuilt capability

4. Solver Infrastructure

of **make**: All intermediary or final targets that have no mutual dependencies can be compiled in parallel, taking advantage of today’s multi-processor systems. Observations on a modern quad-core workstation indicate more than three-fold speedup of a complete rebuild through parallel building (using the `-j` flag).

Selecting build configurations

Since WORHP does not use a capability-based build tool like Automake, paths to tools and available external dependencies (like SuperLU, the AMPL solver library, or MATLAB) have to be stored and organized for every build platform, particular machine, or setup, with the option to easily and non-disruptively add to the list. The obvious solution is the creation of *machine configuration files*, one for each particular machine or setup. The machine configuration files follow the naming convention `<user/host>-<vendor><bitness>.mk` and are stored in the `Config` directory. Each machine configuration file defines a number of variables (actually constants), which are later used by the makefile. Since they are directly included, these files are written in makefile-syntax. Listing 4.5 shows an example.

```
# Compiler and system
COMPILER := gnu
BITNESS  := 64
BUILD    := Linux

# System-specific paths
PREFIX := /home/agbueskens
LIB_PATH_AMPL := $(PREFIX)/lib64
INC_PATH_AMPL := $(LIB_PATH_AMPL)
LIB_PATH_SUPERLU := $(PREFIX)/lib64
INC_PATH_SUPERLU := $(PREFIX)/include/superlu

# compilers and flags
CC := gcc
XC := g++
FC := gfortran
```

Listing 4.5: Abridged machine configuration file, defining paths and compilers for a specific machine or setup

To select the currently active machine configuration file, WORHP has a single, central *build configuration file*, `Config.mk`, shown in listing 4.6. Besides selecting the machine configuration file, the build configuration file allows to toggle various WORHP modules, enable building of dynamic (shared) libraries, select the enabled linear algebra solvers, and add various debugging features that are compiled into the solver only on specific request—these debugging features consist of additional output for license management, the XML module, memory management and others. To avoid any runtime overhead, these features are controlled by the preprocessor, instead of a runtime parameter.

```

PLATFORM = cluster-gnu64
COMFLAGS = optimise

ENABLE-MA97      = yes
ENABLE-MA57      = no
ENABLE-SUPERLU   = no

ENABLE-WORHP     = yes
ENABLE-QPSOL     = yes
ENABLE-MATLAB    = no
ENABLE-XML       = yes
ENABLE-LICENSE   = no

DEBUG-XML        = no
DEBUG-MEM        = no
DEBUG-LIC        = no

```

Listing 4.6: Abridged build configuration file, defining modules to be included and various debugging options

Integration with automatic code generation

To ensure that changes to code generation definition file are always represented in the build output, the build infrastructure is aware of the dependencies of auto-generated files on both the definition and template file and a recipe is provided to instruct `make`, how to update out-of-date code. This extends the abovementioned transitive dependency list to *target* \rightarrow *library* \rightarrow *object files* \rightarrow *source files* \rightarrow *definition/template file*. The variable `AUTOGEN_FILES` is used by the makefile stubs to define all auto-generated source files (in addition to specifying them in `SRC_C` or similar).

Due to the timestamp-based operation of `make`, the respective files are only re-generated if required, and otherwise left alone. This is important, since not every build platform is required to provide AutoGen, but instead just compiles the auto-generated code. In some instances, if the WORHP source is checked out from Subversion, timestamps may suggest wrongly that the auto-generated files need updating; if AutoGen is present on the respective machine, no harm is done, but in the case where it is *not*, the code generation recipes in the main makefile need to be deactivated, or a dummy no-op AutoGen can be provided.

4.2. Version and configuration info

In some cases, especially when checking for availability of a feature or status of a known bug, it is sometimes necessary to be able to query the version and configuration information of a given WORHP binary.

AMPL interface

The AMPL interface has a straightforward way of providing version information: Since the interface is implemented as an executable binary that is run by the AMPL executable, it may also be run as a stand-alone binary with command-line arguments. In keeping with established conventions, running `worhp -v` will generate output of the form

```
version 1.0.2014 (Linux 2.6.18), driver(20130122), ASL(20090316)
```

i.e. printing versions of the WORHP library, the kernel, the AMPL driver, and the ASL backend. The driver date is derived from subversion, using the `svn info --xml` command that returns—among other pieces of information—the date of the last change in an XML node of the form `<date>2012-01-23T14:47:38.546875Z</date>`, which can be parsed using `sed`, and subsequently integrated into the binary through preprocessor defines. An alternative and less intricate approach is to use the `svn:keywords` functionality of Subversion, although this has neither been tested nor implemented, yet.

Library interface

The library interface exports the `WorhpVersion` routine

```
void WorhpVersion(int *major, int *minor, int *patch)
```

that provides a programmatic way of querying the solver version, but has two shortcomings: One is forced to first compile and link a test program to then query and print the version, and it does not provide information on the enabled modules of WORHP, or the compiler used to create the library.

On Linux platforms using the ELF binary format, there is an elegant solution to this problem, which is also used by the GNU C library; besides providing the C standard library functionality, it can be executed to produce a small amount of text output stating the library version and included modules, as demonstrated in listing 4.7.

To generate a shared library that doubles as an executable, some intervention into the produced ELF format is necessary: the library has to specify an *interpreter* and an *entry point*.

```

$ /lib/x86_64-linux-gnu/libc.so.6
GNU C Library (Ubuntu EGLIBC 2.15-0ubuntu10.3) stable release version
↳ Copyright (C) 2012 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.
Compiled by GNU CC version 4.6.3.
Compiled on a Linux 3.2.30 system on 2012-10-05.
Available extensions:
  crypt add-on version 2.1 by Michael Glad and others
  GNU Libidn by Simon Josefsson
  Native POSIX Threads Library by Ulrich Drepper et al
  BIND-8.2.3-T5B
libc ABIs: UNIQUE IFUNC
For bug reporting instructions, please see:
<http://www.debian.org/Bugs/>.

```

Listing 4.7: Output of eglibc on Ubuntu 12.04

Interpreter

The ELF *interpreter* is specified as a simple character string that is placed into the appropriate section of the ELF object by a compiler directive; when producing executables, this is automatically performed by the compiler, but omitted when producing shared libraries. The interpreter of existing executables can be identified with the `readelf` tool:

```

$ readelf -l /bin/ls

Elf file type is EXEC (Executable file)
[...]

Program Headers:
[...]
  INTERP      0x0000000000000200 0x000000000000400200 0x000000000000400200
               0x000000000000001c 0x000000000000001c  R      1
               [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
[...]

```

The ELF interpreter, i.e. the helper program for shared library executables on 64-bit Linux platforms is `/lib64/ld-linux-x86-64.so.2`, and its 32-bit counterpart is `/lib/ld-linux.so.2`; conditional compilation needs to be employed to select the appropriate one for the current platform. Despite the ever-increasing diversity and resulting incompatibilities between the various Linux distributions, using these two interpreters (actually runtime-linkers) seems to remain a minimum consensus.

To manually add this ELF `.interp` section to a binary, we use a C code snippet defining the interpreter as a string and using the GNU `__attribute__` syntax to specify the ELF section:

```
const char elf_interpreter[] __attribute__((section(".interp"))) =  
    "/lib/ld-linux.so.2";
```

Entry point

To specify the *entry point*, which again is automatically performed by the linker when producing an executable, the `-e` option to the linker specifies the function to be used as entry point (usually `main` for executables). To embed this into the GCC command-line that links the shared library, the `-Wl` syntax for passing options to the linker has to be used, resulting in

```
-Wl,-e,entry_point
```

to define `entry_point` as the entry point function; commas need to be used to prevent the usual tokenization by the compiler driver of the linker options at whitespace, otherwise `-e entry_point` would be passed to the compiler instead of the linker (resulting in complaints, because GCC does not understand this option).

Combining both these techniques is used to have `libworhp.so` print informative output shown in listing 4.8.

```
$ libworhp.so  
WORHP shared library for Linux, version 1.0.2014  
Copyright (C) 2013 SFZ Optimierung, Steuerung und Regelung.  
Distribution of WORHP is subject to terms and conditions.  
The authors disclaim all warranties, including without limitation,  
merchantability or fitness for a particular purpose.  
  
Available modules:  
  MA97 solver support  
  SuperLU solver support  
  XML module  
Compiled with:  
  gcc (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3  
  GNU Fortran (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3
```

Listing 4.8: Sample output of shared `libworhp` on Linux platforms

4.3. Testing Approach

Debugging is twice as hard as writing the code in the first place. Therefore, if you write code as cleverly as possible, you are, by definition, not smart enough to debug it.

(Brian Kernighan)

Although the development process of WORHP has never been subject to strict formalization, it does share some characteristics with the *test-driven* development paradigm.

Test-driven development is characterized by creating tests for desired functionality *before* creating the software to be tested. This approach guarantees good test coverage and additionally provides a simple metric for changes to the software. Regression Testing is a canonic addition to this development method, its main difference being that its tests provoke known (and usually corrected) defects in the software, instead of testing desired functionality. In the combined approach, the test suite is first populated with tests for functionality that is to be developed, and over the lifespan of the software, tests for identified bugs are added to prevent regressions. The GNU Compiler Collection embraces this approach and has accumulated over 40,000 such tests over the course of its 25-year lifetime.

Since WORHP is built to solve NLP, its test suite will naturally consist of optimization problems, accompanied by the technical infrastructure needed to run all tests and verify their outcome. To make a meaningful statement about WORHP's capabilities, the test set needs to be as diverse as possible. The default test suites are a 920-problem set written in AMPL, consisting of a subset of CUTeR plus the Schittkowski problems from [63], and the 68-problem COPS test set. The *genuine* CUTeR collection is provided as 1147⁴ files in SIF format. Since WORHP does not (yet) have a SIF interface, the AMPL alternative is used for testing. Although neither is a true subset of the other, we shall refer to the set as “(AMPL-)CUTeR”.

Both CUTeR and COPS are recognized as standard test sets for benchmarking and comparing NLP solvers by the mathematical optimization community, with COPS even being specifically designed as solver comparison tool, and CUTeR containing the 124 Hock/Schittkowski NLP test problems[32] many of which were specifically written to exploit known weaknesses in SQP solvers.

Testing in our sense means to feed all or a subset of the optimization problems of the abovementioned test sets to the solver. Test results are derived from some canonic quantities for NLP, among which are the objective function value, the constraint violation, an optimality measure, iteration and function evaluation counts and user time, and the solver status (solved to optimal or acceptable level, or some error condition). Test runs, such as a typical before/after check when a modification is introduced, can then be compared according to an arbitrary performance measure.

⁴as of 2012-06 – the collection is updated irregularly, see <ftp://ftp.numerical.rl.ac.uk/pub/cuter/mastsif.updates>

4. Solver Infrastructure

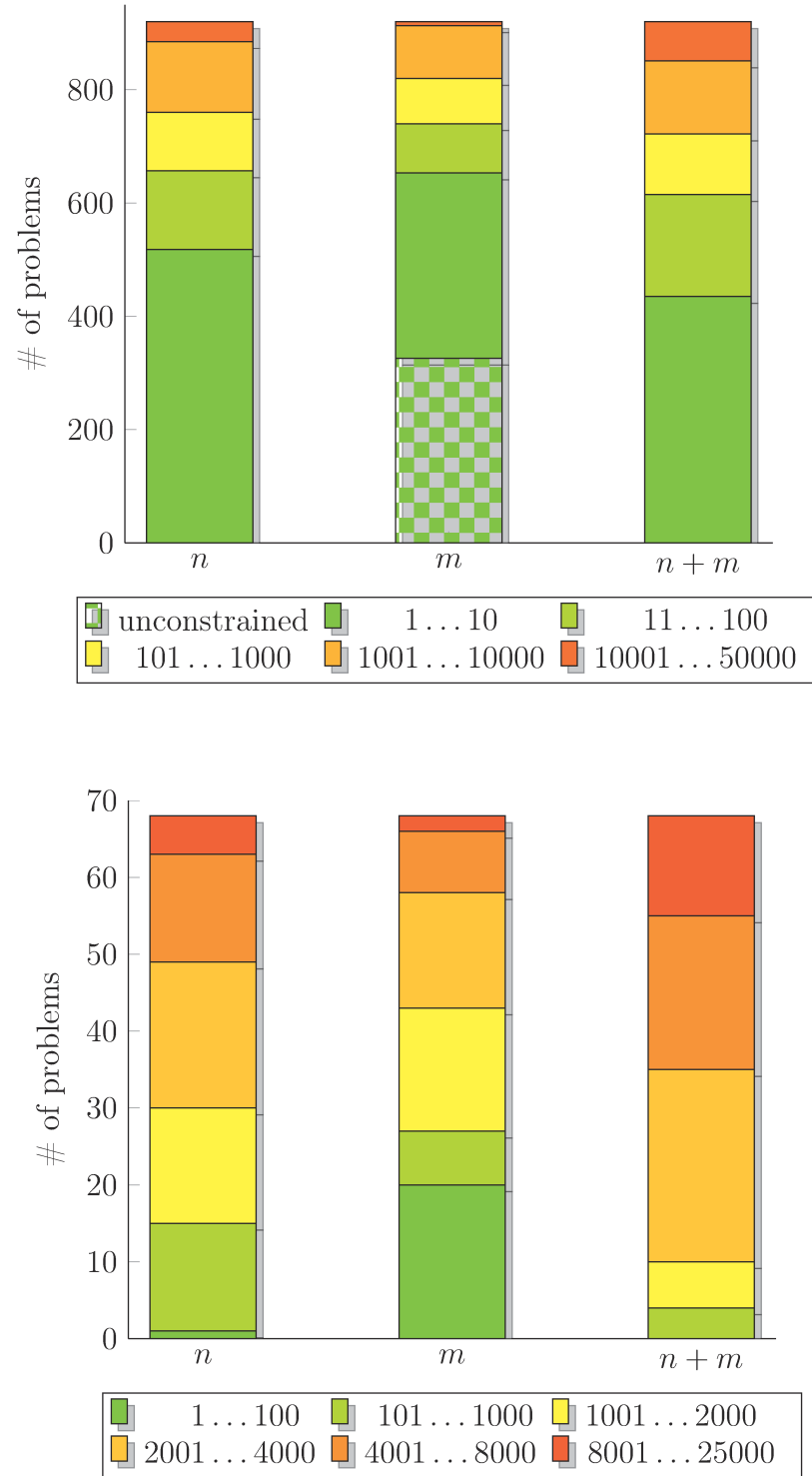


Figure 4.2.: Size distributions of the (AMPL-)CUTer and COPS 3.0 test sets. Sizes are measured as number of variables n , number of constraints m , and sum of both. The graph shows the size distribution with respect to five quasi-logarithmically scaled categories.

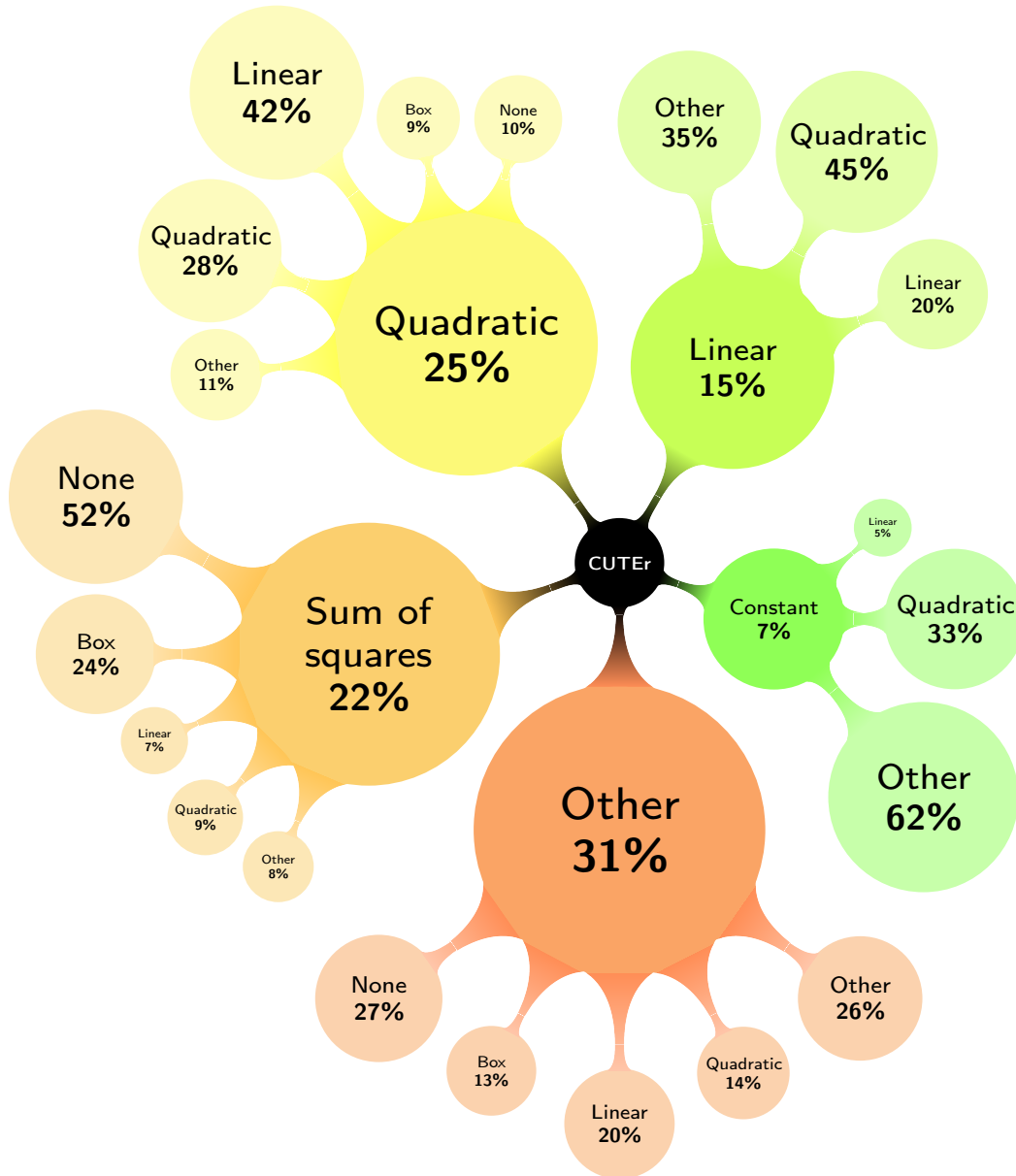


Figure 4.3.: Categories of the (AMPL-)CUTer test set: The first-level (inner) nodes categorize the objective, and the second-level (outer) nodes the constraints. For conciseness, network constraints were counted as linear ones, fixed variables as box constraints, and no objective as constant objective.

4.4. Testing Infrastructure

This section is devoted to the technical infrastructure that enables testing. Historically, this part of WORHP has undergone very major development steps, starting from bash- and file-based sequential list processing and evaluation through generated HTML code, until reaching today’s intricate parallel infrastructure implemented in Perl.

4.4.1. Parallel testing script

Given that WORHP’s primary tests are written as AMPL models, they can be run independently of each other, which means that the process of iterating over a test set can be trivially parallelized. The solver infrastructure provides a script that runs the solver on each of a given set of AMPL models, thereby gathering and condensing the results into a final summary (see listing 4.9 for an example summary).

```

Total ..... 313
  Successful ..... 313 (100.00%)
    Optimal ..... 313 (100.00%)
      Optimal Solution Found ..... 311
      Optimal Solution Found by Low-Pass Filter ..... 1
      Optimal Solution Found, maybe non-differentiable . 1
    Acceptable ..... 0 ( 0.00%)
  Unsuccessful ..... 0 ( 0.00%)

wall      7.08  (7.08s)
user      4.79  (4.79s)
sys       0.78  (0.78s)
speedup   0.7

```

Listing 4.9: Summary for a run of the Hock/Schittkowski test set on a dual-core machine.

The testing script consists of two Perl scripts `run.pl` and `node.pl` and three Perl modules containing auxiliary subroutines. The interaction between both scripts is a master-slave constellation, where a single *master* (namely `run.pl`) maintains a list of *jobs* and assigns them to idle workers (*slaves* – `node.pl`), until all jobs have been done; a *job* in this context is the process of running the solver on a single AMPL model, and parsing the solver output.

Communication with the user happens through `run.pl`, which spawns one instance of `node.pl` per available CPU core, then waits for feedback from the slaves, starting one communication thread per “live” slave. As soon as slaves report in as being alive, the master starts assigning jobs to the slaves, and waits for the results, until the job list is empty and all slaves reported back their results, in which case the results are processed and a summary is presented to the user. Figure 4.4 illustrates the communication scheme in more detail.

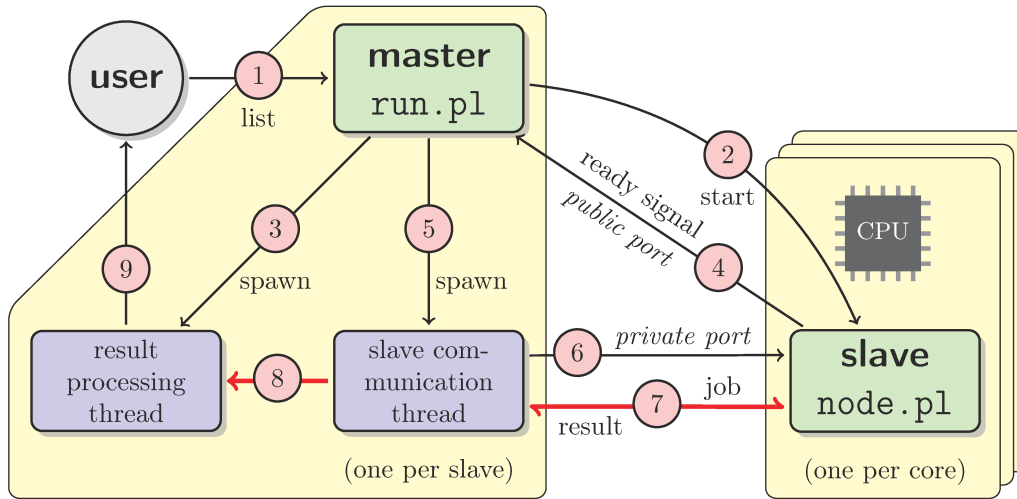


Figure 4.4.: Master-slave communication pattern of the testing script.

1. The user starts the script with a list of problems to be tested.
2. The master starts one slave process per available core.
3. The master spawns a single thread to process the results.
4. Each slave sends a ready signal to the master.
5. The master spawns a communication thread for every slave that reported in.
6. Each communication thread establishes a private channel to its slave.
7. The communication threads hand out jobs to the slaves and wait for results.
8. The communication threads append each result to a single shared queue.
9. The result processing thread processes and condenses the queued results.

Two elements are pivotal to this construction: The bookkeeping of the job list and the incoming results, and the communication between master and slave.

The first bash-based implementations provided the job list through a file that was used like a stack by the workers. Using two or more processes or threads in parallel, however, requires a locking mechanism to prevent duplicates or missing problems, but bash provides no support for locking, other than filesystem-based approaches. This in turn causes massive problems with parallel read/write access to the job list or lock files. Even the PVFS available on the ZeTeM cluster does not provide reliable parallel file access, and suffers from even higher latencies than local or NFS file systems; its high performance property seems to apply to few operations on large files, but not to many operations on small files.

Summing up the above, bash and file-based implementations are too unreliable and much too slow for efficient parallelization of jobs. The already high latency of (file) I/O operations is further aggravated by the fact that a majority of the CUTer problems only have few variables and constraints and are therefore solved in few milliseconds; figure 4.2 shows that roughly half of the set has less than 10 variables, and two thirds less than 100,

4. Solver Infrastructure

and even fewer constraints. It is therefore necessary to have sub-microsecond access to the job list, hence a performant implementation has to hold it in memory. This requires explicit communication between master and slave, because the slaves cannot retrieve jobs from a globally visible file anymore. The latency of this communication link needs to be lower (preferably by at least one order of magnitude) than the time spent on executing the job.

Since bash does not provide adequate tools for either, the current implementation uses Perl, with the master-slave communication implemented through Ethernet connections and the job and result lists implemented on top of a thread-safe deque.

The deque is a `Thread::Queue`, which is fast and thread-safe. The Ethernet connection on multi-core machines (i.e. an SMP setup) will be routed through the loopback device, which incurs very low latencies, and in clusters or MPP setups typically has sub-microsecond latency. The ZeTeM cluster, for example, is equipped with high-performance Infiniband and Gigabit Ethernet, which have proven to achieve sufficiently low latencies for running hundreds of jobs in parallel without performance problems—ping latencies usually lie in the 50 – 100 μ s range and thus do not dominate the total time, even for small problems.

Higher network latencies are commonly found in distributed (cloud) computing, where the individual nodes are not connected through a local network but over regular internet connections, which results in typical latencies in the 20 – 40 ms range, i.e. between two and three orders of magnitude greater than in local clusters. Grouping individually small tasks into bigger jobs that again take (noticeably) more time to finish is an option to keep the network latency from dominating the total time and increase the CPU workload.

The advantage of using TCP connections is that the test infrastructure can be run without modification on very different machine types and network topologies, such as a multi-core desktop machine, a dedicated compute cluster, or a distributed computing grid. Only the method of starting the slaves has to be tailored to the present infrastructure: On a multi-core machine, the slaves are started as child processes, while on the ZeTeM cluster, the PBS scheduler distributes and starts the slaves.

Termination

With all jobs distributed, the next important question concerns the termination of the parallel run script: How does the script notice that all jobs have been completed? The least-technical approach is to compare the total number of jobs against the number of received results; once the latter reaches the former, all jobs have been completed. This approach is simple and efficient, but vulnerable to faults: A single job that fails to produce results will cause the whole run to fail, since the final processing and condensing of all results can only be triggered as soon as all results are available. It is therefore necessary to find a more fault-tolerant solution. In case of failing jobs, this will result in incomplete results, but flawed results are preferable over no results at all.

The chosen approach monitors the number of *slave communication threads* instead of the number of *results*: After starting the slaves and the result processing thread (2 and 3 in

figure 4.4), the master starts a thread that is waiting for incoming TCP connections on the public port (4), and for every such connection spawns a communication thread (5) that uses a dedicated private port to connect to its associated slave (6). If the slave dies or if it reported its results and the communication thread has no more jobs to submit, it terminates itself. Monitoring the number of live slave communication threads is therefore a feasible termination criterion that is robust against faults: If all slave communication threads have terminated, all jobs have been sent to the slaves and either returned results or were lost. To sum up, this approach is more robust than counting results, but still has following flaws:

- It is more complex than the result-counting approach; in particular, the main thread first has to wait for at least one slave communication thread to *start*, before waiting for all threads to *finish* again—without this barrier, there is a race condition in which the main thread terminates right away, if no slave communication thread has been spawned, yet. To implement such a barrier, Perl’s `cond_wait` and `cond_signal` are used.
- The master thread that listens for incoming TCP connections on the public port will never terminate; as soon as all possible slaves have been started, no additional connections will come in, leaving the master thread blocked. Since the Perl thread implementation does not offer mechanisms to terminate a thread from outside, the thread cannot terminate or be terminated gracefully. Fortunately, this is of little practical relevance, because Perl terminates all threads if the spawning program ends, but this provokes warning messages about “running and detached threads” that may irritate inexperienced users.
- Slaves may suddenly die, either due to errors in the called program (in our case the WORHP binary) or due to external influences, such as hardware errors, hung-up kernels or out-of-memory situations. This approach was devised to handle dying slaves gracefully, but should nevertheless try and obtain results for every job. If the slave communication thread notices that its slave has died, it therefore re-submits the last job to the job queue to have another slave process it. This is reasonable if the slave died due to some external influence, but will result in a cascade of dying slaves, if the job is responsible.

4.4.2. Solver output processing

Condensing the results is concerned with parsing the solver output, performing operations (string comparisons, count, sum, mean, ...) on various numeric quantities or status strings and formatting the results for output—step 9 in figure 4.4. The parsing is performed by redirecting the solver output through a solver-specific parser (since the output formats of different NLP solvers differ widely), which performs line-by-line pattern matching to extract counts of function evaluation, major and minor iterations, as well as timing information, achieved results (objective, feasibility and optimality) and the termination status. Listing 4.10 shows an example of the condensed output on the master node.

4. Solver Infrastructure

The solver output parsers are implemented as *stateless* functions invoked for every line of solver output, which complicates matters, if some piece of information has to be derived from multiple lines – most prominently the constraint count, which is often displayed over various lines, differentiating between equality and inequality constraints, or upper/lower/both bounds for box constraints. A workaround is to make the output parsing routines *stateful* by using global variables, such that summation over various lines is possible.

The solver output parsers (one for each NLP solver) have proven to be quite fragile, since subtle changes in the output format of the solver may cause portions of the parser to fail, which in turn leads to undefined values in the sweep script.

Timing the solvers is performed by starting them through the external `time` command, providing wall time, user time and sys time. This approach was chosen, because some NLP solvers do not provide timing routines, whereas the internal timing results of others differ significantly from external timing. A uniform external timing tool is therefore the fairest comparison between different solvers.

Performance Considerations

Running the (AMPL-)CUTer test set produces 111,110 (sic) lines of output⁵ with an overall 8,360,502 bytes, amounting to an average of 120 lines per problem, and roughly 250 lines or 18,200 bytes per second on an Intel i7-2600 system with 8 (virtual) cores. Scaling up to the 320 cores of the ZeTeM cluster, this could theoretically produce 10,000 lines per second with 728 KB/s (≈ 5.6 Mbps) of solver output per second to be sent through the network and parsed by the master node. While a 100 Mbit Ethernet connection can easily handle 5 Mbps of data, it is unclear whether Perl can parse this continuous data stream in real-time. In extreme cases, this could lead to overflowing buffers and data loss and consequently in incorrect results.

With the sweep functionality being embedded into the parallel run script, the major parsing effort is therefore delegated to the slaves to keep the CPU load of the master node and the amount of network traffic as low as possible. The master node only receives the parsed results, which are sent over the network in the form of fixed-layout strings like

```
DONE tag=000000:name=hs005:n=2:m=4:nf=11:ng=0:iter=10:
  ↳ obj=-1.9132229545E+00:con=0.0000000000E+00:kkt=7.7493642313E-08:
  ↳ status=Optimal Solution Found:tuser=0.00:tsys=0.00
```

The master caches the received result strings in a `Thread::Queue` deque, which feeds into the result processing thread (step 8 in figure 4.4). To process, each string is tokenized, and Perl's inbuilt regular expression matching is applied to extract the values. Information on queue length is indicated below the test results; low values indicate that the result processing thread is able to keep up with the incoming stream of results, while high values indicate overload.

⁵trunk, revision 1949, default settings

```

$ run.pl worhp Lists/test
Using (up to) 4 threads to run 12 problems
Sep 19 2012 10:15:17 [11] aircrfta ..... 3 iterations : Optimal Solution Found
Sep 19 2012 10:15:17 [10] aircrftb ..... 17 iterations : Optimal Solution Found
Remaining: 3pk, airport, aljazzaf, allinit, allinitc, allinitu, alsotame, argauss, arglina, arglinb
Sep 19 2012 10:15:17 [ 9] aljazzaf ..... 20 iterations : Optimal Solution Found
Sep 19 2012 10:15:17 [ 8] allinit ..... 5 iterations : Optimal Solution Found
Sep 19 2012 10:15:17 [ 7] 3pk ..... 28 iterations : Optimal Solution Found
Sep 19 2012 10:15:17 [ 6] allinitu ..... 8 iterations : Optimal Solution Found
Sep 19 2012 10:15:17 [ 5] allinitc ..... 6 iterations : Optimal Solution Found
Sep 19 2012 10:15:17 [ 4] alsotame ..... 4 iterations : Optimal Solution Found
*** All jobs sent to workers ***
Sep 19 2012 10:15:17 [ 3] airport ..... 10 iterations : Optimal Solution Found
Sep 19 2012 10:15:17 [ 2] arglinb ..... 3 iterations : Optimal Solution Found
Sep 19 2012 10:15:17 [ 1] arglina ..... 1 iterations : Optimal Solution Found
Sep 19 2012 10:15:17 [ 0] argauss ..... 54 iterations : Acceptable Solution Found
Total ..... 12
Successful ..... 12 (100.00%)
  Optimal ..... 11 ( 91.67%)
    Optimal Solution Found .... 11
  Acceptable ..... 1 ( 8.33%)
    Acceptable Solution Found . 1
  Unsuccessful ..... 0 ( 0.00%)

wall 0.48 (0.48s)
user 0.22 (0.22s)
sys 0.05 (0.05s)
speedup 0.5

Results written to /home/dwassel/worhp/trunk/bin/results.worhp.20120919.101517

(Queue length: max 1, avg: 0.15)

```

Listing 4.10: Complete output of the run script for a 10-problem list on a quad-core machine.

4.5. Parameter Tuning

Sed quis emendabit ipso emendator?
(But who will optimize the optimizer?)

(with apologies to Decimus Iunius Iuvenalis)

Being a complex piece of mathematical software, WORHP can be influenced by more than 100 runtime parameters. While the effects of some of them are obvious, for instance `MaxIter`, `Timeout` or `TolFeas` (which are probably the most-used parameters and easily understood by users), the influence of many other parameters is much more difficult to assess, either because it is subtle and its details are intricate and require deep knowledge of WORHP’s algorithmic or theoretical background, or because it changes major algorithmic details and thus potentially leads to completely different sequences of iterates. When regarding groups of parameters, instead of single parameters in isolation, the complexity of assessing parameter changes is complicated further (and probably growing exponentially with the size of the group).

4.5.1. Why tune solver parameters?

If the proper choice of parameters is difficult and mostly unintuitive, it is natural to ask why users should struggle with it; since WORHP (and every other sufficiently sophisticated software) is *generic*, its default parameters try and reflect a minimum consensus for all possible NLP, which is a huge and very diverse problem class—when viewed pessimistically, this will necessarily be a setting that works badly for everyone, the software analogue to “one size fits all”. Thus, though not strictly necessary, tuning is often emphatically advised when one intends to seriously solve optimization problems.

“Tuning” may have very different meanings to different users and applications:

- In the prototype phase, parameter tuning may be necessary to achieve convergence at all, since the model still has rough edges.
- If the underlying model is complex and somehow “hard” to solve, or if the achievable precision is limited, careful tuning is often necessary to obtain satisfactory results.
- If the optimization result is subject to high precision requirements, parameter tuning can ensure that these requirements are met.
- If a class of models has to be minimized frequently, the aim of parameter tuning will be to speed up the optimization process.

Tuning is worthwhile, because the possible returns can be significant, but it requires a) deep mathematical, numerical and algorithmic insight into both the solver and the application, or b) great amounts of raw computing power to exhaustively sweep the parameter space on a set of representative examples.

Since approach a) is essentially limited to experienced NLP developers who are *also* deeply knowledgeable about the potential application, approach b) is, though inelegant, a much

more feasible and powerful method for parameter tuning. The advent of multi-processor machines, compute clusters, or even non-uniform grids of distributed computers make a tremendous amount of computational power available to any application that can run in parallel or distributed. A systematic, exhaustive sweep of the parameter space is such an application, being *embarrassingly* parallel⁶.

4.5.2. Sweeping the Parameter Space

In WORHP terminology, a *sweep* consists of running the solver on a given optimization problem (or set of optimization problems) for a number of pairwise distinct⁷ parameter settings and condensing the results in a human-readable way. However, generating the parameter settings quickly (exponentially so) becomes an onerous task when sweeping over more than a single parameter. Consider the following example:

Sweep over all feasible, i.e. `BFGSminblocksize` \leq `BFGSmaxblocksize` combinations of

$$\begin{aligned} \text{BFGSminblocksize} &\in \{1, \dots, 10\}, \\ \text{BFGSmaxblocksize} &\in \{1, \dots, 10\}, \\ \text{BFGSrestart} &\in \{10, \dots, 50\}, \end{aligned}$$

which results in $\frac{1}{2} \cdot 10 \cdot (10 + 1) \cdot 41 = 2255$ distinct parameters triples. Generating the corresponding parameter files manually, or even semi-automatic is clearly out of the question. Hence the generic testing script described in section 4.4.1 was (significantly) extended to perform parameter sweeps, to the point where each test run is algorithmically considered as a sweep over a single parameter setting. To perform a sweep, the script generates the parameter settings, runs all problems with all parameter settings and condenses the (potentially large) quantity of results into tables and plots for simple evaluation. The sweep script also runs all problems with the reference parameter set to establish a baseline for comparison. The different parameter settings are differentiated by *tags*, which are 6-digit numbers, starting from 000000 for the reference run, and counting up; this convention assumes that there will never be more than $10^6 - 1$ parameter combinations, but is reasonably simply extended to more digits, should the need arise.

Specifying Parameter Combinations

The sweep script has three major modes for generating parameter variations of a parameter: Each parameter can take all values from a given list, iterate through a range of values or be taken as random sample. All three modes can be freely mixed.

Communication with the sweep script takes place through command-line arguments to specify the solver executable and the file containing a list of problems to iterate over, and a simply structured text file to define the sweep parameters. Listing 4.11 shows an

⁶This is indeed the common technical term for problems that can be parallelized with little or no effort.

⁷Not a functional requirement, but the alternative is only really useful for consistency checking.

```

# how many random samples to use
samples 200
# mode      name                type      start      stop      op      step
range      ArmijoBeta           double    1.5e-1     3.5e-1    +       0.001
range      RelaxStart            double    1.0e-4     1.0e+4    *       1.1
rand       qp.ipBarrier                double    1.0e0      1.0e2
list       MaxIter                    int       100 500 1000 5000 10000
list       UserHM                  bool

```

Listing 4.11: Example parameter combination input file for the sweep script. This setting would result in $\left(\frac{0.35-0.15}{0.001} + 1\right) \left\lceil \frac{\log(10^8)}{\log(1.1)} \right\rceil \cdot 200 \cdot 5 \cdot 2 \approx 7.8 \cdot 10^7$ combinations, which is unrealistically high, even with a powerful compute cluster available, since the parameter files alone would occupy roughly 900 GiB (at 4 KiB block size).

(unrealistic) example of such a sweep file, demonstrating the possible modes. Each item consists of at least 3 tokens, separated by arbitrary whitespace: the first token specifies the mode, the second one the (case-sensitive) complete parameter name, and the third its type (since the script is agnostic of this). The meaning of subsequent tokens depends on the mode:

- In list mode, each token is a list item. For boolean types, where only one sensible list exists (modulo order), no list specification is required or recognized.
- In range mode, four additional tokens are expected, defining an interval and an operation to produce the next variation, i.e. the iteration is defined by $p_0 := \text{start}$ and $p_{i+1} = p_i \text{ op step}$, as long as $p_{i+1} \leq \text{stop}$. The “reverse” iteration, where $\text{start} > \text{stop}$ and $x \text{ op step} < x$ is handled as well.
- In random mode, two tokens define an interval, in which the random samples are supposed to lie. The `samples` definition specifies the number of random samples to generate; if more than one parameter is to be generated randomly, the sample count is global, i.e. for each combination of random parameters, a new sample is generated for all of them. Integer and boolean variables are distributed normally over the interval; since most real parameters may range over various orders of magnitude, double precision parameters are distributed log-normally.
- The `samples` value also generates samples, if no parameters are chosen randomly. This is useful to ensure consistency by testing the same parameter constellation repeatedly.

Algorithmic Details of the Sweep Script

The sweep functionality of the parallel run script serves two main purposes: To generate the parameter combinations and condense the results to enable simple comparisons.

To generate all combinations of list or range parameters, a modification of the simple counting algorithm is used: Each parameter p_i is supposed to attain k_i enumerable values $\{v_i^1, \dots, v_i^{k_i}\}$, which usually are pairwise distinct; for the sake of generating the

combinations, it is irrelevant whether these values are taken from a list, or generated by iterating through a given range.

To apply the counting algorithm, we identify each value v_i^j with its index j , i.e. we define bijections $\phi_i: \{v_i^1, \dots, v_i^{k_i}\} \rightarrow \{1, \dots, k_i\}$. For each parameter, we represent its current value by a “digit” $d_i \in \{1, \dots, k_i\}$. Every combination of parameter values can then be represented by a “number” $d_n d_{n-1} \dots d_2 d_1$. To iterate through all combinations, we start with $11 \dots 11$ and count up until reaching $k_n k_{n-1} \dots k_2 k_1$; the difference to the usual mode of counting is that each digit is counted in its own base. Given a number $d_n d_{n-1} \dots d_2 d_1$, the corresponding parameter combination is $(\phi_n^{-1}(d_n), \phi_{n-1}^{-1}(d_{n-1}), \dots, \phi_2^{-1}(d_2), \phi_1^{-1}(d_1))$.

Random parameters are handled by fixing the current combination of the list and range parameters (if any) and generating the (globally) specified number of samples of all random parameters. The total number of parameter sets generated thus is equal to $\text{samples} \cdot \prod_{i=1}^n k_i$. Sampling is used, even if no random parameters are specified; this can be used to perform consistency and reliability testing of deterministic parameter sweeps.

Condensing the final results is, by comparison, a minor task, since the parallel run script already performs the necessary groundwork by collecting all results. The sweep functionality on top of this computes various sums, means and median values to facilitate comparisons across (potentially many) runs with different parameter settings.

The non-trivial user task of comparing the sweep results and picking the “best” parameter setting is facilitated by generating *plots* and a heuristic *merit value* computed for all runs. The plot feature is limited to the case where only a single parameter is under consideration, using gnuplot to produce 2D plots of the parameter value against

1. number of optimal terminations,
2. number of successful terminations (i.e. optimal or acceptable),
3. user time,

as shown in figure 4.5. A sensible extension is to generate (projections of) 3D plots when *two* parameters are considered, but this is not yet implemented.

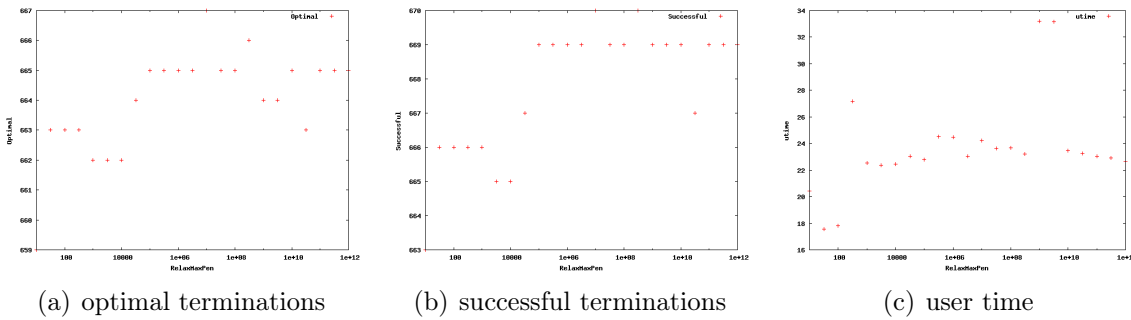


Figure 4.5.: Graphs generated by gnuplot for a single-parameter sweep

The heuristic merit value is supposed to weigh various factors and provide a scalar measure of goodness. Very similar to the merit function used by the NLP line search,

4. Solver Infrastructure

choosing a good merit function for comparing sweep results is largely heuristic, even if the ingredients are obvious. Its current incarnation (after an amount of try-and-error) is computed as

$$10n_{\text{opt}} + 5n_{\text{acc}} - 20n_{\text{fail}} + \log\left(\frac{n_{\text{iter}}^{\text{ref}}}{n_{\text{iter}}} \cdot \frac{n_{\text{F}}^{\text{ref}}}{n_{\text{F}}} \cdot \frac{n_{\text{G}}^{\text{ref}}}{n_{\text{G}}} \cdot \frac{t_{\text{user}}^{\text{ref}}}{t_{\text{user}}}\right)$$

with the usual $\max\{1, n\}$ precaution to avoid zeros, where

- $n_{\text{opt/acc/fail}}$ = number of optimal/acceptable/unsuccessful terminations,
- n_{iter} = overall number of major iterations,
- $n_{\text{F/G}}$ = overall number of objective/constraint function evaluations,
- t_{user} = overall user time,
- σ^{ref} = value of the reference run, i.e. tag [000000].

The merit value for each tag is then used to present the “best” tags to the user, a heuristic intended to enable a quick inspection. Listing 4.12 shows a (rather extreme) example of this.

```
Somehow best tags:
[000000] ( 3130.00)
[000004] ( 3031.03)
[000003] ( 554.57)
[000002] ( -2448.00)
```

Listing 4.12: Example of four sweep tags with the corresponding heuristic merit values.

To prevent ambiguities if merit values coincide, it is concatenated with the tag in a string sense and then interpreted as real number; this approach is elegantly simple (also technically, since Perl has no strict typing) and at the same time guarantees to produce pairwise distinct merit values without changing the order of already distinct values.

4.5.3. Examples

Since any piece of writing on numerical methods is lacking an essential ingredient without actual numerical results, we will consider some instances where the parallel testing framework is used to tune some exemplary parameters, and also to analyze *itself* with respect to the consistency of the obtained results.

Tuning RelaxMaxPen and surprising consistency results

As a first example, we will try and tune the `RelaxMaxPen` parameter that controls the upper bound on the penalty parameter η_r for the constraint relaxation variable δ (introduced in section 1.3.4).

We can qualitatively predict that very small values of **RelaxMaxPen** will likely cause more problems to fail and lower computational times: higher failure rate, because the QP solver is unable to force the relaxation variable to sufficiently small values, affecting convergence towards feasible points; lower computational times, because WORHP will perform only few iterative increments of the relaxation penalty before reaching its upper bound. Large values of **RelaxMaxPen** will result in higher failure rates as well, since η_r is an eigenvalue of the extended Hessian matrix, hence it affects the condition number. Intermediate values should achieve a compromise between too lax handling of the relaxation variable and too high a condition number of the extended Hessian matrix.

Figure 4.6 supports our prediction and strongly suggests choosing **RelaxMaxPen** = 10^7 as a good default value that minimizes time and maximizes the number of successfully and optimally solved problems (at least with respect to the CUTer and COPS test sets).

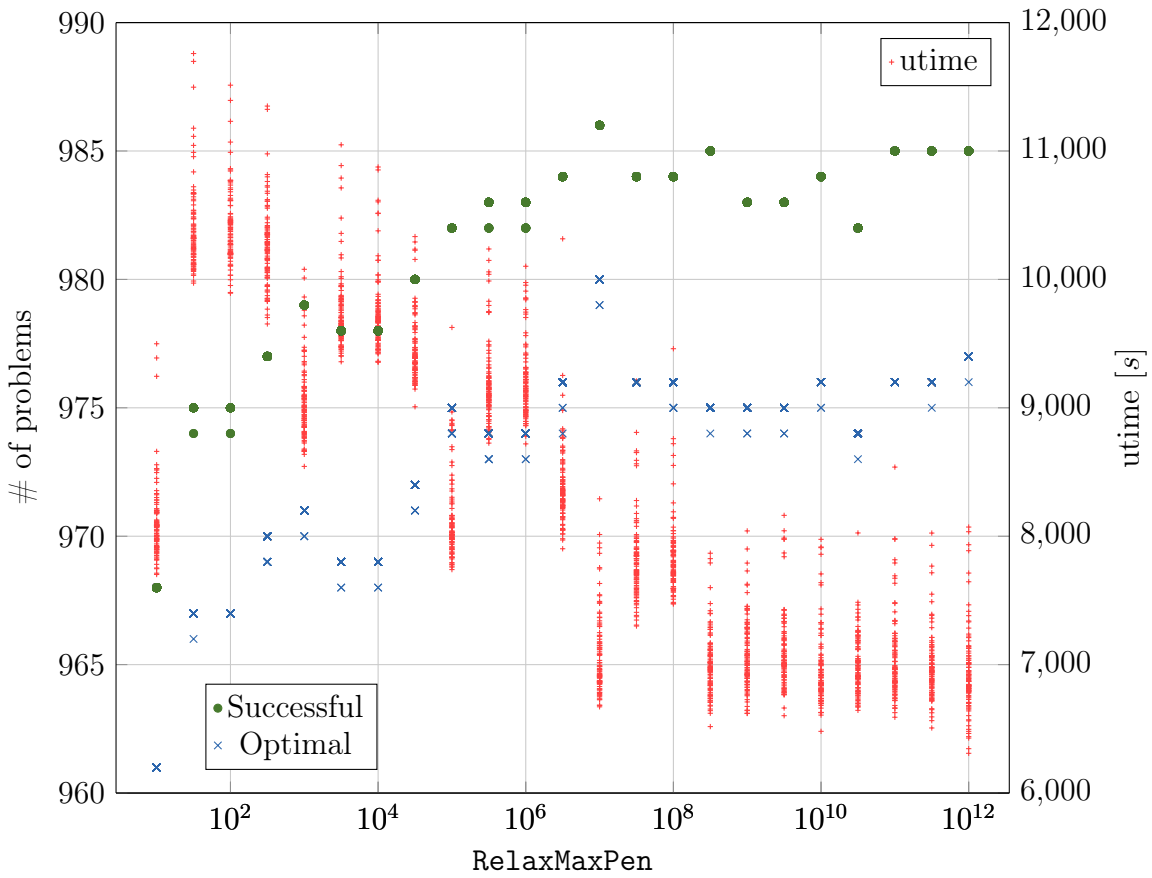


Figure 4.6.: Parameter sweep results for $\text{RelaxMaxPen} \in \left\{10^{\frac{n}{2}} \mid n = 2, \dots, 24\right\}$ on CUTer+COPS, plotting 100 samples each to test for consistency and reliability.

Besides supporting our initial prediction, figure 4.6 also has surprising implications: While a certain dispersion of the timing results is to be expected, we do not only observe a high dispersion of the continuous results, but also variations in the discrete ones, i.e. the number of successful and optimal terminations. This implies that occasional jobs fail to

4. Solver Infrastructure

run or report back to the master node. While sampling is an obvious means to increase the reliability of timing results, it is surprising that the same seems to hold for the integer results, which implies that the setup on the ZeTeM cluster has certain stability issues.

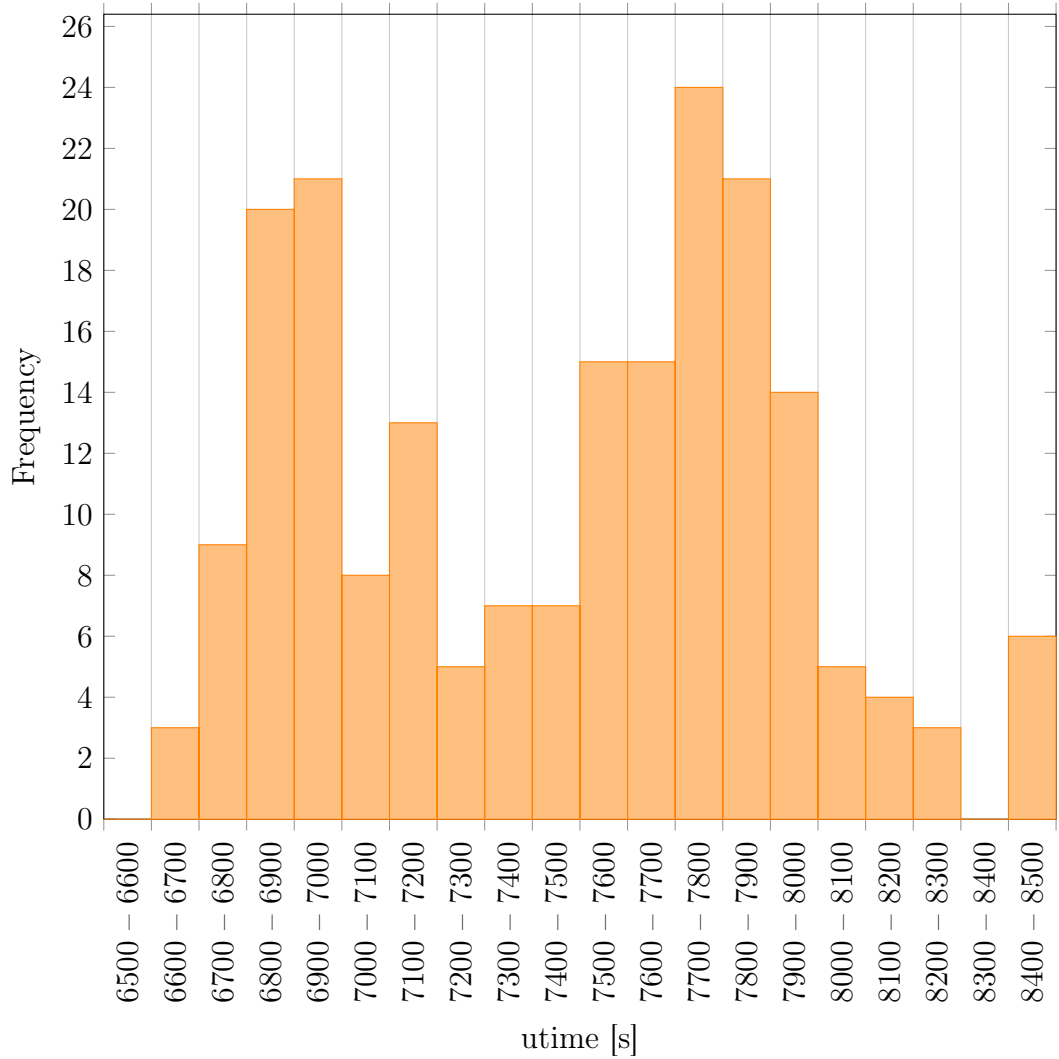


Figure 4.7.: Dispersion of utime results for $\text{RelaxMaxPen} = 10^7$ sweep, showing 200 samples.

Figure 4.7 shows the distribution of 200 timing results for $\text{RelaxMaxPen} = 10^7$; it is slightly worrying that the distribution is far from a (possibly skewed) normal distribution, and in fact not even unimodal, despite the rather large sample size. Given that the sample parameter lies well in the middle of the whole sweep range, and that the sweep script submits jobs with monotonically increasing (or decreasing) parameter values in range mode, “warm-up” or similar effects do not apply; it is safe to assume that the cluster was under constant maximum load during all jobs with $\text{RelaxMaxPen} = 10^7$.





The presence of two different types of nodes in the cluster will lead to slightly broader distributions, but does not explain the observed bimodal distribution, either. Without

more specific data on the recorded sweep run, it seems pointless to speculate about possible causes, however. We therefore close our analysis with the observation that the distribution bears some resemblance to a sum of two different normal distributions, i.e. the bimodality may be the effect of an unknown, but nevertheless deterministic, cause after all.

Consistency of timing results

Although the main goal of parameter tuning for WORHP has been robustness, i.e. solving as many problems to optimal or at least successful level, has long been the focus of parameter tuning, performance tuning is of interest as well; the goal of performance tuning is obviously to minimize the amount of required user time, which *may* be a contradicting aim to increasing robustness, but not necessarily so in all cases. Unlike measuring robustness, which is expressed in terms of integers and impervious to load level and speed of the machine running the test (assuming deterministic behavior), measuring performance is highly sensitive to these issues. It is therefore of interest, what level of consistency of timing measurements the sweep infrastructure is able to deliver. This is tested on the ZeTeM cluster (i.e. mixed SMP/MPP) and a multi-core (pure SMP) machine.

The ZeTeM cluster consists of two types of nodes: Few 16-core nodes, dubbed “SMP” and many 8-core nodes, named “MPP”. Technically speaking, all nodes can be used as SMP nodes, and the whole setup is MPP, so the node names are indicative, rather than accurate technical denominations. Besides having more cores, the SMP nodes are also equipped with more memory and have slightly higher clock frequencies ($4 \times$ Opteron 8378, 2.4 GHz vs. $2 \times$ Opteron 2378, 2.3 GHz). Using an ad hoc estimate, we would expect jobs on the SMP nodes to be $(\frac{2.4}{2.3} - 1) \approx 4.3\%$ faster than jobs on MPP nodes. Unfortunately, table 4.1 suggests that this estimate over-simplifies matters.

Type	cores/nodes	utime
SMP	32/2	 7565
MPP	16/2	 6928
MPP	32/4	 6674
SMP	16/2	 6601

◀ better

Table 4.1.: Timing consistency between SMP and MPP nodes. Indicated utime is the median of (each) 10 runs of CUTer+COPS, distributed over four different configurations of nodes and cores.

Comparing the MPP 16/2 configuration with SMP 16/2 indeed shows a 4.9% speed advantage of the SMP nodes, which is close to our prediction. The SMP 32/2 configuration,

4. Solver Infrastructure

however, is significantly *slower* than any other, and by almost 15% in comparison to the SMP 16/2 configuration.

The most probable explanation is bandwidth contention between the ($32/2 = 16$) WORHP instances on the fully loaded SMP nodes, where all other cases only run 8 instances in parallel. Since 100 Mbit Ethernet is able to deliver significantly higher throughput than the communication for job submission and results, either TCP latency, or memory bandwidth is left to blame. In other words: The observed slowdown with 16 parallel instances is most probably due to a saturation of a specific bandwidth that is *not* (yet) saturated by 8 instances.

Given that few problems from CUTer, but rather more from COPS are medium-scale, the test set as a whole definitely puts a strain on the memory bandwidth, if by chance various medium-scale problems are submitted to the same node. On the other hand, CUTer contains many problems that can be solved by WORHP in mere milliseconds, which is of similar order as typical network latencies. To conclude: While some form of bandwidth saturation is the most likely explanation, it remains unclear *which* bandwidth is being saturated, without performing more extensive measurements. Preventing the SMP nodes from being fully loaded is a feasible workaround that exhibits the expected performance differences between SMP and MPP nodes.

On a multi-core desktop (i.e. SMP) machine, running the sweep script in “consistency mode”, i.e. running samples of identical parameter sets shows very little variation:

```
Sweep: Writing parameter files in background.
Number of sweeps: 25
*** All parameter files written ***
Using (up to) 4 threads to run 17472 problems
```

Tag	Success	Optimal	Failed	Iter	F evl	G evl	utime
[000000]	661	648	11	15/8	148/11	109/5	11.34
[000001]	661	648	11	15/8	148/11	109/5	11.64
[000002]	661	648	11	15/8	148/11	109/5	11.80
[000003]	661	648	11	15/8	148/11	109/5	11.57
[000004]	661	648	11	15/8	148/11	109/5	11.73
[000005]	661	648	11	15/8	148/11	109/5	11.60
[000006]	661	648	11	15/8	148/11	109/5	11.63
[000007]	661	648	11	15/8	148/11	109/5	11.48
[000008]	661	648	11	15/8	148/11	109/5	11.46
[000009]	661	648	11	15/8	148/11	109/5	11.72
[000010]	661	648	11	15/8	148/11	109/5	11.84
[000011]	661	648	11	15/8	148/11	109/5	11.68
[000012]	661	648	11	15/8	148/11	109/5	11.80
[000013]	661	648	11	15/8	148/11	109/5	11.36
[000014]	661	648	11	15/8	148/11	109/5	11.86
[000015]	661	648	11	15/8	148/11	109/5	11.12
[000016]	661	648	11	15/8	148/11	109/5	12.06
[000017]	661	648	11	15/8	148/11	109/5	11.33
[000018]	661	648	11	15/8	148/11	109/5	11.81
[000019]	661	648	11	15/8	148/11	109/5	11.36
[000020]	661	648	11	15/8	148/11	109/5	11.70

[000021]	661	648	11	15/8	148/11	109/5	11.42
[000022]	661	648	11	15/8	148/11	109/5	12.04
[000023]	661	648	11	15/8	148/11	109/5	11.55
*** All jobs sent to workers ***							
[000024]	661	648	11	15/8	148/11	109/5	11.87
[000025]	661	648	11	15/8	148/11	109/5	10.99

The distribution of the `utime` values is reasonably close to a normal distribution (72% probability), and their standard deviation $\sigma \doteq 0.26$ is almost two orders of magnitude smaller than the values. The most important observation is that the last `utime` value is a clear outlier. This is probably due to the lower machine load in the last sweep run, which puts a smaller strain on caches and memory bandwidth.

Tuning `ScaleFacObj` and sweep automation

Let us now consider the `ScaleFacObj` parameter, which defines a scaling target with respect to the objective. If this scaling mode is enabled (by the `scaledObj` flag), the objective is scaled by the factor

$$c^+ := \min \left\{ 1, \frac{\text{ScaleFacObj}}{|\frac{1}{c}f(x)|} \right\},$$

which is updated in every major iteration. The motivation behind this scaling factor is to leave reasonably well-scaled objectives alone, while reining in the badly-scaled ones. We consider a fine-grained parameter sweep for $10^{-5} \leq \text{ScaleFacObj} \leq 10^5$.

Since higher values cause WORHP to apply scaling to fewer of the test problems, we can qualitatively expect lower values to produce better results, if badly scaled objectives are responsible for unsuccessful terminations with the default settings.

Considering the sweep results figure 4.8, we can make various observations:

1. The results are subject to pronounced dispersion. One possible interpretation is that among the roughly 1000 test problems, few have problem conditions that tax the numerical methods in WORHP to their limits, hence minor scaling variations may be the decisive element between optimal, acceptable or unsuccessful termination of the solver.

This is a general observation in parameter sweeps: Some test problems are particularly sensitive to certain solver parameters and may show quasi-stochastic behavior in the sweep results. The set of these test problems is parameter-dependent, although some problems (such as some instances of the `palmer`, the `vanderm` and the `himmel` series) are particularly conspicuous. In our case, the dispersion width suggests that 3 or 4 problems have this quasi-stochastic sensitivity to `ScaleFacObj`.

These “quasi-stochastic” test problems have proven to be problematic for the parameter sweep, if computational resources are limited, and the overall sample rate is low, since they may suggest highly localized maxima (of successful/optimal terminations) or minima

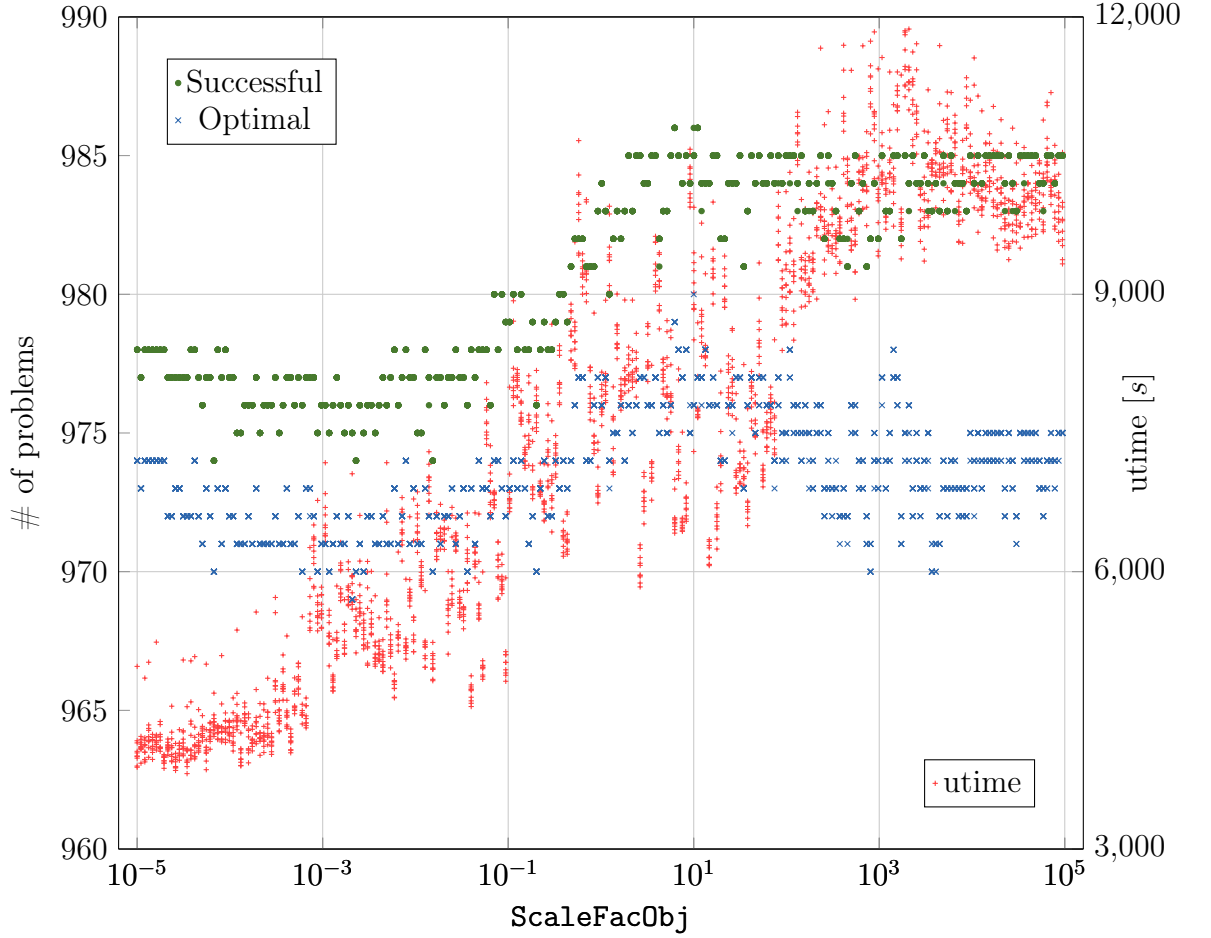


Figure 4.8.: Parameter sweep results for $\text{ScaleFacObj} \in \{10^{-5} \cdot 1.1^n \mid n = 0, \dots, 2415\}$ on CUTer+COPS, using 10 samples each.

(of utime), even in unfavorable regions of the parameter space. The high sampling rate and wide intervals that are made possible by distributing the parameter sweep over a powerful computational cluster enable us to move from localized *extrema* to robust *regions* of the parameter space.

To continue our observations:

2. The computational times are subject to high variance as **ScaleFacObj** grows, but seem to stabilize again between 10^3 and 10^5 . This effect is explained by the action of **ScaleFacObj**: The greater the trial parameter, the fewer problems are modified, hence the behavior stabilizes; the smaller the parameter, the *more* problems are modified, but gradually transformed into feasibility problems, if very small parameters are chosen: Although the two problems

$$\min f(x) \text{ s.t. } g(x) = 0$$

and

$$\min 10^{-20} f(x) \text{ s.t. } g(x) = 0$$

are equivalent from a theoretical point of view, the second one degenerates into a feasibility problem $g(x) = 0$ when solved numerically. It is therefore the “middle” region, where many test problems are affected by the parameter—and hence show variance—, but whose objective still has numerical influence.

3. We can observe two local maxima with respect to optimal & successful terminations, and one (boundary-)minimum of the utime values. Had we added additional “intelligence”, in form of support vector machines or global optimization techniques, to the sweep script to automatically identify optimal parameters, it might return 10^{-5} as optimal choice. However, as explained earlier, this choice would essentially transform optimization problems into feasibility problems, which is clearly against the user’s intentions.

Gathering and incorporating all observations, the current default of `RelaxMaxPen` = 10^1 is a suitable choice (at least with respect to the test set), since it hits the global maximum of optimal & successful terminations, and does not interfere with thoroughly well-scaled problems. The timing results for this region are hard to interpret through the high dispersion, but on average lie well below the maximum times reached around `RelaxMaxPen` = 10^3 .

4.5.4. Conclusions

The parameter sweep was introduced to enable parameter tuning, both generic and problem-specific. It is intended to somewhat ease the burden of requiring deep knowledge of both the optimizer and the application to be able to tune solver parameters. While it has yet to be applied by users, it is not intended as developer-exclusive tool; running the sweep script is sufficiently simple to allow wider use in any case, but the interpretation of the results is not necessarily as simple as the graphs suggest. If users are applying the parameter sweep to self-chosen test sets, we can assume, however, that they are able to interpret the results and would therefore not “fall for traps” such as the potentially misleading result of the `RelaxMaxPen` sweep discussed earlier.

This anticipates one major conclusion: While the parameter sweep is simple to run, blindly trusting its results can be dangerous, which is the reason why the sweep script has deliberately not been equipped with additional intelligence to automate the parameter choice: The sweep technique still requires a reflecting user to scrutinize its results.

The second conclusion concerns consistency and reliability: As the multi-core test indicates, running the sweep script on a (preferably unloaded) modern desktop machine yields reliable results that are very consistent across repeated runs; the only observed variation concerns the time measurements. To achieve higher accuracy here, a small (2–5) number of samples can be used in addition to the (deterministic) parameter variations to calculate a median of the timing results as highly reliable measure.

As the examples suggest, conditions are slightly worse on the ZeTeM cluster, caused by unreliable nodes with randomly dying slaves and currently unexplained issues that result

in high dispersion of timing results. Since even the integer measurements show variation, sweep runs on the cluster always have to use sampling to achieve reliable results. If robustness is the major concern, a rather small number of samples can already guard against false results: If we assume a probability of 10^{-4} (1 in 1000) for a test problem to fail because of technical problems (in reality, it is probably still lower), a single run of CUTer+COPS has a $1 - (1 - 10^{-4})^{988} \approx 9.4\%$ probability of returning a false result due to one or more technical failures; this is uncomfortably high. However, using 2 or 3 samples, we can lower the probability of *all* test runs returning a false result to 0.89% and 0.083%, respectively, which seems acceptable.

Achieving comparably reliable timing results on the cluster seems to require many more samples, considering the dispersion observed in figure 4.7, which is not even normally distributed (probability $< 4.2 \cdot 10^{-7}$). It still seems sensible to use the median value of the results of multiple samples, instead of the minimum, since deviations are present in both directions, for instance if a slave with a long-running problem dies immediately after starting it.

All in all, the parallel parameter sweep is a powerful tool, although it requires some care in interpreting its results. The reliability of its results is excellent on multi-core machines, and can be increased to satisfactory levels on the cluster by the use of sampling. It is unique with its capability of “*optimizing the optimizer*”, which has likely never been considered for NLP solvers as systematically as described here.

4.6. Future directions for testing

When viewed from a software engineering angle, the current testing regime of WORHP is very narrowly focused on performance and less on verification and validation in the software-technical sense. The current form of testing could be regarded as a combination of *system testing*, which usually is the last or last-but-one step in formalized software testing, and *performance testing*. With continued development and maintenance of WORHP, it will be sensible to add more software-engineering oriented forms of testing: *unit tests*⁸ perform freestanding tests on single routines or modules, such as WORHP’s CS module that provides routines for handling sparse matrices. This form of testing is easier to perform than system testing, since each module has only few, well-defined functions of lower complexity than the overall system, and thus requires fewer or simpler tests to exhaustively test it for defects.

Continuous Integration (CI) is a quality assurance technique that merits partial inclusion into the development process. Given the existing run script for providing test results and the configuration and build system, an initial setup requires only a continuous integration server that monitors the code repository (Subversion[57] in WORHP’s case) and triggers test and build runs under certain conditions – usually at every commit, when following the CI rules to the letter. Examples of such CI servers are Jenkins[44], written in Java,

⁸Engineers might call this “module-in-the-loop”

but easily applicable to other “eco-systems” by plugins, and bitten[65], based on trac[66], both written in Python.

Besides extending the testing approach to new technical forms of testing, it is highly necessary to extend the test sets by new problems, problem classes and using different interfaces—the CUTer and COPS tests are currently exclusively run through the AMPL interface. This may obscure existing weaknesses of other interfaces or WORHP’s numerical and algorithmic core, since AMPL provides analytic derivatives, whose precision depends only on the machine precision and the math library, and internally simplifies the problem, removing variables or constraints. Formerly existing problems in handling fixed variables were only discovered through an application based on the MATLAB interface, since AMPL removes these.

Extending the test set with new test problems, and most importantly, beyond AMPL is therefore critical for WORHP’s intended use as a general-purpose, robust NLP solver.

Appendix A

Examples

A.1. Bypassing const-ness

These two examples use C to demonstrate bypassing a `const` qualifier on a routine argument by changing it through a global pointer.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  /* Used for global access to n in main */
5  static int *np;
6  void routine1(const int n);
7  void routine2(const int n);
8
9  int main (void) {
10     int n = 42; np = &n;
11     routine1(n);
12     exit(EXIT_SUCCESS);
13 }
14 void routine1(const int n) {
15     routine2(n);
16     printf("After routine2: n = %d\n", n);
17 }
18 void routine2(const int n) {
19     (*np)++; /* Increment n */
20     printf("After increment: n = %d\n", n);
21 }
```

Listing A.1: Example for bypassing const-ness and call-by-value.

In example listing A.1, `routine2` needs to use a global pointer to (in a way) change its argument `n`, since it inherited its const-ness from `routine1` and we are unwilling to change the whole calling hierarchy.

A. Examples

If we run this code, we find that `n` keeps its value of 42 in both routines, although its “master copy” in `main` *has* been incremented. While this is not at all astonishing to seasoned C programmers, `n` being passed by value, i.e. every routine having a local copy, it holds a potential element of surprise to the perfunctory inspector and thus violates the good programming practice that *code should do what you expect it to do*.

The fact that `n` does *not* change in both routines is in keeping with its const-ness, but contradicts the initial assumption that `n` *should* change, since it has been incremented after all.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  /* Used for global access to n in main */
5  static int *np;
6  void routine1(const int *n);
7  void routine2(const int *n);
8
9  int main (void) {
10     int n = 42; np = &n;
11     routine1(&n);
12     exit(EXIT_SUCCESS);
13 }
14 void routine1(const int *n) {
15     routine2(n);
16     printf("After  routine2: n = %d\n", *n);
17 }
18 void routine2(const int *n) {
19     (*np)++; /* Increment n */
20     printf("After increment: n = %d\n", *n);
21 }
```

Listing A.2: Example for bypassing const-ness and call-by-reference.

Listing A.2 differs from the previous example only in that `n` is not passed by value, but by reference. If we compile and run this version, we find that it prints 43 twice. Again, no C programmer should be caught by surprise, but the same statement as for listing A.1 holds, only under opposite signs: the assumption that `n` should change holds true this time, but the const-ness claimed by the routine signature is (somehow) violated, which is surprising again.

Examples A.1 and A.2 are tiny. If we mentally scale them up to a complete software package with thousands of LOC distributed over various files, we may well arrive at the point where changing *anything* breaks the code in non-obvious ways. When a software project arrives at this point, it may well be beyond salvation.

Bottom line: Use of global data, combined with Aliasing, easily causes unexpected effects, creating severe maintenance liabilities, and should thus be avoided.

List of terms

Aliasing describes the situation where a single piece of data (in memory) can be accessed through two or more pointers. The fact that Fortran forbids aliasing by default accounts for its high performance, since it allows compilers to make aggressive use of caching and greatly simplifies dependency analysis. In contrast, C allows aliasing by default, hence C compilers have to work more conservatively to guarantee data consistency. 133

AMPL is **A** Modeling Language for **M**athematical **P**rogramming[22]. It allows to formulate optimization problems in an intuitive language closely resembling mathematical notation, and uses symbolic operations to simplify the problem and provide exact (to machine precision) derivatives. 95, 96, 104, 107, 110, 137, 143

Application Binary Interface defines the interface between programs or libraries on the assembler level, i.e. byte order (see endianness), stack convention, etc.. Software compatibility on the ABI level is often desirable to allow for simple upgrade paths, but challenging, since there are significant differences between different compilers (or versions of them) and platforms. 145

Application Programming Interface in our sense is the definition of types and functions used to communicate with a given piece of software, often a program library. Software development differentiates between *stable* and *evolving* APIs, where the first remains static or at least downwards-compatible over various versions, whereas the latter may change in incompatible ways between different versions of the software. The GNOME libxml2[73] is an example of a stable API, while the Linux kernel's API is deliberately left unstable. 139, 145

ASL supposedly an acronym for “AMPL solver library” provides the back-end library for hooking NLP (or other) solvers to AMPL. It is available from <http://www.netlib.org/ampl/solvers>. 104

AutoGen “is a tool designed to simplify the creation and maintenance of programs that contain large amounts of repetitious text. It is especially valuable in programs that have several blocks of text that must be kept synchronized.” [47]. AutoGen is based on Scheme, using the guile[5] library. It is used by WORHP to generate a

large portion of the data structure-related code. 69, 95

barrier is a concept used in multi-threaded software. A barrier is a code point that has to be reached by a set of threads before the program continues; it causes faster threads to wait for the slower threads to “catch up”. Barriers are typically used when results from multiple threads are condensed. 113

bash the **B**ourne-**a**gain **s**hell is the standard shell on many current GNU/Linux systems. It is mostly compatible with the older *Bourne Shell* **sh** introduced with Unix V7, but also adopted some features of the *Korn-Shell* **ksh** and the *C-Shell* **csh**. 83, 110, 111

BLAS the **B**asic **L**inear **A**lgebra **S**ystem defines a set of routines that perform basic operations on vectors and matrices for higher-level code to rely on, so that each architecture can provide (in the engineering sense) optimized versions. Routines are grouped into *levels* 1–3 according to their operands (scalar-vector, matrix-vector and matrix-matrix). 39, 95

CamelCase is a convention for concatenating words by capitalizing the first letter of every interior word, as in *CamelCase* or *computeThisNumber*. Using underscores to connect words (as in *compute_this_number*) can be thought of as the complementary convention. 47

Continuous Integration is a form of software quality insurance developed that focuses on tightly coupling small, incremental changes with highly automated testing and building infrastructure. Its central aim is to expose conflicts and errors introduced by changes early though continued automated building and testing of the software under development. 129, 145

COPS the **C**onstrained **O**ptimization **P**roblem **S**et is a collection of large-scale, whose “primary purpose [...] is to provide difficult test cases for optimization software. Problems in the current version of the collection come from fluid dynamics, population dynamics, optimal design, mesh smoothing, and optimal control.”. COPS 3.0 is documented in [16, 17]. 107

CUTEr an acronym for “CUTE revisited” is a collection of over 1,100 optimization problems in SIF format that serves the purpose of providing a standard set of problems to test and compare optimization codes. WORHP is benchmarked with a 920-problem set that contains roughly 750 problems of the complete collection that is documented at [56]. 95, 107, 111

Debugging derived from the term “bug”, which is nowadays used to describe any kind of defect in software. Its origin lies in Harvard, where on September 9, 1947, engineers working on the Mark II computer found a moth stuck in relay #70, causing an arithmetic test to fail. Grace Murray Hopper is reported to have coined the term “debugging” on this occasion. 107

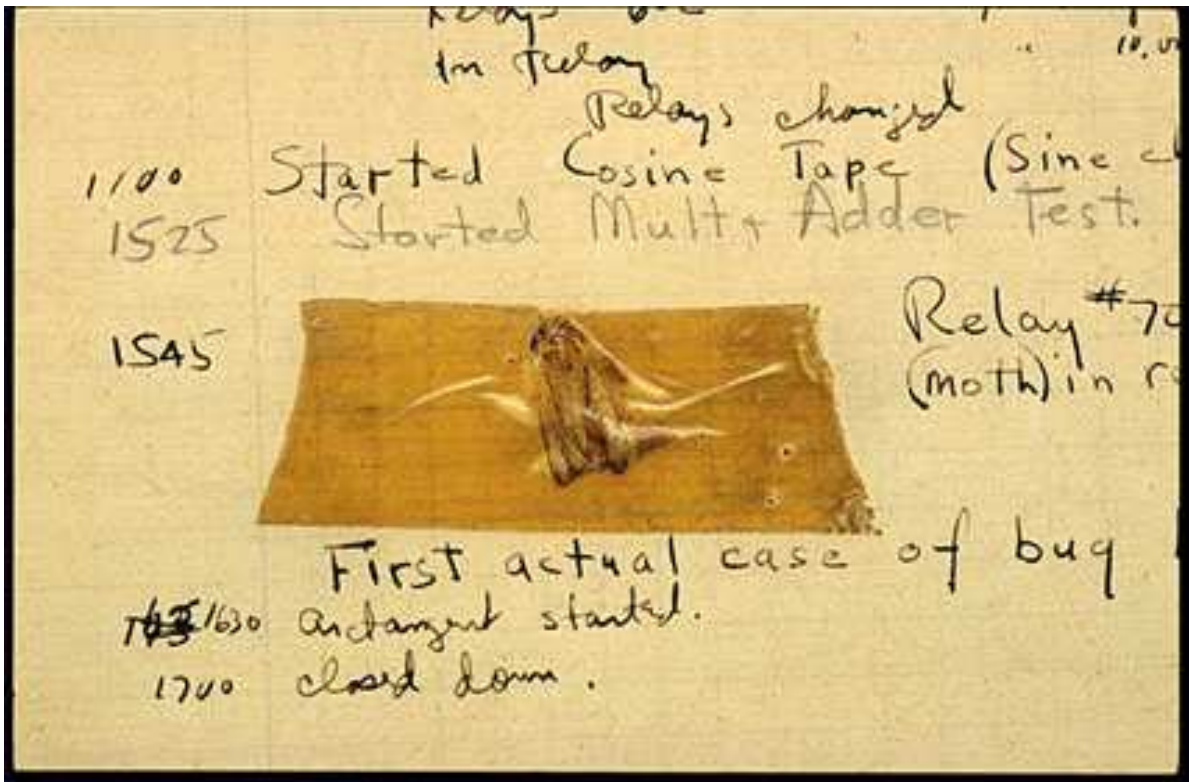


Figure A.1.: The world's first computer bug, glued into the Mark II logbook. The comment reads "First actual case of bug being found".

deque is a contraction of the slightly unwieldy word *dequeue*, which in turn is an abbreviation for double-ended queue, i.e. a list of objects that allows elements to be added and removed from *both* "ends". 112, 114

DOM the **D**ocument **O**bject **M**odel is a W3C standard that defines a tree-like representation of an XML document and an Application Programming Interface (API) for accessing and modifying it. 85, 86

Doxygen is a documentation system[71] for C, C++, Fortran and other languages that parses the source code for special comments, very similar to Javadoc[39], but tailored to the respective languages' syntax for comments. 95

DTD short for **D**ocument **T**ype **D**efinition is a format to define a document type for SGML (and thus XML and HTML) documents by writing grammar-like rules. A validating parser can use a DTD to validate that a document conforms to the rules of a thus-defined document type. 85

Dynamic Link Library Shared library format on Windows platforms. To link libraries or executables to a Dynamic Link Library, some compilers need an Import Library. In most cases .dll is used as file suffix. 95, 140, 145

ELF the **E**xecutable and **L**inkable **F**ormat is used as standard binary format for object

code, libraries and executables on many Linux and Unix platforms, replacing the older `a.out` and `COFF` formats. An ELF object can be examined with several tools, among them `readelf` and `objdump`. 62, 104, 105, 143

endianness is a property of a processor architecture that describes the bit storage order of integers. *Big-endian* architectures save the MSB first (at the lowest bit of a word) and the LSB last (at the highest bit of a word), whereas *little-endian* has the exact opposite storage order. Arabic numbers, for instance, are written down in big-endian order, whereas the German ordering of dates as DD.MM.YYYY can be thought of as little-endian. The Intel x86(-64) architecture is little-endian, while SPARC and PowerPC are historically big-endian (actually bi-endian nowadays, supporting both modes). 57, 137

HSL (formerly the Harwell Subroutine Library) is a collection of threadsafe ISO Fortran codes for large scale scientific computation [34]. 95

Import Library Specialized static library on Windows platforms needed by the linker to link against a Dynamic Link Library. Native Windows tools use the `.lib` suffix, but the MinGW-related tools also use `.dll.a`. 139

Ipopt “is an open-source solver for large-scale nonlinear continuous optimization. It can be used from modeling environments, such as AMPL, GAMS, or Matlab, and it is also available as callable library with interfaces to C++, C, and Fortran. Ipopt uses an interior point method, together with a filter linear search procedure”[74]. The solver Ipopt is presented in [75]. 38

JSON the **J**ava**S**cript **O**bject **N**otation[13] is a standard for serializing JavaScript objects, i.e. it defines a mapping between objects and a string representation of them. JSON supports numeric, logical and string data types and used C-like `{ }` notation for denoting hierarchies. 50

LAPACK the **L**inear **A**lgebra **P**ackage provides an extensive set of higher-level routines for “solving systems of simultaneous linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, and singular value problems”. 95

LISP is a functional language based on λ -calculus. LISP is an acronym for **L**ist **P**rocessing, but the humorous *Lots of Irritating Superfluous Parentheses* is regarded a more appropriate description by some. (X)emacs is an example of a well-known application written in a LISP dialect called Emacs Lisp. 143

locale defines the country-/language-specific conventions for date and time format, number representation (in particular decimal point), currency symbol, time zone and others. 87

magic number is a derogatory term for the use of literal numbers in code whose justification is not obvious. Examples are conditionals like `if (iFlag == 31415)`

(where 31415 should be replaced by a descriptive constant), dimensions like `char string[13]` or even arithmetic like `i = 60*j`; since `i` and `j` are completely non-descriptive, does 60 describe the number of seconds in a minute, of years between three successive Great Conjunctions of Jupiter and Saturn or of carbon atoms in a buckyball? One main criticism of magic numbers is their burden on maintenance, since refactoring named constants is trivial against finding and replacing (the correct!) number literals in extensive code. 47

Massively Parallel Processing abbreviated as **MPP** describes parallel computing on many processors that do not have shared, but *distributed* memory, i.e. the processors are coupled loosely by network or similar connections. Parallel programming that uses explicit message passing, such as MPI, is required on MPP architectures. 143, 145

MATLAB is an extensive development environment for numerical algorithms[49]. It operates numerically on matrices and vectors and is very suitable for prototyping and quick visualization, but lacks the performance of compiled languages. 96

METIS is a set of routines for partitioning graphs, partitioning finite element meshes, and producing fill-reducing orderings for sparse matrices[43, 50]. 95

MinGW Minimalist GNU for Windows[52] is a project that ports part of the GNU toolchain to Windows, which includes (a subset of) the Windows API, compilers from the GCC and binutils. It allows the creation of native Windows binaries, but does explicitly *not* emulate a POSIX-like environment, like cygwin does. 140

name mangling is the mapping of a function name in the source code to its assembler name in the generated object code. For C, the mapping is usually direct, while C++ employs complex mappings to handle polymorphism on the linker level (example: `long foo(std::vector<double> &v, size_t n)` is mangled to `_Z3fooRSt6vectorIdSaIdEEm` by `g++`). Fortran compilers also mangle function and subroutine names in a vendor-specific way. 62

Nonlinear Programming The Name of the Game: Nonlinear Programming is the process of minimizing a smooth (nonlinear) objective function subject to smooth (nonlinear) equality and inequality constraints, usually by iterative numerical methods like WORHP. 145

Object-oriented Programming is a programming paradigm popularized with the advent of C++, although its inception dates back many years before that. Its main distinction from other programming paradigms, such as *functional* or *imperative*, is the focus on data encapsulated in *objects* with operations defined on them. Important concepts are inheritance, polymorphism and encapsulation. 38, 145

Perl is an extensive scripting language, initially designed as “glue” language with emphasis on text processing, but nowadays extended by various modules to a general-purpose language suitable for many applications. One of its distinctive features is its rather

large body of keywords that purposefully includes redundancies (`unless(cond)` is a graphic example of this, intended as more intuitive alternative to `if(not(cond))` that needs to be used in other languages). Perl is often characterized by the “there is more than one way to do it” (TIMTOWTDI) paradigm, which facilitates writing Perl programs but may result in code that is hard to understand. Perl also follows the so-called “do what I mean”-approach for interpreting language constructs in non-surprising ways. 69, 110, 142

PVFS originally named **P**arallel **V**irtual **F**ile **S**ystem[61] is a file system for distributed computing focused on delivering high performance and scalability to petabyte dimensions. 111

Python is a universal, multi-paradigm programming language with a strong focus on clarity, readability and expressiveness, backed by a very extensive standard library. Distinct features of Python are significant whitespace (blocks are defined by indentation) and the small size of its set of builtin functionality and keywords. The language philosophy is somewhat opposite to Perl in providing a minimalistic, redundancy-free, easily extensible language that allows (and encourages) a single, clear, “canonical” implementation of any given problem. Python is therefore considered as very suitable for teaching, but in contrast to other didactic programming languages, is also actually being used for serious software projects. 69

R is a free, open-source system for statistical computation and visualization, documented in [58]. 83

Ragel is a state machine compiler that targets languages like C, C++, Java and Ruby. It uses a regular language to define a grammar, and to embed actions to perform during the scanning/parsing process. Ragel then creates free-standing code that defines a finite-state machine to scan/parse the defined grammar. 85

Regression Testing is a test method applicable to software producing output that can somehow be verified or checked against a reference, with the aim to identify and correct modifications to the software that cause it to fail on inputs, which were handled correctly before. A regression test is performed by (usually automatically) iterating over a set of inputs and verifying the outputs, e.g. by comparing against some reference output. Regression testing is most powerful, if for every identified defect an input to trigger it is added to the test set – this approach can very effectively prevent the re-surfacing of bugs that have been fixed before. The GCC is a noticeable example of a software package that relies heavily on regression testing for developing and accepting bugfixes: the test suite of GCC 4.6 consists of more than 40,000 individual pieces of code. 107

Reverse Communication is a software architecture convention that places tasks usually performed in the bowels of a software package (the callee) back into the callers hands, e.g. the evaluation of a function: instead of implementing a function and providing a pointer to it to the callee, who then calls the function at his own disposal, the caller, when prompted by the callee, is given control, performs the

function's action and hands the control back to the callee. This reversal of the usual interaction between caller and callee has coined the term. 32, 41, 43, 66, 145

Scheme is a LISP dialect with a minimalistic design philosophy[67], aimed at allowing enough flexibility and providing extension tools to extend the language to the users needs, rather than defining an exhaustive set of language features. 69, 95, 137

sed short for **stream editor** is a stream-oriented text processor on unixoid platforms (although ports to other platforms, such as Windows exist). It uses a very concise syntax for searching, replacing, stack manipulations and such, which actually make it Turing-complete; accomplished sed programmers have implemented a calculator, a Linux to Unix assembler converter, or Sokoban(!) using sed. 69, 104

SIF the Standard Input Format is a text format for specifying linear, quadratic and nonlinear optimization problems, closely linked with the NLP solver LANCELOT[12] that is based on SIF as input format, and the MPS format for linear problems, of which it is a superset. A **sifdecode** tool is available to generate FORTRAN code from SIF files, which can be used to solve SIF-encoded problems with other solvers. In contrast to AMPL or similar formats, SIF is highly non-intuitive, requiring the problem to be formulated as a high-level structural representation. The format is described in [7]. 107

SuperLU is a general purpose library for the direct solution of large, sparse, nonsymmetric systems of linear equations on high performance machines[15, 48]. 95, 96

symbol versioning is an extension to ELF that allows to annotate *symbols* (i.e. functions) in a shared library with a version, and to provide different versions of one symbol to a caller. It is used to retain backwards-compatibility even if the functionality of existing functions changes. The GNU libc makes ample use of symbol versioning, in order to support existing binaries without recompilation after libc updates. 51

Symmetric Multi-Processing abbreviated as **SMP** describes parallel computing on systems where all processors have access to a single (large) amount of shared memory, hence communication between processors only requires locks or semaphores, but data does not have to be passed around in messages, as required by Massively Parallel Processing. OpenMP is a typical choice for programming on SMP machines. 145

sys time is the subset of a computer's CPU time that is used by the kernel on behalf of a process, and as such is complementary to user time. The sys time should usually be much lower than the user time for a numerical application, unless it performs intense I/O operations. The total CPU time used by and for a given process is the sum of user and sys time. 114

Unified Solver Interface is a marketing-compatible term referring to the uniform (**opt**, **wsp**, **par**, **cnt**) interface of many WORHP routines. 32, 38, 40, 146

- unit of least precision** is the absolute spacing between two consecutive floating point numbers, abbreviated as *ulp*. It is related to the *machine epsilon*, which describes the same concept normalized to the floating point number 1.0. The IEEE standard dictates that the result of elementary arithmetic operations with floating point numbers be within 0.5 ulp of the exact result. 146
- user time** is the subset of a computer's CPU time that the kernel allocates to a process. It excludes time used by the kernel on behalf of this process, e.g. for file operations, and the time spent sleeping, when the kernel allocated CPU time to other processes. For single-thread processes, user time is always less than wall time, since it shares the CPU with the kernel and other processes. 107, 114, 143, 144
- valgrind** is an extensive and invaluable debugging and profiling tool for Linux. It uses dynamic recompilation of *binary* program code into an internal representation to enable a very wide range of possible diagnostics, including memory debugging, heap and cache profiling, diagnosing race conditions between threads and producing call graphs[53, 6, 54, 55]. 70, 84
- wall time** is the physical time used by a process, such as a stopwatch operated by the user (or the clock on the *wall*) would measure it. While it gives an indication of the perceived speed of a process, it is misleading on machines under load, where many active processes compete for CPU time and where the user time is a more appropriate way of measuring and comparing (performance) timings on computers. For single-thread processes, wall time is a strict upper bound for the user time, but multi-threaded processes usually reverse this order, since user time (at least on Linux systems) is cumulative for multiple CPUs. 114, 144
- Workspace Management Table** 2D integer table used by WORHP to track start/end-indices and sizes of Workspace Slices. 79, 146
- Workspace Slice** Array slice of one of WORHP's large workspace arrays.. 144
- XML** the eXtensible Markup Language is a Unicode-based markup language for creating machine-readable documents[8]. An XML document has a tree-structure and is organized by annotating its content with tags and attribute-value pairs. 50, 96, 139
- YAML** Ain't Markup Language[19] is a recursive acronym for a plain-text serialization language. It supports key-value pairs, lists, associate arrays and hierarchies, which it denotes by indention, and distinguished between integer, float, boolean and string data. 50
- ZeTeM cluster** is a cluster of 36 computing nodes (4 SMP nodes with 16 cores + 32 MPP nodes with 8 cores) with a total of 320 Opteron cores and a theoretical peak performance in the order of 100 GFlops. TORQUE[36] is used as resource manager and MOAB[35] as job scheduler. 111, 112, 123, 124

List of acronyms

- ABI** Application Binary Interface. 89
- API** Application Programming Interface. 41, 89, 139
- CI** Continuous Integration. 129
- DAG** directed acyclic graph. 94
- DLL** Dynamic Link Library. 95, 96
- FSM** finite-state machine. 85
- GCC** GNU Compiler Collection. 95, 106, 107, 141, 142
- GNU** GNU's not Unix. 69, 92, 94, 98, 99, 104, 141
- IWMT** Integer Workspace Management Table. 79
- LOC** Lines of code. 133
- LSB** least significant bit. 57, 140
- MPP** Massively Parallel Processing. 124
- MSB** most significant bit. 57, 140
- NLP** Nonlinear Programming. 38
- OOP** Object-oriented Programming. 38, 42
- RC** Reverse Communication. 32, 46, 66
- RWMT** Real Workspace Management Table. 79
- SMP** Symmetric Multi-Processing. 124

List of acronyms

SQP Sequential Quadratic Programming. 1

ulp unit of least precision. 50

USI Unified Solver Interface. 32, 38, 40, 44

WMT Workspace Management Table. 79

Bibliography

- [1] David Abrahams and Vladimir Prus, *Boost.Build*, <http://www.boost.org/doc/tools/build/>.
- [2] *GNU Automake*, <http://www.gnu.org/software/automake/>.
- [3] Ned Batchelder, *cog*, <http://nedbatchelder.com/code/cog/>.
- [4] Bryan Blackburn, Joshua Root, Rainer Müller, Ryan Schmidt, et al., *MacPorts*, <http://www.macports.org/>.
- [5] Jim Blandy, Greg Harvey, et al., *Guile, the GNU Ubiquitous Intelligent Language for Extensions*, <http://www.gnu.org/software/guile/>.
- [6] Michael D. Bond, Nicholas Nethercote, Stephen W. Kent, Samuel Z. Guyer, and Kathryn S. McKinley, *Tracking bad apples: Reporting the origin of null and undefined value errors*, Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2007) (Montreal, Canada), October 2007, pp. 405–422.
- [7] I. Bongartz, A.R. Conn, N.I.M. Gould, and Ph.L. Toint, *CUTE: Constrained and Unconstrained Testing Environment*, ACM Transactions on Mathematical Software **21** (1995), no. 1, 123–160.
- [8] Tim Bray, Jean Paoli and C. M. Sperberg-McQueen, Eve Maler, and François Yergeau, *Extensible Markup Language (XML) 1.0*, <http://www.w3.org/TR/REC-xml/>.
- [9] C. G. Broyden, *The convergence of a class of double-rank minimization algorithms. II. The new algorithm*, J. Inst. Math. Appl. **6** (1970), 222–231.
- [10] Christof Büskens, *Optimization Methods and Sensitivity Analysis for Optimal Control Problems with Control and State Constraints*, Dissertation, Universität Münster, Münster, 1998.
- [11] ———, *Real-Time Optimization and Real-Time Optimal Control of Parameter-Perturbed Problems*, Professorial dissertation, Universität Münster, Münster, 2002.
- [12] A.R. Conn, N.I.M. Gould, and Ph.L. Toint, *LANCELOT, A Fortran Package for Large-Scale Nonlinear Optimization (Release A)*, Springer Series in Computational Mathematics, no. 17, 1992.

- [13] D. Crockford, *JavaScript Object Notation (JSON)*, <http://www.json.org/>.
- [14] E. Cuthill and J. McKee, *Reducing the bandwidth of sparse symmetric matrices*, ACM '69 Proceedings of the 1969 24th national conference (New York, NY, USA), 1969, pp. 157–172.
- [15] James Demmel, Stanley C. Eisenstat, John Gilbert, Xiaoye S. Li, and Joseph W. H. Liu, *A supernodal approach to sparse partial pivoting*, SIAM J. Matrix Analysis and Applications **20** (1999), no. 3, 720–755.
- [16] E. Dolan and J. Moré, *Benchmarking Optimization Software with Performance Profiles*, Mathematical Programming **91** (2002), 201–213.
- [17] E. Dolan, J. Moré, and T. Munson, *Optimality measures for performance profiles*, SIAM Journal on Optimization **16** (2006), 891–909.
- [18] Paul Eggert and Eric Blake, *GNU Autoconf*, <http://www.gnu.org/software/autoconf/>.
- [19] Clark C. Evans, *YAML Ain't Markup Language (YAML)*, <http://www.yaml.org/>.
- [20] R. Fletcher, *A new approach to variable metric algorithms*, Comput. J. **13** (1970), no. 3, 317–322.
- [21] Anders Forsgren, Philip E. Gill, and Margaret H. Wright, *Interior methods for nonlinear optimization*, SIAM Review **44** (2002), no. 4, 525–597.
- [22] Robert Fourer, David M. Gay, and Brian W. Kernighan, *AMPL – A Modeling Language for Mathematical Programming*, 2nd ed., Brooks/Cole Publishing Company / Cengage Learning, 2002.
- [23] K. R. Frisch, *The logarithmic potential method of convex programming*, Memorandum of may 13, University Institute of Economic, Oslo, May 1955.
- [24] Jean-loup Gailly and Mark Adler, *A Massively Spiffy Yet Delicately Unobtrusive Compression Library*, <http://www.zlib.net/>.
- [25] *The GNU Compiler Collection*, <http://gcc.gnu.org/>.
- [26] Carl Geiger and Christian Kanzow, *Numerische Verfahren zur Lösung unrestringierter Optimierungsaufgaben*, Springer, 1999.
- [27] ———, *Theorie und Numerik restringierter Optimierungsaufgaben*, Springer, 2005.
- [28] J. A. George, *Computer implementation of the finite element method*, Report, Stanford Dept. of Computer Science, 1971.
- [29] Philip E. Gill, Walter Murray, and Margaret H. Wright, *Practical optimization*, Academic Press, 1981.
- [30] D. Goldfarb, *A family of variable-metric methods derived by variational means*, Math. Comp **24** (1970), 23–26.
- [31] Shih-Ping Han, *Superlinearly Convergent Variable Metric Algorithms for General Nonlinear Programming Problems*, Math. Programming **11** (1976/77), no. 3, 263–282.

- [32] Willi Hock and Klaus Schittkowski, *Test Examples for Nonlinear Programming Codes*, Lecture Notes in Economics and Mathematical Systems, vol. 187, Springer-Verlag, 1981.
- [33] Bill Hoffman, Ken Martin, Brad King, Dave Cole, Alexander Neundorf, Clinton Stimpson, et al., *CMake*, <http://www.cmake.org/>.
- [34] *HSL(2011). A collection of Fortran codes for large scale scientific computation*, <http://www.hsl.rl.ac.uk/>.
- [35] Adaptive Computing Inc., *MOAB HPC Suite*, <http://www.adaptivecomputing.com/products/hpc-products/moab-hpc-basic-edition>.
- [36] ———, *TORQUE Resource Manager*, <http://www.adaptivecomputing.com/products/open-source/torque>.
- [37] Santa Cruz Operation Inc., *System V Application Binary Interface*, <http://refspecs.freestdards.org/elf/abi386-4.pdf>.
- [38] ISO/IEC 9899:1999, *Programming languages - C*, ISO International Organization for Standardization.
- [39] *Javadoc*, <http://www.oracle.com/technetwork/java/javase/documentation/index-137483.html>.
- [40] ISO/IEC JTC1/SC22, *Information technology - Programming languages - Fortran*, draft N1601 of the Fortran 2003 standard.
- [41] Patrik Kalmbach, *Effiziente Ableitungsbestimmung bei hochdimensionaler nichtlinearer Optimierung*, Dissertation, Universität Bremen, Bremen, 2011.
- [42] Narendra Karmarkar, *A new polynomial time algorithm for linear programming*, *Combinatorica* **4** (1984), no. 4, 373–395.
- [43] George Karypis and Vipin Kumar, *A fast and highly quality multilevel scheme for partitioning irregular graphs*, *SIAM Journal on Scientific Computing* **20** (1999), no. 1, 359–392.
- [44] Kohsuke Kawaguchi, R. Tyler Croy, and Andrew Bayer, *Jenkins CI*, <http://jenkins-ci.org/>.
- [45] Kemper, Anna Elise, *Filtermethoden zur Bewertung der Suchrichtung in NLP-Verfahren*, Diploma thesis, Zentrum für Technomathematik, Universität Bremen, 2010.
- [46] Steven Knight, Gary Oberbrunner, Greg Noel, Sergey Popov, et al., *SCons*, <http://www.scons.org/>.
- [47] Bruce Korb, *AutoGen*, <http://www.gnu.org/software/autogen/>.
- [48] X. Sherry Li, James Demmel, John Gilbert, Laura Grigori, and Meiyue Shao, *SuperLU*, <http://crd.lbl.gov/~xiaoye/SuperLU/>.
- [49] MathWorksTM, *MATLAB*, <http://www.mathworks.com/products/matlab/>.

- [50] *Serial Graph Partitioning and Fill-reducing Matrix Ordering*, <http://glaros.dtc.umn.edu/gkhome/views/metis>.
- [51] P.A. Miller, *Recursive make considered harmful*, AUUGN Journal of AUUG Inc. **19** (1998), no. 1, 14–25.
- [52] *MinGW*, <http://www.mingw.org/>.
- [53] Nicholas Nethercote and Julian Seward, *valgrind*, <http://valgrind.org/>.
- [54] ———, *How to shadow every byte of memory used by a program*, Proceedings of the Third International ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE 2007) (San Diego, California, USA), June 2007, pp. 65–74.
- [55] ———, *Valgrind: A framework for heavyweight dynamic binary instrumentation*, Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007) (San Diego, California, USA), June 2007, pp. 89–100.
- [56] Dominique Orban, *a Constrained and Unconstrained Testing Environment, revisited*, <http://http://magi-trac-svn.mathappl.polymtl.ca/Trac/cuter>.
- [57] C. Michael Pilato, Ben Collins-Sussman, and Brian W. Fitzpatrick, *Version Control with Subversion*, 2nd ed., O’Reilly Media.
- [58] R Core Team, *R: A language and environment for statistical computing*, R Foundation for Statistical Computing, Vienna, Austria, 2012, ISBN 3-900051-07-0.
- [59] Craig Rasmussen, Matthew Sottile, and Christopher Rickett, *Chasm Language Interoperability Tools*, <http://chasm-interop.sourceforge.net/>.
- [60] Martin Reddy, *API Design for C++*, 1 ed., Morgan Kaufmann, February 2011.
- [61] Robert Ross, Thomas Ludwig, Pete Wyckoff, et al., *Parallel Virtual File System*, <http://www.pvfs.org/>.
- [62] Klaus Schittkowski, *NLPQLP*, <http://www.ai7.uni-bayreuth.de/nlpqlp.htm>.
- [63] ———, *More Test Examples for Nonlinear Programming Codes*, Lecture Notes in Economics and Mathematical Systems, vol. 282, Springer-Verlag, 1987.
- [64] D. F. Shanno, *Conditioning of quasi-Newton methods for function minimization*, Math. Comp **26** (1970), 647–656.
- [65] Edgewall Software, *Bitten – A continuous integration plugin for Trac*, <http://bitten.edgewall.org/>.
- [66] ———, *trac – Integrated SCM & Project Management*, <http://trac.edgewall.org/>.
- [67] Michael Sperber, R. Kent Dybvig, Matthew Flatt, Anton van Straaten, Robby Findler, and Jacob Matthews, *Revised [6] Report on the Algorithmic Language Scheme*, Cambridge University Press, April 2010.
- [68] Richard M. Stallman, Roland McGrath, and Paul Smith, *GNU Make*, <http://www.gnu.org/software/make/>.

- [69] Adrian D. Thurston, *Ragel State Machine Compiler*, <http://www.complang.org/ragel/>.
- [70] Adrian D. Thurston, *Parsing computer languages with an automaton compiled from a single regular expression*, Lecture Notes in Computer Science (Taipei, Taiwan), vol. 4094, 2006, 11th International Conference on Implementation and Application of Automata (CIAA 2006), pp. 285–286.
- [71] Dimitri van Heesch, *Doxygen*, <http://www.stack.nl/~dimitri/doxygen/>.
- [72] Gary V. Vaughan, Bob Friesenhahn, Peter O’Gorman, and Ralf Wildenhues, *GNU libtool*, <http://www.gnu.org/software/libtool/>.
- [73] Daniel Veillard, *Libxml2*, <http://www.xmlsoft.org/>.
- [74] Andreas Wächter, *Ipopt*, <http://www.coin-or.org/projects/Ipopt.xml>.
- [75] Andreas Wächter and Lorenz T. Biegler, *On the Implementation of a Primal-Dual Interior Point Filter Line Search Algorithm for Large-Scale Nonlinear Programming*, Mathematical Programming **106** (2006), no. 1, 25–57.
- [76] Robert B. Wilson, *A Simplicial Algorithm for Concave Programming*, Ph.D. thesis, Harvard University, 1963.
- [77] Mario Zechner, *Lightweight XML parser*, <http://www.badlogicgames.com/wordpress/?p=1712>.

